TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl VI: Echtzeitsysteme und Robotik

PARAMETER EXPLORING POLICY GRADIENTS AND THEIR IMPLICATIONS

Frank Sehnke

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ-Prof. Dr. D. Cremers

Prüfer der Dissertation:

1. Univ.-Prof. Dr. P. van der Smagt
2. Univ.-Prof. Dr. H. J. Schmidhuber, IDSIA/Schweiz

Die Dissertation wurde am 03. 05. 2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 09. 10. 2012 angenommen.

Dedicated to my loving wife Susanne Sehnke.

Science is like sex:
sometimes something useful comes out, but that is not the reason we are doing it.

— Richard P. Feynman

ABSTRACT

Reinforcement Learning is the most commonly used class of learning algorithms which lets robots or other systems autonomously learn their behaviour. Learning is enabled solely through interaction with the environment. Today's learning systems are often confronted with high-dimensional and continuous problems. To solve those, so-called Policy Gradient methods are used more and more often.

The PGPE algorithm developed in this thesis, a new type of Policy Gradient algorithm, allows model-free learning in complex, continuous, partially observable and high-dimensional environments. We show that tasks like grasping of glasses and plates with a human-like arm can be learned with this method without prior knowledge, solely with pure model-free reinforcement learning in a simulation environment. Also, the balancing of a humanoid robot perturbed by external forces, as well as dynamic walking behaviour of a mass-spring system could be learned. In all experiments, PGPE learned the given tasks more efficiently than well-established methods. In addition, the use of PGPE is not restricted to robotics. Among several investigated methods, it was the most successful in cracking non-differentiable physical cryptography systems. PGPE is suitable for training multidimensional recurrent neural networks to play Go, or for fine-tuning deep neural nets for computer vision.

In the scope of this thesis, the principles used, the advantages and disadvantages as well as the differences with regard to well-established methods are derived and analysed in detail.

ZUSAMMENFASSUNG

Reinforcement Learning (Bestärkendes Lernen) ist die am häufigsten verwendete Klasse von Lernalgorithmen, um Robotern oder anderen Systemen das selbständige Erlernen ihres Verhalten zu ermöglichen. Lernen geschieht hierbei allein durch Interaktion des Systems mit seiner Umwelt. Heutige lernende Systeme haben es oft mit hochdimensionalen und kontinuierlichen Problemen zu tun. Hierfür kommen vermehrt die so genannten Policy Gradient Methoden zum Einsatz.

Der in dieser Arbeit entwickelte PGPE-Algorithms, ein neuer Typ von Policy Gradients, ermöglicht modellfreies Lernen in komplexen, kontinuierlichen, nur teilweise beobachtbaren und hochdimensionalen Umgebungen. Wir zeigen, dass hiermit ohne Vorwissen, durch reines modellfreies bestärkendes Lernen in einer Simulationsumgebung, Aufgaben wie das Greifen von Gläsern und Tellern mit einem dem menschlichen Arm nachempfundenen Roboter erlernt werden. Auch das Balancieren eines humanoiden Roboters der von externen Kräften gestört wird, sowie das dynamische Laufen eines Masse-Feder Systems wurden erlernt. In allen Experimenten lernte PGPE die Aufgaben effizienter als etablierte Methoden. Der Einsatz von PGPE beschränkt sich dabei nicht auf die Robotik. Sie ist die erfolgreichste Methode unter den

untersuchten um nicht differenzierbare physikalische Kryptographie Systeme zu brechen. Sie ist geeignet um multidimensionale rekurrente neuronale Netze zu trainieren, Go zu spielen oder um tiefe neuronale Netze für die Bildverarbeitung nachzutrainieren.

Die Prinzipien, welche hierbei zur Anwendung kamen, die Vor- und Nachteile sowie die Unterschiede gegenüber den etablierten Methoden werden im Rahmen der Arbeit im Detail hergeleitet und analysiert.

PUBLICATIONS

During this thesis the following publications were published:

- Frank Sehnke, Alex Graves, Christian Osendorfer, and Jürgen Schmidhuber. **Multimodal Parameter-exploring Policy Gradients.** The Ninth International Conference on Machine Learning and Applications, ICMLA 2010.
  Derivation of the MultiPGPE method incorporated in chapter 5
- Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. **Parameter-exploring policy gradients.** Neural Networks, 23(2), March 2010. Journal Publication
  Incorporated as the main publication of this thesis in chapters 2, 3, 5, 6, 8
- Frank Sehnke, Christian Osendorfer, Jan Sölter, Jürgen Schmidhuber, and Ulrich Rührmair. **Policy gradients for cryptanalysis.** In W. Duch K. Diamantaras and L. Iliadis, editors, Proceedings of the International Conference on Artificial Neural Networks, ICANN 2010. Springer-Verlag Berlin Heidelberg, 2010.
  Ideas, figures and tables incorporated in chapter 9
- Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srini Devadas, and Jürgen Schmidhuber. **Modeling attacks on physical unclonable functions.** In Proceedings of the 17th ACM Conference on Computer and Communications Security, ACM CCS 2010
- Mandy Grüttner, Frank Sehnke, Tom Schaul, and Jürgen Schmidhuber. **Multi-dimensional deep memory go-player for parameter exploring policy gradients.** In W. Duch K. Diamantaras and L. Iliadis, editors, Proceedings of the International Conference on Artificial Neural Networks, ICANN 2010. Springer-Verlag Berlin Heidelberg, 2010.
  Incorporated in chapter 11
- Thomas Rückstieß, Frank Sehnke, Tom Schaul, Daan Wierstra, Sun Yi, and Jürgen Schmidhuber. **Exploring parameter space in reinforcement learning.** Paladyn Journal of Behavioral Robotics, 1(1):14-24, 2010. Journal Publication
- Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. **PyBrain.** Journal of Machine Learning Research, 2010. Journal Publication
- Thomas Rückstieß, Martin Felder, Frank Sehnke, and Jürgen Schmidhuber. **Robot learning with state-dependent exploration.** In 1st International Workshop on Cognition for Technical Systems, 2008.
- Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. **Policy gradients with parameter-based exploration for control.** In J. Koutnik V. Kurkova, R. Neruda, editor, Proceedings of the International Conference on Artificial Neural Networks, ICANN 2008, Part I, LNCS

5163, pages 387-396. Springer-Verlag Berlin Heidelberg, 2008.
Incorporated in chapters 5

- Frank Sehnke, Thomas Rückstieß, Martin Felder, and Jürgen Schmidhuber. **Parametric policy gradients for robotics.** In 1st International Workshop on Cognition for Technical Systems, 2008.

ACKNOWLEDGMENTS

There are several people I would like to thank because without them this thesis would not exist or it would not have turned out the same. First of all, I want to thank Jürgen Schmidhuber, my supervisor and "Doktorvater" for the opportunity to write my PhD studies with his guiding advice and assistance. His rather strong opinions influenced me a lot and his input allowed me to become exposed to many inspiring topics and people. I also want to thank Patrick van der Smagt for his very valuable advice and for the fact that he was willing to be my supervisor. His suggestions for improving my thesis were worth his weight in gold. I also want to thank Alexander Graves who had endless patience in correcting my less-than-perfect English and was a source of priceless advice and inspiration. Special thanks also goes to Christian Osendorfer who was a constant sparring partner and I also want to thank him especially for his well-founded mathematical knowledge which was extremely helpful. I also want to thank Thomas Rückstieß and Martin Felder for their stimulating ideas and fruitful discussions.

Thanks go also to my students. Especially I want to thank Ahmed Mahmoud and Mandy Grüttner, whose help was very inspiring and much appreciated. I also want to thank two students of our group, Ralf Stauder and Jan Sölter for the fruitful discussions.

I thank Jan Peters from MPI Tübingen, Daan Wierstra and Tom Schaul, both at that time from IDSIA Lugano, for introducing me to policy gradient methods and for teaching me everything necessary to get started. I want to thank all three of them for the very kind cooperation, for the helpful discussions and for their constant good advice. I want to thank Jan Peters for acting almost like an additional supervisor.

Furthermore, I want to thank Alois Knoll for giving me the opportunity to conduct my PhD thesis at his chair and to use the chairs resources. I also want to thank him for his way of helping me with organizational issues, and for keeping all the *red tape* to a minimum. Special thanks go to Gerhard Schrott, who was a great help with his kind, calming character and his endless wisdom. Unconventional thanks go to our "secretoids" Amy Bücherl, Gisela Hibsch and Monika Knürr. I cannot imagine how I would have ever conducted my thesis without them. They supported me both in an administrative capacity and emotionally.

I also want to thank Michael Klimke, head of IGSSE, very much for his ongoing support and sympathy. I thank Srini Devadas from MIT for his insights and his help and the very pleasant cooperation.

I want to thank all my colleagues and friends for their endless patience and encouragement which was required to write a PhD thesis.

Finally, I want to thank my parents and my wife who, with their continual support, gave me the opportunity to take the necessary steps that resulted in this thesis.

CONTENTS

ACRONYMS

Arb-PUF	-	Standard Arbiter PUF
CMA-ES	-	Covariance Matrix Adaptation - Evolution Strategies
CoSyNE	-	Cooperative Synapse Neuro-Evolution
CRP	-	Challenge Response Pair
ES	-	Evolution Strategies
ESP	-	Enforced Sub-Population
ET	-	Eligibility Trace
FD	-	Finite Difference
FF-loop	-	Feed-Forward loop
FF-PUF	-	Feed-Forward Arbiter PUF
GA	-	Genetic Algorithm
GPOMDP	-	Gradient of a Partially Observable Markov Decision Process
GUI	-	Graphical User Interface
IM	-	Importance Mixing
LR	-	Logistic Regression
LSTM	-	Long Short Term Memory
MDLSTM	-	Multi Dimensional Long Short Term Memory
MDP	-	Markov Decision Process
MDRNN	-	Multi Dimensional Recurrent Neural Network
ML	-	Machine Learning
MLP	-	Multi Layer Perceptron
NAC	-	Natural Actor Critic
NES	-	Natural Evolution Strategies
NN	-	Neural Network
NumPy	-	Numeric Python (Library)
ODE	-	Open Dynamics Engine
PEPG	-	Parameter Exploring Policy Gradients
PG	-	Policy Gradients
PGPE	-	Policy Gradients with Parameter-based Exploration
POMDP	-	Partially Observable Markov Decision Processes
PUF	-	Physical Unclonable Function
PyBrain	-	Python-Based Reinforcement learning, Artificial Intelligence and Neural network library
RBM	-	Restricted Boltzmann Machines
REINFORCE	-	REward Increment = Nonnegative Factor times Offset Reinforcement times Characteristic Eligibility
RL	-	Reinforcement Learning
RNN	-	Recurrent Neural Network
Rprop	-	Resilient backPROPagation

SANE	-	Symbiotic Adaptive Neuro-Evolution
SDE	-	State Dependent Exploration
SPSA	-	Simultaneous Perturbation Stochastic gradient Approximation
SVM	-	Support Vector Machine
TCP/IP	-	Transmission Control Protocol / Internet Protocol
UDP	-	User Datagram Protocol
XOR-PUF	-	XOR Arbiter PUF

Part I

PROBLEM DEFINITION AND STATE OF THE ART

1 INTRODUCTION

1.1 MOTIVATION

Properties of practical robots

A practical robot should be capable of both adapting to new situations, and of being adjusted by the users to suit their needs. To make robots accessible in this manner (also for non-specialists) it must be possible to guide their behaviour without having to program a computer. Ideally, it should be possible for a user to teach a robot by example, just as they would teach a fellow human.

Currently, the situation is that robots *learn* new tasks mainly by a specialist programming the required behaviour by hand or by using a technique called *Teach-In*. Teach-In means that a human guides the robot (usually a robot arm) in executing a certain task. The robot records the movement and reproduces it from there on.

If robots should be introduced into households and the like, this has to change drastically. Non-specialists must be able to teach robots the tasks they want them to do.

Problems in creating practical robots

To achieve this, several problems have to be solved. One of them is how to define new goals for the robot using natural human communication. Another one is how to show a robot naturally how a human would do something, i.e. in such a way that the robot can learn by imitation. Thirdly, how to learn new behaviours assuming the task is known and well defined. And finally, how to build an adaptive model of the environment, which helps to predict the consequences of certain actions in a changing environment.

The main problem this thesis focuses on is the third problem domain—the actual learning of new tasks. As we will see later in this thesis, realistic learning via trial and error—known as model-free Reinforcement Learning (RL)—typically needs about 10,000 to 100,000 trials to learn even simple tasks like grasping an object from a table. This seems like a lot at first glance and the main issue is to reduce the number of trials needed by making the learning methods more effective. If an effective learning method can be found the number of trials needed can be further reduced by employing an internal world model (much like humans do in the grown-up stage).

The importance of effective model-free learning

Computation time gets cheaper every year. Nonetheless one of the major tasks of Machine Learning (ML) is to make as efficient use as possible of the computational resources at hand. This is especially true for systems that have to learn without models, because a good exact model is truly hard to come by. But it is also true for systems that learn with a model, because it reduces computation time that can instead be used to exploit better models or explore larger search spaces.

Aim of the thesis

This thesis will investigate all of the four aforementioned problems to some extent, but the main focus will be on how to learn as effectively as possible in a model-free framework for sophisticated high-dimensional robot behavioural tasks. As we will see, the techniques developed to achieve this goal are also effective in other problem domains.

This thesis will demonstrate how one can train a robot arm to grasp plates and glasses from arbitrary positions in the workspace—how to train a humanoid robot to stand robustly while perturbed by external forces and how to train mass-spring systems shaped like a cube to learn a variety of tasks, such as running and finding food, to name only a few. These tasks are within the computational scope of today's computer technology and this thesis develops a method that can be used *out of the box* to learn these kinds of behaviours by pure, model-free reinforcement learning and it explains in detail how to do this.

1.1.1 *Reinforcement Learning for Robotics*

In RL the system is ignorant how the optimal behaviour looks like

RL generally tries to optimise an agent's behaviour in its environment. But in contrast to supervised learning, the optimal behaviour is unknown. The system is just told how good the executed actions were. This is usually done by giving the system a scalar value, often in every time step, that is called *reward*. This kind of learning setting is ideal for robotic behaviour learning, because a behaviour can be learned just by means of trial and error without prior knowledge of the optimal behaviour that is in most cases unknown. Especially if one follows the above goal to teach robots new behaviours in a natural human way by demonstrating what one wants the robot to do, RL is the kind of learning class that is certainly most appropriate.

Learning in RL comprises a cycle of interactions with the environment. Within every time step the system observes its environment via its sensors. The situation that the system and the environment are in, is called *state*. The part of this state which is visible for the system is the so called *observation*.

The system learns by getting reward for good behaviour and punishment for less optimal behaviour

The system and the environment change into a new state by the execution of an action by the system. Every state-action pair is coupled with a reward based on how appropriate the action was in the corresponding state.

The goal of RL is to maximise the expected future reward. Classical RL operates in discrete state and action spaces. For that reason, tables are usually used for the state-action transitions in classical RL. The entries of the table resemble the so called Q-value, that is an estimate of the expected future reward for the particular state-action pair.

Policy

After some experience with the environment the Q-values are good estimates of the future reward. The strategy with which the actions are chosen in the current states is called *policy*. The best policy for the system to follow (if the reactions of the environment are sufficiently well known), is to choose the action in every state that promises the highest future reward.[1]

1 This is called a greedy policy, because it allows only actions that promise the most reward *income*.

1.1.2 *Policy Gradients*

Classical RL is difficult to apply to problems with unlimited continuous state and action spaces

There exists a wide variety of methods in the field of RL. The classical methods (see Section 3.1.1) act by estimating what action *promises* the most reward in the long run. This estimation of the so called value function is hard to do if we do not have discrete distinguishable actions to take, but instead have the infinite possibilities of choosing continuous actions, like for instance which movement direction and speed is to be taken for a robot arm to approach the target object. Reinforcement learning algorithms based on value function approximation are very successful if actions and states are discrete, and therefore value approximation can be performed with discrete lookup tables. If the same techniques are applied to continuous action state spaces (and hence continuous function approximation has to be used), most of the algorithms fail to generalise. Additionally, only under a few constraint situations convergence guarantees could be obtained theoretically [Sutton and Barto, 1998]. The main problem lies in the use of the ϵ-greedy (see Section 3.2.1) policy updates of the usual classical RL method. Without going too much into detail at this point, the ϵ-greedy strategy does not ensure an improvement of the behaviour when applied to approximate value functions [Bertsekas and Tsitsiklis, 1996]. The approximation errors can result in oscillations or divergence of the algorithms. Several simple toy problems were investigated where this ill behaviour can be found [Baird and Moore, 1999; Bertsekas and Tsitsiklis, 1996].

PG methods are well suited for continuous state-action spaces

We therefore concentrate on Policy Gradients (PG, see Section 3.1.3). A policy in this respect is the strategy by which the actions are taken. PG methods do not take actions due to the estimation of future reward—they use the reward they gather directly to decide which policy to use in the future without estimating any values of future reward. PG methods are in fact among the few feasible optimisation strategies for complex, high-dimensional reinforcement learning problems with continuous states and actions [Benbrahim and Franklin, 1997; Peters and Schaal, 2006; Schraudolph et al., 2006; Peters et al., 2005].

PG methods have strong convergence guarantees in continuous state-action spaces. A theoretically solid framework for policy gradient estimation from sampled data has already been found [Konda and Tsitsiklis, 2000; Sutton et al., 2000].

All these reasons led to the decision for the use of PG methods rather than classical RL approaches.

1.1.3 *Exploration in Parameter Space*

PG methods have a rather slow convergence behaviour due to high variance in the gradient estimate

A significant problem of policy gradient algorithms such as REINFORCE [Williams, 1992] is, however, that the high variance in their gradient estimates leads to slow convergence. To understand why the PG gradient has a high variance (i.e. why it is noisy) one has to know how exploration in PG methods work. Exploration in PG methods is done by (roughly speaking) adding noise to the taken action—so the system always acts a bit differently than it would normally without this explorative noise. The reward gathered will differ depending on environmental factors that the system cannot control and because the system acts always a bit differently. This coupled *reward-to-taken-action*

information is used to derive the gradient. To get to a new behaviour that executes the task in a notably different way, these random changes to the actions have to be done frequently over time. In this exploration strategy lies one of the main sources of the high noise in the gradient estimate. It is hard to define which parts of the randomly changed policy are responsible for the changed reward. As one example for illustrating this problem, let us assume the system *re-lives* the exact same situation twice. Each time a different random change to the action is chosen. Afterwards the system receives a reward. In the same situation the system did two different things—what action is now responsible for the changed reward? This is just one illustration of the overall problem of the special PG credit assignment problem.

Various approaches have been proposed to reduce this variance [Baxter and Bartlett, 2000; Aberdeen, 2003; Peters and Schaal, 2006; Sutton et al., 2000]. However, none of these methods address the underlying cause of the high variance, which is that repeatedly sampling from a probabilistic policy has the effect of injecting noise into the gradient estimate at every time step. Furthermore, the variance increases linearly with the length of the history [Munos and Littman, 2006], that is the time steps one trial takes to solve the task, since each state depends on the entire sequence of previous samples.

Exploring in parameter space reduces the high variance in the gradient estimate drastically

As an alternative, we propose to define the policy by a distribution over the parameters of a controller. That means that we explore by changing the parameters directly and test how this new set of parameters changes the behaviour of the system and how the reward received is affected by that. The parameters are sampled from this distribution only once at the start of each sequence, and after that, the controller doesn't change over one episode or history—it is deterministic during the trial. Since the reward for each sequence depends on a single sample only, the gradient estimates are significantly less noisy, even in stochastic environments.

Among the advantages of exploring in parameter space by these means is that we do not have the same credit assignment problem. In contrast, a standard policy gradient method must first determine the reward gradient with respect to the policy—it must judge how the changed actions resulted in the changed reward. Then the system has to differentiate the parameters with respect to that reward gradient—that means the system has to figure out how the parameters have to be changed in order to make the good behaviour which was just executed more likely or the bad behaviour less likely in the future. This results in two drawbacks. Firstly, it assumes that the controller is always differentiable with respect to its parameters, otherwise the parameters cannot be continuously changed due to the changed actions. Secondly, it makes optimisation more difficult, because very different parameter settings can determine very similar policies, and vice-versa. So, it can be hard to decide what parameter set to choose in order to generate the observed behaviour because there are several different parameter sets producing this behaviour.

Exploration in parameter space allows the use of non-differentiable controllers

Based on these reasons, we see great potential in using Parameter Exploring Policy Gradients (PEPG) and the thesis will focus on this class of algorithms.

1.1.4 *Our Approach*

Our approach is called Policy Gradients with Parameter-based Exploration (PGPE). PGPE and its variants are derived in detail in Chapter 5. As mentioned above in PGPE we will define the policy by a distribution over the parameters of a controller. The parameters are sampled from this distribution at the start of each sequence, and afterwards the controller is deterministic. We will also introduce refinements for sampling and reward normalisation. We will present variants with multi-modal parameter distributions and refinements that make PGPE usable in infinite horizon settings. We will show in the course of this thesis that PEPG methods and especially PGPE and its variants have superior properties in dealing with complex high-dimensional robotic RL tasks and that PGPE can also be applied to RL tasks that are as distinct from robotics as crypt-analysing Physical Cryptography Systems or learning to play Go with Multidimensional Recurrent Neural Networks.

1.2 THESIS CONTRIBUTION

This section outlines the rest of this document, explaining the contributions that were made in this thesis.

- Part I, Chapter 2: Defines the problem domains we try to cope with PEPG methods and gives some implications.
- Part I, Chapter 3: Depicts the state of the art in robotic RL and RL as a whole.
- Part I, Chapter 4: Concludes and gives an overview of the introduction, problem definition and state of the art part.
- Part II, Chapter 5: Derives PGPE and its variants from the general framework of episodic reinforcement learning in a Markovian environment.
- Part II, Chapter 6: Highlights the properties of PGPE and shows the relation to other ML fields.
- Part II, Chapter 7: Concludes and summarises the methods and new contributions part.
- Part III, Chapter 8: Explains the robotic benchmarks we used and gives results and comparisons to competing methods on this benchmarks.
- Part III, Chapter 9: Explains Physical Cryptography, its currently existing variants and gives results and comparisons to competing methods on this problem domain.
- Part III, Chapter 10: Explains how Deep Nets with Restricted Boltzmann Machines can be used for RL and how PGPE can be used to fine-tune the Deep Net weights.
- Part III, Chapter 11: Explains the board game Go and shows how an artificial Go player can be learned with PGPE and how PGPE competes to other methods on this problem domain.
- Part III, Chapter 12: Concludes and summarises the experimental part.

- Part IV, Chapter 13: Gives a conclusion of the experience gathered in this thesis with PEPG methods an summarises the properties of PEPGs in comparison to the RL field.
- Part IV, Chapter 14: Gives ideas of interesting fields of future work with PEPGs.
- Part V, Appendix A: Gives the implementation details of PGPE and the used robotic benchmarks in the open source ML Library PyBrain.
- Part V, Appendix B: Gives condensed Python code for the standard (SyS) PGPE implementation, ready to use as stand alone RL learner.

1.3 NOTATION

We use the standard notation throughout the thesis with some exceptions:

- We write vectors in bold like $\boldsymbol{\sigma}$ instead of using upper case letters or using vector arrows to highlight that we talk about a vector and not a single value.
- We use bold upper case letters for matrices like $\boldsymbol{\Sigma}$.
- We use for the index of used samples a superscript instead of a subscript for readability. r_t^n is therefore the reward r gathered in sample n at time step t.

2 PROBLEM DEFINITION

In all robotic problems we assume a MDP or POMDP setting

In all robotic problem domains in this thesis we assume that the environment the agent is working in can be modelled as a Markov Decision Process (MDP). Most environments provide thereby only partial information of their state to the agent, therefore called Partially Observable Markov Decision Processes (POMDP). This means that not all environmental variables are available but a selection that is accessible i.e. via realistic sensors. This makes recurrence or memory in the controllers necessary to some degree to access time dependent hidden variables or the like.

2.1 MARKOV DECISION PROCESSES

Def.: MDP

We follow the principal definition of MDP and POMDP from *Planning and acting in partially observable stochastic domains* [Kaelbling et al., 1998]. Our setting is that of an agent taking actions in an environment following a policy. Denote the state of the environment at time t as s_t and the action at time t as a_t. Because we are interested in continuous state and action spaces (usually required for robotic control tasks), we represent both a_t and s_t with real-valued vectors. Each state–action pair gives the agent a scalar reward $r_t(a_t, s_t)$. The agent's actions stochastically depend on the current state and some real-valued parameter vector Θ: $a_t \sim p(a_t|s_t, \Theta)$. We assume that the environment is Markovian, i.e. that the probability distribution over the possible next states is conditionally independent of all previous state-action pairs given the current one: $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$. We refer in most cases to a sequence h of state-action pairs of length T produced by an agent as a *history*: $h = [s_{1:T}, a_{1:T}]$. In this MDP framework the agent can fully observe the whole environment.

2.2 PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES

Def.: POMDP

In POMDP the agent perceives the environment by means of sensors or the like, that provides only a partial view of the complete state of the environment. In a POMDP the agent needs to estimate so called hidden states by means of calculation from observable variables or by means of memory (i.e. the speed of a joint if only a position sensor exists).

The agent can only use observations instead of the true state in a POMDP

One still denotes the state of the environment at time t as s_t and the action at time t as a_t. In contrast to 2.1 we cannot observe state s_t directly, instead we have to work with the observation o_t. The history changes now to: $h = [o_{1:T}, a_{1:T}]$. We also define the history up to the time step t as $h(t) = [o_{1:t}, a_{1:t}]$ with $t < T$. Because the reward is still dependent on the state-action pair the agent must find a reliable estimate for $r_t(a_t, s_t)$. Now the agent's actions stochastically depend on the history $h(t)$ and parameter vector Θ: $a_t \sim p(a_t|h(t), \Theta)$ if some form of recurrence or memory can be assumed or on the current observation

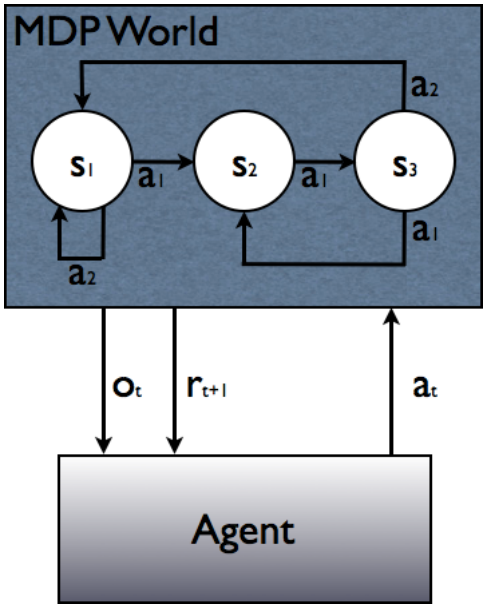


Figure 2.1: The POMDP setting. The world is assumed to be Markovian. The agent is only provided partly with the information of the world (usually through the available sensors), the so called observation. The agent must decide on a good action due to its observation.

o_t and parameter vector Θ: $a_t \sim p(a_t|o_t, \Theta)$ if not. We still assume that the environment is Markovian, i.e. that the probability distribution over the possible next states is conditionally independent of all previous state-action pairs given the current one.

2.3 LONG TERM REWARD AND EPISODIC TASKS

Episodic reward

Rewards come often after some time delay or after a series of good actions. In this thesis this is not a major point, because we use in nearly all cases a strictly episodic approach. In an episodic task definition the agent receives reward only once after a complete history or episode. This has the advantage that the complete behaviour is evaluated. In a non-episodic task definition there rewards are received frequently an additional credit assignment problem arises—the temporal credit assignment problem. The temporal credit assignment problem describes the difficulty to assign the reward to the distinct action that lead to the reward. Reward, and the action(s) responsible for the reward are often not timely aligned. In an episodic task definition one evaluates the overall policy directly while in an non-episodic or infinite horizon task one evaluates certain actions. There are a lot of cases there the higher amount of information of the infinite horizon reward leads to faster learning, while there are also a lot of examples there the temporal delayed structure of the reward is misleading and can result in suboptimal convergence. With episodic task definitions the learning is in most cases more robust. For InfHorPGPE (see Chapter 5.3) we cope with the problem of time delayed reward in a special way.

Temporal Credit Assignment Problem

We used a wide spectra of robotic benchmarks to elaborate our new method class, reaching from simple optimisation functions like the Rastrigin function over simple standard benchmarks like the Pole Balancing and Ship Steering up to sophisticated robotic simulations, like grasping objects with a 7-DoF robotic arm or standing robust against perturbations with an 11-DoF humanoid robot.

See chapter 8 for the different kind of robotic scenarios used and the details on the involved controllers.

3 STATE OF THE ART

3.1 REINFORCEMENT LEARNING

RL - high need of evaluations

As mentioned in section 1.1.1 Reinforcement Learning (RL) is the method of choice for learning robotic behaviours. Unfortunately, as no direct information about the optimal behaviour like in the case of supervised learning is available, learning requires a large number of trials.

Modelling and exploration are therefore critical components of RL, affecting both the number of required trials and the quality of the found solution. The kind of exploration heavily depends on the kind of learning algorithm that is used and also on the kind of modelling involved. Standard RL methods like SARSA [Sutton and Barto, 1998] or Q-Learning [Watkins and Dayan, 1992] use discrete sets of states and a discrete set of actions to respond to the states the agent is in. This makes it possible to use a value table assigning every state-action pair an expected reward in the long run, a so called value. This is already a model because it predicts the outcome in terms of a reward if a certain action is taken in a certain state, though it does not say something about the details how this reward comes to be.

Classical RL is discrete and uses value tables

Exploration in this case can be done either completely random—a strategy that is surprisingly common—by performing a random action with a predefined probability. Or exploration can be done by choosing an action proportional likely to its reward value. Also one can think of exploring actions there the reward outcome is rather uncertain for the given state to achieve a better model.

Exploration in classical RL by random action selection

This kind of RL methods are hard to extend into domains with continuous state and action spaces, though fruitful efforts have been made by clever clustering the state space and/or using function approximators [Sutton and Barto, 1998].

Exploration of classical RL is hard to extend to continuous domains

3.1.1 *Classical*

SARSA In classical value based RL we usually work with discrete sets of states and actions. For that reason, every action a_i taken in state s_j has a, to the agent, unknown expected reward. This expected reward is called a Q-value. Usually these Q-values are updated by experience. This update does not only take the reward for the action taken in the current state into account, but also the expected future reward by the discounted future reward received from the next state-action observation. SARSA updates the Q-values by a so called on-policy strategy. That means that the discounted rewards of the really taken actions are used to update the current Q-value.

Def.: Q-value, expected future reward for certain action in certain state

SARSA is an on-policy strategy

From this technique the name SARSA stems. For updating the Q-value one needs the current **S**tate, the **A**ction taken, the resulting **R**eward,

the next **S**tate the agent will find himself in and the next **A**ction the agent will choose in the new state. The update rule for SARSA is the following:

Def.: SARSA update rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad (3.1)$$

Q-LEARNING The main drawback of SARSA is the on-policy learning. During high rates of explorations the Q-values will be estimated too low, compared to the greedy policy. Therefore in Q-Learning the update of the Q-values is done off-policy [Watkins and Dayan, 1992], meaning that not actually chosen actions are used to estimate the Q-values, but the best action available. The update rule changes to:

Q-Learning is an off-policy strategy

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t)(1 - \alpha) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a)) \quad (3.2)$$

or written in the form of Eq. (3.1)

Def.: Q-Learning update rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (3.3)$$

Now the Q-value related to a state action pair resembles the expected reward if always the optimal actions are taken in the future.

3.1.2 *Evolution*

Fitness replaces the reward of RL in evolution

Evolutionary methods use even less information from the environment then RL methods usually do. They discard the temporally information given by the standard RL environment that gives a reward signal at every time-step during learning. In evolution an overall reward called fitness is calculated that describes the performance of the agent or solution over a complete trial or episode. Surprisingly this still leads in a lot of cases to faster and most of the time to a more robust convergence on good solutions than RL methods. Examples are the lead of CMA-ES [Hansen and Ostermeier, 2002] in the standard benchmark list of pole balancing. Sometimes—so it seems—the temporal distribution of reward in RL is misleading even over the consideration of eligibility traces.

Exploration in evolution — creating changed parameter sets

Exploration in evolutionary algorithms is done by using changed parameter sets that define the behaviour of the agent. The changes are called mutations (sometimes mixing of parameter sets is involved called crossover). The exploration is controlled by means of how frequent mutations occur and to which extend the mutations change the parameters. In comparison to PG this seems to be the main advantage of evolution.

The evolutionary method is not derived mathematically from the basic principles of RL

However drawbacks of evolutionary algorithms are that they do not derive their methodology from the basic principle of maximising the expected reward in a mathematical straight forward way and that they use a population where the information of the non surviving solutions is just discarded (model based evolution [Streichert, 2007] makes here a notable difference).

All evolutionary algorithms work according to the so-called evolutionary cycle: The differences between the different evolutionary algorithms is the coding, if or to which extend crossover and mutation is used and the different strategies for mutation and selection.

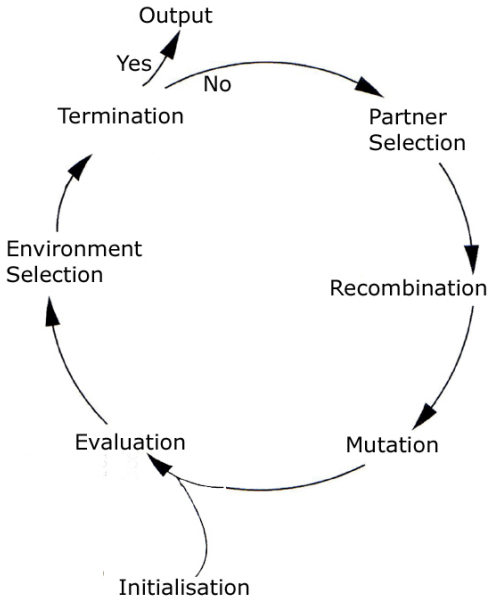


Figure 3.1: All evolutionary algorithms follow the evolutionary cycle. Based on Weicker [2002].

GENETIC ALGORITHM The most famous representative of evolutionary algorithms is the Genetic Algorithm (GA). Historically GAs use binary or at least discrete coding. Nowadays also continuous GAs are used, but for discrete genomes the principle holds that all alleles needed for the optimal solution are already present in the initial population and need only to be combined by crossover. The crossover operator as primary operator is originated from this philosophy while mutation is more or less employed to recover lost alleles.

GA works best on discrete genomes

For continuous genomes this assumption does not hold and therefore mutation must work differently and must have more impact.

GA works also on continuous genomes

By using crossover as primary operator GAs use medium sized populations and a medium selection pressure to have a good chance of combining good alleles together. This resembles the divide and conquer approach.

EVOLUTION STRATEGY In robotic behaviour learning, we deal with continuous controllers like NN. For this reason we chose Evolution Strategies (ES; [Schwefel, 1993]) as the representative of the evolutionary field. ES is well suited to learn NN controllers due to the kind of mutation, the independence of a crossover operator and most prominent the strategy parameters that allow for fast exploration.

ES is specialised on continuous genomes

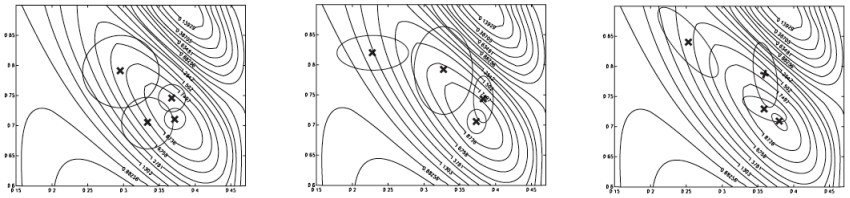


Figure 3.2: Different kinds of search strategy for the ES mutation operator and how the search pattern changes. (Source [Streichert and Ulmer, 2005])

Algorithm 1 The Genetic Algorithm with a population size of λ, m-Tournament selection and n problem parameters.

Initialise $\theta^{1\dots\lambda}$ randomly

while TRUE **do**
  " Evaluation
  evaluate $r^{1\dots\lambda} = r(h(\theta^{1\dots\lambda}))$

  " m-Turnament Selection
  for $j = 1$ to λ **do**
    for $j = 1$ to m **do**
      $i_m \sim [1 \dots \lambda]$
    end for
    $\theta'^{,j} = max(\theta^{i}|r^{i})$
  end for

  " 1-Point Crossover and Mutation
  for $j = 1$ to λ **do**
    $i \sim [1 \dots \lambda]$; $k \sim [1 \dots \lambda]$
    $c \sim [1 \dots n]$
    $\theta^{j} := \theta'^{,i}_{[1\dots c[} \circ \theta'^{,k}_{[c\dots n]}$ " $\circ$ = concatenation
    Mutate θ^{j} accordingly
  end for
end while

SANE, ESP, COSYNE There are a lot more evolutionary approaches. Some are not fixed to a certain parameter set size like Genetic Programming (GP) or in the case of NeuroEvolution (NE) the algorithm called NeuroEvolution of Augmenting Topologies (NEAT; [Stanley and Miikkulainen, 2002]). We stick to problems with fixed parameter sets because also PGPE operates in this domain.

A line of NE algorithms is the SANE, ESP, CoSyNE line. We want to point to this field of NE because the most sophisticated and effective algorithm from this group, Cooperatively Coevolved Synapses (CoSyNE; [Gomez et al., 2008]), has some similarities to MultiPGPE (see Section 5.2).

Crossover is destructive in highly correlated search spaces like NN weight matrices

In NN the weights tend to be highly correlated. Crossover is therefore a highly destructive force if one deals with NN. As a result one can optimise the NN by mutation strategies only (e.g. ES) or one can try to use cooperative coevolution to make a more parallel search. For the later one has to find separable subunits of the problem. In NN the subunits are straight forward the neurones with their connecting weights. Every neurone solves in this view a subproblem—combining good neurones together in one network results in a good NN. CoSyNE goes with this view to the maximum and views every weight in the network as one subunit. The result is that every weight is in fact a population of real values. If good values for every weight are selected they build a good NN. The several subpopulations have to work cooperatively together.

For more details on the subject we would like to point the reader to the corresponding publications of Faustino Gomez [Gomez and Miikkulainen, 1999; Gomez et al., 2006; Gomez and Schmidhuber, 2005; Gomez et al., 2008]

Algorithm 2 The Evolution Strategy with a population size of λ, a parent pool of size μ with (μ, λ)-Best Selection and n problem parameters.

Initialise $\theta^{1\dots\lambda}$ randomly
Initialise σ to σ_{init}

while TRUE **do**
  " Evaluation
  evaluate $r^{1\dots\lambda} = r(h(\theta^{1\dots\lambda}))$

  " Best Selection
  for $j = 1$ to μ **do**
    $i = maxarg(r)$
    $\theta'^{,j} = \theta^{i}$
    $\sigma'^{,j} = \sigma^{i}$
    set r^{i} to r_{min}
  end for

  " Mutation
  $curIndi = 1$
  for $j = 1$ to λ **do**
    $\theta^{j} = \theta'^{,curIndi}$
    $\sigma^{j} = \sigma'^{,curIndi}$
    Mutate σ^{j} accordingly
    $\theta^{j} = \mathcal{N}(\theta^{j}, \sigma^{j})$
    $curIndi = (j \mod \mu) + 1$
  end for
end while

3.1.3 *Policy Gradients*

Policy Gradient (PG) methods, so called because they follow the gradient in policy space instead of deriving the policy directly from a value function, are among the few feasible optimisation strategies for complex, high-dimensional reinforcement learning problems with continuous states and actions [Benbrahim and Franklin, 1997; Peters and Schaal, 2006; Schraudolph et al., 2006; Peters et al., 2005]. They are not bound to discrete action and state spaces, they use for instance the parameter set of a continuous function approximator as basis of optimisation.

Policy Gradients are not bound to discrete action and state spaces

The goal of PG as for all RL methods is to find the parameters θ of the controller or model of the agent that maximise the agent's expected reward

$$J(\theta) = \int_H p(h|\theta) r(h) dh. \tag{3.4}$$

Because we assume a stochastic environment the expected reward depends on the probabilities that a certain role out or history h is observed given the set parameters, denoted with $p(h|\theta)$. Maximising the expected reward, however, is done in very different ways throughout the PG methods. In the following we roughly outline the discussion in [Peters and Schaal, 2008a].

Algorithm 3 The Finite Difference algorithm.

Initialise $\Delta_{\theta,0}$ to $\Delta_{\theta init}$

while TRUE **do**
  adapt $\Delta_{\theta,t}$ accordingly
  $i \sim [0, 1, \ldots, number\ of\ parameters - 1]$
  $$\theta^+ = \begin{cases} \theta_{t,j} + \Delta_{\theta,t} & \text{if } j = i \\ \theta_{t,j} & \text{else.} \end{cases}$$
  $$\theta^- = \begin{cases} \theta_{t,j} - \Delta_{\theta,t} & \text{if } j = i \\ \theta_{t,j} & \text{else.} \end{cases}$$
  evaluate $r^+ = r(h(\theta^+))$
  evaluate $r^- = r(h(\theta^-))$

  update $\theta_{t+1} = \theta_t + \alpha \Delta_{\theta,t} \frac{r^+ - r^-}{2}$
end while

FINITE DIFFERENCES Finite Difference (FD) methods originate from the field of stochastic optimisation. The approach is straight forward and important to understand for this thesis, because PGPE is strictly speaking a FD method. FD are the oldest policy gradient methods used already in the 1950s. In the basic FD approach, a set of parameters θ that define the problem solution is varied by a small value Δ_θ for one parameter θ_i at a time. Therefore we have a vector $\boldsymbol{\Delta}_\theta$ with Δ_θ at position i and 0 at all other positions of the vector. For each such single changed parameter set $\theta + \boldsymbol{\Delta}_\theta$ the quality of the solution is evaluated generating an estimate for

In basic FD one parameter is changed by a small value at a time for exploration

$$J^n = J(\theta + \boldsymbol{\Delta}_\theta).$$

The expected reward of the solution can be defined as

$$\boldsymbol{\Delta}_{J^n} \approx J(\theta + \boldsymbol{\Delta}_\theta) - J_{base}.$$

We chose the central difference approach to get J_{base}, because it is used in SPSA in the next paragraph as well as in PGPE in its most current form. Therefore we carry out 2 samples instead of one with the form

Def.: Sample creation

$$\theta^+ = \theta + \boldsymbol{\Delta}_\theta, \quad \theta^- = \theta - \boldsymbol{\Delta}_\theta.$$

The expected return is then

$$\boldsymbol{\Delta}_{J^n} \approx \frac{J(\theta^+) - J(\theta^-)}{2}.$$

The corresponding update rule for parameter θ_i is therefore defined by

Def.: FD update rule

$$\Delta\theta_i = \alpha \Delta_{\theta_i} \frac{(J(\theta^+) - J(\theta^-))}{2}. \tag{3.5}$$

This simple approach is already very powerful. However FD has also some major drawbacks. Δ_θ and its evolution during learning is a manually set parameter that needs a lot of experience of the experimentator

Algorithm 4 The Simultaneous Perturbation Stochastic gradient Approximation algorithm.

Initialise $\Delta_{\theta,0}$ to $\Delta_{\theta init}$

while TRUE **do**
  adapt $\Delta_{\theta,t}$ accordingly
  $b \sim \mathcal{B}(-1,1)$
  $\theta^+ = \theta_t + b\Delta_{\theta,t}$
  $\theta^- = \theta_t - b\Delta_{\theta,t}$

  evaluate $r^+ = r(h(\theta^+))$
  evaluate $r^- = r(h(\theta^-))$

  update $\theta_{t+1} = \theta_t + \alpha\Delta_{\theta,t}\frac{r^+ - r^-}{2}$
end while

and knowledge about the problem. Otherwise the learning process is likely to get unstable and the learning will fail. Also in problems with highly correlated parameters the change of one parameter at a time is prone to get stuck. Choosing Δ_θ in a constant manner also can easily lead into local optima and last the method is more prone to stochastic environments as standard PG methods are, although this effect on realistic robotic applications was widely overestimated.

SIMULTANEOUS PERTURBATION STOCHASTIC GRADIENT APPROXIMATION Especially in robotics, most RL problems are highly correlated in the parameters so that changing one parameter at a time is not leading efficiently to a solution. For that reason in Simultaneous Perturbation Stochastic gradient Approximation (SPSA; [Spall, 1998a,b]) all parameters are changed simultaneously. The parameter sets for evaluation are now $\theta^+ = \theta + \boldsymbol{\Delta}_\theta, \theta^- = \theta - \boldsymbol{\Delta}_\theta$ where $\boldsymbol{\Delta}_\theta$ is drawn from a Bernoulli distribution scaled by the time dependent step size $\epsilon(t)$, i.e. $\Delta\theta_i(t) = \epsilon(t)\,\mathrm{rand}(-1,1)$. In addition, a set of meta-parameters is used to help SPSA converge. The step size ϵ decays according to $\epsilon(t) = \frac{\epsilon(0)}{t^\gamma}$ with $0 < \gamma < 1$. Similarly, the step size α decreases over time with $\alpha = a/(t+A)^E$ for some fixed a, A and E [Spall, 1998a]. The choice of initial parameters $\epsilon(0), \gamma$, a, A and E is critical to the performance of SPSA. In [Spall, 1998b] some guidance is provided for the selection of these coefficients (note that the nomenclature differs from the one used here). With the commonly used central difference approach the final update rule looks very similar to FD:

In SPSA all parameters are changed by a small value for exploration

$$\Delta\theta = \alpha\Delta_\theta \frac{(J(\theta^+) - J(\theta^-))}{2}. \tag{3.6}$$

Def.: SPSA update rule

SPSA performs much more stable in a lot of robotic RL tasks than FD, but still there are the drawbacks of the uniformly distributed exploration that is prone to local minima and the sensitivity to noise in the environment. Also SPSA has a lot of meta-parameters that have to be manually tuned and again a lot of experience is needed here.

REINFORCE While FD methods explore by changing the parameter set directly, likelihood ratio methods explore by changing the deterministic output of the controller in a random way (mostly by defining the

Likelihood ratio methods explore in action space

actions as a normal distribution with μ being the deterministic output of the controller and a given Σ). Thereby the PG method is learning the extend of the noise and better actions by the reward received from the environment.

The most basic and most prominent likelihood ratio method is REINFORCE [Williams, 1992]. REINFORCE again tries to maximise $J(\Theta)$ by estimating $\nabla_\theta J$ and use it to carry out gradient ascent optimisation. Noting that the reward for a particular history is independent of Θ, we can use the standard identity $\nabla_x y(x) = y(x)\nabla_x \log x$ (sometimes called the log trick) to obtain

$$\nabla_\theta J(\theta) = \int_H p(h|\theta)\nabla_\theta \log p(h|\theta) r(h) dh. \tag{3.7}$$

Since the environment is Markovian (as stated in section 2.1 in chapter 2), and the states are conditionally independent of the parameters given the agent's choice of actions, we can write

$$p(h|\theta) = p(s_1)\Pi_{t=1}^{T} p(s_{t+1}|s_t, a_t) p(a_t|s_t, \theta).$$

Substituting this into Eq. (3.7) yields

$$\nabla_\theta J(\theta) = \int_H p(h|\theta) \sum_{t=1}^{T} \nabla_\theta p(a_t|s_t, \theta) r(h) dh. \tag{3.8}$$

Sampling is used for the gradient estimate

Clearly, integrating over the entire space of histories is unfeasible, and we therefore resort to sampling methods

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} \nabla_\theta p(a_t^n|s_t^n, \theta) r(h^n). \tag{3.9}$$

where the histories h^i are chosen according to $p(h^i|\theta)$. The question then is how to model $p(a_t|s_t, \theta)$. In policy gradient methods such as REINFORCE, the parameters θ are used to determine a probabilistic *policy* $\pi_\theta(a_t|s_t) = p(a_t|s_t, \theta)$. A typical policy model would be a parametric function approximator whose outputs define the probabilities of taking different actions. In this case the histories can be sampled by choosing an action at each time step according to the policy distribution, and the final gradient is then calculated by differentiating the policy with respect to the parameters.

By assuming a normal distribution ρ over the actions a_t the derivative results in:

Def.: REINFORCE update rule

$$\begin{aligned} \nabla_\mu \log p(\mathbf{a_t}|\boldsymbol{\rho}) &= \frac{(\mathbf{a_t} - \boldsymbol{\mu})}{\boldsymbol{\sigma}^2}, \\ \nabla_\sigma \log p(\mathbf{a_t}|\boldsymbol{\rho}) &= \frac{(\mathbf{a_t} - \boldsymbol{\sigma})^2 - \boldsymbol{\sigma}^2}{\boldsymbol{\sigma}^3}. \end{aligned} \tag{3.10}$$

This log-likelihoods can be propagated through the controller to determine the parameter change.

There are several advantages to this concept compared to FD methods. PG methods are less prone to noise in the environment, they are less prone to get stuck in a local minima. Also PG methods can be applied to infinite horizon tasks, while FD methods are strictly episodic.

The REINFORCE exploration leads to a high variance in the gradient estimate

However, this approach has also some drawbacks. Sampling from the

Algorithm 5 The REINFORCE algorithm with state independent exploratational noise.

Initialise θ to θinit
Initialise $\boldsymbol{\sigma}$ to σ_{init}

while TRUE **do**
  for $t = 1$ to T **do**
    calculate deterministic action $\mathbf{a_t} \sim p(\mathbf{a}|\mathbf{s_t}, \boldsymbol{\theta})$
    $\mathbf{out_t} \sim \mathcal{N}(\mathbf{a_t}, \mathbf{I}\sigma^2)$
    execute $\mathbf{out_t}$
    append $[\mathbf{out_t}, \mathbf{a_t}, \mathbf{s_t}]$ to h
  end for
  $r = r(h)$

  define $\mathbf{Teach} = (r - b)(\mathbf{out} - \mathbf{a})$ as teaching signal for BackProp
  update $\boldsymbol{\sigma} = \boldsymbol{\sigma} + \alpha_\sigma (r - b) \frac{\mathbf{I}(\mathbf{out}-\mathbf{a})^2 - \mathbf{I}\sigma^2}{\boldsymbol{\sigma}}$
  execute BackProp with $\mathbf{Teach}$
  update baseline b accordingly
end while

policy on every time step leads to a high variance in the sample over histories, and therefore to a noisy gradient estimate, like we already discussed briefly in the introduction of this thesis. Also one needs a differentiable controller while a FD method can also optimise non-differentiable controllers.

GPOMDP GPOMDP takes into consideration that future actions do not depend on past rewards. If this is taken into account, the variance can be reduced compared to REINFORCE. Because we only handle episodic tasks we do not go into details about GPOMDP and refer the reader to [Baxter and Bartlett, 2001].

Future actions do not depend on past rewards

NATURAL ACTOR CRITIC Natural Actor Critic (NAC) uses the natural gradient instead of the REINFORCE gradient for optimising the parameters of the agent. The raw or *vanilla* gradient is normalised by the Fisher information matrix:

$$\nabla_\Theta J_{nat}(\Theta) = G(\Theta)^{-1} J(\Theta) \tag{3.11}$$

Estimation of the Fisher matrix $G(\Theta)$ is hard, but curiously [Sutton et al., 2000] could demonstrate that the estimate of the return can be replaced by a compatible (linear) function approximatior with parameters w. If the return is replaced by the function approximator the derivation simplifies drastically to:

$$\nabla_\Theta J_{nat}(\Theta) = w \tag{3.12}$$

So that the resulting policy improvement step is simply $\Delta\Theta = \alpha w$, and w is easy to estimate. For details please refer to [Peters and Schaal, 2008b].

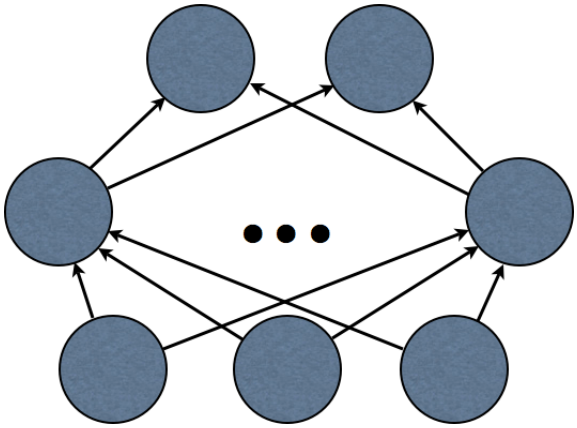


Figure 3.3: Standard Neural Network. The output of the output neurones is the deterministic action of the agent. This is how the controller is defined for FD, ES and the like.

The natural gradient is the more direct path to the optimal solution

NAC has the advantage of faster convergence and avoids premature convergence of *vanilla gradients*. This is due to choosing a more direct path to the optimal solution in parameter space by using the natural gradient. The natural policy gradient is also independent of the coordinate frame chosen for expressing the policy parameters.

But still NAC uses for exploration perturbations of the actions in every time step that results further in noisy gradient estimates for tasks with very long episodes/histories.

3.2 EXPLORATION

A principle problem in Reinforcement Learning is the exploration/exploitation dilemma. The agent needs to explore to find better solutions. The critical question is how big should the amount of exploration be and how long should an agent explore before it settles for the best found solution. A good exploration strategy carefully balances exploration and greedy policy execution.

3.2.1 *Exploration in Reinforcement Learning*

In classical RL like SARSA and Q-Learning we deal with discrete actions. There are several techniques to handle exploration for this learning methods [Thrun, 1992; Wiering and Schmidhuber, 1998]. The most prominent method is ϵ-greedy exploration. In ϵ-greedy a random action is chosen with the probability ϵ, otherwise the action with the best value (with the highest expected reward) is chosen. Usually the value for ϵ decreases over time.

ϵ-greedy exploration

3.2.2 *Exploration in Policy Gradients*

In PG and other RL methods that deal with continuous action-state spaces one cannot use ϵ-greedy anymore or executing a complete random action with ϵ probability is not effective. The most common way of exploring in such cases is to add normal distributed noise to the deterministic (greedy) action of the agent. The executed action

Standard PG explores by adding noise to the actions

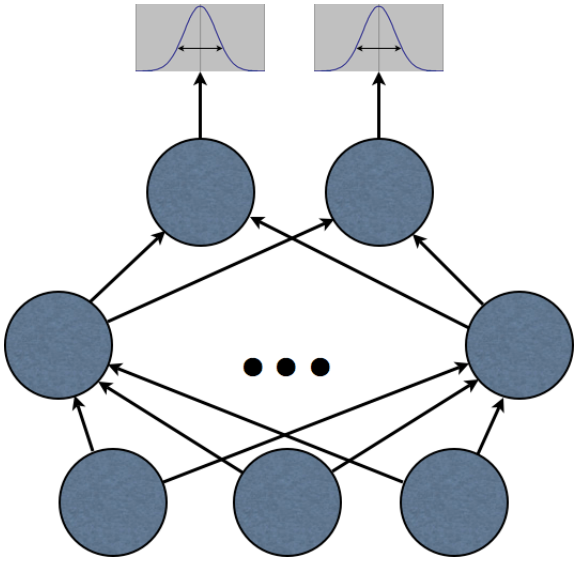


Figure 3.4:
Neural Network with state independent stochastic output. The output of the output neurones is perturbed by normal distributed noise. The deterministic output of the network is thereby the mean of the normal distribution. The standard deviations are free parameters that have to be learned also. This is how the controller is defined for PG methods in the state independent exploration noise case.

is so to speak always a bit different to the action the agent would execute without the explorative noise. The amount of noise is adapted during learning, usually by the same mechanisms that change the parameters of the agent itself. The exploration noise can thereby be independent of the state or state dependent. A positive side effect is that this kind of exploration realises also a stochastic policy. So if stochastic actions are favourable in some situations, the agent will keep a certain standard deviation for the exploration noise forever, thereby automatically realising a stochastic policy.

The major disadvantage of this kind of exploration is that a new difference vector for the action is drawn in every time step. This leads to a very noisy trajectory and therefore to a very noisy gradient estimate. One can introduce a probability of changing the actions for every time step so that the actions are perturbed less frequently, but this also hampers exploration and the use of stochastic policies.

The PG gradient is noisy due to the kind of exploration used

Despite these problems, many current algorithms [Williams, 1992; Peters and Schaal, 2008b; Riedmiller et al., 2007b] use this kind of Gaussian, action-perturbing exploration. Various approaches have been proposed to reduce the high variance that stems from this kind of exploration [Baxter and Bartlett, 2000; Aberdeen, 2003; Peters and Schaal, 2006; Sutton et al., 2000]. However, none of these methods address the underlying cause of the high variance, which is that repeatedly sampling from a probabilistic policy has the effect of injecting noise into the gradient estimate at every time step. Furthermore, the variance increases linearly with the length of the history [Munos and Littman, 2006], since each state may depend on the entire sequence of previous samples.

Exploration can also be done by perturbing the parameters

An alternative to the action-perturbing exploration is to directly manipulate the parameters θ of the policy. This is done in FD methods. Here a certain Δ is used to change a single parameter at a time (FD) or all parameters simultaneously (SPSA) at the beginning of an episode and execute the agent with this new parameter set. This eliminates the noise in the gradient estimate from the above exploration method, but it has

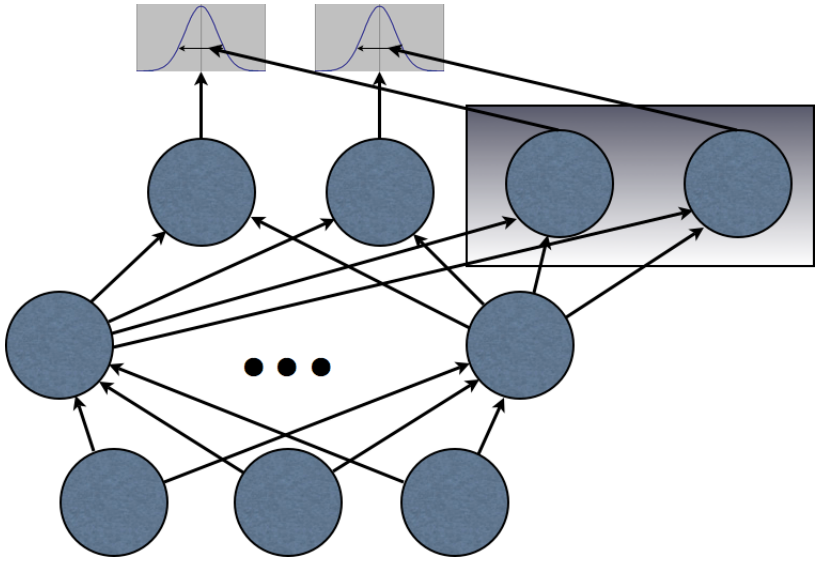


Figure 3.5: Neural Network with state dependent stochastic output. The output of the output neurones is perturbed by normal distributed noise. The deterministic output of the network is thereby the mean of the normal distribution. The standard deviations are the output of additional output neurones. A good choice for the standard deviations is therefore achieved by setting the right weights/parameters in the network. This is how the controller is defined for PG methods in the state dependent exploration noise case.

other drawbacks. First of all, the exploration space is bigger. An agent usually has more parameters than action dimensions, most of the time several orders of magnitudes bigger. This is not so much a drawback for finding the right parameters, because the search space is for both the final parameter set, but if one tracks the amount of exploration for every exploration dimension the effort for doing exploration in parameter space is bigger. Second, learning in stochastic environments is more difficult for FD methods than for standard PG. This effect was however overestimated as also our experiments show. Otherwise a stochastic policy can easily be realised by using an agent with state dependent (exploration) noise as used for standard PG methods, but in contrast this noise isn't used for exploration anymore.

3.2.3 *Exploring in Evolution*

Exploration in Evolution is similar to the FD exploration in the way that the parameter set is changed at the beginning of an episode and then is evaluated. In contrast to FD in evolution a population of solution candidates (individuals) is used. The single parameter sets (called genomes) are derived by adding a normal distributed difference vector to the parent parameter sets. In evolution no gradient is estimated. The n best individuals are taken over and the exploration continues from these parameter sets on. But the main difference that makes evolution superior over FD methods in most cases is that they use normal distributed changes in the parameter sets rather then using fixed Δ changes.

3.2.4 *Exploring in Parameter Space*

Exploration in parameter space is more robust

Some of the mentioned methods above used exploration in action space while some used exploration in parameter space. While studying the different methods in practice it can be roughly observed that the param-

eter exploring methods are learning more robustly. That is especially true for the evolutionary methods. On the other hand, they are more prone to noise in the environment and tend more to overfitting. Also it is a bit more involved to learn stochastic policies with parameter exploration.

The simplest way of exploring in parameter space is the FD or SPSA way. But one quickly realises that the uniformly perturbation of FD methods is not sufficient. The methods are likely to get stuck in local minima or have problems with highly correlated search spaces. The approach of ES is much more robust against these problems and in the case of CMA-ES it is one of the top methods of solving complex robotic tasks. CMA-ES on the other hand has the problem that the genome grows quadratically with the number of problem parameters due to the covariance matrix that is used for exploration. We are especially interested in high-dimensional learning tasks like the later explained grasping with a 7-DoF arm or the robust standing of a 11-DoF humanoid robot. Controllers are likely to have over 1000 parameters and so we refer to the local mutation operator of ES. Here we already have the possibility of the learning method to adjust a normal distribution to the properties in each problem dimension separately.

We concentrate in methods without covariance matrix for exploration

In our method we will derive the update rules from the basic setting of RL in contrast to ES. In principle the algorithm will fall in the class of FD, but will use normal distributed independent perturbations as is usual in PG and ES. We will show that this combination of 3 worlds has several advantages over the state of the art in robot behavioural RL.

An interesting alternative to the *black and white* distinction between exploration in action and in parameter space is the so called State Dependent Exploration (SDE) [Rückstieß et al., 2008]. Here one can use standard PG methods with all the recently developed improvements without change. The exploration however is done by an extra exploration *module* that is itself parameterised and generates difference vectors that are a function of the state. The parameters of the exploration module can be learned similar to PGPE. The main advantages are (i) that one can use the standard PG methods and just replace the exploration; (ii) The exploration module can often be parameterised with less parameters than the controller of the agent (but not too small to have still complex exploration).

State Dependent Exploration as an alternative

The disadvantages are that one cannot use this method for non-differentiable controllers anymore and that one needs to propagate the log-likelihoods via back-propagation or similar gradient decent methods through the controller.

To date the two methods have not been compared directly, but from personal communication we assume that SDE is superior in flat controllers with a low action to parameter dimension ratio, while PGPE is superior in deep controller architectures with a higher action to parameter dimension ratio. Exploring the properties of the different exploration and learning strategies is in our opinion an interesting field of future work.

4 PART SUMMARY AND CONCLUSION

We motivated the use of RL in robotics in Chapter 1, especially the use of PG. We motivated the use of parameter based exploration and gave considerations of the advantages and drawbacks.

In Chapter 2, we explained the MDP/POMDP setting RL takes place in. We highlighted the difference between episodic and ongoing tasks/rewards.

In Chapter 3, we explained the classical RL methods SARSA and Q-Learning, artificial evolution with the specific algorithms GA and ES and some insights in SANE, ESP and CoSyNE. We showed how PG methods work starting with FD over SPSA to REINFORCE and gave some insights into GPOMDP and NAC.

We highlighted how exploration works in the different kinds of ML methods and we identified the two principal ways of exploration in PG: exploration by perturbing the actions of the agent and exploration by perturbing the parameters of the agent.

We gave a short discussion in several passages in this Part explaining why exploring in parameter space is the better exploration strategy for the field of RL we are interested in.

Part II

NEW CONTRIBUTION

5 PARAMETER EXPLORING POLICY GRADIENTS

The class of Parameter Exploring Policy Gradients

The core contribution of this thesis is the derivation of the new class of Policy Gradient (PG) methods called Parameter Exploring Policy Gradients (PEPG). The defining property of all PEPG methods is that they explore by using samples in parameter space and estimate a log-likelihood gradient directly on parameter level. The basic method of PEPGs is the so called Policy Gradients with Parameter-based Exploration (PGPE) algorithm.

Goal: Deriving PGPE

In this Chapter we derive the PGPE algorithm and its variants from the general framework of episodic reinforcement learning in a Markovian environment (see 2.1 for MDP and POMDP explanations). In doing so we highlight the differences between PGPE (and its variants) and standard policy gradient methods such as REINFORCE (see 3.1.3). Consider an agent interacting with an environment. Denote the state of the environment at time t as s_t and the action at time t as a_t. Because we are interested in continuous state and action spaces (usually required for control tasks), we represent both a_t and s_t with real valued vectors. Each state–action pair gives the agent a scalar reward $r_t(a_t, s_t)$. The agent's actions stochastically depend on the current state and some real valued parameter vector θ, with:

$$a_t \sim p(a|s_t, \theta). \tag{5.1}$$

Def.: a_t, action selection

We assume that the environment is Markovian, i.e. that the probability distribution over the possible next states is conditionally independent of all previous state–action pairs given the current one:

$$s_{t+1} \sim p(s|s_t, a_t). \tag{5.2}$$

Def.: s_{t+1}, state transition

We refer to a sequence h of state–action pairs of length T produced by an agent as a *history*:

$$h = [s_{1:T}, a_{1:T}]. \tag{5.3}$$

Def.: h, history

Given the above formulation we can associate a cumulative reward with each history h by summing over the rewards at each time step:

$$r(h) = \sum_{t=1}^{T} r_t. \tag{5.4}$$

Def.: r, episodic reward

In this setting, the goal of reinforcement learning is to find the parameters θ that maximise the agent's expected reward

$$J(\theta) = \int_H p(h|\theta) r(h) dh. \tag{5.5}$$

Def.: J, expected reward

An obvious way to maximise $J(\theta)$ is to find $\nabla_\theta J$ and use it to carry out gradient ascent. Noting that $r(h)$ is independent of θ, and using the standard identity

$$\nabla_x y(x) = y(x) \nabla_x \log y(x),$$

we can write

$$\nabla_\theta J(\theta) = \int_H p(h|\theta) \nabla_\theta \log p(h|\theta) r(h) dh. \tag{5.6}$$

Since the environment is Markovian, and the states are conditionally independent of the parameters given the agent's choice of actions, we can write

$$p(h|\theta) = p(s_1) \prod_{t=1}^{T} p(s_{t+1}|s_t, a_t) p(a_t|s_t, \theta).$$

Substituting this into Eq. (5.6) yields

$$\nabla_\theta J(\theta) = \int_H p(h|\theta) \sum_{t=1}^{T} \nabla_\theta \log p(a_t|s_t, \theta) r(h) dh. \tag{5.7}$$

Clearly, integrating over the entire space of histories is unfeasible, and we therefore resort to sampling methods

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log p(a_t^n|s_t^n, \theta) r(h^n), \tag{5.8}$$

where the histories h^i are chosen according to $p(h^i|\theta)$. The question then is how to model $p(a_t|s_t, \theta)$. In policy gradient methods such as REINFORCE the parameters θ are used to determine a probabilistic *policy*

$$\pi_\theta(a_t|s_t) = p(a_t|s_t, \theta).$$

Gradient estimation with policy distribution

A typical policy model would be a parametric function approximator whose outputs define the probabilities of taking different actions. In this case the histories can be sampled by choosing an action at each time step according to the policy distribution, and the final gradient is then calculated by differentiating the policy with respect to the parameters. However, sampling from the policy on every time step leads to a high variance in the sample over histories, and therefore to a noisy gradient estimate.

Gradient estimation with parameter distribution

PGPE addresses the variance problem by replacing the probabilistic policy with a probability distribution over the parameters θ, i.e.

$$p(a_t|s_t, \rho) = \int_\theta p(\theta|\rho) \delta_{F_\theta(s_t), a_t} d\theta, \tag{5.9}$$

where ρ are the parameters determining the distribution over θ, $F_\theta(s_t)$ is the (deterministic) action chosen by the model with parameters θ in state s_t, and δ is the Dirac delta function. The advantage of this approach is that the actions are deterministic, and an entire history can therefore be generated from a single parameter sample. This reduction in samples-per-history is what reduces the variance in the gradient estimate (see [Sehnke et al., 2010a]). As an additional benefit the parameter gradient is estimated by direct parameter perturbations, without having to *backpropagate* any derivatives, which allows the use of non-differentiable controllers.

One sample per history reduces variance

Direct parameter perturbations allow the use of non-differentiable controllers

The expected reward with a given ρ is

$$J(\rho) = \int_\Theta \int_H p(h, \theta|\rho) r(h) dh d\theta. \tag{5.10}$$

Differentiating this form of the expected return with respect to $\boldsymbol{\rho}$ and applying the log trick as before we obtain

$$\nabla_{\boldsymbol{\rho}} J(\boldsymbol{\rho}) = \int_{\Theta} \int_{H} p(h, \boldsymbol{\theta}|\boldsymbol{\rho}) \nabla_{\boldsymbol{\rho}} \log p(h, \boldsymbol{\theta}|\boldsymbol{\rho}) r(h) \, dh \, d\boldsymbol{\theta}. \tag{5.11}$$

Noting that h is conditionally independent of $\boldsymbol{\rho}$ given $\boldsymbol{\theta}$, we have

$$p(h, \boldsymbol{\theta}|\boldsymbol{\rho}) = p(h|\boldsymbol{\theta}) p(\boldsymbol{\theta}|\boldsymbol{\rho})$$

and therefore

$$\nabla_{\boldsymbol{\rho}} \log p(h, \boldsymbol{\theta}|\boldsymbol{\rho}) = \nabla_{\boldsymbol{\rho}} \log p(\boldsymbol{\theta}|\boldsymbol{\rho}),$$

we have

$$\nabla_{\boldsymbol{\rho}} J(\boldsymbol{\rho}) = \int_{\Theta} \int_{H} p(h|\boldsymbol{\theta}) p(\boldsymbol{\theta}|\boldsymbol{\rho}) \nabla_{\boldsymbol{\rho}} \log p(\boldsymbol{\theta}|\boldsymbol{\rho}) r(h) \, dh \, d\boldsymbol{\theta}, \tag{5.12}$$

where $p(h|\boldsymbol{\theta})$ is the probability distribution over the parameters $\boldsymbol{\theta}$ and $\boldsymbol{\rho}$ are the parameters determining the distribution over $\boldsymbol{\theta}$. Integrating over the entire space of histories and parameters is again unfeasible, and we therefore resort to sampling methods. This is done by first choosing $\boldsymbol{\theta}$ from $p(\boldsymbol{\theta}|\boldsymbol{\rho})$, then running the agent to generate h from $p(h|\boldsymbol{\theta})$:

$$\nabla_{\boldsymbol{\rho}} J(\boldsymbol{\rho}) \approx \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\rho}} \log p(\boldsymbol{\theta}|\boldsymbol{\rho}) r(h^n). \tag{5.13}$$

Sampling with parameter distribution

5.1 UNIMODAL PARAMETER DISTRIBUTIONS—PGPE

In the original formulation of PGPE, ρ consisted of a set of means $\{\mu_i\}$ and standard deviations $\{\sigma_i\}$ that determine an independent normal distribution for each parameter θ_i in $\boldsymbol{\theta}$ of the form

$$p(\theta_i|\rho_i) = \mathcal{N}(\theta_i|\mu_i, \sigma_i^2).$$

Some rearrangement gives the following forms for the derivative of $\log p(\boldsymbol{\theta}|\rho)$ with respect to μ_i and σ_i:

PGPE update rules

$$\nabla_{\mu_i} \log p(\boldsymbol{\theta}|\boldsymbol{\rho}) = \frac{(\theta_i - \mu_i)}{\sigma_i^2},$$

$$\nabla_{\sigma_i} \log p(\boldsymbol{\theta}|\boldsymbol{\rho}) = \frac{(\theta_i - \mu_i)^2 - \sigma_i^2}{\sigma_i^3}, \tag{5.14}$$

which can then be substituted into (5.13) to approximate the $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ gradients that gives the PGPE update rules. Note the similarity to REINFORCE [Williams, 1992]. But in contrast to REINFORCE, $\boldsymbol{\theta}$ defines the parameters of the model, not the probability of the actions.

5.1.1 *Sampling with a baseline*

Given enough samples, Eq. (5.13) will determine the reward gradient to arbitrary accuracy. However each sample requires rolling out an entire state–action history which is expensive. Following [Williams, 1992], we obtain a cheaper gradient estimate by drawing a single sample $\boldsymbol{\theta}$ and comparing its reward r to a baseline reward b given by a moving average over previous samples. Intuitively, if $r > b$ we adjust $\boldsymbol{\rho}$ so as to increase the probability of $\boldsymbol{\theta}$, and $r < b$ we do the opposite.

Moving average baseline

If, as in [Williams, 1992], we use a step size $\alpha_i = \alpha\sigma_i^2$ in the direction of positive gradient (where α is a constant) we get the following parameter update equations:

Baseline PGPE update rules

$$\Delta\mu_i = \alpha_\mu(r-b)(\theta_i-\mu_i),$$
$$\Delta\sigma_i = \alpha_\sigma(r-b)\frac{(\theta_i-\mu_i)^2-(\sigma_i)^2}{\sigma_i}. \tag{5.15}$$

5.1.2 *Symmetric sampling*

While sampling with a baseline is efficient and reasonably accurate for most scenarios, it has several drawbacks. In particular, if the reward distribution is strongly skewed then the comparison between the sample reward and the baseline reward is misleading. A more robust gradient approximation can be found by measuring the difference in reward between two symmetric samples on either side of the current mean. That is, we pick a perturbation $\boldsymbol{\epsilon}$ from the distribution $\mathcal{N}(0,\boldsymbol{\sigma})$, then create symmetric parameter samples

Def.: Symmetric Samples

$$\boldsymbol{\theta}^+ = \boldsymbol{\mu}+\boldsymbol{\epsilon}, \quad \boldsymbol{\theta}^- = \boldsymbol{\mu}-\boldsymbol{\epsilon}.$$

Defining r^+ as the reward given by $\boldsymbol{\theta}^+$ and r^- as the reward given by $\boldsymbol{\theta}^-$, we can insert the two samples into Eq. (5.13) to obtain

$$\nabla_{\mu_i} J(\boldsymbol{\rho}) \approx \frac{\epsilon_i(r^+-r^-)}{2(\sigma_i)^2}, \tag{5.16}$$

which resembles the *central difference* approximation used in finite difference methods. Using the same step sizes as before gives the following update equation for the $\boldsymbol{\mu}$ terms

SyS-PGPE μ update rule

$$\Delta\mu_i = \frac{\alpha_\mu\epsilon_i(r^+-r^-)}{2}. \tag{5.17}$$

The updates for the standard deviations are more involved. As $\boldsymbol{\theta}^+$ and $\boldsymbol{\theta}^-$ are by construction equally probable under a given $\boldsymbol{\sigma}$, the difference between them cannot be used to estimate the $\boldsymbol{\sigma}$ gradient. Instead we take the mean $(r^+ + r^-)/2$ of the two rewards and compare it to the baseline reward b. This approach yields

SyS-PGPE σ update rule

$$\Delta\sigma_i = \alpha_\sigma\left(\frac{r^++r^-}{2}-b\right)\left(\frac{\epsilon_i^2-(\sigma_i)^2}{\sigma_i}\right). \tag{5.18}$$

Compared to the method in Section 5.1.1, symmetric sampling removes the problem of misleading baselines, and therefore improves the $\boldsymbol{\mu}$ gradient estimates. It also improves the $\boldsymbol{\sigma}$ gradient estimates, since both samples are equally probable under the current distribution, and therefore reinforce each other as predictors of the benefits of altering $\boldsymbol{\sigma}$. Even though symmetric sampling requires twice as many histories per update, our experiments show that it gives a considerable improvement in convergence quality and time.

Algorithm 6 The PGPE algorithm with Reward Normalisation and Symmetric Sampling. $\mathbf{T}$ and $\mathbf{S}$ are $P \times N$ matrices with P the number of parameters and N the number of histories. The baseline is updated accordingly after each step. α is the learning rate or step size.

Initialise $\boldsymbol{\mu}$ to $\boldsymbol{\mu}_{\text{init}}$
Initialise $\boldsymbol{\sigma}$ to $\boldsymbol{\sigma}_{\text{init}}$

while TRUE **do**
  for $n = 1$ to N **do**
    draw perturbation $\boldsymbol{\epsilon}^n \sim \mathcal{N}(\mathbf{0}, \mathbf{I}\boldsymbol{\sigma}^2)$
    $\boldsymbol{\theta}^{+,n} = \boldsymbol{\mu} + \boldsymbol{\epsilon}^n$
    $\boldsymbol{\theta}^{-,n} = \boldsymbol{\mu} - \boldsymbol{\epsilon}^n$
    evaluate $r^{+,n} = r(h(\boldsymbol{\theta}^{+,n}))$
    evaluate $r^{-,n} = r(h(\boldsymbol{\theta}^{-,n}))$
    if $r^{+,n} > m$ then $m := r^{+,n}$
    if $r^{-,n} > m$ then $m := r^{-,n}$
  end for

  $\mathbf{T} = [\, t_{ij} \,]_{ij}$ with $t_{ij} := \epsilon_i^j$

  $\mathbf{S} = [\, s_{ij} \,]_{ij}$ with $s_{ij} := \frac{(\epsilon_i^j)^2 - \sigma_i^2}{\sigma_i}$

  $\mathbf{r}_T = \left[\frac{(r^{+,1} - r^{-,1})}{2m - r^{+,1} - r^{-,1}}, \ldots, \frac{(r^{+,N} - r^{-,N})}{2m - r^{+,N} - r^{-,N}} \right]^T$

  $\mathbf{r}_S = \left[\frac{(r^{+,1} + r^{-,1} - 2b)}{2(m-b)}), \ldots, (\frac{(r^{+,N} + r^{-,N} - 2b)}{2(m-b)}) \right]^T$

  update $\boldsymbol{\mu} = \boldsymbol{\mu} + \alpha \mathbf{T}\mathbf{r}_T$
  update $\boldsymbol{\sigma} = \boldsymbol{\sigma} + \alpha \mathbf{S}\mathbf{r}_S$
  update baseline b accordingly
end while

5.1.3 *Reward Normalisation*

As a final refinement, we make the step size independent from the (possibly unknown) scale of the rewards by introducing a normalisation term. Let m be the maximum reward the agent can receive, if this is known, or the maximum reward received so far if it is not. We normalise the $\boldsymbol{\mu}$ updates by dividing them by the difference between m and the mean reward of the symmetric samples, and we normalise the $\boldsymbol{\sigma}$ updates by dividing by the difference between m and the baseline b, yielding

Normalising gradient with the maximal (observed) reward

$$\Delta\mu_i = \frac{\alpha_\mu \epsilon_i (r^+ - r^-)}{2m - r^+ - r^-},$$
$$\Delta\sigma_i = \frac{\alpha_\sigma}{m - b} \left(\frac{r^+ + r^-}{2} - b \right) \left(\frac{(\epsilon_i)^2 - (\sigma_i)^2}{\sigma_i} \right), \tag{5.19}$$

Standard PGPE update rules

where

$$\epsilon_i \stackrel{\text{def}}{=} \theta_i - \mu_i.$$

5.2 MULTIMODAL PARAMETER DISTRIBUTIONS — MULTIPGPE

In MultiPGPE $\boldsymbol{\rho}$ consists of a set of mixing coefficients $\{\pi_i^k\}$, means $\{\mu_i^k\}$ and standard deviations $\{\sigma_i^k\}$ defining an independent mixture of Gaussians for each parameter θ_i:

$$p(\theta_i|\rho_i) = \sum_{k=1}^{K} \pi_i^k \mathcal{N}(\theta_i|\mu_i^k, (\sigma_i^k)^2), \tag{5.20}$$

where

$$\sum_{k=1}^{K} \pi_i^k = 1,\ \pi_i^k \in [0,1]\ \forall i,k.$$

The derivatives of $\log p(\boldsymbol{\theta}|\boldsymbol{\rho})$ are now as follows:

MultiPGPE update rules

$$\begin{aligned}
\nabla_{\pi_i^k} \log p(\boldsymbol{\theta}|\boldsymbol{\rho}) &= l_i^k, \\
\nabla_{\mu_i^k} \log p(\boldsymbol{\theta}|\boldsymbol{\rho}) &= l_i^k \frac{(\theta_i - \mu_i^k)}{(\sigma_i^k)^2}, \\
\nabla_{\sigma_i^k} \log p(\boldsymbol{\theta}|\boldsymbol{\rho}) &= l_i^k \frac{(\theta_i - \mu_i^k)^2 - (\sigma_i^k)^2}{(\sigma_i^k)^3},
\end{aligned} \tag{5.21}$$

where

$$l_i^k \stackrel{\text{def}}{=} \frac{\mathcal{N}(\theta_i|\mu_i^k, (\sigma_i^k)^2)}{p(\theta_i|\rho_i)}.$$

These can again be substituted into Eq. (5.13) to approximate $\nabla_{\boldsymbol{\rho}} J(\boldsymbol{\rho})$.

We use a fix number of 10 Gaussians per mixture. The μ and σ of each Gaussian is chosen in a way that the mixture forms a uniform distribution over the search interval like shown in Fig. 5.1.

5.2.1 *Simplified MultiPGPE*

From the sum and product rules follows that sampling from a mixture distribution like defined in (5.20) is equivalent to first choosing a com-

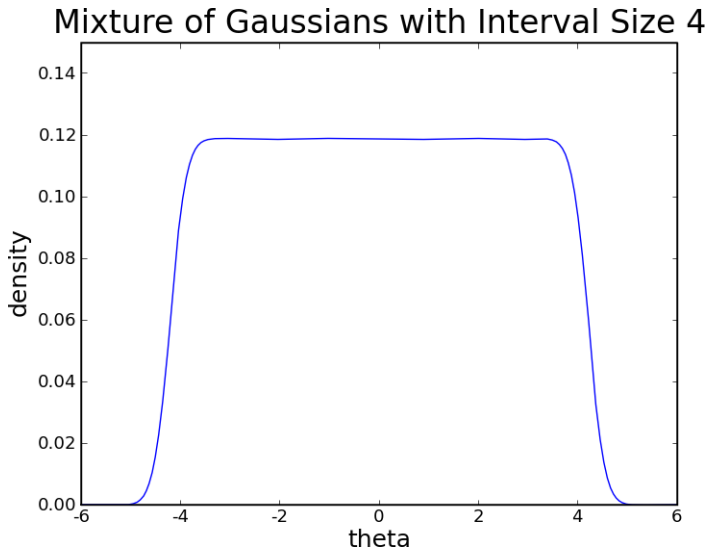


Figure 5.1: 10 Gaussians forming the closest possible approximation to an equal distribution in the interval -4 – 4.

ponent according to the probability distribution defined by the mixing coefficients, then sampling from that component:

$$p(\theta_i|\rho_i) = \sum_{k=1}^{K} p(k|\boldsymbol{\pi})\mathcal{N}(\theta_i|\mu_i^k, (\sigma_i^k)^2),$$

if we define $\pi_k = p(k|\boldsymbol{\pi})$ as the prior probability of picking the k^{th} Gaussian.

This suggests the following simplification to the MultiPGPE gradient calculations: first pick k with probability π_i^k, then set

$$l_i^k = 1, \quad l_i^{k'} = 0 \; \forall k' \neq k. \tag{Simplification}$$

Substituting into Eq. (5.21) we see that

$$\nabla_{\pi_i^k} \log p(\boldsymbol{\theta}|\boldsymbol{\rho}) = 1$$

and the other gradients reduce to their unimodal form in Eq. (5.36).

As well removing the computational effort of calculating the l_i^k terms, this simplification has other advantages. First, it tends to push the mixing coefficients towards one or zero, which prevents multiple components with similar means and variances *shadowing* each other, and speeds up the decision between overlapping components (see Fig. 6.6). Second, as we will see in Section 5.2.3, it allows us to improve convergence by picking symmetric parameter samples from either side of the mean of the chosen component.

Simplification allows Symmetric Sampling in MultiPGPE

5.2.2 *Sampling with a baseline*

As for PGPE we can use a step size $\alpha_i = \alpha\sigma_i^2$ in the direction of positive gradient (where α is a constant) in combination with a baseline b. In doing so we get the following parameter update equations for MultiPGPE:

$$\begin{aligned}
\Delta\pi_i^k &= \alpha_\pi (r-b) l_i^k, \\
\Delta\mu_i^k &= \alpha_\mu (r-b) l_i^k (\theta_i - \mu_i^k), \\
\Delta\sigma_i^k &= \alpha_\sigma (r-b) l_i^k \frac{(\theta_i - \mu_i^k)^2 - (\sigma_i^k)^2}{\sigma_i^k}.
\end{aligned} \tag{5.22}$$

MultiPGPE baseline update rules

5.2.3 *Symmetric sampling*

For the simplified version of MultiPGPE we also can use SyS. That is, we pick again a perturbation $\boldsymbol{\epsilon}$ from the distribution $\mathcal{N}(\mathbf{0}, \boldsymbol{\sigma})$, then create symmetric parameter samples

$$\boldsymbol{\theta}^+ = \boldsymbol{\mu} + \boldsymbol{\epsilon}, \quad \boldsymbol{\theta}^- = \boldsymbol{\mu} - \boldsymbol{\epsilon}.$$

This is only possible for the simplified version of MultiPGPE described in Section 5.2.1, since the full mixture distribution used for MultiPGPE is not symmetrical. Again defining r^+ as the reward given by $\boldsymbol{\theta}^+$ and r^- as the reward given by $\boldsymbol{\theta}^-$, we can insert the two samples into Eq. (5.13) and make use of Eq. (5.21) with $l_i^k = 1$ to obtain

$$\nabla_{\mu_i^k} J(\boldsymbol{\rho}) \approx \frac{\epsilon_i (r^+ - r^-)}{2(\sigma_i^k)^2}, \tag{5.23}$$

which resembles again the *central difference* approximation used in finite difference methods. Using the same step sizes as before gives the following update equation for the μ terms

Simplified MultiPGPE μ update rule

$$\Delta\mu_i^k = \frac{\alpha_\mu \epsilon_i (r^+ - r^-)}{2}. \tag{5.24}$$

Equivalently to the unimodal case θ^+ and θ^- are by construction equally probable under a given σ and result from the same distribution with mixing coefficient π, the difference between them cannot be used to estimate the σ or π gradient. Instead we take the mean $(r^+ + r^-)/2$ of the two rewards and compare it to the baseline reward b. This approach yields

Simplified MultiPGPE σ update rule

$$\begin{aligned}\Delta\sigma_i^k &= \alpha_\sigma \left(\frac{r^+ + r^-}{2} - b\right)\left(\frac{\epsilon_i^2 - (\sigma_i^k)^2}{\sigma_i^k}\right),\\ \Delta\pi_i^k &= \alpha_\pi \left(\frac{r^+ + r^-}{2} - b\right).\end{aligned} \tag{5.25}$$

Note again that SyS is only possible for the simplified version of MultiPGPE. This is the reason that no l_i^k term appears in above equations because it is assumed to be 1.

5.2.4 *Reward Normalisation*

Normalising gradient with maximal (observed) reward

Also for the multimodal case it is important to make the step size independent from the (possibly unknown) scale of the rewards by introducing a normalisation term. Let m be the maximum reward the agent can receive, if this is known, or the maximum reward received so far if it is not. We normalise the μ updates by dividing them by the difference between m and the mean reward of the symmetric samples, and we normalise the π and σ updates by dividing by the difference between m and the baseline b, yielding

Standard MultiPGPE update rules

$$\begin{aligned}\Delta\pi_i^k &= \frac{\alpha_\pi}{m-b}\left(\frac{r^+ + r^-}{2} - b\right),\\ \Delta\mu_i^k &= \frac{\alpha_\mu \epsilon_i^k (r^+ - r^-)}{2m - r^+ - r^-},\\ \Delta\sigma_i^k &= \frac{\alpha_\sigma}{m-b}\left(\frac{r^+ + r^-}{2} - b\right)\left(\frac{(\epsilon_i^k)^2 - (\sigma_i^k)^2}{\sigma_i^k}\right),\end{aligned} \tag{5.26}$$

where

$$\epsilon_i^k \stackrel{\text{def}}{=} \theta_i - \mu_i^k.$$

For non-simplified MultiPGPE the reward normalisation is applied to the baseline sampling of Sec. 5.2.2:

$$\begin{aligned}\Delta\pi_i^k &= \alpha_\pi l_i^k \frac{r-b}{m-b},\\ \Delta\mu_i^k &= \alpha_\mu l_i^k \frac{r-b}{m-b}(\theta_i - \mu_i^k)\\ \Delta\sigma_i^k &= \alpha_\sigma l_i^k \frac{r-b}{m-b}\frac{(\theta_i - \mu_i^k)^2 - (\sigma_i^k)^2}{\sigma_i^k}.\end{aligned} \tag{5.27}$$

Algorithm 7 The simplified MultiPGPE algorithm with Reward Normalisation and Symmetric Sampling. $\mathbf{T}$ and $\mathbf{S}$ are $P \times N$ matrices with P the number of parameters and N the number of histories. The baseline is updated accordingly after each step. α is the learning rate or step size.

Initialise $\boldsymbol{\pi}$ to $\boldsymbol{\pi}_{\text{init}}$
Initialise $\boldsymbol{\mu}$ to $\boldsymbol{\mu}_{\text{init}}$
Initialise $\boldsymbol{\sigma}$ to $\boldsymbol{\sigma}_{\text{init}}$

while TRUE **do**
  for $n = 1$ to N **do**
    draw Gaussians $\mathbf{k}^n \sim p(k|\boldsymbol{\pi})$
    draw perturbation $\boldsymbol{\epsilon}^n \sim \mathcal{N}(\mathbf{0}, \mathbf{I}\boldsymbol{\sigma}_{\mathbf{k}}^2)$
    $\theta^{+,n} = \boldsymbol{\mu}_{\mathbf{k}} + \boldsymbol{\epsilon}^n$
    $\theta^{-,n} = \boldsymbol{\mu}_{\mathbf{k}} - \boldsymbol{\epsilon}^n$
    evaluate $r^{+,n} = r(h(\theta^{+,n}))$
    evaluate $r^{-,n} = r(h(\theta^{-,n}))$
    if $r^{+,n} > m$ then $m := r^{+,n}$
    if $r^{-,n} > m$ then $m := r^{-,n}$
  end for

  $\mathbf{T} = [\, t_{ij} \,]_{ij}$ with $t_{ij} := \epsilon_i^j$

  $\mathbf{S} = [\, s_{ij} \,]_{ij}$ with $s_{ij} := \frac{(\epsilon_i^j)^2 - \sigma^2_{i,k_i^j}}{\sigma_{i,k_i^j}}$

  $\mathbf{r}_T = [\frac{(r^{+,1} - r^{-,1})}{2m - r^{+,1} - r^{-,1}}, \ldots, \frac{(r^{+,N} - r^{-,N})}{2m - r^{+,N} - r^{-,N}}]^T$

  $\mathbf{r}_S = [\frac{(r^{+,1} + r^{-,1} - 2b)}{2(m-b)}), \ldots, (\frac{(r^{+,N} + r^{-,N} - 2b}{2(m-b)})]^T$

  update $\boldsymbol{\pi} = \boldsymbol{\pi} + \alpha \mathbf{r}_S$
  update $\boldsymbol{\mu} = \boldsymbol{\mu} + \alpha \mathbf{T}\mathbf{r}_T$
  update $\boldsymbol{\sigma} = \boldsymbol{\sigma} + \alpha \mathbf{S}\mathbf{r}_S$
  update baseline b accordingly
end while

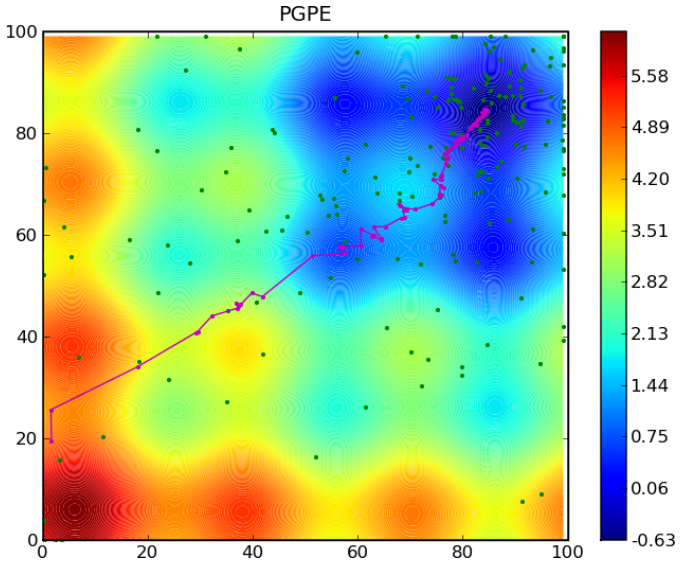


Figure 5.2: Convergence example of PGPE. The green markers are the samples. The magenta markers are the means over the learning process. One can nicely see that PGPE goes directly for the global optima ignoring all local optima on the way.

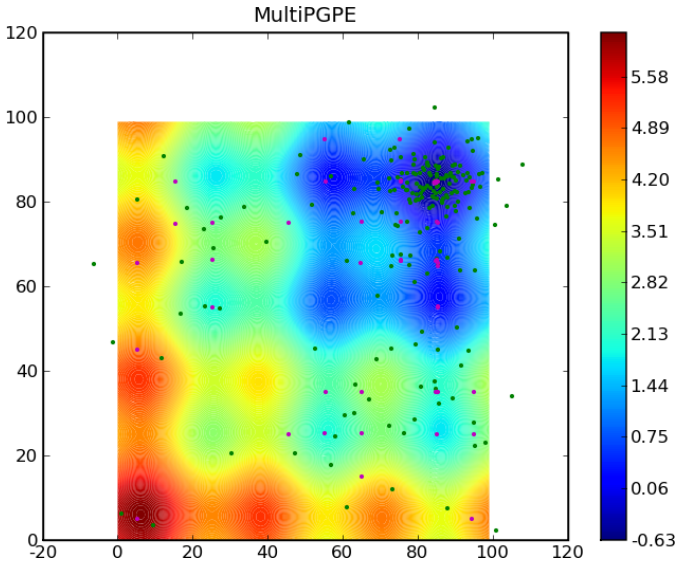


Figure 5.3: Convergence example of MultiPGPE. The green markers are the samples. The magenta markers are the means over the learning process. One can see that MultiPGPE conducts a global search eventually condensing on the global optima.

5.3 INFINITE HORIZON PGPE

PGPE is episodic

In the original formulation of PGPE, $\boldsymbol{\rho}$ consisted of a set of means $\{\mu_i\}$ and standard deviations $\{\sigma_i\}$ that define the parameters θ chosen from $\boldsymbol{\rho}$ at the beginning of each episode. The original PGPE algorithm is therefore strictly episodic. To apply PGPE to infinite horizon settings, this mechanism has to be changed. In so-called Infinite Horizon PGPE (IHPGPE), changing the parameters and learning are carried out simultaneously, while interacting with the environment.

Simultaneous exploration and learning for Infinite Horizon PGPE

The agent still receives a scalar reward $r_t(a_t, s_t)$ for each state–action pair (a_t, s_t). However this reward is no longer summed over an episode. Furthermore, the agent's actions now depend stochastically on a real valued parameter vector θ that changes over time, as well as on the current state. Eq. (5.1) therefore changes to:

$$a_t \sim p(a|s_t, \boldsymbol{\theta_t}). \tag{5.28}$$

Def.: a_t, time-dependent action selection

The state transition is independent of θ, so Eq. (5.2) is also valid for IHPGPE. Since we no longer have a cumulative reward over an episode, and the parameter set is time-dependent, the expected reward changes to:

$$J(\theta) = \sum^{T} \int_a p(a|s_t, \boldsymbol{\theta_t}) r_t(a_t, s_t) da. \tag{5.29}$$

Time-dependent expected reward

Note that Eq. (5.29) does not take into account that the reward r_t at time step t was also *earned* by earlier state action-pairs. The backward dependency of rewards is well known in RL, and the standard solution is to use *Eligibility Traces* (ET, [Sutton and Barto, 1998] Ch.7). With an ET one defines a decay constant γ with $0 < \gamma < 1$. The so-called *backward view* of ET is that the influence of past state action pairs, and therefore the credit assignment to past θs, decays over time with:

Eligibility traces

$$r_{\theta_u} = r_t \gamma^{t-u}, \tag{5.30}$$

where t is the current time step and u is a previous time step.

The *forward view* of ET is obtained by summing over all weighted future rewards for the state action pair a certain time step:

$$r_u = (1-\gamma) \sum_{t=u}^{T} r_t \gamma^{t-u}. \tag{5.31}$$

Def.: r_u, weighted future reward

The term $(1-\gamma)$ is used for normalisation, because of the following statement:

$$\sum_{x=0}^{\infty} \gamma^x \to \frac{1}{1-\gamma}.$$

If we substitute this definition of credit assignment into (5.29) we get:

$$J(\theta) = (1-\gamma) \sum_{u=0}^{T} \sum_{t=u}^{T} \int_a p(a|s_u, \boldsymbol{\theta_u}) r_t(a_t, s_t) \gamma^{t-u} da. \tag{5.32}$$

By following the same argumentation that led to Eq. (5.13) for the episodic case we get:

$$\nabla_{\boldsymbol{\rho}} J(\boldsymbol{\rho}) \approx \frac{1}{T}(1-\gamma) \sum_{u=0}^{T} \sum_{t=u}^{T} \nabla_{\boldsymbol{\rho_u}} \log p(\boldsymbol{\theta_u}|\boldsymbol{\rho_u}) r_t \gamma^{t-u}. \tag{5.33}$$

In normal PGPE the parameters $\boldsymbol{\theta_t}$ are drawn them from the distribution $p(\boldsymbol{\theta}|\boldsymbol{\rho})$ at the beginning of an episode. In the infinite horizon case the parameters θ_t can be drawn consecutively as follows:

Drawing parameters consecutively

$$p(\theta_{i,t+1}|\rho_{i,t}) = \begin{cases} \mathcal{N}(\theta_{i,t+1}|\mu_{i,t}, \sigma_{i,t}^2) & \text{if } rand_{i,t} < c, \\ \theta_{i,t} & \text{else.} \end{cases} \tag{5.34}$$

where $rand_{i,t} < c$ is the likelihood of changing a single parameter at a given time step. Most commonly a random number from the interval $[0 \dots 1[$ is drawn and compared to a constant $0 \leqslant c < 1$. The constant c should be chosen so that the expected frequency of changing a single parameter is in the same order of magnitude as an episode length would be in a similar episodic task definition.

Perturbation frequency

Alternatively, one could draw all parameters at a certain time step simultaneously:

Drawing parameters simultaneously

$$p(\boldsymbol{\theta_{t+1}}|\boldsymbol{\rho_t}) = \begin{cases} \mathcal{N}(\boldsymbol{\theta_{t+1}}|\boldsymbol{\mu_t}, \boldsymbol{\sigma_t^2}) & \text{if } rand_t < c, \\ \theta_t & \text{else.} \end{cases} \tag{5.35}$$

This is equivalent to subdividing the state–action sequence into artificial episodes, which is the standard way of obtaining fitness estimates for evolutionary algorithms in an infinite horizon setting. Given that PGPE does not have individuals or candidate solutions that must have a fixed genome or parameter set over the evaluation time, it seems preferable to perturb the parameters consecutively. The main drawback of using simultaneous perturbations is that every artificial episode starts with a different initial condition, namely the end result of the behaviour of the last parameter set. This introduces large amounts of noise to the evaluation.

During asynchronous perturbation the behaviour changes slightly but continuously. This means that the final reward is much closer to the reward that would be obtained with a fixed parameter set, thereby drastically reducing the noise in evaluation.

Some rearrangement yields the following forms for the derivative of $\log p(\boldsymbol{\theta_u}|\boldsymbol{\rho_u})$ with respect to $\mu_{i,u}$ and $\sigma_{i,u}$:

IHPGPE derivatives

$$\begin{aligned} \nabla_{\mu_{i,u}} \log p(\boldsymbol{\theta_u}|\boldsymbol{\rho_u}) &= \frac{(\theta_{i,u} - \mu_{i,u})}{\sigma_{i,u}^2}, \\ \nabla_{\sigma_{i,u}} \log p(\boldsymbol{\theta_u}|\boldsymbol{\rho_u}) &= \frac{(\theta_{i,u} - \mu_{i,u})^2 - \sigma_{i,u}^2}{\sigma_{i,u}^3}, \end{aligned} \tag{5.36}$$

which can then be substituted into (5.33) to approximate the $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ gradients that gives the IHPGPE update rules.

5.3.1 *Simplified Infinite Horizon PGPE*

We assume that c is sufficiently small that the average perturbation frequency per parameter is comparable with the episode length in an episodic version of the same task. In this case, one can observe that the parameters remain roughly constant until far enough in the future that the contributions of the single summands of Eq. (5.31) are negligible.

Simplification: Ignoring eligibility traces

One can therefore assume that the impact of the eligibility traces is not very large (or to put it another way, that the error caused by ignoring the effect of previous parameter values on the current reward is small, since the parameters are changing slowly).

Furthermore, tracking the whole parameter set over several time steps and calculating the full eligibility trace over these sets is computationally costly.

We therefore suggest a simplified version of IHPGPE that neglects the eligibility traces and assumes that c and γ are chosen in a ratio that the negative effect of doing so can be neglected.

We can then use Eq. (5.29) directly as the expected reward, allowing us to derive the following gradient estimate:

$$\nabla_{\boldsymbol{\rho}} J(\boldsymbol{\rho}) \approx \frac{1}{T} \sum^{T} \nabla_{\boldsymbol{\rho}_t} \log p(\boldsymbol{\theta}_t | \boldsymbol{\rho}_t) r_t. \tag{5.37}$$

IHPGPE gradient estimate

5.3.2 *Sampling with a baseline*

For the same reasons given in Section 5.1.1, we obtain a cheaper gradient estimate by using a single parameter set $\boldsymbol{\theta}$ at time step t and comparing its reward r_t to a baseline reward b. Again intuitively, if $r_t > b$ we adjust $\boldsymbol{\rho}$ so as to increase the probability of $\boldsymbol{\theta}_t$, and $r_t < b$ we do the opposite. Again we use a step size $\alpha_i = \alpha \sigma_i^2$ in the direction of positive gradient (where α is a constant). This gives the following parameter update equations for simplified IHPGPE:

Moving average baseline

$$\begin{aligned} \Delta\mu_i &= \alpha_\mu (r_t - b)(\theta_{i,t} - \mu_{i,t}), \\ \Delta\sigma_i &= \alpha_\sigma (r_t - b) \frac{(\theta_{i,t} - \mu_{i,t})^2 - (\sigma_{i,t})^2}{\sigma_{i,t}}. \end{aligned} \tag{5.38}$$

IHPGPE baseline update rules

5.3.3 *Reward Normalisation*

We can once again make the step size independent of the (possibly unknown) scale of the rewards by introducing a reward normalisation term. Let m_t be the maximum reward the agent can receive at time step t, if this is known, or the maximum reward received so far if it is not. We normalise the $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ updates by dividing them by the difference between m and the baseline b, yielding

Normalising gradient with maximal (observed) reward

$$\begin{aligned} \Delta\mu_i &= \alpha_\mu \frac{r_t - b}{m_t - b} (\theta_{i,t} - \mu_{i,t}) \\ \Delta\sigma_i &= \alpha_\sigma \frac{r_t - b}{m_t - b} \frac{(\theta_{i,t} - \mu_{i,t})^2 - (\sigma_{i,t})^2}{\sigma_{i,t}}. \end{aligned} \tag{5.39}$$

Normalised baseline IHPGPE update rules

Algorithm 8 The simplified IHPGPE algorithm with Reward Normalisation. $\mathbf{T}$ and $\mathbf{S}$ are $P \times N$ matrices with P the number of parameters and N the number of histories. The baseline is updated accordingly after each step. α is the learning rate or step size.

Initialise $\boldsymbol{\mu}$ to $\boldsymbol{\mu}_{\text{init}}$
Initialise $\boldsymbol{\sigma}$ to $\boldsymbol{\sigma}_{\text{init}}$

while TRUE **do**
  for $i = 1$ to number of parameters **do**
    if $rand \sim [0 \ldots 1[\ < c$
    draw perturbation $\epsilon_i \sim \mathcal{N}(0, \sigma_i^2)$
    $\theta_i = \mu_i + \epsilon_i$
  end for
  evaluate $r = r(h(\boldsymbol{\theta}))$
  if $r > m$ then $m := r$

  update $\boldsymbol{\mu} = \boldsymbol{\mu} + \alpha \frac{r-b}{m-b}(\boldsymbol{\theta} - \boldsymbol{\mu})$
  update $\boldsymbol{\sigma} = \boldsymbol{\sigma} + \alpha \frac{r-b}{m-b} \frac{\mathbf{I}[(\boldsymbol{\theta}-\boldsymbol{\mu})^2 - (\boldsymbol{\sigma})^2]}{\boldsymbol{\sigma}}$.
  update baseline b accordingly
end while

6 PGPE PROPERTIES

6.1 RELATIONSHIP TO OTHER ALGORITHMS

In this section we attempt to evaluate the differences between PGPE, SPSA, ES and REINFORCE. Figure 6.1 shows an overview of the relationship of PGPE to the other compared learning methods. Starting with each of the other algorithms, we incrementally alter them so that their behaviour (and performance) becomes closer to that of PGPE. In the case of SPSA we end up with an algorithm identical to PGPE; for the other methods, the transformed algorithm is similar but still inferior to PGPE. For details of the used benchmarks please refer to Chapter 8 of Part "Results and Comparisons".

Going from SPSA, ES, Reinforce to PGPE

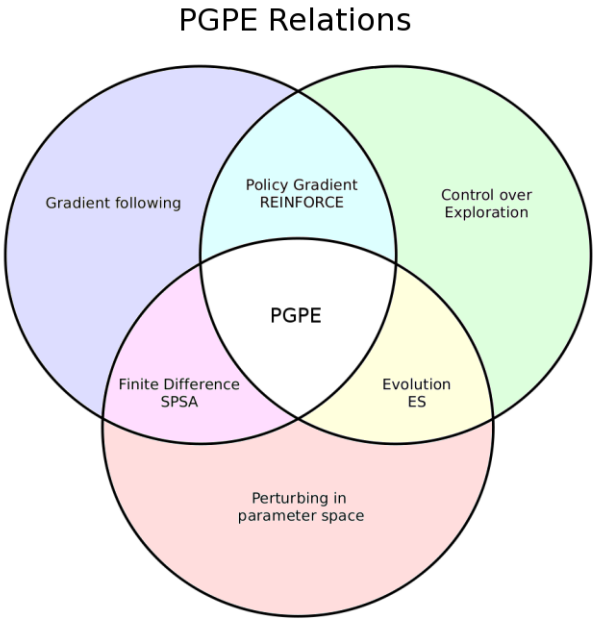


Figure 6.1: Relationship of PGPE to other stochastic optimisation methods.

6.1.1 *From SPSA to PGPE*

Three changes are required to transform SPSA into PGPE. First, the uniform sampling of perturbations is replaced by Gaussian sampling. Second, the finite differences gradient is correspondingly replaced by the likelihood gradient. Third, the variances of the perturbations are turned into free parameters and trained with the rest of the model. Initially the Gaussian sampling is carried out with fixed variance, just as the range of uniform sampling is fixed in SPSA.

Changes needed to go from SPSA to PGPE

Figure 6.2 shows the performance of the three variants of SPSA on the walking task. Note that the final variant is identical to PGPE (solid line). For this task the main improvement results from the switch to Gaussian sampling and the likelihood gradient (circles). Adding adaptive

Impact of changes

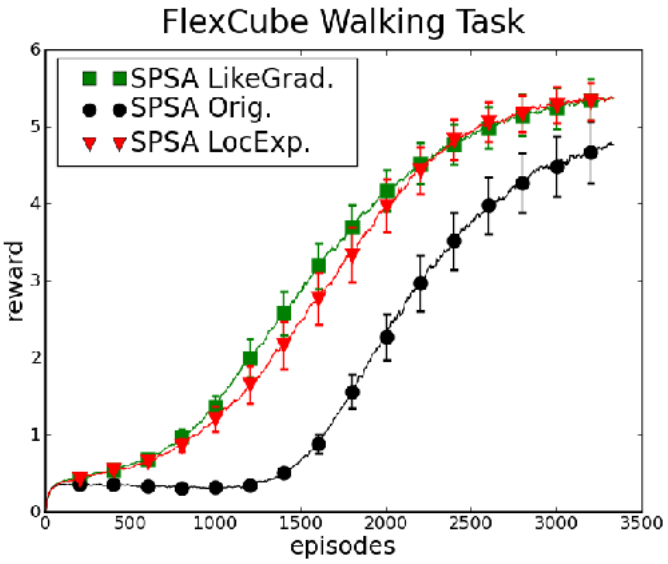


Figure 6.2: Three variants of SPSA on the FlexCube walking task: the original algorithm (SPSA Orig.), the algorithm with normally distributed sampling and likelihood gradient (SPSA LikeGrad.), and with adaptive variance (SPSA LocExp.). All plots show the mean and half standard deviation of 40 runs.

variances actually gives slightly slower learning at first, although both methods converge later on.

The original parameter update rule for SPSA is

$$\theta_{i,t+1} = \theta_{i,t} - \alpha \frac{y_+ - y_-}{2\epsilon_t} \tag{6.1}$$

with $y_+ = r(\theta + \Delta\theta)$ and $y_- = r(\theta - \Delta\theta)$, where $r(\theta)$ is the evaluation function and $\Delta\theta$ is drawn from a Bernoulli distribution scaled by the time dependent step size ϵ_t, i.e. $\Delta\theta_{i,t} = \epsilon_t \, \mathrm{rand}(-1, 1)$. In addition, a set of meta-parameters is used to help SPSA converge. The step size ϵ decays according to $\epsilon_t = \frac{\epsilon_0}{t^\gamma}$ with $0 < \gamma < 1$. Similarly, the step size α decreases over time with $\alpha = a/(t + A)^E$ for some fixed a, A and E [Spall, 1998a]. The choice of initial parameters ϵ_0, γ, a, A and E is critical to the performance of SPSA. In [Spall, 1998b] some guidance is provided for the selection of these coefficients (note that the nomenclature differs from the one used here).

From uniform to Gaussian sampling

To switch from uniform to Gaussian sampling we simply modify the perturbation function to $\Delta\theta_{i,t} = \mathcal{N}(0, \epsilon_t)$. We then follow the derivation of the likelihood gradient outlined in Section 5.1, to obtain the same parameter update rule as used for PGPE (Eq. (5.22)). The correspondence with PGPE becomes exact when we calculate the gradient of the expected reward with respect to the sampling variance, giving the standard deviation update rule of Eq. (5.22).

Including standard deviation update rule

Reduction of meta parameters from 6 to 3 for going from SPSA to PGPE

As well as improved performance, the above modifications greatly reduce the number of hand-tuned meta-parameters in the algorithm, leaving only the following: a step size α_μ for updating the parameters, a step size α_σ for updating the standard deviations of the perturbations and an initial standard deviation σ_{init}. We found that the parameters $\alpha_\mu = 0.2$, $\alpha_\sigma = 0.1$ and $\sigma_{\text{init}} = 2.0$ worked very well for a wide variety of tasks.

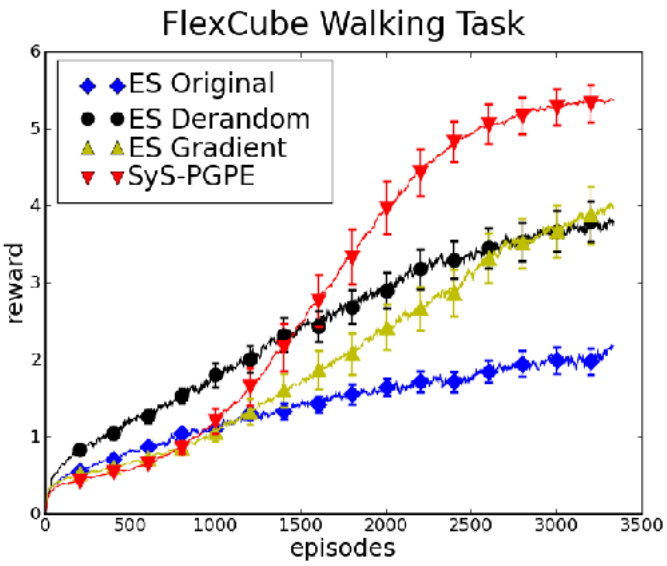


Figure 6.3: Three variants of ES applied on the FlexCube walking task: the original algorithm (ES Original), derandomised ES (ES Derandom) and gradient following (ES Gradient). PGPE is shown for reference. All plots show the mean and half standard deviation of 40 runs.

6.1.2 *From ES to PGPE*

Changes needed to go from ES to PGPE

We now examine the effect of two modifications that bring ES closer to PGPE. First, we switch from standard ES to derandomised ES [Hansen and Ostermeier, 2001], which somewhat resembles the gradient-based variance updates found in PGPE. Then we change from population-based search to following a likelihood gradient. The results are plotted in Figure 6.3.

Impact of changes

As can be seen, both modifications bring significant improvements, although neither can match PGPE. While ES performs well initially, it is slow to converge to good optima. Partly this is because, as well as having stochastic mutations, ES has stochastic updates to the standard deviations of the mutations and the coupling of these terms slows down convergence. Derandomised ES addresses that problem by using a deterministic standard deviation update rule instead, based on the change in parameters between the parent and child. Population-based search has advantages in the early phase of search, when broad, relatively undirected exploration is desirable. This is particularly true for the multimodal fitness spaces typical of realistic control tasks. However in later phases convergence is usually more efficient with gradient based methods. Applying the likelihood gradient, the ES parameter update rule becomes

$$\Delta\theta_i = \alpha \sum_{m=1}^{M} (r_m - b)(y_{m,i} - \theta_i), \tag{6.2}$$

where M is the number of samples and $y_{m,i}$ is parameter i of sample m.

6.1.3 *From REINFORCE to PGPE*

We previously asserted that the lower variance of PGPE's gradient estimates is partly due to the fact that PGPE requires only one parameter sample per history, whereas REINFORCE requires samples every

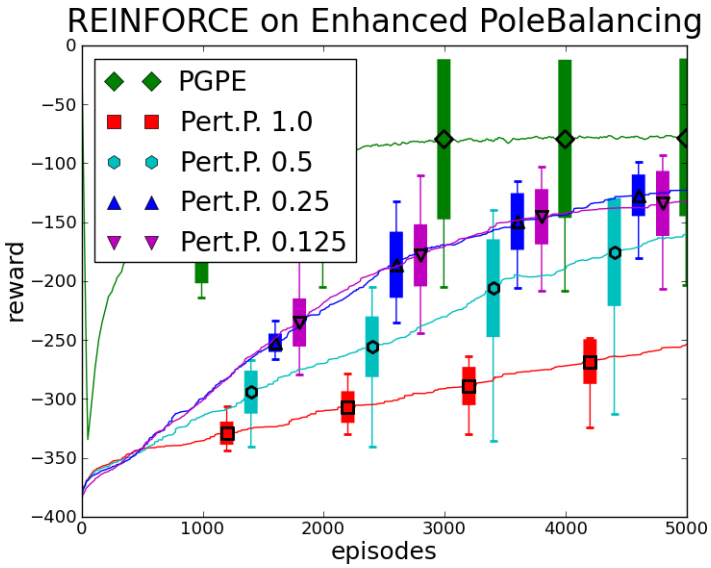


Figure 6.4: REINFORCE applied on the pole balancing task, with various action perturbation probabilities (1, 0.5, 0.25, 0.125). PGPE is shown for reference. All plots show the mean and half standard deviation of 40 runs.

Do less frequent perturbations improve REINFORCE?

time step. This suggests that reducing the frequency of REINFORCE perturbations should improve its gradient estimates, thereby bringing it closer to PGPE.

Fig. 6.4 shows the performance of episodic REINFORCE with a perturbation probability of 1, 0.5, 0.25, and 0.125 per time step. In general, performance improved with decreasing perturbation probability. However the difference between 0.25 and 0.125 is negligible. This is because reducing the number of perturbations constrains the range of exploration at the same time as it reduces the variance of the gradient, leading to a saturation point beyond which performance does not increase.

Reducing policy perturbation frequency constrains exploration

Note that the above trade off does not exist for PGPE, because a single perturbation of the parameters can lead to a large change in the overall behaviour.

State Dependent Exploration as alternative

A related approach is taken in [Rückstieß et al., 2008] where the exploratory noise in each time step is replaced by a state-dependent exploratory function (SDE) with additional parameters. Rather than generating random noise in each time step, the parameters of this exploration function are drawn at the beginning of each episode and kept constant thereafter, which leads to smooth trajectories and reduced variance. SDE allows to use any gradient estimation technique like Natural Actor-Critic or the classical REINFORCE algorithm by Williams, while still ensuring smooth trajectories, a property that PGPE naturally has.

6.2 CENTRAL SEARCH PROPERTY

An important difference between PGPE and MultiPGPE is that PGPE uses a single normal distribution that is centred at the origin of the search space while MultiPGPE uses a uniform distribution over a certain search interval.

PGPE searches centred around the origin of the search space

While assuming in PGPE the origin of the search space as good starting point and global search can be substituted in growing the standard deviation for exploration to a big enough amount is in a lot of cases a very good and beneficial one, it can also be an disadvantage. We will

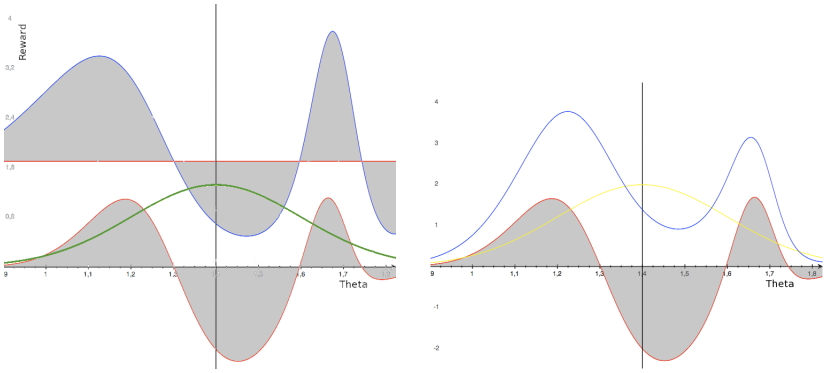


Figure 6.5: In the left figure an example reward function is shown (upper curve, blue) with the baseline (red) that is to be expected by using the normal distribution over the parameter θ (bold, green). The expected reward frequency can be seen for the baselined reward (lower curve, red). The reward integral is clearly bigger for the left part of the centre between the two optima (black vertical line). This gives an example why PGPE tends to be more attracted by broad optima (in relation to the current search distribution). Broad optima produce more *better than baseline rewards* than a narrow optima does.

see in section 8.1 that the use of an arbitrary distribution is beneficial in at least two cases: (i) If there is a global structure in the search space that leads the way to the global optima that is not near the origin of the search space. (ii) If the search space is truly multi modal.

6.3 FLAT MINIMA PROPERTY

PGPE was originally designed for robot control. In this context it is vitally important to find a solution that is robust to sensor noise and environmental perturbations. A very general way to improve robustness in parametric models is to search for a *flat* optimum in parameter space [Hochreiter and Schmidhuber, 1997] where the model remains in a local optimum even if its parameters are slightly modified. PGPE shows a strong tendency to favour flat optima over optima having a very narrow attractor. The reason is that the weight perturbations used by PGPE to determine the gradient approximate an integral over a small region of weight space. This automatically favours flat optima, since the expected reward in broad flat regions is higher than in deep narrow ones. See Fig. 6.5 for an example.

Flat *optima are more robust against sensor noise*

PGPE prefers flat optima

However, in some situations (such as the *balancing* solutions described in Section 8.1.3) deep narrow optima should not be overlooked. In this context PGPE suffers because there is no way to control its preference for flat minima. With MPGPE on the other hand, increasing the number of Gaussians allows the algorithm to focus on smaller regions on parameter space and therefore to find narrower and deeper optima.

User has no control over flat optima property in PGPE

In Section 8.1.1 PGPE was unable to find the global optima because of the very broad local optimum at the edge of parameter space. It could only escape the local optimum by greatly increasing the variances of its single Gaussians per parameter. However, with such a large variance, it would be very unlikely to collect enough samples from the attractors of

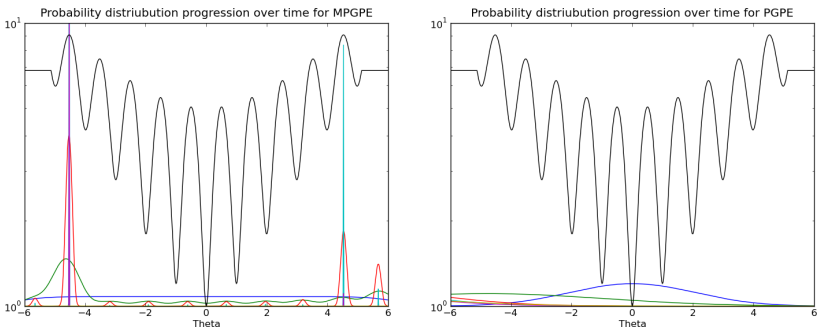


Figure 6.6: The figures show an example of MultiPGPE (left) succeeding PGPE (right) failing to find a global maximum in a multimodal landscape. In both cases the different coloured lines show successive stages of optimisation, in the order blue, green, red, aqua and purple. MultiPGPE adjusts the means of its components to match the locations of the multiple optima, then homes in on the two global optima by increasing the corresponding mixing coefficients and reducing the variances, before eventually choosing one of them. PGPE, on the other hand, can only move its single mean left or right. This results in a much cruder search through parameter space, guided by the average gradient leading up to both sides, rather than the precise values at the individual peaks. This coarse-grained search misses the global optimum and finds the broad local optimum at the extreme left instead.

the global optima to motivate it to move towards them. For MPGPE, using 10 Gaussians per mixture is sufficient to overcome this effect.

6.4 OVERESTIMATION

PGPE follows the global structure in the search space — sometimes too fast

PGPE substitutes a global search by growing each search standard deviation to a suitable amount. Therefore PGPE could follow too fast the direction given by the global structure in the search space, thereby "overlooking" the global optimum. This drawback is illustrated in section 8.1.3. While this problem can be handled by adjusting (decreasing) the two step size parameters, it still conflicts with the convergence speed.

MultiPGPE however is doing a global search and therefore adapting an arbitrary search distribution to the gathered experience or reward. It is obvious that an "overlooking" of the global optima is much less likely for MultiPGPE because there is no fast directed adaptation of the parameter set.

However, this global search comes at a price in terms of convergence speed. This results in the fact that MultiPGPE is slower in convergence where this global search is not needed, because the assumptions of PGPE hold.

Also, how much Gaussians per mixture are optimal for a given problem is a completely open question and very hard to estimate. We will use in Part III "Results and Comparisons" 10 Gaussians per Mixture. This was a good number to counter the overestimation problem, but maybe in some examples this rather high number of Gaussians weakens the flat optimum property too much and results in finding extreme, unstable solutions.

As we show in section 8.1 the slowdown is not large though. The convergence speed up in cases where the PGPE assumptions do not hold combined with the increase in the likelihood to find better solutions in addition with the possibility to have control over the flat minimum property should be worth the slowdown in convergence if the PGPE assumptions hold.

On the other hand the PGPE algorithm is much simpler and (in cases where the reward evaluation is not computational expensive) it is much faster to compute.

7

PART SUMMARY AND CONCLUSION

In Chapter 5, we derived the PGPE algorithm and its variants MultiPGPE and IHPGPE from the general framework of episodic reinforcement learning in a Markovian environment. In doing so, we highlighted the differences between PGPE and standard policy gradient methods such as REINFORCE. We showed how to use a baseline approach and how to use symmetric sampling, if possible, for all variants of PGPE. We also provided a Reward Normalisation scheme and simplifications of the algorithms for every PGPE variant where such are useful. An algorithm in pseudo code was provided for the best instance of every PGPE variant.

MultiPGPE was designed to cope with one of the problems of PGPE, namely that a normal distributed search is sometimes not sufficient in highly multimodal search spaces. It uses a mixture of Gaussians to counter this problem.

IHPGPE lifts PGPE's restriction to episodic tasks, by using asynchronous parameter perturbations over time and thus generating gradients continuously.

In Chapter 6, we evaluated the differences between PGPE, SPSA, ES, as well as REINFORCE, and discussed the properties of PGPE.

To transform SPSA into PGPE, we identified three changes that are required. First, the uniform sampling of perturbations is changed into a Gaussian sampling, with the effect of the method being less likely to get stuck in a local optima, and correlated parameters are easier to track. Second, the finite difference gradient is replaced correspondingly by the likelihood gradient, yielding a speed-up in convergence time. Third, the variances of the perturbations are turned into free parameters and trained with the rest of the model, thus enabling to learn effectively in problems with parameters that are very different in their order of magnitude.

We also examined the effect of two modifications that bring ES closer to PGPE. First, we switch from standard ES to derandomised ES, resembling the gradient based variance updates found in PGPE. Then we change from population based search to following a likelihood gradient, resulting in a speed-up in convergence time.

Another experiment was to change the exploration in REINFORCE by transferring the explorational noise to the parameters of the controller. This resulted in a speed-up in convergence time that couldn't be explained by the less frequent perturbations alone.

We discussed the use of SDE as an alternative to PGPE. With SDE one can use standard PG methods with all the developed improvements of the last years by still keeping the exploration parameter based.

We stated that the central search property of PGPE can be a drawback, if several good optima lie in opposite directions. MultiPGPE can overcome this problem.

The flat optima property of PGPE was discussed, where we identified a possible drawback. There is no control over the extend to which the algorithm tends to converge to flat shallow optima instead of going for narrow deep ones. We showed that MultiPGPE can overcome this issue by using different numbers of Gaussians per mixture.

We also examined if PGPE tends to overestimate the gradients' reach if the global error surface constantly slopes in a certain direction, eventually moving too fast in parameter space and overlooking the global optimum. It is open for discussion to which extent this problem is mainly an artificial one.

Overall, PGPE and its variants are mathematically soundly based on the principles of general Reinforcement Learning (in contrast to ES), more flexible than Finite Difference methods and provide a less noisy gradient compared to Policy Gradient methods. The idea is that PGPE should conserve the robustness of ES, the simplicity in use of SPSA and the flexibility of PG. The empirical data in PART III will tell us whether these assumptions hold.

Part III

RESULTS AND COMPARISONS

8 ROBOTIC BENCHMARKS

In the last chapters we introduced and derived the class of Parameter Exploring Policy Gradients (PEPG) and discussed their properties. A Machine Learning (ML) method that claims to be a general usable Reinforcement Learning (RL) method has also to be tested on real life problems and standard benchmarks.

This chapter shows how effective PEPGs are

In this chapter we show that PEPGs are indeed very effective, especially in high-dimensional robotic RL tasks. The main advantage of PGPE in complex robotic RL tasks is that it does not have the same credit assignment problem than standard PG methods have. By contrast, a standard policy gradient method must first determine the reward gradient with respect to the policy, then differentiate the parameters with respect to that reward gradient resulting in two drawbacks. First, it assumes that the controller is always differentiable with respect to its parameters, which our approach does not. Second, it makes optimisation more difficult since very different parameter settings can determine very similar policies and vice-versa.

We compare PGPE and its variants to several state of the art RL methods

In the following we use benchmark scenarios with increasing complexity to show the properties of PGPE and its dis-/advantages compared to other applicable fields in detail. These scenarios allow us to model problems of similar complexity to today's real-life RL problems [Müller et al., 2007; Peters and Schaal, 2006]. For some experiments we also compare the performance of PGPE with and without symmetric sampling (SyS). In sections below we test PGPE therefore on several control experiments and compare its performance with REINFORCE [Williams, 1992], Evolution Strategies (ES, [Schwefel, 1993]), Simultaneous Perturbation Stochastic Adaptation (SPSA, [Spall, 1998a]) and episodic Natural Actor Critic (eNAC, [Peters and Schaal, 2008b,a]).

We compare by reward against episodes (common ground for all RL methods)

For all experiments we plot the agent's reward against the number of training *episodes*. An episode is a sequence of T interactions of the agent with the environment where T is fixed for each experiment during which the agent makes one attempt to complete the task. For all methods, the agent and the environment are reset at the beginning of every episode.

For eNAC and REINFORCE we employed an improved algorithm that perturbs the actions at randomly sampled time steps instead of perturbing at every time step.

We compare mainly to ES with local mutation operator

For all the ES experiments we used a local mutation operator. In most cases we did not examine correlated mutation and covariance matrix adaptation-ES because both mutation operators add $n(n-1)$ strategy parameters to the genome; given the more than 1000 parameters for the largest controller, this approach would lead to a prohibitive memory and computation requirement. In addition, the local mutation operator is more similar to the perturbations in PGPE, making it easier to compare the algorithms.

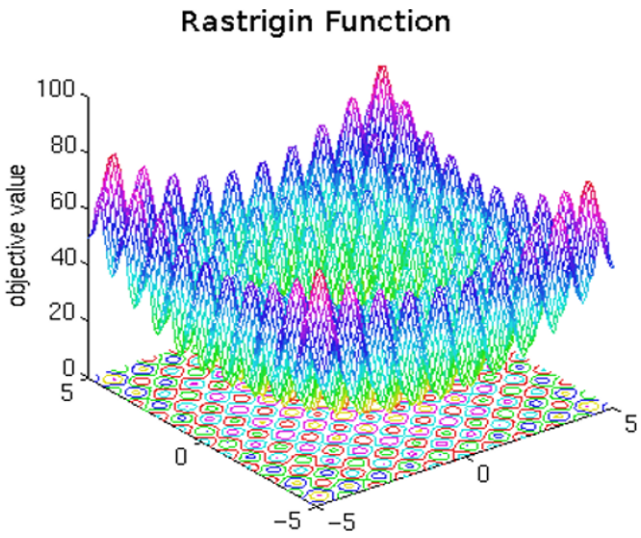


Figure 8.1: The 2D Rastrigin function for $\vec{x} \in (-5.12, 5.12)$. Note the four global maxima near the corners and the single global minimum in the centre.

All plots are showing the average results of 40 independent runs if not stated otherwise. All the experiments were conducted with hand-optimised meta-parameters, including the perturbation probabilities for eNAC and REINFORCE. For PGPE we used the meta-parameters $\alpha_\mu = 0.2, \alpha_\sigma = 0.1$ and $\sigma_{\text{init}} = 2.0$ in all tasks.

8.1 STANDARD BENCHMARKS

In this section we use standard benchmarks to test the details of the PGPE variants. The standard benchmarks are computationally less complex than sophisticated robot simulations and can highlight distinct features of the PGPE variants.

We compare the PGPE variants with the Rastrigin, Ackley and Inverted Pendulum benchmarks

We used the Rastrigin and Ackley functions as well as the inverted pendulum benchmark for demonstrating the differences in learning behaviour between MultiPGPE and PGPE (we also used ES for comparison). We used 10 Gaussians per mixture for MultiPGPE and its variants in all experiments.

Ship Steering is used to evaluate SyS

The ship steering benchmark was used to investigate the differences between PGPE with and without Symmetric Sampling (SyS).

Compariing to eNAC and REINFORCE with enhanced Pole Balancing benchmark

We used the enhanced pole balancing benchmark for comparison with eNAC and REINFORCE. This standard benchmark was already used by [Riedmiller et al., 2007b] for evaluating eNAC.

We will use the following abbreviations throughout to distinguish the PGPE algorithm variants:

- PGPE: standard PGPE (Sec. 5.1) with baseline sampling (Sec. 5.1.1)
- SyS-PGPE: PGPE (Sec. 5.1) with symmetric sampling (Sec. 5.1.2)
- MPGPE: standard MultiPGPE (Sec. 5.2) with baseline sampling (Sec. 5.2.2)
- SyS-MPGPE: simplified MultiPGPE (Sec. 5.2.1) with symmetric sampling (Sec. 5.2.3)

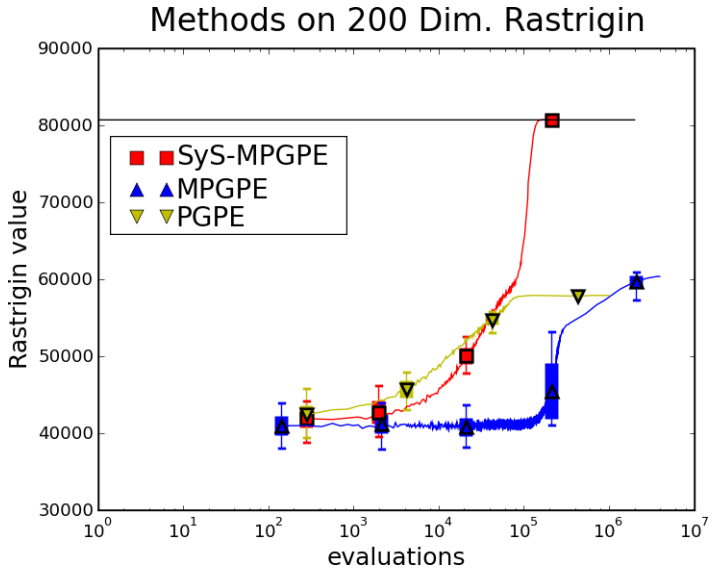


Figure 8.2: PGPE, MPGPE and SyS-MPGPE maximising the 200 dimensional Rastrigin function. All plots show the mean and standard deviation of 40 runs.

8.1.1 *Rastrigin Function*

As is standard practice for the Rastrigin benchmark, our experiments restrict all the x_i values to the interval $[-5.12, 5.12]$. This ensures that the global maxima lie close to, but not at, the extremal values of the domain.

Our first experiments compared the ability of PGPE, MPGPE and SyS-MPGPE to maximise the 200 dimensional Rastrigin function. For this task the meta parameters were $\alpha_\mu = \alpha_\sigma = 0.0125$ and $\alpha_\pi = 0.00625$ for SyS-MPGPE and PGPE and $\alpha_\mu = \alpha_\sigma = 0.025$ and $\alpha_\pi = 0.0125$ for MPGPE. The results, presented in figure 8.2, show that PGPE converges to a suboptimal solution. As we will see, its failure stems from its use of unimodal parameter distributions, which are confused by the 2^n equal

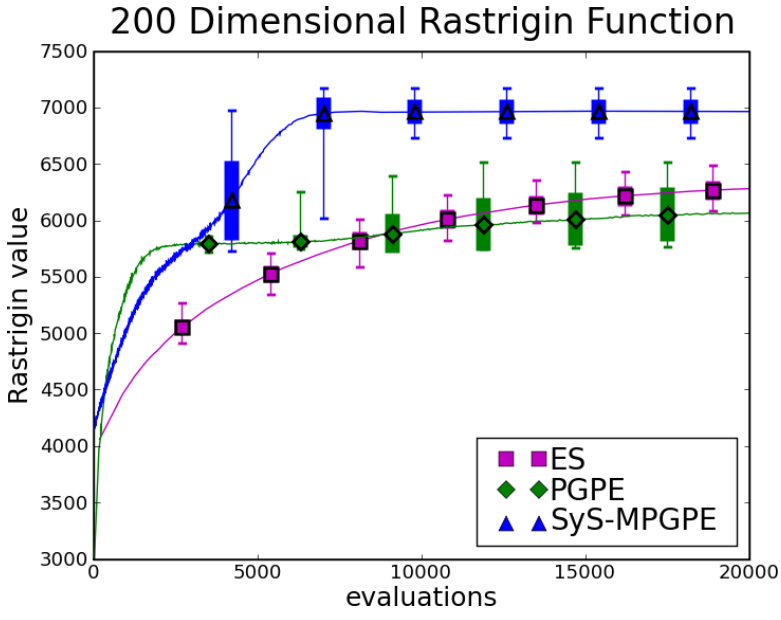


Figure 8.3: SyS-MPGPE, PGPE and ES minimising (left) and maximising (right) the Rastrigin function. All plots show the mean and standard deviation of 100 runs.

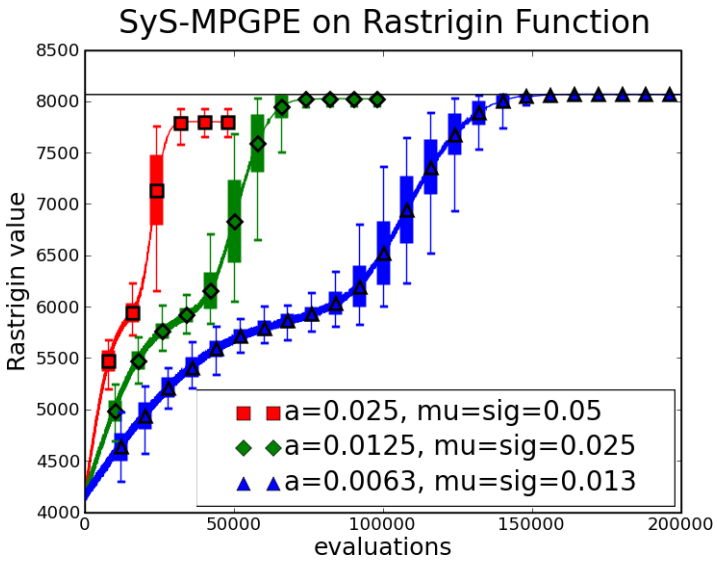


Figure 8.4: SyS-MPGPE maximising the 200 dimensional Rastrigin function with different meta-parameters. All plots show the mean and half standard deviation of 100 runs.

but widely separated global optima. This results in an overextension of the parameter variances during the search process.

MPGPE also fails to find a global optimum in the timeframe we set for computational reasons. In particular it is very slow at the initial stages of adaptation. The switch to SyS-MPGPE brings a dramatic speedup, reaching a global optimum in about $1.5 \cdot 10^5$ evaluations.

Our second set of experiments compared SyS-MPGPE, PGPE and ES at both maximising and minimising the 200 dimensional Rastrigin function with the standard meta parameters. The PGPE and SyS-MPGPE meta parameters were therefore $\alpha_\mu = \alpha_\sigma = 0.2$ and $\alpha_\pi = 0.1$. For ES we hand-optimized the meta parameters to give the best performance possible within the time-frame set by the PGPE experiments; in particular we chose the best population size found in a thorough search in the range from (6, 36) to (600, 3600). As can be seen from Figure 8.3, the advantage of SyS-MPGPE over PGPE in the maximisation task vanishes in the minimisation task.

At first glance this seems surprising, since one would expect the multimodal surface of the Rastrigin function to favour MPGPE for both kinds of optimisation. However, while the Rastrigin function has 2^n global maxima, it has only one global minimum. Therefore PGPE no longer needs to overextend the variance in its parameter distributions to successfully optimise; all it needs is enough variance to escape the local minima—just like MPGPE. Although ES slightly outperforms PGPE at maximisation, SyS-MPGPE is clearly better than ES at both tasks.

Figure 8.4 shows how the performance of SyS-MPGPE varies with different meta parameters. While its ability to find an optimum appears robust to parameter choice, the time taken to converge can vary substantially.

Figure 8.5 shows how the number of evaluations required for convergence grows with the dimension of the problem for SyS-MPGPE. The meta parameters were halved for every factor of 10 by which the dimensionality was increased (from $\alpha_\mu = \alpha_\sigma = 0.025$ and $\alpha_\pi = 0.0125$ for 20 dimensions to $\alpha_\mu = \alpha_\sigma = 0.00625$ and $\alpha_\pi = 0.003125$ for 2000

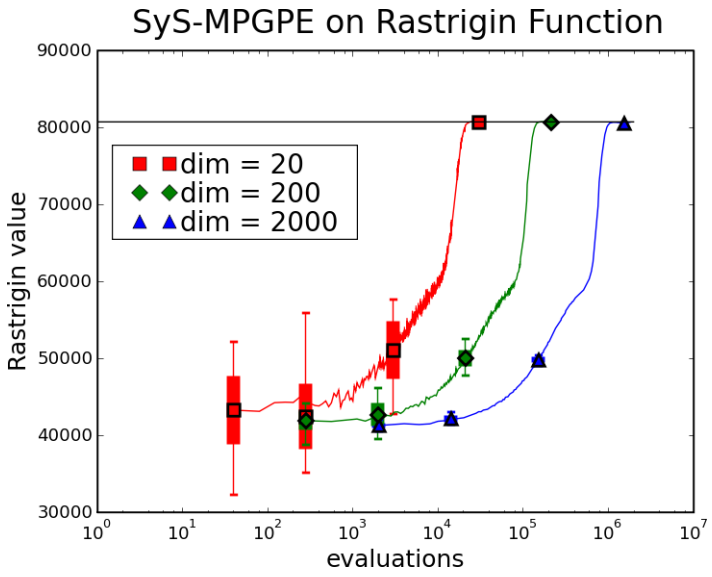


Figure 8.5: SyS-MPGPE maximising the n dimensional Rastrigin function for different values of n. All plots show the mean and half standard deviation of 40 runs.

dimensions). Note that training time grows sub-linearly with the number of dimensions (increasing the number of dimensions by a factor of 10 increases the number of required evaluations by a factor about 6.5). Given that the number of local optima in the Rastrigin function grows exponentially with the number of dimensions, this bodes well for the scalability of SyS-MPGPE to high-dimensional parameter spaces. The n-dimensional Rastrigin function, illustrated in Fig 8.1, is a well-known optimisation benchmark with one global minimum, 2^n global maxima, and a exponential growing number of local optima. It is defined as follows:

$$\mathrm{Ras}(\vec{x}) = 10n + 10 \cdot \sum_{i=1}^{n} \cos(2\pi x_i) - \sum_{i=1}^{n} x_i^2. \tag{8.1}$$

Def.: Rastrigin function

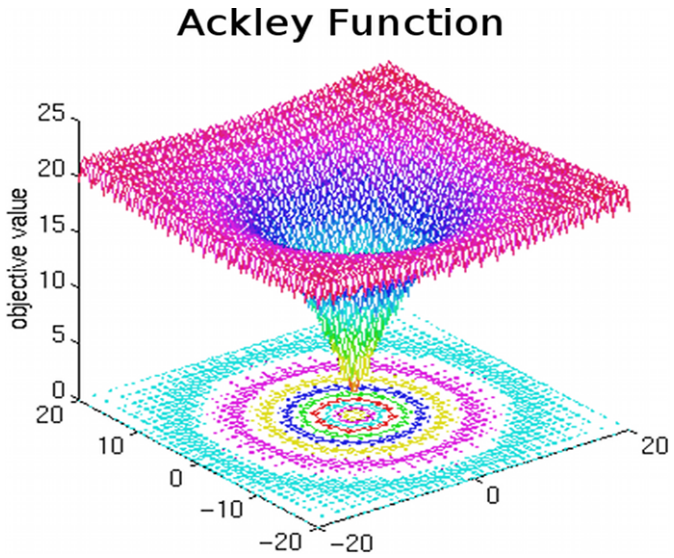


Figure 8.6: The 2D Ackley function for $\vec{x} \in -20 - -20 \times -20 - -20$. Note the huge amount of local minima and the sharp global minimum.

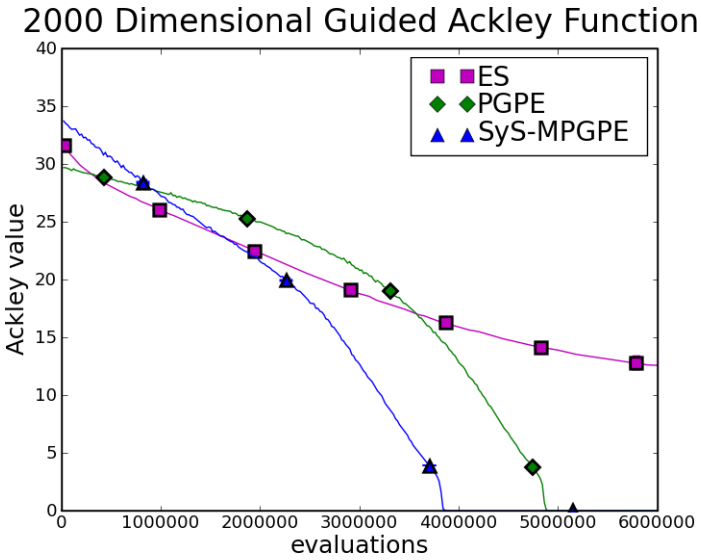


Figure 8.7: SyS-MPGPE, PGPE and ES minimising the Ackley (left) and guided Ackley (right) functions. All plots show the mean and standard deviation of 40 runs.

8.1.2 *Ackley Function*

The Ackley function, illustrated in Fig. 8.6, is another well-known optimisation benchmark. In this case the challenge is that the attractor basin for the single global minimum gets narrower as the number of dimensions increases. At very high dimensions, finding the minimum becomes the proverbial search for a "needle in a haystack". Like the Ras-

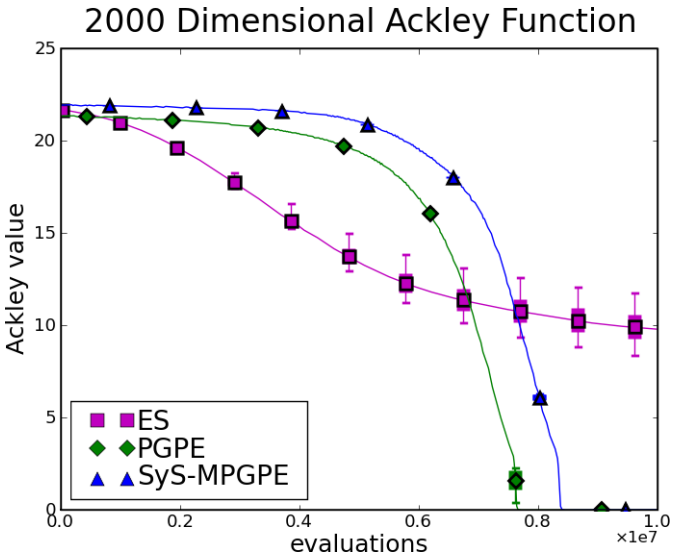


Figure 8.8: SyS-MPGPE, PGPE and ES minimising the Ackley functions. All plots show the mean and standard deviation of 40 runs.

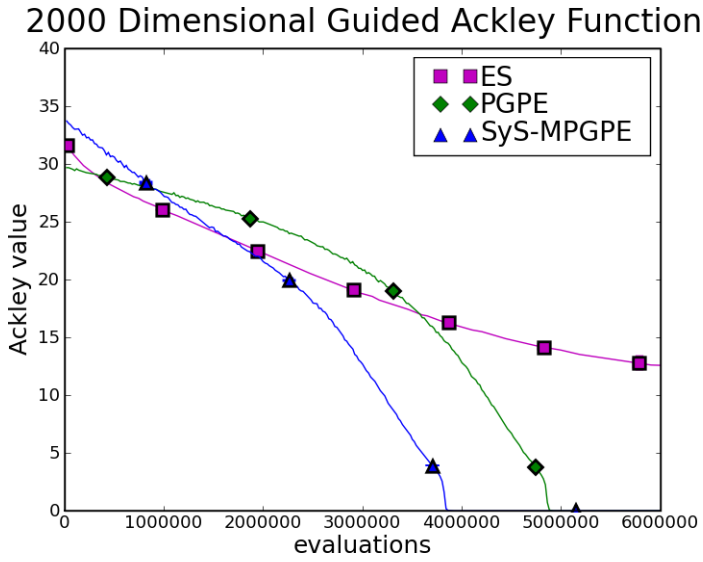


Figure 8.9: SyS-MPGPE, PGPE and ES minimising the guided Ackley function. All plots show the mean and standard deviation of 40 runs.

trigin function it also has many local minima to deceive the optimiser. It is defined as follows:

$$f_1(\vec{x}) = \exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} {x_i}^2}\right)$$
$$f_2(\vec{x}) = \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right)$$
$$\mathrm{Ack}(\vec{x}) = 10n + e - 10n f_1(\vec{x}) - f_2(\vec{x}). \tag{8.2}$$

To accentuate the property of MultiPGPE we want to demonstrate here, namely that global structure enables MultiPGPE to aim much faster for the optimal region of the search space, we consider a variant $g(\vec{x})$ of the Ackley function, in which an additional term is added to Eq. (8.2) as follows

$$g(\vec{x}) = \mathrm{Ack}(\vec{x}) + \frac{1}{n}\sum_{i=1}^{n}|\vec{x_i}|. \tag{8.3}$$

Def.: Guided Ackley function

We refer to this function as the *guided Ackley* function, because the extra term guides the optimiser towards the global minimum by adding a global gradient. We restricted the x_i values to the interval $[-32, 32]$, as is standard for the Ackley benchmark.

We evaluated SyS-MPGPE, PGPE and ES on the 2000 dimensional guided Ackley function. For SyS-MPGPE and PGPE the standard meta parameters of $\alpha_\mu = \alpha_\sigma = 0.2$ and $\alpha_\pi = 1.0$ were used. For ES we hand-optimised the meta parameters as before, this time giving a population size of (2000,12000), because the standard population size was to small to give good results. The results are presented in Fig. 8.9. As can be seen in this figure, MultiPGPE converges more quickly by a wider margin than PGPE. This is because the global structure we added (Eq. (8.3)) helps MultiPGPE to decide early for the "right" search region, and therefore for the right Gaussian in the mixture. Here MultiPGPE searches directly with much less exploration than PGPE does (one Gaussian of the mixture has a smaller starting σ than the broad single

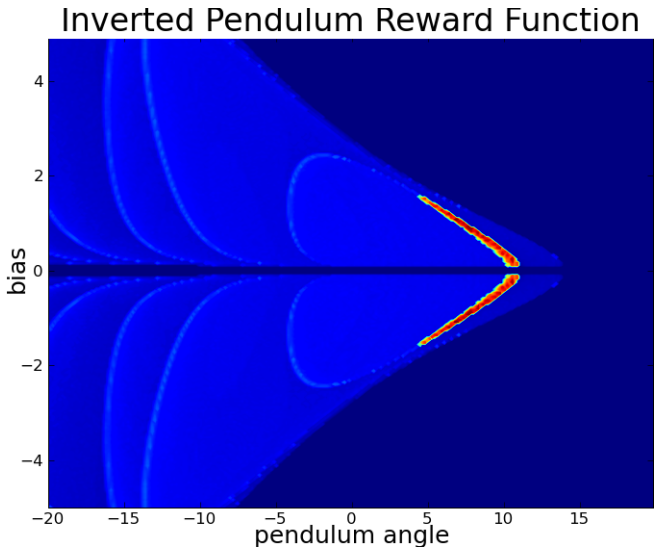


Figure 8.10: A 2D slice of the reward space for the inverted pendulum task. The plot shows how the reward varies with the pendulum angle and the bias weight while the angular velocity is fixed at 25. Note the sharp global optimum and the smooth gradients leading towards the multiple local optima (which correspond to spinning the pole).

Gaussian of PGPE). This effect is increased if the global optimum lies far away from the centre of the search space, because PGPE needs to increase the exploration to reach the optimal region and then needs to shrink the exploration again to converge on the optimum.

8.1.3 *Inverted Pendulum*

The Inverted Pendulum is another well-known benchmark scenario. The task is to swing a pendulum to an upright position, and balance it there, using a motor attached to its axis. The motor is not strong enough to move the pendulum up directly, so that the only way to solve the task is to swing back and forth several times to gain momentum. The agent receives a reward only while the pendulum is almost completely vertical (meaning that the absolute value of the joint angle is less than 0.14

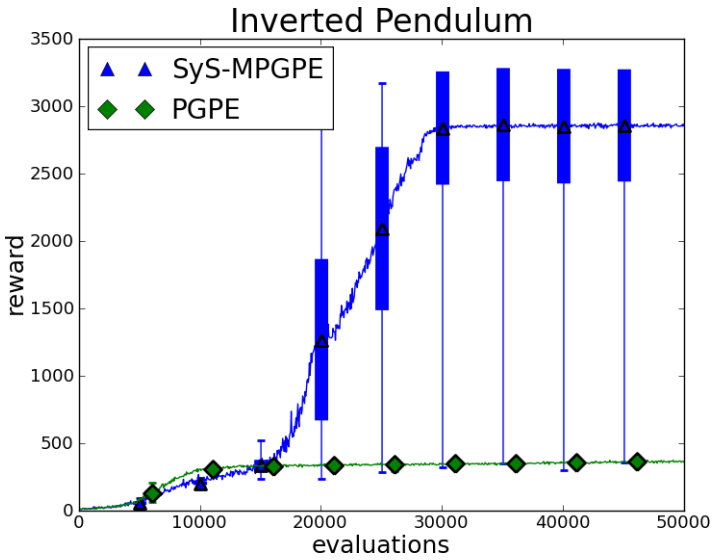


Figure 8.11: SyS-MPGPE and PGPE on the Inverted Pendulum. All plots show the mean and standard deviation of 10 runs.

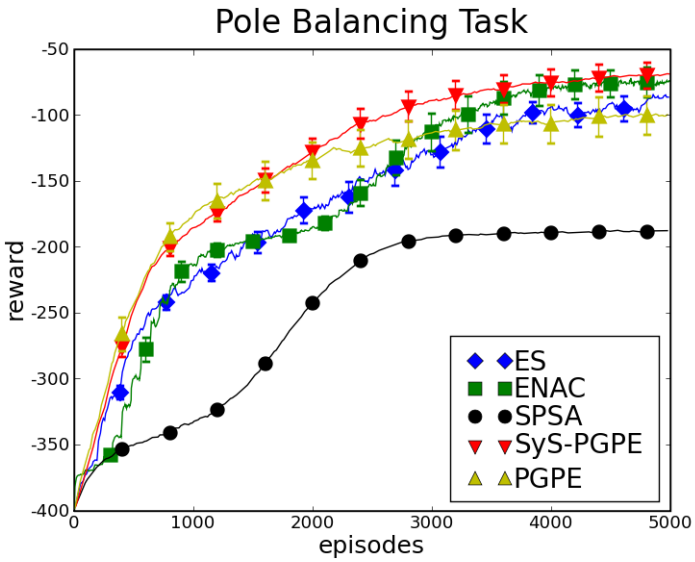


Figure 8.12: PGPE with and without SyS compared to ES, SPSA and eNAC on the pole balancing benchmark. All plots show the mean and half standard deviation of 40 runs.

Radians). The input variables to the agent are a bias signal, the angle of the pendulum and the angular velocity of the pendulum. The starting search interval was $[-20, 20]$ for all parameters in our experiments, but they were allowed to exceed these limits during learning. We used the ODE physics engine [ODE, 2010] to simulate the motion of the pendulum. Note that we used a much harder version of this benchmark. In the standard task the pendulum needs only one extra swing to gather enough momentum to finally swing up. Here the motor is under-powered such that at least 3 extra swings are needed to gain enough momentum.

The enhanced Inverted Pendulum benchmark

The main difficulty of the task is the gradient of most of the reward space leads to suboptimal solutions, as illustrated in Fig. 8.10. These solutions correspond to parameters that will spin the pendulum around the axis, thereby collecting a reward only during the brief period when the pendulum passes through the upright position. To put it simply, most RL methods end up spinning rather than balancing the pendulum.

We compared SyS-MPGPE and PGPE on this benchmark with the meta parameters $\alpha_\mu = 0.05, \alpha_\sigma = \alpha_\pi = 0.025$. As can be seen from figure 8.11, PGPE converges to a much lower reward (the spinning behaviour) than SyS-MPGPE. In fact, SyS-MPGPE only failed once in ten trials to find the global optimum. As with the Rastrigin function, PGPE is confused by the multi-modal reward surface and the sharp global optimum. SyS-MPGPE, on the other hand, uses its multi-modal distributions to track both regions of the reward space simultaneous and eventually choose the better one.

8.1.4 *Enhanced Pole Balancing*

The next scenario is the extended pole balancing benchmark as described in [Riedmiller et al., 2007a]. In contrast to [Riedmiller et al., 2007a] however, we do not initialise the controller with a previously chosen stabilising policy but rather start with random policies. In this task the agent's goal is to maximise the length of time a movable cart

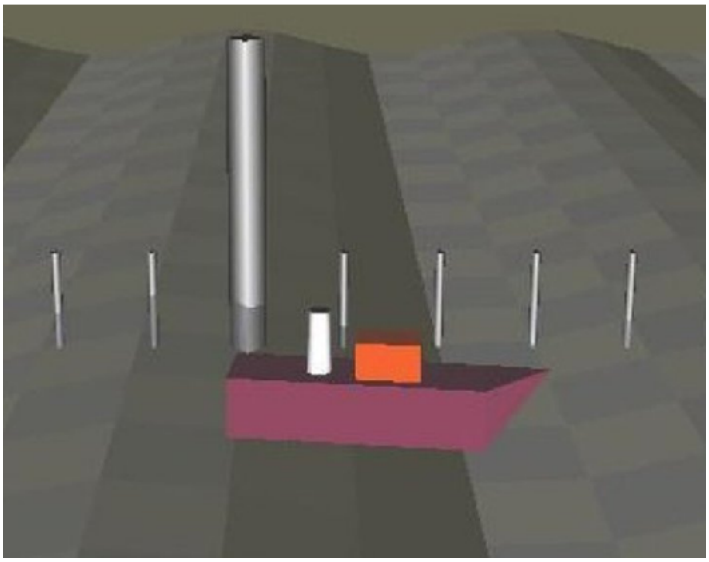


Figure 8.13: The ship steering simulation. The colour of the "cargo container" corresponds to the agent's reward. The colour changes continuously from green for maximal reward to red for minimal reward. The scale in the background shows the distance the ship has travelled.

can balance a pole upright in the centre of a track. The agent's inputs are the angle and angular velocity of the pole and the position and velocity of the cart. The agent is represented by a linear controller with four inputs and one output unit. The simulation is updated 50 times a second. The initial position of the cart and angle of the pole are chosen randomly.

Figure 8.12 shows the performance of the various methods on the pole balancing task. All algorithms quickly learned to balance the pole, and all eventually learned to do so in the centre of the track. PGPE with SyS was both the fastest to learn and the most effective algorithm on this benchmark. In this benchmark the advantage of PGPE over SPSA is very clear. From the pure "balancing" optimum the "canyon" to the "balance in the middle of the track" optimum is narrow and highly correlated in the problem dimensions. The probability to find the way from one

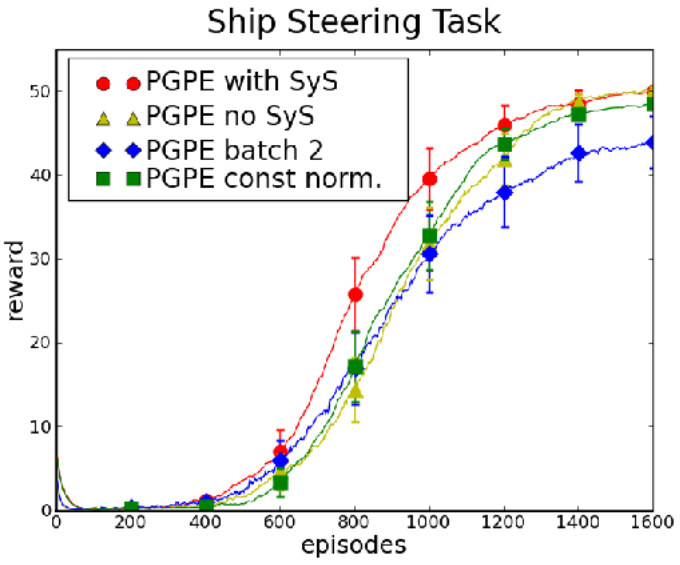


Figure 8.14: PGPE with SyS, without SyS, with two samples batch size and with classical reward normalisation. All plots show the mean and half standard deviation of 40 runs.

#	markers	SyS	reward normalisation	# samples
1	red circle	yes	yes	2
2	green triangle	no	yes	1
3	blue diamond	no	yes	2
4	yellow square	yes	no	2

Table 1: Overview of algorithms used in the ship steering task. The results are plotted in Fig. 8.14 with respective marker shapes and colours.

optima to the other is very unlikely with uniform sampling—thus SPSA gets stuck in the "balancing only" optima.

8.1.5 *Ship Steering*

In this task an ocean-going ship with substantial inertia in both forward motion and rotation (plus noise resembling the impact of the waves) is simulated. The task in this scenario was to keep the ship on course while keeping maximal speed. While staying on a predefined course with an error less than 5 degrees, the reward is equal to the ship's speed. Otherwise the agent receives a reward of zero. The framework has 2 degrees of freedom and a 3-dimensional observation vector (velocity, angular velocity, error angle). The agent is represented by a linear controller with 3 inputs and 2 output units. The simulation is updated every 4 seconds. The initial angle of the ship is chosen randomly.

In this experiment we only compare different versions of our algorithm with each other. Table 1 gives an overview of the different properties used. As can be seen from the results in Fig. 8.14, the task could be solved easily with PGPE. PGPE with SyS (version 1) is faster in convergence speed than its non-symmetric counterpart (version 2). However, both versions 1 and 2 reach the same optimal control strategy.

Symmetric Sampling is a very effective exploration strategy

The improvement cannot be explained by the fact that SyS uses 2 samples rather than one, as can be seen when comparing it to the third version: PGPE with random sampling and 2 samples batch size. This algorithm performs even worse than the two others. Hence the improvement does in fact come from the symmetry of the two samples. Version 4 of the algorithm assumes that the maximum reward r_{max} is known in advance. Instead of the reward normalisation introduced in Eq. (5.39) the reward is then simply divided by r_{max}. Our experiments show, that even if knowledge of r_{max} is available, it is still beneficial to use the adaptive reward normalisation instead of the real maximum, since it accelerates convergence during the early learning phase. In [Wierstra et al., 2008a] other beneficial reward normalisation techniques are discussed, especially for bigger batch sizes.

8.2 THE FLEXCUBE ENVIRONMENT

8.2.1 *Mass-Spring Systems*

The need for fast but complex robotic benchmarks

State of the art reinforcement learning methods deal with complex, only partial observable and noisy problems. Implementing new ways

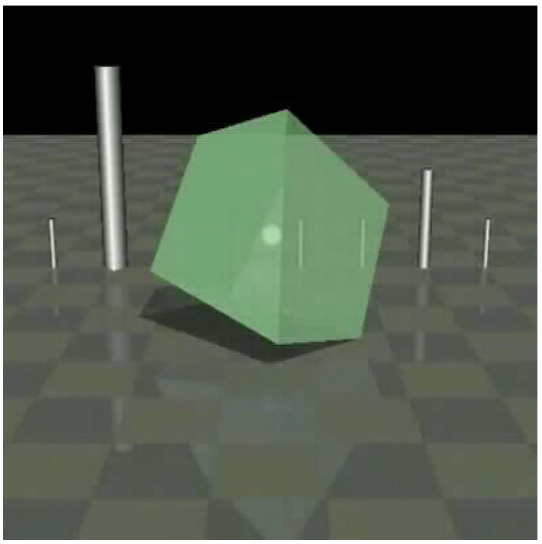


Figure 8.15: The FlexCube mass-spring system.

of reinforcement learning and trying to compare different learning methods, requires a framework that is adequate complex. On the other hand the framework must be fast, so that comparing methods can be done just in time.

Mass-spring systems are fast to calculate but can provide complex RL environments

We investigated in how simple mass-spring systems can be used for fast but complex benchmarking in robotic RL. The physics of mass-spring systems are governed by linear equations the so called spring laws that can be calculated very efficiently on a computer. We were interested if one can build some form of simulated physical agent out of mass points and springs that can learn tasks comparable in complexity to today's robotic tasks. We therefore constructed one of the simplest basic form, a cube. Surprisingly, with this basic form as agent tasks like growing, jumping, static and dynamic walking and approaching food sources with different sensors can be achieved already. The spring morphology provides thereby a noisy, partial observable and also continuous state space. The framework is a test case for all methods that should be capable of handling partial observable Markov decision processes (POMDPs) with mapping from continuous state- to continuous action space. But despite of that principal complexity, the framework is still fast and easy to compute. The success of this basic form convinced us to stay with it for now and we termed the setting FlexCube.

8.2.2 *FlexCube Environment*

The FlexCube environment is a mass-particle system with 8 particles. The particles are modelled as point masses on the vertices of a cube, with every particle connected to every other by a spring (see Fig. 8.15). The agent can set the desired lengths of the 12 edge springs to be anywhere between 0.5 to 1.5 times the original spring lengths. Included in the physics simulation are gravity, collision with the floor, a simple friction model for particles colliding with the floor and the spring forces. We refer to this scenario as the *FlexCube* framework. Though relatively simple, FlexCube can be used to perform sophisticated tasks with continuous state and action spaces.

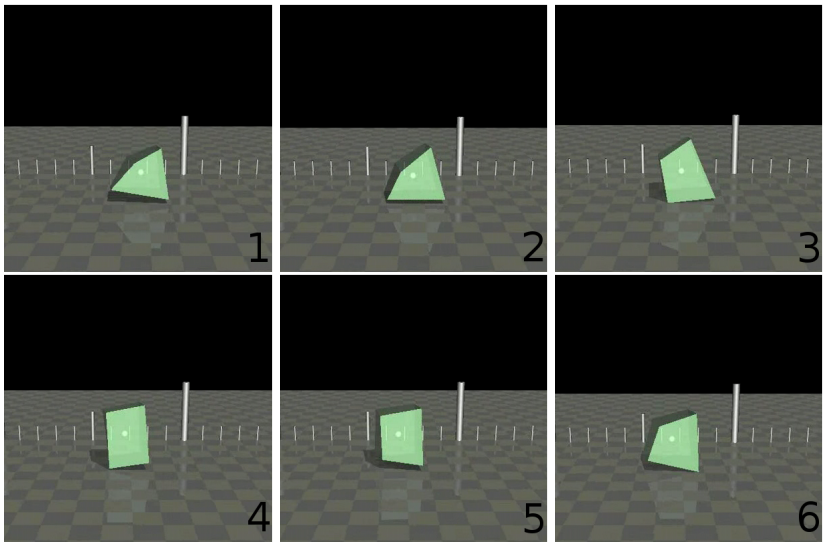


Figure 8.16:
From left to right, a typical solution which worked well in the walking task is shown:
1. Stretching forward.
2. Off the ground.
3. Landing on front.
4. Retracting back.
5. Bouncing off front vertices, landing on back vertices.
6. Stretching forward (cycle closed).

The FlexCube Environment provides therefore twelve synchronous, continuous controllable edges that provide a complex continuous action space. The available sensors are contact sensors for the mass points, length sensors for the springs, "smell" sensors that give distance to food source, a visual sensor, that gives via a ray-tracer a 32x32 picture of the world.

8.2.3 *FlexCube Tasks*

The FlexCube can learn to grow and shrink

GROW SHRINK The grow task is the most simple one we implemented for the FlexCube environment. The goal is to "grow" the cube to a maximum volume with the optimal solution, obviously to maximise the edge spring lengths. The task is not very complex especially because there is no dynamic behaviour needed and it was mainly used to check if new learners are working at all. Its inputs are the 12 current edge spring lengths and the 12 previous desired edge spring lengths (fed back from its own output at the last time step). The policy of the agent is represented by a Jordan network [Jordan, 1986] with 24 inputs, 1 hidden unit and 12 output units.

The FlexCube can learn to walk *statically and dynamically*

WALK In this case the task is to make the cube *walk*—that is, to maximise the distance of its centre of gravity from the starting point. Its inputs are the 12 current edge spring lengths, the 12 previous desired edge spring lengths (fed back from its own output at the last time step) and the 8 floor contact sensors in the vertices. The policy of the agent is represented by a Jordan network [Jordan, 1986] with 32 inputs, 10 hidden units and 12 output units.

The task exists in two different flavours, the static and the dynamic walking task. The difference is in how fast the desired spring lengths are allowed to change. For a slow change only static walking is possible, for fast changes a dynamic walking pattern is expected.

The FlexCube can learn to jump

JUMP For the jumping task the goal is to maximise the distance between the floor and the lowest mass point of the cube at any time step during the episode. Its inputs are the 12 current edge spring lengths, the

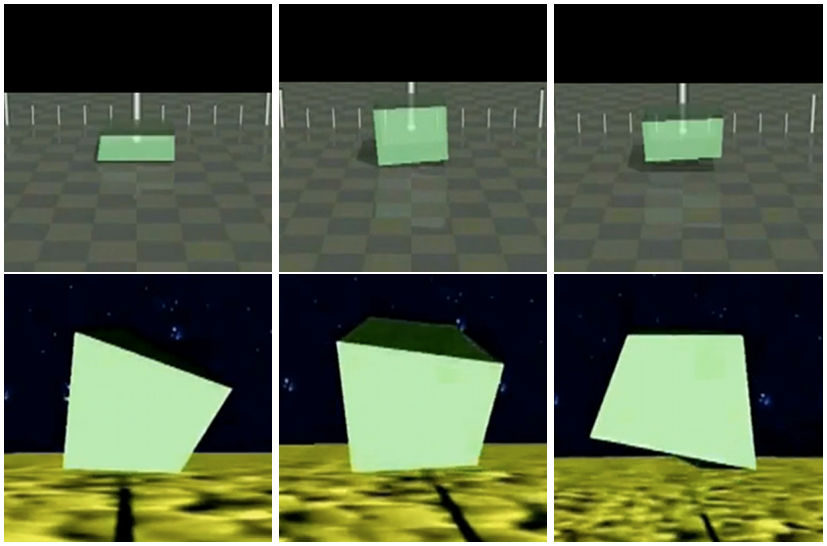


Figure 8.17: From left to right, two typical solutions which worked well in the jumping task are shown: Upper row: 1. Squeezing to the ground. 2. Expanding rapidly. 3. Squeezing together in the air to reach maximum distance from the floor. Lower row: 1. Jumping of right to gain momentum. 2. Jumping off left to gain more momentum. 3. Final jump from the right.

12 previous desired edge spring lengths (fed back from its own output at the last time step) and the 8 floor contact sensors in the vertices. The policy of the agent is represented by a Jordan network [Jordan, 1986] with 32 inputs, 10 hidden units and 12 output units. The task does not seem very complex at first glance, but surprisingly complex dynamical behaviours result out of learning this task. The most effective behaviour was leaning on one side, then jumping on the other to gain momentum or better to gain extra spring contraction in the absorbing springs and then make the final jump (see Figure 8.17 and video [Sehnke, 2010]).

The FlexCube can learn to navigate to food sources

TARGET APPROACH In the target approach task, so called food sources are put into the simulation. The name however is just for intuition. The goal of the task is to reach the food source as fast as possible with either static or dynamic directed walking. The sensory inputs for this task are the 12 current edge spring lengths, the 12 previous desired edge spring lengths (fed back from its own output at the last time step), the 8 floor contact sensors in the vertices and 4 smell sensors that give the distance to the food source but not the direction. The agent has to triangulate the direction to the food source. Therefore the 4 sensors are placed on the 4 top mass points. The policy of the agent is represented by a Jordan network [Jordan, 1986] with 36 inputs, 10 hidden units and 12 output units. This task is the most sophisticated one we created in this environment. The behaviour solving the task must develop a directed walking behaviour that not only moves the cube as fast as possible, but also changes the cube's orientation due to the values of the 4 smell sensors. The task could be learned at least by PGPE with surprising

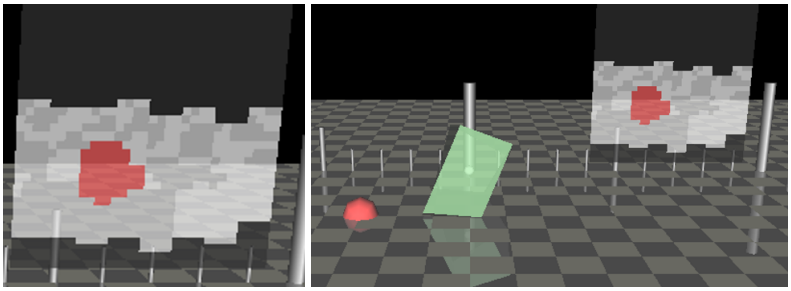


Figure 8.18: The FlexCube "retina" projected in the GL viewer.

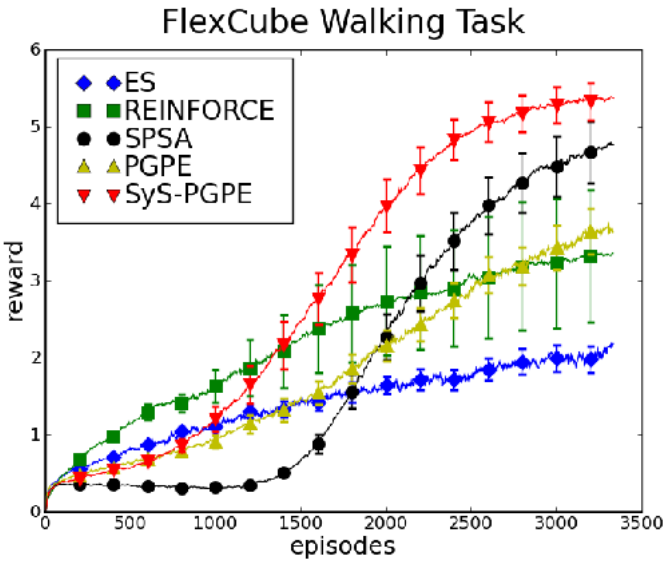


Figure 8.19: PGPE with and without SyS compared to ES, SPSA and REINFORCE on the FlexCube walking task. All plots show the mean and half standard deviation of 40 runs.

accuracy. The most effective behaviour moved the cube in a dynamic walking pattern to the food sources with different locations and even stopped walking on the food source. A behaviour that gives a strange live like impression if watched in the viewer (see video [Sehnke, 2010]).

Can the FlexCube learn to navigate to a food source by sight only?

OPTICAL TARGET APPROACH For the optical target approach task we included a ray-tracer to generate a visual input for the FlexCube. The "retina" is a 32×32 pixel plane that is aligned to the front face of the cube (distortions through the squeezing of the cube are common). The ray-tracer only tracks the chessboard floor and the food sources. No shadows are included and also the irrelevant markers and different shadings are excluded to keep the ray-tracer fast. A typical image of the retina activation can be seen in Figure 8.18. The FlexCube agent gets a new picture as input at every time step. For this task the smell sensors from the last paragraph are not available for the agent. A problem that can be observed is that due to the low resolution of the retina the maximum distance of the food source to be detected reliably in the picture is rather low. A fovea approach could counter this issue, but that is material for future work.

To give the user an impression what the cube currently "sees", the retina is projected in the OpenGL world, shown in Figure 8.18.

The remaining sensory inputs for this task are the 12 current edge spring lengths, the 12 previous desired edge spring lengths (fed back from its own output at the last time step) and the 8 floor contact sensors in the vertices. The agent has to extract the direction to the food source from the images. The policy of the agent is represented by a Jordan network [Jordan, 1986] with 36 inputs, 10 hidden units and 12 output units. However, 4 of the inputs are not direct inputs from sensors, but the outputs of visual preprocessing. We investigated the use of Restricted Boltzmann Machines (RBM) as preprocessing networks. This task is suited for testing agents that use preprocessing of visual data before the actual RL step.

The task itself was not learned to this day and remains future work. However, some interesting insights in using RBMs in a time continuous

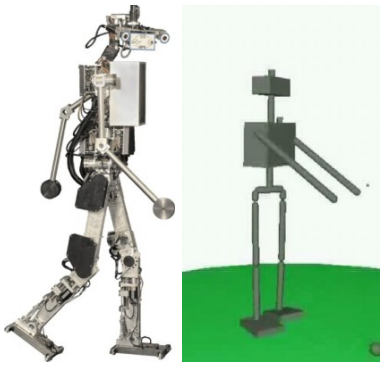


Figure 8.20: The real Johnnie robot (left) and its simulation (right).

environment that provides ordered inputs instead of the usual used randomised input patterns could be gained already.

8.2.4 *FlexCube Results*

Figure 8.19 shows the results on the walking task. All the algorithms learn to move the FlexCube. PGPE substantially outperforms the other methods, both in learning speed and final reward. Here SyS has a big impact on both as well. Figure 8.16 shows a typical scenario of the walking task. Figure 8.17 shows the optimal jumping behaviour. For better understanding please refer to the video on [Sehnke, 2009] and [Sehnke, 2010].

8.3 THE JOHNNIE ENVIRONMENT

8.3.1 *Johnnie Environment*

The Johnnie environment is one of several ODE environments developed for PyBrain [pyb; Schaul et al., 2010]. The ODE environment resembles a physical world, in which arbitrary objects can be placed to interact with each other and the surrounding. The physical behaviour is simulated with the ODE Physics Engine while the scene is rendered using OpenGL. Objects in the world can be easily built using the XODE XML specification. Forces can be applied to every joint and custom sensors can return every value from ODE. The simulation is based on the biped robot Johnnie [Ulbrich, 2008]. The lengths and masses of the body parts, the location of the connection points and the range of allowed angles and torques in the joints were matched with those of the original robot. Due to the difficulty of accurately simulating the robot's feet, the friction between them and the ground was approximated by a Coulomb friction model. The framework has 11 degrees of freedom and a 41-dimensional observation vector (11 angles, 11 angular velocities, 11 forces, 2 pressure sensors in feet, 3 degrees of orientation and 3 degrees of acceleration in the head).

The Johnnie environment is already very sophisticated and the simulation is rather slow compared to the benchmarks presented in the last sections. The Johnnie environment is therefore not suitable for testing, its rather a final benchmark for evaluating different learning methods on a realistic robotic application field. The work on the Johnnie simulation was the project of the author of this thesis for the Cognitive

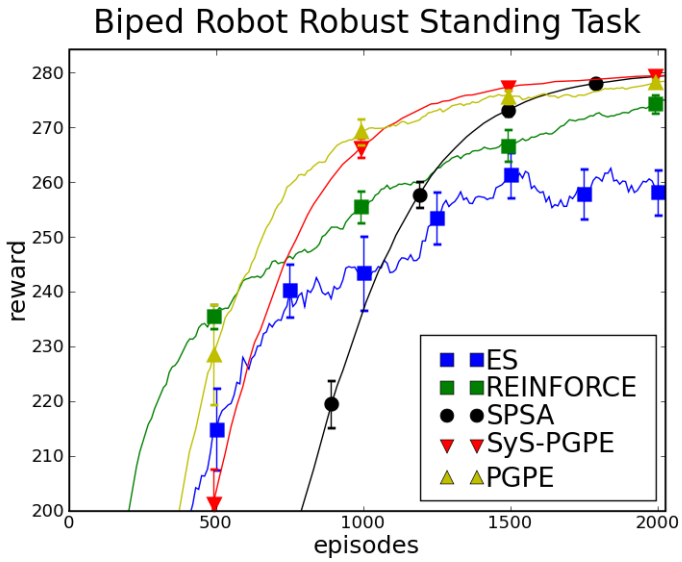


Figure 8.21: PGPE with and without SyS compared to ES, SPSA and REINFORCE on the robust standing benchmark. All plots show the mean and half standard deviation of 40 runs.

Technical Systems (CoTeSys) excellence cluster of the Technische Universität München. The goal was to improve walking behaviours of the real Johnnie Robot, that is an humanoid biped walking robot with a hight of 1.80m and 26 Kg weight (see [Sehnke, 2010] 3:35min for a real life video of Johnnie). The Johnnie environment demonstrates therefore a real today's robotic application.

8.3.2 *Johnnie Tasks*

The controller for all following Johnnie tasks was a Jordan network [Jordan, 1986] with 41 inputs, 20 hidden units and 11 output units.

Johnnie can learn to stand

STANDING TASK The first task in the Johnnie environment is the standing task. The goal of the task is to keep the robot standing. Due to the existence of the feet that have a noticeable size, the task can be completed by finding a stable posture.

Johnnie can learn to jump

JUMPING TASK In the jumping task, the goal is to maximise the height of the head at any point in the episode. A jumping behaviour is the usual solution. The most common learned behaviour is a step forward to gain some momentum followed by a single legged upward jump.

Johnnie can learn to keep standing even if perturbed by external forces

ROBUST STANDING TASK The task in this scenario was to keep a simulated biped robot standing while perturbed by external forces. The aim of the task is to maximise the height of the robot's head, up to the limit of standing completely upright. The robot is continually perturbed by random forces (depicted by the particles in Figure 8.22) that would knock it over unless it counterbalanced.

Can Johnnie learn to walk from scratch?

WALKING TASK In this case the task is to make the robot walk—that is, to maximise the distance of its centre of gravity from the starting point. To prevent the robot from learning a crawling behaviour, the

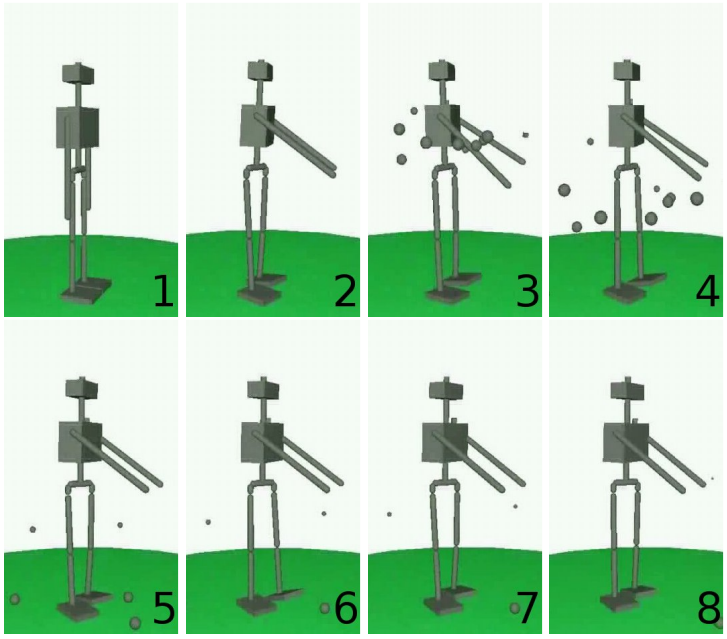


Figure 8.22: From left to right, a typical solution which worked well in the robust standing task is shown: 1. Initial posture. 2. Stable posture. 3. Perturbation by heavy weights that are thrown randomly at the robot. 4. - 7. Backsteps right, left, right, left. 8. Stable posture regained.

distance travelled is multiplied by the height of the head. This task is not learnable from scratch in one go, at least not with the available methods. Therefore we implemented two horizontal bars that guided the head of the robot. The robot could still lose balance and fall, but only a few centimetres, so that it was easily able to recover.

8.3.3 *Johnnie Results*

As can be seen from the results in Fig. 8.21, the task was relatively easy, and all the methods were able to quickly achieve a high reward. REINFORCE learned especially quickly, and outperformed PGPE in the early stages of learning. However PGPE overtook it after about 500 training episodes. Figure 8.22 shows a typical scenario of the robust standing task. For more details please refer to the video on [Sehnke, 2009].

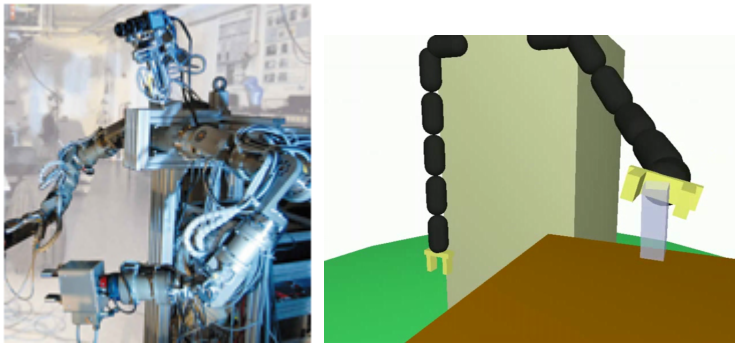


Figure 8.23: The real CCRL robot (left) and its simulation (right).

8.4 THE CCRL ENVIRONMENT

8.4.1 *CCRL Environment*

The simulation based on the CCRL robot [Buss and Hirche, 2008] is the second Open Dynamics Engine environment used. The lengths and masses of the body parts and the location of the connection points were matched with those of the original robot. Friction was approximated by a Coulomb friction model. The framework has 8 degrees of freedom (per arm) and a 35-dimensional observation vector (8 angles, 8 angular velocities, 8 forces, 2 pressure sensors in hand, 3 degrees of orientation and 3 values of position in hand, 3 values of position of object). This environment is well suited to learn object grasping and manipulation tasks. Like for the Johnnie environment, the CCRL environment is not suitable for testing, its rather a final benchmark for evaluating different learning methods on a realistic robotic application field. The work on the CCRL simulation was part of a project proposal of the author of this thesis for the Cognitive Technical Systems (CoTeSys) excellence cluster of the Technische Universität München. The goal was to show that adaptive behaviour is possible with the CCRL architecture by our means of learning methods (see [Sehnke, 2010] 2:25min for a real life video of CCRL robot). The CCRL environment demonstrates therefore a real today's robotic application.

8.4.2 *CCRL Tasks*

The controller for all tasks was a Jordan network [Jordan, 1986] with 35 inputs, 10 hidden units and 8 output units.

The CCRL simulation can learn to reach a given position with the gripper

POINTING TASK The task is to bring the hand to target positions in the work space of the arm. The task was learned in 3 phases, with progressively more difficult initial positions for the target. In the first phase, the target was always at the same place in the centre of the workspace. In the second phase it was normally distributed around the centre of the work space (standard deviation of 10cm). In the last phase it was placed with equal probability anywhere in the work space. Every phase required 10.000 episodes and used the final controller of the preceding phase.

The CCRL simulation can learn to grasp an object from a fixed position

SIMPLE GRASPING TASK In the simple grasping task a table is added to the scenery. An object is placed at the very edge of the table. The task then is similar to the first phase of the pointing task, the hand has to reach the object position. However, the hand has to reach this position in a horizontal aligned manner, not to push away the object. The grasping itself is done as a "reflex" if the distance of the hand to the object undergoes a certain threshold. This task is designed in this way because in that manner it can be learned from scratch in one phase in about 10.000 episodes.

The CCRL simulation can learn to grasp different objects from arbitrary position

GRASPING TASK The task in this scenario was to grasp an object from different positions on a table. The task was learned in 4 phases, with progressively more difficult initial positions for the object. In the

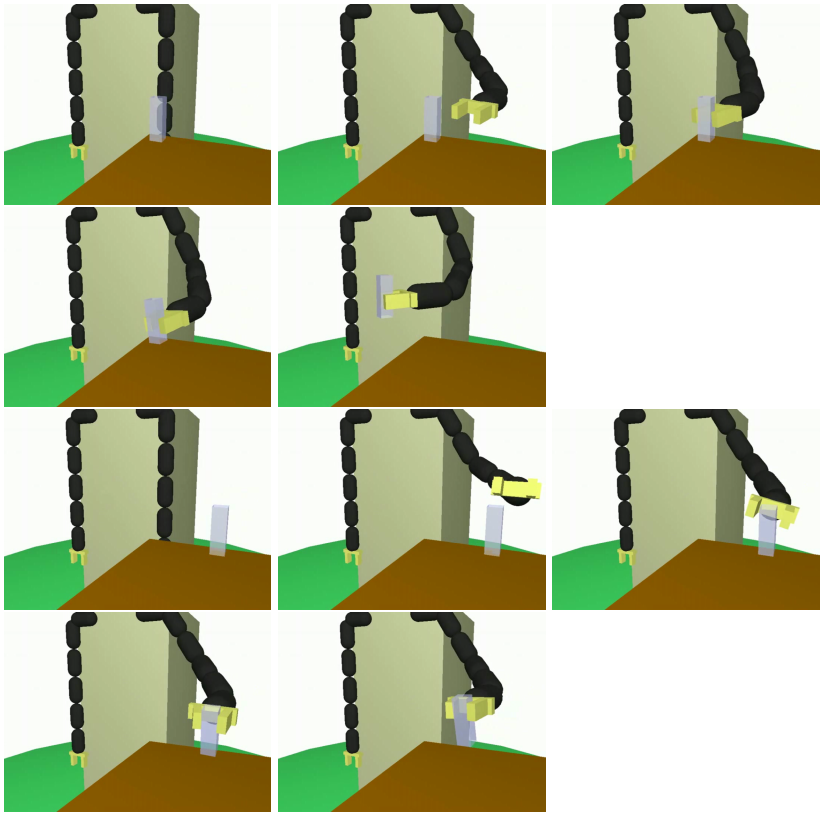


Figure 8.24: From left to right, a typical solution which worked well in the grasping task is shown for 2 different positions of the object with the same controller: 1. Initial posture. 2. Approach. 3. Enclose. 4. Take hold. 5. Lift.

first phase, the object was always in the same place on the edge of the table.

In the second phase it was still in a fixed position, but away from the edge. In the third phase it was normally distributed around the centre of the reachable region (standard deviation of 10cm). In the last phase it was placed with equal probability anywhere in the reachable area. Every phase required 10.000 episodes and used the final controller of the preceding phase.

In contrast to the simple grasping task the actual grasping was done by the controller and so the timing of the grasping had to be learned. Figure 8.24 shows a typical solution of the grasping task. For more detailed views of the solution please see the video on [Sehnke, 2009].

8.4.3 *CCRL Results*

We used the simple grasping task for evaluation because the incremental learning of the variable grasping tasks makes it hard to visualise the results in a diagram. We compared PGPE to a related method called Natural Evolution Strategies (NES, [Wierstra et al., 2008b]) that is a modification of CMA-ES but converged on surprisingly similar principles than PGPE. Because NES was developed for optimising problems with small to mid-dimensional tasks, there are some important differences. NES uses the natural gradient and therefore big sample batches and a covariance matrix for optimisation, while PGPE uses the vanilla gradient and only two samples per learning update step and only the diagonal of the covariance matrix to circumvent the quadratic growth in memory requirements. Therefore NES is more effective in problem

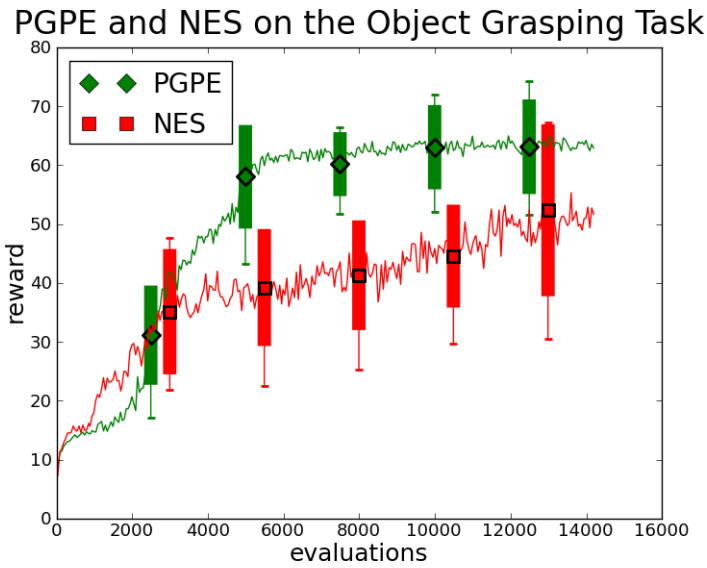


Figure 8.25:
The performance of PGPE on the simple grasping task compared to the related method Natural Evolution Strategies.
Source: Rückstieß et al. [2010]

domains with less then about 50 dimensions while PGPE is more effective for problem domains with more than about 50 dimensions. For this experiment we used, in contrast to the above setup, a controller with less parameters. The task could be learned with a NN with only 48 weights, what is directly in the interval where both kinds of algorithms should behave similar effective. As it can be seen in Figure 8.25 both algorithms perform nearly the same. Both algorithms manage to learn to grasp the object from scratch in reasonable time. After about 5,000 episodes the controller grasps the object securely, just the time it needs to get hold of the object is improved from there on.

9 PHYSICAL CRYPTOGRAPHY

9.1 PHYSICAL CRYPTOGRAPHY

Modern cryptography uses so-called one-way functions. One-way functions are named by their defining property that they are easy to evaluate but hard to invert. A cryptographic usable one-way function should be easy to access and to compute, but it should be practically infeasible to invert. Usually modern cryptography uses numerical one-way functions. Their advantages are obvious if it comes to accessibility. The famous public key method is maybe the most prominent instance of cryptography that uses a numerical one-way function.

Physical Unclonable Functions (PUF) replace the numerical one-way function by a physical one. The defining properties stay thereby the same. A physical system is needed that is easy to access and fast to evaluate, but practically impossible to invert, or to use a more intuitive word in the case of physical systems, it should be impossible to predict. Several candidates for PUFs have been proposed. In this thesis, we concentrate on the electrical PUFs proposed. [Pappu et al., 2002], [Gassend et al., 2002].

Beside several other reasons, there is one main reason to omit the comfortable numerical one-way functions in favour of a physical one-way function: Numerical one-way functions are based one a secret key. If the secret key and the cryptography method is known to an attacker, the security of the cryptography system is completely broken. The user can also be completely unaware that a third person got knowledge of the secret key. Also secret keys can be extracted by a variety of methods in todays computer systems, like viruses and the like.

If one replaces the one-way function with a physical one, the complex internal structure of the PUF replaces the role of the secret key in usual crypto systems. Because the user will sooner or later be aware that the physical device that implements the physical one-way function is missing, retrieving the secret part of the crypto system unnoticed is much harder. Therefore the secure storing of the PUF is not the main issue anymore. Because of that new constraints arise. If an attacker gets access to the device for a short time (so that the owner is not aware of the missing device) it should be practically impossible to read out the complete information. It also should be very unlikely that the attacker can predict the complete input-output behaviour of the PUF with the knowledge he gathered in that short time.

Naturally the last point is where machine learning techniques are promising candidates for successfully attacking the security properties of PUFs. We will investigate to which extend PGPE is an effective method of attacking PUF security by learning a sufficient model of the PUF and therefore predicting its input-output behaviour by using a limited amount of read out information from the PUF that can be gathered in a very short period of time. We focus on PUFs that implement

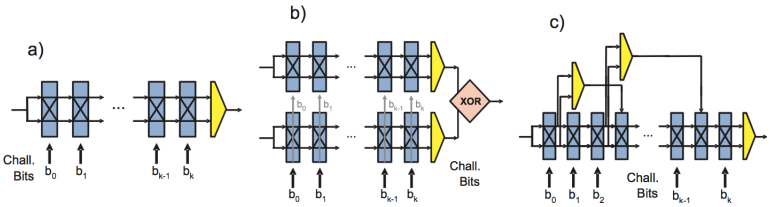


Figure 9.1: Illustration of the architectures of a Standard Arbiter PUF (a), XOR Arbiter PUF (b) and Feed Forward Arbiter PUF (c). The challenge bits b_i at each stage decide if the two incoming signals propagate in parallel through the stage, or if their paths are crossed. All signal paths have slightly different run time properties due to uncontrollable, small fabrication variations. An arbiter element depicted as yellow at the end of the Standard Arbiter PUF (a) decides which of the two signals arrived first, and correspondingly outputs 0 or 1. In an XOR Arbiter PUF (b), the output of several Standard Arbiter PUFs is XORed. In FF Arbiter PUFs, signals at earlier stages of the circuit are fed into an arbiter element, whose output is applied as external bit at later stages of the circuit.

a non-differentiable model where supervised learning methods can not be applied.

9.1.1 *Physical Unclonable Functions*

All electrical PUFs analysed in this thesis have some common characteristics (see figure 9.1): The PUF consists of a number of switches. The challenge bits b_i decide if the switch i is switched or not. The two incoming signals propagate therefore either in parallel through the stage, or their paths are crossed. All signal paths have slightly different run time properties due to uncontrollable, small fabrication variations like the exact width or the precise length of the path. An arbiter element at the end of the PUF decides which of the two signals arrived first, and correspondingly outputs 0 or 1. This is the basic layout of an arbiter PUF. If one assumes that the delay of a complete path from the signal start to the arbiter is the sum of the single delays at each stage, the PUF can be described by a very simple linear model that is also differentiable. Gassend et al. established a compact parametric linear model that subsumms the delays into a notation of delay differences so that only one parameter is left per stage instead of 4 and an additional parameter for the whole PUF that functions like a bias [Gassend et al., 2004]. The notation Gassend uses is:

$$\Delta = \vec{w}^{\mathsf{T}} \vec{\Phi} \tag{9.1}$$

where $\vec{w}$ and $\vec{\Phi}$ are of dimension $k+1$. The parameter vector $\vec{w}$ encodes the delays differences for every stage. The feature vector $\vec{\Phi}$ encodes the challenge transformed in a way that it also encodes the parity up to the given bit. See [Gassend et al., 2004] [Lim, 2004] [Majzoobi et al., 2008a] [Majzoobi et al., 2008b] for details.

It was shown by [Lim, 2004] that the linear delay model represents an *in-silicon* PUF sufficiently well. We therefore focus on finding the parameters of a given linear delay model like suggested in [Majzoobi et al., 2008b].

STANDARD ARBITER PUF. The Standard Arbiter PUF (Arb-PUF) is completely described by the above mentioned linear delay model eq. (9.1). For an n-stage Arb-PUF we need to find an $n+1$ sized parameter vector that describes the input-output behaviour of the PUF best.

XOR ARBITER PUF. Because the Arb-PUF was very easy to model even with simple approaches like a perceptron, ways to make the model

more complex were investigated. [Suh and Devadas, 2007] therefore suggested using several Arb-PUFs (see Fig. 9.1 b) and XOR their output to a final output of the so called XOR-PUF. Interestingly, this rather simple appearing measure proves to result in one of the most resilient PUF structures.

FEED FORWARD ARBITER PUF. Feed Forward Arbiter PUFs (FF-PUF) are another approach to strengthen the resistance against ML modeling attacks [Gassend et al., 2004; Lee et al., 2004; Lim, 2004; Majzoobi et al., 2008b]. Again, this PUF type is based on the Arb-PUF. Additional arbiters are used to transmit signals from earlier stages of the PUF to replace challenge bits at later stages of the PUF. This change in architecture makes the PUF not only more resilient to ML modeling attacks, it also makes the model non-differentiable and so resistant to supervised learning approaches.

9.1.2 *Attacking PUFs with Machine Learning*

We investigated the scenario where an attacker has obtained the PUF for a limited time and could read out a certain amount of responses to given challenges, so-called Challenge-Response-Pairs (CRP), or that the attacker gains knowledge of a certain number of CRPs by intercepting the communication of the PUF owner. The goal of the attacker is to construct a model with the given CRPs that predicts the complete input-output behaviour of the PUF sufficiently well to pose a threat to the PUFs security.

SUPERVISED APPROACHES SVMs and Perceptrons were successfully used to cryptanalyse Arb-PUFs, XOR-PUFs with only 2 XOR inputs and FF-PUFs with only one FF-Loop (Gassend et al. [2004]; Lim [2004]; Majzoobi et al. [2008b]). This approaches failed however on XOR-PUFs with more than 2 XORed Arb-PUFs and on FF-PUFs with more than one loop. In [Rührmair et al., 2010] and [Sölter, 2009] Logistic Regression (LR) was successfully applied to higher order XOR-PUFs. But on higher order FF-PUFs also the LR approach failed. This architecture resisted any approach to interpret the model in a differentiable way to make supervised learning possible.

REINFORCEMENT LEARNING APPROACHES No methods from the field of RL have been used to attack PUFs, mainly because the value functions usually implemented are not applicable to the highly discrete nature of the PUF output.

Policy Gradients make an exception here, but they assume a differentiable model to back-propagate the log-likelihoods to the parameters of the model. If that property of the PUF model is given supervised techniques can be applied that have much better convergence properties.

PGPE however can be applied in the RL framework to non-differential models like the FF-PUF. That was done in [Sehnke et al., 2010b] and the results are presented here in more details.

EVOLUTIONARY APPROACHES Evolutionary Algorithms have the same property with respect to the model they can train - they can be

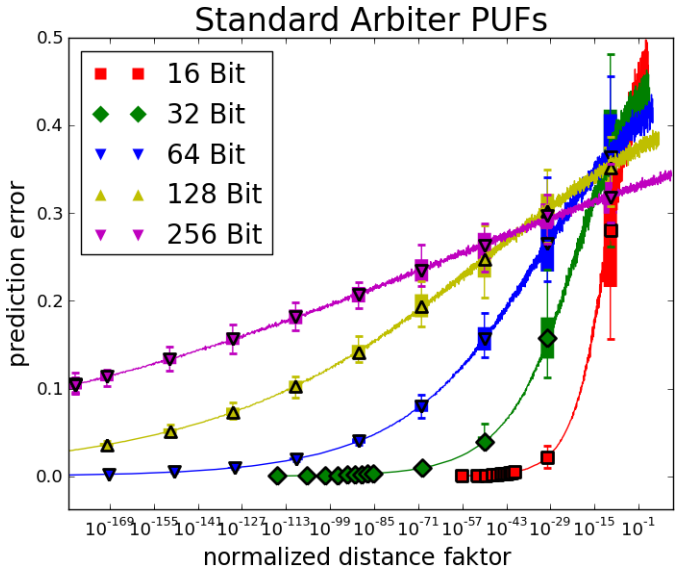


Figure 9.2: The reproduction error of Arb-PUF instances with certain euclidean distance to the original instance.

applied to non-differential models. Evolutionary Algorithms, in this case Evolution Strategies (ES) were used in [Rührmair et al., 2010] for cryptanalysing PUFs. In this thesis we will show that ES is a strong tool for attacking PUFs, but that PGPE is the more reliable and faster choice in all investigated cases.

9.2 RESULTS

Again we used the standard implementation of PGPE with the PGPE standard meta-parameters: 2-Sample Symmetric Sampling, starting standard deviation for exploration as the standard deviation assumed for the PUFs and step sizes of 0.2 and 0.1 for the parameter and the sigma update. We also applied the usual reward normalisation for PGPE.

For ES we used a comma-best selection with population size (6,36) like in previous chapters and the usual values for τ dependent on the PUF model dimension.

Table 2: The evaluations needed to achieve an average prediction rate of 90% and 95% with ES and PGPE. "E" marks the columns with the average evaluations, while "E/Bit" marks the columns that show the evaluations needed per number of bits.

ES on Arb-PUFs					**PGPE on Arb-PUFs**			
	90%		95%		90%		95%	
Bit	E	E/Bit	E	E/Bit	E	E/Bit	E	E/Bit
16	446	27.88	720	45.00	118	7.38	190	11.88
32	878	27.44	1530	47.81	219	6.84	384	12.00
64	1879	29.36	3589	56.08	467	7.30	834	13.03
128	4230	33.05	9480	74.06	1080	8.44	1890	14.77

9.2.1 *Standard Arbiter PUF*

PROPERTIES. To give some information on how hard the modeling problem is for the different PUF architectures, we show how the reproduction error grows with increasing distance to the original parameter

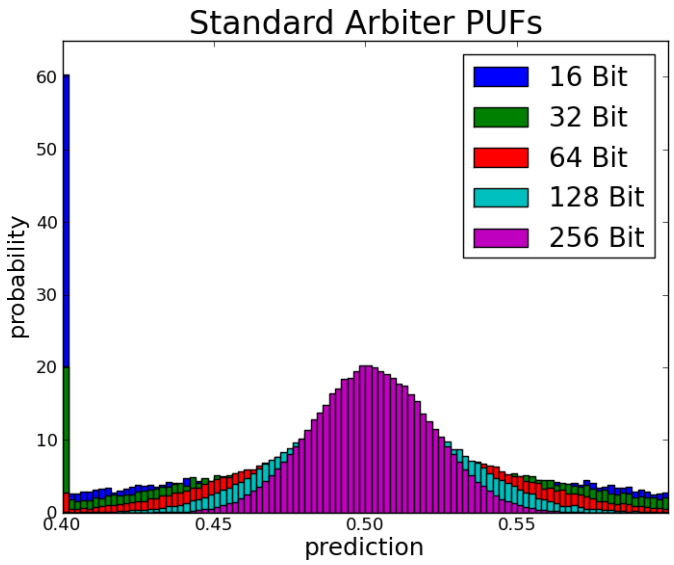


Figure 9.3:
The histogram of 10,000 samples, sampled with 100,000 CRPs. The samples were drawn from a normal distribution and compared with the original instance. The histogram shows the prediction error distribution for Arb-PUFs.

set like shown in figure 9.2. We also provide an histogram for each PUF architecture that shows the distribution of reproduction errors with randomly initialised models like in figure 9.3 (same standard deviation as assumed for the PUFs fabrication variances). Both figures give already a hint that the Arb-PUF is very easy to model.

RESULTS. A prediction rate of 99% could be achieved in 20,000 evaluations for all investigated PUF dimensions like shown in table 2. Furthermore, the results suggest that the number of evaluations needed, grows only linearly with increasing number of bits. PGPE needs only a forth of the evaluations to converge compared to ES. The figures 9.4 depict this result nicely.

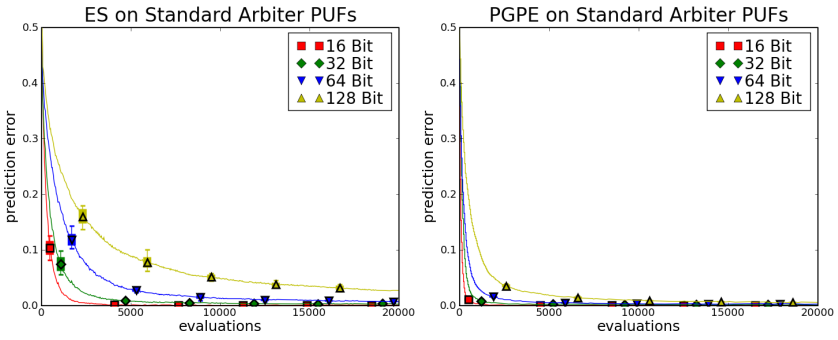


Figure 9.4:
The best of 10 runs on each Arb-PUF architecture with ES and PGPE.

Table 3: The number of evaluations needed to achieve a prediction rate of 90% and the rate of runs that achieved this prediction rate in the given maximal number of evaluations with ES and PGPE. "E" marks the columns with the evaluations needed, while "Rate" marks the column that shows the rate of successful runs.

	ES on XOR-PUFs				PGPE on XOR-PUFs			
	E		Rate		E		Rate	
XOR	16 Bit	64 Bit	16 Bit	64 Bit	16 Bit	64 Bit	16 Bit	64 Bit
2	1080	5508	100%	100%	315	1350	100%	100%
3	3031	17460	90%	50%	556	5620	50%	90%
4	5796	-	30%	0%	1690	-	40%	0%
5	-	-	0%	0%	2810	-	40%	0%

9.2.2 *XOR Arbiter PUF*

PROPERTIES. The reproduction error grows much faster with increasing distance to the original parameter set than in the Arb-PUF case (figure 9.5). This has major effects on the histogram for the XOR-PUF architecture as well (figure 9.6). Both figures give a hint that the XOR-PUF is much harder to model than the Arb-PUF. Noteworthy is how fast the number of samples that significantly divert from 0.5 reproduction error drops with the number of XORed Arb-PUFs.

RESULTS. A prediction rate of 90% could be achieved for ES in less than 20,000 evaluations for 16-bit up to 4 XORed Arb-PUFs and for 64-bit up to 3 XORed Arb-PUFs like shown in table 3. Furthermore the results suggest that the number of evaluations needed, grows exponentially with increasing number of XORed Arb-PUFs. PGPE needs in average only a third of the evaluations to converge compared with ES. With PGPE we were also be able to learn the 16-bit 5 XOR PUF. The figures 9.7 and 9.8 depict this result nicely.

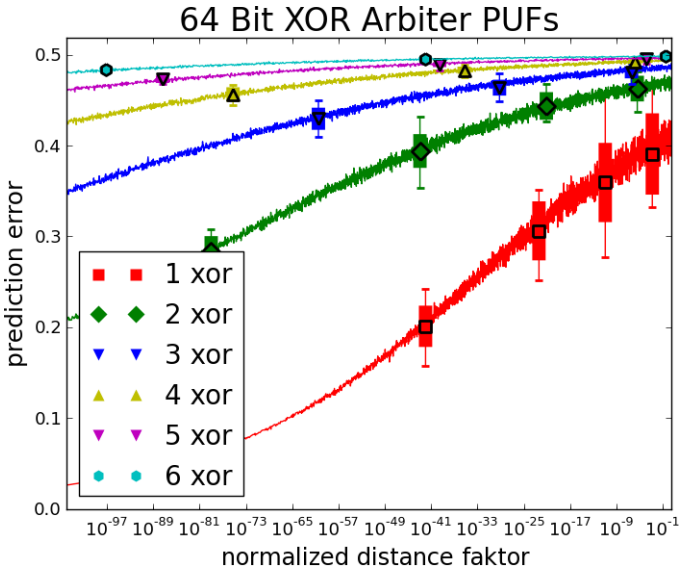


Figure 9.5: The reproduction error of 64Bit XOR-PUF instances with certain euclidean distance to the original instance. Here the distance is normalised for the exponential growth with distance. Obviously, the error increases very rapidly with the number of dimensions.

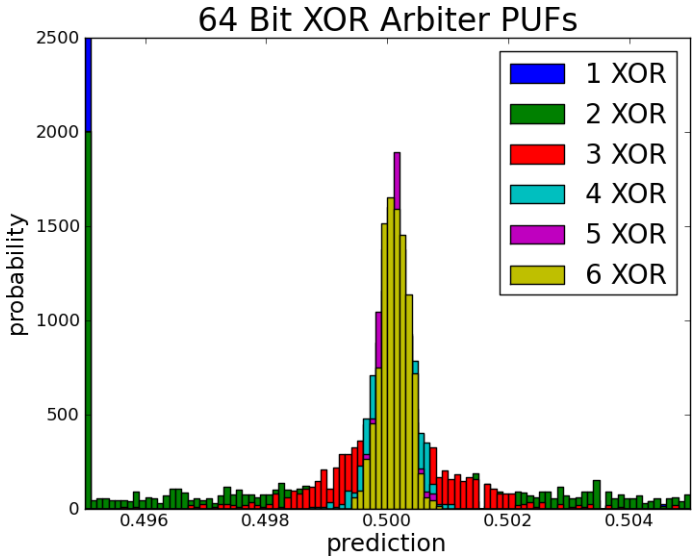


Figure 9.6:
The histogram of 10,000 samples, sampled with 10,000,000 CRPs. The samples were drawn from a normal distribution and compared with the original instance (also normal distributed drawn). The histogram shows the prediction error distribution.

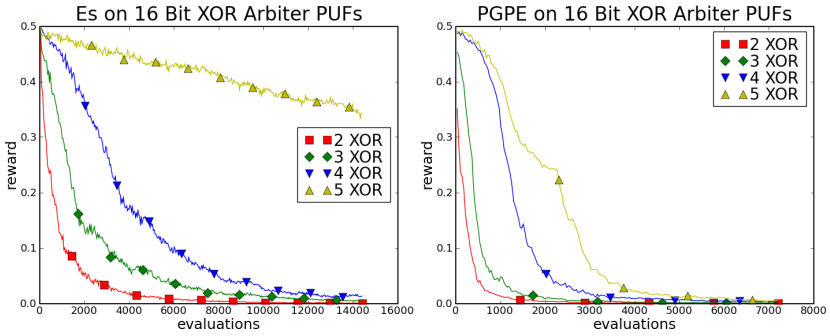


Figure 9.7:
The best of 10 runs on each XOR-PUF architecture with a 16-bit input vector with ES and PGPE.

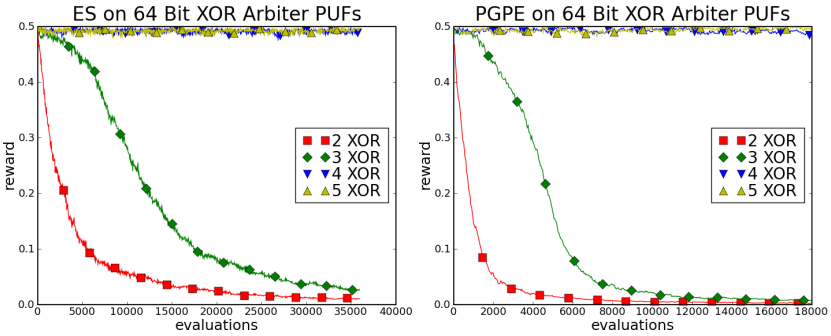


Figure 9.8:
The best of 10 runs on each XOR-PUF architecture with a 64-bit input vector with ES and PGPE.

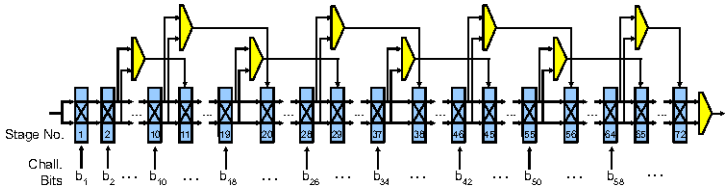
Table 4: The evaluations needed to achieve a prediction rate of 90% and 95% for the best run out of 40 (80 for 9+10 FF) for ES and out of 10 (20 for 9+10 FF) for PGPE. E stands for the required number of evaluations, and E/FF symbolises the required evaluations divided by the number of FF loops. "Best" marks the columns with the best results after 72,000 evaluations (ES) and 36,000 evaluations (PGPE).

	ES on FF-PUFs					PGPE on FF-PUFs				
	90%		95%		Best	90%		95%		Best
FF	E	E/FF	E	E/FF	Result	E	E/FF	E	E/FF	Result
5	10200	2040	17100	3420	99.3%	2900	580	3730	746	99.6%
6	21200	3533	42300	7050	97.7%	7840	1307	9340	1557	99.5%
7	10100	1443	22700	3243	97.4%	4710	673	6580	940	98.2%
8	17600	2200	53100	6638	95.5%	4840	605	21350	2669	96.1%
9	-	-	-	-	89.2%	-	-	-	-	89.3%
10	37500	3750	-	-	93.4%	15480	1548	-	-	90.9%

9.2.3 *Feed Forward PUF*

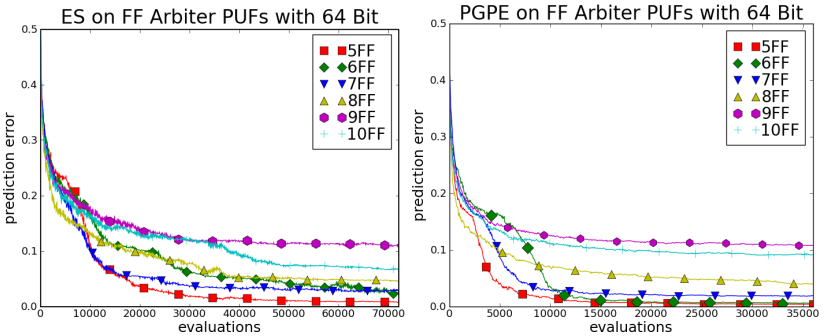
PROPERTIES. The reproduction error grows much faster with increasing distance to the original parameter set than in the Arb-PUF case (figure 9.11), although the growth in the XOR-PUF case seems to be more extreme. This also effects the corresponding histogram for this architecture (figure 9.12). The histogram also shows that local minima arise in the search space of this PUF architecture. To show this in more detail we refer to figure 9.13. This sketch of the FF-PUF search space is achieved by learning 40 individual solutions with PGPE. The found delay sets (and the original one) were ordered by means of similarity via a one dimensional self organising map [Kohonen, 1995]. The areas between two sets is evaluated with the linearly interpolated parameter sets. All 3 figures give a hint that the FF-PUF is much harder to model than the Arb-PUF. Noteworthy is that the FF-PUF architecture has the additional advantage of having a non-differentiable model.

Figure 9.9: The architecture of the FF-PUFs that we employed in our ML experiments, shown for the 8 FF-loop case.



RESULTS. A prediction rate of 90% could be achieved for ES in less than 40,000 evaluations for 64-bit FF-PUFs up to 10 FF-loops and for PGPE in less than 20,000 evaluations like shown in table 4. Furthermore,

Figure 9.10: The best of 40 runs (80 for 9+10 FF) with ES and PGPE on each FF-PUF architecture.



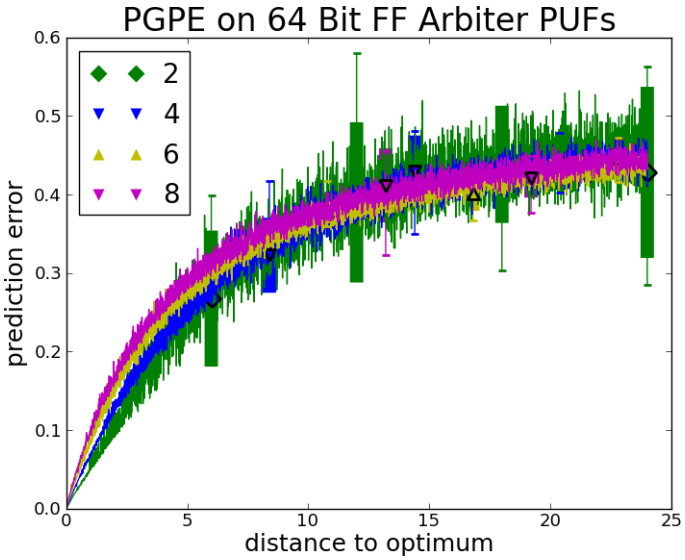


Figure 9.11:
The reproduction error of 64 Bit FF-PUF instances with certain euclidean distance to the original instance. Because the dimensionality is not changing much for increasing numbers of FF-Loops, we skip a normalised plot for briefness.

the results suggest that the number of evaluations needed grows at least polynomial with increasing numbers of FF-loops. PGPE needs in average only a third of the evaluations to converge compared to ES. The figure 9.10 depict this result nicely.

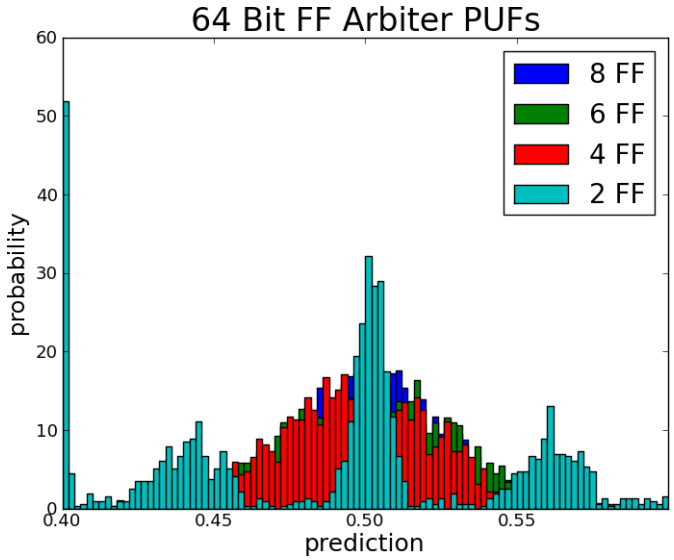


Figure 9.12:
The histogram of 10,000 samples, sampled with 100,000 CRPs. The samples were drawn from a normal distribution and compared with the original instance. The histogram shows the prediction error distribution for FF-PUFs.

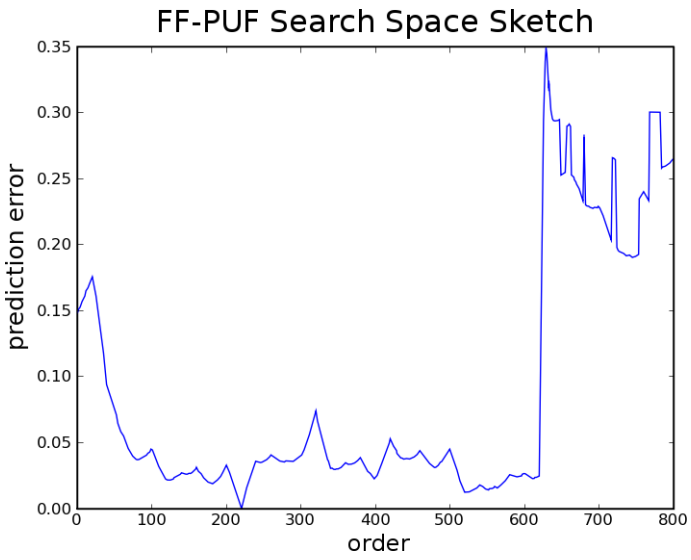


Figure 9.13: Sketch of the FF-PUF search space. 40 individual solutions were learned with PGPE until convergence. The so found delay sets (and the original) were ordered by similarity via a one dimensional self organising map.

.

10

PATTERN RECOGNITION WITH DEEP NETWORKS FOR REINFORCEMENT LEARNING

Sensor information can get to complex for direct RL

In Reinforcement Learning (RL) if the sensor variables get too complex to use them for RL directly, one usually uses a modular setup. For example if a robot has to learn to navigate using vision (a camera picture), usually computer vision modules are used to preprocess the image and present the agent a feature vector that contains the relevant information that was extracted from the image.

In the last years deep believe nets became very popular for extracting high-level information out of complex input data in a completely unsupervised way. The name part deep stems from the fact that this nets are build with many layers and are therefore deep nets. The name part believe stems from the fact that this nets are generative.

RBM deep believe nets are well suited to transform complex data like camera pictures in for RL usable representations

The most prominent learning method for deep believe nets is the so called Restricted Boltzmann Machines (RBM). This is a completely unsupervised learning method. The construction of higher representations of information content in the net is therefore determined by the structure in the data itself.

RBMs have shown to produce very impressive results for classification in vision tasks. They are abel to produce localised *receptive fields* and can, on higher levels of the net, generate translation independent output to visual stimuli.

For this thesis our attention got to RMBs, because we could use deep neural nets that can also be learned by PGPE because PGPE can cope with the extremely high-dimensional space such a deep net produces. The idea was to use RBM learning in combination with RL in an online

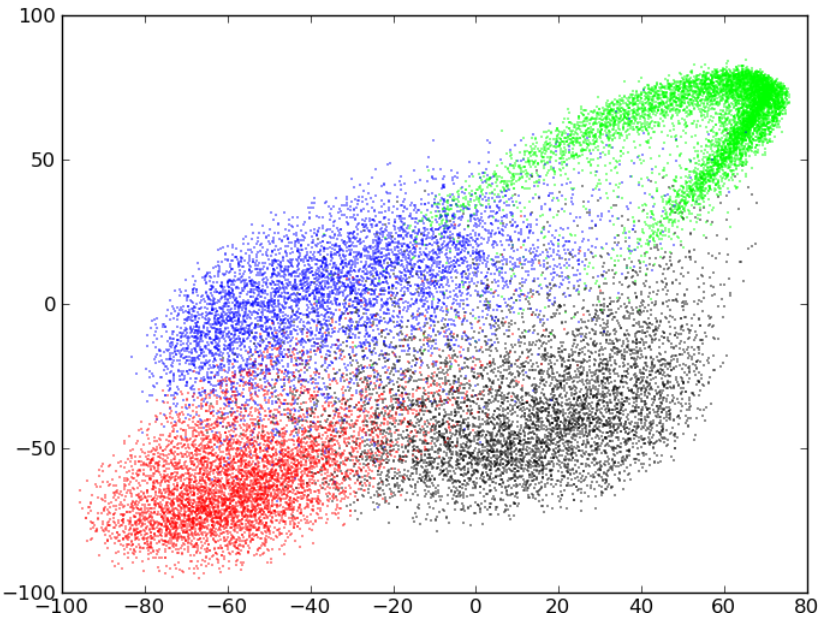


Figure 10.1:
The distribution of digits processed by the deep net after standard RBM training.
Red = 0;
Green = 1;
Blue = 2;
Black = 3.

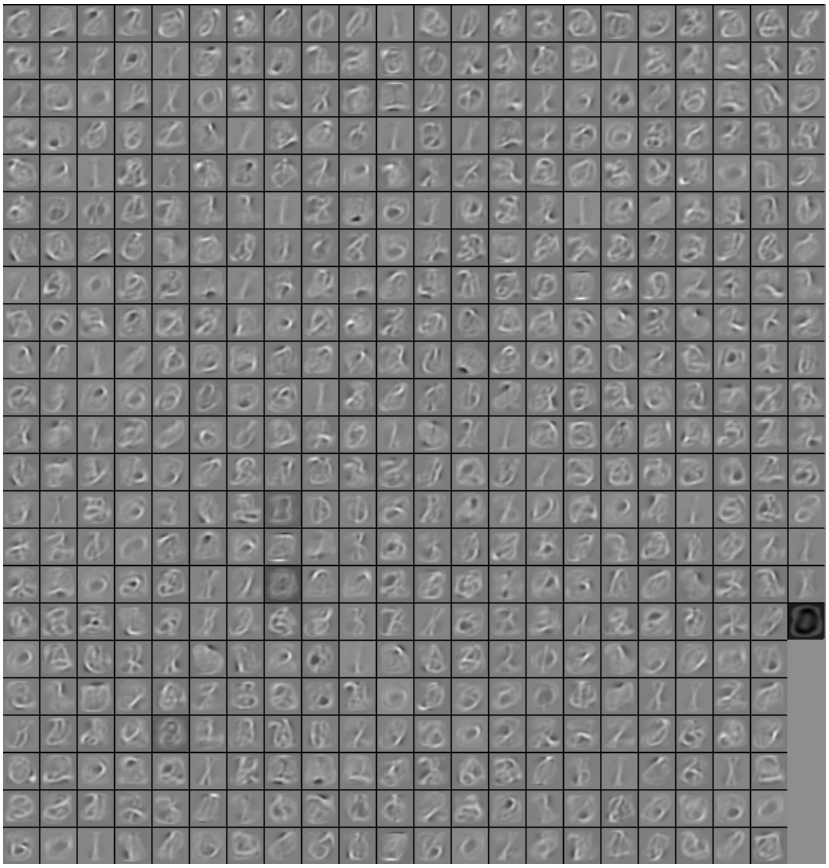


Figure 10.2: The weight vectors of the first hidden layer for standard RBM training. Every *square* is the weight vector of one neurone represented in the 2D structure of the digit images (28x28).

fashion and let PGPE fine-tune even the deep net weights due to the RL gradient.

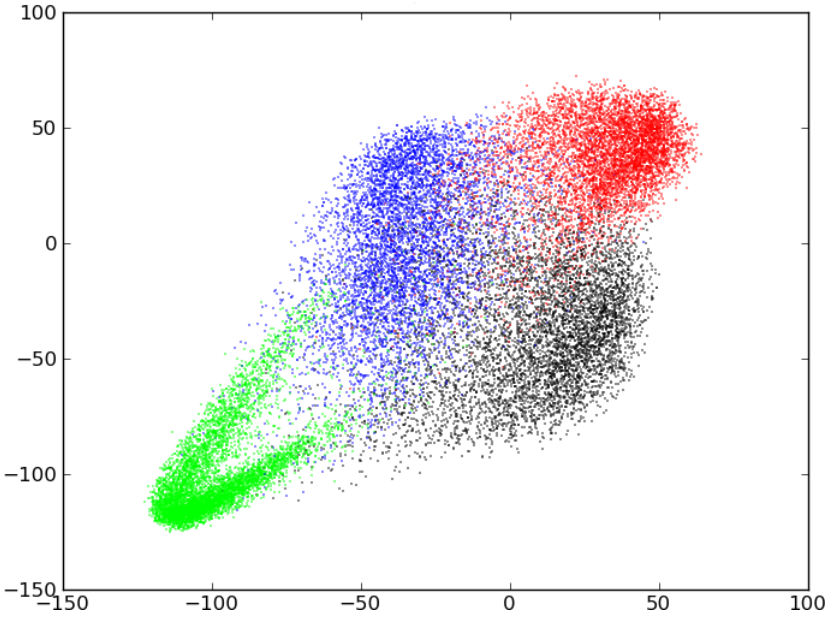


Figure 10.3: The distribution of digits processed by the deep net after online RBM training. Red = 0; Green = 1; Blue = 2; Black = 3.

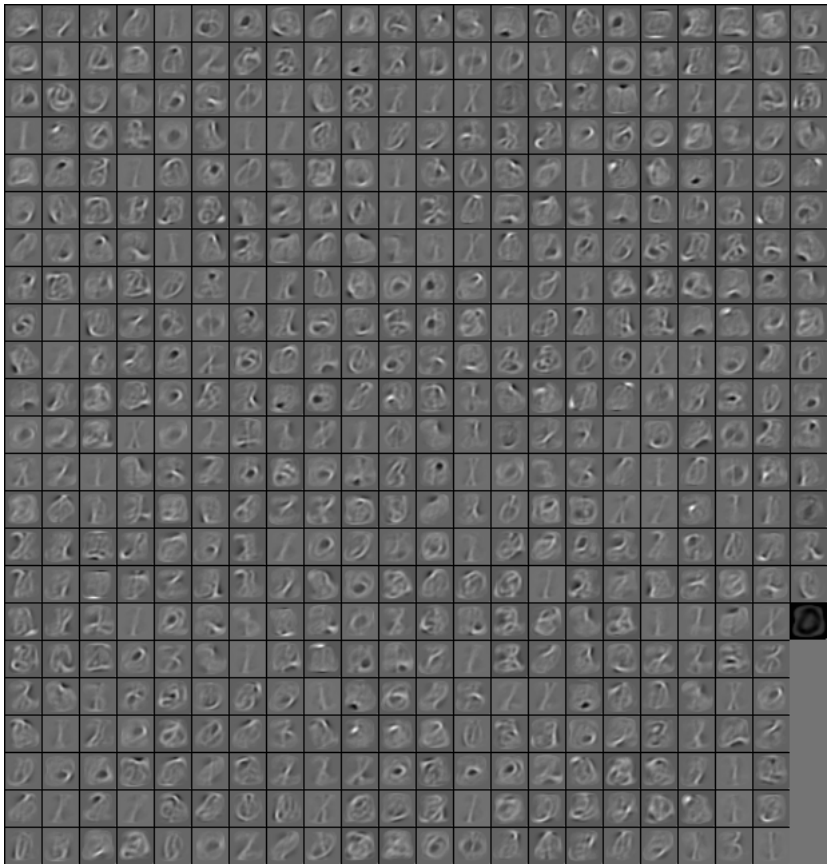


Figure 10.4:
The weight vectors of the first hidden layer for online RBM training. Every *square* is the weight vector of one neurone represented in the 2D structure of the digit images (28x28).

10.1 RBM ONLINE LEARNING

RBMs are usually trained layer by layer. If the weight matrix of the first layer has converged, it is fixed and the next layer is learned by calculating the activations through the first now fixed layer and using this activations as inputs for the second layer. For RL it would be advantageous that the deep net learns online all layers so that the net produces an output (that will change over time) that the agent can use. Making a random walk through the lab of several hours first, then train the deep net layer by layer and then let the agent learn via RL seems kind of impractical.

Learning all layers at once

The first question we had to answer therefore is if a deep net can be learned online for all layers at once. The danger could be that later layers could adapt already to the noisy *nonsense* activations of not converged earlier layers.

To evaluate the effects of online learning we used (for reasons of visibility, only) the first 4 digits from the MNIST set [LeCun and Cortes, 1998]. So we had 6,000 hand written numbers for the digits 0–3 each.

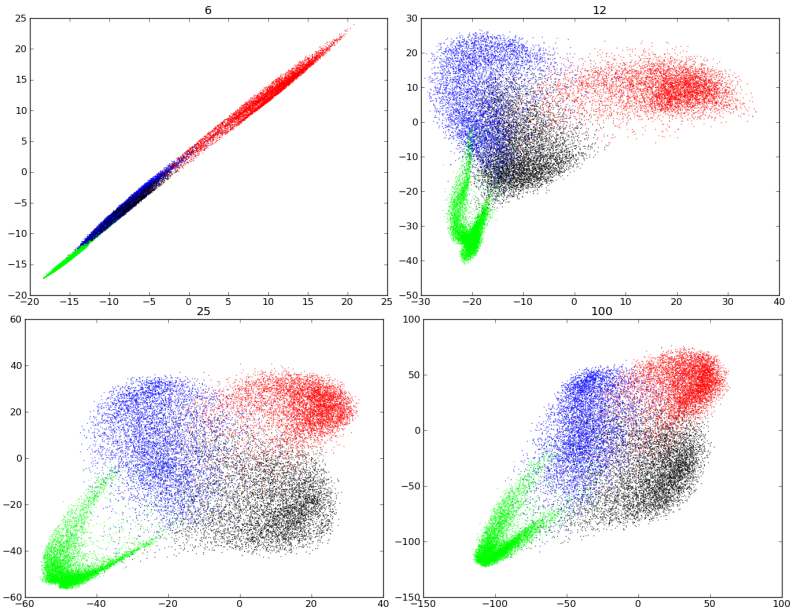


Figure 10.5: The distribution of digits processed by the deep net after 6, 12, 50 and 100 epochs of online RBM training. Red = 0; Green = 1; Blue = 2; Black = 3.

If we use the standard RBM learning method with the following meta-parameters:

- learning step size = 0.1
- sparseness factor = 0.001
- L1 regularisation factor = 0.001
- L2 regularisation factor = 0.0001
- no momentum

We get a clear ordered and separated output for the different digits while using a deep net of the following architecture: Input, 784 → Hidden1, 500 → Hidden2, 500 → Hidden3, 500 → Output, 2

Because we narrowed the output down to only 2 output neurones, their activity can be easily plotted, like can be seen in Figure 10.1. Also we visualised the weight vectors of the neurones in the fist hidden layer to give an impression how well the needed local features were formed by the RBM, like can be seen in Figure 10.2.

If we now switch from the standard layer by layer learning method to learning all layers at once we see that the quality of the output doesn't change. We still observe a good clustering of the different digits (see Figure 10.3).

Online learning with RBMs seems to be no obstacle for the use in RL.

One additional advantage for practical reasons is that one can observe the evolution of the distribution of the deep net, because from the first epoch on one gets an output of the whole net. Figure 10.5 shows such an development of the deep net output.

10.2 RBM LEARNING WITH ORDERED PATTERNS

RBMs are usually trained by presenting the whole training set as a batch or in mini-batches with randomly permutated order. This guarantees that the network *sees* the different patterns distributed evenly. For RL

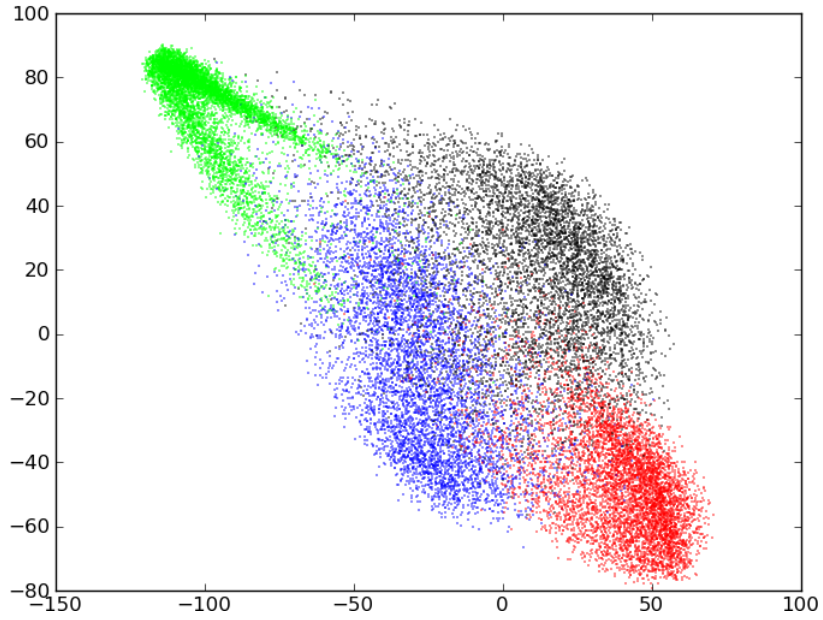


Figure 10.6:
The distribution of digits processed by the deep net after ordered online RBM training.
Red = 0;
Green = 1;
Blue = 2;
Black = 3.

this would change dramatically. Assuming a robot with a camera on top that tries to navigate via the camera picture would see one part of the lab for many frames before it moves and sees a different part of the lab again for many frames. The patterns would be presented to the deep net in an highly ordered fashion. Thus it would adapted to the new patterns and most likely would forget the old patterns. The questions

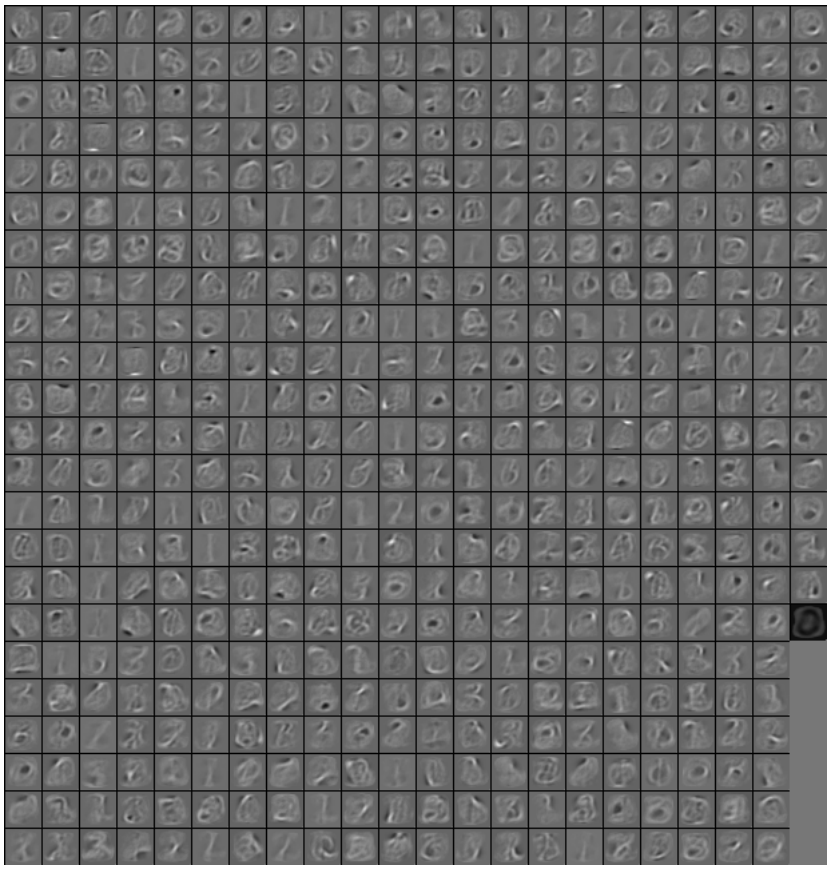


Figure 10.7:
The weight vectors of the first hidden layer for ordered online RBM training. Every *square* is the weight vector of one neurone represented in the 2D structure of the digit images (28×28).

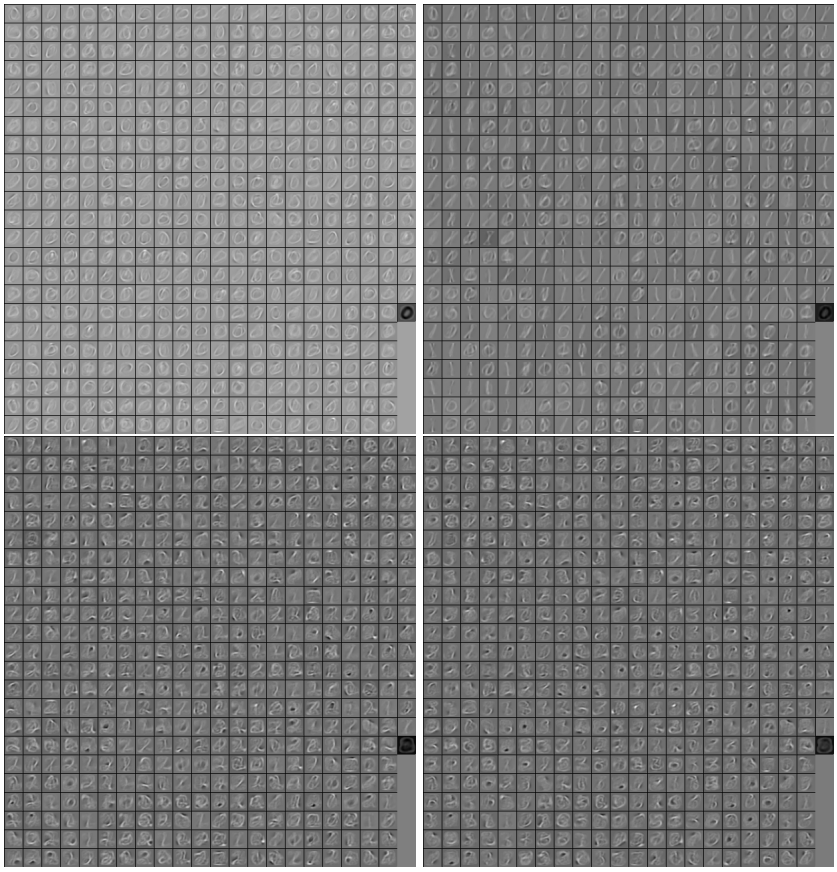


Figure 10.8: The weight vectors of the first hidden layer after the first four stages of RBM training.

therefore arises how one can train an RBM with highly ordered pattern sequences. We approached the solution of using a memory. For our tests we used a memory that can hold 6000 patterns. The memory should consist of a pattern distribution that represents the up to now presented patterns evenly. For achieving this, we introduce a counter variable c that keeps track of the number of patterns presented to the deep net from the environment. The probability that a new pattern is stored in the memory is then straight forward $p = \frac{MemSize}{c}$. If the memory size is exceeded, a random element is deleted from the memory.

By using a memory the data can be presented in a highly ordered fashion as long as the memory is big enough to store every main category of patterns. For the first 4 digits of the MNIST set a good solution is found with a memory size bigger than 3000 and the quality is no further improving with a memory size of about 5000 – 6000.

The training of the deep net was done in the following order:

- 50 epochs of training only 0s
- 50 epochs of training only 1s and the memory patterns of 0s
- 50 epochs of training only 2s and the memory patterns of 0s and 1s
- 50 epochs of training only 3s and the memory patterns of 0s, 1s and 2s
- 50 epochs of training the memory patterns of all 4 digits

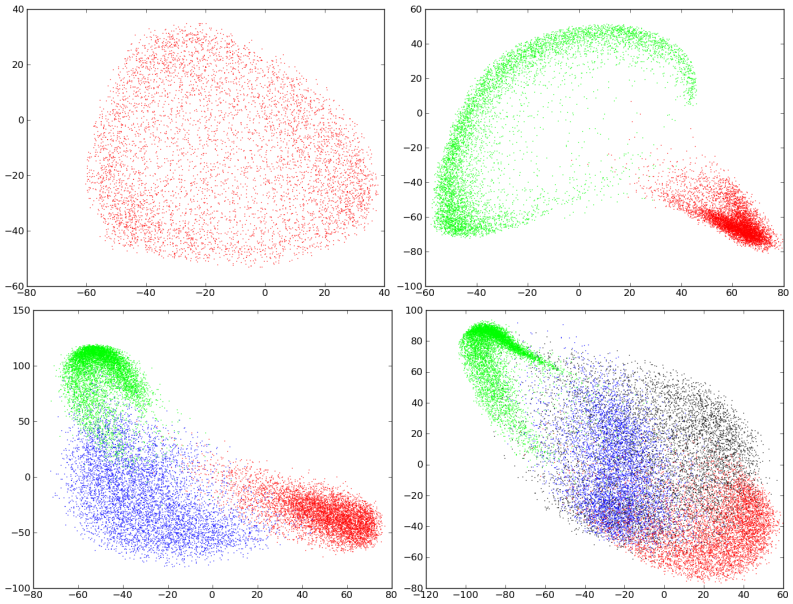


Figure 10.9:
The distribution of digits processed by the deep net after the first four stages of RBM training.
Red = 0;
Green = 1;
Blue = 2;
Black = 3.

As can be seen in Figure 10.6 the quality is not reduced by this kind of learning. Interestingly also the weight vectors of the first hidden layer look very similar (see Figure 10.7). With the use of the memory an evenly feature generation could be achieved in the RBM. Figure 10.8 shows how the features evolve. Every subfigure shows the status of the hidden units after one of the above stages. Figure 10.9 shows how the distribution of the net output evolves under this conditions.

This brief experiments suggest therefore that deep RBM nets can be trained in an RL setting without loss of efficiency.

10.3 POST TRAINING OF RBMS WITH PGPE

Deep nets can be fine tuned by PGPE efficiently

Now that we have all ingredients to make a deep RBM net for RL we use

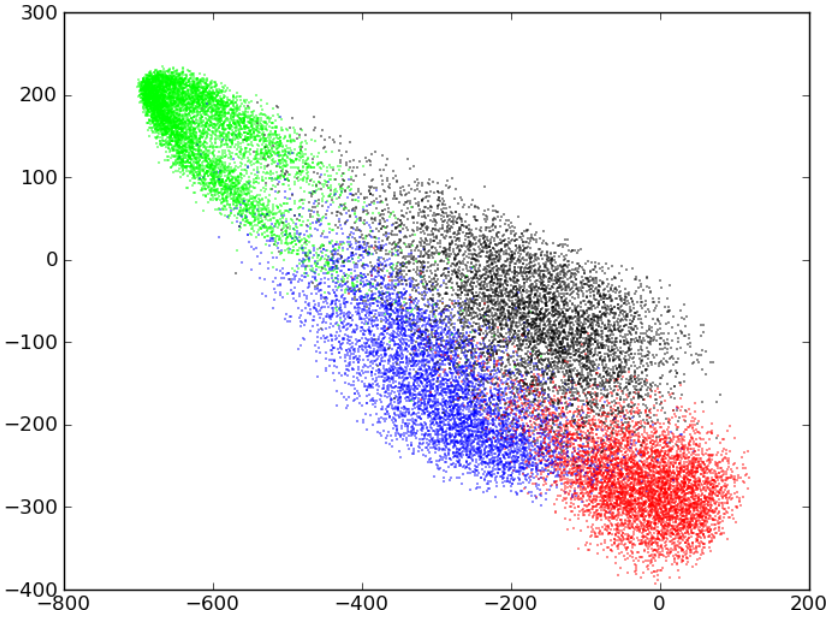


Figure 10.10:

The distribution of digits processed by the deep net after ordered online RBM training and PGPE training.
Red = 0;
Green = 1;
Blue = 2;
Black = 3.

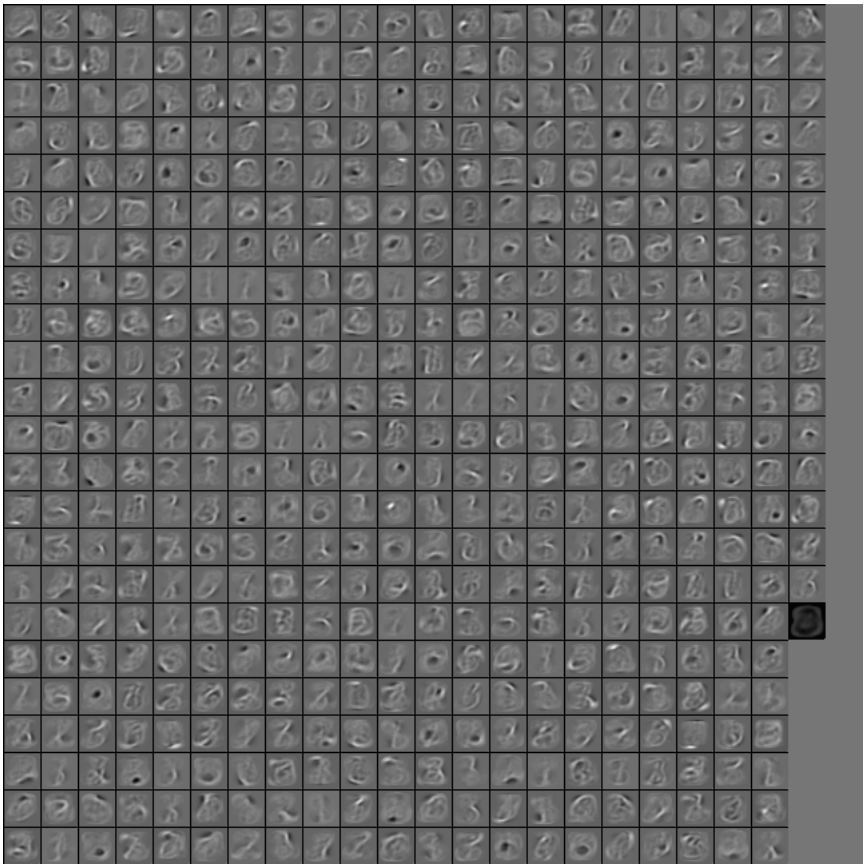


Figure 10.11: The weight vectors of the first hidden layer of the deep net after ordered online RBM training and PGPE training. Red = 0; Green = 1; Blue = 2; Black = 3.

the fact that PGPE can cope with NN with thousands of weights. Fine-tuning in RBM classification is usually done by using Backpropagation. This is possible even for a deep net, because the unsupervised RBM training step brought the weight near to a good optima so that the Backpropagation algorithm doesn't suffer from the vanishing gradient problem anymore. Because we want to test the usability of deep RBM nets in RL we don't use the labels if the MNIST set directly. We define a reward function that draws 100 random point pairs and calculates their distance. Pairs with equal label add their distance as negative number while pairs with different labels add their distance as positive number. This definition of reward reinforces output distributions there patterns from the same class are projected near together and patterns from different classes are projected far away. It also includes the in robotic RL tasks often encountered noise in the reward by choosing random pattern pairs. It is also a rather fast way of evaluating the quality of the deep net.

Figure 10.10 shows the improvement by retraining the deep net with PGPE. Interestingly the retraining with PGPE changes only very little in the weight vectors of the hidden neurones, like can be seen in Figure 10.11, while it makes drastically changes in the weight vectors of the output neurones.

10.4 DISCUSSION

It is certainly far fetched to claim that the MNIST test set is representative for testing if deep RBM nets are usable in RL tasks. On the other hand the MNIST data set is well known and its properties related to RBMs are well understood. So it is an easy way to test if in principle deep RBM nets are trainable in an online and ordered fashion. We hope (and are convinced) that the findings here hold also if a sophisticated e.g. optical RL task is executed in this manner. The optical target approach task of the FlexCube environment (see Section 8.2) would be an ideal candidate in our opinion to test this. This kind of experiments is a promising field of future work.

11 ARTIFICIAL GO PLAYER

In Artificial Intelligence frequent benchmarks are different kinds of board games. Artificial board-game players are therefore a widely-studied area of research. While board games like checker and chess have nowadays expert level artificial players, the Asian game of Go has eluded the development of artificial expert players. This is curious, because Go has quite simple rules that govern the game.

In this chapter an attempt is shown to provide a framework with which an expert level Go player could be learned by means of a Multi-Dimensional Recurrent Neural Networks (MDRNN) with Long Short-Term Memory cells (LSTM, in combination called MDLSTM) as controller and PGPE as learning method.

The choice of MDRNNs as controllers was made because they naturally handle the 2D data of the board and the rotational invariance of the game.

PGPE can train MDLSTM networks that seem best suited for an artificial Go player

PGPE is well suited to train the parameter rich MDLSTM networks with its standard and multiplicative weights as well as the recurrent weights. This chapter is—from the viewpoint of this thesis—the proof that PGPE can elegantly cope with this kind of complicated structure and is therefore well suited to learn recurrent networks with multiplicative weights without changing the learning algorithm in any way. For PGPE the weight matrix of MDLSTMs is a set of parameters like all the parameter sets we saw in the last chapters. Still PGPE learns the involved weight matrices very effectively.

11.1 THE BOARD GAME GO

Go is one of the least board-games with no artificial expert players

In contrast to Checkers, that has been recently solved completely by brute force, or Chess where expert-level artificial players are around for some time that achieve their skill by intelligently altered tree searches, Go has resisted an AI solution (at least in the expert-level). Monte

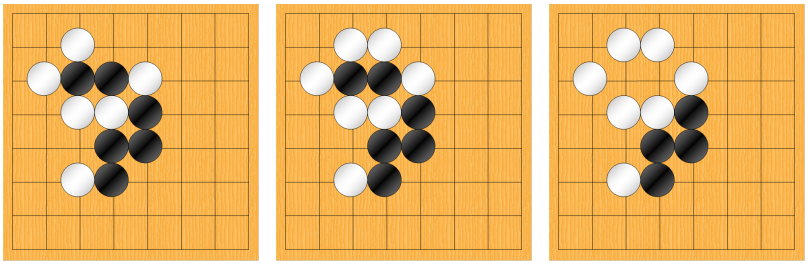


Figure 11.1: The figure shows a typical situation in Go. On the left hand side the white player has to make a move. He decides to capture a group of black stones (second figure) which is removed from the board (third figure)(source [Grüttner, 2008]).

Carlo Tree Search in combination with Reinforcement Learning (see e.g. [Bouzy and Chaslot, 2006; Gelly and Silver, 2007]) showed some success recently and may well be a good way to the final goal. Also a lot of research has been done using Neural Networks (see e.g. [Grüttner, 2008] for an overview).

For faster evaluation the capture game is used

Evaluating different methods by playing full Go games is much too time consuming. We therefore use the so called Capture Game. The Capture Game is a simplified version of Go that conserves some main strategies and is played out much faster.

How to play the Capture Game is well explained in [Grüttner et al., 2010]:

> The players alternately make a move by placing a stone on the board and try to enclose a group of opposing stones which is called capturing. There are some rules which specify where a stone can be placed on the game board, see [Grüttner, 2008] for details. The goal of Go is to capture more stones than the opponent player, Figure 11.1 illustrates a capturing scenario.
>
> The Capture Game, also called Atari-Go or Ponnuki-Go, has the same rules as Go, except passing is not allowed and the goal of the game changed: to win the game one has to be the first who captures at least one opposing stone.
>
> This kind of Go is often used for teaching new players. It is easier to learn the basic strategies, because the goal is achieved earlier and easier than for regular Go. Furthermore the Capture Game is a subproblem of Go and therefore can be used very well in computer science to compare different techniques.
>
> [...]
>
> The original game board consists of 361 (19×19) fields, but it is possible to use smaller board sizes to teach the game or to shorten the play time of the game. Nevertheless the main strategies stay the same. Therefore it is possible to train on a small board size and play on bigger ones.

The last part of the quote suggests that one can train MDRNNs on small board sizes and use this players on bigger boards.

Following publications on how Go and subtypes of Go are used as benchmarks for ML algorithms are suggested: [Bouzy and Chaslot, 2006], [Konidaris et al., 2002] and [Stanley and Miikkulainen, 2004].

11.2 MDRNN REPRESENTATION

MDRNNs are naturally suited for the 2 dimensional nature of the board game and can be scaled to bigger boards

In [Graves, 2007] a new Neural Network architecture, MDRNN, was developed that has been shown to be highly suited for problems with multi-dimensional inputs. It naturally represents the 2-dimensional board of Go. Unlike standard Recurrent Neural Networks (RNN), MDRNNs can easily learn to represent spacial structures and recognise patterns under translations and rotations by 90°. Also MDRNNs have the very useful property that they can be scaled to higher dimensions while preserving their behaviour [Schaul and Schmidhuber, 2009]. So

for instance one can train an MDLSTM on a small (e.g. 5×5) Go board and later play on a bigger one (e.g. 7×7). The MDLSTM will still show the same basic strategies on the bigger board that it learned on the small one.

An approach to train Neural Networks to play Go, that is relevant to our approach is the recent work of [Schaul and Schmidhuber, 2009] that has used black-box optimisation methods like CMA-ES [Hansen and Ostermeier, 2001], which unfortunately does not scale well to larger numbers of weights.

For the network representation of the agent we follow the terminology and discussion of [Graves, 2007; Grüttner, 2008; Schaul and Schmidhuber, 2009; Grüttner et al., 2010]. Multi-dimensional Recurrent Neural Networks (MDRNN) are able to use higher dimensional data directly, they can easily be scaled to bigger problem instances. MDRNNs were used successfully for vision [Graves et al., 2007], handwriting recognition [Liwicki et al., September 2007] and different applications of Go [Schaul and Schmidhuber, 2009], [Wu and Baldi, 2007], [Schaul and Schmidhuber, 2008], [Grüttner, 2008].

To fit this class of NN to the game Go we define the problem as two dimensional and represent the board directly as 2D input. To fit in the MDRNN scheme the width and height of the board are represented as sequences like described in [Grüttner et al., 2010]:

> Therefore we introduce *swiping* hidden layers which *swipe* diagonally over the board. The four directions that arise out of the described situation are the following: $D = \{\nearrow, \searrow, \nwarrow, \swarrow\}$.
>
> As exemplary hidden layer we describe the layer $h_{\nearrow}$, which swipes diagonally over the board from bottom-left to top-right, in detail. At each position (i,j) of the board we define the activation $h_{\nearrow(i,j)}$ as a function of the weighted input $in_{(i,j)}$ and the weighted activations of the previous steps $h_{\nearrow(i-1,j)}$ and $h_{\nearrow(i,j-1)}$ which leads to:
>
> $$h_{\nearrow(i,j)} = f(w_i * in_{(i,j)} + w_h * h_{\nearrow(i-1,j)} + w_h * h_{\nearrow(i,j-1)}) \tag{11.1}$$
>
> where f is a function (e.g. $f = tanh$). On the boundaries fixed values are used: $h_{\nearrow(i,0)} = h_{\nearrow(0,i)} = w_b$. An illustration of $h_{\nearrow}$ for the game Go can be found in Figure 11.2.

The output layer resembles all swiping directions by combining them. The output is described by:

$$out_{i,j} = g\left(\sum_{\diamond \in D} w_o * h_{\diamond(i,j)}\right) \tag{11.2}$$

g is the logistic function in our case, but could be any sigmoid function.

LSTM cells tackle the problem of occasionally high influence of far away stones in Go

Using Eq. (11.2) the network has access to the whole board. This access decays however by iterating over the recurrent connections. This problem is known as short-term memory effect of standard RNNs and solved by using so called Long Short-Term Memory (LSTM) cells [Graves, 2007]. LSTMs are using the aforementioned multiplicative weights as *gates* to protect the stored information over time. LSTMs have a

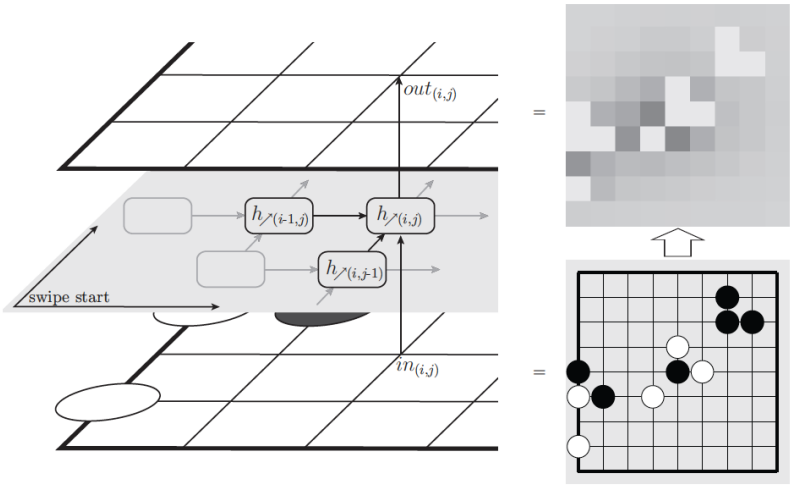


Figure 11.2: On the left hand side the schematic illustration of a MDRNN shows how the output consists of a swiping hidden layer in one direction. The right hand side illustrates the output (top) to the corresponding input (bottom). The brighter the square, the lower the preference to perform the corresponding move (source [Schaul and Schmidhuber, 2009]).

long history of successes, some examples are [Graves, 2007], [Grüttner, 2008], [Graves et al., 2007].

Combining LSTMs with MDRNNs by using swiping layers were named MDLSTM [Schaul and Schmidhuber, 2009].

[Grüttner et al., 2010] calculates the number of free parameters for this experiment with:

> With the given suggested architectures of MDRNNs (MDLSTMs) it follows that we have 12 (52) parameters which have to be evaluated. We will give a short calculation for MDRNNs. Our network consists of four (identical) hidden layers. The hidden layer is modelled by k neurones. Each neurone is connected with a weight w_o to the output layer and with two weights w_i to the input layer. Furthermore the neurones of the hidden layer are fully connected to each other which leads to k^2 weights which we call w_h. Additionally we have k weights w_b which are fixed and model the borders of the recurrent connections. All together we get $k + 2k + k^2 + k = 4k + k^2$ weights. The calculation for LSTMs is similar. With consideration of the additional weights of the LSTM-cells it follows that we have $16k + 5k^2$ weights.
>
> We decided to use two neurones which leads to 12 (52) parameters or weights.

Note the important fact that the number of parameters or weights is independent of the board size. It depends only on the amount of used hidden neurones. If we use a bigger board, the sequence the network has to compute is longer, everything else stays the same. This is the reason why one can learn on small boards and still play on bigger ones.

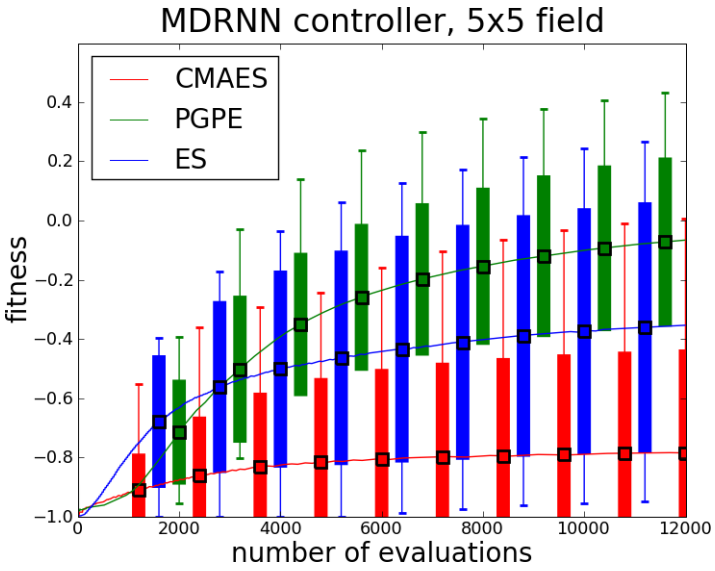


Figure 11.3: Performance of a MDRNN network on a 5×5 board. The plots give the fitness to every of the 12000 episodes as well as the standard deviation and min/max-values (average over 10 independent experiments). Source [Grüttner et al., 2010]

11.3 EXPERIMENTS AND RESULTS

In this chapter we compare PGPE with ES as well as CMA-ES. We compare the different algorithms on several small board sizes and with different MDRNNs. For ES we used $\mu = 5$ and $\lambda = 30$ with local mutation for standard ES. The Capture Game, the MDRNNs as well as ES and PGPE were used from the open-source Machine Learning library PyBrain [Schaul et al., 2010].

The experiments are set up in the following order:

- MDRNN network on 5×5 board
- MDRNN network on 7×7 board
- MDLSTM network on 5×5 board
- MDLSTM network on 7×7 board

We compare to the greedy player

A depth first search greedy Go player is used as *opponent* for evaluating the individuals. The Greedy Go Player follows simple rules in an orderly fashion. We define the liberties of a group of stones owned by the Greedy Player as p and the liberties of a group of stones owned by the opponent as q. The rules of the Greedy Player are executed in the following order:

- Count q.
- Check if group with $q = 1$ exists $\rightarrow$ capture, victory.
- Count p.
- If group with $p = 1$ exists (loss on next opponent move) place stone to enlarge group.
- If all groups have $p > 1$ choose a move which maximises the sum $p - q$.

The used implementation of the greedy player has the possibility to pass. The Capture Game, however, does not allow passing. We decided therefore in [Grüttner et al., 2010] to do a random move instead.

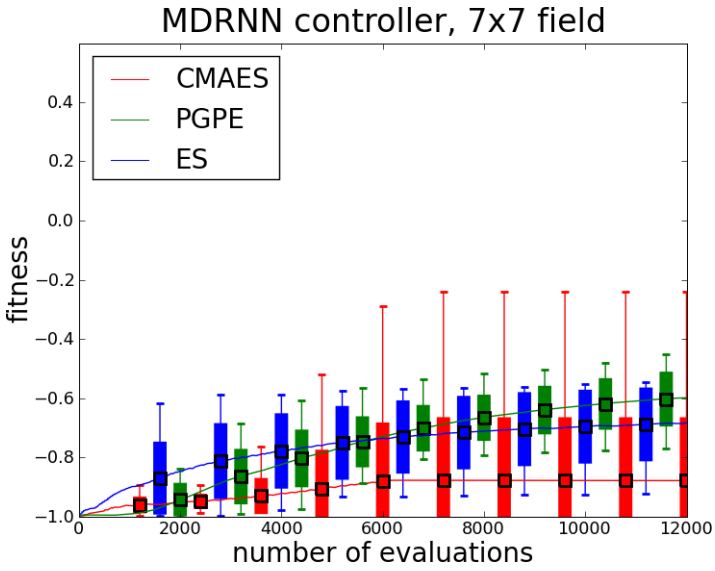


Figure 11.4: Performance of a MDRNN network on a 7×7 board. The plots give the fitness to every of the 12000 episodes as well as the standard deviation and min/max-values (average over 10 independent experiments). Source [Grüttner et al., 2010]

Every agent played 40 games against the greedy player. The outcomes were averaged over the 40 games and scaled between -1 (for never wins) and 1 (for always wins).

Figures 11.3 to 11.6 illustrate the results published in [Grüttner et al., 2010]:

> As we can see PGPE mostly converges faster than ES and CMA-ES. Primarily with the increasing of the number of parameters the advantages of PGPE towards ES increase.
>
> Nevertheless neither ES nor PGPE has converged within the 12000 episodes to the maximum fitness value 1. This holds for the best individuals of each generation, too. In our experiments the best result of a single run of PGPE converges to 0.5 which is equivalent to a victory rate of 75% (see Figure 11.5). While 75% is already a good result (better

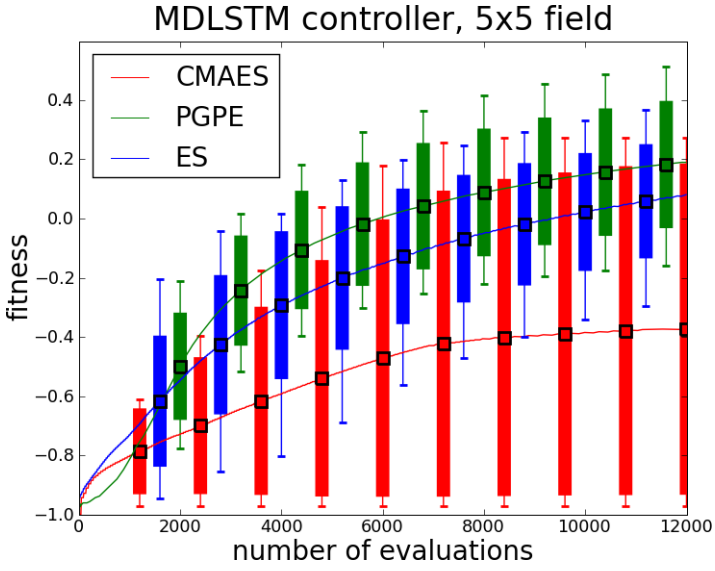


Figure 11.5: Performance of a MDLSTM network on a 5×5 board. The plots give the fitness to every of the 12000 episodes as well as the standard deviation and min/max-values (average over 10 independent experiments). Source [Grüttner et al., 2010]

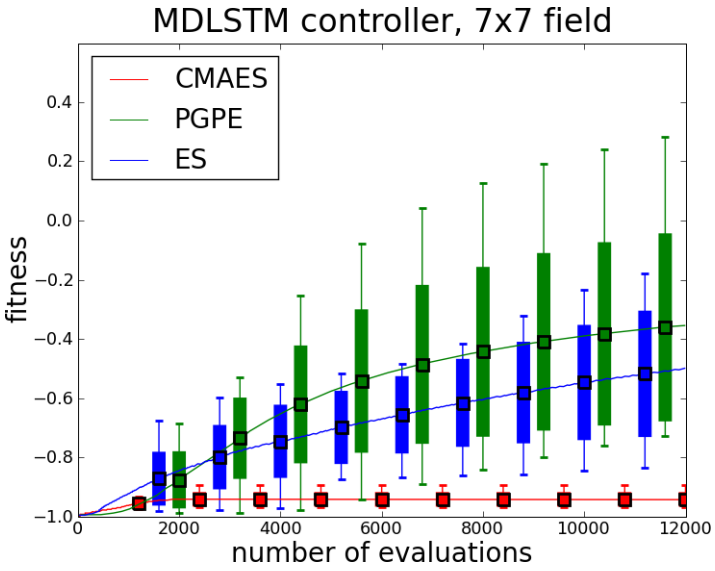


Figure 11.6: Performance of a MDLSTM network on a 7×7 board. The plots give the fitness to every of the 12000 episodes as well as the standard deviation and min/max-values (average over 10 independent experiments). Source [Grüttner et al., 2010]

than human beginner level) ongoing learning could (and should) still improve the results.

Furthermore the use of MDLSTMs leads to better results than MDRNNs. This strength of MDLSTMs is accompanied by a long training time towards MDRNNs. Our observations are similar to [Graves, 2007], [Schaul and Schmidhuber, 2009], [Grüttner, 2008].

Another fact we could read from our resulting plots is a big standard deviation. This observation leads to the suggestion that the standard meta parameters for PGPE and ES are not optimal for this problem domain and that meta parameters that favour a more thorough exploration combined with longer learning cycles should provide better and more stable results.

The last issue is seen as promising future work also from the stand point of this thesis.

12 PART SUMMARY AND CONCLUSION

In Chapter 8 we presented several robotics benchmarks: From test functions like the Rastrigin and the Ackley function, over simple robotic tasks like the inverted pendulum or enhanced cartpole balancing, to sophisticated tasks like robust balancing of a humanoid robot or the grasping of several objects with a 7 DoF robot arm.

The benchmark functions served to demonstrate the properties of PGPE and MultiPGPE, because these benchmarks are quick to calculate and can easily be scaled in their dimensionality. We verified our findings on the inverted pendulum.

On the enhanced cartpole benchmark, we demonstrated that PGPE outperforms NAC, SPSA, ES and Reinforce. We showed that this cannot be explained by the lower perturbation frequencies alone.

On the Johnnie and the CCRL Benchmark, we finally showed that not only is PGPE able to learn sophisticated tasks—like robust standing or grasping of objects—model-free from scratch, but it also does so faster and more reliable then all other compared methods.

In Chapter 9, PGPE's ability to cope with different kinds of search spaces present in arbiter PUFs was examined. We highlighted that PGPE can also be used for PUFs with non-differentiable models. We investigated the performance of PGPE in the cryptoanalysis of electrical PUFs.

In Chapter 10, we showed how to combine RBMs with PGPE by first demonstrating how to learn RBMs unsupervised in an online-fashion, and then how to fine tune the resulting network by means of RL, using PGPE.

In Chapter 11, it was shown that PGPE can also be utilised to train recurrent networks, even if they possess multiplicative weights. PGPE is also able to train MDRNNs as well as the so called MDLSTMs. We showed a way of learning the asian board game Go with this two multi dimensional recurrent network architectures.

Part IV

CONCLUSION AND FUTURE WORK

13 CONCLUSION AND SUMMARY

13.1 CONCLUSION

From the intensive study of parameter exploring policy gradients, especially from the studies of PGPE we have gathered a wide variety of observations. The most visible difference to standard PG methods was that PGPE could be used with the unchanged standard meta-parameters for nearly all problems. Especially in robotic tasks there standard NN were used as controllers the set of standard meta-parameters was always optimal. This makes PGPE very useful as "out of the box" method.

PGPE has very few meta-parameters that could be used with the standard settings in nearly all problems

Also parameter-based exploration led to a very robust convergence behaviour, comparable to the robustness of evolutionary methods. Because the gradient is normalised in a way so that the hypothesis nether exceeds the sampling range, we never observed runaway gradients or situations of wild oscillations that can be observed if standard PG methods are used with ill chosen meta-parameters. In very few cases we encountered problems with premature convergence. In this rare cases setting the two step sizes to a smaller value than standard was enough to cope with the too early convergence.

PGPE is very robust

As a guideline for setting the 3 meta-parameters we propose the following procedure: Set the step size for the μ adaptation $\alpha_\mu = 0.2$. Set the step size for the σ adaptation (the exploration adaptation) $\alpha_\sigma = 0.1$. Finally set the starting σs to 20% of the as useful interpreted search interval for each dimension or to the standard deviation of the assumed solution space if that is known. For standard NN, MLP for example, with logistic or tanh output function a starting σ of 2.0 was found to be optimal. If the results show that the algorithm converges too fast to an suboptimal solution, decrease the two step sizes accordingly. In all cases the two step sizes were best chosen if the ratio of 2 was kept. So one could define only one step size and derive the two single step sizes from that without the danger of using an algorithm that shows an ill convergence behaviour.

How to set the 2-3 meta-parameters for PGPE

Interestingly the estimation for the starting σs is exactly the same as for the strategy parameters in ES that fulfil the same role of adapting the exploration.

One general observation from our experiments was that the longer the episodes of the investigated learning task, the more PGPE outperformed policy gradient methods. This was also true for FD and Evolution. So generally one could broaden the statement to "the longer the episodes of the investigated learning task, the more methods with parameter-based exploration outperformed methods with action-based exploration. This effect is a result of the variance increase in REINFORCE gradient estimates with the number of time steps per episode. As most interesting real-world problems require much longer episodes than in our experiments, this improvement can have a strong impact. For example, in biped walking [Benbrahim and Franklin, 1997], object manipulation [Peters and Schaal, 2006] and other robot control

The longer the episodes the more parameter exploring methods outperform standard PG methods

tasks [Müller et al., 2007] update rates of hundreds of Hertz and task lengths of several seconds are common.

Action based normal distributed exploration is not effective in complex RL tasks!

As a summary of the standard PG versus PGPE comparison we claim that exploration by adding normal-distributed noise to the actions is not effective. At least a state dependent exploration like suggested by [Rückstieß et al., 2008] should be used so that at least a new policy is tested with every episode. We think that one should go even one step further and don't derive the parameter changes by backpropagating changed actions through the controller but directly derive the parameter changes. This is a straight forward and robust solution (like our experiments and several publications by other researchers show, e.g. [Miyamae et al., 2010; Fasel et al., 2010])

Another observation was that symmetric sampling has a stronger impact on tasks with more complex, multi-modal reward functions, such as in the FlexCube walking task.

Parameter based exploration has higher sampling complexity

A possible objection to parameter-based exploration in general and to PGPE is that the parameter space is generally higher dimensional than the action space, and therefore has higher sampling complexity. However, standard policy gradient methods in fact train the same number of parameters—in PGPE they are just trained explicitly instead of implicitly. Additionally, results of the last years [Riedmiller et al., 2007a] as well as our results indicate that this drawback was overestimated in the past. In this thesis we presented experiments where PGPE successfully trained a controller with more than 1000 parameters, but only 11 output/action dimensions. Another issue is that PGPE, at least in its basic and most effective form, is episodic, because the parameter sampling is carried out once per history. This contrasts with policy gradient methods, which can be applied to infinite horizon settings as long as frequent rewards can be computed. IHPGPE should be an alternative to tackle this problem, but firstly we estimate that IHPGPE is not so effective than PGPE on a comparable episodic task and the practicability hasn't been shown in experiments up to now. So the main field of PGPE will at least in the near future be episodic in nature (like all other parameter-based methods).

PGPE is episodic

We see PGPE as convergence of the different related fields. In Evolution Strategies the tendency to use derandomised strategies is obvious. With derandomised ES the change of the exploration is derived directly from the experience gathered in the last samples (like in PGPE).

Natural Gradients need a batch of samples that tends to be rather big in high-dimensional problems. While the gradient itself is more direct pointed to the optimum, we realised that this advantage is superseded by the drawback of having a batch. One can use moving batches and important mixing [Sun et al., 2009] to weaken this drawback to some extend, but in our experience to be able to use an update every two samples is a much bigger advantage then using the natural gradient. This is especially true for high-dimensional problems. Also the natural gradient made the convergence process less robust in our experience and robustness is one key feature of PGPE. However, recent studies of [Miyamae et al., 2010] and [Wierstra et al., 2008b] show that there are indeed examples there using the natural gradient is favourable.

Using Rprop [Riedmiller and Braun, 1993] has the same drawback than using the natural gradient, it needs a rather big batch of samples.

Because the PGPE gradient does not tend to get stuck in local minima anyway, using the Rprop gradient brings in our optinion no advantages in this case.

Using a covariance matrix is of a big advantage and would fit easily in the PGPE framework (the σ vector is just replaced by the Σ covariance matrix). Sadly the covariance matrix grows quadratically with the problem dimension and therefore is not suitable for the problem classes PGPE aims for. Again [Miyamae et al., 2010] and [Wierstra et al., 2008b] show that for problem instances with small dimensions using a covariance matrix can be very effective.

PGPE is robust as ES, fast as PG and simple like FD

PGPE therefore inherited the robustness Evolution is renown for, due to the parameter based sampling. It inherits further more the fast gradients from PG while being simple or even simpler then a standard finite difference algorithm like SPSA.

In our opinion PGPE is the perfect merge of this 3 fields of RL and is one of the best today existing algorithms for high-dimensional reinforcement learning. PGPE has therefore drawn also attention in the ML community like several citations in journals and high rated conference publications show [Du et al.; Miyamae et al., 2010; Boularias, 2010; Fasel et al., 2010; van Hasselt, 2011; Zhao et al., 2011a,b].

13.2 SUMMARY

We have motivated why RL is important in robotics in Chapter 1 and highlighted why especially Policy Gradients are useful. We motivated the use of parameter based exploration and gave considerations of the advantages and drawbacks.

In Chapter 2, we explained the MDP/POMDP setting RL takes place in. We highlighted the difference between episodic and ongoing tasks/rewards.

In Chapter 3, we explained the classical RL methods SARSA and Q-Learning, artificial evolution with the specific algorithms GA and ES as well as giving some insights in SANE, ESP and CoSyNE. We showed how PG methods work starting with FD over SPSA to REINFORCE and gave some insights into GPOMDP and NAC.

We highlighted how exploration works in the different kinds of ML methods and we identified the two principal ways of exploration in PG: exploration by perturbing the actions of the agent and exploration by perturbing the parameters of the agent. We gave a short discussion in several passages explaining why exploring in parameter space is the better exploration strategy for the field of RL we are interested in.

In Chapter 5, we derived the PGPE algorithm and its variants MultiPGPE and IHPGPE from the general framework of episodic reinforcement learning in a Markovian environment. In doing so, we highlighted the differences between PGPE and standard policy gradient methods such as REINFORCE. We showed how to use a baseline approach or how to use symmetric sampling, if possible, for all variants of PGPE. We also provided a reward normalisation scheme and simplifications of the algorithms for every PGPE variant where this was useful. An algorithm in pseudo code was provided for the best instance of every PGPE variant.

We showed that MultiPGPE copes with the problem of PGPE that a normal distributed search is sometimes not sufficient in highly multimodal search spaces. It uses a mixture of Gaussians to counter this problem.

IHPGPE negates the restriction of PGPE to be used only in episodic tasks by using asynchronous parameter perturbations over time and generating continually gradients.

Furthermore, we evaluated the differences between PGPE, SPSA, ES and REINFORCE and discussed the properties of PGPE in Chapter 6.

To transform SPSA into PGPE, we identified three changes that are required. First, we changed the uniform sampling of perturbations to a Gaussian sampling, with the effect that the method is less likely to get stuck in a local optima, and correlated parameters are easier to track. Second, the finite difference gradient was replaced correspondingly by the likelihood gradient with a speed up in convergence time. Third, the variances of the perturbations were turned into free parameters and trained with the rest of the model making it possible to learn effectively in problems with parameters that are very different in their order of magnitude.

We also examined the effect of two modifications that bring ES closer to PGPE. First, we switched from standard ES to derandomised ES, resembling the gradient based variance updates found in PGPE. Then we changed from population based search to following a likelihood gradient resulting in a speed up in convergence time.

We also changed the exploration in REINFORCE by transferring the explorational noise to the parameters of the controller. This resulted in a speed up in convergence time that couldn't be explained by the less frequent perturbations alone.

We discussed the central search property of PGPE and that it can be a drawback, if several good optima lie in opposite directions. We showed that MultiPGPE can overcome this problem.

We discussed the flat optima property of PGPE and the drawback that one cannot control to which extend the algorithm tends to converge to flat shallow optima instead of going for narrow deep ones. We showed that MultiPGPE can overcome also this issue by using different numbers of Gaussians per mixture.

We also discussed that PGPE tends to overestimate the gradients reach, if the global structure leads constantly in a certain direction, eventually moving to fast in the parameter space and overlooking the global optima.

Overall, PGPE and its variants are mathematically soundly based on the principals of general Reinforcement Learning (in contrast to ES), more flexible than Finite Difference methods and provide a less noisy gradient compared to Policy Gradient methods.

We presented in Chapter 8 several robotic benchmarks: from test functions like the Rastrigin and the Ackley function, over simple robotic tasks like the inverted pendulum or the enhanced cartpole balancing to sophisticated tasks like the robust balancing of an humanoid robot or the grasping of several objects with an 7-DoF robot arm.

On the benchmark functions, we could demonstrate the properties of PGPE and MultiPGPE because this benchmarks are fast to calculate and

can easily be scaled in their dimensionality. We verified our findings on the Inverted Pendulum.

On the enhanced Cartpole benchmark, we have demonstrated that PGPE outperforms NAC, SPSA, ES and REINFORCE and that this cannot be explained with less perturbation frequencies alone.

On the Johnnie and the CCRL Benchmark, we have finally shown that not only PGPE is able to learn sophisticated tasks like robust standing or grasping of objects model-free from scratch, but it also does so faster and more reliable then all other compared methods.

We furthermore have shown in Chapter 9 how PGPE can cope with different kinds of search spaces that are present in arbiter PUFs today. We highlighted that PGPE can also be used for PUFs with non-differentiable models. We investigated the performance of PGPE in the cryptanalysis of electrical PUFs.

In Chapter 10, we showed how to combine RBMs with PGPE by first showing how to learn RBMs unsupervised in an online-fashion and then how to fine tune the resulting network by RL with PGPE.

In Chapter 11, it was shown that PGPE can also learn recurrent networks even ones with multiplicative weights. PGPE is also able to learn MDRNNs or the so called MDLSTMs. We showed a way of learning the asian board game Go with this kind of architecture.

14 FUTURE WORK

We investigated a lot of topics and questions that couldn't be all answered in this thesis and resemble therefore topics of future work.

First of all, there are the unanswered questions regarding PGPE and its variants itself. One issue for future work would be to establish whether Williams' local convergence proofs for REINFORCE can be generalised to PGPE. Another would be to combine PGPE with recent improvements in policy gradient methods, such as natural gradients and base-line approximation [Peters et al., 2005]. However, a very nice work has been published recently that shows how to apply optimal baselines to PGPE and that this optimal baseline versions of PGPE performs better than standard PGPE Zhao et al. [2011a].

MultiPGPE and Infinite Horizon PGPE are easy to combine and should improve the Infinite Horizon PGPE in the same way it improved the raw PGPE algorithm. But first, Infinite Horizon PGPE should be verified experimentally. Here, the FlexCube environment with the Optical Target Approach Task seems to be very well suited for future investigations.

Glasmachers et al. [2010] has developed a method called Important Mixing (IM) in a PGPE related, multi-sample/population-based method called Natural Evolution Strategies (NES, Wierstra et al. [2008b]). IM reuses old samples in a way that keeps the current distribution intact, saving the evaluation for this sample. This results in a great speed up for NES .

Because in PGPE one gets a *band* of symmetric samples, it is obvious that PGPE could be improved by this technique too. A merging of symmetric sampling and IM is therefore a very promising field of future work for NES and PGPE.

MultiPGPE introduces a new meta parameter, the step size α_π for adjusting the mixing coefficients π_i^k. This new meta parameter seems to be more dependent on the actual problem than the other meta parameters already present in PGPE. This is actually the major drawback of MultiPGPE, since PGPE convinces with its very limited number of meta parameters that are also very stable over a wide field of problems. With MultiPGPE, we increase the set of meta parameters from 3 to 4, and the new parameter is not as stable. Therefore, an important issue for future work is to show how this step size can be estimated for certain problem domains.

It also has to be assessed how much impact this enhancement of PGPE has on real-world applications, now that the theoretical benefits have been demonstrated.

State Dependent Exploration (SDE) as mentioned in Chapter 3.2.4 is an interesting exploration alternative. To date, the two methods haven't been compared directly, but from personal communication we assume that SDE is superior in flat controllers with a low action to parameter dimension ratio, while PGPE is superior in deep controller architectures with a higher action to parameter dimension ratio. Exploring the

properties of the different exploration and learning strategies is in our opinion an interesting field of future work.

Secondly, there are also still open questions regarding the problems we have solved with PGPE. A problem that can be observed in the FlexCube environment is that due to the low resolution of the retina the maximum distance of the food source to be detected reliably in the picture is rather low. A fovea approach could counter this issue. The optical target approach task itself was not learned to this day and remains also future work.

For general Go and other real-world problems, there are a lot of open problems. First of all, more episodes are necessary. Furthermore, future applications with a stack of MDRNNs are possible as suggested in [Schaul and Schmidhuber, 2008]. For such applications PGPE seems appropriate.

An interesting future application for Go would be the research of the influence of PGPE on scaling MDRNNs as well as determining the best ratio between game board size and PGPE setup (especially using non-standard meta parameters like smaller step sizes for more thorough exploration and better end behaviour). Besides, PGPE could be used for relearning the scaled controllers.

As suggested for ES in [Grüttner, 2008], one could use Co-Evolution to further improve the PGPE results. For PGPE, this would mean the fitness is evaluated not only against the Java-Player but also against the best up to now observed learned controller and the current mean parameter set controller.

Furthermore, Lazy Evaluation could be used to speed up learning in the Go scenario. This approach adapts the number of games per evaluation depending on the fitness. In the beginning, 3 or 4 games could be enough for an evaluation step, whereas at the end of convergence, it would be profitable to use up to 100 games to calculate the fitness value in order to distinguish the slight changes in performance at that point of learning.

Like already briefly mentioned, the high standard deviation suggests that a higher rate of exploration would be favourable for the overall performance and stability of the Go agents. For PGPE, this would mean to decrease the values for the two step sizes that are normally set to $\alpha_\mu = 0.2$ and $\alpha_\sigma = 0.1$. More thorough exploration comes at the price of longer convergence time. Therefore, the already mentioned Lazy Evaluation approach is furthermore helpful.

We suggest for a future investigation of the Go scenario to use PGPE on a 5x5 field with MDLSTMs, in a combined Co-Evolution evaluation setting with optimised meta parameters and incremental scaling up the field size and retraining, till full field size and optimal game behaviour for this setting is reached.

We hope that we can investigate some of the questions ourself, but we hope even more that other researchers may find some topics worth investigating.

Part V

APPENDIX

A

PYBRAIN

PyBrain is a versatile machine learning library for Python. Its goal is to provide flexible, easy-to-use yet still powerful algorithms for machine learning tasks, including a variety of predefined environments and benchmarks to test and compare algorithms. Implemented algorithms include Long Short-Term Memory (LSTM), policy gradient methods, (multidimensional) recurrent neural networks and deep belief networks. [Schaul et al., 2010] PyBrain is an open source product. More details can be found at [pyb].

We included all variants of PGPE from the former Chapters in PyBrain as well as all described robotic benchmarks and RL test functions.

A.1 UDP INTERFACE

To prevent a slowdown in simulation time we started by dividing the graphical output of the simulation from the physical simulation itself. The goal was to have a server that calculates the simulation as fast as possible and an external client that can connect on the server to observe the progress of the learning. Therefore we developed an UDP-Interface that could connect a client to a server via UDP and transmit *numpy* arrays.

The UDPServer waits until at least one client is connected. Then it sends a list to the connected clients (can also be a list of *numpy* arrays!). Several clients can be connected to the server and the same data is sent to all clients.

Options for the constructor for the server are the server IP and the starting port (2 adjacent ports are used). The UDPClient tries to connect to a UDPServer till the connection is established. The client then receives data from the server and parses it.

The options for the client constructor are server-, client IP and the starting port (again 2 adjacent ports will be used).

The requirements for server and client are the packages *socket* and *scipy*. FlexCube environment, all ODE environments and the ShipSteering environment are examples that use the UDP connection.

A.2 FLEXCUBE AND VIEWER

The FlexCube environment was included in the RL environments under pybrain/rl/environments/. The flexcube folder contains the files masspoint.py (the basic class of a masspoint and its methods for use in a spring particle system), objects3d.py (defines basic forms for OpenGL), sensors.py (defines all sensors that can be used in the FlexCube environment), tasks.py (defines all task definitions) and viewer.py (the OpenGL viewer that can be connected to the server via the above described UDP connection.)

```
# The server class
class UDPServer(object):
  def __init__(self, ip="127.0.0.1", port="21560", buf="1024"):
    #Socket settings
    self.host = ip
    self.inPort = eval(port) + 1
    self.outPort = eval(port)
    self.buf = eval(buf) #16384
    self.addr = (self.host, self.inPort)

    #Create socket and bind to address
    self.UDPInSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    self.UDPInSock.bind(self.addr)

    #Client lists
    self.clients = 0
    self.cIP = []
    self.addrList = []
    self.UDPOutSockList = []

  # Adding a client to the list
  def addClient(self, cIP):
    self.cIP.append(cIP)
    self.addrList.append((cIP, self.outPort))
    self.UDPOutSockList.append(socket.socket(socket.AF_INET, socket.SOCK_DGRAM))
    self.clients += 1

  # Listen for clients
  def listen(self):
    if self.clients < 1:
      self.UDPInSock.settimeout(60)
      try:
        cIP = self.UDPInSock.recv(self.buf)
        self.addClient(cIP)
      except: pass
    else:
      # At least one client has to send a sign during 2 seconds
      self.UDPInSock.settimeout(60)
      try:
        cIP = self.UDPInSock.recv(self.buf)
        newClient = True
        for i in self.cIP:
          if cIP == i:
            newClient = False
            break
        #Adding new client
        if newClient:
          self.addClient(cIP)
      except:
        self.clients = 0
        self.cIP = []
        self.addrList = []
        self.UDPOutSockList = []

  # Sending the actual data to all clients
  def send(self, arrayList):
    sendString = repr(arrayList)
    count = 0
    for i in self.UDPOutSockList:
      i.sendto(sendString, self.addrList[count])
      count += 1
```

Listing 1: The UDP Server Class

```
# The client class
class UDPClient(object):
  def __init__(self, servIP="127.0.0.1", ownIP="127.0.0.1", port="21560", buf="
      2048"):
    #UDP Sttings
    self.host = servIP
    self.inPort = eval(port)
    self.outPort = eval(port) + 1
    self.inAddr = (ownIP, self.inPort)
    self.outAddr = (self.host, self.outPort)
    self.ownIP = ownIP
    self.buf = eval(buf) #16384

    # Create sockets
    self.createSockets()

  # Listen for data from server
  def listen(self, arrayList=None):
    # Send alive signal (own IP adress)
    self.UDPOutSock.sendto(self.ownIP, self.outAddr)
    # if there is no data from Server for 10 seconds server is propably down
    self.UDPInSock.settimeout(60)
    data = self.UDPInSock.recv(self.buf)

    try:
      try:
        arrayList = eval(data)
        return arrayList
      except:
        print "Unsupported data format received from", self.outAddr, "!"
        return None

    except:
      print "Server has quit!"
      return None

  # Creating the sockets
  def createSockets(self):
    self.UDPOutSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    self.UDPOutSock.sendto(self.ownIP, self.outAddr)
    self.UDPInSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    self.UDPInSock.bind(self.inAddr)
```

Listing 2: The UDP Client Class

The code is by far to extensive to state it here. We therefore refer the reader to the open source code at [pyb]. For details on the implemented simulation, the available sensors and the viewer please refer to Chapter 8.

A.3 ODE AND VIEWER

The Open Dynamics Engine for Python was mainly included into PyBrain by Thomas Rückstieß. He also implemented the first OpenGL view of the ODE environments. Martin Felder and Thomas Rückstieß also implemented a parser that can parse simple rigid body and joint definitions into the XML code required by ODE. As part of this thesis, the OpenGL viewer was enhanced by the possibility of colouring different body elements. It was integrated in the UDP connection framework which was already used for the FlexCube. Several instantiations of robots in ODE were generated, like the Johnnie and the CCRL framework. Therefore also new sensors were implemented like poise sensors. As before, the code is to extensive to state here and we therefore refer again to [pyb].

A.4 PGPE IMPLEMENTATIONS

The PGPE variants fitted very well in the PyBrain framework. Especially because a class of algorithms called optimisation methods already existed in PyBrain (created by Tom Schaul and Daan Wierstra). In this class of algorithms Thomas Rückstieß and Tom Schaul had already created a Finite Difference class (see Listing 3) that we only had to adapt slightly to our requirements to have a basis for PGPE:

So for PGPE we could inherit from FiniteDifferences with overwriting the parts that define PGPE in the field of FD. That is for the constructor the exploration type (global for one standard deviation for all parameter perturbations, local for a standard deviation per parameter), the learning rates for the μ and the σ update and the initial value of the sigmas. Also the extra memory for the standard deviations has to be allocated and a baseline has to be defined. See Listing 4.

The perturbation method has to be overwritten because we now use a normal distributed perturbation with the defined σs as standard deviation. In contrast the learning method has to be completely rewritten. First of all we generate the two symmetric samples, then we use the evaluations for generating a reward normalised log likelihood gradient. See Listing 5.

```
__author__ = 'Thomas Rueckstiess, ruecksti@in.tum.de, Tom Schaul'

from scipy import ones, zeros, dot, ravel, random
from scipy.linalg import pinv

from pybrain.auxiliary import GradientDescent
from pybrain.optimization.optimizer import ContinuousOptimizer

class FiniteDifferences(ContinuousOptimizer):
    """ Basic finite difference method. """
    epsilon = 1.0
    gamma = 0.999
    batchSize = 10

    # gradient descent parameters
    learningRate = 0.1
    learningRateDecay = None
    momentum = 0.0
    rprop = False

    def _setInitEvaluable(self, evaluable):
        ContinuousOptimizer._setInitEvaluable(self, evaluable)
        self.current = self._initEvaluable
        self.gd = GradientDescent()
        self.gd.alpha = self.learningRate
        if self.learningRateDecay is not None:
            self.gd.alphadecay = self.learningRateDecay
        self.gd.momentum = self.momentum
        self.gd.rprop = self.rprop
        self.gd.init(self._initEvaluable)

    def perturbation(self):
        """ produce a parameter perturbation """
        deltas = random.uniform(-self.epsilon, self.epsilon, self.numParameters)
        # reduce epsilon by factor gamma
        self.epsilon *= self.gamma
        return deltas

    def _learnStep(self):
        """ calls the gradient calculation function and executes a step in
            direction of the gradient, scaled with a small learning rate
            alpha. """

        # initialize matrix D and vector R
        D = ones((self.batchSize, self.numParameters))
        R = zeros((self.batchSize, 1))

        # calculate the gradient with pseudo inverse
        for i in range(self.batchSize):
            deltas = self.perturbation()
            x = self.current + deltas
            D[i, :] = deltas
            R[i, :] = self._oneEvaluation(x)
        beta = dot(pinv(D), R)
        gradient = ravel(beta)

        # update the weights
        self.current = self.gd(gradient)
```

Listing 3: The Finite Difference Class

```
class PGPE(FiniteDifferences):
  """ Policy Gradients with Parameter Exploration (ICANN 2008)."""
  #:exploration type
  exploration = "local"
  #: specific settings for sigma updates
  learningRate = 0.2
  #: specific settings for sigma updates
  sigmaLearningRate = 0.1
  #: Initial value of sigmas
  epsilon = 2.0
  #:lasso weight decay (0 to deactivate)
  wDecay = 0.0
  #:momentum term (0 to deactivate)
  momentum = 0.0
  #:rprop decent (False to deactivate)
  rprop = False

  def _additionalInit(self):
    if self.sigmaLearningRate is None:
      self.sigmaLearningRate = self.learningRate
    self.gdSig = GradientDescent()
    self.gdSig.alpha = self.sigmaLearningRate
    self.gdSig.rprop = self.rprop
    #Stores the list of standard deviations (sigmas)
    self.sigList = ones(self.numParameters) * self.epsilon
    self.gdSig.init(self.sigList)
    self.baseline = None
```

Listing 4: The PGPE Class Constructor

```
  def perturbation(self):
    """ Generate a difference vector with the given standard deviations """
    return random.normal(0., self.sigList)

  def _learnStep(self):
    """ calculates the gradient and executes a step in the direction
        of the gradient, scaled with a learning rate alpha. """
    deltas = self.perturbation()
    #reward of positive and negative perturbations
    reward1 = self._oneEvaluation(self.current + deltas)
    reward2 = self._oneEvaluation(self.current - deltas)

    self.mreward = (reward1 + reward2) / 2.
    if self.baseline is None:
      # first learning step
      self.baseline = self.mreward
      fakt = 0.
      fakt2 = 0.
    else:
      #calc the gradients
      if reward1 != reward2:
        #gradient estimate with likelihood gradient and normalization
        fakt = (reward1-reward2)/(2.*self.bestEvaluation-reward1-reward2)
      else: fakt=0.
      #normalized sigma gradient with moving average baseline
      norm = (self.bestEvaluation-self.baseline)
      if norm != 0.0:
        fakt2=(self.mreward-self.baseline)/(self.bestEvaluation-self.baseline)
      else: fakt2 = 0.0
    #update baseline
    self.baseline = 0.9 * self.baseline + 0.1 * self.mreward
    # update parameters and sigmas
    self.current = self.gd(fakt*deltas-self.current*self.sigList*self.wDecay)
    if fakt2 > 0.: #for sigma adaption alg. follows only positive gradients
      if self.exploration == "global":
        #apply sigma update globally
        self.sigList = self.gdSig(fakt2 * ((self.deltas ** 2).sum() - (self.
            sigList ** 2).sum())/(self.sigList * float(self.numParameters)))
      elif self.exploration == "local":
        #apply sigma update locally
        self.sigList = self.gdSig(fakt2 * (deltas * deltas - self.sigList * self
            .sigList) / self.sigList)
      else:
        raise NotImplementedError(str(self.exploration) + " not a known
            exploration parameter setting.")
```

Listing 5: The PGPE Class Methods

B

THE PGPE ALGORITHM

We wanted to give you a stand alone PGPE code that one can use directly for its problem (Listing 1). Note that the PGPE class needs NumPy. We also assume that a problem class exists that provides the problem dimension and an evaluation method that returns an episodic reward. Make an instance of PGPE with the problem as an attribute and call cycle() with the number of world-interactions you assume suitable.

```
class PGPE():
    def __init__(self, problem):
        self.prop = problem #The broblem instance
        self.proSize = self.prop.proSize #The size of the problem
        self.pop = random.rand(self.proSize,2).astype('f').copy() #The batch
        self.epsilon = 2.0 #Initial value of sigmas, adapted to rpoblem
        self.baseline=0.0 #Suitable starting baseline (e.g. 0.99 * first reward)
        self.best=-1000000.0 #Replace with -inf or whatever suits your problem
        self.original = zeros((self.proSize),'f') #Stores the parameter set
        self.sigList=ones((self.proSize),'f')*self.epsilon #Stores sigmas
        self.deltas=zeros((self.proSize),'f') #Difference vector for exploration
        self.alphaT = 0.2 #The parameter update stepsize
        self.alphaS = 0.1 #The sigma update stepsize

    def learn(self):
        #check if reward is the best observed up to now and ident. better reward
        if self.fit[0] > self.fit[1]: self.reward=self.fit[0]
        else: self.reward=self.fit[1]
        if self.reward > self.best: self.best = self.reward

        #calc the gradients
        if self.fit[0] != self.fit[1]:
            #PGPE symmetric gradient estimate and normalization
            gM=(self.fit[0]-self.fit[1])/(2.0*self.best-self.fit[0]-self.fit[1])
        else: gM=0.0
        #normalized sigma gradient with moving average baseline
        gS=(self.reward-self.baseline)/(self.best-self.baseline)
        self.baseline=0.9*self.baseline+0.1*self.reward #update baseline

        # update parameters and sigmas
        self.original += self.alphaT*gM*self.deltas
        if gS > 0.0: #For sigma adaption alg. follows only positive gradients
                dif = (self.deltas**2).sum()-(self.sigList**2).sum()
                self.sigList+=self.alphaS*gS*(dif)/self.sigList

    def cycle(self, worldInteractions):
        for lkj in range(worldInteractions/2):
            self.deltas=random.normal(0.0, self.sigList) #Perturbations
            self.pop[:,0] = self.original+self.deltas #"positive" sample
            self.pop[:,1] = self.original-self.deltas #"negative" sample
            self.fit = self.prop.evaluate(self.pop) #Evaluate samples
            self.learn() #Use new information for parameter update
```

Listing 1: Stand alone standard PGPE algorithm

www.pybrain.org. (Cited on pages 74, 125, and 128.)

Open Dynamics Engine, Jan 2010. URL http://www.ode.org/. (Cited on page 67.)

D.A. Aberdeen. *Policy-Gradient Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, Australian National University, 2003. (Cited on pages 6 and 23.)

L. Baird and A.W. Moore. Gradient Descent for General Reinforcement Learning. *Advances in neural information processing systems*, pages 968–974, 1999. (Cited on page 5.)

J. Baxter and P. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001. (Cited on page 21.)

J. Baxter and P.L. Bartlett. Reinforcement learning in POMDPs via direct gradient ascent. In *Proc. 17th International Conf. on Machine Learning*, pages 41–48. Morgan Kaufmann, San Francisco, CA, 2000. (Cited on pages 6 and 23.)

H. Benbrahim and J. Franklin. Biped dynamic walking using reinforcement learning. *Robotics and Autonomous Systems Journal*, 1997. (Cited on pages 5, 17, and 113.)

D.P. Bertsekas and J.N. Tsitsiklis. Neuro-dynamic programming. Master's thesis, Athena Scientific, Belmont, MA, 1996. (Cited on page 5.)

A. Boularias. *Predictive Representations For Sequential Decision Making Under Uncertainty*. PhD thesis, Universite Laval, Quebec, 2010. (Cited on page 115.)

B. Bouzy and G. Chaslot. Monte-Carlo Go Reinforcement Learning Experiments. In *In IEEE 2006 Symposium on Computational Intelligence in Games*, pages 187–194. IEEE, 2006. (Cited on page 102.)

M. Buss and S. Hirche. Institute of Automatic Control Engineering, TU München, Germany, 2008. http://www.lsr.ei.tum.de/. (Cited on page 77.)

X. Du, J. Zhai, and K. Lv. Algorithm Trading using Q-Learning and Recurrent Reinforcement Learning. *positions*, 1:1. (Cited on page 115.)

I. Fasel, A. Wilt, N. Mafi, and C. Morrison. Intrinsically motivated information foraging. In *Proceedings of the IEEE International Conference on Development and Learning (ICDL)*, 2010. (Cited on pages 114 and 115.)

B. Gassend, DE Clarke, M. van Dijk, and S. Devadas. Silicon physical random functions. In *ACM Conference on Computer and Communications Security-CCS*, pages 148–160, 2002. (Cited on page 81.)

B. Gassend, D. Lim, D. Clarke, M. Van Dijk, and S. Devadas. Identification and authentication of integrated circuits. *Concurrency and Computation: Practice & Experience*, 16(11):1077–1098, 2004. (Cited on pages 82 and 83.)

S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *ICML; Vol. 227*, 2007. URL http://portal.acm.org/citation.cfm?id=1273496.1273531. (Cited on page 102.)

T. Glasmachers, T. Schaul, S. Yi, D. Wierstra, and J. Schmidhuber. Exponential natural evolution strategies. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 393–400. ACM, 2010. (Cited on page 119.)

F. Gomez, J. Schmidhuber, and R. Miikkulainen. Efficient non-linear control through neuroevolution. 2006. (Cited on page 16.)

F. Gomez, J. Schmidhuber, and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *The Journal of Machine Learning Research*, 9:937–965, 2008. ISSN 1532-4435. (Cited on page 16.)

F. J. Gomez and J. Schmidhuber. Co-evolving recurrent neurons learn deep memory POMDPs. In *Proc. of the 2005 conference on genetic and evolutionary computation (GECCO), Washington, D. C.* ACM Press, New York, NY, USA, 2005. (Cited on page 16.)

F.J. Gomez and R. Miikkulainen. Solving non-Markovian control tasks with neuroevolution. In *International Joint Conference on Artificial Intelligence*, volume 16, pages 1356–1361. Citeseer, 1999. (Cited on page 16.)

A. Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. PhD thesis, Technische Universität München, 2007. (Cited on pages 102, 103, 104, and 107.)

A. Graves, S. Fernández, and J. Schmidhuber. Multi-Dimensional Recurrent Neural Networks, 2007. (Cited on pages 103 and 104.)

M. Grüttner. Evolving Multidimensional Recurrent Neural Networks for the Capture Game in Go, 2008. (Cited on pages 101, 102, 103, 104, 107, and 120.)

M. Grüttner, F. Sehnke, T. Schaul, and J. Schmidhuber. Multi-dimensional deep memory go-player for parameter exploring policy gradients. In W. Duch K. Diamantaras and L. Iliadis, editors, *Proceedings of the International Conference on Artificial Neural Networks*. ICANN 2010, Springer-Verlag Berlin Heidelberg, 2010. (Cited on pages 102, 103, 104, 105, 106, and 107.)

N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9:159–195, 2001. (Cited on pages 47 and 103.)

N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 312–317. IEEE, 2002. ISBN 0780329023. (Cited on page 14.)

S. Hochreiter and J. Schmidhuber. Flat minima. *Neural Computation*, 9 (1):1–42, 1997. (Cited on page 49.)

M.I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. *Proc. of the Eighth Annual Conference of the Cognitive Science Society*, 8:531–546, 1986. (Cited on pages 71, 72, 73, 75, and 77.)

L.P. Kaelbling, M.L. Littman, and A.R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101 (1-2):99–134, 1998. (Cited on page 9.)

T. Kohonen. *Self-Organizing Maps*. Springer-Verlag, 1995. (Cited on page 88.)

V. Konda and J. Tsitsiklis. Actor-critic algorithms. *In Advances in Neural Information Processing Systems*, 12, 2000. URL `citeseer.ist.psu.edu/konda01actorcritic.html`. (Cited on page 5.)

G. Konidaris, D. Shell, and N. Oren. Evolving Neural Networks for the Capture Game. In *Proceedings of the SAICSIT Postgraduate Symposium*, 2002. (Cited on page 102.)

Y. LeCun and C. Cortes. The MNIST database of handwritten digits, 1998. (Cited on page 93.)

J.W. Lee, D. Lim, B. Gassend, G.E. Suh, M. Van Dijk, and S. Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *Proceedings of the IEEE VLSI Circuits Symposium*, pages 176–179. Citeseer, 2004. (Cited on page 83.)

D. Lim. *Extracting Secret Keys from Integrated Circuits*. Msc thesis, MIT, 2004. (Cited on pages 82 and 83.)

M. Liwicki, A. Graves, S. Fernández, H. Bunke J., and Schmidhuber. A novel approach to on-line handwriting recognition based on bidirectional long short-term memory networks. In *Proc. 9th Int. Conf. on Document Analysis and Recognition*, pages 367–371, September 2007. (Cited on page 103.)

M. Majzoobi, F. Koushanfar, and M. Potkonjak. Lightweight secure PUFs. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 670–673. IEEE Press, 2008a. (Cited on page 82.)

M. Majzoobi, F. Koushanfar, and M. Potkonjak. Testing techniques for hardware security. In *Proceedings of the International Test Conference (ITC)*, pages 1–10, 2008b. (Cited on pages 82 and 83.)

A. Miyamae, Y. Nagata, I. Ono, and S. Kobayashi. Natural policy gradient methods with parameter-based exploration for control tasks. In *Proceedings of the Twenty-Fourth Annual Conference on Neural Information Processing Systems (NIPS)*, 2010. (Cited on pages 114 and 115.)

H. Müller, M. Lauer, R. Hafner, S. Lange, A. Merke, and M. Riedmiller. Making a robot learn to play soccer. *Proceedings of the 30th Annual German Conference on Artificial Intelligence (KI-2007)*, 2007. (Cited on pages 59 and 114.)

R. Munos and M. Littman. Policy gradient in continuous time. *Journal of Machine Learning Research*, 7:771–791, 2006. (Cited on pages 6 and 23.)

R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld. Physical one-way functions. *Science*, 297(5589):2026, 2002. (Cited on page 81.)

J. Peters and S. Schaal. Policy gradient methods for robotics. In *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006. (Cited on pages 5, 6, 17, 23, 59, and 113.)

J. Peters and S. Schaal. Reinforcement learning of motor skills with policy gradients. In *Neural Networks*, pages 682–697, 2008a. (Cited on pages 17 and 59.)

J. Peters and S. Schaal. Natural actor-critic. *Neurocomputing*, 71(7-9): 1180–1190, 2008b. ISSN 0925-2312. doi: http://dx.doi.org/10.1016/j.neucom.2007.11.026. (Cited on pages 21, 23, and 59.)

J. Peters, S. Vijayakumar, and S. Schaal. Natural actor-critic. In *Proceedings of the Sixteenth European Conference on Machine Learning*, 2005. (Cited on pages 5, 17, and 119.)

M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proceedings of the IEEE international conference on neural networks*, volume 1993, pages 586–591. San Francisco: IEEE, 1993. (Cited on page 114.)

M. Riedmiller, J. Peters, and S. Schaal. Evaluation of policy gradient methods and variants on the cart-pole benchmark. In *ADPRL-2007*, 2007a. (Cited on pages 67 and 114.)

M. Riedmiller, J. Peters, and S. Schaal. Evaluation of policy gradient methods and variants on the cart-pole benchmark. In *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*, 2007b. (Cited on pages 23 and 60.)

T. Rückstieß, M. Felder, and J. Schmidhuber. State-Dependent Exploration for policy gradient methods. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases 2008, Part II, LNAI 5212*, pages 234–249, 2008. (Cited on pages 25, 48, and 114.)

T. Rückstieß, F. Sehnke, T. Schaul, D. Wierstra, Y. Sun, and J. Schmidhuber. Exploring parameter space in reinforcement learning. *Paladyn*, 1 (1):14–24, 2010. (Cited on page 79.)

U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas, and J. Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 237–249, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866335. URL http://doi.acm.org/10.1145/1866307.1866335. (Cited on pages 83 and 84.)

T. Schaul and J. Schmidhuber. A scalable neural network architecture for board games. In *Proceedings of the IEEE Symposium on Computational Intelligence in Games (CIG 08)*, 2008. (Cited on pages 103 and 120.)

T. Schaul and J. Schmidhuber. Scalable neural networks for board games. In *International Conference on Artificial Neural Networks (ICANN)*, 2009. (Cited on pages 102, 103, 104, and 107.)

T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, and T. Rückstießand J. Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 11:743–746, 2010. (Cited on pages 74, 105, and 125.)

N. Schraudolph, J. Yu, and D. Aberdeen. Fast online policy gradient learning with smd gain vector adaptation. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*. MIT Press, Cambridge, MA, 2006. (Cited on pages 5 and 17.)

H.P.P. Schwefel. *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc. New York, NY, USA, 1993. (Cited on pages 15 and 59.)

F. Sehnke. PGPE – Policy Gradients with Parameter-based Exploration – Demonstration video: Learning in Robot Simulatons., 2009. http://www.pybrain.org/videos/jnn10/. (Cited on pages 74, 76, and 78.)

F. Sehnke. PyBrain – Demonstration video: Learning in Robot Simulatons., 2010. http://www.youtube.com/watch?v=fEM7YDNonSE. (Cited on pages 72, 73, 74, 75, and 77.)

F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010a. (Cited on page 32.)

F. Sehnke, C. Osendorfer, J. Sölter, J. Schmidhuber, and U. Rührmair. Policy gradients for cryptanalysis. In *Proceedings of the International Conference on Artificial Neural Networks*, 2010b. (Cited on page 83.)

J. Sölter. *Cryptanalysis of Electrical PUFs via Machine Learning Algorithms*. Msc thesis, Technische Universität München, 2009. (Cited on page 83.)

J.C. Spall. An overview of the simultaneous perturbation method for efficient optimization. *Johns Hopkins APL Technical Digest*, 19(4): 482–492, 1998a. (Cited on pages 19, 46, and 59.)

J.C. Spall. Implementation of the simultaneous perturbation algorithm forstochastic optimization. *Aerospace and Electronic Systems, IEEE Transactions on*, 34(3):817–823, 1998b. (Cited on pages 19 and 46.)

K.O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. ISSN 1063-6560. (Cited on page 16.)

K.O. Stanley and Risto Miikkulainen. Evolving a Roving Eye for Go, 2004. URL `http://www.cs.utexas.edu/users/nn/downloads/papers/stanley.gecco04.pdf`. (Cited on page 102.)

F. Streichert. *Evolutionary Algorithms in Multi-Modal and Multi-Objective Environments*. Logos verlag berlin, isbn 978-3832515522, 2007, University of Tübingen, 2007. (Cited on page 14.)

F. Streichert and H. Ulmer. JavaEvA - a java framework for evolutionary algorithms. Technical Report WSI-2005-06, Centre for Bioinformatics Tübingen, University of Tübingen, 2005. URL `http:`

//w210.ub.uni-tuebingen.de/dbt/volltexte/2005/1702/. (Cited on page 15.)

G.E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual Design Automation Conference*, page 14. ACM, 2007. (Cited on page 83.)

Y. Sun, D. Wierstra, T. Schaul, and J. Schmidhuber. Efficient Natural Evolution Strategies. In *Genetic and Evolutionary Computation Conference (GECCO)*, 2009. (Cited on page 114.)

R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998. (Cited on pages 5, 13, and 41.)

R.S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, 2000. (Cited on pages 5, 6, 21, and 23.)

S.B. Thrun. The role of exploration in learning control. *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pages 527–559, 1992. (Cited on page 22.)

H. Ulbrich. Institute of Applied Mechanics, TU München, Germany, 2008. http://www.amm.mw.tum.de/. (Cited on page 74.)

H. van Hasselt. Insights in reinforcement learning: formal analysis and empirical evaluation of temporal-difference learning algorithms. *SIKS dissertation series*, 2011(04), 2011. (Cited on page 115.)

C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8: 279–292, 1992. (Cited on pages 13 and 14.)

K. Weicker. *Evolutionäre Algorithmen*. Vieweg + Teubner, 2002. ISBN 3519003627. (Cited on page 15.)

M. Wiering and J. Schmidhuber. Efficient Model-Based Exploration. *From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior*, 1998. (Cited on page 22.)

D. Wierstra, T. Schaul, J. Peters, and J. Schmidhuber. Fitness expectation maximization. In *Parallel Problem Solving from Nature*. 2008a. (Cited on page 69.)

D. Wierstra, T. Schaul, J. Peters, and J. Schmidhuber. Natural evolution strategies. In *Proceedings of the Congress on Evolutionary Computation (CEC08), Hongkong*. IEEE Press, 2008b. (Cited on pages 78, 114, 115, and 119.)

R.J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992. (Cited on pages 5, 20, 23, 33, 34, and 59.)

L. Wu and P. Baldi. A scalable machine learning approach to go. In *in Advances in Neural Information Processing Systems 19*, pages 1521–1528. MIT Press, 2007. (Cited on page 103.)

T. Zhao, H. Hachiya, G. Niu, and M. Sugiyama. Analysis and improvement of policy gradient estimation. *Neural networks : the official journal of the International Neural Network Society*, pages 1–30, October 2011a. ISSN 1879-2782. doi: 10.1016/j.neunet.2011.09.005. URL http://www.ncbi.nlm.nih.gov/pubmed/22019189. (Cited on pages 115 and 119.)

T. Zhao, H. Hachiya, G. Niu, and M. Sugiyama. Analysis and improvement of policy gradient estimation. In *NIPS*, October 2011b. doi: 10.1016/j.neunet.2011.09.005. URL http://www.ncbi.nlm.nih.gov/pubmed/22019189. (Cited on page 115.)

This thesis was typeset with LaTeX 2_ε using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)

The typographic style was inspired by Bringhurst's genius as presented in *The Elements of Typographic Style*. It is available for LaTeX via CTAN as "`classicthesis`".