Department of Applied Informatics
Faculty of Mathematics, Physics and Informatics
Comenius University in Bratislava

Bimanual object grasping using a robotic simulator

Bachelor's Thesis

Andrej Sekáč

Bratislava, 2014

Katedra Aplikovanej Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského v Bratislave

Obojručné uchopovanie objektov v robotickom simulátore

Bakalárska práca

Študijný program: Aplikovaná Informatika
Študijný odbor: 2511 Aplikovaná Informatika
Školiace pracovisko: Aplikovaná informatika
Školiteľ: prof. Ing. Igor Farkaš, PhD.

Andrej Sekáč Bratislava, 2014

53781498

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Andrej Sekáč
Study programme:	Applied Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	9.2.9. Applied Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Bimanual object grasping using a robotic simulator

Aim: Implement and test a reinforcement learning algorithm that will allow the humanoid robot (iCub) to learn to grasp large objects bimanually.

Literature: Zdechovan, L. (2012). Modelovanie uchopovania objektov pomocou neurónových sietí v robotickom simulátore iCub. Master's thesis, Comenius University in Bratislava.
Pecháč, M. (2013). Samoorganizácia senzomotorických reprezentácií a ich využitie pri uchopovaní objektov. Master's thesis, Comenius University in Bratislava.

Comment: Take inspiration from previous theses on this topic defended at FMPI.

Keywords: cognitive robotics, grasping, reinforcement learning

Supervisor:	doc. Ing. Igor Farkaš, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. PhDr. Ján Rybár, PhD.
Assigned:	10.10.2013
Approved:	21.10.2013

doc. RNDr. Damas Gruska, PhD.
Guarantor of Study Programme

....
Student

....
Supervisor

53781498

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Andrej Sekáč
Študijný program:	aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	9.2.9. aplikovaná informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický

Názov:	Obojručné uchopovanie objektov v robotickom simulátore / *Bimanual object grasping using a robotic simulator*
Cieľ:	Implementovať a otestovať algoritmus na báze učenia posilňovaním, ktorý umožní humanoidnému robotovi (iCub) naučiť sa obojručne uchopovať veľké objekty.
Poznámka:	Inšpirujte sa predchádzajúcimi záverečnými prácami z danej oblasti obhájenými na FMFI.
Kľúčové slová:	kognitívna robotika, uchopovanie, učenie posilňovaním

Vedúci:	doc. Ing. Igor Farkaš, PhD.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	doc. PhDr. Ján Rybár, PhD.
Dátum zadania:	10.10.2013
Dátum schválenia:	21.10.2013

doc. RNDr. Damas Gruska, PhD.
garant študijného programu

[illegible]
....
študent

[illegible]
....
vedúci práce

I hereby declare that this thesis is my own work and that all sources I have used or quoted have been indicated and acknowledged as complete references.

. . . .

I would like to express my sincere gratitude to prof. Ing. Igor Farkaš, PhD., for his support and helpful guidance throughout the whole project. His optimistic approach helped to overcome every complication that occured. Also I would like to thank Loretta Schindler, my family, the Pink Floyd and all of my friends for hours of support and inspiration.

Abstract

Our thesis focuses on the learning process of bimanual grasping and its application on robots. We designed our model on a basis of cognitive robotics. We used neural networks and reinforcement learning methods suitable for continuous space. Then we connected our model with the iCub robot using a computer simulator. The solution is divided into two subcomponents. The reaching based on reinforcement learning and the grasping implemented by simple algorithm. This thesis describes our model in detail and explains achieved results.

Keywords: cognitive robotics, grasping, reinforcement learning

Abstrakt

Táto práca sa zameriava na proces učenia bimanuálneho uchopovania a jeho využitie v robotike. Navrhli sme náš model na báze kognitívnej robotiky, pričom sme použili neurónové siete a metódy učenia posilňovaním vhodné pre spojitý priestor. Navrhnutý model sme potom spojili s robotom iCub prostredníctvom počítačového simulátora. Riešenie je rozdelené na dva komponenty. Siahanie, založené na učení posilňovaním, a uchopovanie, implementované jednoduchým algoritmom. Táto práca detailne popisuje náš model a vysvetľuje dosiahnuté výsledky.

Kľúčové slová: kognitívna robotika, uchopovanie, učenie posilňovaním

Contents

List of Figures

Introduction

The development of technologies brings new possibilities every day. Every day humanoid robots are harder to recognize from humans and their movement abilities are more and more precise. However, usually they just follow preprogrammed commands so the challenge is to make them behave as humans do.

Cognitive robotics believes this can be achieved by modeling the learning process of humans. Our goal is to implement and test an algorithm that would allow a robot to manipulate large objects. We were inspired by cognitive robotics and its approaches and decided to solve our problem with a biologically plausible method.

One of the best humanoid robots for research purposes is the iCub. The iCub was developed by RobotCub Consortium and has 53 degrees of freedom. For the testing of our model we used the iCub simulator, a simulator modeling the iCub robot.

Real robots and also the iCub simulator operate in a continuous space and therefore we based our model on CACLA (Continuous Actor–Critic Learning Automaton). CACLA belongs to reinforcement learning methods which base the learning process on an interaction with the environment.

Chapter 1

Background

In chapter 1 we will discuss some fundamental knowledge, that we need to solve our task. Our goal is to implement and test reinforcement learning algorithms for bimanual grasping process. In the next pages we will focus on introducing and describing all necessary topics as human bimanual coordination skills and restrictions, artificial neural networks and reinforcement learning approaches.

1.1 Bimanual coordination

Bimanual coordination is a prototype of a complex motor skill which promotes the integration of components of the desired task into a meaningful result (Swinnen and Wenderoth, 2004). Use of both hands simultaneously has become important for human life and therefore complexity of human bimanual skills has increased. This allows human beings to learn to coordinate hands in simple tasks as pulling or pushing, where actions of both hands are similar, but also in more difficult tasks as playing musical instruments, where roles of each hand are strongly different.

1.1.1 Study frameworks

From the research emerged two major approaches that are widely used by scientists to study bimanual control. These are the information processing and the dynamic pattern perspective.

The information processing perspective considers the bimanual movement a specific case of a dual task performance that is limited by a structural interference caused by a lack of neural resources. This concept of the neural crosstalk focuses on studying performance limitations in a case when the assigned task is different for each hand. It assumes that completing bimanual tasks can be done by suppressing the neural interference achieved by training or merging tasks into a global control structure.

The second approach, dynamic pattern theory, studies biological systems as coherent global patterns composed of many subcomponents. The organization is a result of cooperation among subcomponents, and it is not maintained by any superior entity. The bimanual coordination can be similarly interpreted as a self-organized behaviour in systems with many degrees of freedom.

Despite their differences, both frameworks search for constraints affecting the motor coordination.

1.1.2 Limitations and constraints

The bimanual movement is limited by constraints. Some of them are constraints which can be observed during single limb movement, others are unique for the bimanual coordination. Contraints can be associated with a motor output. For example, tasks requiring a homologous muscle activation show more stable coordination than tasks using non-homologous muscle groups. Other type of constraints is linked with a sensory input. Moving a finger in synchrony with an auditory beat is much more stable than moving it in alternation with the beat. Similar differences occur with a visual input. It seems that the coordination of movement is formed on all levels of the neural hierarchy.

Experiments also show that humans prefer to use common time frames for moving all effectors. This allows them to easily keep a stable performance even on simple rythms such as (2:1, 3:1) where frequency of one limb can be divided by frequency of the other. More difficult rythms (5:3, 3:2) result in a worse performance and tend to lose an original pattern and to approximate to simpler rythms.

Directional constraints are also important. Limbs moved toghether in the same direction show a stable and accurate performance as opposed to movements in different directions. This behaviour suggests that motor constraints reinforce each other in order to increase the quality of movement.

1.2 Artificial Neural Networks

Artificial neural networks are computational models inspired by the human central neural system. Our brain is capable of completing certain computations much faster than digital computers. This is achieved by the parallel nature of brain as well as complexity and nonlinearity. All these are characteristics of brain as a information-processing system. The brain consists of neurons, small and simple computation units connected into networks using synaptic connections. The ability to perform computations emerges from modifying weights of interconnections between cells, a process called training. Networks are also capable of changing their topology in order to respond better to their desired task. The plasticity of connections enables the adaptation of network to changing environment (Haykin, 1999).

1.2.1 Main features

Artificial neural networks possess several unique, interesting and important features which make them widely used (Haykin, 1999).

Nonlinearity. A neural network can be nonlinear if is made of nonlinear neurons. This nonlinearity is distributed throughout the whole network which makes it suitable for solving some special problems.

Generalization. A neural network, thanks to its distributed nature, is capable of the generalization, returning reasonable output even for an input previously unseen.

Adaptivity. Neural networks are capable of adapting to changes in the surrounding environment. This means that they are able to retrain or modify synaptic weights according to small changes in conditions of the environment, whether it is similar new environment or the original nonstationary environment changed in time. The adaptivity of a network also poses a problem called the stability-plasticity dilemma. Small disturbances can destabilize the system if it adapts too much to minor differences or noises in input data. To create a truly robust system it is necessary to find the right balance between adapting to and ignoring spurious disturbances.

Contextual information. The structure and the state space of a network can contain a contextual knowledge possibly useful for the designer of network.

Fault tolerance. A neural network can be trained in a way which makes it more robust and fault tolerant. Thanks to the fact that the information is distributed throughout the whole network, errors and damage done on connections between neurons has to be extensive in order to degrade the performance of network seriously. Minor faults lead only to a graceful degradation and the retention of solid quality of the output.

Parallelism. This feature makes neural networks suitable for some tasks which require a massive computation power.

1.2.2 Main applications

Artificial neural networks have variety of uses thanks to its unique abilities. However, ANNs are unable to form a complex solution and in real use they are usually integrated into a working system and provide solutions for subproblems. A wise use of ANN can lead to a system with abilities which would be otherwise impossible. Most important tasks for which ANN are being used are:

Pattern classification and recognition. Neural networks are capable of learning to sort input patterns to desired classes or to recognize already known patterns.

Pattern association. A neural network can be taught to model some associations between input patterns and the output.

Function approximation. Thanks to nonlinearity, neural networks are capable of the approximation of nonlinear functions with high accuracy.

Feature extraction. A neural network is suitable for extracting features of the input, for example effectively reducing dimensionality of the input.

Filtering. A right learning algorithm can form a neural network into a highly sophisticated filter.

Auto-associative memory. Linear neural networks can be used as auto-associative memory. These are easy for the computation and they can reconstruct a noisy input with good accuracy.

Prediction. Recursive neural networks possess the ability to recognize, generate and also predict sequences.

1.3 Reinforcement Learning

Reinforcement learning is an approach to learning processes which focuses on interaction with the environment. The learning by interacting with the surrounding environment, without any teacher, is probably one of the most common in living organisms. Understanding the connections between actions and following results is crucial for humans as well as animals to live in this world and achieve any goals that arise (Sutton and Barto, 1998).

1.3.1 Description

Reinforcement learning is a goal oriented approach in which the learner tries to learn how to map situations to actions to achieve this goal. He does not know which actions to choose, but he has to explore possibilities to find the ones which offer the best rewards. Important is that actions may influence the reward of not only the current situation, but also the reward of the next situation and all that follows. Trial-and-error and delayed reward are considered to be ones of the most distinguishing features of the reinforcement learning.

The reinforcement learning is always defined by a learning problem that needs to be solved and not by learning algorithms. Unique for reinforcement learning is pursuing the right balance of exploration and exploitation. An agent has to explore possible actions to find ones that yield a high reward. But to obtain high amounts of reward it has to choose actions which offer good rewards. By always prefering exploration the agent may end up with a low amount of reward and of course the sticking to known actions may lead to ignoring actions much better. To achieve best results it is necessary to try exploring and favoring actions that appear to lead to good rewards.

1.3.2 Reinforcement Learning Problem

The reinforcement learning problem is a framework for tasks where learning from interaction to achieve goal is present. Any approach of solving such task is considered a reinforcement learning method. Entity which makes decisions and learn is called the agent. Everything around the agent is considered to be the environment. The agent interacts with the environment by performing actions and the environment responds to the agent with new situations and rewards. Reward is a signal whose value is determined by the environment according to the reward function. The goal, the agent is pursuing, is to maximize the total reward over time. By specifying all necessary attributes of an environment we define an instance of reinforcement learning problem.

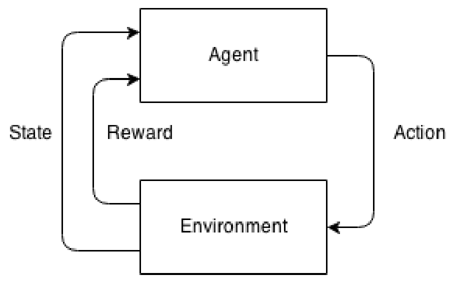

Figure 1.1: Reinforcement learning schema (Sutton and Barto, 1998).

Chapter 2

Methods

Chapter 2 introduces methods and algorithms which we use in order to solve our task. We will discuss the iCub simulator, as it is our environment, the neural coding of input data, necessary for a successful learning, and the CACLA algorithm.

2.1 iCub robot

iCub is a humanoid robot designed by RobotCub Consortium as a goal of RobotCub project (RobotCub, 2014a). The goal was to create a robot in a size of a small child for cognition research. The project was finished in 2010 after 5 years of work. The resulting iCub robot (Figure 2.1) is being constructed by Italian Institute of Technology in Genova (IIT). The final iCub is cca 1 m tall and weighs around 24kg. It has 53 degrees of freedom throughout the whole body including control of eye cameras. For our task is important that 7 degrees of freedom are in each arm and 9 in each hand.

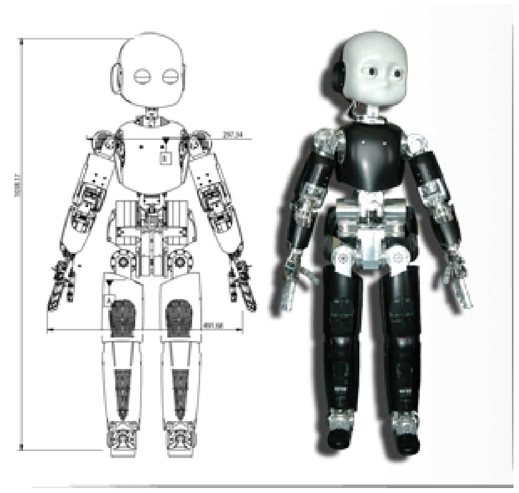

Figure 2.1: iCub body schema and real implementation (RobotCub, 2014a).

2.1.1 Simulator

Unfortunately most of the research teams cannot afford the iCub for their projects and the machine learning often requires thousands of repetitions for good results. This would be hardly possible to do on a real robot and therefore programmers from IIT created an iCub computer simulator for testing purposes (Figure 2.2). The iCub simulator is a open source project based on the ODE physics library and OpenGL (RobotCub, 2014a). It should be a faithful copy of a real robot and should behave the same way as the real robot would do.

2.1.2 YARP

YARP stands for Yet Another Robotic Platform, its designers realized that there are too many unique middleware solutions for different robot projects and development of such software is expensive and slows down research. Therefore they tried to come up with a solution that could be widely used

Figure 2.2: iCub computer simulation (RobotCub, 2014a).

and help everyone.

YARP is like a control neural system of robot (RobotCub, 2014b). It connects robot devices and sensors and allows sending data through these connections. It is written in C++ as a set of simple to use libraries and tools. It is capable of working with different protocols (e.g. TCP), can be used using many programming languages (e.g. Java, Python, C#, Ruby) and runs under all commonly used operating systems. All these features makes it sensible alternative for a robotic middleware.

2.2 Neural coding

For a better behaviour of an artificial neural network it is sometimes necessary to encode the input data, for example to increase its dimensionality. Similar information coding processes have been observed by neuroscientists in animal brains. There are several different approaches and encoding possibilities but here we will discuss only a few of them.

2.2.1 Population coding

Experiments show that the neural system of living beings encodes received information by a population of cells, rather than by a single cell (Pouget et al., 2000). It means that different values are expressed by different combinations of activations of neurons included in the encoding population. This implementation of information representation has its own beneficial properties. It makes the information much more robust, damage on one or just few cells leads only to a minor degradation of the quality of information. The population coding also offers the instantiation of nonlinear functions and mechanisms for noise removal.

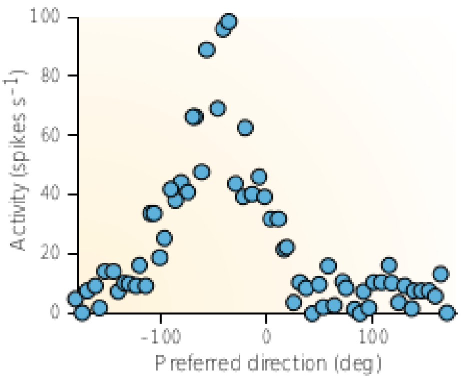

Figure 2.3: The standard population coding model. Activity of a population of neurons in monkey's brain in response to an object moving at -40° (Pouget et al., 2000).

2.2.2 Sparse coding

Sparse coding is an approach to the information encoding similar to the population coding. To encode an input data uses sparse coding also a set of neurons. The difference is, that the sparse coding requires the amount of active neurons for each value to be low. This gives it useful attributes like a high representational capacity, good generalization characteristics or a high fault tolerance with low interference rates (Földiák, 2002).

2.3 CACLA

We decided to use CACLA to solve our problem, in this section we will discuss its background and principles.

2.3.1 Actor–critic methods

Actor-critic methods are reinforcement learning methods using the temporal-difference error (TD error). These methods use two separate structures for the policy and the value function. The policy is represented by the actor and selects actions according to current state. The value function is the critic and it estimates quality of performed actions. This output is in form of a TD error and is used in the learning of both the actor and the critic (Sutton and Barto, 1998).

The critic is usually a function, which maps a value to a state. By comparing estimates of the starting state and the state into which chosen action had lead to, we get a TD error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.1}$$

where V is the value function (critic) and γ is the discount factor. Resulting δ_t is used to evalute the action a_t chosen in state s_t. Positive value of TD error suggests that the chosen action was good and improved our position, thus the probabilty to select a_t in the future should be increased. Negative δ_t indicates a poor action which should not be selected anymore.

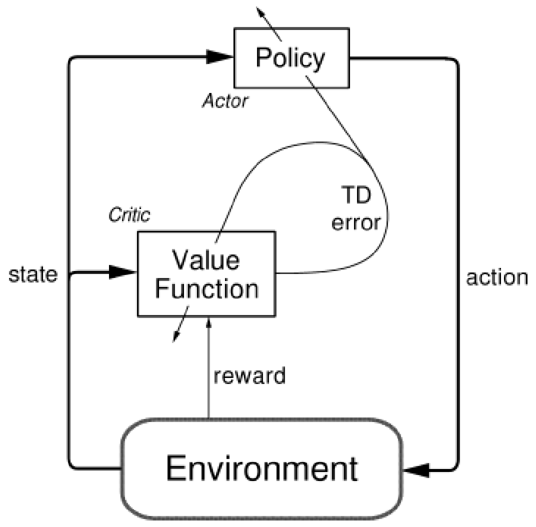

Figure 2.4: The architecture of Actor–Critic learning schema (Sutton and Barto, 1998).

Figure 2.4 demonstrates a typical architecture of actor-critic methods. Even though the architecture and the TD error value function is common for actor-critic methods, ways of selecting actions and processing the critic output may differ a lot.

2.3.2 Continuous Actor–Critic Learning Automaton

As the iCub simulator environment represent for reinforcement learning continuous state and action space, it was necessary to use an algorithm, which could handle this constraint. CACLA is capable of operating in such environment and thus is a suitable candidate for us.

CACLA is an actor-critic reinforcement learning method (van Hasselt, 2012). It differs from other actor-critic methods especially in the use of the TD error. The critic is a mapping $V : S \times \Theta \rightarrow \Re$, which tries to approximate V^{π}, where π is the current policy. The actor is a mapping $Ac : S \times \Psi \rightarrow A$, which always outputs a single action. CACLA requires an exploration during the learning to assure that $a_t \neq Ac(s_t, \psi_t)$. This is necessary because after performing a good action, we need to update the output of the actor towards this action. Without exploration actors output would be equal to the action and updating of the parameters would be impossible.

Many of other actor-critic methods update the actor according to the TD error everytime. When the error is negative the actor is updated in the opposite direction. CACLA updates the actor only when the TD error is positive as there is no knowledge about the opposite action. In addition CACLA pays only a little attention to the size of the TD error as small values could lead to slow learning and these values are less important than the distance to the action a_t.

Algorithm 1 CACLA learning algorithm

```
1: Initialize $\Theta_0$.
2: Initialize $\Psi_0$.
3: Initialize $s_0$.
4: for $t \in \{0,1,2,...\}$ do
5:     Select and explore $a_t \sim \pi(s_t, \Psi_t)$
6:     Perform $a_t$, receive $r_{t+1}$ and $s_{t+1}$
7:     $\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$
8:     $\Theta_{t+1} = \Theta_t + \alpha_t(s_t)\delta_t \nabla_\Theta V_t(s_t)$
9:     if $\delta_t > 0$ then
10:        $\Psi_{t+1} = \Psi_t + \beta_t(s_t)(a_t - Ac(s_t, \Psi_t))\nabla_\Psi Ac(S_t, \Psi_t)$
11:    end if
12:    if $s_{t+1}$ is terminal then
13:        Reinitialize $s_{t+1}$
14:    end if
15: end for
```

Algorithm 1 shows the learning flow of CACLA. The flow is also influenced by few parameters. γ is the discount factor, σ is the exploration factor, α is the learning rate of the critic and β is the learning rate of the actor. All of them may or may not change during the learning process.

At the first place is the initialization of necessary vectors and matrices. The algorithm then runs in a loop beginning with selecting an action according to the current policy π. As we have mentioned before the selected action must differ from action which would be generated by the actor alone. How much the chosen action differs is influenced by aforementioned σ. After performing the selected action, the critic receives the actual reward and new state from the environment. The critic then computes TD error from the estimated reward, the actual reward and also the predicted reward from new state s_{t+1}. How much predicted rewards influence the TD error depends on the discount factor γ. If $\gamma = 0$, the critic ignores any possible future development. In step 8 the critic is updated according to the TD error to improve the estimate. If the TD error is positive, thus indicating a good action, the actor is updated towards a_t. This update can be based on the distance between the original output and by the exploration modified resulting action $\|a_t - Ac(s_t, \Psi_t)\|$. The computation of an error in action space is another important difference from the majority of other actor-critic methods. If the new state is terminal, in positive or negative way, the state is reinitialized.

Chapter 3

Model

Chapter 3 describes our design for a solution of our problem. According to research, it seems that grasping objects using hands can be divided into two separate processes of reaching and grasping (Kawato and Oztop, 2009). With this in mind we designed our solution based on two movement modules.

3.1 Reaching module

The reaching module has an objective to move both hands close to the target object. To accomplish this objective the reaching module consists of two CACLA modules, one for each hand. Each of this hand CACLA modules is formed by a complete CACLA instance based on artificial neuron networks. The critic and the actor are implemented using multi-layer perceptrons using their function approximation ability (Haykin, 1999). Networks use the hyperbolic tangent as the activation function on all layers. The learning takes place for each hand separately and consists of episodes. These episodes end, when the hand reaches final position or when the maximum number of movements is reached. This number was set to 30. After each episode, the hand is reset to starting position. Target positions during learning are being selected from a set of final positions. The reaching module then computes from the position and dimensions of target object final positions of both hands and starts a reaching procedure. In the reaching procedure both hands move simultaneously towards their target positions.

3.1.1 Architecture of neural networks

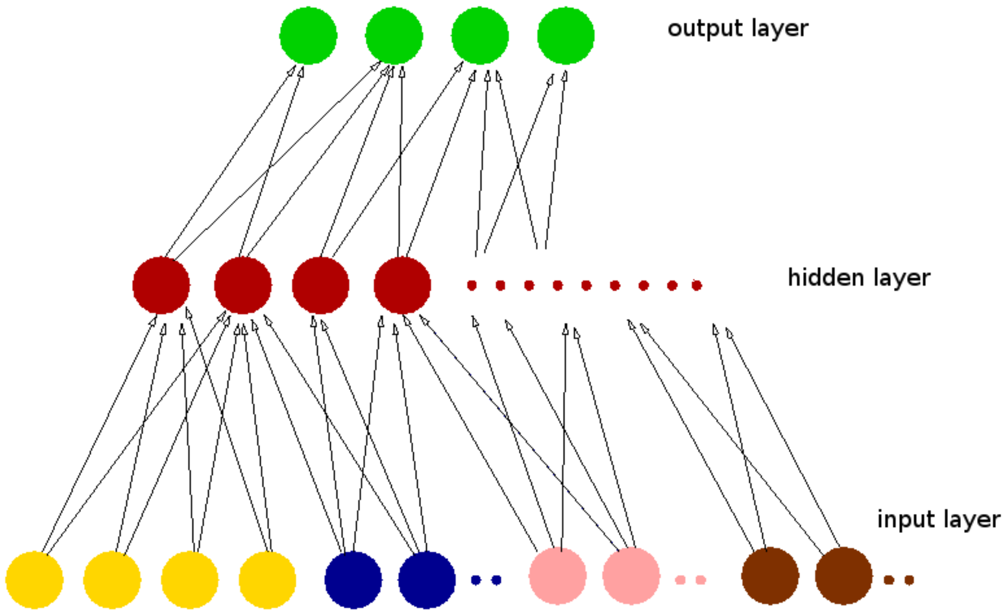

Figure 3.1: The architecture of multi-layer perceptrons implementing the actor.

During the designing of the networks architecture we took inspiration from master's thesis on which we base our solution (Zdechovan, 2012). The input layer consists of 31 neurons. 4 represent current degrees of freedom of hand. These values are rescaled into $\langle -1, 1 \rangle$ interval, because ranges of possible values are different for each degree of freedom. The other 27 neurons represent 3 coordinates of the target position in space. These coordinates are encoded using the population coding by 9 neurons each. The number of neurons on the hidden layer is a subject of experiments. Output layer has a different number of neurons for the actor and the critic. In case of the actor there are 4 neurons representing changes in degrees of freedom. The output layer of the critic has only one neuron, as the critics output is a single number.

The list of parameters is extended by a parameter called the exploration degradation factor, a number $\langle 0, 1\rangle$ by which is σ multiplied every episode. The values of parameters are also subject of experiments.

3.1.2 CACLA modification

After first testing a problem occured. Due to randomly generated weights of both the critic and the actor, learning processes took a long time. In some cases the learning took so long, that the exploration factor had decreased too much before it was able to find the right way. Combined with the slow iCub simulator we decided that we will use a modification of algorithm similar to the one used by Zdechovan (2012).

The first N episodes of learning we use so called reward learning rule for deciding whether to update the actor or not. Instead of using δ_t from equation 2.1, we decide by comparing the actual rewards. The algorithm 1 is then modified in step 9, where the condition is $r_{t+1} > r_t$. This means, that the actor ignores the critic, while he is not giving reasonable estimates. The rest of learning process uses the standard algorithm.

This modification showed good results even for small N values and saved us a lot of time.

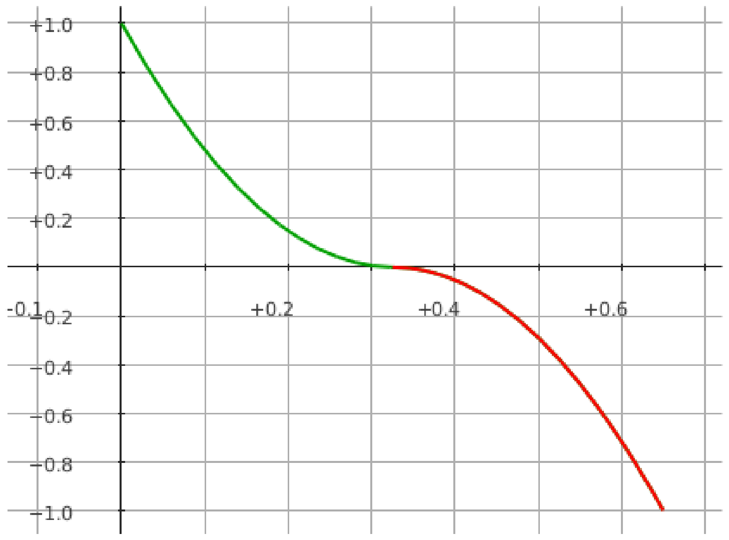

Figure 3.2: The reward function (Zdechovan, 2012).

3.1.3 Reward function

Reinforcement learning depends a lot on the quality of the reward function. A good reward function must reflect the nature of problem. We use the reward function also used by Zdechovan (2012). This function is non-linear and favors small distances from the target position with high rewards (Figure 3.2). The environment use this equation for the computation of reward:

$$r(d) = f_r(d)^2 \operatorname{sgn}(f_r(d)), \tag{3.1a}$$

$$f_r(d) = 1 - 2 * (d/d_{max}), \tag{3.1b}$$

where d is the distance of hand from the target position and $d_{max} = 0.65$ is the maximal distance.

3.2 Grasping module

Compared to the reaching module is the grasping module much more simple. The process of grasping and lifting the target object is implemented by an simple algorithm, which sends commands to both hands. The hands then push against each other and slowly move upwards thus lifting the object. For easier use is the grasping module connected to the reaching module and can request reaching process.

Chapter 4

Implementation

Chapter 4 discusses about implementation details of this thesis. The whole implementation was programmed using C++ language. This was useful, because both YARP and iCub simulator are also programmed in C++. For implementation we used already programmed neural networks libraries from Pecháč (2013) and took some inspiration for connections with YARP from Zdechovan (2012). This allowed us to focus more on experimenting with learning results. The final application is a simple program using the programmed libraries to start learning or grasping process.

4.1 Neural networks

The base of neural networks consists of adopted libraries, which implement low level parts. Here are programmed classes of neuron, three-layer neural networks and their updating algorithms, such as back-propagation algorithm.

4.2 iCub simulator

We have implemented a static class which takes control of communicating with the running simulator. It creates objects, deletes objects and returns positions of objects or iCub hands. All these functions require an open port to the iCub simulator world.

4.3 YARP connection

For the communication with the iCub simulator we programmed an adapter class of YARP. This class initializes ports to both hands and the world. Then it opens control interfaces and position encoders for both hands and sets the velocity for movement. These interfaces receive commands for moving hands and encoders tell the actual values of degrees of freedom. An instance of a YARP adapter is always created at the start of our application and then is used as parameter for the creation of all other components, such as the reaching module.

4.4 Reaching module

Implementation of class operating CACLA algorithm was also inspired by the existing solution from Pecháč (2013), but several modifications were required. The whole reaching module contains the operating class of reaching module and one pair of a set of classes implementing the CACLA algorithm. This set includes the environment class, the actor-critic encapsulation class and the CACLA algorithm class. The CACLA algorithm class is connected to an actor-critic instance for easy manipulation with the neural networks and also communicates with its own environment instance that is through YARP connected to the simulator.

4.4.1 Exploration

There are several ways how to explore actions. We decided to use the Marsaglia polar method to generate standard normal random variables (Marsaglia and Bray, 1964). We generate a variable for each value of an action and then we multiply these variables by the exploration factor σ. Resulting values we sum with the action.

Chapter 5

Experiment

This chapter describes the results of learning processes and compares differences in values of parameters.

5.1 Course of learning

As already mentioned, the learning process consists of 30 step episodes. During learning we collected the accumulated reward of each episode. This accumulated reward is a sum of all rewards acquired for each step of episode. Our reward function outputs values in interval $[-1.1]$ and therefore values of accumulated reward can possibly vary in the interval $[-30.30]$. It is clear that the accumulated reward cannot actually reach the value -30 nor 30, because the initial state is never the final state and some steps, with reward lower than 1, must be taken. As the initial state differs for each episode, it is not possible to determine an optimal value for the accumulated reward, but our experiments show, that accumulated reward higher than 20 is satisfactory for our reaching purposes.

At first we tried to generate final positions from a block in space, but these experiments led only to chaotic results and small accumulated rewards.

Therefore we created a set of five representative final positions and trained models using this set. We will focus on results of only right hand, because both models fulfill the same task and their learning progresses were equivalent.

Although we tried to manipulate all parameters, only some of them influenced the data significantly. These were mainly the exploration factor, exploration degradation factor and number of episodes using the reward rule. For other parameters we found stable values, which were mostly used. In all presented simulations we used these values: $\alpha = 0.1, \beta = 0.1, \gamma = 0.1$ and the number of neurons on hidden layer was 20 for both networks.

5.2 Reward rule

We use a slightly modified CACLA algorithm for our task. The first N episodes use the real reward for updating the actor. The following figures demonstrate different behaviour of learning according to different values of N.

In Figure 5.1 we can see that thanks to the reward rule modification we acquire high rewards very quickly. It is clear that after a change to the standard algorithm, performance of learning gets worse, but it is still capable of reaching satisfying rewards. We started with quite high values of N, but experiments showed, that it is not necessary. Figure 5.2 shows that the performance of learning is good even when $N = 75$. In both cases we used the exploration factor 0.25 and the exploration degradation 0.985.

We tried to find the best value, which would help us save time but would not influence the learning process too much. In the end N stabilized at 50 episodes.

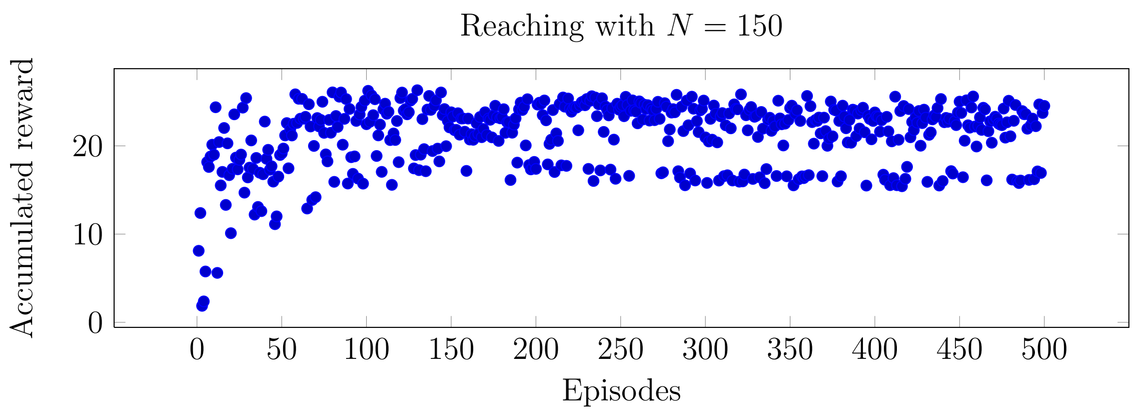

Figure 5.1: Accumulated rewards for $N = 150$.

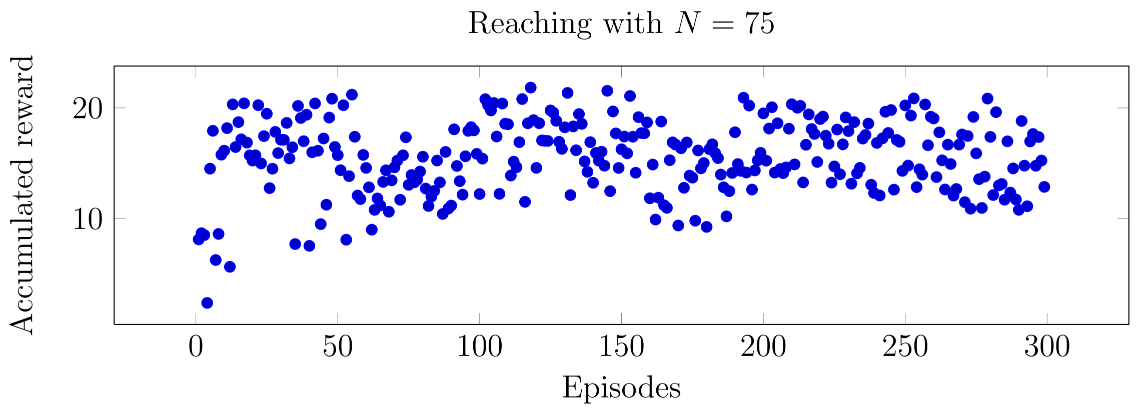

Figure 5.2: Accumulated rewards for $N = 75$.

5.3 Exploration factor

Even though at first we experimented with all the parameters, changes in the exploration factor were the most interesting. We struggled to find the best combination of the exploration factor and its degradation factor. Even the slightest modifications resulted in major differences in the performance.

We began with lower values of the exploration factor where $\sigma = 0.1$. This setup however did not end with an interesting learning outcome. Several learning simulations showed that a good value for σ lies around 0.25. Our simulations also revealed that the exploration degradation factor is equally important to σ. A value too small resulted in a poor learning progress, because σ decreased before the network reached a satisfying state. And a value too high lead to a destabilization in the ending phase of learning.

The following figures describe differences between individual setups. In Figure 5.3 we can see that after the 50th episode, where the algorithm returns to updating the actor according to the critic, the performance drops down, but then again starts to rise. Unfortunately the exploration factor then decreases and no further improvement occurs in the rest of learning.

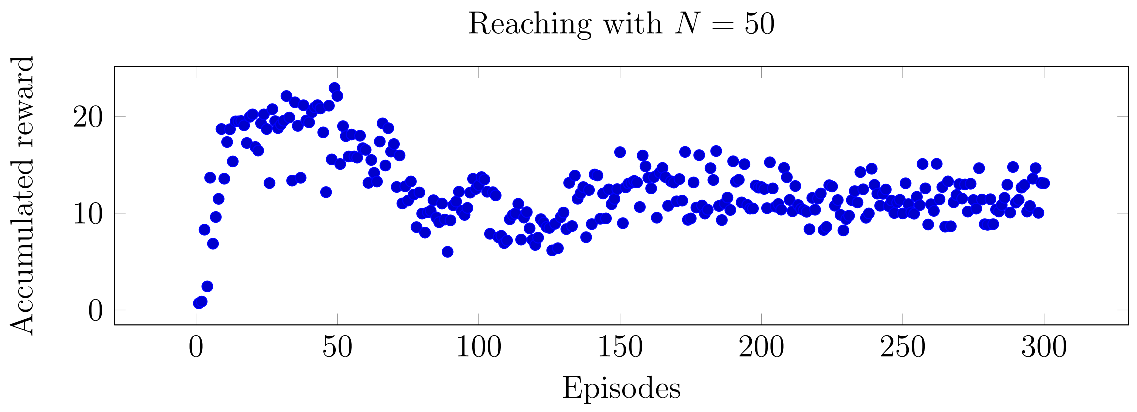

Figure 5.3: Accumulated rewards for $\sigma = 0.25$ and the exploration degradation equal to 0.985.

Figure 5.4 shows us how is the learning process influenced when $\sigma = 0.3$ and does not change. After the N-th episode we see a strong fluctuation caused by the higher value of σ. The performance does not improve anymore because σ is constant.

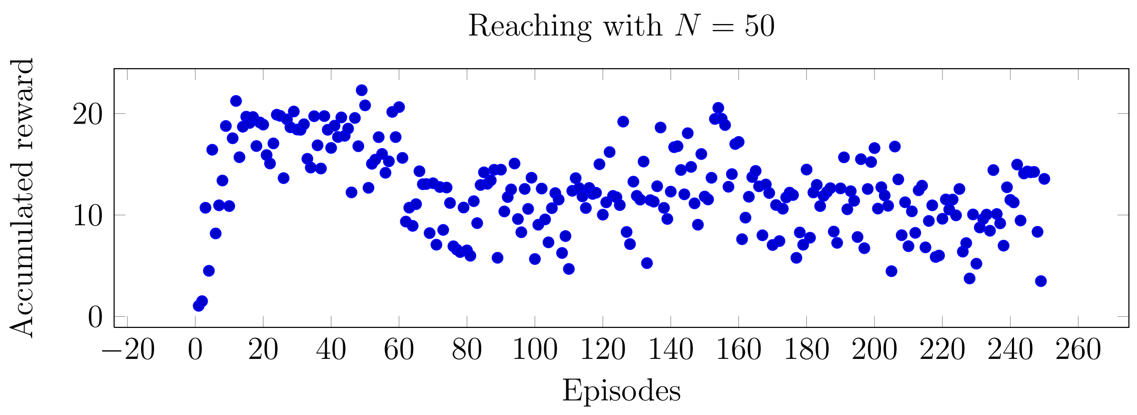

Figure 5.4: Accumulated rewards for $\sigma = 0.3$ and the exploration degradation equal to 1.0.

Figure 5.5 reveals that with the right combination of values we can achieve outstanding results.

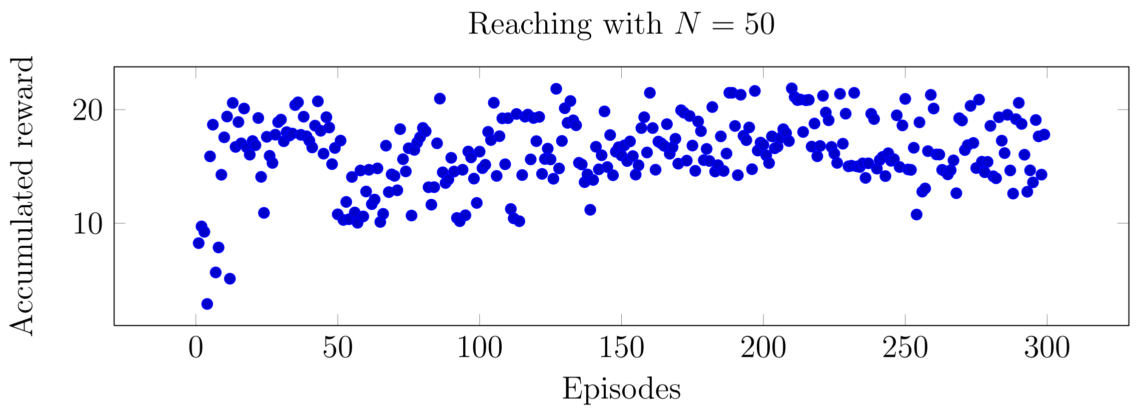

Figure 5.5: Accumulated rewards for $\sigma = 0.25$ and the exploration degradation equal to 0.99.

Position	Average distance
1	6.11
2	3.23
3	5.16
4	2.80
5	3.05

Table 5.1: Average distances to the target position approximated in cm.

5.4 Testing

We tested the learning by starting a reaching procedure and collecting the final distance to target position. Each row in Table 5.1 shows an average error of 10 attempts to reach one final position. The average error of the whole set is smaller than 5 cm. Even though this value may be too high for some problems, for our task it is sufficient.

We tested the grasping procedure only visually. The following figures show that a correctly learned reaching module allows the grasping module to successfully lift objects.

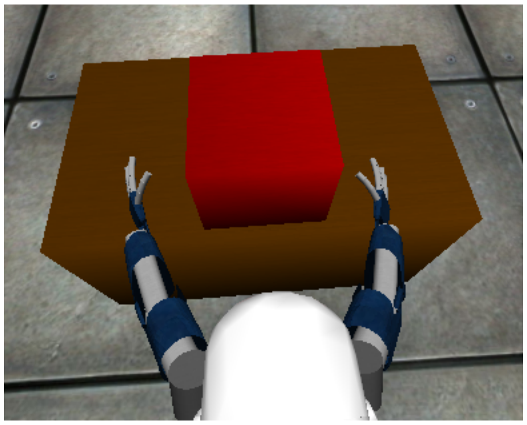

Figure 5.6: The iCub reaching an object.

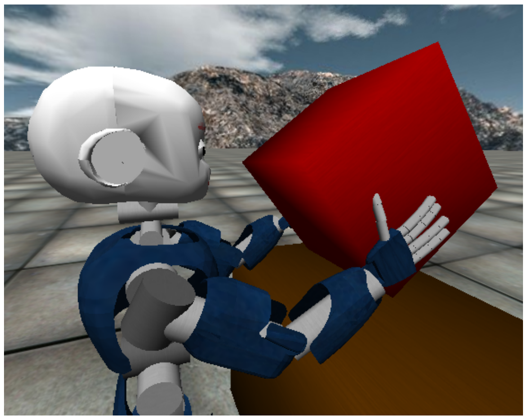

Figure 5.7: The iCub grasping and lifting an object.

Conclusion

Our goal was to implement and to test a reinforcement learning algorithm for bimanual grasping of objects in a robotic simulator. We designed and implemented a model for the object grasping task using the CACLA. We connected this model with the iCub simulator environment and tested its performance and possibilities. Our solution was based on solutions from two master's theses, Zdechovan (2012) and Pecháč (2013).

Our model extended the standard CACLA algorithm by a minor modification. We tested our model with different combinations of parameters. Then we compared recorded results and included the interesting ones in our thesis. The testing of successfully finished learning courses confirmed our expectations and validated our model. The simulated robot was able to learn to reach for target objects and lift them using our grasping module. The generalization ability of the trained model was not excellent, but it was satisfactory for our task.

During the testing of our model we encountered a serious problem with the iCub simulator. Sometimes it got stuck in the middle of the learning process and this often resulted in the data corruption. We did not manage to solve this problem and it slowed our progress.

We believe that in the future our solution could be extended by modeling bimanual constraints during the learning process or by learning the robot more delicate manipulating with grasped objects.

Our model proved to be able to accomplish our goal when used with right values of parameters. Therefore we believe we made a successful step in the development of neural models for the iCub simulator.

Bibliography

Földiák, P. (2002). Sparse coding in the primate cortex. In *The Handbook of Brain Theory and Neural Networks*. MIT Press.

Haykin, S. (1999). *Neural Networks*. Prentice Hall.

Kawato, M. and Oztop, E. (2009). Models for the control of grasping. In *Sensorimotor Control of Grasping*, pages 110–124. Cambridge University Press.

Marsaglia, G. and Bray, T. A. (1964). A convenient method for generating normal variables. *Siam Review*, 6(3):260–264.

Pecháč, M. (2013). Samoorganizácia senzomotorických reprezentácií a ich využitie pri uchopovaní objektov. Master's thesis, Comenius University in Bratislava.

Pouget, A., Dayan, P., and Zemel, R. (2000). Information processing with population codes. *Nature Reviews, Neuroscience*, 1:125–132.

RobotCub (2014a). *iCub manual – wiki for robotcub and friends*. http://eris.liralab.it/wiki/Manual, http://www.icub.org/.

RobotCub (2014b). *Official documentation*. http://eris.liralab.it/yarp/.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning An Introduction*. MIT Press.

Swinnen, S. P. and Wenderoth, N. (2004). Two hands, one brain: cognitive neuroscience of bimanual skill. *Trends in Cognitive Sciences*, (8):18–25.

van Hasselt, H. (2012). Reinforcement learning in continuous state and action spaces. In *Reinforcement Learning: State of the Art*, volume 12, pages 207–251. Springer Berlin Heidelberg.

Zdechovan, L. (2012). Modelovanie uchopovania objektov pomocou neurónových sietí v robotickom simulátore icub. Master's thesis, Comenius University in Bratislava.

A CD supplement

The attached CD contains all source codes and also an electronic version of this document.