NTNU
Det skapende universitet

Robotlæring for slangeroboter

Christian Monzo Brandvold

Master i elektronikk
Oppgaven levert: Januar 2011
Hovedveileder: Geir Mathisen, ITK
Biveileder(e): Aksel A. Transeth, SINTEF IKT
Sigurd A. Fjerdingen, SINTEF IKT

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

Oppgavetekst

Roboter utformet som slanger - slangeroboter - har et stort potensial innen områder som søk og redning, og inspeksjon og vedlikehold.

Slangeroboter med aktive hjul er en spesialisert form for slangeroboter. De aktive hjulene er fordelaktig i menneskeskapte miljøer som kontorgulv, fabrikker og ventilasjonssystemer. De aktive hjulene og den artikulerte kroppen som robotene består av robotene tilbyr en effektiv plattform for forflytning både horisontalt og vertikalt.

En slangerobot må være i stand til å utføre avanserte bevegelser for å navigere komplekse menneskeskapte miljøer som rørkonstruksjoner. Temaet for denne masteroppgaven er å utvikle kontrollstrategier for bevegelser av en slangerobot på flate plan og i rørstrukturer. En preliminær versjon av en tidligere utviklet simulator for simulering av hjulbasert fremdrift med en slangerobot kalt Piko vil være tilgjengelig.

Forslag til arbeid
Bli kjent med simulatoren og lag en kort veiledning, for eksempel: installere simulator for utvikling, lese sensor signaler, kontrollere de ulike grader av frihet, og utføre simuleringer uten visualisering.
1. Utvikle og implementer nødvendige forbedringer i simulatoren.
2. Utfør en kort litteraturstudie innen følgende emner relevante for oppgaven:
a. Reinforcement Learning (RL)
b. Funksjonsapproksimatorer for RL
c. Nåværende rørinspeksjonsroboter og deres egenskaper (grad av autonomi, design, etc)
3. Formuler én eller flere bevegelse primitiver (kontrollere), og undersøk hvordan optimalisere kontrollestrategien basert på bevegelsesprimitiver ved bruk av robotlæring.
a. Utforske spesielt et bevegelsesprimitiv som tillater en slangerobot med hjul å løfte hodet så høyt som mulig ved implementering i simulatoren.

Kommentarer
- Robotens læringsstrategier kan være basert på resultater fra [2].
- Hvis en annen type læringsstrategi enn Reinforcement Learning er brukt (f.eks evolusjonære algoritmer), bør litteraturstudien reflektere dette.

1. Fjerdingen, S.A., Liljebäck, P. and Transeth, A.A., A snake-like robot for internal inspection of complex pipe structures (PIKo), in Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems , St. Louis, USA, Oct 11-15, 2009, pp. 5665-5671.
2. Fjerdingen, S.A., Kyrkjebø, E., Transeth, A.A., AUV Pipeline Following using Reinforcement Learning, in Proc. Int. Symp. on Robotics , München, Germany, June 8-11, 2010, to appear.

Oppgaven gitt: 30. august 2010
Hovedveileder: Geir Mathisen, ITK

Robot learning for snake robots

Christian Monzó Brandvold

January 30, 2011

Abstract

Developing a control strategy for a wheeled snake robot can be difficult given the number of parameters involved. In this thesis we have studied the use of a reinforcement learning framework to develop a control strategy that allows a wheeled snake to lift its head as much as possible. The learning process has been done using a simulator developed for SINTEF's pipe inspection robot PIKo.

The reinforcement learning methodology used has been CACLA with an RBF network as function approximator. Various alternatives have been proposed and used for the action space in simulations showing positive results.

Issues with the simulator have been detected and workarounds proposed for them.

Contents

List of Figures **5**

1 Introduction **7**
1.1 Motivation . . . 7
1.2 Background . . . 7
1.3 Contribution . . . 8

2 Background **9**
2.1 State of the art . . . 10
2.1.1 Commercial products . . . 10
2.1.2 Research prototypes . . . 12
2.2 PIKO . . . 16
2.3 Control strategies . . . 18
2.3.1 Function approximators . . . 19
2.3.2 The Markov Property . . . 23
2.3.3 Reinforcement Learning Methods . . . 24
2.3.4 Sarsa . . . 24
2.3.5 Actor-Critic . . . 24
2.3.6 Actor-Critic Learning Automaton (Acla) . . . 24
2.3.7 Cacla . . . 25

3 The PIKo simulator **27**
3.1 Initial state of the simulator . . . 27
3.2 Tutorial . . . 27
3.2.1 Installation . . . 27
3.2.2 Reading of sensors . . . 30
3.2.3 Control . . . 31
3.2.4 Performing simulations without visualizations . . . 32
3.3 Modifications . . . 33
3.3.1 The Artificial Neural Network (ANN) class . . . 33
3.3.2 The Reinforcement Learning class . . . 36

3.3.3 Resetting the robot 39
3.3.4 Simulation without visualization 40
3.3.5 Unstability 41
3.3.6 "External" forces 41
3.3.7 Stabilizing the actions 41
3.3.8 Simulation setup 42

4 Implementation 45
4.1 Initial setup 45
4.1.1 State representation 45
4.1.2 Reinforcement Learning Method 46
4.1.3 Function Approximator 47
4.2 First approach 49
4.3 Second approach 50
4.4 Third approach 52

5 Simulation results 53
5.1 First approach 53
5.2 Second approach 55
5.3 Third approach 58

6 Discussion and further work 63
6.1 Discussion of results 63
6.2 Further work 64

7 Conclussions 65

8 References 67

A Simulator 69
A.1 Installation 69

List of Figures

2.1 Example of a smart pipeline inspection gauge (PIG) and diagram of its components . . . 11
2.2 Versatrax 300. Crawler with supplementary cart . . . 11
2.3 LISY. Lateral Launch System . . . 12
2.4 MFI5-48 . . . 12
2.5 Junxing XP-W series . . . 13
2.6 MAKRO1.1 . . . 14
2.7 MRINSPECT V . . . 15
2.8 Explorer . . . 15
2.9 The snake robot PiKO . . . 16
2.10 Illustration of a module of the robot PIKo . . . 16
2.11 Scheme for vertical climbing showing the push-points used on each side of the pipe. . . . 17
2.12 Diagram of single-layer artificial neural network . . . 19
2.13 Actor-Critic architecture . . . 25

3.1 Diagram of the implemented RBFN. Notice that N designates the sum of the values of the nodes of the hidden layer . . . 34
3.2 Plot of the benchmark function from equation 3.3 . . . 37
3.3 Screen captures showing how the model rises from the ground instead of being positioned over it as expected . . . 40
3.4 Screen captures showing random behaviour. . . . 43
3.5 Diagram of the observed behaviour when a joint is subject to gravity . . . 44
3.6 Diagram of how the settings of an action are sustained. . . . 44

4.1 Representation of the PIKo robot in the state-space . . . 46
4.2 Diagram of the proposed RBFN . . . 48
4.3 Mapping of the RBFN to the workspace . . . 48
4.4 Sequence of steps oriented to achieve an S-shape . . . 51

5.1 Diagram showing how using the value of the hypotenuse of a a triangle 54
5.2 Height in metres achieved at the end of each episode with only the first joint active. X-axis denotes the number of episodes simulated. 56
5.3 Height in metres achieved at the end of each episode with only the first joint active and increasing the number of episodes simulated. X-axis denotes the number of episodes simulated. . 57
5.4 Height in metres achieved at the end of each episode with the 2 first joints active and increasing the number of episodes simulated. X-axis denotes the number of episodes simulated. . 58
5.5 Height in metres achieved at the end of each episode with the 3 first joints active and increasing the number of episodes simulated. X-axis denotes the number of episodes simulated. . 59
5.6 Height in metres achieved at the end of each episode with the 2 first joints active and increasing the number of episodes simulated. X-axis denotes the number of episodes simulated. . 60
5.7 Height of the robots head during 5 episodes of 100 steps . . . 60
5.8 Height of the robots head during 2 episodes of 500 steps . . . 61

Chapter 1

Introduction

This master's thesis is a collaboration between SINTEF and NTNU. The background is a SINTEF project aimed to develop a wheeled snake robot for inspection of pipelines which resulted in the robot PIKo. The simulator used throughout this thesis is a development part of this project to facilitate the development of control strategies for it.

1.1 Motivation

Wheeled snake robots, or *serpentine robots*, are a type of robots suitable for inspection of pipelines. Their snake-like design allow them navigation through narrow pipelines and sort out obstacles such as L-bends and T-joints. However, their redundancy of actuators increase the complexity of steering and control. Robot learning is proposed as an alternative to develop control strategies by allowing a model of the robot to learn which is the best way to perform a given task.

1.2 Background

The inspection of pipelines is currently done by Pipeline Inspection Gauges (PIG) or by robotic crawlers which are typically controlled remotely and lack the capability to perform autonomous inspection.

Research projects have developed different kinds of articulated robots for pipe inspection being some of them of the "snake-like" variant. However, the accent has been put on the design of the robot rather than control strategies being usually controlled remotely and having a set of limited commands to

sort possible obstacles.

The PIKo simulator offers realistic simulations based on the serpentine robot PIKo.

1.3 Contribution

In this paper we confront the problem of lifting the head of the robot PIKo as much as possible by developing a reinforcement learning framework in the PIKo simulator.

The classes and data structures necessary to work with CACLA-based reinforcement learning have been added to the simulator code including the implementation of an RBF network.

A reference tutorial for the simulator has been written with indications for how to use it for development and simulation.

During the work on this paper, various issues with the simulator were detected. As improving of the simulator is not in the scope of the topic only workarounds were proposed and implemented so the work on reinforcement learning could be done.

Chapter 2

Background

In this chapter we first present the State of the art in pipeline inspection both in the commercial sector and projects in research. In the second part we explain the characteristics of the pipeline inspection robot PIKo designed at SINTEF. The last part of this chapter is dedicated to control strategies and related concepts.

2.1 State of the art

All over the world, pipelines are used for transport of liquids and gases. Although pipelines suppose a low maintenance alternative compared to e.g. road transport or shipping they are subject to corrosion, cracking, natural disasters and other phenomena which can cause leaks and even major environmental disasters making inspection of pipelines crucial. Although inspection of surface-placed pipelines can be relatively easy the inspection of buried down pipelines, or unreachable for other reasons can be tricky or very expensive. To solve this scenario there has been developed solutions to perform the inspection from inside the pipelines. In this section we present commercially available solutions as well as research prototypes.

2.1.1 Commercial products

The necessity to inspect otherwise unreachable pipelines has driven the development of special purpose devices. There are basically two branches of such devices depending of the size, length and complexity of the pipeline network they are used at. The most common for large-scale operations are the "PIGS"s. A PIG (Pipeline Inspection Gauge, Figure 2.1) is a device of roughly the diameter of the pipeline which it is sent through. They use the pressure of the liquid or gas transported by the pipeline to advance letting them travel large distances (over 450 Km [1]) with the advantage of not having to cut the service. Pigs were first used for inside cleaning of the pipelines but nowadays they can be equipped with an array of sensors for magnetic flux leakage or ultrasound. However, one specific pig is limited to a narrow range of diameters and cannot sort butterfly valves and although they can record data while travelling they may not be always the best option.

For smaller constructions or for inspection that requires a more interactive approach we find push cameras (which are little more than cameras mounted on more or less rigid rods) and robotic "crawlers". The typical crawler is composed of a single module although some products can carry a passive trailer for sensors and use a tether cable for direct control and power supply and their range is therefore limited by its length (although it can provide ranges up to 1.6 Km [1] as is the case for the Versatrax 300 from figure 2.2).

In comparison to pigs which are pretty much uncontrollable, crawlers can

[1] `http://inuktun.com`

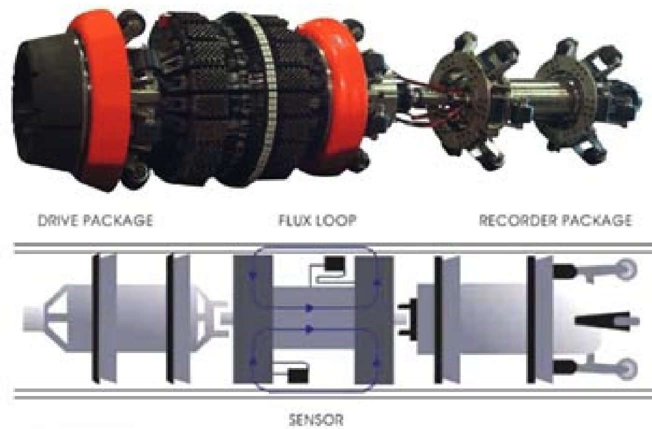


Figure 2.1: Example of a smart pipeline inspection gauge (PIG) and diagram of its components

be operated on-site and provide more manoeuvrability. As is the case with pigs, crawlers are heavily conditioned by pipe-diameter and have problems sorting T-bends. At this point no existing solution is able to start climbing through a branch although they are able to climb through a vertical elbow. As a patch to solve these shortcomings, some vendors [2] offer dedicated features like the mounting of an extra probe for lateral inspection without diverging from the classic one-unit configuration. This probe is launched from the main body of the crawler and into a lateral pipeline as seen in figure 2.3. Usually their means for inspection are limited to the use of cameras.

[2] http://www.atlas-inspection.com/ibak-lateral.html

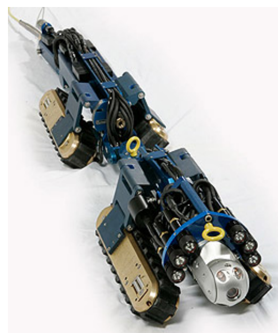


Figure 2.2: Versatrax 300. Crawler with supplementary cart

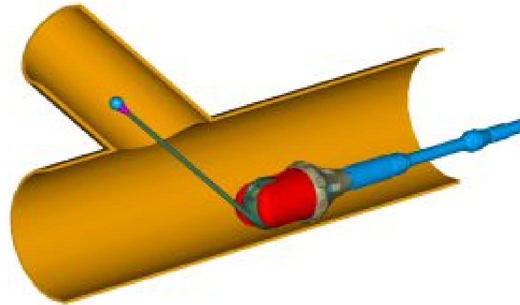


Figure 2.3: LISY. Lateral Launch System

Although the previous constitute the more common alternatives, more advanced crawlers get rid of the tether cable in favour of wireless communication and extended range. Models such as the articulated snake-like robot like the MFI5-48 shown in figure 2.4 from *itRobotics*[3] provide autonomous detection of corrosion and wall thinning using Magnetic Flux Leakage (MFL) sensors and can sort bends of 90 and 180 degrees. It records data during the inspection that can later be transferred and analysed.

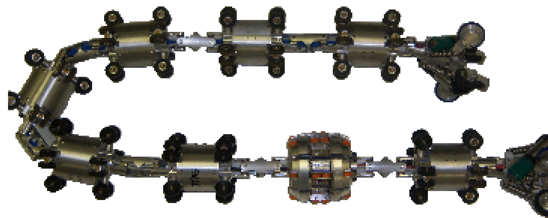


Figure 2.4: MFI5-48

Another example is the Junxing XP-W [4](Figure 2.5) series which offers a range of up to 10Km, infrared cameras and the use of X-rays for welding line testing.

2.1.2 Research prototypes

Research on inspection robots is an active field and oriented to search and rescue missions in ruins and debris and inspection of otherwise unreachable

[3]http://www.itrobotics.com/product_mfi.html

[4]http://www.alibaba.com/product-gs/309210804/X_ray_pipeline_crawler_detecting_system.html

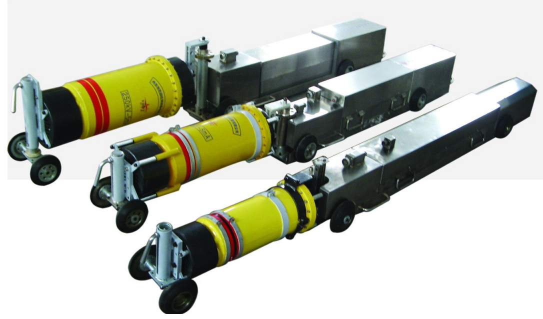


Figure 2.5: Junxing XP-W series

infrastructures where articulated, "snake-shaped", robots for inspection is one of the desing alternatives that have received much attention. Snake-shaped robots designate two different types differentiated by their locomotion model:

- "Snake robots" are articulated, multi-segmented, robots which are biologically inspired by snakes. They achieve propulsion by the movements of their joints. This means they are more efficient on surfaces that present obstacles they can use as supporting points or high friction coating on the robot itself.

- "Serpentine robots" are articulated, multi-segmented, robots that rely on the use of wheels or other propulsion mechanism such as caterpillar traction.

Existing projects put the accent on design of these robots putting weight on power consumption, increased manoeuvrability and autonomy. However, as they are mostly concept models, their control strategies are usually limited to remote control or capability-oriented strategies as e.g. climb a specific obstacle. The research of more advanced control strategies for them has been relegated as background matter for further work.

In this section we describe the following projects.

- MAKRO

- MRINSPECT

- Explorer

MAKRO

The MAKRO-PLUS project is an articulated, "serpentine", robot designed for autonomous navigation through pipes of varying diameter using wheels for propulsion. It's able to simultaneously climb a step and turning [2] [3]. The task of inspection is divided in two: a planner and an action controller. Given a model of the sewer system with pipes and manholes, the planner decides the intermediate points of the route and the action controller decides how to get there. Figure 2.6 shows one the MAKRO1.1 robot.

Figure 2.6: MAKRO1.1

As MAKRO relies on a model of the pipe structure when it confronts an obstacle such as a step, the action controller is able to calculate the necessary steps to overcome it using backtracking.

MRINSPECT

The MRINSPECT project has developed a series of robots capable of free movement in all types of pipes [4]. For its fifth version, MRINSPECT V, it was developed selective drive mechanism that uncouples unneeded wheels when possible lowering the power consumption.

Figure 2.7: MRINSPECT V

Explorer

Explorer is a large, long-range, untethered, modular inspection robot developed at Carnegie Mellon University [5] (figure 2.8).

Figure 2.8: Explorer

The navigation is based on a set of commands sent by an operator in real-time through a wireless 802.11b communication.

2.2 PIKO

PIKO [6] (Fig. 2.9) is a snake-like robot designed to navigate through complex pipe structures of varying diameters. These "complex pipe strutures" are comprised by vertical and horizontal straight segments joined by L- or T-joints and obstacles such as valves.

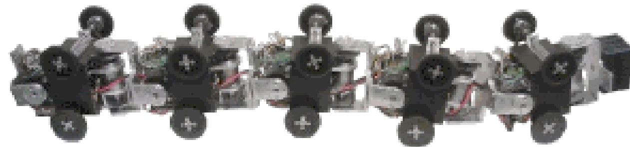


Figure 2.9: The snake robot PiKO

PiKO is composed by 5 modules interconnected by rotational joints allowing both vertical and horizontal movement through a servo motor for each direction. This servos allow the ability to lift three modules on it's own. See [6] for in-depht information.

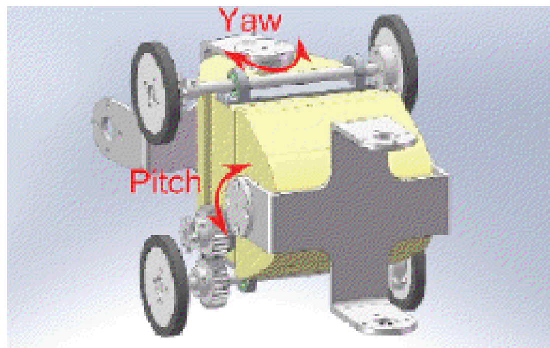


Figure 2.10: Illustration of a module of the robot PIKo

The locomotion is achieved by 4 wheels at each module, one couple of wheels below the module and the other one on top of it as seen in figure 2.10. One servo at each module drives synchronously all the modules wheels.

Horizontal movement is performed similarly to a train with distributed distributed traction like the ICE 3. Vertical climbing relies on an opposing-push strategy where the joints are used to create a pushing force against the

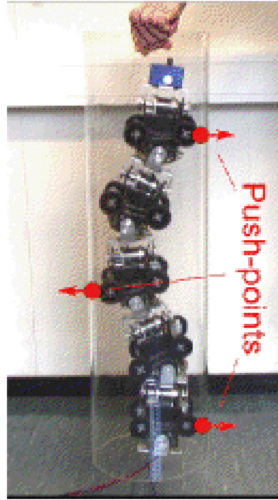


Figure 2.11: Scheme for vertical climbing showing the push-points used on each side of the pipe.

opposite sides of the pipe as seen in figure 2.11.

Table 2.1 shows the parameters of each module.

Weight of module	1.252
Length between joint axes	0.150 m
Max joint travel	±75°
Max continuous joint torque	11.5 Nm
Max joint speed	28°/s
Module width	0.130 m
Module height	0.140 m

Table 2.1: Parameters for a module

2.3 Control strategies

In this section we present different control strategy approaches

"Hard coded" . This category comprises control strategies which are custom developed for the specific robot. It requires comprehensive understanding of the robot's capabilities and dynamic behaviour. The before-mentioned factors added to the necessity of knowledge of the possible scenarios makes this alternative inflexible.

Evolutionary Methods EA uses some mechanisms inspired by biological evolution: reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the environment within which the solutions "live". Evolution of the population then takes place after the repeated application of the above operators.

Because you only need to encode the success or failure of the controller, these method provide "natural solutions" and they explore all the policy-space. On the backside, they require a very long training phase and the future transfer of the resulting controllers evolved in simulation to the physical robot can be very difficult.

Reinforcement Learning In Reinforcement Learning, RL, an agent must solve a task and has a set of actions available from which it chooses an action (or set of actions) it believes can solve the task. The agent receives feedback information from the environment about its performance and uses it to modify its behaviour.

This means Reinforcement Learning is appropriate for situation where the agent can interact with the environment and receive feedback from it. Since the agent can interact with the environment it is not necessary to have a complete knowledge of it.

This method requires well-defined value functions and function approximators.

2.3.1 Function approximators

Function approximators reduce the dimensionality of a space. Some function approximators are artificial neural networks and connectionist structures.

Artificial Neural Networks (ANN)

An artificial neural network (ANN) is a mathematical model or computational model that is inspired by the structure and/or functional aspects of biological neural networks. A neural network consists of an interconnected group of artificial neurons, and it processes information using a connectionist approach to computation. In most cases an ANN is an adaptive system that changes its structure based on external or internal information that flows through the network during the learning phase. Modern neural networks are non-linear statistical data modeling tools. They are usually used to model complex relationships between inputs and outputs or to find patterns in data.

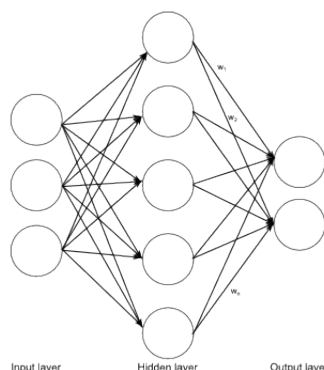


Figure 2.12: Diagram of single-layer artificial neural network

The word network in the term 'artificial neural network' is given by the inter-connections between the neurons in the different layers of each system. A basic ann has three layers. The first layer act as input and send data via connections to the second layer of neurons and then via more connections to the third layer of output neurons. More complex systems will have more layers of neurons with some having increased layers of input neurons and output neurons. Some networks can even be adaptative, varying the number if necessary. The connections store parameters called "weights" which are used to manipulate the data in the calculations. Figure 2.12 shows an artificial neural network with one hidden layer.

The network function $f(x)$ is defined as a composition of other functions $g_i(x)$, which can further be defined as a composition of other functions. This can be conveniently represented as a network structure, with arrows depicting the dependencies between variables. A widely used type of composition is the non-linear weighted sum, where $f(x) = K(\sum_i w_i g_i(x))$, where K (commonly referred to as the activation function) is some predefined function, such as the hyperbolic tangent. It will be convenient for the following to refer to a collection of functions g_i as simply a vector $g = (g_1, g_2, ..., g_n)$.

Two topologies are most common, feed-forward and recurrent networks. In recurrent neural networks (RNNs), connections between neurons may form directed cycles. These cycles allow the network to have internal states instead of being a static input-output transformation. The resulting behaviour is a temporally dynamic behaviour.

Learning What has attracted the most interest in neural networks is the possibility of learning. There are three major learning paradigms, each corresponding to a particular abstract learning task. These are supervised learning, unsupervised learning and reinforcement learning.

For applications where the solution is dependent on some data, the cost must necessarily be a function of the observations, otherwise we would not be modelling anything related to the data. It is frequently defined as a statistic to which only approximations can be made. As a simple example, consider the problem of finding the model f which minimizes $C = E\lfloor(f(x) - y)^2\rfloor$, for data pairs (x, y) drawn from some distribution $\mathcal{D}$. In practical situations we would only have samples from and thus, for the above example, we would only minimize $\hat{C} = \frac{1}{N}\sum_{i=1}^{N}(f(h_i) - y_i)^2$. Thus, the cost is minimized over a sample of the data rather than the entire data set.

When $N \to \infty$ some form of online machine learning must be used, where the cost is partially minimized as each new example is seen and is often used when N is fixed. However, it is most useful in the case where the distribution changes slowly over time.

Supervised learning In supervised learning, we are given a set of example pairs $(x, y), x \in X, y \in Y$ and the aim is to find a function $f : X \to Y$ in the allowed class of functions that matches the examples. In other words,

we wish to infer the mapping implied by the data. Tasks that fall within the paradigm of supervised learning are pattern recognition (also known as classification) and regression (also known as function approximation). The supervised learning paradigm is also applicable to sequential data (e.g., for speech and gesture recognition). This can be thought of as learning with a "teacher," in the form of a function that provides continuous feedback on the quality of solutions obtained thus far.

Unsupervised learning In unsupervised learning, some data x is given and the cost function to be minimized, that can be any function of the data x and the network's output, f .

The cost function is dependent on the task (what we are trying to model) and our a priori assumptions (the implicit properties of our model, its parameters and the observed variables).

As a trivial example, consider the model $f(x) = a$, where a is a constant and the cost $C = E\lfloor (f(x) - y)^2 \rfloor$. Minimizing this cost will give us a value of a that is equal to the mean of the data. The cost function can be much more complicated. Its form depends on the application: for example, in compression it could be related to the mutual information between x and y, whereas in statistical modelling, it could be related to the posterior probability of the model given the data. (Note that in both of those examples those quantities would be maximized rather than minimized).

Unsupervised learning oftn used estimation problems such as the estimation of statistical distributions, compression and filtering.

Reinforcement learning In reinforcement learning, data x are usually not given, but generated by an agent's interactions with the environment. At each point in time t , the agent performs an action y_t and the environment generates an observation x_t and an instantaneous cost c_t, according to some (usually unknown) dynamics. The aim is to discover a policy for selecting actions that minimizes some measure of a long-term cost; i.e., the expected cumulative cost. The environment's dynamics and the long-term cost for each policy are usually unknown, but can be estimated.

More formally, the environment is modeled as a Markov decision process

(MDP) with states $s_1, ..., s_n \in \mathcal{S}$ and actions $a_1, ..., a_m \in \mathcal{A}$ with the following probability distributions: the instantaneous cost distribution $P(c_t|s_t)$, the observation distribution and the transition , while a policy is defined as conditional distribution over actions given the observations. Taken together, the two define a Markov chain (MC). The aim is to discover the policy that minimizes the cost; i.e., the MC for which the cost is minimal.

ANNs are frequently used in reinforcement learning as part of the overall algorithm. Tasks that fall within the paradigm of reinforcement learning are control problems, games and other sequential decision making tasks.

Local Function Approximation with Connectionist Structures

ANNs as previously described are referred to as *global* function approximators. For use in robotics, a different type of function approximator referred to as *local* function approximator have recently become more popular. The main difference between a global and a local function approximator is that changing the weights of a global approximator may have global consequences, whereas changing the weights of a local approximator only have have consequences for a local area in the state-space surrounding the changed neuron (for connectionist approximators). The problem with global approximations is that a change with global effects is both uncontrollable and possibly unsafe, as the result of an experiment in one part of the state-space may lead to changes in a completely separate part of the state-space.

A typical example of local connectionist approximator is the Radial Basis Function Network (RBFN). An RBFN has typically one hidden layer, where each neuron in the hidden layer has its own centroid. For each input, the distance to each centroid is computed. The output of the hidden neurons is some non-linear function of the distance, e.g. the Gaussian. Thus, each kernel neuron in an RBFN computes an output that depends on a radially symmetric function. The output of the network is a weighted sum of the result of each neuron. For instance:

$$y(t) = \sum_i w_i K\left(\frac{x(t) - z_i}{\sigma_i}\right) = \sum_i w_i g\left(\frac{\|x(t) - z_i\|}{\sigma_i}\right), \qquad (2.1)$$

where w are tunable weights, K is the kernel function - in this instance the Gaussian g , x is the current input, z the centroid locations (mean of the Gaussian), and σ the width of the kernel (variance of the Gaussian). (Refer-

ence)

The Cerebellar Model articulation Controller (CMAC), also known as the Cerebellar Model Arithmetic Computer or tile coding, is a type of neural network popularized for RL by Sutton and Barto (reference) in their seminal book in reinforcement learning. A CMAC is computed as a weighted sum of kernel functions, but the kernel functions in this case are somewhat different than with ordinary ANNs or RBFNs. The kernels are buil from multiple layers of overlapping grids (discretizations) of the state-space, where each grid may be discretized in one to any number of sub-dimensions of the complete state-space dimensions. As such, it is a hierarchical build of discretizers of the state-space as opposed to the aforementioned single-layer networks.

2.3.2 The Markov Property

In a reinforcement learning context, an agent receives information from the environment which it uses to define a *state*. This state is then used by the agent to make decisions. If this state is a compact expression of all relevant information it is said to have the *Markov property*. For example, in a chess-game, the position of all the pieces define the state without the need to know the history of movements that led to it. If a state has the state signal has the Markov Property, [8] gives the next definition the states' probability distribution:

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t|, \}. \tag{2.2}$$

for all s', r, s_t and a_t.

If a reinforcement learning task satisfies the Markov property it is called a *Markov decision process, MDP,* or *finite* MDP if the state and action spaces are finite. Given any state a and action s, the probability of each possible next state, s', is

$$\mathcal{P}^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}. \tag{2.3}$$

Similarly, from [8], given any current state and action, s and action am together with any next state, s, the expected value of the newxt reward is

$$\mathcal{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}. \tag{2.4}$$

Together, (2.3) and (2.4), specify the dynamics of a finite MDP.

2.3.3 Reinforcement Learning Methods

Q-learning

Q-learning is well known off-policy reinforcement learning algorithm which directly approximates the optimal action-value function. Its advantages is that it can reach optimal solutions and an important use amount of research behind it. Unfortunately, it cannot handle continuous action spaces and sometimes lack convergence when used with function approximators. Its equation is:

$$Q_{t+1}(s_t, a_t) \stackrel{\alpha_t}{\longleftarrow} t_t + \gamma max_a Q_t(s_{t+1}, a) \tag{2.5}$$

2.3.4 Sarsa

Sarsa is a version of Q-Learning where the update is determined by the value of the last action taken instead of the one with highest value. Otherwise, it can reach optimal solutions as long as the rate of exploration decreases properly. Since it is a version of Q-learning, it cannot handle continuous spaces. Sarsa requires to know the next action making it a bit more complex to implement than Q-learning.

$$Q_{t+1}(s_t, a_t) \stackrel{\alpha_t}{\longleftarrow} t_t + \gamma Q_t(s_{t+1}, a) \tag{2.6}$$

2.3.5 Actor-Critic

Actor-Critic methods use two splits the estimation of the value function and the policy. It introduces two new structures, the *actor*, which involves the policy selection, and the *critic*, which estimates the value function and criticizes the actor's actions. Figure 2.13 shows the signal flows. Instead of focusing on the expected discounted rewards,Actor-Critic methods learn preference values for the actions. It does not however, handle continuous spaces.

The equation for the action values is:

$$P_{t+1}(s_t, a_t) = P_t(s_t, a_t) + \alpha(r_t + \gamma V_t(s_{+1} - V_t s_t)) \tag{2.7}$$

2.3.6 Actor-Critic Learning Automaton (Acla)

Acla's algorithm observes the updates resulting after each action and increments the preference for that action if the value of the state has increased

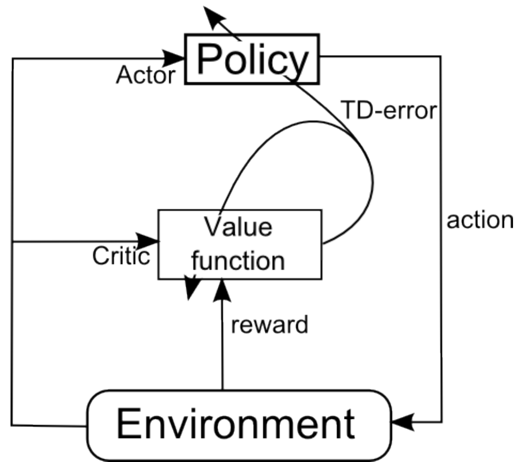


Figure 2.13: Actor-Critic architecture

and vice-versa. The update made to the *actor* if the action was good is:

$$\begin{cases} P_{t+1}(s_t, a_t) \stackrel{\alpha_t}{\longleftarrow} 1 & \text{, if the action was good} \\ P_{t+1}(s_t, a_t) \stackrel{\alpha_t}{\longleftarrow} 0 & \text{, otherwise} \end{cases}$$

Note that the updated value is always between 0 and 1.

As stated in section 2.3.5 it learns preference values rather than the sum of expected rewards but its learning is faster due to the use of state values.

2.3.7 Cacla

Cacla can be considered as Acla for continuous spaces. The difference lies in that it stores (either in a table or a function approximation) a value or vector with an approximation to the optimal action.

Cacla is usually fast and easy to implement. Basically, you retrieve the output from the actor, explore around this value and perform an action. If the value of the state has increased you update the actor towards that action. Schematically:

$$V_{t+1}(s_t) \stackrel{\beta_t}{\longleftarrow} r_t + \gamma V_t(s_{t+1})$$

If

$$\delta_t > 0$$

then

$$A_{t+1}(s_t) \xleftarrow{\alpha_t} a_t$$

Where:

$$\delta_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t)$$

Chapter 3

The PIKo simulator

3.1 Initial state of the simulator

The PIKo simulator is part or a SINTEF research project on the development of a pipe-inspection robot that has resulted in the snake robot PiKo. This project was also the background for an EiT project in 2009 which resulted in the PiKo simulator. In the spring of 2010 it was further improved with more realistic simulations, new sensors and a new control strategy based the Follow-the-Leader [9].

This chapter describes how to install the simulator for development, how to control the modelled robot under simulations and how to read its sensors.

3.2 Tutorial

3.2.1 Installation

At this stage, PIKo is only available as source code but it can be built on both Windows- and Linux-platforms. This section explains the steps necessary to run it. PIKo relies on 2 external open source libraries: the Open Dynamics Engine (ODE [10]) and the 3D engine irrlicht [11]. These libraries are necessary to run the simulator. Provided with the source code there were some short guidelines for the installation which are incomplete for the Microsoft Visual Studio environment. Following are the instructions for both platforms for the version developed during this thesis-writing.

Preliminary This step is common for both platforms. Extract the content of simulator.zip to a directory of your choosing. You will end up with the following directory structure:

```
DIRECTORY/simulator/
---ea/
---media/
---simulator/
```

Linux

1. Install the packages *libode-dev* and *libirrlicht-dev* from your package-manager or download the source code or prebuilt binaries from ODE[1] and Irrlicht[2] and follow their instructions respecting the following directory structure.

   ```
   DIRECTORY/simulator/
   ---ea/
   ---irrlicht
   ---media
   ---ode
   ---simulator
   ```

2. Compile the simulator from `simulator/simulator` using the Makefile.

   ```
   $cd simulator/simulator
   $makefile
   ```

3. Run the simulator.

   ```
   $./a.out
   ```

4. To compile and run the *EA* project follow the same steps but from `simulator/ea`.

[1] http://www.ode.org
[2] http://irrlicht.sourceforge.net

Windows

1. Download the *Windows*-version of the libraries ODE [10] and Irrlicht [11] from their websites.

2. Install ODE

   (a) Extract the content of the ODE package to DIRECTORY\simulator\ode. If you get something like "ode-VERSION" rename it to "ode".

   (b) Open a console and move to DIRECTORY\ simulator\ode\build and execute the command `premake4` for your Visual Studio version (at this moment it support versions 2002 to 2008). In this case VS2008 is used. The `with-demos`-option is necessary because PIKo needs the *drawstuff* library which is build through this option.

   ```
   premake4 --with-demos vs2008
   ```

   (c) The previous command will have created a new directory. Move into this directory, open the VC++ solution file and build it.

   (d) Add to the Windows PATH-variable the path to where the *dll* file was created (probably `..ode\lib`)

3. Install Irrlicht

   (a) Extract the content of the Irrlicht package to DIRECTORY\simulator\ode. If you get something like "irrlicht-VERSION" rename it to "irrlicht".

   (b) Create a new directory inside `irrlicht\include` called `irrlicht` and move all the header files to the new directory. This step is necessary to allow the project to be compatible with both platforms.

   (c) Add to the Windows PATH-variable the path to where the *dll* file is (probably `..irrlicht\bin\Win32-VisualStudio\bin`)

4. To run the simulator open the project file in `simulator\simulator`, build the project and select the menu item *Debug->Start debugging* or press the F5 key.

Appendix A.1 refers other subject alien to the typical installation process.

3.2.2 Reading of sensors

Types of sensors:

1. Pressure sensors The pressure sensors implemented in the simulator can be consulted through the function:

   ```
   void Train::getJointForces(int jointNumber, double *up,
   double *down, double *left, double *right)
   ```

   For a given joint, this method returns the total force perceived by each of the sensors of the joint.

2. Distance measurements. The PIKo simulator provides an extensive array of sensor for measurement of different kinds of distances.

   (a) The distance from the nose of the robot is consulted through the function:

   ```
   double Train::getDistanceNose(dGeomID space)
   ```

   (b) The sonar distance measurement is provided by the function:

   ```
   double Train::getSonar(dGeomID space, double openingAngle)
   ```

   (c) IMU-data. ODE provides inertial measurements of the modeled bodies. This information may be read with:

   ```
   void Train::getIMU(double **position, double **direction,
   double **linearVelocity, double **angularVelocity,
   double **linearAcc, double **angularAcc)
   ```

   At this moment, the linear and angular accelerations are not implemented yet.

   (d) Laser measurement. This function returns an array of distances in a horizontal plane.

   ```
   void Train::getLaserScanner1D(dGeomID space, int points,
   double openingAngle, double* result)
   ```

   The resulting array contains *points* measurements equi-distributed in the given opening angle.

   (e) The Time-of-flight camera modeled in the simulator works combining 2 laser measurements. For each horizontal point it does a vertical scan.

```
void Train::getTimeOfFlightCam(dGeomID space, int
verticalPoints, int horizontalPoints, double
verticalOpeningAngle, double horizontalOpeningAngle,
double **result)
```

(f) Travelled distance along the path of the first cart as the sum of the distance of each step.

```
double Train::getTravelledDistance(void)
```

(g) Wheel distance. This value is calculated from the wheel rotation which is more realistic than the travelled distance although it is affected by wheel-spin. Only one of the wheels of the first cart is used for this measurement.

```
double Train::getWheelDistance(double timestep)
```

3.2.3 Control

The simulator use a ”Follow the leader”-algorithm control strategy which propagates backwards the settings from the first cart to the rest of them when it’s moving. This means that given a point in the path, every cart will do the same actuations when it passes over it.

It is possible to control the robot with using the following keys:

- W: forward
- S: backward
- Q: lift head
- E: lower head
- A: turn left
- D: turn right
- R: reset robot

It should be talen into account that the simulator will not prevent the user from trying movements where the resultants forces are to big resulting in an ”exploding” robot.

To allow more complex and elaborate movements it is necessary to program it giving inputs for each joint and motor. The functions available for

this are:

```
void Train::setJointAngle(int jointNumber, int direction, dReal angle)

void Train::setJointTorque(int jointNumber, int direction, dReal torque)

void Train::setJointVelocity(int jointNumber, int direction, dReal velocity)
```

To consult the current values for these settings. The simulator provides the following functions:

```
double Train::getJointAngle(int jointNumber, int direction)

void Train::getIMU(double **position, double **direction,
double **linearVelocity, double **angularVelocity,
double **linearAcc, double **angularAcc)

void Train::getJointForces(int jointNumber, double *up,
double *down, double *left, double *right)
```

3.2.4 Performing simulations without visualizations

The visualization part of the simulator is independent of the dynamics simulation. When you want to test out a new strategy you will typically create a new set of classes to implement it and you will have to create an instance of the simulator somewhere using a call similar to

```
simulator = new Simulator(-1.5, 0, 0.2, 0.05, visual);
```

where *visual* is a boolean parameter which will be `true` if you wan graphic visualization and `false` otherwise.

The simulator will not provide you with sensor information by itself so you will also have to consult them manually with the methods described earlier in this chapter. To get the necessary `train`-instance you can use the function:

```
Train* Simulator::getTrain()
```

3.3 Modifications

Using the actual PIKo robot to study the use of reinforcement learning strategies would prove very time-consuming. In part because it would require to get to know in depth PIKo's programming, partly because it would be harder to systemize the learning process and partly because PIKo was not in working order in the beginning of this work. These factors, among others, made that from the beginning this thesis' work would be based on the existing PIKo-simulator. However, in it's original stage the simulator lacked features necessary for the use of reinforcement learning. This section describes the modifications introduced to the source code to achieve the desired functionality.

3.3.1 The Artificial Neural Network (ANN) class

To be able to use a Radial Basis Function network (RBFN) as our function approximator it was necessary to implement it first. Based on the information already described in subsection 2.3.1 we implemented an RBF network in the simulator. The resulting RBF network class is composed by:

- An array with N_i elements for storing of the values of the nodes of the input layer
- An array with N_h elements for storing of the values of the nodes of the hidden layer
- An array with N_h elements for storing of the values of the nodes of the normalization layer
- An array with N_o elements for storing of the values of the nodes of the output layer
- An array with $N_h x N_o$ elements for the weights of the edges between the normalization layer and the output layer

N_i, N_h and N_o are parameters given to the construction method of the class. At this point there are not provided any methods to change this values.

The motivation behind the introduction of the normalization layer is to provide an implementation that can be used in a later development of this work if necessary but is ignored in this stage.

The Radial Basis Function used is:

$$RBF(\theta) = e^{-\frac{\|\vec{\theta} - \vec{c}\|}{2\sigma^2}} \tag{3.1}$$

Where $\vec{\theta}$ is a vector with the input parameters, $\vec{c}$ is a vector with the components of the center of the node and σ is the width of the Gaussian of the function and is defined in the header file. Because of the limitations described in subsection 4.1.1, the function that computes this RBF function is limited to an input vector with only 2 components.

The computation of the network follows the diagram in figure 3.1:

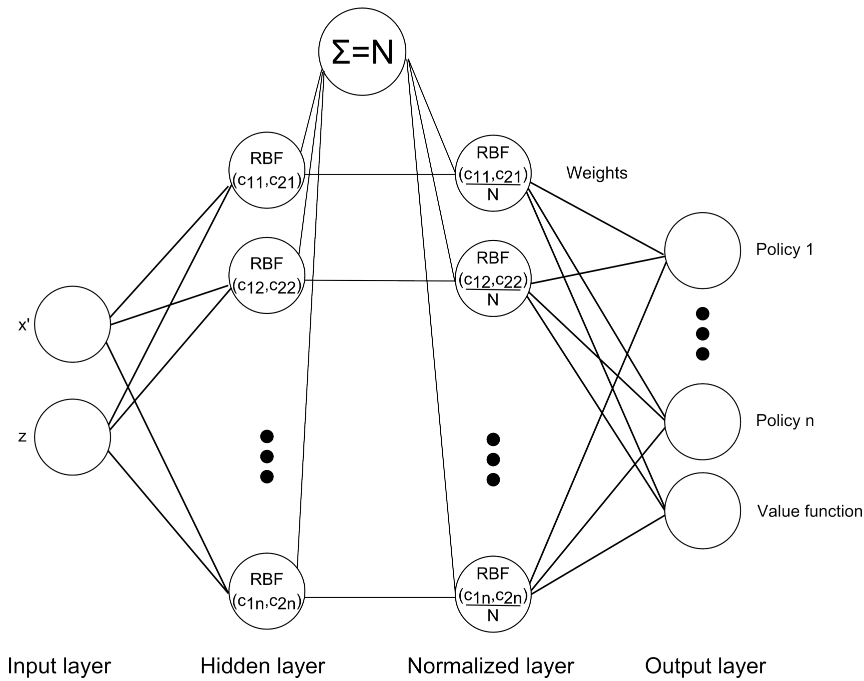


Figure 3.1: Diagram of the implemented RBFN. Notice that N designates the sum of the values of the nodes of the hidden layer

For the training of the network we added a function, *updateWeights* that updated the the weights according to:

- The output being trained, *output*
- The value towards it's trained, *expected*
- The learning rate, *rate*

This function calculates the difference between the output value computed by the network and the value that it is expected to compute and then multiplies it by the desired learning rate. Then it calculates the amount by which each weight should be modified and updates the weight.

Now, if we remember the expression that is computed by the network (equation 2.1) we get that:

$$
\begin{aligned}
RBFN(t) &= \sum_i w_i K\left(\frac{x(t) - z_i}{\sigma_i}\right) \\
&= \sum_i w_i g\left(\frac{\|x(t) - z_i\|}{\sigma_i}\right) \\
&= w_1 g\left(\frac{\|x(t) - z_1\|}{\sigma_1}\right) + w_2 g\left(\frac{\|x(t) - z_2\|}{\sigma_2}\right) + \cdots + w_n g\left(\frac{\|x(t) - z_n\|}{\sigma_n}\right)
\end{aligned}
\tag{3.2}
$$

As we can see in 3.2, the value of the node gives as this amount in which the weight affects to the total result.

The resulting algorithm is:

error ← rate*(expected - outputLayer[output])
for all node "i" in normalizedLayer **do**
  update ← error*normalizedLayer[i]
  weights[$weight_{i,output}$] ← weights[$weight_{i,output}$] + update
end for

where $weight_{i,output}$ is shorthand for the index of the weight going from node i to the given output.

Other functions provided allow writing and reading weight values to/from a file. These functions are:

- dumpWeights. Writes to the file "weights.txt" the values of the weights corresponding to the output given as parameter
- dumpWeightsAll. Writes to the file "out.txt" the values of all the weights of the newtwork.
- restoreWeights. Sets the values of the weights of the network to those read from the file "weights.txt"

The write functions do not add any metadata of the network such as the number of weights. Correspondingly, the restore function does not check if the file is suitable for the current network leaving the responsibility to the user.

Other miscellaneous functions in the class are:

- horizontalDistance. Given a position it returns the module of the vector from $(0, 0)$ to (x, y) with sign. This function is used for the simplification of the state-space described in chapter 4.

Testing the ANN

The ANN class is essential for the proper functioning of the learning. To assure its correct behaviour it was trained to approximate a complex function. "Complex" in this context means a differentiable function with frequent monotony changes. The selected benchmark function was:

$$f(x) = \frac{-2cos(\frac{1}{x^2})}{x} + 2xsin(\frac{1}{x^2}) \quad (3.3)$$

The testing also turned positive to understand better how radial basis functions and training is done. After distributing the centres along the x-axe (giving them coordinates of the type (x_i,0 and avoiding x=0) with different separations and learning rates the plot of figure 3.2 was obtained.

The training process consisted of sweeping the x-axe indefinitely. The training was aborted when the all the difference with the the model function was less than 1% at each one the centres. This process was repeated 10 times.

3.3.2 The Reinforcement Learning class

The *rl* class implements the functions and data structures associated with the reinforcement learning algorithm. If we remember how the reinforcement algorithm is and particularly the CACLA version of it there are the following main parts:

1. Selection of action that should be taken
2. Perform the action
3. Observe the results of the action and perform the corresponding updates

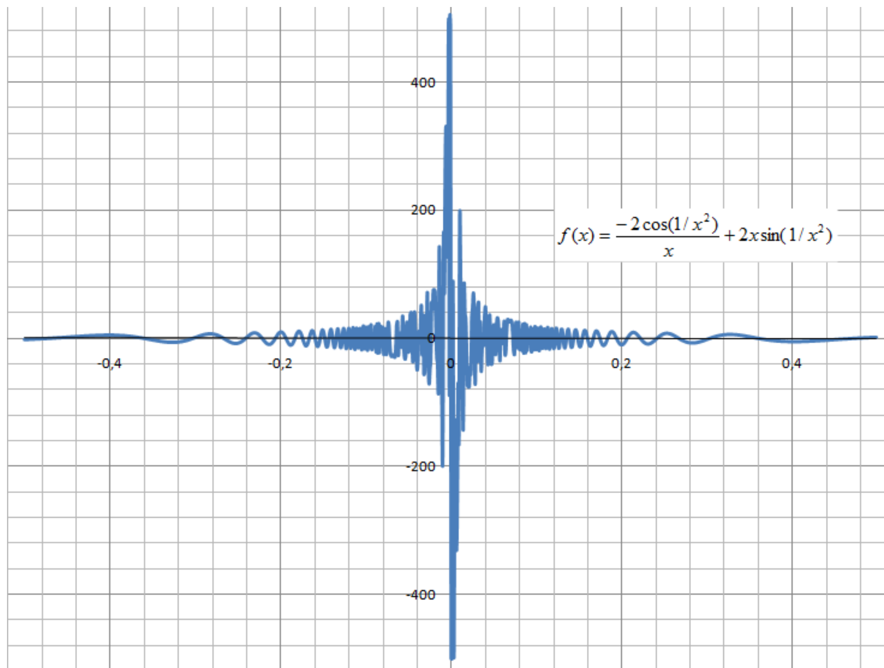


Figure 3.2: Plot of the benchmark function from equation 3.3

In this section we will see how this functionality has been achieved by the functions: *preStep*, *nextAction* and *postStep*.

The "preStep" function

The *preStep* function addresses point 1 of the parts described in subsection 3.3.2. The involved actions are:

- Updating the variables that store the state previous to an action
- Fetch from the network the next action and the value of the current state
- Invoke the the function responsible to perform the action

The information of the previous state are the position, height and value of this state. The third point involves another function responsible for the exploration of new actions. This task has been done in different approaches as we will explain in chapter 4 but at this point of the report we will just name the different versions.

nextActions This version uses Gaussian exploration. Gaussian exploration involves retrieving a random generated number from a Gaussian distribution centred on the greedy action retrieved from the function approximator and with a variance which decreases over time for convergence of the policy.

nextActions6 This version uses an ϵ-greedy strategy for the exploration problem. This strategy involves taking the greedy action (the one retrieved from the function approximator) with probability 1-ϵ at each step. For convergence reasons this, ϵ is gradually decreased over time.

actionAngle This version merges the exploration and the action controller (to be seen in subsection 3.3.2). First it performs Gaussian exploration as described earlier and then it sets the necessary variables for the action.

The "actions" function

The *actions* function receives an index for the action it has to perform acting as an action controller. These actions and indices are:

1. Decrease the angle of a joint by a fixed amount
2. Increase the angle of a joint by a fixed amount
3. Decrease the speed
4. Increase the speed
5. Reset the train to its original configuration. Used when at the end of a simulation episode.

With "original configuration" we mean repositioning it at the same point as in the beginning of the simulation episode and reset its joint angles and speed to 0.

Since this controller is a version of the FTLController already implemented and because we aimed to use the FTL strategy in the learning process, the controller also includes the code necessary for the use of this strategy.

The postStep function

The *postStep* function is responsible of point 3 of the parts described in subsection 3.3.2. This function calculates the value of the new state, the rewards and the corresponding temporal difference error and update the RBFN according to this error.

3.3.3 Resetting the robot

The learning process requires a variable number of iterations, or episodes. In each episode the robot takes a certain number of actions which are then evaluated to update (or train) the function approximator. Given this thesis' objective (lifting the front module, or "head", as much as possible), the starting point of each episode can theoretically be arbitrary wherever in the modelled space. Under these circumstances the function approximator would learn how to lift its head wherever in the scenario. However, this approach would require large amounts of memory and time, the first because of the need to map all the space to nodes and the latter because of the need to train all these nodes. Chapter 4 describes the decisions taken to minimize the requirements. In this section we address the consequence of one of this decisions which involves resetting the robot to its initial state.

We consider as original state position (0,0,0), with all the joint angles at 0 degrees and speed 0 and an empty event queue (used by the FTL algorithm). This is done by the new function *resetTrain* in the Train class which also initialize more low-level variables such related to torques and the PID-regulator implemented. To check its proper behaviour we made a version of the FTL controller already implemented and added a new keyboard event for the resetting. The tests were made at low speeds and looked favourable.

Later it was discovered that the model rises up from the ground (figure 3.3 when it is repositioned for each new simulation episode and it's not solved by positioning it above ground level.

Furthermore, later it was discovered that there are more factors involved which is the case of stored inertia in the physics engine. This behaviour had not been observed before because the physics engine has been treated as black box. This stored inertia showed to be altering the initial state producing unwanted movements in the joints and linear speed of the model. The workaround that has been used later consists of a "stabilizing loop" which is

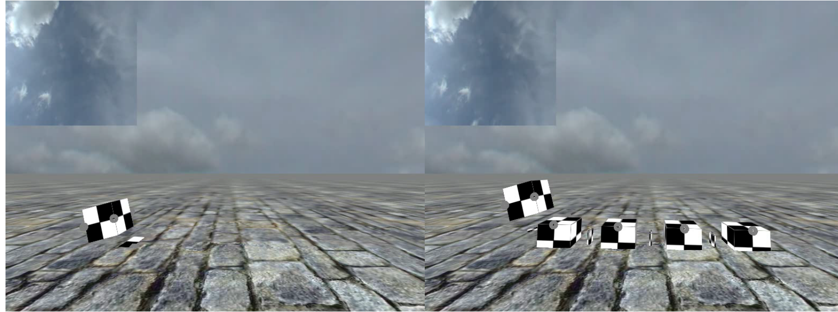


Figure 3.3: Screen captures showing how the model rises from the ground instead of being positioned over it as expected

run when the model is reset. This loop performs simulation steps and resetting of the aforementioned parameters as long as there still is movement or the angles are different from 0. If the stable state is not reached in a certain number of iterations the resetTrain function is invoked again.

3.3.4 Simulation without visualization

The simulator, or better said the simulator class, allows to perform simulations without visualization as it was before this thesis started. The constructor method for the simulator requires a parameter which indicates if it's going to allow visualization or not. The only issues was that if you use the any of the controllers provided that rely on user interaction the interaction requires the visualization to be active and that the updating of the "head camera" of the robot is made directly in the main loop. Now the selection of visualized simulation is set by a constant, VISUAL, defined in the programs main file. This constant is used in the call to the simulators construction method and it protects the controllers the parts of code that requires visualization to be active.

There has also been included functions that write data about the running simulation to files allowing easier monitoring of the simulation. this functions are:

- tracing. Writes to a file the position of the head and the values of the vertical angles of the joints.
- heights. Writes to a file the to values intended to be the height and the maximum height and the values of the vertical angles of the joints.

3.3.5 Unstability

It was observed early that the simulated model could suddenly start behaving randomly starting to spin around the screen. The cause was traced to the presence to strong forces in the simulated components. To minimize this unwanted effects there was introduced limits to the values of speed and angles which seemed to solve the problem. Figure 3.4 shows an example of the aforementioned behaviour.

3.3.6 "External" forces

Late in the development it was detected that a module in unstable balance, e.g. positioned vertically facing upwards, received the signal to decrease or increase slightly their angle this would make the joint travel all its range as seen in figure 3.5.

This behaviour has been observed to be caused because the call to modify an angle is given this only indicates to the joint that it has to move in a certain direction and then gets overridden by the physics of the model. In this case, the workaround has been to introduce an array with the target angles for the joints and setting them to these values before each simulation step. This role is performed by the *angles[]* array and the *refreshAngles* in the RL class.

3.3.7 Stabilizing the actions

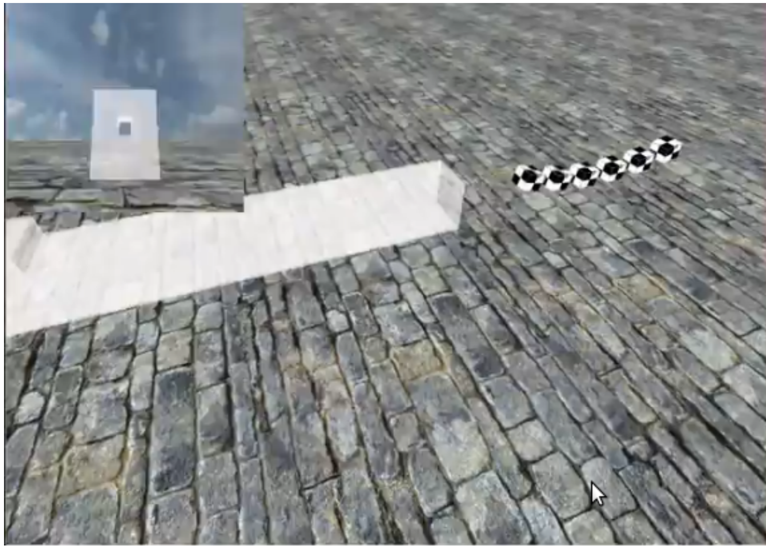
In sections 3.3.3 and 3.3.6 we have seen that one simulator step is not enough to complete an action. To allow the completion of an action, in particular angle settings, more simulator steps are needed.

SIGNAL_FREQ is a new constant added to the main program. Its role is to define the frequency with which a new action is taken or, with other words, for how many simulator steps we keep active the same action. This translates to keep the same values for the angles of the joints and to keep the set speed and is informed to the *preStep* function with boolean parameter that informs if a new action should be tried during that cycle or if it should continue with the old one. Figure 3.6 shows this scheme as a chronogram.

3.3.8 Simulation setup

The setup for a simulation session can be done through the following constants:

- Number of episodes. Set through the constant EPISODES defined in *main.c*
- Number of steps. Set through the constant STEPS defined in *main.c*
- Simulation with visualization. Set through the constant VISUAL in *main.c*. Set to 0 if no visualization is wanted and t another value otherwise.
- Timestep of the simulator. Set through the constant TIMESTEP defined in *main.c*
- The number of nodes are defined through the constants V_FEATURES and H_FEATURES in *RL.h*
- The distance between nodes is defined through the constant DIST in *RL.h*
- The horizontal coordinate from which the horizontal nodes start is defined through the constant START in *ANN.h*



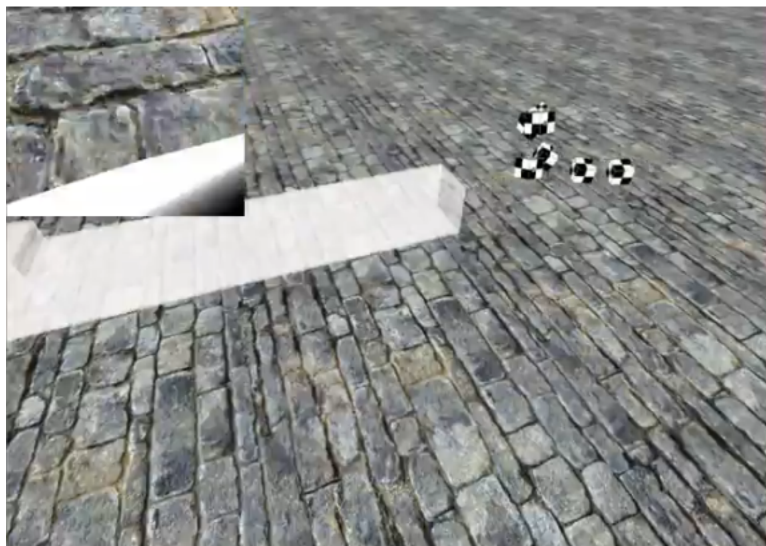


Figure 3.4: Screen captures showing random behaviour.

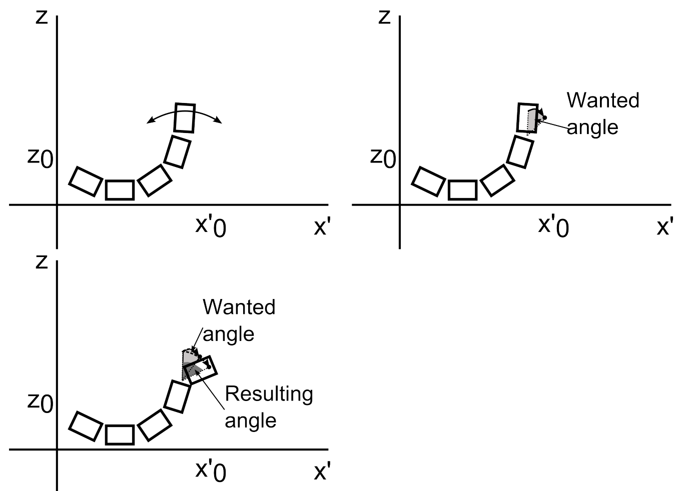


Figure 3.5: Diagram of the observed behaviour when a joint is subject to gravity

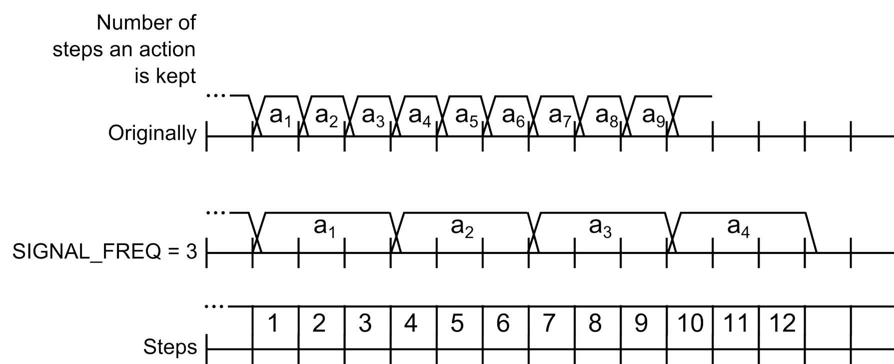


Figure 3.6: Diagram of how the settings of an action are sustained.

Chapter 4

Implementation

The objective of this study is to infer a motion primitive that will endow the snake robot PIKo to lift it's head (front module) as much as possible using a reinforcement learning. This chapter presents the strategies followed to attain this goal and the problems than surged during the process.

The development was following an incremental-evolutive cycle which made that some problems or incidents appeared during testing and experiments. This incremental identification of problems caused the re-orientation of the approach causing that earlier approaches were abandoned before solving the identified problems.

4.1 Initial setup

The experimental setup required the selection of:

- A state representation
- A reinforcement learning method
- A function approximator

4.1.1 State representation

The purpose of PIKo is the inspection of pipelines which means that the realm which it interacts is composed of the 3 dimensional space it will be inspecting. The original intention was to base the learning on the existing Follow-the-Leader (FTL) algorithm already implemented in the simulator

and since this algorithm propagates backwards the settings for joint angles and speed of the first module depending of the travelled distance, the state is represented by the fist module's coordinates (x_0, y_0, z_0) in space. Furthermore, since the action of lifting the "head" only requires to components (horizontal and vertical) the space of interaction was correspondingly reduced leaving out the depth dimension. Control of speed is done throgh the high level controls provided although the simulator gives the possibility of low level adjustment of the speed of each module. However, this would increase in a great degree the complexity of the learning process.

$$(x', z) = \left(\frac{(x+y)}{\|(x+y)\|}, z \right) \tag{4.1}$$

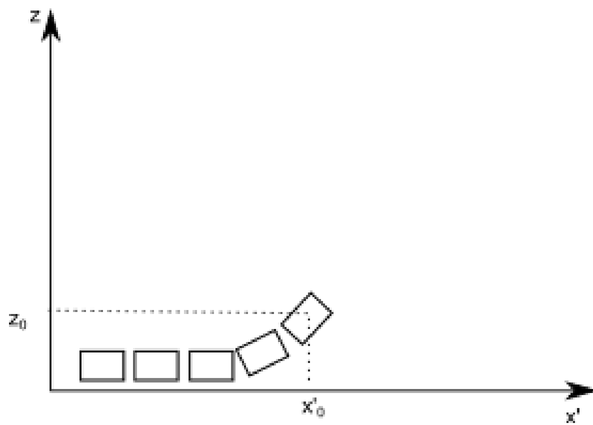


Figure 4.1: Representation of the PIKo robot in the state-space

For flexibility, the horizontal component is a projection x' of the x and y components over a normalized vector with origin in the simulators point $(0, 0, 0)$ to the first module's coordinates (x_0, y_0, z_0).

4.1.2 Reinforcement Learning Method

Of the different reinforcement learning methods presented in chapter 2 it was considered that the most suitable would be CACLA based on its capability to handle continuous spaces (as well as discrete ones [12]).

The CACLA algorithm in pseudo-code is as follows:

Initialize all actions A_0 for all states
Initialize values V_0 for all states
Select state, s_0
for each step t=1,2,3... **do**
  Select a_t
  Perform a_t, observe r_t, s_{t+1}
  if s_{t+1} is terminal **then**
    $V_{t+1}(s_t) \xleftarrow{\beta_t} r_t$
    Select new s_{t+1} (set the configuration for the next episode)
  else
    $V_{t+1}(s_t) \xleftarrow{\beta_t} r_t + \gamma V_t(s_{t+1})$
  end if
  if $\delta_t > 0 and the action was exploratory$ **then**
    $A_{t+1}(s_t) \xleftarrow{\alpha_t} a_t$
  end if
end for

For this project, the *actor*, A, and *critic*, V, will be stored using a function approximator presented in the next section. The initial configuration will be the same for each episode which is the initial position of the robot when you start the simulator. γ is a discount factor while α_t and β_t are the learning rates for the *actor* and the *critic*. The reward r_t appears in the Temporal Difference error, δ_t.

$$\delta_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t) \tag{4.2}$$

For the selection of the action for each step, a_t, was made using an exploration-exploitation balancing strategy depending on the experiment. Then the internal data-structures of the simulator are set according to the action the simulator advances. When finished, the evaluation of the new state is made and the updates to the critic and the actor are performed if necessary.

4.1.3 Function Approximator

A Radial Basis Function Network (RBFN) was selected as function approximator because it scalability and local approximation capabilities. The scalability proved to be useful to reduce the simulation time depending on the different scenarios.

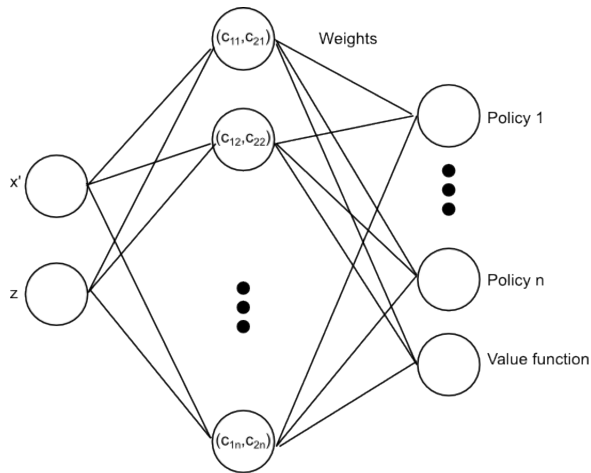


Figure 4.2: Diagram of the proposed RBFN

The RBFN which was used consist of an input layer with 2 nodes corresponding to each of the coordinates of the first module, one hidden layer with the centroids and an output layer with nodes for each policy and the value function (figure 4.2).

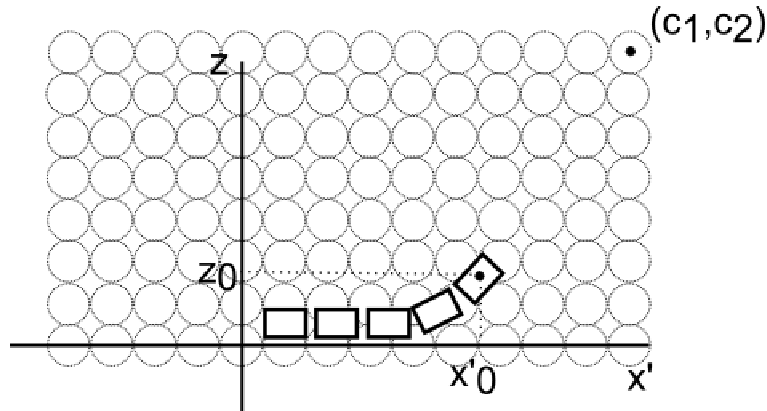


Figure 4.3: Mapping of the RBFN to the workspace

The centroids are mapped to the workspace following a grid of equidistant points as seen in figure 4.3.

4.2 First approach

The first approach made use of the FTL-algorithm already implemented in the simulator. Taking this as starting point the set of actions selected was:

- decrease the speed a fixed amount a_1
- increase the speed a fixed amount a_2
- decrease angle a fixed amount a_3
- increase angle a fixed amount a_4

This way a_1 and a_2 belong to one dimension and a_3 and a_4 to another one. We define one policy for each one of these dimensions and alternate between them when we decide an action.

Because of the use of a_1, a_2 the centroids had to cover a large area of space which added to desirable high resolution of the state-space required a large amounts of nodes (around 2500) penalizing the simulation time. Remember that the implementation of the FTL-algorithm requires the robot to travel a certain distance (the distance between to consecutive modules) to propagate its settings to next module.

Besides, this incapability to directly control the rest of the joints of the robot rests manoeuvrability in closed spaces as pipelines.

The strategy for exploratory actions was originally a Gaussian exploration where de probability to perform action a in step t is:

$$\pi_t(s_t, a) = \frac{1}{2\pi\sigma} e^{-(a - Ac_t(s_t))^2/2\sigma} \tag{4.3}$$

where σ is the with of the Gaussian, $\pi_t(s_t, a)$ denotes the policy and π is the constant.

This strategy did not seem to work as expected so it was later replaced by an ϵ-strategy where ϵ denotes the probability of making an exploratory action. This change in strategy required some changes in the code. In particular it was now necessary to store information about if the action was exploratory or not.

The output was interpreted as follows:

$$\begin{cases} A_t(s_t) > 0 & \text{increase speed/angle} \\ A_t(s_t) = 0 & \text{nothing} \\ A_t(s_t) < 0 & \text{decrease speed/angle} \end{cases}$$

Different rewarding strategies where tested. One of the first ones was giving an increasing reward each time the robot surpassed a milestone. This was inspired by the labyrinth-problem presented in [8] where rewards only are awarded if the agent finds its way to the exit. It was also awarded a big negative reward if it escaped the boundaries of the area covered by the centroids.

This strategy seemed to work satisfactorily in the first stages of the training, reaching the physical maximum, but would not converge to this solution.

Results presented in chapter 5.1 inspired another reward strategy where the reward received was the actual height of the "head" but this showed to be too "generous" as it always was positive and would eclipse the temporal difference error provided by the value function. The final strategy was to use the height difference between the current step and the previous which complies to the basic idea that increase the height is good.

During this stage, the actions made to reset the model of the robot for each new episode comprised repositioning it to the original location, reinitialize the angles of the joints to 0 radians and clear the queue of events used by the FTL-algorithm and variables that kept the speed and the angle of the front-most joint.

4.3 Second approach

After observing the results from the first approach presented in chapter 5 it was decided to change the approach to the problem.

This time we turn away from the FTL strategy and focused on the individual joints. The motivation for this was that just modifying the angles of the joints it would be possible to reach maximum height by moving the centre of masses to achieve a balance point that let the snake achieve a greater height by means of an S-shape. Figure 4.4 explains this reasoning.

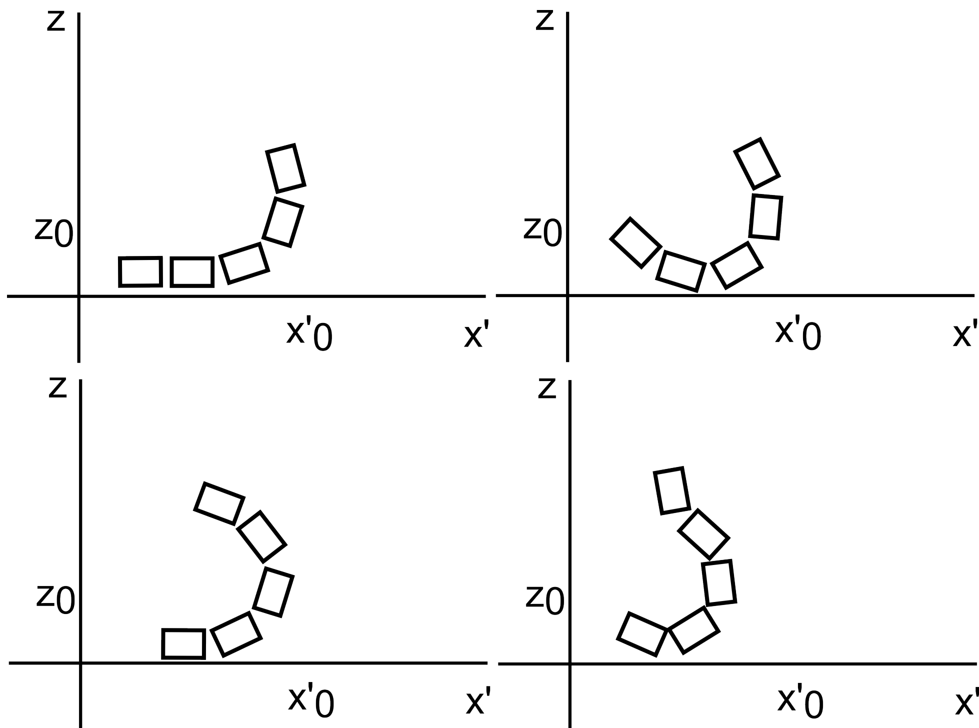


Figure 4.4: Sequence of steps oriented to achieve an S-shape

The number of outputs of the network was increased to adapt to the new learning goals which is training the different joints independently. Since actuating on all the joints at each step, i.e. having a vector of actions, would make it hard to isolate which of these actions was the main contributor to reaching a better state (reaching a greater height). For this reason it was decided that at each step only one joint would be affected. The way to achieve this functionality was by introducing a counter of steps. This counter would then be the remainder of this counter divided by the number of joints being trained, $counter\%|jointsTrained|$.

It was also discovered that the actions were not deterministic, i.e. performing an action did not really meant that this action would take place as expected. A consequence of this is that were not studying a Markov state either. The issue was observed while modifying the angle of a joint in a small degree resulted in a movement far greater than expected and motivated the modification addressed in subsection 3.3.6.

This issue motivated a more in-depth monitoring of the behaviour of the model showing that the learning process started before the model was stabi-

lized at its initial position. This issue has been addressed in subsection 3.3.3.

A final issue observed and described in the results 5.2 involved the limiting of the travelling range of the joints.

4.4 Third approach

Another final approach was used. This approach was more oriented to the continuous action space the problem present and inspired by article [13]. In this article the authors describe how to use reinforcement learning to improve the walking behaviour of a 4 legged walking robot. The relevant part for this thesis is the definition of the policies. Instead of defining the actions as fixed increments or decrements and learn if the action should be to increase or decrease the angle as we have been doing until now they define the policies as the angle the joint should present.

Turning to our set-up, this meant changing the action controller (subsection 3.3.2). Since we were not going to use the FTL strategy any longer and that we were not going to use the actions that affected the speed motivated the implementation of a new function for the decision and performing of actions (function *actionAngle* described in subsection 3.3.2.

Chapter 5

Simulation results

In this chapter we describe the results of the simulations for each of the approaches described in chapter 4. For each approach we describe the configuration of the simulator in terms of action space and goal of the approach and results of these.

5.1 First approach

The initial approach to the problem was using the *follow-the-leader*, or FTL, strategy. In this strategy only the first joint is directly manipulable and the values of its angles are transferred to the second joint when the travelled distance of the robot surpasses the length of one module (15 cm). Each time the robot surpass this distances, the settings are propagated backwards in a queue-like fashion. An important remark is that the counter for travelled distance is incremental and never decreases even if the robot moves backwards. A consequence of this is that the settings of a joint is never propagated back to the previous joint.

Given the use of the FTL strategy which use only speed and angle settings, the action-space used was:

1. Decrease the angle of a joint by a fixed amount
2. Increase the angle of a joint by a fixed amount
3. Decrease the speed
4. Increase the speed

Actions where selected alternatively between decrease/increase the angle and decrease/increase the speed using Gaussian exploration with a decreasing variance over time. According to this, the network was set to 3 outputs: 1 for the angle, 1 for the speed and 1 for the value function. If the policy output was negative this was interpreted as an decrease of angle/speed and an increase if it was positive. If the output was 0 no action was taken.

The RBF network was set to different spacings between centres ranging from 0.15 (length of one of the PIKo modules) to 0.01. The variance of the radial basis function of the centroids (nodes of the network) ranged from 1.42 times the spacing distance and 5 as a constant for all the centroids. The factor 1.42, an approximation to the square root of 2, is the minimum diameter of a centroid (or variance of its Gaussian) necessay to cover all the state space as seen in figure 5.1.

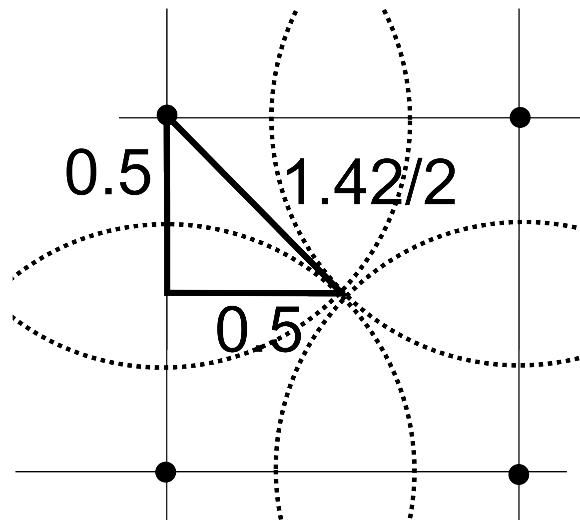


Figure 5.1: Diagram showing how using the value of the hypotenuse of a a triangle

The reward strategy used consisted of a series of "milestones", altitudes at which it received a big reward. At the rest of the positions it did not receive any reward.

The learning rates used where int the order of 0.0001 to 0.1 for both the policies and the value function.

The simulations showed that the robot learned that in a first phase the best action was to increase the angle of the first join, however, after a while

the FTL algorithm caused that this maximum opening angle was propagated to the rest of the joints giving the snake an "O"-shape where the head and the tail kept touching each other. When in this position, it was not able to straighten up its head.

A hypothesis for this behaviour is that once it have reached this "O"-shape the head is between two milestones and cannot infer that straightening the head upwards is perceived as better.

Since observing that the head could be "marooned" between two milestones the reward strategy was changed to be the height at each step and it was added another reward proportional to the "verticality" that started after achieving a certain height. This strategy proved to be quite bad showing no convergence at all. This is easily explained since any action will give a positive reward. And since CACLA updates the policy every time the TD-error is positive it would always update towards the last action taken.

The reward strategy was changed again to give a reward according to the difference in height achieved by each actions. The verticality criteria was kept. The learning converged to a state similar to the one achieved with the first strategy based on milestones without reaching verticality.

Altering the learning parameters did not affect the overall results. This is explained by the weight the rewards supposed and the low number of episodes simulated. This number was around 100 and 1000 with about 200 steps for each episode.

Another issue with the use of FTL is that the robot needs to advance through the scenario to propagate the settings increasing the size of the network needed to cover the space state and, as consequence, the time needed for the learning process.

5.2 Second approach

In the second approach the FTL algorithm was cast aside to limit the action space (and by that the number of nodes). Not using the FTL strategy means we no longer require to modify the speed of the snake leaving to possible actions to:

1. Decrease the angle of a joint by a fixed amount

2. Increase the angle of a joint by a fixed amount

The exploration strategy was changed to an ϵ-greedy strategy which started with an ϵ of 0.5 at was decreased over time. The motivation was that it seemed to make more sense to explore the opposite action when the output of the policy is interpreted so binary.

As reward strategy we kept the height differences from the first approach but stopped using the vertical criteria since the same goal should be achieved by increasing the resolution of the network by decreasing the spacing between centroids.

The simulations started with only the first joint active to check that everything worked as it should. It was detected that sometimes the learning converged to the maximum possible but other times it would go the other way around. The problem was traced to reset action after each episode provoking that the learning started before the model was stabilized on top of the ground. This problem has been addressed in subsection 3.3.3.

Another issue observed was that the servo in charge of the joint did not manage to achieve de desired angle of the joint being subject to the gravity of the modelled world. This issue motivated the modification described in subsection 3.3.6.

Once these issues were worked around and increased the number of episodes we started getting more consistent results as seen in figure 5.2.

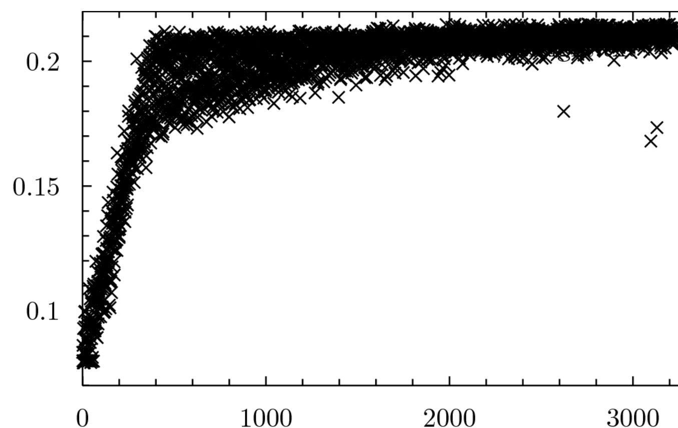


Figure 5.2: Height in metres achieved at the end of each episode with only the first joint active. X-axis denotes the number of episodes simulated.

However, if increased even more the number of episodes simulated we detected a consequence seen in figure 5.3.

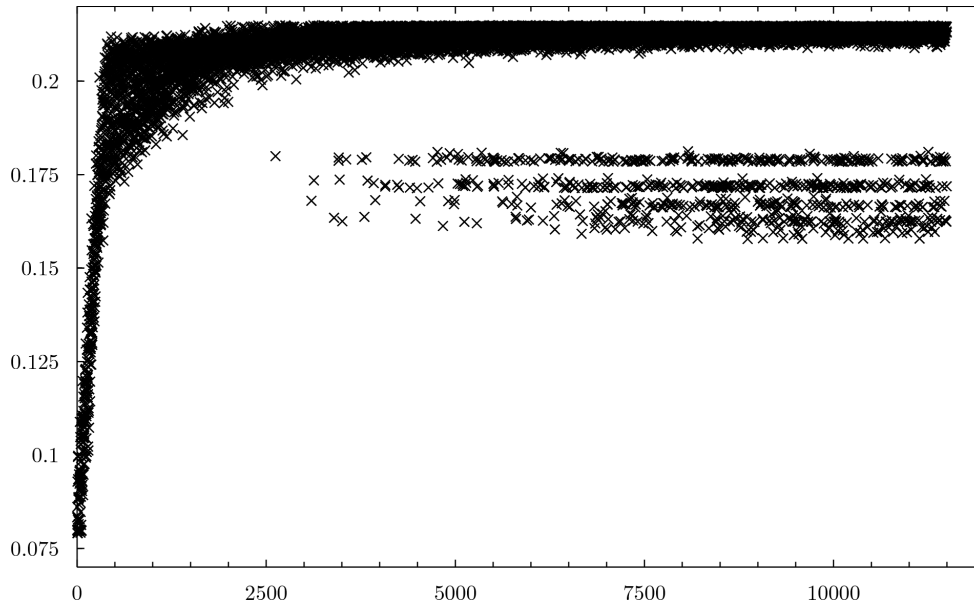


Figure 5.3: Height in metres achieved at the end of each episode with only the first joint active and increasing the number of episodes simulated. X-axis denotes the number of episodes simulated.

As we can see, the simulation converges to maximum possible height but after 2500 simulated episodes some strange results start appearing. By studying the path of movement of the head it was observed that after reaching the maximum height the head bounced violently. The explanation for this phenomenon is given to a too big speed of the head provoking a sudden bounce when it reaches the maximum range of the joint.

After capping the maximum range of the joint this problem seemed to be solved or at least not observed and simulations with 2 active joints were ran showing a typical behaviour seen in figure 5.4.

First, we can see that the final position of each episode converges to 2 different heights and slowly decrease. Second, after about 20000 episodes everything colapses. The first issue is traced to 1 of the joints being unable to converge to one angle. The second issue is more difficult to explain. A hypothesis is that it is related to the value function overriding the TD-error. Decreasing the learning rate to solve it or slow it down.

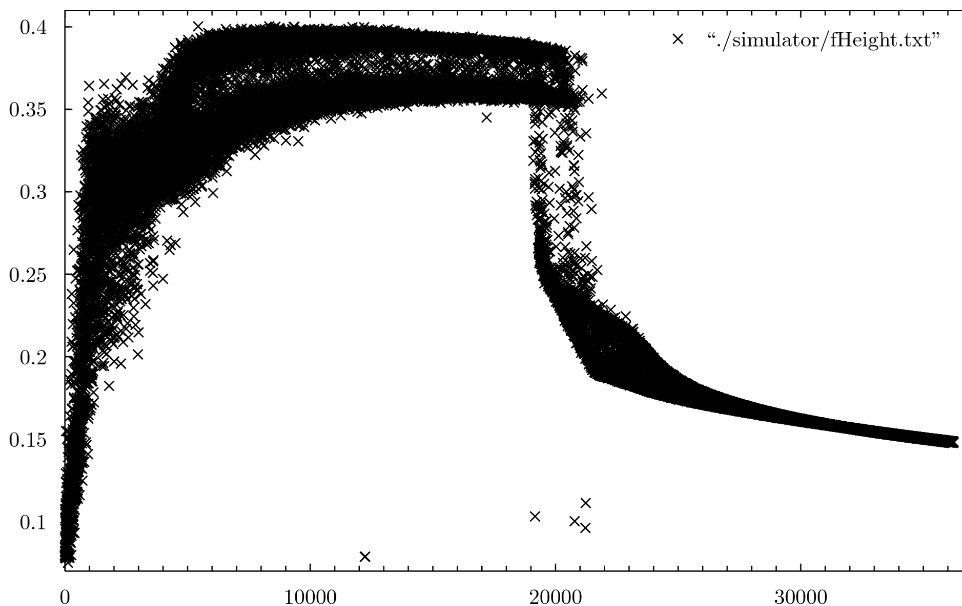


Figure 5.4: Height in metres achieved at the end of each episode with the 2 first joints active and increasing the number of episodes simulated. X-axis denotes the number of episodes simulated.

Simulations with 3 joints showed the following behaviour 5.5.

Here we see again how the model "converges" to two different positions. Again this was traced to one of the joints being unable to converge to a best position.

5.3 Third approach

In the third approach the policies were changed to learn an optimal angle instead of learning if it should increase or decrease the angle.

The exploration strategy was changed back to Gaussian exploration since it now made more sense to explore around an angle.

In this approach the modification described in subsection 3.3.7 for stabilizing actions was also introduced.

This alternative reached maximum possible height quickly and conver-

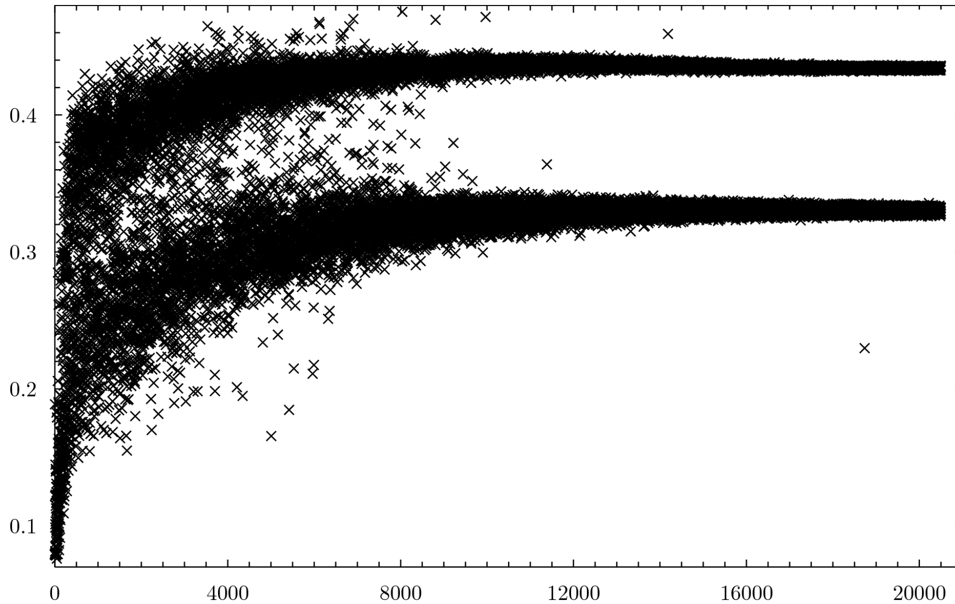


Figure 5.5: Height in metres achieved at the end of each episode with the 3 first joints active and increasing the number of episodes simulated. X-axis denotes the number of episodes simulated.

gence inside the range defined by the variance of the Gaussian exploration as seen in figure 5.6

However, if we check the path followed by the head we see that the maximum height is reached and then it starts loosing height as we can see in figure 5.7.

By increasing the number of steps considerably it was possible to reach a more stable result as seen in figure 5.8.

Most of these results where obtained using a network of 3 horizontal nodes by 5 vertical nodes with a spacing of 0.05 m. Increasing the number of nodes considerably slowed down an already long during learning process (using a 3x5 network for 7500 episodes of 2000 steps takes around 10 hours on the computer used for development).

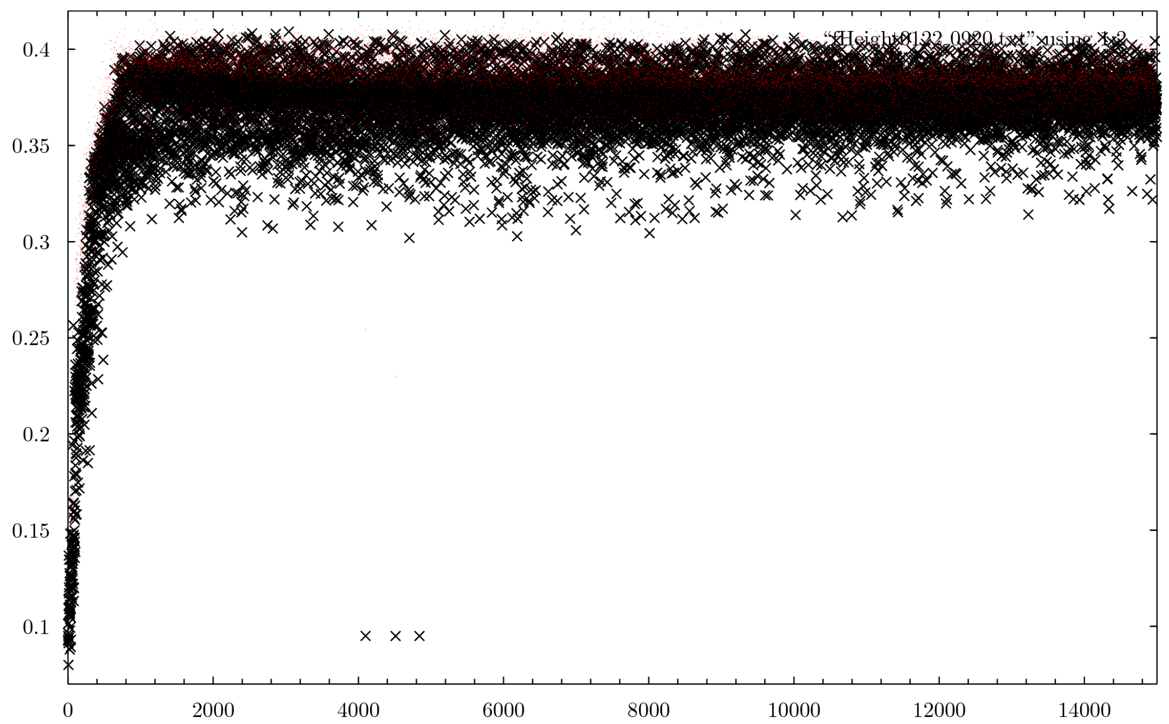


Figure 5.6: Height in metres achieved at the end of each episode with the 2 first joints active and increasing the number of episodes simulated. X-axis denotes the number of episodes simulated.

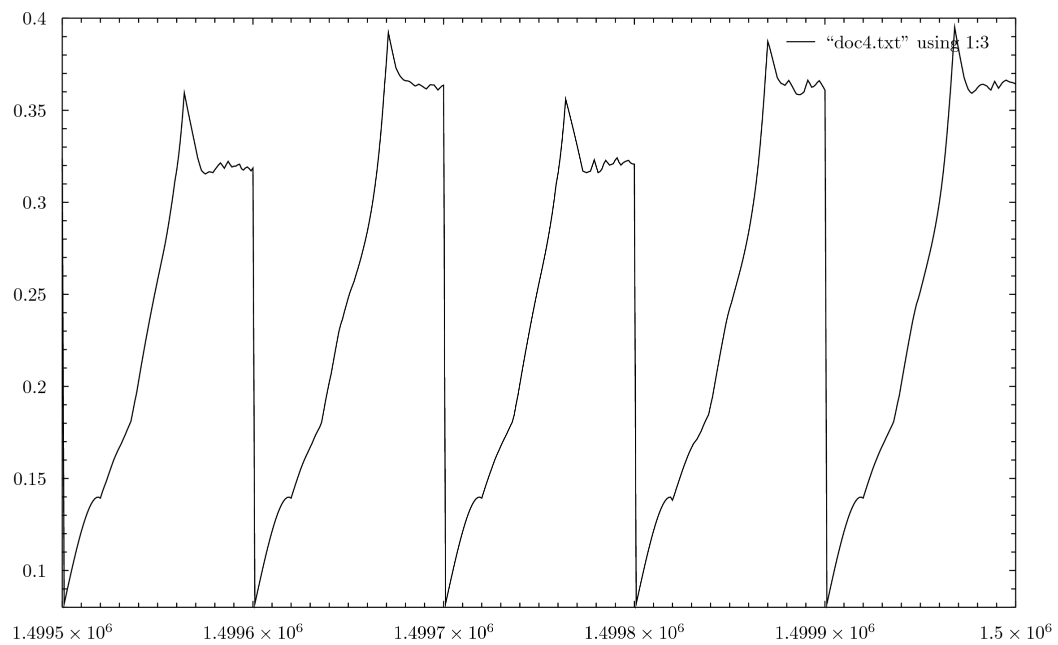


Figure 5.7: Height of the robots head during 5 episodes of 100 steps

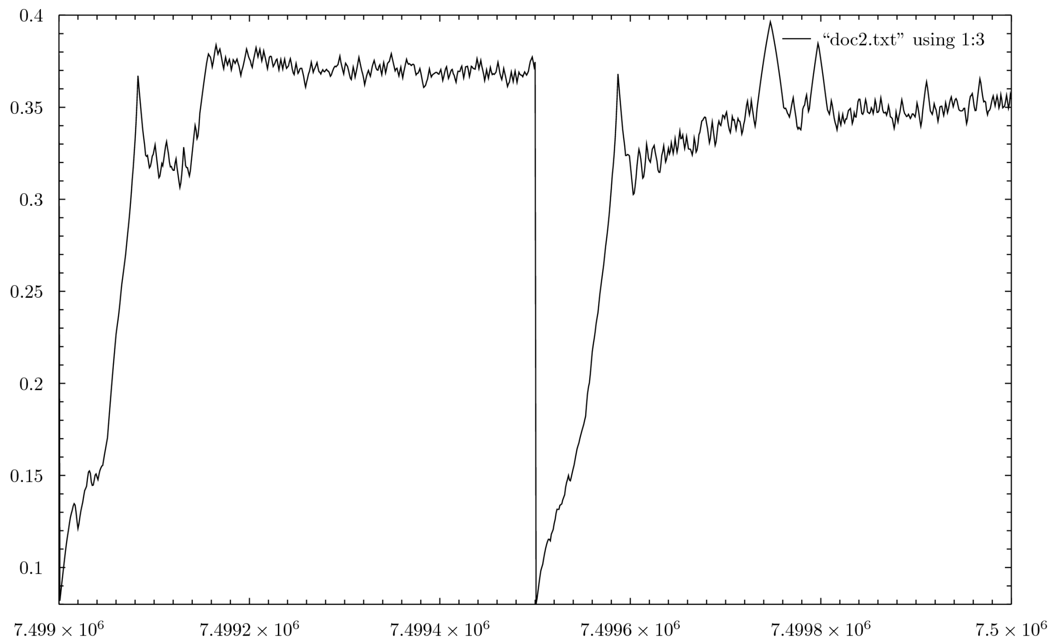


Figure 5.8: Height of the robots head during 2 episodes of 500 steps

Chapter 6

Discussion and further work

6.1 Discussion of results

Despite difficulties posed by at the moment unknown issues with simulator being used, each of the different approaches has showed to learn how to lift the head of the robot although the result is heavy influenced by the chosen parameters.

In terms of time required the most important parameter is the number of centroids used in the network. Not only because of the calculations that has to be done for each step but also because increasing the state space implies increasing the number of nodes that have to learn. In the simulations that have been done a 3x5 network proved to be able to learn a "good enough" result taking into account the limitations of its resolution. Narrowing of the number of nodes was done running simulations with many nodes and then observe which were the extreme horizontal and vertical positions reached.

The learning rates, both of the value function and the policies affect directly the number of times a state has to be visited to learn the optimal policy.

The reward strategy used is the main tool we have to indicate how good an action has been in the initial phase of the learning. This can be seen in the case of using the height difference between two steps. In the initial phase the value function's output is close to 0 and the policy learning is guided by the reward. Later, when the policy is being refined the height between two actions is much smaller and the value function which has been increasing its value gains importance in the compute of the TD-error.

The FTL strategy, because of its requirement of travelling a distance to propagate changes seems unsuitable for vertical movements and especially in confined spaces such as pipelines.

The earlier results are affected by the simulator issues discovered later and by shorter simulation sessions. Without the issues and longer simulation sessions the results might have been better.

The third approach which learn policies as specific angles learns faster while the others are limited by their discrete action sets.

The type of exploration strategy affects the outcome of the learning process and one strategy does not need to be good for every situation.

6.2 Further work

This thesis has focused on a basic reinforcement learning framework based on CACLA leaving out optimizations such as eligibility traces, more refined reward strategies.

Other optimizations of RBF networks such as adaptive number of nodes and adaptive width of nodes have not been tied either and could improve the time required for learning and the results (e.g. during the simulations there were situations where a larger resolution would have allowed more refined solutions).

The simulator need more refinement. During this thesis some problems were detected and workarounds for them proposed. However, improving the simulator was not in the scope of the topic and should be sorted out in better ways. Without a more realistic simulation environment the control strategies developed may turn unusable.

Other research lines are developing control strategies for more complex situations such as climbing up an L-bend or T-bends.

Chapter 7

Conclussions

Using reinforcement learning with an RBF network as function approximator for development of control strategies for serpentine robots is possible. Even using a basic set-up without more advanced features have proved to be able to learn a basic objective such as lifting the robot's head as much as possible.

The learning process is heavily affected by the parameters involved. Concepts such as the state and action space should be narrowed down from the beginning to allow for a more efficient use of time.

Various issues with the PIKo simulator have been detected and workarounds proposed. Better solutions for these issues would increase the quality of the simulations and reduce simulation time.

Chapter 8

References

[1] B. Guo, S. Song, J. Chacko, and A. Ghalambor, *Offshore Pipelines*. Gulf Professional Publishing, 2005.

[2] H. Streich and O. Adria, "Software approach for the autonomous inspection robot makro," in *Proceedings of the 2004 IEEE International Conference on Robotics & Automation*, 2004.

[3] B. Klaassen, H. Streich, E. Rome, and F. Kirchner, "Simulation and control of the segmented inspection robot makro," 2002.

[4] S.-g. Roh, D. Kim, J.-S. Lee, H. Moon, and H. Choi, "In-pipe robot based on selective drive mechanism," *International Journal of Control, Automation and Systems*, vol. 7, pp. 105–112, 2009. 10.1007/s12555-009-0113-z.

[5] H. Schempf, E. Mutschler, V. Goltsberg, G. Skoptsov, A. Gavaert, and G. Vradis, "Explorer: Untethered real-time gas main assessment robot system," in *1st International Workshop on Advances in Service Robotics, ASER03*, March 2003.

[6] S. Fjerdingen, P. Liljebck, and A. Transeth, "A snake-like robot for internal inspection of complex pipe structures (piko)," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

[7] S. Fjerdingen, E. Kyrkjeb, and A. Transeth, "AUV pipeline following using reinforcement learning," *SR/ROBOTIK 2010 : Proceedings for the joint conference of ISR 2010 (41st International Symposium on Robotics) und ROBOTIK 2010 (6th German Conference on Robotics)*, 2010.

[8] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[9] T. Hegge, T. Ingebretsen, U. Myhre, and V. kland, “EiT 2010 SINTEF: Piko, snakerobot.”.

[10] `http://www.ode.org`, “Open dynamics engine.”

[11] `http://irrlicht.sourceforge.net/`, “Irrlicht engine - a free open source 3d engine.”

[12] H. van Hasselt and M. Wiering, “Using continuous action spaces to solve discrete problems,” *Proceedings of the International Joint Conference on Neural Networks*, 2009.

[13] H. Kimura, T. Yamashita, and S. Kobayashi, “Reinforcement learning of walking behaviour for a four-legged robot,” *Proceedings of the 40th IEEE Conference on Decision and Control*, 2001.

[14] “Wikipedia.”

Appendix A

Simulator

A.1 Installation

Although building the original simulator in Linux was quite painless this was not the case on Windows. The following corrections had to be made:

- Correct the previous "include"-paths of the project.
- Find out how to let the code be built on both platforms.
- Increase the reserved stack size. Windows and Linux handle programs memory-requirements differently. While UNIX-like operating systems allocate more memory when it's needed this is different in Windows and it's necessary to adjust this in the project properties.

The 64 bits version of Windows 7 doesn't like the *dll* library built using the procedure described in chapter 3. A workaround is copying the *dll* to the *Debug* directory of the project. There are better ways to solve this problem which imply using a specific SDK from Microsoft, tune the Windows registry and more but this is not in the scope of this study. However, the curious can consult e.g `http://jenshuebel.wordpress.com/2009/02/12/visual-c-2008-express-edition-and-64-bit-targets/` to get an idea.

Due to problems with dependencies on MSVC++, the quickest way to get the EA-controller running is by creating an empty project and include all the files of the EA project and the simulator.