Migrating robot control systems, towards the universality of robotic brains

Paul Neculoiu
Department of Artificial Intelligence
University of Groningen

August 30, 2012

Supervisors:
Dr. Marco Wiering (Artificial Intelligence, University of Groningen)
Dr. Tijn van der Zant (Artificial Intelligence, University of Groningen)

university of groningen / faculty of mathematics and natural sciences / artificial intelligence

Contents

Acknowledgements

Foremost I would like to express my sincere gratitude to my advisors Dr. Marco Wiering and Dr. Tijn van der Zant for the support in my study and research, for their help and their patience in guiding me throughout this thesis and for the countless talks we've had on science and philosophy that have contributed to me coming out smarter out of this study than when I first started.

Secondly, I owe my loving thanks to Ning for having been there for me throughout this turmoil, as well as my parents who birthed and raised me and supported me in my endeavours.

Finally I would like to thank God for being just a fairy tale. Had this not been my realization at a young age I would likely not have questioned reality as I had known it, walked the path of science and wanted to play god myself with robots.

Thank you everyone for making this thesis possible!

A new age of robotics may soon be upon us

Abstract

Currently robot control systems are specifically designed, engineered and fine tuned for particular problems on particular robots. This leads to a significant waste of man-hours of engineer and Phd level work to implement and reimplement or adapt controllers for similar tasks on different robots resulting in an inefficient robotics industry as a whole. Thus the need to automate or at least semi-automate controller reusability arises. In this project we investigate the hurdles that need to be overcome in attaining controller universality and look into possible methods to bootstrap controllers to the different robot sensors and actuators. A case study was conducted on performing the migration of a controller from a wheeled robot with no mobile vision system (The Pioneer robot) to a legged robot with a mobile head mounted camera (An Aldebaran Nao). The two robots' different modalities makes the task challenging. What does it mean for two different robots to perform the same task? Machine learning methods were deployed using artificial neural networks (ANN) to learn the entire sensor abstraction - decision system - robot motor API tree, leaving just sensor feature extraction and low level motor controls in the hands of engineers. The method works reasonably well, effectively linking a number of controllers designed for a Pioneer onto the Nao's sensors and actuators. While preliminary, these methods provide insight into the future prospects of robots programming themselves and learning from each other with the help of humans.

Chapter 1

Introduction

The field of artificial intelligence, as applied to robotics [1], today faces two grave problems: scalability and portability of control systems. Often systems work well on small problems but fail to scale up to real world applications, either due to the combinatorial explosion [2, 3, 4, 5, 6], making the computational requirements for larger problems not achievable in practice, or the curse of dimensionality [7, 8, 9, 10], which makes solutions that work well for toy problems be severely stuck in local minima when attempting to scale them up. Solutions for complex problems, when found, however are usually problem specific. This is true not just for robotics but for the whole field of AI. Teams across the world consisting of Phds, Post-docs, other graduate and undergraduate students, programmers and engineers, spend numerous man-hours building and rebuilding robot control structures virtually from scratch every time these structures need to be implemented on a different robot. This leads to an overall unproductive robotics industry as a whole, making a surge in robot development difficult and unlikely.

It would be desired to save on all the specialized man-hours of work in favour of automatic or human assisted semi-automatic systems of transplanting controllers. The differences between different robots trying to perform similar tasks are not big problems logistics-wise, but since solutions are so problem specific, performance hinges on even the slightest detail of the implementation. This makes transferring solutions to different contexts complicated.

At its basis a robot control system for an autonomous robot can be roughly described as follows:

Sensory inputs (pixel values from cameras, joint angles from robot limbs, sonars and other sensors) are processed via predetermined computer vision, sensor extraction and pattern recognition techniques and converted into a more knowledge rich format (SIFT [11] features, object detection, location and size, etc). This processed data is then used to drive a high level robot behaviour that decides how the system should behave as a whole (move towards, away, turn left, right, grab, speak, change memory state, etc). The result of this behaviour is then passed on to the low level controller that turns the decision into motor and actuator commands (turn left wheel, move head, move arm, do nothing, etc). This is later on picked up by the robot hardware API to physically control actuators. This can be expressed in a graph as shown in Figure 1.1.

Variations may exist on the schema (in particular systems some sub-ensembles may differ, be merged

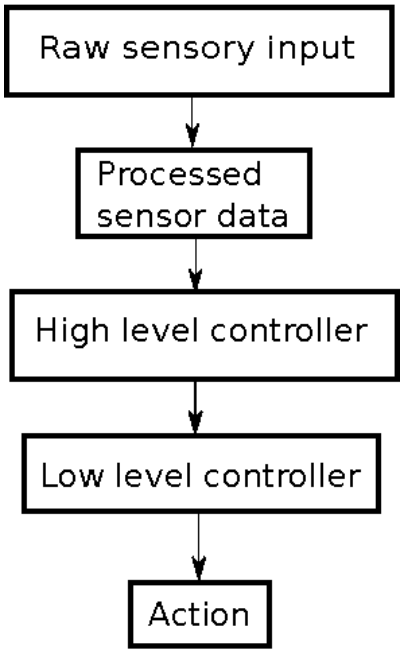


Figure 1.1: Basic robot control system schema.

or missing altogether) but most robotic systems follow a similar implementation. What we aim to transfer between robots is the high level controller. This is where the bulk of AI research goes to, the rest being predetermined methods that may be very hardware dependent. One cannot perform visual object detection on a robot that does not have a camera, although the same object can be detected using, for example, a depth sensor. Should this sensor abstraction be possible, at the controller level, there is no difference between sensors. Conversely, from a controller's point of view, actuators are equivalent if they can perform the same action. Namely it doesn't matter whether the robot has legs or wheels as long as the effects of the locomotion process are considered equivalent. What exactly defines the equivalence in either sensor or actuator space is to be decided by the evaluator of the respective task.

This gives the input-action pipeline a layered structure, with each layer representing a certain abstraction of data processing. The current thesis investigates at which layers of abstraction does the robot specificity stop and the generality of the high level controller, the decision making unit, begin. It also investigates machine learning methods to bootstrap transplanted controllers and reattach them to the abstraction layers of the new receiving robot. Such a methodological baseline is necessary in order to establish a starting point for further research into control system portability.

1.1 Related work

Abstraction layers are not a new concept, computer systems are usually represented as consisting of five abstraction levels: hardware, firmware, assembler, operating system and processes [12]. These abstraction layers encode the processes taking place at the abstraction layers above, and are usually

not transparent to those higher layers and at their core, the different layers perform very different functions.

The TCP/IP stack [13, 14] also implements abstraction layers, namely the link layer, internet layer, transport layer and application layer. Each of these layers contribute to the overall performing of the data transfer function in networks.

Abstraction layers do not reduce only to hardware interpretation, the Python interpreter [15] and the Java Virtual Machine [16] literally function as software abstraction layers, interpreting the program code themselves and translating the instructions into run-able byte code. This allows nearly any type of program written in those respective languages to be run independently of the operating system, and to some extent, of hardware architecture.

We desire a similar feature, a series of robot abstraction layers to allow for replacement and interoperability of components from within the robot processing structure. However, no such structure is currently in existence, although there have been some attempts at creating universally applicable robotics methods.

For example, in [17] the authors present a framework that facilitates integration of action, detections and interactions with human subjects. The aim is to facilitate adaptive human robot interaction in open-ended tasks. This establishes an "EgoSphere", an internal modelling framework of the robotic framework to perform robust object and action learning and recognition on distinct robotic platforms with the goal of importing the respective feature from one robot to another.

The authors of [18] criticize the low degree of code reusability in robotic systems and the focus on manipulator-level programming and not task level programming. They then propose a robot independent programming environment (RIPE). The environment attempts to separate the task-level, supervisory control, real-time control and device drivers into separate entities and program them in an independent manner. The system relies heavily, however on direct interoperability of the various subsystems.

The greatest importance seems to be placed on the advancement of level programming in [19]. Here the authors define a set of primary actions or primitives to simplify robot programming. They propose a solution that involves decomposing a goal into subtasks so that it can be resolved using those respective primitives. The system, however, relies on faultless implementation of those primitives and does not facilitate adaptability.

Later they propose a task matrix framework for programming humanoid robots [20] in a platform-independent manner. However, they still do not facilitate learning and adaptability. Their components, do, however contain no robot specific code and are truly robot-independent. However, experiments were only run in simulation, so the different sensor experiences of the two robots were not taken into account. Correct sensor abstraction and action performance were taken as a given. In addition, the two robots were relatively similar humanoids.

Robot independence has been achieved to some extent in mostly logistics tasks such as navigation [21]. In this scenario, the authors used directed graphs to provide a landmark-based representation of maps. These landmarks are then used to provide language-based directions to enable a robot to steer between landmarks. These use a motion description language and not actual motor commands, thus making this respective subsystem robot independent.

Most of these approaches, however, perform significant degrees of modelling, thus making a wide range of assumptions regarding the world. Robots that operate in real-unconstrained environments, are not closed systems however. They are part of their environment. Robots interact with their environment and observe the changes that occur. Research including these issues exist but is relatively limited. In [22], a framework for online learning of outdoor robot control systems is presented. The authors make use of genetic programming to learn controllers for robot navigation, while ensuring the parameters of their training methods were robot independent, so that the algorithm can be moved between different robots with minimal changes. While not an actual controller in itself, it is a learning method for controllers that can be transported and generate the required controller. This can be very versatile.

1.2 Research question

This raises the following question: **How can robot controllers be scaled, adapted and transferred between different robots?**

The problem is relatively trivial when confronted with identical machines, all one needs to do is copy the entire controller to the new robot. However, once the machines begin to differ in sensors and actuators the question takes on a whole new scope. Different sensors lead to different sensor abstractions, if such abstractions even exist for both robot modalities in the first place. Some abstractions may not even be directly interpretable by the controller. Robots can differ in such a way that arbitrary controller transfer may not be directly possible. A number of possible causes may be:

1. Sensor abstractions have different scales and domains of definition.

   For instance both robots can have cameras mounted, but if the cameras are not identical, then recognized objects will not be in the same location in the image for both robots. Or objects recognized on one camera might not always be recognized on another because colors are slightly different or one camera has more noise or a different focal depth for example. One robot could detect an obstacle, for example, while a different robot in a similar position but different sensory capabilities cannot. See Figure 1.2 for such an example.

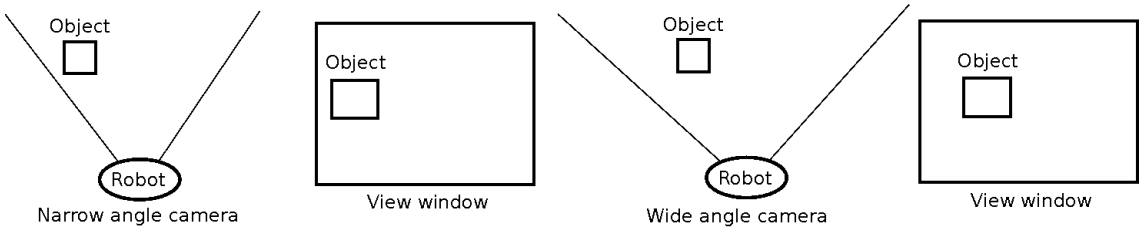


Figure 1.2: A robot with a narrow angle camera (left) sees objects further away from the center of the image than a robot with a wide angle camera (right).

2. Having the same sensor abstractions is not even possible.

   Suppose one robot abstracts data, like recognizing objects from one sensor modality such as cameras, while the other robot does not have a camera and thus can not recognize that object

in the same manner. In such a case, either alternatives need to be sought out or the attempt to migrate the controller should be abandoned since it's not technically feasible.

3. Differences between robot motors and actuators exist.

   These lead to complications. High level commands such as "move forward", "turn right", etc are valid for all mobile robots and "grab object" is valid for all robots with grippers, but actual implementation differs. A robot with larger wheels can turn just as well as a robot with smaller wheels, it just needs to spin its wheels slower. In this case merely the scale of the actuator function differs and the robot behaves similarly both on a large scale (turning) as well as on a low scale (turning smoothly). On the other hand a legged robot may move and turn by using its legs. Legs are a more complicated piece of machinery than wheels. A two wheeled robot can be commanded with just two control vectors (setting the speed of turning for each wheel) while a biped robot for example is more complex in both degrees of freedom as well as dynamics. A robot with 2 legs and 6 degrees of freedom per leg leads to a total of 12 values to be controlled. Preprogrammed mappings exist at times simplifying this phenomenon, reducing for example the amount of variables needed to control the robot to three denoting walking straight, strafing and turning. Out of which, strafing is not a behaviour that wheeled robots can perform as well. Where is the line drawn? How do you judge what is an equivalent action and what is not? The effect of walking is also different. While turning with legs on a larger scale is similar as on a wheeled robot, the end effect of pointing in a different direction being the same, on a smaller scale it's very different. While wheeled robots keep turning in a smooth manner, a legged robot will jog and jolt all of the sensors while in action creating noise and possibly influence its own behaviour.

4. There are differences in processing power on the robots as well as differences in software APIs.

   These differences can affect the way the robots perceive not only their sensory inputs but how these sensory inputs evolve and thus how time passes and motion takes place. Do the robots function in continuous time? Or can they only physically take input and give output at predefined intervals? Can both robots do more than one action at the same time? All of these are issues that need to be taken into account when attempting to transfer controllers.

A number of rules and procedures need to be set up and the problem needs to be divided into semi-independent subproblems. While having independent subproblems would be ideal to solving such tasks, due to the reasons stated above this is not exactly feasible. Sensors are influenced by different actuators, while behaviours control actuators by using input from sensors. While some distinction can be made, these ensembles can not be separated fully. Thus a secondary research question rises: **How to define the different abstraction layers to fulfil the previous research question?** It is clear that some layers of abstraction need to be defined, some similarities between robots found and some differences acknowledged. What does it mean to perform a similar action? What does it mean to sense the same thing? Does the behaviour need to change on a new robot to obtain the same effect?

And in the end we reach the last issue we want to tackle. **How to perform robot to robot controller migration with the least amount of modelling?** Thus far, most research attempts to create an inner environment for the robots to abstract observations and actions to. However, this still requires qualified man-hours of work for every robot that one wants to bring into the respective

framework. Modelling is rigid, modelling is most of the time non adaptive, modelling is hard to scale and transfer. Modelling makes assumptions. In this work, we want to perform controller migration without the use of heavy world and context modelling.

1.3 Motivation for research

Why is this type of research needed? There is only so much we can do to teach the machines in the long run. If we are to have truly intelligent machines they need to learn and teach themselves, and carry on the knowledge they gain to different machines and different bodies. They need to learn together and share the knowledge, without having to learn it again from scratch once they get a physical upgrade. Success of this approach could prove vital for robot independent AI and for the long term development of intelligent machines.

While attempts at controller universality do exist, something of this scale and scope has not been previously attempted. This approach is new and, as a consequence, it lies at the forefront of universal robot controller research. While of an exploratory, incipient nature, the results of what this thesis presents, can set the basis for what could one day redefine machine intelligence and reshape the world of robotics for ever. This alone is more than enough motivation.

1.4 Outline of thesis

In the next chapter, chapter 2, the methodology and the architecture proposed to solve the research question is presented along with a presentation of the various machine learning methods used throughout the thesis. In chapter 3 we investigate possible methods to learn a subsection of the proposed architecture, namely the controller part. This is followed by chapter 4, presenting a method to learn a different component of the proposed architecture, namely finding equivalences between inputs. Learning equivalent outputs is presented in chapter 5, while chapter 6 discusses the performance of the approach and the overall applicability of the architecture proposed.

Chapter 2

Theoretical framework and methods

A robot control system for an autonomous robot is the decision making unit responsible for processing input and memory states and making decisions about actions and new memory states. The aim in this thesis is to transfer controllers from one robot to a different robot **of a different modality**. This puts us face to face with the issues presented in section 1.2. Around the controller lies an entire framework that connects it to the various parts of the robot, bringing information from sensors to it and taking information from it to actuators. In the typical robot, the information flows as follows:

1. The robot acquires raw data from the sensors (pixel values from cameras, joint angles from limbs, sonar values and other sensors).
2. The data gets processed into salient, intelligible information for the controller (detected obstacle, recognized object, distance to an item).
3. The controller uses that information to make high level decisions (moving forward, grabbing an object, avoiding an obstacle, talking to a person).
4. The values of those decisions are transformed into low level control vectors (movement translates into wheel rotation, joint values, parameters for speech synthesis, etc).
5. The control vectors are fed into the robot's API which applies them to motors and actuators to generate the specified action.

Of these, steps 1 and 5 are robot specific, depending on the hardware/API/middleware of the robotic platform. What the robot can or cannot see or do is unavoidably limited by its physical and middleware capabilities.

Step 2 generally depends on the problem being tackled. Knowledge is extracted from the data sources using pattern recognition techniques [23] to detect objects, people, faces, recognize speech, etc. Anything that can turn raw data into salient information. Some techniques are already standard, while others are the focus of ongoing research. Depending on the method of deployment, some can be scaled up and used on multiple sensors and multiple types of sensors while others are fine tuned to be sensor and even vendor specific. Some detection systems may work on a wide variety of cameras, for example, while others may be so fine tuned they only work on one type of camera and any fluctuation in parameters might make it break down. Since this is not the focus of the current thesis, details of

knowledge extraction from sensor data will not be covered here. This step is partly robot specific, depending on circumstance.

Step 3, the controller designed to generate a standardized policy is not robot specific, or at least it's not supposed to be so since this is the item of interest that we desire to be transferred between robots. The final, outward manifestation of the controller's outputs is what we aim to make universal.

The inputs to the controller are pretty much standardized on one robot from what step 2 outputs. However, running step 2 on a different robot may yield different values (Perhaps from having different sensors or running different information extractors on the raw sensor data). However, if those values still contain the information required for step 3 but in a different format, then this can be remapped into the proper encoding. For example if a detector returns the corners of a detection, this contains roughly the same information as when a detector returns the detection's center, width and height. One can be mapped to another with relative accuracy.

Step 4 is a more complicated issue of control. Considerable fine tuning is required to attain the desired effect. The robots need to be both physically as well as API-wise capable of performing the action intended by the controller. If that is indeed possible, a feat best estimated by a human controller then there remains only the issue of determining what action, or what combination of actions, on one robot is the equivalent of what combination of actions on the other robot.

2.1 Hardware/Software setup

For this project two distinct robotic platforms are used: an ActiveMedia Pioneer 2 robot and an Aldebaran Nao seen in figure 2.1. The Nao robot is a bipedal humanoid robot 58 centimetres tall and weighing 4.3 kilograms. It is equipped with two cameras mounted in the head, two ultrasonic sonar sensors mounted in the chest, a microphone and various touch sensitive sensors. It is usually used in the Robocup [24] soccer standard platform league competitions, as well as in various bipedal research activities and human-robot interaction. The Pioneer robot is used for various research activities regarding mobile robots and it is usually modified to carry various payloads. The particular Pioneer robot used in this thesis is the one deployed by the University of Groningen's RoboCup@Home team [25]. For this task it has been equipped with an immobile Logitech webcam at height of around 70 centimetres and was stripped down of other sensory functionality.

Both robots are controlled through a software architecture developed in Python [26] that can acquire sensor information from different sources, implement a behaviour based [27] control structure and control multiple robotic platforms (simultaneously if required). The architecture was successfully deployed in the RoboCup@Home competition on behalf of the BORG team [25].

2.2 Issues of development

The component we wish to transfer is the high level controller, the decision maker. This is the heart of the system. Since a controller was originally constructed to work with a particular robot, the input abstractions and decisions it requires have been fine tuned specifically for that robot by its

(a) Nao robot. Picture taken from http://www.sais.se

(b) Pioneer 2 robot. Picture taken from http://cs.brynmawr.edu

Figure 2.1: Nao and Pioneer robots.

designers. Since a new robot may have different inputs and therefore different input abstractions, these abstractions would not be compatible with what the controller requires. Controller outputs would also be transformed later on into API commands for the robot. After the controller has migrated, though, the new robot would require new API commands. Thus, controller outputs also need to be adapted for this task. We therefore come to the conclusion that the problem needs to be split up into a number of distinct sub problems.

2.2.1 Data acquisition

Proper sensor data acquisition is of paramount importance. While information can take many shapes and sizes, it is vital that the information the controller requires is indeed there. Thus, making sure that the robot can extract the required information either in the form of carefully constructed detectors or large information extractors, with sufficient data to find the required correlations.

For this case, simple color blob detectors were constructed and deployed for object detection. The video stream from either robot's camera is captured, processed, evaluated and converted into input vectors for the controllers. Since the focus of this research is not object detection, simple red and green blob detectors were constructed and deployed. Due to differences in pixel values in the cameras, the detectors are slightly different for the different robots in order to facilitate similar detections.

A schematic of the sensor acquisition and processing can be seen in figure 2.2.

The detector outputs the relative position of the object on the screen, with (0,0) being the center, (-1.0,0) being maximum left side, (1.0,0) maximum right side, (0,-1.0) maximum top side and (0,1.0) maximum bottom side. The size varies with the percentage of the total image that the object takes up (from 0.0 to 1.0). A variable denoting whether the object is visible (1) or not (0) was also added to the input vector.

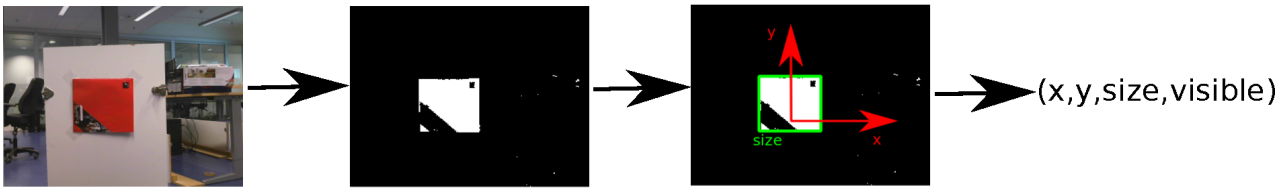


Figure 2.2: The sensory data acquisition pipeline. Raw images are used to detect the target (red notebook) and its properties (relative position on the screen and size) extracted to be used as input vector.

2.2.2 Input-input mapping

While the outputs of the detectors may vary and cause complications for transferring the control systems, this does not mean that the task is utterly impossible. As long as the information is there or is obtainable, it should not matter what form it takes. It is irrelevant if the robot with the larger field of view sees its target while the one with the smaller field of view doesn't if the latter is able to reorient parts of its body (namely the head) to bring its target back into sight. The two robots would still be in the same state and require the same high level decision to perform the task. The different informational modalities describe, in essence, the same state and can thus be mapped onto each other. See Figure 2.3 for an example.

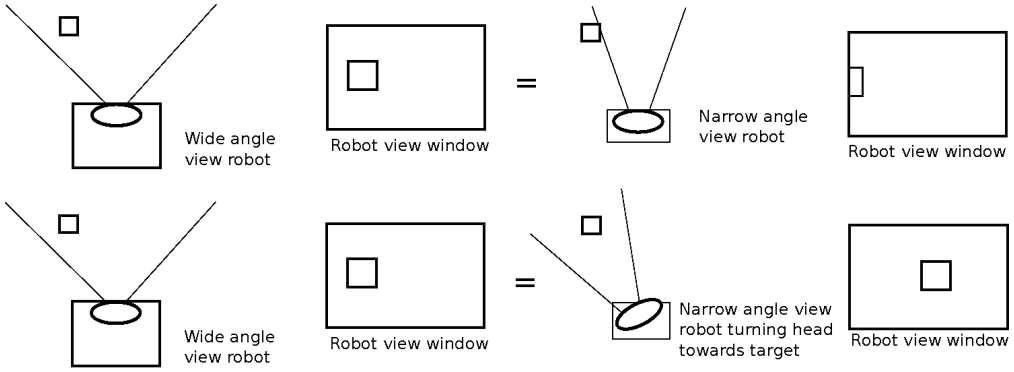


Figure 2.3: The state of a wide angle view robot with a fixed head is equivalent to the state of a narrow angle view robot with a moving head with regards to a particular set of actions, such as deciding which way yo turn the body. The state in this example is expressed through a 3-valued vector for the robot on the left side (x,y coordinates of the object and the size denoting the distance) and a 5 valued vector for the one on the right (x,y,size and the yaw and pitch angles of the robot's head).

If the information required to perform the task exists in both robots, regardless of sensor modality, then is is possible to create an adapter that allows the use of one robot's (the transfer robot) sensors with the controller of another robot (the original). The controller's outputs can then be mapped back onto the transfer robot's motor interface instead of the original one (Figure 2.4).

By using machine learning techniques, it is possible to provide the baseline for a full behavioural reproduction and optimization on the new body using these adapters. As long as the information is sufficiently correlated between sensor abstractions and both bodies are capable of performing the

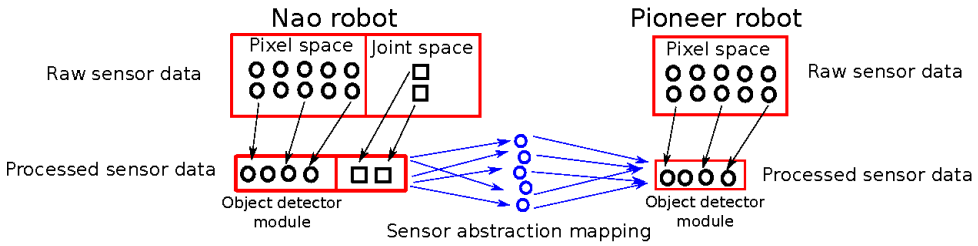


Figure 2.4: The new robot's input abstraction is mapped onto the original one's and later processed by the controller. Here the mapping is performed using Artificial Neural Networks (described in further detail in section 2.3)

required task this is possible.

The challenge lies in finding ways to define equivalent states for the two robots and to find a way to map between the states in a fashion that is stable and sufficiently general. Since there is no literature available on this topic, this section of research is largely exploratory.

2.2.3 Learning controllers

The next major issue involves learning the controller itself. While not directly necessary, as the original controller can still work quite well, it does benefit the final full system assembly. Doing so would have two major advantages: 1) The learned controller in the end fits in better with the entire architecture, allowing for the entire system to be optimized later and 2) it turns the controller from a hard controller into a soft controller. This allows for some hard definitions which in the original robot function as designed but in the new robot can no longer attain the same functionality, to perform an approximation of the original function. For example, if the controller tests an if clause on a hard value, for example: "if x == 1.0" , but the sensor input, being an approximation of the input that the controller expects, is not exactly 1 but 0.98 then the entire controller will fail. However if we also learn an approximation of that statement then it is easier to fit within the range at which the controller outputs an approximation of what is required. The controller could even not be defined or hard-coded itself, but learned from a human demonstration. This boils down to using apprenticeship learning to learn the state-action mapping, which is an established domain and has been researched extensively.

A schematic of the entire control scheme can be seen in figure 2.5. Here, processed sensor data in the form of object detections are fed through a controller who's outputs are fed as a control vector into the API of the robot and generates actions. This controller can then be approximated and, when combined with the results from 2.2.2 and 2.2.4 form one integrated control pipeline on the new robot that can later be globally optimized.

2.2.4 Matching outputs

The end goal is to perform actions that have the same effect on the new robot as on the previous one. Thus, some action equivalence needs to be drawn between the different robots. Action equivalence

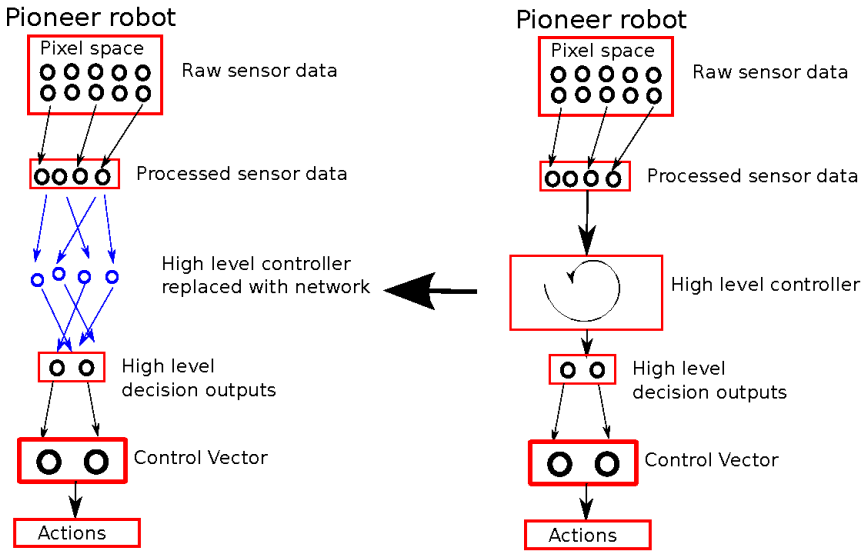


Figure 2.5: The controller on the original robot (Pioneer) gets replaced by an approximation. In this case performed with Neural Networks (described in further detail in section 2.3). The defined controller (right) is approximated with Neural Networks (left) to perform a softer approximation of the same function.

between all possible actions would be preferable, but at the very least, action equivalence between actions displayed by the controllers is necessary. To obtain such action equivalence, we are required to obtain corresponding control vectors for the API between robots. Since the information regarding the action already exists in the controller output, it is not actually required to map API control vectors to API control vectors. It is sufficient, and not more technically challenging to just map the controller output to the required API values. The full migration scheme can be seen in figure 2.6.

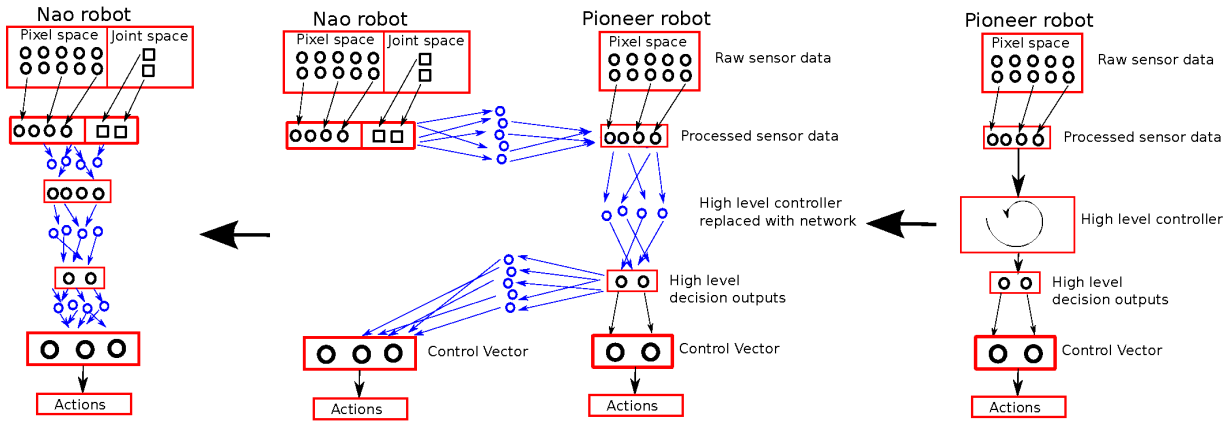


Figure 2.6: The complete migration scheme. The controller gets approximated by soft computing approaches and equivalences are found between the inputs and outputs (actions) of the secondary robot (Nao) and those of the primary robot (Pioneer). In the end a Neural Network approximation of the entire processing pipeline activates on the new robot to perform the same behaviour.

However, a context in which these action and control vector equivalences can be drawn is harder to define. What exactly does constitute equivalent actions when the modalities are so different? In the paradigm employed by this thesis, two actions are equivalent if they both cause the same state transition. If taking action 1 gets you from state A to state B and action 2 also gets you from state A to state B then these two actions are equivalent. However, questions arise as to how does this paradigm function in a non-discrete environment such as the real world, where observations are continuous or discrete but of high density? Literature on this topic is scarce, but this thesis explores two possible solutions to the problem.

2.3 Proposed learning methods

Artificial neural networks [28] have proven to be versatile learning tools and have been used to solve a wide variety of problems including robot control [29, 30, 31, 32]. They use a number of units (neurons) that compute nonlinear (typically sigmoid) functions of weighted sums of their inputs. Networks with sufficient such neurons and proper parameters have been proven to act as universal approximations. Neural networks come in different flavours. The ones deployed and examined in this thesis are presented below.

2.3.1 Feedforward Neural Networks

Feedforward neural networks (FFNN), seen in figure 2.7, are the evolution of Rosenblatt's perceptron [33]. FFNNs have one or more hidden layers which take an input vector and compute a non linear function of weighted sums of these vectors:

$$hidden_j = f(\sum_i (weight_{ij} * input_i) + bias_j) \tag{2.1}$$

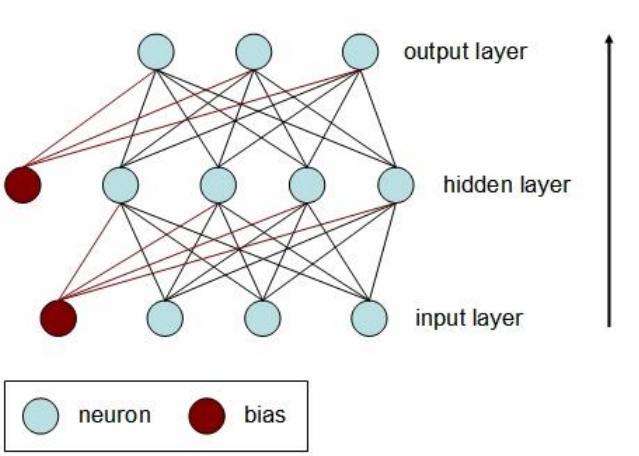


Figure 2.7: Typical feedforward neural network structure with 1 hidden layer. Image taken from http://statsoft.com

If there are several hidden layers, their activations get computed by applying nonlinear functions of the previous hidden layer:

$$tophidden_k = f(\sum_j weight_{jk} * bottomhidden_j + bias_k) \tag{2.2}$$

The function f is usually a smoothed step function such as hyperbolic tangent:

$$f(x) = tanh(x) \tag{2.3}$$

or the logistic function:

$$f(x) = s(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

Outputs are computed in a similar manner except the function is usually the identity function $f(x) = x$

The preferred method of training for neural networks is backpropagation [34]. The algorithm tries to minimize the error

$$E = desiredvalue - actualvalue \tag{2.5}$$

by using a process of gradient descent for which we need to calculate the gradient of the error with respect to each of the weights that leads to the respective unit

$$\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_n}) \tag{2.6}$$

and each weight is updated with the increment

$$\triangle w_i = -\gamma \frac{\partial E}{\partial w_i} \tag{2.7}$$

where γ represents a learning constant, a proportionality parameter which defines the step length of each iteration in the negative gradient direction. The learning problem now reduces to the question of calculating the gradient of a network function with respect to its weights. Once a method to compute the gradient is found, we can expect to find a minimum of the error function where $\nabla E = 0$. This is done by computing the contribution of each weight to the total output error. This is relatively simple for output units which are generally linear (thus the error is also linear), but for hidden units the derivative of the transfer function must be computed. Luckily, for both tanh and sigmoid functions these are relatively easy to compute:

$$tanh'(x) = 1 - tanh^2(x) \tag{2.8}$$

and

$$s'(x) = s(x)(1 - s(x)) \tag{2.9}$$

The method works by first forward computing the network outputs, calculating the error at the outputs and at all the hidden units then adjusting all of the weights in the direction opposite to the error. Then repeating the process until an adequate stopping criterion has been reached (usually a very low error or a sufficient number of iterations).

The algorithm can be effectively summarised as:

1. Initialize the weights in the network (often randomly)
2. Compute the forward pass: Output = neural-net-output(network,input)
3. Knowing the desired output (Target), calculate the error $E = Target - Output$ at the output units.
4. Compute $\triangle w_h$ for all the weights from the hidden layer to the output layer (Backward pass)
5. Compute $\triangle w_i$ for all the weights from the input layer to the hidden layer. (Backward pass)
6. Update the weights of the network
7. Return to (2) until the stopping criterion has been satisfied (low enough error or enough training epochs have been attempted)

2.3.2 Recurrent Neural Networks

Recurrent neural networks [35] or RNNs are an extension on FFNNs that also include recurrent connections between hidden units within the same layer or from different layers and in some instances even from the output units back to the hidden units. An example can be observed in figure 2.8. These connections store information from previous time steps the network was run. In this way, past states of the inputs are also considered when computing the network's outputs. This can be beneficial in, for instance, partially observable environments where integrating the current data with data from the past acs as a form of internal memory. Training with backpropagation [36] is done in much the same way as with FFNNs by unravelling the network through time, effectively flattening it to become a larger FFNN. The issue with training RNNs is that the error gradient becomes zero for far back time steps. For the current problem, however, this is not a major issue as the controllers deployed do not usually make use of data from previous steps in their decisions and when they do the relevant data is a relatively small number of time steps in the past.

The activation of a hidden unit for example is

$$hidden_j(t) = f(\sum_i (weight_{ij} * input_i(t)) + \sum_l (weight_{lj} * hidden_l(t-1)) + bias_j) \tag{2.10}$$

in the case of one single fully connected hidden layer. Depending on the actual implementation of the structure this may differ.

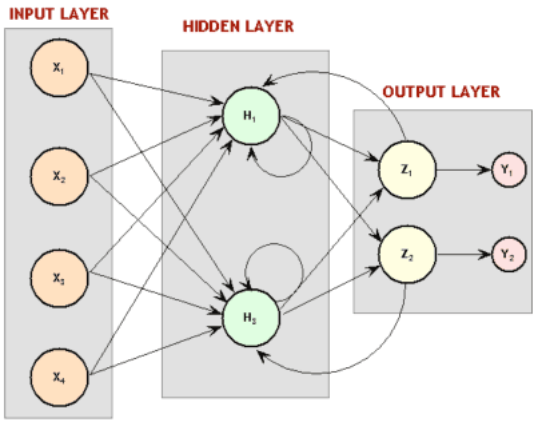


Figure 2.8: Recurrent neural network with 1 hidden layer and output to hidden connections. Picture taken from http://www.information-management.com/

2.3.3 Echo State Networks

As a departure from the traditional school of thought, echo state networks [37, 38] or ESN (see figure 2.9), are recurrent neural networks with a sparsely connected hidden layer known as a 'reservoir'. Connectivity is typically 1% according to the original literature, however, other researchers [39] have obtained good results with significantly higher connectivity, and for the current implementation this restriction on connectivity is not considered as connectivity is allowed to emerge out of the training method. The connectivity and weights are typically randomly assigned and fixed, while the weights of the output neurons can be learned so that the network can reproduce the desired patterns. Usually a simple linear readout function is used as the reservoir is generally highly non linear in itself.

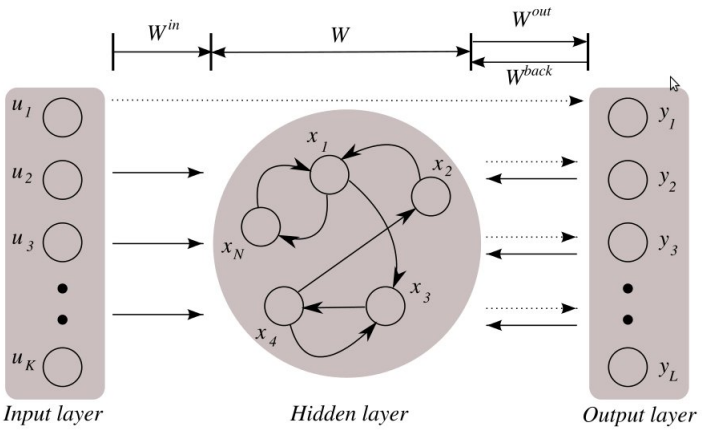


Figure 2.9: The Echo state network. The 'reservoir' is a sparsely connected fully recurrent neural network that does multiple nonlinear dynamic transformations of the input vector and potentially the output or other results. Only the output connections are trained in this case, the rest remaining static.

The activation states of the reservoir units are updated using

$$reservoir(t) = f(W^{in} * inputs(t) + W * reservoir(t-1) + W^{back} * outputs(t-1)) \tag{2.11}$$

where f is the transfer function for the reservoir units, W^{in} are the weights from the input to the reservoir, W are the recurrent reservoir weights and W^{back} are the weights connecting back from the output to the reservoir. The output is calculated using

$$outputs(t) = f^{out}(W^{out}(inputs(t), reservoir(t), outputs(t-1))) \tag{2.12}$$

where f^{out} is the output activation function (typically the hyperbolic tangent or identity function) and $(inputs(t), reservoir(t), outputs(t-1))$ is the concatenation of the input, reservoir and output vector. W^{out} are the output weights.

W^{in}, W and W^{back} are generated randomly, and W is scaled to have spectral radius of α where $\alpha < 1$. This is to ensure the echo state property, meaning the signal decays with time. Failing to do so eventually leads to large oscillations or the neurons entering saturation.

The reservoir is initialized arbitrarily, and the states are updated for $t = 0, 1, ...T$ using equation 2.11. Network states before a washout time T_0 are discarded due to their dependency on the initial state. The rest of the network states are collected in a matrix M and the correct outputs for the respective inputs are collected into a matrix T. The output weights are computed by evaluating the pseudo inverse matrix of M and multiplying it by the transpose of the targets T.

$$W^{out} = M^{+}T^{T} \tag{2.13}$$

While this works relatively well, results are still dependent of the network's initialization. In here a method to optimize the network structure is used derived from simulated annealing [40]:

- The network is initialized randomly, with a spectral radius lower than one.
- Training proceeds as in a typical environment. States are collected and the output weights are computed using the pseudo inverse method for linear regression.
- Gaussian (initially μ=0 and σ=1) noise is added to the input and recurrent weights and the network states are collected again, the outputs recomputed and performance re-evaluated.
- If the new error is smaller than the previous error then the new reservoir is retained as the network of choice and the old reservoir is discarded. There is also a small chance that the new network is retained regardless of the improvement in error. In the current implementation, this starts at 10% and decreases with the same factor as the noise (see below). This probability thus takes the shape of $p(a) = 0.1\sigma$.
- Decrease the σ of the noise by a certain factor (in this case 0.99 of its previous value). Thus the added noise σ follows a function of $\sigma = 0.99^t$ where t is the current epoch (starting at 0).
- Repeat from the third step until the error no longer decreases or the maximum number of training epochs has been reached.

While there is no available literature elaborating this method, it has shown to work well in practice for optimizing reservoirs. Learning performance is compared in section 3.4.

2.4 Methods discussion

In this chapter an integrated mechanism for migrating controllers between robots was proposed, and the methods for deployment of this mechanism have been presented. This constructs the theoretical basis for the implementation and experiments conducted in chapters 3, 4 and 5. Building upon the proposed framework, we explore the performance of these methods to learn controllers (chapter 3), establish input equivalence between the different robots (chapter 4) and learning models of robot dynamics and learning equivalent outputs from such models (chapter 5). Each section builds on top of the results of the previous one, leading to fully working robot control mechanisms in the later sections. While a separation of the three sub problems is attempted, these are ultimately, intrinsically, linked as embodiment is, even philosophically still, not an inherently modular concept.

Chapter 3

Learning behaviours

3.1 Introduction

Robotic controllers are systems that make decisions regarding actions that the robot needs to take, potentially based on input from sensors or the state of internal memory. These are often referred to as behaviours or policies. The decisions then transform into signals for motors and actuators in order to set the robot in motion and thus produce the desired effect. Policies are often hand programmed by researchers and engineers and fine tuned to achieve the desired effect.

It may be the case, though, that programming policies by hand is not a feasible endeavour. Coding may either take too long, or be too difficult to perform or the static nature of the method may not suit the needs. Policy designers may want to further autonomously refine or optimize their controllers after their initial implementation and this cannot be done with simple, hard coded methods. One may possibly want to avoid defining rules and processes by hand. To that end, learning from demonstration (LFD) [41, 42, 43], or apprenticeship learning is the process of learning policies by observing examples of application of the policies by outside sources or demonstrators. Robots will then try to imitate or reproduce the demonstrated policy using machine learning techniques. The method is a proven alternative, receiving growing investigation from the scientific community.

3.2 Related work

LFD has seen various successful implementations. Initial research has focused on simple optimisation tasks such as teaching a robot to perform the inverse pendulum swing up task [44]. This was carried out by means of learning a task model and a reward function from human demonstration in order to compute an appropriate policy.

More complex tasks have been demonstrated in [45] where several external cameras were used to observe a human demonstrator showing a robot how to serve water in cups. The robot then added the motions to its motion library and reproduced the task. The framework was also able to generalize its actions to some extent and pour water at locations previously not shown demonstrations for.

In [46] the authors use multiple demonstrations to teach a task as the robot learns just the correspondence between his own possible actions and the actions performed by the human demonstrator. After the robot has learned to generalize they use the teacher's feedback to refine the particular task knowledge.

In [47] the robot is taught actions by a human demonstrator in a specially designed learning center. By making use of extra sensors such as data gloves for precise finger position measurements they guide the learning process of a grasping task in order to facilitate robot grasping and external cameras recording actions at all times.

Connectionist approaches have been explored in [48]. Here authors constructed a model of human imitation of abstract, two-arm movements. The model constructs a hierarchy of artificial neural networks, which are abstractions of brain regions involved in visio-motor control. Human demonstrations of arm movements were collected using video and marker based tracking systems and the model was then validated in simulation.

Apprenticeship learning has also been demonstrated on autonomous cars for path planning tasks [49]. Here, a number of possible good trajectories were demonstrated for a car to follow in a parking lot by learning the respective trade-offs to make in various situations. The car was then able to reproduce various trajectories, retaining to some extent the driving style of the demonstrator.

The authors of [50] used reinforcement learning to learn different walking styles from demonstration by determining the appropriate reward function and showed that the particular reward function can be also applied to different environments and scenarios. The implementation is also adaptive, as the demonstrator can refine the process with just a few additional examples.

A group of researchers even went one step further to demonstrate to the robot how not to perform actions [51]. A teacher fails to demonstrate a particular behaviour and the robot learns a policy that tries not to reproduce those results by using those demonstrations as constraints on the search space.

3.3 Methods

3.3.1 Proposed approach

The controllers are learned using LFD, by running the original controllers from various locations in front of their targets and recording state(observation)-action(controller outputs) pairs. Feed Forward, Recurrent and Echo state networks are then trained on the datasets previously acquired. See section 2.3 for details. Only the Pioneer robot was used in this case, as it is the only robot a controller was defined for.

3.3.2 Experimental setup

The task at hand is to learn controllers facilitating the Pioneer robot equiped with a webcam as its sole sensor to 1) navigate towards a target on its screen, 2) avoid a target on its screen and 3) both.

Acquisition of observable states takes place as described in section 2.2.1. Usually a human demonstrator would show the robot its desired behaviour. However, to ease the experimental process a controller has been programmed by hand and used as a demonstrator. The controller outputs a force vector based computed from the size of the object on the screen and its position. The force vector represents an abstraction of the strength of the forward movement and turning of the robot.

Approach behaviour

For this experiment, a controller has been designed that tries to keep its target object in the middle of the screen and taking up a size between 35% and 50% of the total visible area. If the target is more to the left it will turn left, if it is more to the right, it will turn right. If it is smaller than 35% of the screen size it will move forward with a speed of approximately 0.3m/s. If it is larger than 50% it will move backward with a speed of approximately 0.06m/s. 35% and 50% correspond to roughly 60 to 40 cm away from the target. The distance was chosen for safety reason so the robot does not accidentally collide with its intended target.

The robot was placed in 9 positions in front of the target (see figure 3.1). These positions are aligned in a square grid of roughly 35cm apart. The controller was run from those positions, and the robot moved towards its target and stopped when the stopping condition was achieved. Input (target parameters) and output (control vectors) data was collected during the runs (5 runs for each point to ward off potential noise in the system).

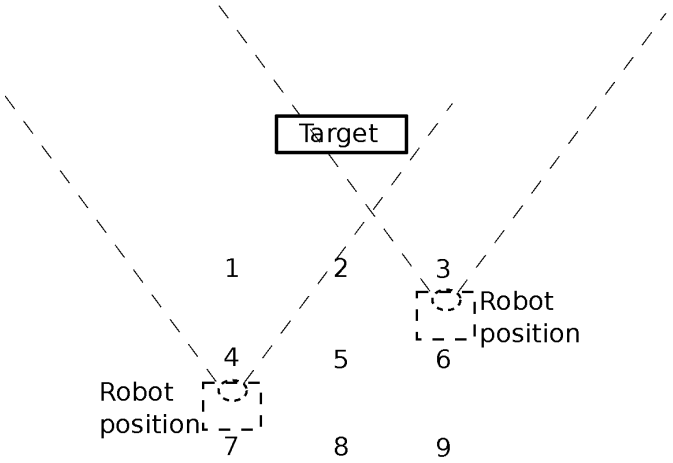


Figure 3.1: The robot is placed in front of the numbered markers and faces the target. Two possible robot placements and the robot's field of view are presented here.

The data collected (input-output pairs) was then used with the various learning methods to learn new controllers in the following manner:

1. The demonstrator controller was run from the 9 locations and input-output pairs were recorded. Namely, object parameters, (x position, y position, size, visibility) and control vectors (movement vector, turning vector). A total of 8254 such data pairs were collected from the 9x5 demonstrations.

2. The data pairs are divided into 40 training demonstrations (6891 training data points) and 5 testing demonstrations (1363 testing data points).

3. Run the learning methods: FFNN (with one fully connected hidden layer and 5, 10 and 20 hidden units), RNN (with one fully connected recurrent hidden layer and 5, 10 and 20 hidden units) and ESN (with reservoirs of size 50, 100 and 200 units) to learn the policy demonstrated. Do so in two modes: 1) One network for all the outputs and 2) One network for each of the policy outputs(reported below as mFFNN for structures consisting of more than one FFNN per output for example with an appending number denoting number of hidden units "-5h" for 5 hidden units). Due to the different nature of the respective outputs (one is a motion vector, the other a rotation vector), the possibility that the different hidden units interfere with one another is investigated.

4. The networks of each type with the lowest testing error are retained to be run on the robot.

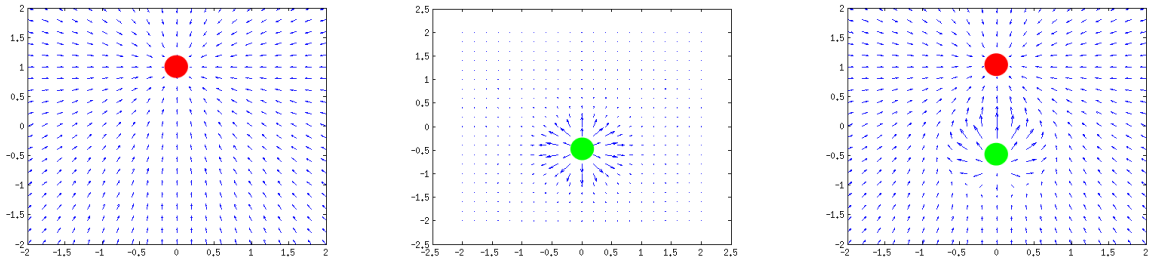
5. Evaluate the time required to complete the policy for each of the 18 controllers and compare them to the original demonstration. Run the robot with the newly learned controller from all 9 positions 4 times and measure the time taken until a full stop. A run was considered completed when the robot no longer moved in any observable fashion after the stopping criterion was reached (target size between 35% and 50% screen occupancy). The results were averaged and reported in section 3.4.

Learning was conducted using the pybrain [52] framework for FFNN and RNN and a python [26] + numpy [53] implementation for the ESN. Learning was conducted on an Intel core i7 2.1 GHz laptop for 100 epochs of learning for FFNN and RNN and 1000 epochs for ESN (ESN needs more steps to search through the neighbourhood, however it does so much faster since it does not need to compute gradients). A learning rate of 0.05 and 0.8 momentum was used for FFNN and RNN and a starting temperature of 0.9 (gaussian noise multiplier) with a decay of 0.99 for the ESN.

Avoid behaviour

For this experiment a controller has been designed to steer away from a target on the screen, which in this case consists of a human fist sized green ball. The controller is designed to turn away from the ball (if the ball resides on the left side of the screen, the controller will turn the robot right, if it resides on the right side of the screen the controller will turn the robot left). This turning speed increases with the square of the size of the obstacle on the screen, thus closer to the robot (if the obstacle is larger the robot will turn faster). Since this is a rather simple, purely reactive behaviour to learn only basic feedforward networks have been tested. Only one demonstration of 1880 datapoints (observation-action pairs) has been collected for training and another one of 1248 was used for testing. The network instances in an epoch with the lowest testing errors were saved and used on the robot.

Training was conducted for 500 epochs for each of the networks (feedforward with 5, 10 and 20 hidden units) using the same machine, software and parameters as in the previous behaviour. However, since this is a purely reactive behaviour that functions on a relatively short term, it is difficult to obtain measurable statistics. The duration of reaction is shorter than the noise induced by the sampling itself (time to get control architecture, camera initialization and initial detections and robot movements started up). As such, the only errors reported are those stemming from the training process (the



(a) Simple abstraction of an approach behaviour represented here as an attraction field.

(b) Simple abstraction of an avoid obstacle behaviour represented here as a repulsive field.

(c) Combination of approach target and avoid obstacle abstraction computed as a sum of its constituent fields.

Figure 3.2: Robot behaviour abstraction, shown above as vector fields representing the direction the robot must head towards seen here from a top down perspective.

error resulting from the testing set) and not from actual deployment of the methods on the robot itself.

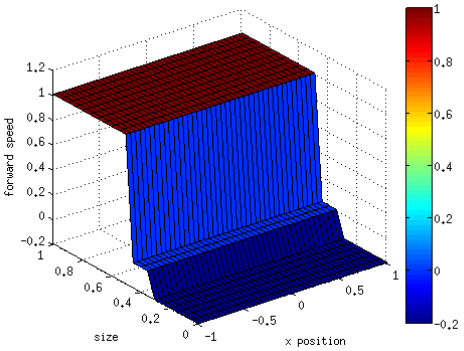
Combining behaviours

Because both controllers above output the same data format (a movement and a turning force), they can be combined relatively easily. According to [27] much of a robot's actions can be represented as a vector field. A conceptual representation of this can be seen in figure 3.2.

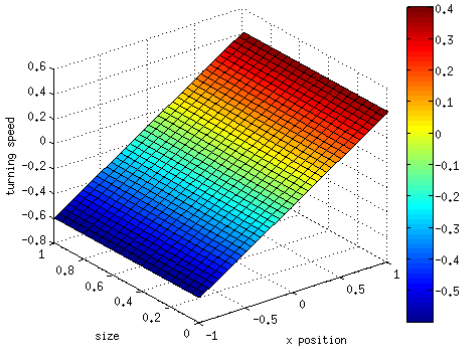
Figure 3.2 is an abstract global interpretation of behaviour composition. The final composed behaviour is obtained by summing up the outputs of its constituent controllers. In this case a move-while-avoid behaviour's output is the summation of the outputs of a move behaviour's output and the output of an avoid behaviour). In reality the span of the vector field would be greatly reduced due to non-global sensors. If you do not know where you are and where your targets and obstacles are your policy cannot deduce a control vector. Thus your policy is restricted to your observable space. This can be detrimental but can still be achieved with some minor approximations based on the last observed state to quite some degree of accuracy.

Figures 3.3 and 3.4 show the progression of the output/control vectors with regard to the input vectors. Adding up the points on the landscapes leads to an overall complex meta-behaviour solution. A thing to take into account is that different behaviours have different targets of different sizes and possibly leading to different sensory changes. Thus the size and position used in one behaviour is different from the one used in the other behaviour since they reflect different inputs.

The approach behavior described earlier (and its neural network approximator) estimates the required vector field within observable space. It does so straight from sensor space without any transition through models or global representations. This can be quite a powerful behavioural composition tool. However, this does give rise to some complications given the current models. Because the state is only partially observable, required vector values cannot be computed in every spot such as when the robot's target and/or obstacle is outside the camera's field of view. This is more thoroughly described

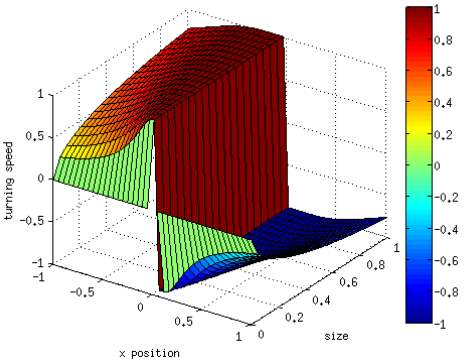


(a) Forward force for the approach target behaviour as a function of size and position on the x axis. The position on the x axis does not influence approach speed at all while the size determines whether the robot is to go forward, back or stand still.

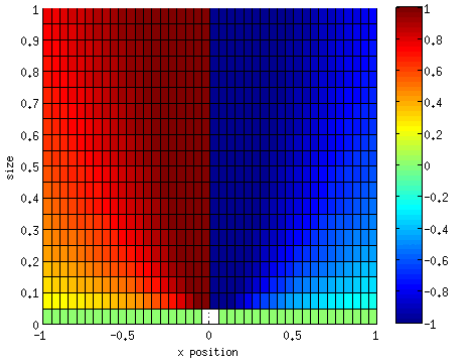


(b) Turning force for the approach target behaviour as a function of size and position of target on the x axis. The more to the left the target is the higher the turning speed.

Figure 3.3: Plots of the force functions for the approach target behaviour (3.3a for the movement force and 3.3b for the turning force) with regards to the size taken on screen and position on the x axis of the desired target.



(a) Turning force for the avoid target behaviour.



(b) Turning force for the avoid target behaviour.

Figure 3.4: Plots of the turning force function for the avoid obstacle behaviour as a function of the x position and size taken up on screen. The robot tends to turn faster the closer the obstacle is to its center and slower if closer to its periphery. Upon crossing the middle line, the direction of turning reverses. Both plots express the same function, just from different perspectives. The move vector is null in this instance so there is no plot to show (The robot's wheels turn in different directions in case of a turn command so there is no induced movement from the turning behaviour).

in figures 3.5, 3.6 and 3.7. As such, some approximations need to be made for such scenarios and extend a behaviour's domain of definition.

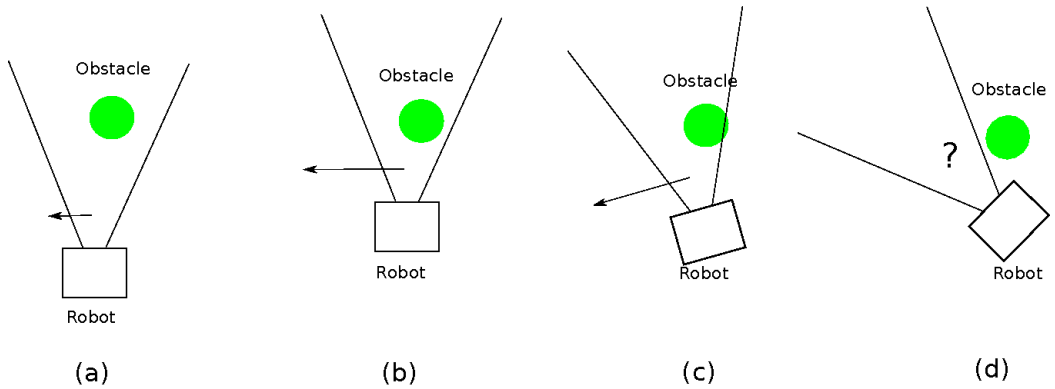


Figure 3.5: Faced with an obstacle the robot turns away from the obstacle (a). The turning strength increases the closer the robot is to the obstacle (b). The robot keeps turning away from the obstacle as long as the obstacle is on screen (c). This process breaks down if the obstacle moves outside the robot's field of view (d).

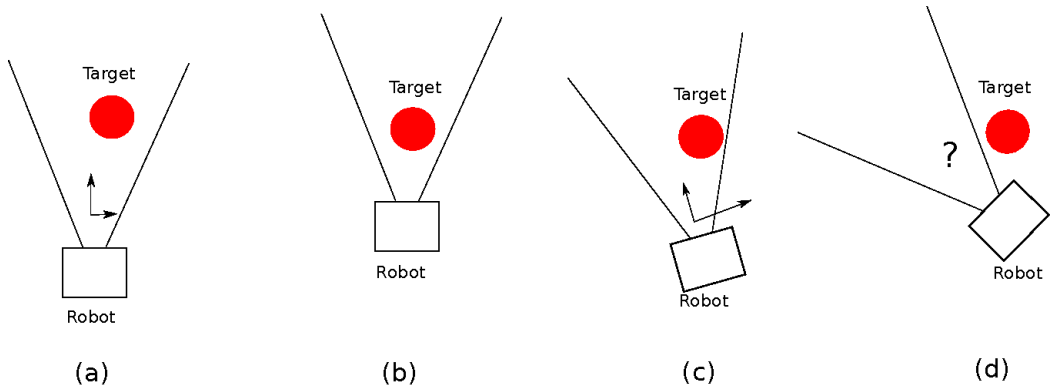


Figure 3.6: The robot generates force vectors to approach the target (a). If the robot is close enough to the target then the force vectors are null (b). The turning force increases as the target deviates from the center of the field of view (c). The process breaks down when the target is outside the robot's field of view (d).

The behaviours were expanded as follows:

For the approach behaviour, the target's position is retained in its last observed position for another 30 seconds. Thus the robot somewhat remembers the object is still in the same position as its last observation (roughly to the utmost left or right of the image and roughly the last observable size thus distance). As a consequence, once the robot ceases to see its target, it will still continue heading towards where it last saw it. While this is not bound to happen when running the approach behaviour itself (unless the target is moved by a human agent) it is very plausible in the instance where one combines the behaviours such that one behaviour pushes the other one outside of its domain of definition.

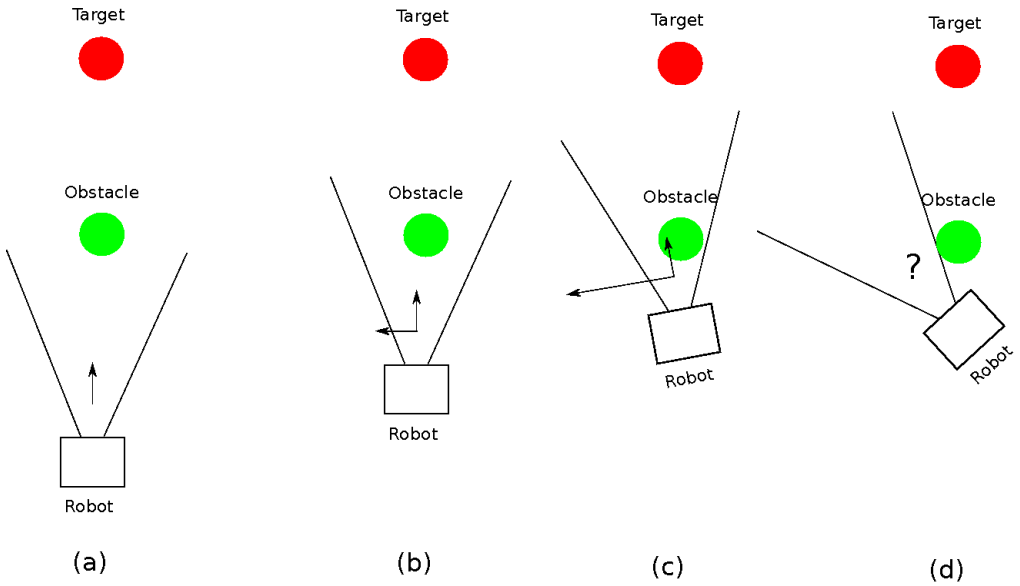


Figure 3.7: Placed in front of both an obstacle to avoid as well as a target to approach, the robot sums up the two impulses. Since the robot is far away the obstacle repulsive force is minimal thus the robot will try to approach its target as usual (a). Getting closer, the robot gets pushed away from the obstacle with increased force while still being attracted to its target (b). As the robot turns, it attempts to turn both towards the target as well as away from the obstacle. Since the obstacle force is stronger when closer, this suppresses the desire to turn towards the target (c). The process breaks down once both move outside the robot's field of view (d).

For the avoid behaviour, the obstacle's position is retained in its last observed position for another 5 seconds. However, the turning rate reduces during this interval by 5% of its previous value every 100ms (equal to multiplying by 0.95). This lets the robot turn increasingly slower over time for the respective 5 seconds.

In both scenarios, once the robot has elapsed its "memory" time, and the objects move out of the visual memory, the target registers as not visible and that controller ceases output (outputs zero for both vectors) and no longer adds to the final solution (the final solution being the sum of the partial solutions). In this manner we have extended the behaviours' domain of definition for an area beyond its initial domain. However, at least for the avoid behaviour, the resulting outputs are time dependent. Initial experiments have shown that recurrent networks perform suboptimal for this sort of tasks and feedforward networks are better suited. Thus the concept of "time", or in this case, "time elapsed since the last observation took place" is simply added as an extra input to the controllers being trained.

This is beneficial for the task in several ways. Unlike regular recurrent networks which must be recomputed at fixed time intervals (for example 10 times per second) to keep consistency, and time delay feedforward neural networks (use several time steps from the past as input for the feedforward network) simply adding time as a variable can sometimes be a way to provide the time component while not significantly altering the reactive nature of feedforward networks. This can to some extent transcend variable time lags which, given our task of ultimately transferring controllers between robots

with different actions that may take place over different time periods (actions on one robot (Nao) may be slower than on the other robot (Pioneer), mostly for physical reasons). Using time as an input for the new input vector (x position, y position, size, time, visibility) can facilitate the later learning of these different mappings also over time. Preliminary testing shows that this works well in practice.

Two extra demonstrations were given to both behaviours. One for the training set and one for the testing set. For the approach behaviour, the object was presented to the robot then taken out of the image through both the left and the right side of the visual field. The robot would follow the object's trace for the determined time until it either saw it again or the time would elapse and it would stop turning. The original 9 demonstrations from 3.3.2 had to be shown again. For the avoid behaviour, the object would be presented to the robot from various distances and the robot would turn until the object would be outside of the visual field. Then the robot would keep on turning, slowing down by a factor of 0.95 each 10th of a second for another 5 seconds. Once the five seconds elapse the behaviour stops and begins to output null vectors unless the obstacle is presented again to the robot. This is repeated several times on both sides of the visual field within one single long demonstration. All demonstrations are then learned again, using the best performing network structures from the previous experiments but with the new demonstrations and the extra temporal input.

The two targets: approach (red) and avoid (green) were both placed in front of the robot as to create an obstacle that the robot is to avoid on its way to its target location. The composed controller was then run for the demonstrated and learned behaviours on the robot. This was done for 30 times each from different starting positions and the success rates (reaching the target and not crashing into the obstacle) were measured and compared.

3.4 Results

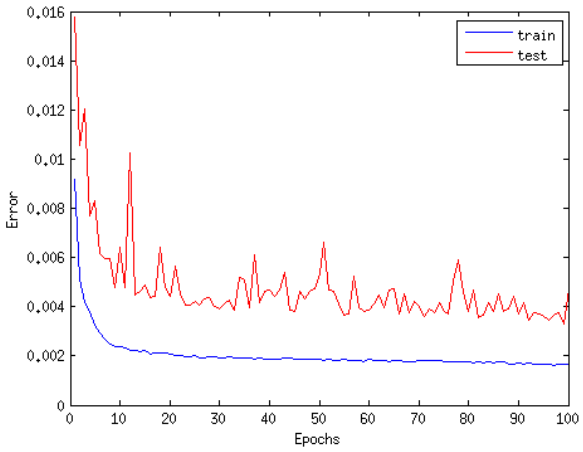
3.4.1 Approach behavior

All of the multiple RNNs (mRNN - one RNN per output) failed to reach the target at all. This was the same with the simple RNN with 20 neurons as well as the multiple ESNs with 50 and 200 neurons. All of the simple ESNs simply produced controllers that did not move the robot at all. The best performance was observed with FFNNs and, to some extent, multiple ESNs. A comparison of testing errors can be seen in table 3.1. Some examples of the progression of the training progress can be seen in figure 3.8.

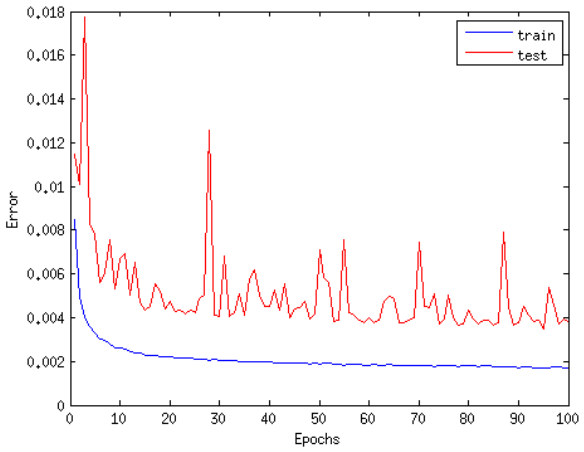
FFNN-5h	**FFNN-10h**	FFNN-20h	mFFNN-5h	mFFNN-10h	mFFNN-20h	RNN-5h	RNN-10h	mESN-100h
0.0033	**0.0033**	0.0037	0.0036	0.0035	0.0036	0.0233	0.045	0.042

Table 3.1: Testing mean square error of successful policies. The feedforward network with 10 units in the hidden layer had the smallest unrounded error.

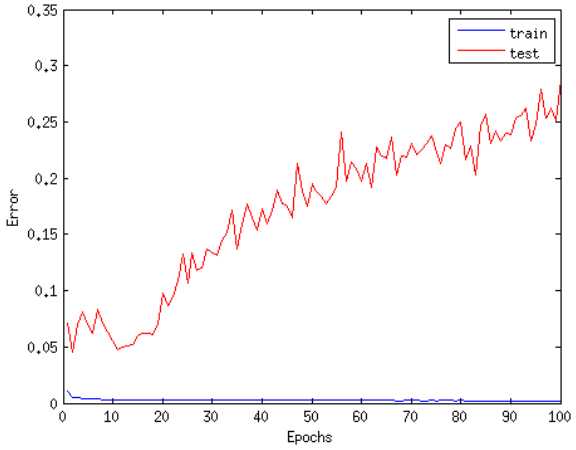
The mean times over 4 runs were recorded and are presented below in table 3.2. Every controller completed the behaviour in times slightly different from the original controller, due to external factors outside of experimental control (camera noise, robot start-up time, evaluation of behaviour completion). Some controllers such as the one derived from multiple feedforward networks with 10 hidden (mFFNN-10h) units even completed it faster, 0.54 seconds faster on average. This was to be expected



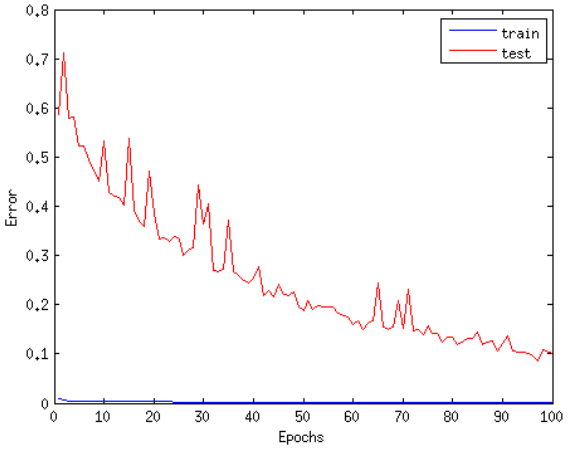
(a) FFNN with 10 hidden units.



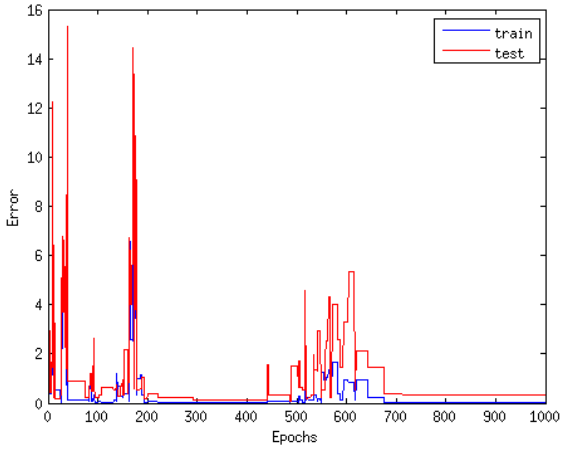
(b) FFNN with 10 hidden units,one per output.



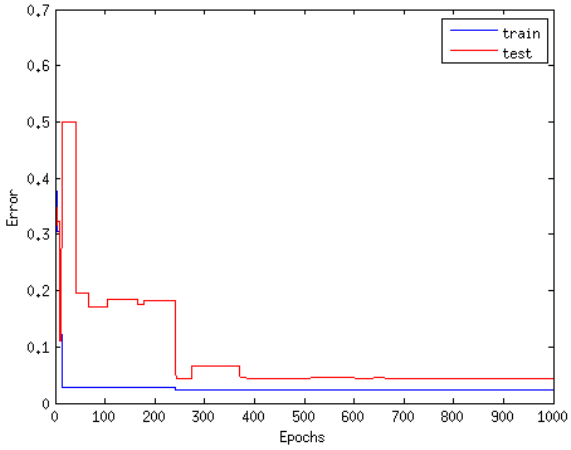
(c) RNN with 10 hidden units.



(d) RNN with 10 hidden units, one per output.



(e) ESN with 100 hidden units.



(f) ESN with 100 hidden units, one per output.

Figure 3.8: Progression of training (blue) and testing (red) errors (MSE) for several neural network models tested for the approach behaviour. 3.8a through 3.8d were trained for 100 epochs while 3.8e and 3.8f for 1000. 3.8e is hard to optimize since usually improving the reservoir for one output decreases performance for the other and the testing error for 3.8c only gets worse with further training. 3.8d could still improve but training was cut short due to time constraints.

as the original implementation was not a perfect demonstration and rounding errors in the learning method could lead to less overshooting of the target.

	Position 9	Position 8	Position 7	Position 6	Position 5	Position 4	Position 3	Position 2	Position 1	Average time
Original	**22.8**	**23.2**	**22.9**	**19.5**	**18.0**	**19.6**	**13.2**	**11.5**	**12.0**	**18.1**
FFNN-5h	28.8	28.1	26.1	22.4	25.5	25.3	15.3	20.2	13.2	22.8
FFNN-10h	21.3	20.1	19.1	19.2	22.0	23.2	13.3	10.3	14.4	18.1
FFNN-20h	23.9	24.6	25.2	19.2	22.1	23.2	13.3	10.3	14.4	19.6
mFFNN-5h	25.1	23.1	26.1	21.7	18.9	24.8	21.4	12.9	16.6	21.2
mFFNN-10h	**22.5**	**20.5**	**22.7**	**15.4**	**18.2**	**21.1**	**13.8**	**10.8**	**12.8**	**17.5**
mFFNN-20h	23.5	23.1	22.8	20.4	19.5	21.3	11.6	11.9	14.1	18.7
RNN-5h	29.6	20.4	26.4	22.6	13.7	19.2	14.9	11.5	10.9	18.8
RNN-10h	24.4	24.2	24.4	28.7	23.9	24.6	14.6	23.8	21.0	23.3
mESN-100h	25.0	26.1	26.8	21.2	21.1	20.0	16.9	14.2	12.6	20.4

Table 3.2: Mean time to task completion of the successful controllers, averaged over the 4 runs. Numbers at the top represent the 9 positions the robot was started from. See figure 3.1 for the respective positions. One network with 10 hidden units for each of the outputs (mFFNN-10h) seems to have performed the quickest.

However, with regard to actually reproducing the demonstrated behaviour itself, the multiple feedforward neural network with 20 hidden units (mFFNN-20h) performed slightly better. The extra neurons allowed it to learn the demonstrator function more accurately, with mFFNN-10h leading to a smoother policy that did not overshoot as much as the original demonstration thus leading to shorter times of completion.

A possible method of improving the error and results would be to perform a weighted average of the output of two or more controllers. However, mFFNN-20h (one FFNN per output with 20 hidden units each) performs sufficiently well in practice so it is not attempted here.

3.4.2 Avoid behaviour

Extrapolating from the successes of the approaching behaviour, the avoidance behaviour was also learned using the same methodology. However, due to the success of the purely reactive controllers in the previous, more complex scenario, only feedforward networks were learned this time. A comparison of their performance can be seen in table 3.3.

FFNN-5h	**FFNN-10h**	FFNN-20h
0.0128	**0.0114**	0.064

Table 3.3: Testing mean square error of learned policies. This being a turning only task, the movement vector was constantly zero but was kept in the network for ease of assembly with the previous controller. The feedforward network with 10 units in the hidden layer had the smallest error.

Learning was successful as the robot was able to properly turn with all three networks. However, the network with 10 hidden neurons was able to reproduce the behaviour more accurately.

3.4.3 Combining behaviours

The controllers presented above were relearned with the extra input and the extra demonstrations marking the extended domain of definition. They were then rerun for the first two experiments and no significant differences (compared to the original learned behaviours) were found between runs. The extra input added to extend the domain of definition of the behaviours does not appear to influence the activity in the initial domain.

In the extended domain (when the robot does actually not see the objects and the time counter since its last observation increases) the robot performs very similar to its original demonstration. For the avoid behaviour, adding the time elapsed since its last observation allows for the time dependent decrease in turning speed. Preliminary experiments showed that this was not possible without a time component.

Due to the fact that the initial behaviours and the enhanced behaviours (both defined/hard coded and learned by networks) are not defined on the same spectrum of inputs it is difficult to compare them. Also, the environment that the extended controllers function in is more dynamic and harder to get objective measures from. However, for the avoid behaviour, the robot using the behaviour learned by neural networks stops roughly facing at the same angle as the original hard coded behaviour to within a +/- 10 degree accuracy level if run from the same spot.

The combined controllers (original defined and learned) were run for 30 times each from various positions in front of the obstacle-target configuration. The robot starts off seeing both target and obstacle and takes actions to approach it. It then gets close to the obstacle which prompts it to take action to avoid. Since the outputs of the controllers are summed up, at first the approaching actions are dominant until the obstacle is reached causing the avoiding actions to be dominant. Upon turning and avoiding it no longer sees anything and the counters since its last observations begin to increment. This leads to the turning to avoid obstacle vector (which is initially significantly bigger) to decrease while the turning towards target vector (pointing in the opposite direction) remains the same. Thus over time the robot will have turned towards its target to the point of seeing it again at which point it is already proven above that it knows what to do. The advance/move forward component of the vector keeps running as usual for the approach behaviour (the avoid behaviour has no such component) making the robot perform an arch around its obstacle. The performance of these runs is presented below in table 3.4.

Defined	Learned
81%	87%

Table 3.4: Success rates of the combination of the approach and avoid behaviours. Success is counted when the robot reaches its target without having crashed into its obstacle.

The implementation (defined) is not flawless as it is very difficult to get all parameters right to attain perfect results but it reflects well on its neural network (learned) equivalent. The learned behaviour retains many of the imperfections of the original implementation (mainly when starting from a rather far away position there isn't sufficient turning). However, it does seem to be performing relatively better than the original. This might be due to some extent from sampling noise, but it is likely a result of the rounding effect of the neural network that doesn't have the jaggedness of if-then-else

statements of the original demonstration. The smoothness allows for the turning behaviour to begin sooner, rather than wait for the time based confirmation that the object is no longer observable.

This yet again shows that the learned behaviour can be optimized later on (by means of weight alteration) as it is already a relatively optimized version of the original implementation.

3.5 Discussion

In this chapter, a number of neural network based approaches to learning from demonstration have been tried and tested. A behaviour was demonstrated from various locations and then learned with neural networks. Several topologies were tested including classical feedforward and recurrent networks as well as newer echo state networks.

Unsurprisingly, feedforward networks had the most success at reproducing the demonstrated behavior. The most successful one was mFFNN-20h with 20 hidden units and one network each per output. This makes sense, as a higher number of hidden units leads to a higher fidelity of reproduction, and the large number of data points (over 8000) made over-fitting difficult. The difference is not that large, however, compared to mFFNN-10h with 10 hidden units. After a certain point increasing the number of hidden units has less effect. Using different networks for each output in such circumstances seems logical as the outputs are of different types and will tend to enforce their own biases on the weights affecting the other outputs. Thus separating them would be a good idea in some instances.

More striking, however, was the failure of recurrent networks to perform beyond simple feedforward networks. Their representational power is significantly higher than that of simple FFNNs. It was not completely unexpected that FFNNs behave better, due to reactive, non-time dependent nature of the demonstrated behaviour, however the complete inability to reproduce it was. Independent networks especially were expected to perform better than coupled networks as it happened with FFNN, but due to the non time dependent nature of the behaviour, as well as the difficulty that comes with discarding the first few startup inputs (just as the robot starts, before the data collects in the recurrent neurons properly) makes RNNs ill-suited for such short demonstrations. Recurrent networks are, thus, quite likely to infer time dependencies where there should not be any.

Echo state networks, could to some extent reproduce the behaviour, provided their outputs were not coupled together to the same reservoir thus making it difficult to optimize. Coupled outputs made the robot unable to move, while uncoupled outputs made it hard for it to set itself within the acceptable range from the target. It is theorized that further training would alleviate the effect.

Overall, this chapter shows, that, while the choice of learning method is still an important factor to consider, depending on application, learning from robot high level controllers is indeed possible. It also shows, to some extent, that learned controllers can even be better than the original implementation, as one of the networks managed to be overall quicker than the initial demonstration. Further refinement would increase performance. However, that is to be left up to future research.

The requirements for some tasks of a time dependent component prompted the introduction of a temporal input. This was done in light of the poor performance of RNNs and the staggering performance of FFNNs. While not guaranteed to work for all problems it has worked for the current one quite well. It does also simplify the task of migrating controllers between robots that inherently function on

different time scales (the time taken to perform a similar action is unavoidable different), as a simple temporal transformation can be performed.

Combining behaviours was also proven possible through the use of standardized, additive input on the behaviour parts, although we suspect that this may not be the most scalable of approaches and for some problems some multiplicative (switching) component might also be needed. Furthermore, learned controllers (and controller mixtures) were performing sufficiently similar to their original demonstrations and sometimes even better. Although the extent to which a better performance than the original indicates a good learning method remains up for discussion. While a good controller never fails, a good learned controller succeeds and fails exactly as much as the original one and does not actually perform better than the original.

However, having a controller or a set of controllers that are adaptable is important in robot to robot controller transfer systems and that is the ultimate goal of this thesis.

Chapter 4

Learning the input correspondence

4.1 Introduction

One of the main difficulties regarding the transfer of controllers from one robot to another lies in the fact that the robots are often dissimilar. Different robots may sometimes have very different sensors, as well as different sensor data extraction modules, to deal with. Since robot controllers often require sensorial input to perform their required actions this leads to a complication regarding input representation. In an ideal scenario, all input to a controller from any robot would be standardized, if not at a sensor level (have the same types of cameras and the same number of cameras for example), at least on a sensor abstraction level. This would mean that, regardless of the type of sensorial input the robot possesses, the information extracted is the same and in the same format. In real scenarios, however, that is often not the case.

Even within the same type of sensing and measuring devices, results outputted are often not identical but differ by a small margin. To compensate for these margins sensors to be used are initially subjected to various calibration procedures. The device that is known to function according to the required standards is called the "standard". The device being calibrated to match the standard is usually called the "test unit" [54]. What the calibration procedure basically does is to perform a certain transformation on the output of the test unit to match the standard. This can be as simple as changing a bias value for a measuring scale or recomputing the focal distance for a camera. This sort of test unit to standard mapping is what is required for the problem at hand.

This chapter investigates the calibration procedure required to match the inputs of one robot to the standard of the controller constructed for a different robot. This is required to allow for controller transferability between robots. The method deployed allows the robot to "envision" itself as another robot with a known functioning controller by finding the equivalence of its own sensors in the other robot's sensor space.

4.2 Related work

Not much literature exists on the sort of sensorial mapping attempted in this thesis. However, extensive work has been done on other calibration procedures.

In [55] the authors present a planar pattern to a camera from a few different orientations to perform calibration. Either the camera or the pattern can be freely moved and the data collected is used to extract metric information from 2D images to be used for 3D computer vision.

The authors of [56] derived formal methods to perform camera calibration for 3D vision and also investigated to some degree the number of calibration points required.

Robot calibration has also been performed [57]. Here, several methods have been investigated to refine the inverse kinematics accuracy of industrial robot arms.

Robot sensors in particular have also been calibrated in [58]. The authors used Lie theoretic methods to determine the precise location of the sensor after performing a motion task.

Robot vision sensors have also been calibrated [59]. Here the authors present several extrinsic calibration methods based on closed form algebraic methods.

None of the above robotics and sensor related works perform similar tasks to the ones attempted in this thesis. It is suspected that this has not really been attempted before. Data mapping work, however, can be seen in autoencoders [60]. Here, however, the task is to map the data onto itself through a bottleneck to force compression through feature extraction. What is desired in this thesis is mapping data from one modality to another as faithfully as possible. However, the basic concepts deployed in autoencoders serve the task at hand well.

4.3 Methods

The task at hand aims to convert the processed input data of a robot (in this case the Nao robot) to a representation compatible with the input for the controllers previously discussed in chapter 3. For this task the Pioneer robot was used for the standard and an Aldebaran Nao robot for the testing unit. Since the controllers were designed to be used with the Pioneer robot configuration, the Nao sensorial input needs to be translated into a Pioneer format input so that the controllers give out the similar responses given similar situations.

Due to the very different morphologies of the two robots, they can not be in exactly the same state, either physical, sensorial or logistic. For example, a logistic state for the Nao robot would be the act of grabbing a cup or a can with its arms, a feat physically impossible for the Pioneer robot that does not posses arms or grippers. The two robots cannot be in a logistical state describing the action of grabbing. Similarly the Nao has sensors that feed information not available to the Pioneer robot. Such an example can be made from the sonars on the Pioneer that are more complex and numerous than those on the Nao, providing a 360 degree sense of the world as opposed to the two frontal sonars that the Nao has.

For the current task, however, we shall restrict the experiment to just the visual stimulus also used in chapter 3. Namely the processed sensor abstractions of viewed objects, in this case a red and green

blob. This can be performed on any machine equipped with a video camera and both the Pioneer and the Nao have such sensors although some differences do exist. The Pioneer camera has a wider field of view, but is immobile, while the Nao camera has a narrower field of view but can change pitch and yaw with the movement of the robot's head. Also, as with most cameras, colors differ and automatic white balance and exposure correction alters colors and lighting in different ways on both cameras. The lens distortion also varies to some degree. Some of these features can be controlled while others cannot. As such, sensor acquisition software needs to be customized, for both robots and even after extensive work, results can still fail to be 100% accurate. However, a working approximation can be obtained, and irregularities between feature extractors can be later solved, to some extend by machine learning.

4.3.1 Proposed approach

The Pioneer sensor outputs are the same as those presented in chapter 3, namely the object positions on the x and y axis, the size and the object, a visibility value and the time since the last observation occurred (x-pos, y-pos, size, time, visible). This is done for both objects, the red follow target and the green avoid obstacle.

The Nao sensor outputs are similar in scope: The objects' positions on the x and y axis, their size and time elapsed since last observed. However, one thing is different. The Nao can move its head to observe more of the surrounding world. Thus two extra variables were added to compensate for the lower field of view of the robot: the yaw and pitch of the robot's head. This gives a sensor output vector seven values: (x-pos, y-pos, size, yaw, pitch, time, visible).

While the Nao sensor abstraction is likely to convey much of the same information as the pioneer sensor abstraction, it is incompatible with the controller designed for the Pioneer which takes the Pioneer abstraction format as input. Thus, it is important that a suitable transformation is found between the Nao view and the controller input requirements. This thesis proposes the expansion of the feedforward neural network paradigm presented in section 2.3 to construct an appropriate "adapter" from the Nao sensor abstraction to the controller. This is performed by using the knowledge of equivalence between Nao and Pioneer states given the required tasks. For this problem, a Nao looking forward and seeing its target in the middle of the screen and slightly up (The Nao is shorter than the Pioneer camera height) is the same as the Nao looking to the right and seeing its target on the left side and up of the screen and the same as the Pioneer looking forward (camera is immobile) and seeing the target in the exact middle of the screen. Thus it is relatively straightforward to use a neural network to construct a decoder from what the Nao sees to what the Pioneer would see under the same circumstances. After training, the results of the decoder are used to feed the data into the robot controller and generate appropriate movement vectors as described in chapter 3.

4.3.2 Experimental setup

To collect data for training, the robots were placed in 13 positions in front of the respective objects and data was collected for a training set. Twelve of them arranged in a grid roughly 30cm apart from each other and one in a position where the robot could not observe anything. Then the robots were placed in another 9 positions and data was collected for a testing set. A schematic of this can

be seen in figure 4.1. This was repeated for both objects used in chapter 3. We investigate whether the different nature of the two objects (one is a red flat notebook, the other is a green round ball) has a significant influence on the final output, despite coming from similar stimuli and using similar abstraction methods. Sample observations can be seen in figure 4.2.

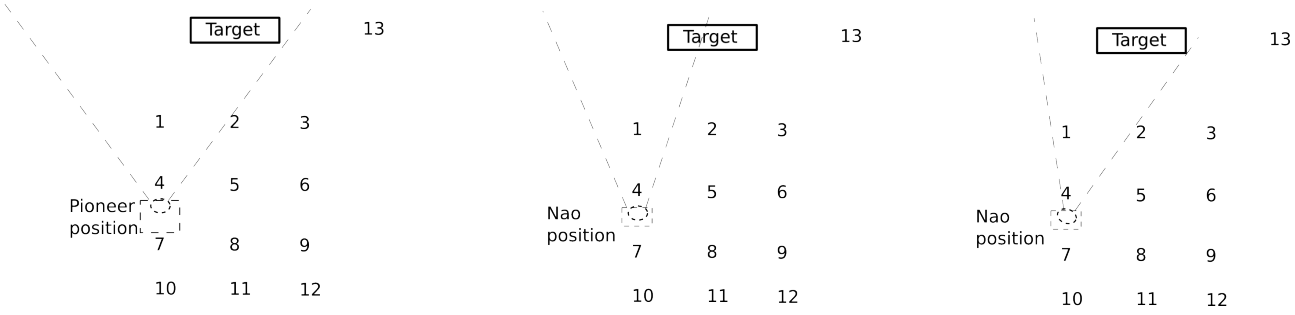


Figure 4.1: Equivalent states for the Pioneer robot (left) and Nao robot (center and right) turning its head to look in two different directions. Equivalent states are states that yield the same action output. In this picture the robots are in position number 4, with number 13 being visible in the image depicting a state in which the robots never see anything. Notice that the Nao robot has a smaller field of view but can turn its head.

The Pioneer data collection involved placing the Pioneer in the required positions and recording 10 observations for each position, within 1 second intervals of each other. While no variables were actively changed during the recording, there was some noise in the observations that was deemed an important factor for the training process.

The Nao data collection involved placing the Nao in the required positions and move its head in successive iterations of 2.9 degrees each from 29 degrees looking up to 23 degrees looking down (pitch) and from 52 degrees looking left to 52 degrees looking right (yaw) for a total of 720 observations for one position. This provides an extensive local observation of the surrounding world for the Nao robot. This is to some extent larger than the Pioneer observation because a Nao robot can turn its head and observe places that the Pioneer robot cannot despite its larger native field of view.

For the purpose of performing the mapping, feedforward neural networks (see section 2.3 for details) were tested with 5, 7 and 10 hidden units. Training was conducted on an Intel core i7 for 1000 epochs each with a learning rate of 0.1, a momentum of 0.8 and a learning rate decay of 0.99 per epoch. There were a total of 9320 data points for training and 6480 for testing purposes. Mapping from the Nao's observations of both its red and green targets was conducted onto the Pioneer's observations of the red and green targets respectively (acquisition described in more detail in section 3.3.2). The networks with the best testing error were saved and their performance noted.

Afterwards, they were applied to the controllers trained in chapter 3. The robots were placed in the 13 positions and the outputs were recorded for both the original and best performing learned controller. We repeat this both for strapping each observation adapter to its own source (red blob observation to red blob observation, and green blob observation to green blob observation) as well as using just one adapter for both objects. In particular using the one derived from training on just the red blob and using the network to map both observations between robots. This gives an indication of the overall effects of stacking several neural networks on top of each other. In the situation of the Nao, it is

(a) Pioneer with locked view (b) Nao looking up and to the left (c) Nao looking straight and to the right

Figure 4.2: Samples of Pioneer and Nao robot views of the two color targets (red notebook and green ball) simultaneously. Note the differences in color and the effects of lens distortion. This leads to complications in extracting accurate world observations.

programmed to turn its head a tenth of the way towards the center of the observation from where it is currently looking every time a new observation comes (approx every 100ms).

4.4 Results

Performance can be seen in table 4.1. Through several training attempts, results were quite consistent. While smaller networks with 5 hidden neurons work better on the red target of a flat surface, the green, round target's dynamics can more easily be learned by more complex networks of 10 hidden neurons. It is suspected that illuminations of a sphere cause the detected surface to vary more in size with different angles of observation. However, it also is more resistant to lens distortions as opposed to a flat observable surface. This indicates that observations are not the same, although it is probably not a significant difference.

	FFNN-5h	FFNN-7h	FFNN-10h
Red target	**0.0074**	0.0090	0.0087
Green target	0.0093	0.0096	**0.0077**

Table 4.1: Average testing error rates(MSE) for the different targets. The robot was placed in front of two different targets in the positions seen in figure 4.1, and the observations were recorded. The red target is a flat notebook, while the green target is a round ball. It is suspected that the differences between detectors as well as targets lead to different detection dynamics and thus different degrees of complexity required to obtain good approximations.

Of greater importance for the overall goal is that the output of the controllers is the same. Regardless of how good the mapping is, it is worthless if the robot fails to follow proper action. The errors from strapping the converters onto the controllers and comparing their outputs with the original output from the pioneer controllers can be found in table 4.2.

As can be seen, all errors are at least one order of magnitude higher than those obtained from just learning the controller and applying it to the Pioneer robot directly. However, observation showed

	Learned controller on Pioneer	Nao with correct converter and original controller	Nao with correct converter and learned controller	Nao with wrong converter and original controller	Nao with wrong converter and learned controller
Error	0.0017	0.0449	0.0657	0.0497	0.0535
STD	0.0038	0.1439	0.2076	0.1518	0.1648

Table 4.2: Mean output errors using the best controller for the approach behaviour obtained in chapter 3. This uses the red target as input, thus a red target converter was constructed. The wrong converter implies using the green target converter to actually convert the input of the red target.

that while the scale of the output differs, the direction of the vector remains consistent. There seems to be a lot of variability in controller output values, often $\pm 20\%$ of the mean value observed (at least for the turning component). This appears to be due to observation noise amplifying with the errors added in the transformation. For this controller, the converter does not seem to be as important. Using the wrong converter appears to not increase error significantly and when also using a learned controller it seems to even decrease the overall error. Due to the large errors adding up this does indicate statistical significance. However, for the avoid controller, the errors of which can be seen in table 4.3, things are different.

	Learned controller on Pioneer	Nao with correct converter and original controller	Nao with correct converter and learned controller	Nao with wrong converter and original controller	Nao with wrong converter and learned controller
Error	0.0722	0.0478	0.1692	0.2697	0.5319
STD	0.1142	0.0916	0.2153	0.2252	0.5316

Table 4.3: Mean output errors using the best controller for the avoid behaviour obtained in chapter 3. This uses the green target as input, thus a green target converter was constructed. The wrong converter implies using the red target converter to actually convert the input of the green target.

Errors in this case seem to be 4 to 5 times bigger if the wrong converter is used. Because this controller is more sensitive to the size and position of the respective object (see figure 3.3) than the previous one, this makes an accurate converter, tuned to the object in focus, preferable. This is despite the fact that both objects stem from the same sensory source and use similar data extraction techniques.

What is even more interesting, however, is the fact that there is high variability even with regard to mapping from the Nao to the Pioneer while in equivalent states, depending on the direction the Nao looks in. This contributes to the error in the final stages of the vectors. Figure 4.3 is a plot of the error varying with the yaw and pitch of the Nao's head for the robot placed in position 4 (in front of the target, with body facing it).

As can be seen, the error when the robot fails to see its target, looks relatively smooth, although it usually is one or two orders of magnitude higher than the area where observation actually takes place. There is a region where error spikes even higher when the robot is looking down and to the left. It is suspected that the network structure used lacked the representational power to accurately map the entire area, despite it being in the training set. Collecting more data points might mitigate the process. Error is also non-uniform even in the region where observation does take place, although the error difference here is relatively low. If the robot does see the target, regardless of where the target is on its image, its decision of action is sufficiently accurate. If however it does not see the target, then

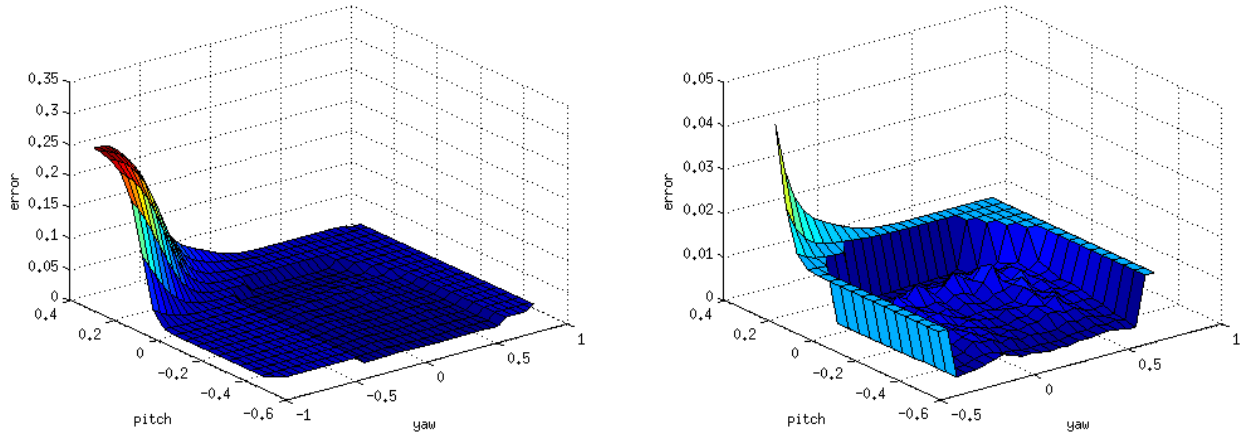


Figure 4.3: Controller output error landscape, for the learned approach controller, using a red target and a Feedforward Network with 5 hidden units, for the entire Nao head movement range (left) and a region zoomed into the area where observation actually takes place (right). Negative yaw values mean head turned right and negative pitch means head looking up. Note that the target is slightly higher than the Nao's head. This exact sample as taken with the Nao placed in position 5 (figure 4.1) The 'peak' in the upper left hand side is likely an artefact of the particular learning run as a consequence of the representational power of just 5 hidden units. The error on the outskirts, when the robot sees nothing is 0.0087 (excluding the peak), while the minimum error in the visible area is 0.00029.

it has difficulties knowing where it is thus what action to take. This is understandable as conditions are ambiguous in this instance. Not seeing the target while being in a certain state overlaps with all other states.

4.5 Discussion

In this chapter a method for mapping different sensor abstractions onto the format required by the controllers was demonstrated. Also a simple calibration procedure for constructing such converters was demonstrated. Performance was shown also in conjuncture with the controllers developed in chapter 3 in light of the ultimate goal of migrating full systems.

While not optimal, the method works sufficiently well, drawing equivalences between Nao and Pioneer states to be usable with the controllers. Feedforward networks have proven to work well for the task. Using one network per output was also attempted in preliminary testing, but performance was below that of the ones presented above, due to overfitting, so they were not expanded on.

In theory, data extracted from other modalities can also be used, as long as it conveys the same information. As could be seen, in the case where the Nao could see its target, error was low, as information received was in accordance to the one received from the Pioneer. Once the robot could no longer see the target, error went up. However this cannot be helped short of reducing a robot's field of exploration. Granted, with proper thought, more complex calibration strategies can also be devised, but that infringes on the universality of the approach.

Another task that raises questions is reverse mapping. Mapping from the lower dimensionality of the Pioneer sensory perception to the higher dimensional Nao sensors. This task would be more complicated but not impossible given sufficiently accurate data. However whether some tasks are indeed feasible to even be attempted is left up to the judgement of the one trying to implement the task. From Nao vision (including the head angles) to Pioneer vision some information compression is performed, and in order to perform the reverse, that information would have to be recreated. While to some extent possible with methods like Markov random fields [61] to generate possible instances of your data equivalents in order to match your format, if the controller relies on information that cannot be extracted from the actual world then it is an overall futile attempt. Granted one can try to measure the error and if the error exceeds a certain threshold declare that the mapping has failed and that sensory equivalence unattainable.

In this scenario the controllers were specifically designed for the Pioneer, so the Pioneer does not need any adapter to fit into the behaviour. It is the baseline. However, should another robot appear, and a Pioneer would not be available to provide the benchmark sensory input, another approximation would be to map the new robot's sensory input onto the Nao's sensory input and then use the already existing adapter to insert data into the controller. But one should always keep in mind the end goal of the adapter, though, is to provide proper input to the controller. If not possible, this may lead to some errors. These errors still have a possibility to be corrected in the end by finding a correspondence between the erroneous controller output and the actual robot API motor commands.

Chapter 5

Learning the output equivalence

5.1 Introduction

Humans, as well as several types of animals can mimic each other's actions. They observe another one of their species performing an action and, sometimes with some training, can reproduce it. This is generally rather simple as an action equivalence can be drawn between the two subjects, the demonstrator and the reproducer, based on knowledge of body equivalence. However, what about performing the same action with different species? A dog fetching a ball is a good example of that. A human can use his or her hand to pick up the ball and transport it from one place to another. However, dogs do not have hands that they can possibly grasp objects with. They pick up the ball in their mouth and relocate it from point A to point B. Within the scope of "ball fetching" that action is roughly equivalent. It changed the world from state A to state B regardless of the exact mechanism that enacted the change.

The problem is similar in the current setting. Equivalent actions must be found between the two different robots, or in this case, since there is no logistic difference between the controller output and the API requirements, a method to map from the controller output to the new robot's API control vector must be found. However, unlike in the previous experiments, it is extremely difficult and impractical to construct a dataset by hand to conduct supervised learning from. Rather, we desire that this dataset be constructed automatically from the robot's own interaction with the environment. Two approaches have been tested in this thesis: The first involves learning models of the two robots' dynamics and then using these models to learn offline the actions that perform the same state transitions. The second skips the model learning stage and learns associations directly from a nearest neighbour [62] approach to detect similar state transitions and thus associate similar actions.

In this section we learn the equivalence between outputs, more specifically between the output of the controller designed for the Pioneer and the Nao's control vector for the API. In order to achieve this we build on top of the results from chapters 3 and 4. We use the best learned controller previously presented, namely the multiple feedforward neural network with 10 hidden units each, and the best input to input converter for the red target and the green obstacle, namely the feedforward neural network with 5 hidden units and the feedforward neural network with 10 hidden units respectively. Training is only conducted once, using the red target as a reference, but the resulting output correspondence is

used for both actions coming from the red target as well as the green obstacle. Actions are considered equivalent regardless of what triggered them. In the end, the performance of the final meta controller developed in chapter 3 but applied through the architecture developed throughout this thesis onto the Nao robot instead of the Pioneer is evaluated and compared to its Pioneer counterpart. The meta controller consists of the overlapping of the learned approach and avoid controllers developed previously.

5.2 Related work

Work directly dealing with the current task is scarce. Most action learning in literature is conducted using reinforcement learning [63, 64]. This works by assigning rewards to state transitions and reinforcing actions that lead to desired state transitions while penalizing actions that lead to unwanted state transitions. The main challenge involves balancing exploration and exploitation of discovered policies.

In [65] a comprehensive overview of system identification techniques is given. The authors show that black box modelling can be used to describe even nonlinear dynamical systems. However, the systems are purely mathematical.

The authors of [66] demonstrate how neural networks as black box models can be used effectively for the identification and control of nonlinear dynamical systems. The systems they analyse, however are pure mathematical systems not directly applied to real world problems.

In [67] system identification was performed on a robotic arm using mathematical models. The authors use least square estimates to determine the parameters for the first degree polynomial (thus linear) state-space equations.

A more assumption free modelling attempt was made in [68] where the authors use Support Vector Regression [69] to describe the behaviour of a hydraulic robotic arm and improve on previous methods.

No work has been found on modelling robot sensor input dynamics from actions, or learning actions from such models. We do not understand why, as it is this author's consideration that internal model building and offline search based on those models can be a powerful tool to generate novel robotic actions.

5.3 Methods

Building on the results from chapters 3 and 4, we now aim to complete the architecture described in chapter 2 by developing a method to learn its last component: the mapping of abstract controller outputs to actual physical actions. Since this is far from a trivial task, we have attempted two possible methods to attain the feat, which are discussed bellow.

5.3.1 Model based learning

A neural network based model of the robot's dynamics is constructed. This model takes as input both the sensorial input (the observed values of the detected target) as well as the output resulting from the controller's decisions or API commands to the robot and predicts the future sensorial input at the next timestep. Note that the use of the word "timestep" here is more out of literary convenience, in real time systems it is impossible to divide computations into **exact** quanta, but usually approximations are arbitrarily close enough to the desired division. Such models would learn to represent the evolution of a robot's senses with regards to its actions. Care must be taken, however, as the effects of noise, overfitting and underfitting, as well as insufficient or inconsistent data can lead to poor models and poor representations.

The reasoning behind learning models, is that, given a good enough model, if the inputs describing the states are the same (which should be the case, considering the results from chapter 4) then simple search algorithms can iterate through possible actions in the model to describe the same or otherwise similar state transitions given by a known set of actions coming from the original robot. All of this can be done offline following the creation of the model, making use of the larger computing power available off the robot and without the need for real-time processing of data required by, for example, reinforcement learning approaches.

Models can be constructed out of data. Time series prediction methods [70, 71, 72] perform similar tasks. Data gets collected from observations of the state-action space and models get constructed out of the collected data that represent the respective dynamics. Due to the success of feedforward Neural Networks in previous sections and the reactive nature of the controllers implemented, this has been made the method of choice to model the robot dynamics as well.

The method presented above can be structured in the following fashion (see figure 5.1):

1. Let both robots perform actions randomly (within reasonable limits, such as avoiding crashing into objects), ensuring observation of the target by the human supervisor, and collect observation states and the actions given.
2. Use Neural Networks to learn to predict the next observation given the values of the current observation and actions for the second robot (the one the controller is being transferred to).
3. Initialize a randomly generated Neural Network that maps from one robot's output actions to the other.
4. Use generated actions and the first robot's observations to compute a predicted future observation through the second robot's model.
5. Compare the output of the prediction to the new actual state. Compute the errors from the outputs (predictions) to the inputs of the prediction model denoting the actions. These inputs are in fact the outputs of the neural network performing the action converter.
6. With knowledge of the errors added by the actions, we can now backpropagate them through the converter network and perform the action to action learning. (Note, that the output of the controller from chapter 3 in the original contains the same information as the actual control vector provided to the API. Therefore these can also be used as actions to map from thus skipping an extra layer of non essential mapping).

7. Repeat from step 5 until predictions consistently match the dataset.

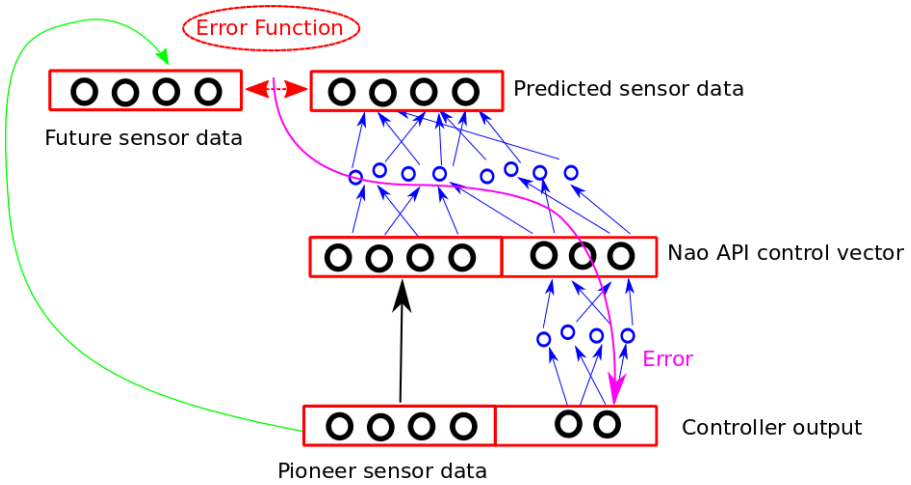


Figure 5.1: Pioneer observations and actions mapped through the neural network are fed through the Nao model and predict future observations. The error function (seen here in purple) from these observations then propagates back through the networks and is used to learn the controller output - control vector mapping.

5.3.2 Model free learning

Since prediction is a complicated affair, a benchmark solution has also been tested. Because the final aim of this section is to map from actions on one robot to actions on the second robot, all we really need is to collect these actions and correlate them both with the similar state transitions. To that end, because the observation space is continuous, a degree of similarity must be computed between the state transitions on both robots.

The method can roughly be described in the following manner:

1. Let both robots perform actions randomly (within reasonable limits, such as avoiding crashing into objects), ensuring observation of the target by the human supervisor, and collect observation states and the actions taken.

2. Iterate through the observations of the first robot and compute the distance between two consecutive observations to all pairs of consecutive observations in the second robot's list.

3. If the minimum distance between one set of consecutive observations from the first robot and another set of consecutive observations from the second robot is lower than a certain threshold then retain the actions associated with the state transitions on both robots and add them to a dataset.

4. Perform supervised learning using a simple feedforward neural network to learn mapping of action values from one robot to another. Note that these do not have to be the exact API command values, and can also be the values of the controller output in the case of the standard

robot (the one who's controller we are migrating), but not in the case of the test robot (the one to which the controller is being transferred).

5.3.3 Experimental setup

Data collection

For these experiments, the same red target and red blob detector was used as in chapters 3 and 4. Both The Nao and the Pioneer were placed in 4 locations in a straight line in front of the target at a distance 0.3, 0.8, 1.3 and 1.8 meters from it. The robots were then given random commands by adding Gaussian noise of mean 0 and unit variance to the controller output for the Pioneer (the movement and turning force normally outputted by the controller) and to the API control vectors for the Nao (these consist of a vector denoting forward movement, one denoting a strafing movement and one denoting a turning action). The exploration procedure works as follows:

1. Generate a Gaussian noise vector. Use the vector as a control vector to command the robot. For the Pioneer, the vector consists of controller output values that then get transformed into API values by the original conversion system. For the Nao, the control vector is generated directly for the API. Do so for 4 seconds continuously.
2. Stop the robot and wait for another 4 seconds. This is done in order to shake off any inertia that can risk damaging the motors.
3. Apply the opposite of the control vector above to return *roughly* to the starting position.
4. Stop the robot again and wait for another 4 seconds.
5. Repeat from step 1. Gather observations every 0.1 seconds. Stop when enough observations have been gathered or various errors accumulated until the robot no longer sees its target.

Note that applying the opposite control vector does not always return you to the original spot, this is highly dependent on robot architecture, API and situation. For example, if you drive forward and left in a car, to return to your original spot you have to drive backwards and left, not backwards and right. Luckily for this scenario, the two robots' dynamics allows for control vector reversal. However, for various reasons, network delay, non-even computation and physical deformities of the robots and/or their environment, noise accumulates and the robots no longer return to their original position. Once the robot deviates sufficiently that the data collection is no longer deemed useful, collection is stopped and the robot is returned to its initial position or is moved onto the next position.

Unlike the Pioneer whose camera is stationary, the Nao has a movable camera, To take advantage of this feature without also adding unnecessary complexity to the model the Nao head is made to move to look towards its target of interest at every new observation. To this end, 6203 observation-action pairs have been collected for the Pioneer and 5770 state-action pairs for the Nao robot. The observations consist of a vector made of the position on the x and y axis of the target, as well as its size and whether it is visible or not. This is in line with the format that the controller designed in chapter 3 uses. For the Nao, the converter designed in chapter 4 is used to convert the Nao input vector to the one required by the controller.

Model based learning

For this experiment feedforward neural networks were trained to learn the prediction function (the model) for the Nao robot. Networks with one hidden layer and 5, 10, 20, 50, 100 and 200 hidden units were tested and the results are reported in the following section. Training data was arranged in pseudo-random order and consisted of 60% of the dataset, while 40% was used for testing. Stochastic gradient descent was then performed to train the network, using a learning rate of 0.01 for 1000 epochs each. Inputs consisted of the x and y position of the target and its size and visibility flag, and the three control vectors the Nao robot uses: the forward, strafing and turning speed. Outputs consist of the x and y positions and target size and visibility variable in the next time step 0.1 seconds ahead in time.

We then initialize a random neural network that maps from the controller output vectors recorded from the Pioneer to the Nao API output vectors. We iterate through the Pioneer data set and feed it through the Nao model using the above neural network. The outputs of the prediction are then compared to the actual future inputs from the Pioneer dataset. The error function is propagated back to the action converter neural network where it is updated through gradient descent. A schema of this is presented in figure 5.1.

Neural networks with 2, 5 and 7 hidden units are tested for the mapping. This action converter is then run on the robot and performance is measured and presented in section 5.4.

Model free learning

The data sets are reconstructed for this task. This is done in two manners:

1. Sequential observations are grouped together to form a state transition. Thus two observations now make up a new data set containing the observations at time t and at time t+1 for both robots. A simple nearest neighbour algorithm is used to test for differences between transitions. In this case the distance function between states is an euclidean metric.

2. The new dataset consists of the difference between two consecutive states, representing the relative transition itself. In this scenario the distance between states is represented as the mean pair-wise product of the robots' state transitions. This defines similarity between state transitions not as much by the states themselves but as the correlation between the relative shift in observation values for each individual component. This is ensured by performing the product of the component vectors. Similar changes will be strongly and positively correlated (product will be positive and large) while dissimilar changes will be either close to null or negatively correlated. Computing the mean of such a value ensures an unitary representation of all composing vectors. The highest correlated pairs of observation changes will be given by the maximum value. The higher this correlation is the more similar the actions are assumed to be as well.

This results in a list of associations between state transitions, and thus robot actions. Since these associations can be based also on relatively large computed distances, only the points with a distance lower than the mean distance between states is retained as a training set for the next stage.

The values of the correlated robot actions are then used to train an action converter from one robot's actions to another's. Or in this case, since the pioneer recordings are based on the controller output, from the controller output to the Nao API vector requirements. Neural networks with 1 hidden layer and 2, 5 and 7 neurons in the hidden layer are tested and performance is indicated bellow. For this task batch gradient descent was used for back propagation on a set of 4474 points.

The best performing neural network is then deployed on the Nao robot, together with the entire system completed from chapters 3 and 4 and the robot's performance is measured over 23 runs. We measure the success rate of the robot performing the stacked behaviours presented in chapter 3. This can be seen in section 5.4. The setup for the experiment is the same as the one in figure 3.2.

5.4 Results

5.4.1 Model based learning

Feedforward neural networks with 5, 10, 20, 50, 100 and 200 hidden units were trained. Mean squared testing error is reported in table 5.1.

FFNN-5h	FFNN-10h	**FFNN-20h**	FFNN-50h	FFNN-100h	FFNN-200h
0.000439	0.000452	**0.000394**	0.000415	0.000406	0.000423

Table 5.1: Testing mean square error of successful policies. The feedforward network with 10 units in the hidden layer had the smallest unrounded error.

Errors are low, with the feedforward neural network with 20 hidden units holding the lowest error. This network represents the prediction model of the robot's dynamics and is then used to train the action mapping. However, simple tests indicate that low error does not imply good prediction. By defining a starting observation with the target in the center of the image on the x axis, a little above the center on the y axis and a size roughly of 0.1, we tell the robot's model to move forward and backward. Expectation is that the observation's size would grow as the robot approaches the target and decrease as the robot gets away from the target. This does not appear to be the case, as can be observed in figure 5.2.

The model fails to properly predict future observations. This is especially visible for longer numbers of time steps. Parts of the observation vectors such as the target size vary widely with commands that should give, for example, a steady increase. A number of factors have been identified for this cause:

1. Errors from the Nao to Pioneer observation conversion seem to play a big role in deforming the dataset, even if they are small. When the Nao performs no locomotion action, thus its control vectors are 0, the head still moves to face its target. As a consequence, observation changes as a function of noise, which adds noise to the overall prediction. The phenomenon also takes place as the robot is moving but is less prevalent as at least there is some variability in motor commands to correlate with observation changes.

2. The network is trained on a certain domain of observation vector values. As such, good generalization can only be guaranteed within that domain (Note that prediction is more a regression

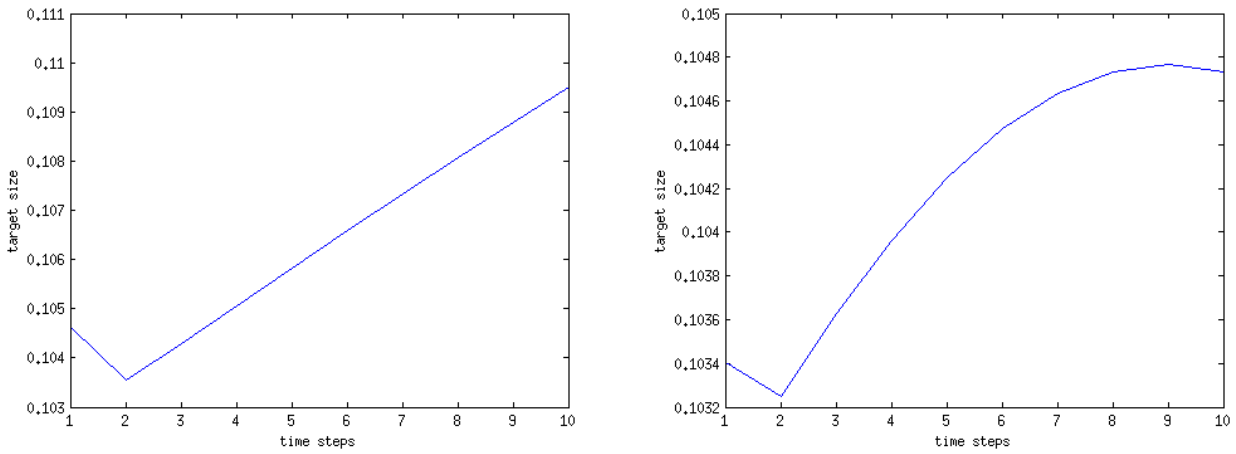


Figure 5.2: Target size on screen for moving forward (left) and moving backward (right) for a Nao robot, as provided by the neural network model. This does not correspond with how the model should be working.

and not a classification problem). Once values start exiting that domain, then results are out of the model's control.

3. Data collected from the robots is also very noisy. The cameras are not the best of qualities and observation vectors at times vary to quite a considerable degree, even in the absence of action. In the absence of action the errors are roughly evenly distributed around a base value, but in motion, as the values change in a more dynamic fashion, these can lead to complication.

The three factors above both contribute to the failure of the model. Noise from both the initial observation as well as the observation converter, drive the model outside its domain of definition and thus predictions become unreliable. Attempts to limit the phenomenon have failed. It is assumed that more data may help with some parts of the noise spectrum but the domain of definition may not be extended.

Attempts to learn neural networks using the above described method, with any of the models above have lead to non performing converters and, as a consequence a non performing robot. The method has ultimately been abandoned.

5.4.2 Model free learning

Euclidean metric

The first approach using an euclidean metric and groups of two states yielded an average distance between data points of $8.849 * 10^{-4}$ while the standard deviation was 0.026. This leaves the similarity to vary rather significantly. A plot of the minimum distance between observations for every data point pair in the Nao dataset to the Pioneer points can be seen in figure 5.3.

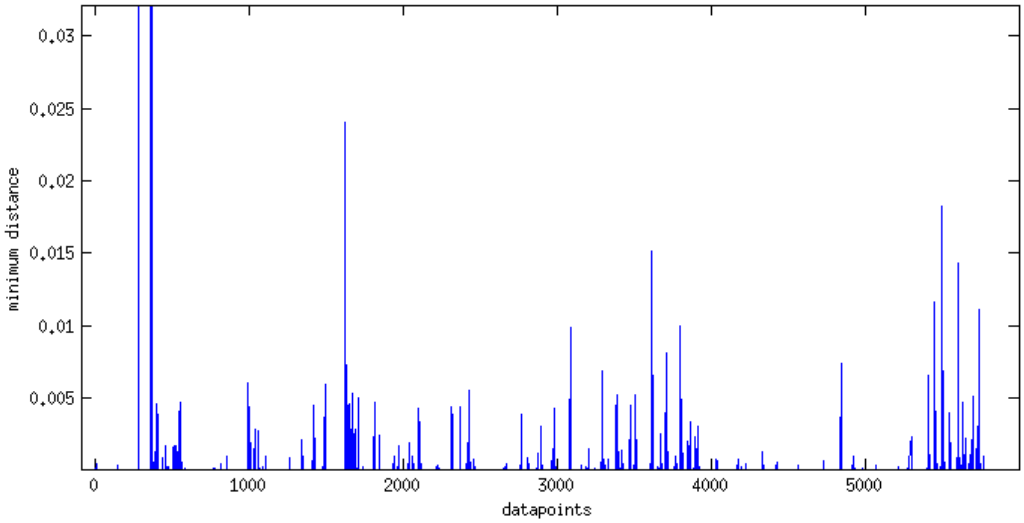


Figure 5.3: The minimum distance between data point pairs and their counterparts in the Pioneer dataset. As can be observed, some distances (and thus error) are orders of magnitude larger than others.

Since the nearest similar state transition in some cases is rather far away, it makes sense that we discard these data points from the neural network action mapping procedure. Thus, only the ones lower than the mean of the dataset are counted towards training in the next step.

Neural networks with 2, 5 and 7 hidden units are trained on the actions associated to the state transitions closest to each other. The testing error can be observed in table 5.2.

FFNN-2h	FFNN-5h	**FFNN-7h**
0.1417	0.1412	**0.1404**

Table 5.2: Testing mean square error of successful policies. The feedforward network with 7 units in the hidden layer had the smallest error.

However, when deployed on the robot, neither of them manage to move the robot. All the converter values are small, close to zero. The robot merely walks in place. The network with 5 hidden units manages to get the robot to turn towards the target in the approach behaviour (section 3.3.2). However, forward locomotion is hindered. This appears to result mainly from the nonlinear progression of the target size value during data collection. The linear and comparatively larger variations of the x axis component make turning easier.

Product based metric

The second approach measured the pair-wise product between transitions of observation (pair-wise difference of consecutive observation values) values for the Nao and Pioneer robot. A plot of the maximum product values for all observation transitions for the Nao robot to the Pioneer robot observation

transitions can be seen in table 5.4. We use the product, which is similar to an actual correlation metric, but takes into account that it is better to match longer action vectors.

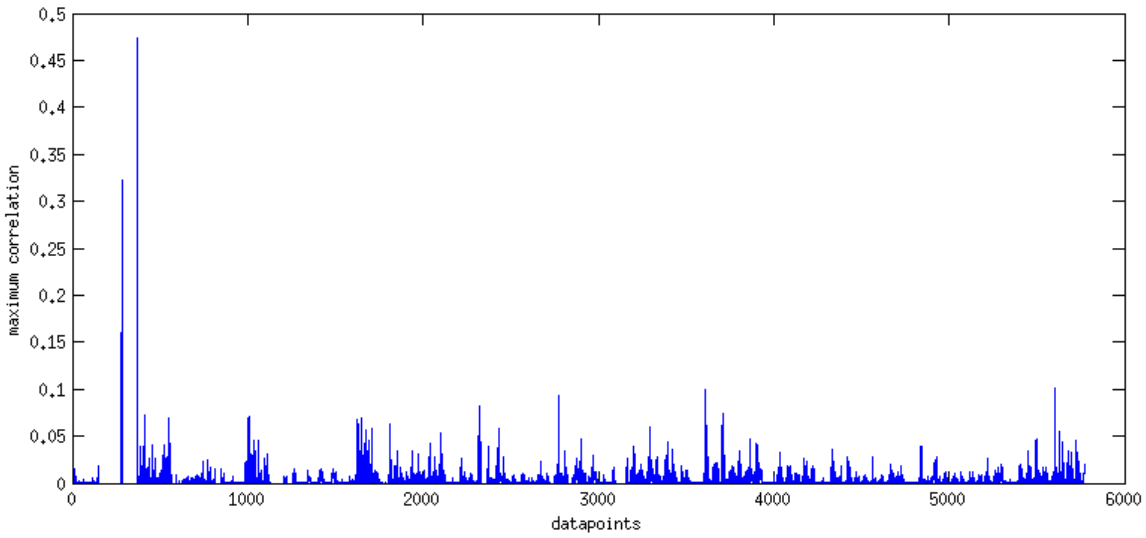


Figure 5.4: The maximum product between observation transitions of the Nao and their Pioneer counterparts. Some observation transitions do not strongly correlate with anything and so it is better to not use those points for training.

Only those products larger than the mean (those strongly correlated), a total of 4473 action pairs were then used for learning the action mapping. Neural networks with 2, 5 and 7 hidden units are trained on these action pairs with the controller outputs as inputs and the API control vectors for the Nao robot as the outputs. The testing error can be observed in table 5.3.

FFNN-2h	**FFNN-5h**	FFNN-7h
0.2615	**0.2600**	0.2602

Table 5.3: Testing mean square error of successful policies. The feedforward network with 5 units in the hidden layer had the smallest unrounded error.

The network with 5 hidden units had the lowest testing error after 1000 epochs of training. However it is not significantly better than the other networks tested.

The successful network was then deployed on the Nao robot in addition to the other subcomponents trained in previous chapters to reproduce the same target/obstacle navigation task the Pioneer robot performed in chapter 3. The final results can be observed in table 5.4.

Of the 23 attempts, 5 failed. Mainly failures stemmed from too powerful turning triggers that forced the robot to turn for 180 degrees while walking to avoid the obstacle. This sometimes leads to the robot turning to not finish going around the obstacle before it's 'seen' timer expires and it no longer remembers the target, at which point it stops.

Pioneer defined controller	Pioneer learned controller	**Nao learned controller**
81%	87%	**78%**

Table 5.4: Success rates of the combination of the approach and avoid behaviours. Success is counted when the robot reaches its target without having crashed into its obstacle or veered off course and has no chance of recovery.

5.5 Discussion

In this chapter we presented a way to construct the final converter for the robot controller migration scheme presented in chapter 2. Several methods have been investigated and tested, and the complete system was in the end evaluated and its performance compared to the performance of the original controllers on the initial robot. While the error is slightly lower, the number of runs is not sufficient to draw upon any statistical difference, as results are relatively dependent also on starting position and environmental conditions that cannot be controlled such as poor lighting leading to poorer observations which lead in the end to a somewhat different behaviour.

In addition, due to the different nature of the robot, the final behaviour cannot really be identical but can only resemble it to a certain degree. Success rate is only a gross approximation of controller performance, as there are other less measurable factors that an observer can realize. For example, the Nao robot predominantly went around the obstacle on the right side, while the Pioneer showed no such preference it its demonstrations or in the learned behaviour. This is due, mainly to a defect in the Nao robot that causes it to naturally veer to the right.

The controller outputs map onto the Pioneer in a typical move forward and turn manner. The Nao also comes with strafing capabilities, which, to some extent, mimic the observational changes of turning. Strafing or turning to the left, leads to the observation migrating towards the right of the screen. It is likely that the turning effect also causes changes on the y axis due to lensing effect while strafing does not. The cause can also stem from the sheer fact that turning causes larger changes in observations while changes resulting from strafing are noisy and small. This was not an expected phenomenon in the original concept, as it was expected that turning on the Pioneer would translate to a mixture of turning and strafing on the Nao and that was not the case (Some strafing did occur as values representing the strafing speed were non zero but the effect was not significant).

While not perfect, the actions are equivalent enough, in the sense that on a macroscopic level, the macro end goal of reaching a target while avoiding an obstacle was reached, and as such, the system was proven successful. Performance can, afterwards be improved by global optimization methods looking to optimize the entire input-controller-action network structure.

Chapter 6

Conclusions and future work

6.1 Summary

In this thesis we tackled the task of transferring robot control systems between robots of different modalities. We explored the possibility of transferring two controllers, one performing the task of approaching a target and one performing the task of avoiding an obstacle, in combination, from an ActiveMedia Pioneer robot to an Aldebaran Nao robot. We also investigated possible methods by which this could be done in a perceptive, cognitive manner, without the rigorous world modelling based on expert knowledge that is widely deployed in robotics today.

To facilitate this, we proposed a series of abstraction layers as part of the robot control schema (figure 1.1), namely the raw data layer, the processed data layer, the decision layer, the interpretation layer and the action layer. We theorized that by dividing the control pipeline in such a manner, we would simplify the transfer process. Thus, we devised a series of simple calibration procedures to match observations and actions between modalities and deployed machine learning techniques based on neural networks to construct adapters that perform the transformations between the various modalities. We also used learning from demonstration to abstract away the controllers, whose hard if-then-else structure is susceptible to failure when dealing with approximations, into neural network representations which coupled with the adapters, present a full neural network based processing and control pipeline, that could later benefit from global optimization methods to optimize the whole control structure. Finally we proposed that equivalences between action layers can be determined by the effects those actions have on the inputs. Similar changes in inputs were theorized to result from similar actions.

A series of experiments were conducted to investigate the effectiveness of the various components proposed in the schema, and of the overall architecture. In chapter 3 we investigated various architectures of artificial neural networks to learn controllers from demonstration on the Pioneer robot. In chapter 4 we looked into the calibration procedures required to train a converter from the Nao input to the input required by the controller designed for and trained on the Pioneer. Finally, in chapter 5 we used one of the converters from the previous chapters on the Nao and tested various methods that use the change in inputs to learn equivalent actions. A final experiment compares the performance of the original Pioneer composed behaviour to the Nao's attempt to perform the same task using the architecture

through the input converter - learned controller - action converter processing pipeline.

6.2 Conclusions

6.2.1 Results

In chapter 3 we have shown that artificial neural networks can be used successfully to learn controllers. We have tested various neural network structures with feedforward neural networks, in this case, having had the best performance. Specifically, multiple neural networks, with one network per output. Echo state networks came at a close second, while recurrent neural networks learned irrelevant correlations and were not able to perform at all regardless of the size of the hidden layer. It is suspected that this is largely due to the reactive nature of the controllers demonstrated, the initial starting position as well as the irregular time intervals that observations are taken. We have shown this to work for both tasks that implied approaching a target, avoiding one as well as demonstrated a method to perform the superposition of tasks.

In chapter 4 we showed that equivalence between inputs of different modalities can be drawn if the information is compressed or retained. In our situation, we had the Nao's detection consisting of 6 values mapped onto the input for the controller designed for the Pioneer robot, which takes 4 values. The information required to perform the action was there, however, and thus a coherent mapping was possible. The reverse would have depended on whether the action a controller designed for the Nao would have required the extra information (such as what direction is the head turned) to make its decision. This is, however, still up to the operator in charge of calibration to decide. We have shown that there is no significant difference, regarding controller output, between a Pioneer in a particular position and a Nao using the converter and a learned controller. The result was not identical, and was also susceptible to a certain degree of noise, but as long as the robot was able to observe the target the controller outputs would be strikingly similar.

Finally in chapter 5 we demonstrated that we can use the changes in sensor space to learn equivalences in action space, namely from the output of the controller to the Nao API. Several methods had been tested, such as predicting future sensor inputs from current sensor inputs and actions taken, and later using exploration of the input-input space to determine equivalent actions, as well as learning action equivalence directly from the distance or correlation between input transitions. Results show that direct learning, especially from correlation performs better than first learning a model with neural networks and then learning from the model. It is suspected that more data might have increased performance, however it would have also increased data collection and training times. Finally, the input converter - learned controller - output converter structure, deployed on the Nao to measure the entire approach-avoid metacontroller performance, did so very well, having obtained success rates similar to the ones of the original controller on the Pioneer robot. Artefacts existed, such as the robot's tendency to always go around the obstacle on the right side, but these did not affect performance and are easily solvable with further optimization.

6.2.2 Research questions

In this thesis, we started off with the following research questions:

1. How can robot controllers be scaled, adapted and transferred between different robots?
2. How to define the different abstraction layers to fulfil the previous research question?
3. How to perform robot to robot controller migration with the least amount of modelling?

After several experiments we are finally able to provide a series of answers:

1. Controllers can in practice be scaled and grown by building smaller controller for smaller tasks and then combining them to form larger, more complex controllers. This is made easy by standardizing their outputs (all controllers output results in the same format), which allows the composition task to be as simple as adding out the results. For some tasks it is possible that a more effective method of constructing metacontrollers exists, however it can also be done in just such a simple manner.

   Controllers need little adaptation to the new robot. Rather it is the connecting components that require adaptation in order to function with both the controller as well as the robot's basic functions. We have shown that we are able to do this with neural networks. At best, the controller may benefit from "adaptation to the adapters", to allow for working with approximations (something the original controller may not natively support), and this we have shown to also be a task that neural networks can tackle.

   Once scaling and adaptation is completed, transferring is directly possible and the new control structure can be directly run on the new robot, provided the programming environment supports it.

2. In this thesis we have constructed our abstraction layers as follows: the raw data layer, the processed data layer, the decision layer, the interpretation layer and the action layer. This is described in figure 1.1. The overall success of the experiments conducted in this thesis have shown that this is a viable division of abstraction layers.

3. Modelling was one thing we have thought to avoid as much as possible. Any expert knowledge brought into the system is a hamper to its scalability. To that extent, we have shown that black box systems, such as neural networks, combined with the proper calibration methods can be a good way to avoid modelling. The only expert knowledge that has been brought into the system was the set up of the calibration procedure and the parameters for training the neural networks. The latter can, however, be automated. We have shown that we can construct the transfer architecture using mostly just data and machine learning.

6.3 Future work

The main issue is that controller migration in itself is an ill-posed problem, as there is no clear definition as to what exactly is an equivalent behaviour that one robot can perform and how accurate this needs to be. One can only assume that those wishing to perform this controller transfer may use common

sense to determine the required threshold for that specific migration task. All in all this work was exploratory into the field of robot independent artificial intelligence, which by the date of the current work had seen little investigation from the scientific community. The current work can serve as a basis for future research into the topic.

Future work may involve investigations into learning and migrating more complex controllers, as well as more complex methods of combining controllers. Other areas of research exist also in searching for better methods to learn action mappings, or investigating ways to use raw sensor data directly and extract relevant features for each controller. In addition, global optimization methods can be used to further refine the results of the proposed architecture after its full assembly. Current results can also be improved, for example by using larger datasets, testing more parameters for training, or investigating automated methods for parameter search. However, due to limited time for data collection, training and testing, this had to be restricted for the current project. Hardware difficulties, typical of working with robots had also to be overcome, but the ultimate success of the method despite various technical difficulties showcases the potential performance of the method.

Bibliography

[1] R. Murphy, *Introduction to AI robotics*. The MIT Press, 2000.

[2] O. Lhomme, "Consistency techniques for numeric csps," in *International Joint Conference on Artificial Intelligence*, vol. 13, pp. 232–237, 1993.

[3] B. Leland, B. Christie, J. Nourse, D. Grier, R. Carhart, T. Maffett, S. Welford, and D. Smith, "Managing the combinatorial explosion," *Journal of chemical information and computer sciences*, vol. 37, no. 1, pp. 62–70, 1997.

[4] K. Fenner, J. Gao, S. Kramer, L. Ellis, and L. Wackett, "Data-driven extraction of relative reasoning rules to limit combinatorial explosion in biodegradation pathway prediction," *Bioinformatics*, vol. 24, no. 18, pp. 2079–2085, 2008.

[5] N. Nilsson, *Principles of artificial intelligence*. Springer Verlag, 1982.

[6] J. McCarthy, "Generality in artificial intelligence," *Communications of the ACM*, vol. 30, no. 12, pp. 1030–1035, 1987.

[7] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 604–613, ACM, 1998.

[8] S. Berchtold, C. Böhm, and H. Kriegal, "The pyramid-technique: towards breaking the curse of dimensionality," in *ACM SIGMOD Record*, vol. 27, pp. 142–153, ACM, 1998.

[9] J. Rust, "Using randomization to break the curse of dimensionality," *Econometrica: Journal of the Econometric Society*, pp. 487–516, 1997.

[10] F. Kuo and I. Sloan, "Lifting the curse of dimensionality," *Notices of the AMS*, vol. 52, no. 11, pp. 1320–1328, 2005.

[11] D. Lowe, "Object recognition from local scale-invariant features," in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 2, pp. 1150–1157, 1999.

[12] A. Tanenbaum, *Structured computer organization*. Prentice Hall PTR, 1984.

[13] R. Braden, "Requirements for Internet Hosts-Communication layers. RFC1122, Internet Engineering Task Force (IETF)," 1989.

[14] R. Braden, "Requirements for internet hosts-application and support," *RFC1123*, 1989.

[15] M. Sanner *et al.*, "Python: a programming language for software integration and development," *J Mol Graph Model*, vol. 17, no. 1, pp. 57–61, 1999.

[16] T. Lindholm and F. Yellin, *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[17] S. Lallée, S. Lemaignan, A. Lenz, C. Melhuish, L. Natale, S. Skachek, T. Zant, F. Warneken, and D. Ford, "Towards a platform-independent cooperative human-robot interaction system: I. perception," in *International Conference on Intelligent Robots and Systems (IROS 2010)*, pp. 4444–4451, 2010.

[18] D. Miller and R. Lennox, "An object-oriented environment for robot system architectures," *Control Systems Magazine, IEEE*, vol. 11, no. 2, pp. 14–23, 1991.

[19] E. Drumwright, V. Ng-Thow-Hing, and M. Mataric, "Toward a vocabulary of primitive task programs for humanoid robots," in *International Conference on Development and Learning (ICDL)*, 2006.

[20] E. Drumwright, V. Hing, and M. Mataric, "The task matrix framework for platform-independent humanoid programming," in *Humanoid Robots, 2006 6th IEEE-RAS International Conference on*, pp. 321–326, IEEE, 2006.

[21] D. Hristu-Varsakelis and S. Andersson, "Directed graphs and motion description languages for robot navigation," in *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, vol. 3, pp. 2689–2694, 2002.

[22] H. Hagras, V. Callaghan, and M. Collry, "Outdoor mobile robot learning and adaptation," *Robotics & Automation Magazine, IEEE*, vol. 8, no. 3, pp. 53–69, 2001.

[23] C. Bishop, *Pattern recognition and machine learning*, vol. 4. springer New York, 2006.

[24] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa, "Robocup: The robot world cup initiative," in *Proceedings of the first international conference on Autonomous agents*, pp. 340–347, ACM, 1997.

[25] J. Chacón, T. van Elteren, B. Hickendorff, H. van Hoof, S. Knuijver, C. Lier, A. Nolte, D. Mutis, P. Neculoiu, C. Oost, and F. Schimbinschi, "Borg-the robocup@ home team of the university of groningen: Team description paper," 2011.

[26] M. Lutz, *Programming python*. O'Reilly Media, Inc., 2006.

[27] R. Arkin, *Behavior-based robotics*. The MIT Press, 1998.

[28] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[29] W. Miller III, "Real-time neural network control of a biped walking robot," *Control Systems Magazine, IEEE*, vol. 14, no. 1, pp. 41–48, 1994.

[30] W. Miller III, R. Sutton, and P. Werbos, *Neural networks for control*. Bradford Books, 1995.

[31] T. Hesselroth, K. Sarkar, P. van der Smagt, and K. Schulten, "Neural network control of a pneumatic robot arm," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 24, no. 1, pp. 28–38, 1994.

[32] S. Shuzhi, T. Lee, and C. Harris, *Adaptive neural network control of robotic manipulators*, vol. 19. World Scientific Pub Co Inc, 1998.

[33] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[34] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural Networks, 1989. IJCNN, International Joint Conference on*, pp. 593–605, IEEE, 1989.

[35] R. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural computation*, vol. 1, no. 2, pp. 270–280, 1989.

[36] F. Pineda, "Generalization of back-propagation to recurrent neural networks," *Physical Review Letters*, vol. 59, no. 19, pp. 2229–2232, 1987.

[37] H. Jaeger, "The echo state approach to analysing and training recurrent neural networks-with an erratum note," tech. rep., Technical Report GMD Report 148, German National Research Center for Information Technology, 2001.

[38] H. Jaeger, *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the echo state network approach*. GMD-Forschungszentrum Informationstechnik, 2002.

[39] K. Ishu, T. van der Zant, V. Becanovic, and P. Ploger, "Identification of motion with echo state network," in *Techno-Ocean'04*, vol. 3, pp. 1205–1210, IEEE, 2004.

[40] S. Kirkpatrick, C. Gelatt Jr, and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[41] B. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, no. 5, pp. 469–483, 2009.

[42] S. Schaal, "Is imitation learning the route to humanoid robots?," *Trends in cognitive sciences*, vol. 3, no. 6, pp. 233–242, 1999.

[43] R. Dillmann, O. Rogalla, M. Ehrenmann, R. Zollner, and M. Bordegoni, "Learning robot behaviour and skills based on human demonstration and advice: the machine learning paradigm," in *Robotics research international symposium*, vol. 9, pp. 229–238, 2000.

[44] S. Schaal, "Learning from demonstration," *Advances in neural information processing systems*, pp. 1040–1046, 1997.

[45] P. Pastor, H. Hoffmann, T. Asfour, and S. Schaal, "Learning and generalization of motor skills by learning from demonstration," in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pp. 763–768, 2009.

[46] M. Nicolescu and M. Mataric, "Natural methods for robot task learning: Instructive demonstrations, generalization and practice," in *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pp. 241–248, ACM, 2003.

[47] R. Dillmann, "Teaching and learning of robot tasks via observation of human performance," *Robotics and Autonomous Systems*, vol. 47, no. 2, pp. 109–116, 2004.

[48] A. Billard, "Learning motor skills by imitation: a biologically inspired robotic model," *Cybernetics & Systems*, vol. 32, no. 1-2, pp. 155–193, 2001.

[49] P. Abbeel, D. Dolgov, A. Ng, and S. Thrun, "Apprenticeship learning for motion planning with application to parking lot navigation," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pp. 1083–1090, 2008.

[50] S. Lee and Z. Popović, "Learning behavior styles with inverse reinforcement learning," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, p. 122, 2010.

[51] D. Grollman and A. Billard, "Donut as i do: Learning from failed demonstrations," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 3804–3809, 2011.

[52] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber, "Pybrain," *The Journal of Machine Learning Research*, vol. 11, pp. 743–746, 2010.

[53] T. Oliphant, *A Guide to NumPy*, vol. 1. Trelgol Publishing, 2006.

[54] D. Skoog and D. West, *Principles of instrumental analysis*, vol. 5. Saunders College Publishing New York, 1985.

[55] Z. Zhang, "A flexible new technique for camera calibration," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 11, pp. 1330–1334, 2000.

[56] R. Tsai, "A versatile camera calibration technique for high-accuracy 3d machine vision metrology using off-the-shelf tv cameras and lenses," *Robotics and Automation, IEEE Journal of*, vol. 3, no. 4, pp. 323–344, 1987.

[57] Z. Roth, B. Mooring, and B. Ravani, "An overview of robot calibration," *Robotics and Automation, IEEE Journal of*, vol. 3, no. 5, pp. 377–385, 1987.

[58] F. Park and B. Martin, "Robot sensor calibration: solving ax= xb on the euclidean group," *Robotics and Automation, IEEE Transactions on*, vol. 10, no. 5, pp. 717–721, 1994.

[59] C. Wang, "Extrinsic calibration of a vision sensor mounted on a robot," *Robotics and Automation, IEEE Transactions on*, vol. 8, no. 2, pp. 161–175, 1992.

[60] G. Hinton and R. Zemel, "Autoencoders, minimum description length, and helmholtz free energy," *Advances in neural information processing systems*, pp. 3–3, 1994.

[61] S. Li, *Markov random field modeling in computer vision*. Springer-Verlag New York, Inc., 1995.

[62] T. Cover and P. Hart, "Nearest neighbor pattern classification," *Information Theory, IEEE Transactions on*, vol. 13, no. 1, pp. 21–27, 1967.

[63] R. Sutton and A. Barto, *Reinforcement learning: An introduction*, vol. 1. Cambridge Univ Press, 1998.

[64] M. Wiering and M. Van Otterlo, *Reinforcement Learning: State-Of-The-Art*, vol. 12. Springer-Verlag New York Incorporated, 2012.

[65] J. Sjöberg, Q. Zhang, L. Ljung, A. Benveniste, B. Delyon, P. Glorennec, H. Hjalmarsson, and A. Juditsky, "Nonlinear black-box modeling in system identification: a unified overview," *Automatica*, vol. 31, no. 12, pp. 1691–1724, 1995.

[66] K. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *Neural Networks, IEEE Transactions on*, vol. 1, no. 1, pp. 4–27, 1990.

[67] R. Johansson, A. Robertsson, K. Nilsson, and M. Verhaegen, "State-space system identification of robot manipulator dynamics," *Mechatronics*, vol. 10, no. 3, pp. 403–418, 2000.

[68] A. Gretton, A. Doucet, R. Herbrich, P. Rayner, and B. Schölkopf, "Support vector regression for black-box system identification," in *Statistical Signal Processing, 2001. Proceedings of the 11th IEEE Signal Processing Workshop on*, pp. 341–344, 2001.

[69] J. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, no. 3, pp. 293–300, 1999.

[70] T. Martinetz, S. Berkovich, and K. Schulten, "Neural-gas' network for vector quantization and its application to time-series prediction," *Neural Networks, IEEE Transactions on*, vol. 4, no. 4, pp. 558–569, 1993.

[71] J. Connor, R. Martin, and L. Atlas, "Recurrent neural networks and robust time series prediction," *Neural Networks, IEEE Transactions on*, vol. 5, no. 2, pp. 240–254, 1994.

[72] T. Masters, *Neural, novel and hybrid algorithms for time series prediction.* John Wiley & Sons, Inc., 1995.