Bernd Bachofner

Unsupervised learning and reinforcement learning for goal-directed decision making on high-dimensional input spaces

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur
Master's degree programme: Telematics

submitted to

Graz University of Technology

Institute of Theoretical Computer Science (IGI)
Inffeldgasse 16b/I, 8010 Graz

Thesis Advisor
Assoc.Prof. Dipl.-Ing. Dr.techn. Robert Legenstein

Graz, May 2016

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit/Diplomarbeit identisch.

Ort	Datum	Unterschrift

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Contents

Contents ii

List of Figures iv

List of Tables v

List of Listings vii

Acknowledgements 1

1 Introduction 5

2 Feature Extraction 7
- 2.1 Slow Feature Analysis . . . 7
- 2.2 The Learning Problem . . . 8
- 2.3 The SFA Algorithm . . . 8
- 2.4 SFA implementation details . . . 9
- 2.5 Hierarchical SFA networks for high-dimensional data . . . 10
- 2.6 Compact representations of visual scenes . . . 10

3 Model Based Learning 11
- 3.1 Reinforcement Learning . . . 11
- 3.2 Markov Decision Process . . . 12
- 3.3 Value Functions . . . 13
- 3.4 What is a Model . . . 13
- 3.5 The R-Max Algorithm . . . 14
- 3.6 Value Iteration . . . 15
- 3.7 Monte Carlo Methods . . . 17
- 3.8 Combining model and planning . . . 19

4 Feature Extraction Practical Section 23
- 4.1 Slow Feature Analysis (SFA) . . . 24
- 4.2 Hierarchical SFA networks for high dimensional data . . . 28
- 4.3 Simple 2D hierarchical SFA network . . . 29
- 4.4 3D hierarchical SFA network . . . 49

5 Model Based Learning Implementation 57
5.1 From Features to Environment . . . 58
5.2 The R-Max Model . . . 59
5.3 The Value Iteration Algorithm . . . 61
5.4 The Upper-Confidence-Bound applied to trees Algorithm . . . 65
5.5 Model Based Agent . . . 70
5.6 Experiments and Results . . . 73

6 Outlook 107
6.1 General Trends . . . 107
6.2 Ideas for Future Work . . . 107

7 Concluding Remarks 109

List of Figures

3.1 ModelAndPlanning 21

4.1 Input signal x1 35
4.2 Input signal x2 36
4.3 Input signal x2 37
4.4 Normalized input signal 38
4.5 Non-linear expansion 39
4.6 Sphered signal 40
4.7 Time derived signal 41
4.8 Output signal 42
4.9 Simplel Maze 60x60 43
4.10 Train32X 44
4.11 Train32Y 44
4.12 Test32X 45
4.13 Test32Y 45
4.14 Train64X 46
4.15 Train64Y 46
4.16 Test64X 47
4.17 Test64Y 47
4.18 3DNetwork 48
4.19 Test64Y 51
4.20 Test64Y 51
4.21 3DTrain32X 52
4.22 3DTrain32Y 52
4.23 3DTest32X 52
4.24 3DTest32Y 53
4.25 3DTrain64X 53
4.26 3DTrain64Y 54
4.27 3DTest64X 54
4.28 3DTest64Y 55
4.29 3DTest64XNoise 55
4.30 3DTest64YNoise 56
4.31 3DNetwork 56

5.1 Environment 3D . . . 76
5.2 Model base learner overview . . . 79
5.3 QValues Value Iteration left . . . 79
5.4 QValues Value Iteration up . . . 80
5.5 QValues Value Iteration right . . . 80
5.6 QValues Value Iteration down . . . 81
5.7 Environment 3D . . . 82
5.8 QValues UCT left . . . 83
5.9 QValues UCT up . . . 83
5.10 QValues UCT right . . . 84
5.11 QValues UCT down . . . 84
5.12 Environment 3D . . . 85
5.13 Environment 3D . . . 87
5.14 QValues Value Iteration left 3D . . . 88
5.15 QValues Value Iteration up . . . 89
5.16 QValues Value Iteration right . . . 89
5.17 QValues Value Iteration down . . . 90
5.18 Environment 3D . . . 90
5.19 QValues UCT left 3D . . . 92
5.20 QValues UCT up . . . 93
5.21 QValues UCT right . . . 93
5.22 QValues UCT down . . . 94
5.23 Environment 3D . . . 94
5.24 Environment 3D . . . 97
5.25 QValues Value Iteration left 3D . . . 98
5.26 QValues Value Iteration up . . . 98
5.27 QValues Value Iteration right . . . 99
5.28 QValues Value Iteration down . . . 99
5.29 Environment 3D . . . 101
5.30 QValues Value Iteration left 3D . . . 103
5.31 QValues Value Iteration up . . . 103
5.32 QValues Value Iteration right . . . 104
5.33 QValues Value Iteration down . . . 104
5.34 Environment 3D . . . 106
5.35 Environment 3D . . . 106

List of Tables

5.1 Value-Iteration parameters 75
5.2 UCT parameters 76
5.3 Q-Values for Value-Iteration in 2D 77
5.4 Q-Values for UCT in 2D 78
5.5 Value-Iteration parameters 88
5.6 Q-Values for Value-Iteration in 3D 91
5.7 UCT parameters 92
5.8 Q-Values for UCT in 3D 95
5.9 Value-Iteration parameters 97
5.10 Q-Values for Value-Iteration in 3D 100
5.11 Value-Iteration parameters 102
5.12 Q-Values for Value-Iteration in 3D 105

Listings

3.1 Pseudo code of the R-Max algorithm . . . 15
3.2 Pseudo code of the Value-Iteration algorithm . . . 16
3.3 Pseudo code of the UCT algorithm . . . 18
4.1 Simple reference implementation of the SFA algorithm . . . 31
4.2 Linear SFA train reference implementation . . . 32
4.3 Linear SFA execute reference implementation . . . 33
4.4 Quadratic SFA train reference implementation . . . 33
4.5 Quadratic SFA execute reference implementation . . . 33
4.6 Linear Regression, to evaluate the SFA network . . . 34

Acknowledgements

I want to thank my advisor, Robert Legenstein, for his immediate attention to my questions and hours of toil in correcting draft versions of this thesis.

I also want to thank my family, without their support and understanding this thesis would not have been possible.

Special mention goes to Werner Trobin, who was a great help in finalizing this thesis. Without his help I never would have developed the knowledge to implement such a project.

Bernd Bachofner
Graz, Austria, May 2016

Abstract

The focus of this thesis is to evaluate if an unsupervised image processing system can provide relevant information for a goal-directed decision making system. We use agent-based navigation in a 3D environment to demonstrate how the decision system works. The visual input to the system is pre-processed by the image-processing system. For this purpose a synthetically generated image sequence of a 3D environment serves as input. The system should be able to extract the most important information out of the image sequence. This information is then given as input to the decision system. The decision system, a so called agent, should be able to make decisions based on data provided by the image processing system, to achieve some goal. We discuss the steps which are necessary to setup such an unsupervised image processing system and we will show how its output is used in the decision making system. A hierarchical Slow Feature Analysis (SFA) network is used to process the high-dimensional input data. The network contains four layers, each layer servs as an input to the next layer. The first layer processes the input data, while the final layer provides us the low dimensional output. The output of the final layer is then processed using Independent-Component-Analysis (ICA) which removes the remaining redundant information. We then discretize the data (or features) calculated in the ICA step which serve as an input for the decision making system. The decision making process is based on a so called model based learning system. This means a model of the environment (state space) is used to support the agent during the decision making process. The model is an abstraction of the environment and keeps statistics and probabilities for each state. In this thesis we compare two algorithms for the learner or decision maker, Value-Iteration and UCT. Both methods belong to the category of Reinforcement Learning (RL), this means the algorithms try to maximize a reward value. Both Value-Iteration and UCT try to attain a high reward by devising an optimal decision policy. The Value-Iteration method is based on dynamic programming methods, while UCT uses Monte Carlo Tree Search (MCTS). Both algorithms perform pretty well in archiving a certain goal; for our specific problem Value-Iteration performs better.

Der Zweck dieser Diplomarbeit ist es die folgende Frage zu beantworten: Ist es möglich ein nicht überwachtes (Unsupervised) Bildverarbeitungssystem einzusetzen, welches die relevanten Informationen für ein zielorientiertes Entscheidungssystem extrahiert. Um zu demonstrieren wie ein solches Entscheidungssystem funktioniert, implementieren wir ein Navigationssystem, welches einen Agent in einer 3D Welt steuert. Der visuelle Input für dieses Navigationssystem wird so aufbereitet, dass das System in der Lage ist jede beliebige Position in der 3D Welt zu finden. Diese Aufgabe wird von unserem Bilderverarbeitungssystem übernommen. Es wurde eine Sequenz von künstlichen 3D Bildern erzeugt, diese Sequenz dient als Input für das Bildverarbeitungssystem. Dem System sollte es möglich sein, die wichtigsten Informationen aus dieser Sequenz zu extrahieren, welche dann als Input für das Entscheidungssystem dienen. Das Entscheidungssystem sollte in der Lage sein Entscheidungen auf der Basis der Daten zu treffen, die vom Bildverarbeitungssystem zu Verfügung gestellt wurden. Im Folgenden zeigen wir was nötig ist, um ein solches Bildverarbeitungssystem zu implementieren. Das System verwendet ein sogenanntes hierarchisches Slow Feature Analysis (SFA) Netzwerk, dies besteht aus vier Schichten (Layer), jede Schicht dient als Input für die nächste. Die erste Schicht verarbeitet die visuellen Daten unserer künstlichen 3D Welt, während die letzte Schicht die auf das Wesentliche reduzierten Informationen liefert. Der Output der letzten Schicht dient als Input für den Independent-Component-Analysis (ICA) Algorithmus, diese Methode entfernt die verbleibenden redundanten Informationen. Nach der Diskretisierung der Daten (Features), werden diese als Input für das Entscheidungssystem verwendet. Die Diskretisierung dient dazu die vom Bildverarbeitungssystem zur Verfügung gestellten Informationen in ein für das Entscheidungssystem verwendbares Format umzuwandeln, mit anderen Worten wird ein sogenannter Zustandsraum (State Space) erzeugt. Das Entscheidungssystem wird von einem Agenten repräsentiert. Dieser Agent trifft Entscheidungen mithilfe eines Modells der Umgebung. Das Model enthält Informationen über jeden Zustand im Zustandsraum (Model-Based-Learning). Das Entscheidungssystem verwendet zwei Algorithmen aus der Kategorie Reinforcement Learning (RL), Value-Iteration und UCT. Reinforcement Learning bedeutet dass die Algorithmen versuchen ihre Belohnung (Reward Value) zu maximieren. Der Value-Iteration Algorithmus verwendet Methoden aus dem Gebiet des dynamischen Programmierens, während der UCT Algorithmus auf Monte Carlo Tree Search (MCTS) basiert. Beide Algorithmen sind in der Lage die an ihnen gestellten Aufgaben zu erfüllen, jedoch ist Value-Iteration für die in dieser Arbeit gestellten Aufgaben die bessere Wahl.

Kapitel 1

Introduction

The goal of this thesis can be described in one sentence: Create a system which is able to plan an agent's behavior based on high-dimensional visual input in 2D or 3D environments. The system can be broken down into two main parts:

- The visual system: A system that is able to extract the most important information out of the image data. This implies some kind of information compression from high dimensional input data (the image) to low dimensional data (features) to simplify further processing.
- The planning system: A system that is able to build a representation of the environment using data provided by the visual system. Based on this representation a so called agent explores the environment to find optimal actions.

The visual system, or feature extraction system, receives a temporal sequence of images, which means that the first image represents the input at time t_0 the second the input at time t_1 and so on until t_n. The feature extraction system uses these high dimensional input data, to generate a low dimensional representation of minimal redundant information. One goal of this thesis is to use unsupervised learning systems for this feature extraction. For the visual system the decision was made to use the so called „Slow Feature Analysis "(SFA) method. This algorithm has been used successfully in similar situations. SFA has been used to create spatial maps from 3D movies [4]. It has also been used as preprocessing for a reinforcement learning system on simple 2D movies [25]. In the latter work, a simple model-free reinforcement learning algorithm was applied. In this thesis we show that SFA can be used as a visual preprocessing system (visual system) for a model-based planner on pseudo realistic 3D movies. A complete description of the algorithm can be found in Chapter 2, the results of the method when applied to simple binary images and 3D images from a simulated environment can be found in Chapter 4.

The planning system: This system is based on 'model-based reinforcement learning'. The data provided by the visual or feature extraction system are used to generate a model of the environment. The agent uses this model to plan its actions on it. This approach is very popular in computer science at the moment, since a lot of scientists think that this is a good way to emulate how the human brain works when we are learning, like for example running. This example sounds strange, how to learn running?!, but when a human tries to run a long distance for the first time, most likely he or she will fail, not only because of the physical strain, but the human also needs to learn how to run such long distances (reward system) and also needs to get familiar with the track (model). Exactly for such situations a so called reward system comes into mind, the first time a human tries to run up a hill he or she most likely runs too quickly, so the consequence is that the human has to slow down. To a runner this is some kind of punishment, the runner is not able to run the complete distance. The next time the human already knows the track and so he or she will run up the hill a bit slower and then will succeed because the runner is

able to run the complete distance, so the human gets a reward.
Similar systems can be used in the world of artificial intelligence. The agent gets a reward when it for example doesn't hit a wall or gets punished when it hits a wall. A description of the theoretical ideas and the algorithms can be found in Chapter 3. How the different algorithms were implemented and the results of the different experiments can be found in Chapter 5.
The final chapter of this thesis provides an overview of potential future work, based on the different findings that have been made during creation of this thesis also some points of improvement have been discovered.
The final chapter 6 also contains some references to papers and web-sites that have been used to create this thesis. A lot of code or programs have been written during the work to create a simulation environment in which all the algorithms can be tested in different situations. A manual on how to use these programs can be found in the attached source code in the 'Readme.txt' file.

Kapitel 2

Feature Extraction

In artificial intelligence or machine learning, a feature is a measurable property of an observed event [1]. As already explained in the introduction of this thesis, the input data to the visual system are high dimensional. This means that the images contain a lot of redundant data, which hamper further processing and can be neglected. The purpose of the feature extraction system is to provide discriminative and independent features. Quite a lot of algorithms are available to perform dimensionality reduction like Principal Component Analysis (PCA) [2], Independent Component Analysis (ICA) [3] etc. Because one of the goals of this Master Thesis was to use only unsupervised learning methods the decision was made to use the so called „Slow Feature Analysis (SFA)“ [4]. SFA is based on learning invariances from temporal input sequences. In our case this means that we have a sequence of images that represent the agent's movement through a synthetically generated 3D world based on the „Two rooms problem“ introduced by Todd Hester and Peter Stone [7]. The main idea of SFA [4] is that localized properties of an environment vary quickly (e.g.: small parts of a image sequence where a bird flies through) while the overall scene (e.g.: the whole image sequence) varies slowly. SFA exploits this fact and picks slowly varying features to obtain an invariant representation of the environment. In contrast to most other feature extraction methods like ICA, where the input signal is processed directly, the SFA algorithm can use linear or nonlinear expansions of the input signal. Further processing can be performed, e.g., by Eigen-decomposition or PCA on the signal, which leads to an solution that provides (if intended) a large number of decorrelated features, that are ordered by their degree of invariance. Another advantage of SFA is, that so called SFA-Nodes can be grouped in a hierarchical network that represents a simple model of the visual system, which on the one hand simulates biological behaviour and on the other hand enables to massive parallelism. All nodes on the same hierarchical level are independent and can therefore be calculated in parallel. Such a hierarchical network exactly fits our problem of extracting relevant features from a large amount of image data produced in this thesis. In the following section a precise explanation of the SFA algorithm is given as well as some experiences which were made during the work with the algorithm on image data. Results of trainings and test data are then presented in a practical section.

2.1 Slow Feature Analysis

The basic idea behind feature extraction and object recognition algorithms is to generate an invariant representation of the sensory inputs (e.g. retina or camera). Consider for example three different objects (Letters) which move through the visual field (whole image) with different directions and speeds. From a high level point of view each of the signals can be represented by three time dependent variables. The first variable indicates the object (which letter is currently visible), the second and third variables indicate the vertical and horizontal location of the object. The receptive field (the object covers many sensors) represents one sensor, that only provides signals of localized features of the objects, such as gray values, dots or edges. These sensors respond to localized features, this means their output signals change quickly.

The output contains information like object identity and location in an implicit way. The vast majority of the calculated input-output functions would generate quickly varying output signals, only small fractions of the generated functions will generate slowly varying output signals.

2.2 The Learning Problem

As mentioned above, Slow Feature Analysis is all about selecting slowly varying features from a large number of input signals. In order to implement such a feature selection algorithm, we first state SFA as a machine learning problem. „ Given a vectorial (potentially high dimensional) input signal $\mathbf{x}(t)$, the goal is to find an input-output function $g(\mathbf{x})$ such that the output signal $y(t) := g(\mathbf{x}(t))$ varies as slowly as possible. “ [4]
Consider the high dimensional input signal $\mathbf{x}(t)$. To find the input-output functions $g_j(x)$ the following steps are performed. The jth output $y_j(t)$ of the SFA algorithm is defined as some function g_j of the input signal

$$y_j(t) := g_j(x(t)). \tag{2.1}$$

The objective of the optimization is to minimize

$$\Delta(y_j) := \langle \dot{y_j^2} \rangle_t, \tag{2.2}$$

where $\dot{y}$ is the derivative of y with respect to time and $\langle . \rangle_t$ denotes the temporal average. The Δ value defined by equation (2.2) is the objective of the learning (optimization) problem and measures the slowness of the output signal. A low value indicates small variations over time, and therefore slowly-varying signals. To avoid trivial solutions such as $y_j(t) = const$ the output is constrained to have zero mean and unit variance. Hence, the optimization is performed constrained to

$$\langle y_j \rangle_t = 0 \tag{2.3}$$

$$\langle y_j^2 \rangle_t = 1 \tag{2.4}$$

Additionally, decorrelation and order is imposed via the constraints

$$\langle y_i y_j \rangle_t = 0 \; \forall i < j. \tag{2.5}$$

This constraint guarantees that different output signal components carry different information. The constraint also yields an order such that $y_1(t)$ is slower varying than $y_2(t)$ which means that $y_1(t)$ is the better output signal than $y_2(t)$. Constraints 2.3 and 2.4 could be dropped and replaced by a single step procedure $\frac{\langle \dot{y_j^2} \rangle}{\langle (y_j - \langle x_j \rangle)^2 \rangle}$ explained in [8] to reduce computational complexity.

The algorithm presented in the following section solves the problem under the constraints (2.3) and (2.4). The problem however is, that the reference algorithm is computationally expensive and therefore, the implemented version differs in some points from the reference version. These changes are minimal and will be explained in the practical section.

2.3 The SFA Algorithm

Given an I-dimensional input signal $\mathbf{x}$(t)= $[x_1(t), ..., x_I(t)]^T$ consider an input-output function $g(\mathbf{x}) = [g_1(\mathbf{x}), ..., g_J(\mathbf{x})]^T$ each component of which is a weighted sum over a set of K non-linear functions $h_k(\mathbf{x})$, i.e. $g_j(\mathbf{x}) := \sum_{k=1}^{K} w_{jk} h_k(\mathbf{x})$. Applying $\mathbf{h} = [h_1, ..., h_K]^T$ to the input signal yields the non linear expanded signal $\mathbf{z}(t) = \mathbf{h}(\mathbf{x}(t))$. After applying the non linear expansion the problem can be handled as linear in the expanded signal components $z_k(t)$. The weight vectors $\mathbf{w}_j = [w_{j1}, ..., w_{jK}]^T$ are then learned and the jth output signal component is given by $y_j(t) = g_j(\mathbf{x}(t)) = \mathbf{w}_j^T \mathbf{h}(\mathbf{x}(t)) = \mathbf{w}_j^T \mathbf{z}(t)$.

As mentioned above the objective of this learning problem is to minimize equation 2.2 which means we have to optimize the weights. This is shown in the following equation

$$\Delta(y_j) = \langle \dot{y}_j^2 \rangle = \mathbf{w}_j^T \langle \dot{\mathbf{z}}\dot{\mathbf{z}}^T \rangle \mathbf{w}_j. \quad (2.6)$$

The non linear function h_k is chosen in a way such that the expanded signal $\mathbf{z}(t)$ has zero mean and a unit covariance matrix. Such a set h_k of non-linear functions can be found by a method called sphering (which is the normalization of the expanded $z(t)$ signal). Zero mean and unit variance of the output signals y_j are then fulfilled automatically if the weight vectors are constrained to be an orthonormal set of vectors (if they are all unit vectors and orthogonal with respect to each other),

$$\langle y_j \rangle = \mathbf{w}_j^T \langle \mathbf{z} \rangle = \mathbf{w}_j^T \times 0 = 0 \quad (2.7)$$

$$\langle y_j^2 \rangle = \mathbf{w}_j^T \langle \mathbf{z}\mathbf{z}^T \rangle \mathbf{w}_j = \mathbf{w}_j^T I \mathbf{w}_j = 1 \quad (2.8)$$

$$\forall j' < j : \langle y_{j'} y_j \rangle = \mathbf{w}_{j'}^T \langle \mathbf{z}\mathbf{z}^T \rangle \mathbf{w}_j = \mathbf{w}_{j'}^T \mathbf{w}_j = 0 \quad (2.9)$$

To find the solution of equation (2.6), we can reformulate the problem to find the normalized weight vector that minimizes $\Delta(y_1)$, which means the solution is the normalized eigenvector of matrix $\langle \dot{z}\dot{z}^T \rangle$ that corresponds to the smallest eigenvalue. The next pair of eigenvectors and eigenvalues with larger eigenvalue produces components of the input output function with next higher delta-values, and so on.

2.4 SFA implementation details

In the following a high level description of the building blocks which are used to create the SFA algorithm is given. A detailed description including Matlab code follows in the practical section.

- Normalization: As already mentioned above the input signal needs to be normalized to a zero-mean signal with unit variance
  $x_i(t) = \frac{\widetilde{x}_i(t) - \langle \widetilde{x}_i \rangle}{\sqrt{\langle (\widetilde{x}_i - \langle \widetilde{x}_i \rangle)^2 \rangle}}$, where $\widetilde{x}_i(t)$ is the input signal.
  There are other methods to normalize the input signal, which are not as computationally expensive as this one. One of these methods is used and explained in the practical section, see Chapter: 4. The normalization fulfills the constrains of $\langle x_i \rangle = 0$ (zero mean) and $\langle x_i^2 \rangle = 1$ (unit variance).

- Expansion: The expansion is used to transform a non-linear problem to a linear one. This can be done by applying non-linear functions $h(\mathbf{x})$ to generate an expanded signal. Linear functions $h(\mathbf{x})$ can be obtained from any monomials of degree one which results in a so called Linear-SFA. Non-linear functions can be obtained from monomials of degree one and two including mixed terms which then results in a so called quadratic-SFA. The quadratic version of the SFA algorithm for a two-dimensional input signal $\mathbf{x}(t) = [x_1(t), x_2(t)]^T$ uses to following expansion: $\mathbf{z}(t) = [x_1, x_2, x_1^2, x_1 x_2, x_2^2]^T$.

- Sphering or whitening: This is the normalization of the expanded signal $\mathbf{z}(t)$ and can be done with Principal Component Analysis (PCA) [2]. Sphering of $\mathbf{z}(t)$ assures that the constraints 2.3-2.5 are fulfilled for orthogonal weight vectors. After sphering the resulting signal $\mathbf{z}(t)$ has unit variance in all directions. The sphered signal $\dot{\mathbf{z}}(t)$ is computed from a signal $\widetilde{z}_t$ as
  $\mathbf{z}(t) := S(\widetilde{z}(t) - \langle \widetilde{z} \rangle)$, where S is the so called sphering matrix which can be calculated using PCA.

- PCA [2]: This can be done as follows: To calculate the PCA the so called Singular Value Decomposition (SVD) algorithm can be used. It is also possible to use Eigen decomposition [10] but in this thesis the SVD was used.
- Time derivative: Here the temporal derivative of the sphered signal $\mathbf{z}(t)$ is calculated. This can be done with the following formula:
  $\dot{\mathbf{z}}(t) = \frac{z(x_0+h)-z(x_0-h)}{2h}$, where h is the step-size, which is the approximation of the derivation along the time axis.
  Another way is to use the integrated Matlab function „filter“, this function is used in the practical section 4.
- Extract the slowest varying direction: The goal here is to find the direction in which the time derivative of the sphered signal varies most slowly. To find this direction one can use Eigen decomposition to find the eigenvector corresponding to the smallest eigenvalue $\langle \dot{\mathbf{z}}\dot{\mathbf{z}}^T \rangle$.

2.5 Hierarchical SFA networks for high-dimensional data

Non linear SFA, like the quadratic version of the SFA, require a significant amount of resources (memory, computation time) due to the high dimensionality. The expanded function space increases quickly with the number of the input signals. This is especially problematic with high dimensional data like visual data, that are heavily used in this thesis. For example the quadratic SFA of an input image with the size 100×100 yields a dimensionality of $50.015.000$, which is a problem even for modern computers.
One way to overcome this is to apply the SFA only on small sub-parts of the input, extract the slowest varying features for each sub-part and then use these outputs as input for another iteration of the SFA [5]. This would solve the problem of dimensional explosion. It should be mentioned here that the final slow features extracted, may not be identical with the results of the SFA which has the whole data as input. The splitting of the input data relies on the locality of feature correlations which holds for natural images.
Hierarchical networks with feed-forward organization can be composed to represent a complex visual system. Refer to practical section of the SFA Chapter 4 for a schematic representation of a SFA network. This approach seems to have a lot of advantages. For example the more layers are used, the larger the receptive filed size gets, which means that even complex structures like whole objects can be recognized. Another advantage is that in each layer all SFAs can be calculated in parallel which fits well to modern CPU architecture. It is also possible to split the calculation to a network of computers and then in a final step concatenate the different results. As mentioned above, image data are heavily used in the thesis and so hierarchical SFA networks were used. A detailed explanation of the used network will be given in the practical section.

2.6 Compact representations of visual scenes

For the representation of visual scenes two features are important. First of all the object identities, which provides the information to recognize an object. Second the configuration of an object, which provides information about the object's position and orientation. All these features are typically slow features and so they will usually be selected by the SFA. So after training a hierarchical SFA network with visual input, the network should be able to extract features like object identity and configuration. Another advantage of hierarchical networks is that for "simple"situations it is capable to retrieve individual features, since they are independent of each other. For example: One feature represents the object's identity while the other feature represents the object's position.

Kapitel 3

Model Based Learning

3.1 Reinforcement Learning

Model Based learning is one approach in a large research field called Reinforcement Learning (RL). So before we can go to Model Based learning the expression Reinforcement Learning needs to be explained in more detail. Reinforcement Learning learns, how to map situations to actions and maximizes a reward value or signal [6]. The learning system is not told which actions to take, as it is the case in supervised learning, instead it must discover which actions yield the most reward by trying them out. In situations like in this thesis this means that actions may not only affect the immediate reward but also the next actions and rewards. The properties of RL can be summarized in three main characteristics:

1. It is a closed loop problem, this means the systems actions influence its later inputs.

2. The system does not have instructions on which action to take next.

3. The consequences of taking an action and receiving a reward is played out over a extended time period.

In our case this means that an agent (learner or decision maker) must be able to sense the current state of the environment and then perform actions on this environment to achieve a goal. RL tries to maximize a reward value.

3.1.1 The Agent-Environment Interaction

The agent interacts with the outside world, the environment. In our case a OpenGL generated 3D world is used as input to the feature extraction system. The feature extraction system then provides features or states, this is the environment of the agent. The agent and the environment interact at each point in time $t = 0, 1, 2, 3, ...n$. At each time step the agent gets information about the environment's state s_t, $s \in S$, where S is a set of possible states (in our case these are the independent features provided by the SFA). Depending on the current state the agent selects an action a_t, $a \in A$, where A is a set of all possible actions (in our case this means Left, Up, Right and Down). One time step later the agent receives a numerical reward $r_{t+1} \in \mathbb{R}$, and finds itself in a new state. At each time step the agent calculates probabilities of selecting each possible action. This is called the agent's policy π_t, where $\pi_t(a|s)$ is the probability that $a_t = a$ if $s_t = s$. The change of the policy depends on the experiences the agent made in the environment. The goal of the agent is to maximize the total reward it receives over the time period.

3.1.2 Discount rate

At each time step the agent receives a reward r. This means the agent receives a sequence of rewards t, $r_{t+1}, r_{t+2}, r_{t+3}, ..,$. We want to maximize the so called 'expected return', where G_t is defined as function of the reward sequence. In a simple scenario the return is the sum of all rewards, $G_t = r_{t+1} + r_{t+2} + ... + r_T$, where T is the final step. A problem with the above formula appears when for example the environment interaction does not stop after a finite number of time steps, but instead continues without limit. In this case the return which we try to maximize could itself be infinite. To overcome the problem a discount rate or factor γ is used. The agent tries to select actions that maximize the sum of discounted rewards. In other words it chooses a_t to maximize the expected discounted reward , $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$, where γ is a parameter, $0 \leq \gamma \leq 1$, called discount rate. The discount rate represents the value of future rewards. This means a reward received k time steps in future is treated as γ^{k-1} times what is worth it received immediately. When choosing $\gamma < 1$ the reward sum is finite, as long as the reward sequence is limited. When $\gamma = 0$ the agent tries to maximize its immediate rewards, this means the agent maximizes only r_{t+1} (this is called a myopic agent). When $\gamma = 1$ the agent is focused on maximizing its future rewards (this is called a farsighted agent). The agent decides on what do to next, by comparing different sequences of rewards. This is done by converting a sequence of rewards to a number called the value of the cumulative reward. The agent combines an immediate reward with other rewards in the future. For example the agent receives the following sequence of rewards: r_1, r_2, r_3, r_4,.... Three ways exist to combine rewards into a value V':

- Total reward: $V = \sum_{i=1}^{\infty} r_i$. Here the value is the sum of all of the rewards, this works if the sum is finite. This is only the case when a stopping state exists and when the agent always has a probability > 0 of entering the stopping state.
- Average reward: $V = \lim_{n \to \infty} (r_1 + ... + r_n)/n$. Here the value is the average of its rewards, averaged over each time period. When the reward is finite, the average will also be finite. However when the total reward is finite, the average reward is zero. The result is that the agent will not be able to choose an action since each has a zero average reward.
- Discounted reward: $V = r_1 + \gamma r_2 + \gamma^2 r_3 + ... + \gamma^{i-1} r_i + ...$, where γ, the discount factor, is a number in the range of $0 \leq \gamma \leq 1$. Future rewards are weighted less than the current reward. If $\gamma = 1$, this would have the same effect as using total reward, and when $\gamma = 0$, the agent would ignore all future reward. $0 \leq \gamma \leq 1$ ensures that the reward is finite, then also the total value will be finite.

3.1.3 Episodic Tasks

A task is called 'Episodic' if the agent-environment interaction breaks down into a sequence of separated episodes. Mathematically this means that each action affects only a finite number of rewards subsequently received during an episode. Instead of only one long sequence of time steps, an episodic-task can be seen as a series of episodes each of which consists of a finite sequence of time steps.

3.2 Markov Decision Process

The Markov Decision Process (MDP) is the most important mathematical construct to understand reinforcement learning. Almost every algorithm which is used in RL is based on MDPs. In this thesis the so called 'Finite MDP' is used which means the state and actions spaces are finite. The MDP is defined by state and action sets and by the one-step dynamics of the environment. Given a state s and an action a, the probability of each possible pair of next state s' and reward r is calculated with $P(s', r|s, a)$ where P is for example the probability that $s_{t+1} = s'$, this means that the next state is the specific

state s'. Given the above formula one can calculate the state transition probabilities the following way, $P(s'|s,a) = \sum_{r \in R} P(s', r|s, a)$.

- $P(s'|s,a)$, specifies the probability of transitioning to state s' given that the agent is in state s and performs the action a
- $R(s,a,s')$, gives the expected immediate reward for doing action a and a transitioning to state s' from state s

3.3 Value Functions

The value function estimates the best action to perform in a state or in other words how good is it to perform a given action in a given state. To describe what 'how good' means, one can say, that it is the future rewards that can be expected. Value functions are defined with respect to particular policies, a policy π, is mapping from a state s and an action a to the probability $\pi(a|s)$, of taking action a when in state s. Consider $V_\pi(s)$ as the value of state s under the policy π, for MDPs $V_\pi(s)$ the so called 'state-value function for policy π' is calculated as, $V_\pi(s) = E_\pi[G_t|s_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s]$, where $E_\pi[G_t|s_t = s]$ is the expected value of G_t, given that $s_t = s$ and the agent follows policy π. G_t is the function of the reward sequence. The 'action-value function for policy $Q_\pi(s,a)$', which is the value of taking action a in state s under policy π can be calculate as $Q_\pi(s,a) = E_\pi[G_t|s_t = s, a_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a]$.

3.4 What is a Model

A model of the environment can be seen as a resource that the agent can use to predict how the environment will respond to the actions it performs [6]. When the agent performs an action a from state s, the model should be able to predict the next state, s' and the reward, r the agent receives by taking this action. In a stochastic model there exist several possible next states and rewards, each with some probability of occurring.
The usual way to learn a model is to learn a tabular maximum likelihood model. The tabular maximum likelihood algorithm maintains a count, $C(s,a)$, of times an action a was taken form state s. Furthermore the algorithm maintains a count, $C(s,a,s')$, of the number of times that each next state s' was reached from (s,a) and the transition probability $T(s,a,s')$ that the next state is s' when performing action a in state s.
The probability of outcome s' is then:

$$P(s'|s,a) = T(s,a,s') = C(s,a,s')/C(S,a) \tag{3.1}$$

The algorithm keeps statistics about the sum of rewards it has received from each transition, $Rsum(s,a)$, and computes the expected reward for a particular state action to be the mean reward received from that state action:

$$R(s,a) = Rsum(s,a)/C(s,a) \tag{3.2}$$

The tabular model learning is easy to use and also easy to implement. It is used in model based algorithms like R-Max [12], which is used in this thesis to generate the model out of the environment. The algorithm is sample efficient and produces good results that can easily be used by planning algorithms like Value-Iteration. A detailed description about R-Max will follow in its own section.

3.5 The R-Max Algorithm

The R-Max algorithm [12] is a model based method, which learns a tabular maximum likelihood model. The advantage of using model based methods compared to model free ones is that they can provide better sample efficiency.
The expression 'sample efficiency' refers to the number of actions that an algorithm needs to learn an optimal policy. Kakade [9] proved a lower bound for sample complexity of
$O(\frac{NA}{\varepsilon(1-\gamma)} \log \frac{1}{\delta})$,
for stochastic domains, where N is the number of states, A is the number of actions, γ is the discount factor. The algorithm finds an ε-optimal policy (this means the actions chosen by the policy, differ only in the boundaries of some ε value over a period in time) with probability $1 - \delta$.
As already mentioned model based methods can be more sample efficient than model free approaches. The question is how many actions are needed to learn an accurate good model. Here a second algorithm namely Explicit Explore or Exploit E^3 is used. This algorithm serves as reference for the R-Max algorithm, because it was the first algorithm that proves to be a near optimal policy in polynomial time.
Before digging deeper into the E^3 algorithm two important expressions needs to be explained.

1. Exploration: Is the process of gathering statistics and knowledge about the environment.
2. Exploitation: Is the process of maximizing the rewards after the agents knows 'enough' about the environment.

E^3 keeps track of visits to each state and treats states with fewer than m visits as unknown. Whenever the agent reaches an unknown state it performs balanced wandering (taking the action that has the lowest probability from that state). By doing this, the agent learns more about the unknown state and the state may become a known state after a couple of visits by the agent. If the agent reaches a known state it performs an exploration policy that gets to an unknown state as quickly as possible. If the probability to reach an unknown state is greater than $\frac{\varepsilon}{2*Rmax}$ (where R_{max} is the maximum reward), this policy is followed, if not, it was proven that planning an optimal policy must result in a policy that is within ε optimal. This means the algorithm plans then an approximate optimal policy and follows it.
„With probability no less than $1 - \delta$, the E^3 algorithm will return a value greater than the optimal value – ε, in a number of steps polynomial in N, T, R_{max}, $\frac{1}{\varepsilon}$ and $\frac{1}{\delta}$." [7].
Both E^3 and R-Max have proven bounds on sample efficiency. They learn a tabular maximum likelihood model and keep counts about known and unknown states. The main difference to E^3 is that the R-Max algorithm replaces unknown transitions in the model with so called transitions to an absorbing state. An absorbing state is a state where all actions of an agent leave the agent in this absorbing state and provides the agent with the maximum reward $Rmax$. The R-Max algorithm is simpler to understand and to implement than the E^3 algorithm, and it employs an implicit explore or exploit mechanism rather than explicitly deciding between the two policies as E^3 does.
A pseudo code implementation of the R-Max algorithm can be seen below, the real implementation is explained in the practical section.

Definition of the input variables for the R-Max algorithm:

- S is defined as a set of states of the world.
- A is defined as a set of actions for example: up, right, down, left.
- m is a threshold that defines when a state can be considered as 'known'. This means that as long as the number of visits to a state is less than m the state is considered to be unknown.
- R_{max} is the maximum reward.

Listing 3.1: Pseudo code of the R-Max algorithm

```
Algorithm R_Max (Inputs S, A, m, Rmax)
// Initialize sr as absorbing state with reward Rmax
for all a in A do
        R(sr ,a) <- Rmax
        T(sr ,a, sr) <- 1
end for
Initialize s to a starting state in the MDP
loop
        // Sample an action from the policy pi
        Choose a = pi(s)
        Take action a, observe r, s_dash
        // Update model
        Increment C(s,a, s_dash)
        Increment C(s,a)
        RSUM(s,a) <- RSUM(s, a) + r
        if C(s,a) >= m then
                // Known state, update model using experience counts
                R(s,a) <- RSUM(s, a) / C(s, a)
                for all s_dash in C(s, a, .) do
                        T(s,a, s0) C(s,a, s0)/C(s,a)
                end for
        else
                // Unknown state, set optimistic model transition to absorbing
                    state
                R(s, a) <- Rmax
                T(s,a, sr) <- 1
        end if
        // Plan policy on updated model
        Call VALUE-ITERATION || UCT
        s <- s_dash
 end loop
```

3.6 Value Iteration

3.6.1 Value of a Policy

One of the main goals of this thesis is to implement an agent based system which can be used to solve problems like path-finding. As mentioned earlier, the states on that the system operates on, are provided by the feature extraction process called the SFA algorithm. The problem for the agent is to decide in each and every state which action to perform, the resulting state depends on both the previous state and the current performed action. To model such problems the so called MDP is used. In the following an explanation of the model is given plus an algorithm (Value-Iteration) is presented to solve this problem. A reward system is the only feedback guiding this process. Depending on the actions the algorithm performs either a reward or punishment is given. Negative rewards are called punishments.

3.6.2 Value of an Optimal Policy

The main task in solving a reinforcement learning problem is to find a policy that yields a high reward over a given time span. For finite MDPs the definition of an optimal policy can be stated as, $\pi \geq \pi'$, where π' is any other possible policy. A optimal policy π is better or at least as good as any other policy π' if its expected return (reward) is greater than or equal to that of π' for all states. Or using value functions, $V_\pi(s) \geq V_{\pi'}(s) \forall s \in S$, where S denotes the complete state space. There exists at least one policy that is better than or equal to all other policies, this is then called an optimal policy [6]. Consider the optimal

state-value function as V^* that is defined as $V^*(s) = \max(V_\pi(s)), \forall s \in S$. The optimal action-value function is defined as $Q^*(s,a) = \max(Q_\pi(s,a)) \forall s \in S$ and $a \in A$, where A denotes the all possible actions. The optimal policy is then denoted as π^* and is defined as $\pi^*(s) = \mathrm{argmax_a Q^*(s,a)}$, where $\mathrm{argmax_a}$ is a function of state s, and its value is one of the actions a that result in the maximum of $Q^*(s,a)$.

3.6.3 Value-Iteration Algorithm

The Value-Iteration method is a procedure for computing an optimal MDP policy and its value. The algorithm uses an arbitrary endpoint, from this endpoint the algorithm works backward, refining an estimate of either $Q^* or V^*$ [11].
Let us define V_k as the state-value function, after the kth update. Q_k is defined to be the Q-function. V_k and Q_k can be defined recursively. The value iteration algorithm starts with an arbitrary function V_0 and makes use of the following equations to get the functions for $k+1$ episode.

$$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma V_k(s')) \text{ for } k \geq 0 \tag{3.3}$$

$$V_k(s) = \max_a Q_k(s,a) \text{ for } k > 0 \tag{3.4}$$

Definition of the input variables, for the Value-Iteration algorithm:

- S, is defined as the set of all states.
- A, is defined as a set of actions for example: up, right, down, left.
- P, is defined as state transition function specifying $P(s_{dash}|s,a)$.
- R, is defined as the reward function $R(s,a,s_{dash})$.
- Theta, is a threshold $\theta > 0$, it is the termination condition of the algorithm.

Definition of the output variables, provided by the Value-Iteration algorithm:

- $pi[s]$, is defined as the approximately optimal policy $\pi(s)$, for state s.
- $V[s]$, is defined as the approximately optimal action-value function $V_\pi(s)$, for state s.

The pseudo code of the Value Iteration can be found below.

Listing 3.2: Pseudo code of the Value-Iteration algorithm

```
Value_Iteration(S,A,P,R, Theta)
Local
    real array Vk[S] is a sequence of value functions
    action array pi[S]
assign V0[S] arbitrarily
k <- 0
repeat
        k <- k + 1
        for each state s do
                Vk[s] = max(sum(
                  P(s_dash|s,a) (R(s,a,s_dash) + gamma * Vk-1[s_dash])))

until for all s abs(Vk[s] - Vk-1[s]) < Theta
for each state s do
        pi[s] = argmaxa(sum(P(s_dash|s,a) (R(s,a,s_dash)+ gamma*Vk[s_dash]))
return pi,Vk
```

The algorithm converges no matter what the initial value function V_0 is. A good initial value function, which is a close approximation of V^* clearly results in a much faster converges than a poorly estimated initial function. The common way for creating an initial function, is to use some heuristic methods or simply use random values, which are used as an initial seed for the value iteration.

Another version of the algorithm is the so called asynchronous value iteration. This version makes use of the multi-core processor technology which has been introduced by the processor vendors in the last years. Instead of sweeping through the states to create a new value function, the asynchronous version updates the states one at a time in any order, and stores the values in a single array. Ideally this algorithm converges faster than the synchronous version, but locking mechanisms may introduce some overhead and the number of states may influence the performance. The more states the model has, the more beneficial the asynchronous version is.
Implementation details and considerations which algorithm version for this thesis was chosen, are explained in the practical section of this document.

3.7 Monte Carlo Methods

In the previous chapter a policy based method has been shown to find the actions that provide the most reward. One possibility for learning such a policy is to use 'Value-Iteration', another option is to use 'Monte-Carlo' methods in particular 'Monte Carlo Tree Search' (MCTS).
The Value-Iteration algorithm is based on dynamic programming methods, this means it needs a complete probability distribution of all possible transitions. The assumption is made, that the model has complete knowledge of the environment. In contrast Monte-Carlo methods may only require sample sequences of states, actions and rewards from the environment. Monte-Carlo methods solve the RL problem based on averaging sample returns (or rewards), therefore Monte-Carlo methods must be used in episodic tasks. An episode is a sequence of state, action and reward tuples, that is obtained by using the model of the environment [13]. When an episode is finished the value estimates and policies are updated. MCTS algorithms build a tree of visited state-action pairs starting from the current state. Such a tree enables the algorithm to reuse knowledge from previously visited states. Simple MCTS algorithms take greedy actions to a specific depth in the tree, then the method takes random actions from there until the episode is finished. Value updates are only performed on states that have been visited between the agent's current state and the end of the so called roll-out. A roll-out is a trajectory of experience from the start state to the end of an episode or to a maximum search depth. This means at each state the algorithm selects an action, leading them to a next state, which is on level deeper in the tree. The action values get updated towards the received rewards following a trajectory that leads to a terminal. A terminal in this case maybe a state or the maximum depth of the tree. As already mentioned such methods can be more efficient due to the fact that dynamic programming methods need to plan over the complete state space [7].
Each roll-out of MCTS consists of four steps:

1. Selection: Starting from root R, recursively select child nodes until leaf node L is reached.
2. Expansion: If L is the terminal state, we are done, otherwise create one or more child nodes and select one C.
3. Simulation: Play a random play-out form node 'C' until a result is achieved.
4. Backpropagation: Update the move sequence (trajectory) with the simulated result.

The original MCTS algorithm can be used in all scenarios which have only positions with finitely many moves and finite length simulation time. In a state all possible moves are determined, for each one k-random simulations are performed until they reach terminal state or maximum depth, and the cumulative rewards are recorded. The action with the highest reward is chosen.

3.7.1 Upper-Confidence-Bound Action Selection

A problem in selection an action is the fact, that at the point when an certain action is selected, it is not clear if the chosen action is optimal. For example greedy actions look best at present, but some other actions may yield better overall results. One solution to this is to introduce the so called upper confidence bound (UCB) [14],

$$a_t = \text{argmax}[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}}] \tag{3.5}$$

where $Q_t(a)$ is the action-value estimate, $N_t(a)$ denotes as the number of times that action a has been selected prior to time t, c controls the degree of exploration whit $c > 0$. If $N_t(a) = 0$, then action a is considered to be a maximizing action. The square root term in the equation is a measure of the uncertainty in the estimate of the value of a. Each time the action a is selected the uncertainty is reduced, $N_t(a)$ is incremented and the overall uncertainty term is decreased. In the case that action a is not selected t is increasing which results in a higher uncertainty value. The $\log$ ensures that the increase gets smaller over time. The above formula is based on the one-armed bandit or slot machine problem, since a slot machine has only one state, no state dependent variables are necessary. This makes the formula easier to understand, the state dependent variables are introduced in the UCT algorithm subsection.

3.7.2 UCT (Upper-Confidence-Bound applied to trees)

The main problem with MCTS based algorithms is their runtime. The deeper the search tree gets, the slower the algorithm performs. To reduce this problem several methods have been introduced, one of these is the so called UCT algorithm [13]. UCT is a roll-out based algorithm that builds its tree by repeatedly sampling episodes from the start state. UCT selects it's actions based on upper confidence bounds using the UCB1 algorithm [14]. Essentially, MCTS + UCB1 yields the UCT algorithm.

3.7.3 The UCT Algorithm

The ideas of the previous section are now used to develop the UCT algorithm. Equation (3.3) can be adapted as follows:

$$a = \text{argmax}_a Q^d(s, a) + 2C_p\sqrt{\frac{\log(C(s, d))}{C(s, a, d)}} \tag{3.6}$$

where $Q^d(s, a)$ denotes as the action-value function at a certain depth d in the tree. C_p denotes a constant based on the reward range of the environment. $C(s, d)$ is a count of visits to each state s at a given depth d in the search tree. $C(s, a, d)$ is a count of the number of times an action a was taken from state s at depth d. By using the equation (3.4) the algorithm samples good actions, it also performs exploration when other actions have a higher upper confidence bound. The UCT algorithm, pseudo code can be seen below, starts from the agent's current state s, with a depth d of 0. Furthermore the algorithm needs a learning rate α and the variable R_{range} provides the value of one step rewards in the domain. The algorithm returns the updated Q-Value of the current state s in depth d of the tree. By closely looking at the pseudo code, one can see that the most important line is number six, where the algorithm recursively calls the UCT method, to sample an action at the next state one level deeper in the tree.

Listing 3.3: Pseudo code of the UCT algorithm

```
UCT (Inputs s, d, alpha, rrange)
1:  if TERMINAL or d == MAXDEPTH then
2:          return 0
3:  end if
```

```
4:  a <- argmax(Q_d(s, a_dash) + 2 * rrange * sqrt(log(c(s, d)) / c(s, a_dash, d)
    ))
5:  (s_dash, r) <- SAMPLENEXTSTATE(s, a)
6:  retval <- r + UCT(s_dash, d + 1, alpha)
7:  c(s, d) <- c(s, d) + 1
8:  c(s, a, d) <- c(s, a, d) + 1
9:  Q_d(s,a_dash) <- alpha * retval + (1 - alpha)* Q(s ,a_dash, d)
10: return retval
```

Some properties of MCTS based algorithm:

- As mentioned above MCTS algorithms have been proven to converge to the min-max evaluation, but the basic version of MCTS can take quite a long time to converge.
- The UCT version of the algorithm should reduce this problem.
- The algorithm generates asymmetrical trees, since the method concentrates on searching its more promising parts.
- Furthermore MCTS algorithms can be used in light or heavy play-outs. Light play-outs are based on random moves, while heavy play-outs use some kind of heuristics to choose what the best move is. The heuristics can be based on previous play-outs, or on some deep knowledge of the environment.
- A further improvement to the algorithm can be made by using domain specific knowledge while building the tree.

In this thesis some of the above mentioned methods are used to improve the performance, detailed description of the used methods will follow in the practical section.

3.8 Combining model and planning

Calculating a policy using a model is called planning [7]. In this chapter the interaction between the model and the planner is explained. There are many ways of how the two parties communicate with each other. In the following a few of them are highlighted.

- A common way of performing this communication is that the agent directly interacts with the environment, the model gets updated at every time step with the latest transition, $< s, a, r, s' >$. Every time the model gets updated (for example when a new state is added), the algorithm re-plans on, see 3.1. The disadvantage of this approach is that the learning of the model is very time consuming due to the computational complexity. On the other hand this method has the advantage to be very efficient in finding an optimal action from one state to another, once the model was learned. This is one reason why this method is used in the thesis. Details about the implementation and the performance of the algorithm will follow in the practical section.
- A way to reduce the computational complexity is to perform so called batch updates. This means that after k actions an agent performed, the model gets updated. The disadvantage of this approach is that the model is less accurate than the model produced by the previous suggestion. So the optimal action from on state to another is not always found.
- The following approach is somehow a combination of the first and second one. The so called DYNA [16] [17] algorithm is a reactive RL architecture. With this algorithm the agent starts either with a real action in the world or with saved experience. For example dynamic programming methods like Value-Iteration need knowledge of the complete state space. In contrast the DYNA

method randomly updates selected states. The DYNA algorithm performs the model updates and the action loop separately. This means in theory, fewer model updates are required than in the direct approach, explained in the first point, with an equal result in performance of the algorithm. Practical tests showed that nevertheless the DYNA algorithm requires many model based updates for the policy to become optimal, so the theoretical advantage of the method compared to the first method is not as significant as expected.

- Another alternative method, which is based on the DYNA algorithm is called Prioritized Sweeping [18]. It can be seen as an improvement of the DYNA method. The difference is that the selection to update a state action pair is handled by priority instead of doing it randomly. The algorithm performs the update based on the expected change in their value, instead of iterating over the entire state space. From a programming point of view this approach is very expensive and leads to dynamic programming techniques. By using dynamic programming a value function is computed for the entire state space after the model has changed. Once the value function was computed it needs then to be recomputed only when the model changes. In contrast to sample based methods like the first one which needs to perform computations on each and every state this approach is more efficient.
- The DYNA-2 [19] method is an advanced version of DYNA and extends the idea of updating its value function using both real and simulated experience to using sample based planning. The algorithm uses two different linear function approximators, one is used for the calculation of the permanent value function (long time memory) and the other is used to calculate the transient (short time memory) value function. The permanent value function is updated through real experiences in the world. Between the actions, the transient value function is updated using UCT on the agent's model of the world. The transient value function calculation is focused on a smaller or local part of the state space and it is used to enhance the global value function. This means that the selection of an action is based on the transient and permanent value functions. DYNA-2 has more or less the same problems as the previous methods, the model update and planning can take a significant amount of time.
- The threaded architecture [15] for real-time model based reinforcement learning uses the advantage of multi-threading to put model learning, planning and acting in three different threads. In such multi-threaded architectures the communication between the different threads is very important. For this purpose four shared data structures are used. All data structures have to be thread safe to work as expected. This is very important, without these protections the outcome of this method is in best case completely random and not deterministic. The 'list of experiences' to be added to the model is the first shared data structure. The other data structures contain the model which is used by the planner, the current state of the planner and the policy of the agent. The model learner thread runs in one loop, the same is true for the planning thread. It is important to note here that some kind of synchronization between the two threads is necessary, otherwise one of the threads may run out of data. The model learner removes experiences from the 'list of new experiences' and updates the model with these experiences. The planning thread then uses these updated experiences provided by the synchronized data structure to run a sample based planning algorithm like UCT. The action thread adds the latest experience to the update list, sets the agents current state and returns the best action for that state. This approach is very flexible, for example the action thread is able to return an action at every update frequency that is required. By using this method it is possible to simulate a real-time behaviour of the agent.

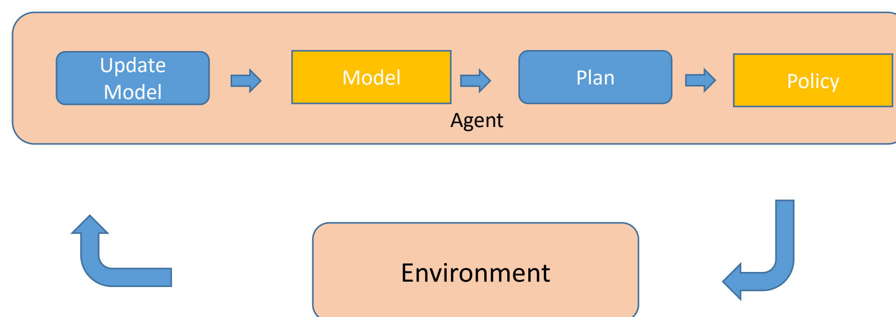

Abbildung 3.1: The typical interaction between the agent and the environment.

Kapitel 4

Feature Extraction Practical Section

In this section we describe in detail how the feature extraction was realized. As already mentioned for feature extraction the Slow Feature Analysis (SFA) algorithm was used. The following description outlines how the algorithm was implemented and which design decisions were made to provide a fast and scalable implementation. To guarantee that the SFA algorithm was implemented correctly, a reference version of the algorithm in Python was used which also served as a benchmark for correctness and performance of out implementation. Techniques like Test Driven Development (TDD) were used to guarantee that the implementation is done in the right way. Since the algorithm is an optimization problem the memory consumption is a mayor concern. Especially when dealing with a large amount of data like in image processing, some 'tricks' must be used. All this considerations will be described in the following sections. Since SFA is mainly based on 'simple' linear algebra, it is crucial to perform operations like matrix multiplications or eigenvalue and eigenvector computations very efficiently. A third party library was used which is highly optimized in the sense of using hardware features provided by modern CPUs. We also considered to use the GPU for such calculations, but the implementation effort to realize an implementation which is more powerful than the CPU calculation was not feasible. Due to the huge amount of data, it was not possible to process all the data with a single SFA. In our case a so called 'Hierarchical SFA network' is necessary to process high dimensional data like them which are used in this thesis. The data structures which are needed to create such networks, are also explained in the following sections, to make things easier, the high level constructions of such networks are very similar to the reference Python implementation. For feature extraction problems like the one in this thesis, it makes sense to put very little Gaussian noise on top of the input data, this helps to avoid the singular matrix problem when solving the eigenvalue and eigenvector problem which is used by the SFA. For such cases an additional data structure is used, a so called SFA-Node, this allows the user of the SFA algorithm to compose different pre and post calculation (like adding noise, quadratic expansion and clipping) which often improves the result of the SFA. The most important section of this chapter is dedicated to the tuning of the parameters, which results in an optimal feature set, therefore a large set of training and test data were used to find the correct parameters for example for the Gaussian noise, the clipping, the receptive field size and so on.
An important part was also the creation of the training and test data for the SFA, but since this is more a programming related problem than an algorithmic or thesis related one, the section is rather brief. All the relevant code can be found attached to this thesis, it is written completely in C++ (C++11/C++14) for platform independence, the GUI elements (including the 3D environment) are created using Qt 5.5 [20]. For linear algebra operations the open source Eigen-Library [21] was used. The complete feature extraction process can be performed in parallel, for this a also free available library called 'Threading Building Block' (TBB) [22] which was created by Intel was used. For high level matrix calculations which are not time critical, the matrix library from boost was used, since this library is open source and highly adaptable due to the heavily used 'Generic Programming' model, it was a perfect choice to solve matrix related problems.

4.1 Slow Feature Analysis (SFA)

In this section we outline how the SFA algorithm was implemented and how it differs from the original version which was introduced in 2. We also illustrate how the SFA algorithm was tested how it compares to the reference implementation. For this purpose the Matlab-like pseudo code from the reference and the code from real implemented is compared. During this comparison the differences are highlighted and reason why they differ is explained.

4.1.1 Simple implementation

The reference implementation of the SFA algorithm follows the suggestions and the pseudo code of the paper [4]. This simple version and implementation of the SFA algorithm were then used as a starting point to develop a more sophisticated version of it, which will then be described in the next section.

The source code in listing 4.1, shows a naive implementation of the SFA algorithm, which is based on the example in the theory section of this thesis. The listing is separated in different sections which are marked by comments. These comments are used to describe what the different parts of the code are doing.

In block 1 the input signals for the algorithm are generated, two sources are used $\widetilde{x}_1(t) := sin(t) + cos^2(11t)$ (see 4.1) and $\widetilde{x}_2(t) := cos(11t)$ (see 4.2), these signals are in the range of $[0, 2\pi]$ and vary very quickly. The two signals which have been created above, are not direct inputs to the algorithm, instead these signals will be mixed to generate one fast varying signal (see 4.3), the algorithm should then find the slowest varying signal which is common to both of the input signals $x1$ and $x2$. At the end of most of the blocks a plot is drawn, to show how the code influences the signals. In the following all the remaining sections in the code will be described and references to the theory section will be made to exactly show how the algorithm was implemented.

In block 2 of listing 4.1, the normalization of the input signals is performed, this procedure full fills the constrains of zero mean and unit variance. The results of these operations can be seen in 4.4.

In block 3 of listing 4.1, the non-linear expansion takes place, this simple implementation is only able to perform a so called quadratic SFA which means a quadratic expansion is performed, for this thesis also the linear SFA is necessary. The non-linear expansion then results in the following output 4.5.

In block 4 the normalization of the expanded signal is performed, this process is also known as sphering or whitening. Sphering can be done by using the Principal Component Analysis (PCA) which was already mentioned in the theory section, but for simplicity this calculation is performed manually using the 'eig' command. As it will be shown in the advanced version of the algorithm (where the PCA is used), PCA provides more stable results and also gives an advantage in the sense of performance over the naive algorithm. Since there is no need to sort the eigen-values when using the PCA. The signal after sphering looks like the following 4.6.

In block 5 the temporal variation in the normalized space is measured. This is done by performing a time derivation of the sphered signal $z(t)$. This step is necessary to determine the direction of minimal variance or the slowest varying signal. The result of the deviation can be seen in figure 4.7.

In block 6 the slowest varying directions of the previously determined directions is extracted. This means finding the direction of least variance in the time derived signals. Again this can be done by using the PCA, the directions are then principal components with the smallest eigenvalues. As already mentioned, due to simplicity,this approach is again done manually by using the 'cov' and the 'eig' com-

mands. Conveniently the 'eig' function yields sorted eigenvalues so that the first eigenvector is equal to the weight vector with the minimum variance. The final step is then to project the sphered signal (see section 4) on to the direction of the smallest variance, which result in the slowest signal generated by the two input signals. The result $y(t)$ can then be seen in figure 4.8.

4.1.2 Advanced implementation

As already mentioned in the previous section, one of the problems with the simple implementation of the SFA algorithm, is that it only supports the quadratic version of the SFA (Quadratic SFA). For the problem that needs to be solved in this thesis the liner version of the SFA is also required. To solve this problem, the advanced version of the algorithm was implemented in a way to support both, linear and quadratic SFA. A further improvement of the advanced version is, that the algorithm is able to perform a training phase as well as an execution phase. This means in the training phase a large training-set is provided to the algorithm, to learn the important features. In the execution phase a smaller test-set is provided to the algorithm, the important features learned in training phase are then used to extract the slow features of the test-set. This means the algorithm is once trained with a large trainings-set (which can take a substantial amount of time) and than executed on any test-set to provide a fast and very good approximation, since there is no need to solve the computational expensive eigenvalue problem in the execution phase.

Linear Slow Feature Analysis

The source code in 4.2, shows the linear version of the SFA algorithm. The listing is separated in different blocks which are marked by comments. These comments are used to describe what the different parts of the code is doing.

In section 1 the covariance matrix of the input signal is calculated. This version of the algorithm takes a few short cuts compared to the simple version, for example the normalization of the input signal is skipped, this works as long as a numerical stable version of the algorithm for calculation the covariance matrix is used. The algorithm used to calculate the covariance matrix is called 'Sample mean and sample covariance' [23]. Also no expansion of the input signal is performed since the liner version of the SFA is described here. The calculation of the covariance matrix also allows us to skip the explicit 'Sphering' calculation which was done in the simple version.

In block 2 the temporal variation of the input signal is calculated. In the simple version of the algorithm the differentiation was done by a simple numeric approximation, in this version the Matlab integrated filter function with the filter coefficients $1, -1$ was used to perform the calculation.

In block 3 the slowest varying signal is extracted. The Matlab integrated 'eig' function which takes two parameters and is known as eigenvalue-decomposition is used to perform this step. The function takes the temporal variation (section 2) as first parameter and the covariance matrix (section 1) as second. With this approach it is possible to perform two calculations (Sphering and slowest varying signal) in one step. This is an advantage compared to the simple version, since fewer calculation steps are needed, and the method is generic in the sense of the size of the input signal. After the eigenvalues and eigenvectors have been calculated, they are sorted in descenting order, so that the smallest eigenvalue and the associated eigenvector are stored at position one of the respective two vectors.

In block 4 the eigenvectors are normalized between -1 and 1. After the normalization, the eigenvectors are stored in the $eigen_vectors$ variable which is return by the function as well as the eigenvalues.

Next, we outline the execution phase of the algorithm. As mentioned above this part of the implementation is executed after the trainings phase. The execution phase produces the output signal $y(t)$ which represents the projection of sphered signal on to the direction of the smallest variance, which was already explained in the theory part and in the previous section. The source code in 4.3, shows the Matlab implementation of the execution phase. Since the code is very simple and also very short, the description won't be split in different sections, instead the implementation will be described in the following. The 'LinearSFAExecute' function takes two parameters, the first one is the input signal, and the second one are the pre-calculated eigenvectors from the training phase. The input is then normalized to zero mean and unit variance and then the output function is calculated. Since these calculations are not very computational expensive, the performance compared to the simple version is better and when implemented in a correct way this version can be used for any kind of input data for example gray scale images, color images, ... the only limitation comes from the computer on which the algorithm is executed.

Quadratic Slow Feature Analysis

In this section the implementation of the Quadratic SFA is explained. As already explained to solve the problems in this thesis Linear SFA and Quadratic SFA are used, the difference between the two types is more or less the non-linear signal expansion. In the Linear SFA no expansion is used, the Quadratic SFA uses polynomials of degree two to expand the input. Putting the input signal in a higher dimensional space makes it easier to find a good separation of the different features, this can be compared with the so called 'Kernel-Trick' used by Support-Vector-Machines. The problem with the quadratic version is that the memory usage compared to the linear version increases dramatically, which means it is limited due to availability of memory, on the machine that we used. The machine we used to create the result for this Thesis has 16 GB of RAM. It was not possible to use all CPU-Cores to parallel calculate as many Quadratic SFAs as cores are available. This is a massive disadvantage of the SFA, but this problem usually occurs during training phase not during execution phase. The training phase is critical because a huge amount of data is necessary to find a good set of weight functions (eigenvectors) that can be used during execution phase. Nevertheless all the problems also occurred during the creation of this thesis and have been solved, some of them elegant and some of them not so elegant...

The source code in 4.4, shows the quadratic version of the SFA algorithm. The listing is separated in different sections which are marked by comments. These comments are used to describe what the different parts of the code are doing.

In block 1 the covariance matrix of the input signal is calculated. This step is actually the same approach as for the linear version of the SFA (see 4.1.2 section 1).

In block 2 the so called whitening is performed. This approach can be seen as normalization of the under section 1 calculated covariance matrix. The whitening matrix is obtained by executing the Principal Component Analysis (PCA) on the covariance matrix. This whitening matrix represents the principal component coefficients of the covariance matrix.

In block 3 the quadratic expansion of the normalized input signal is calculated. First of all the mean of the input signal is subtracted from the input signal, this satisfies the constraint of the SFA algorithm that the input signal must have zero mean. Then the zero mean subtracted input signal is multiplied with the whitening matrix to satisfy the constraint of unit variance which is also required by the SFA algorithm. The last step is to perform the quadratic expansion like it is explained in the theory section.

In block 4 the covariance matrix of the expanded input signal is calculated as well as the mean of the expanded input signal, the mean will then be used later in the execution part of the quadratic SFA.

In block 5 the temporal variance of the expanded input signal is calculated. It is exactly the same approach as in the linear version of the SFA, using the filter function provided by Matlab allows to calculate a numerically stable time derivative of the expanded input signal.

The blocks 6 and 7 do exactly the same as the sections 3 and 4 of the linear version, the only difference is that in the quadratic version the expanded signal is used instead of the normal input signal in the linear version.

In the following the execution phase of the algorithm is explained, as mentioned above this part of the implementation is executed after the trainings phase. The execution phase then produces the output signal $y(t)$ which represents the projection of sphered signal onto the direction of the smallest variance, which was already explained in the theory part and in the previous section. The source code in 4.5, shows the Matlab implementation of the execution phase. Since the code is pretty short and also not too hard to understand, the separation in different sections was skipped. The function 'QuadraticSFAExecute' has five input parameters:

1. input: The input signal
2. eigenvectors: The eigenvectors calculated in training phase
3. whitening matrix: The whitening matrix calculated using the PCA during the training phase
4. mean of input: The mean of the input signal
5. mean of expanded input: Them mean of the non linear expansion calculated during the training phase

The first line of the Matlab source code performs the normalization of the input signal using the whitening matrix determined during the training phase. In the next line the non-linear expansion of the normalized input signal is performed, this is the same step as in the training phase and the same calculation is used.
Then the expanded input signal is normalized to again satisfy the constrains of zero mean and unit variance.
The final step is then to calculate the output signal $y(t)$ which represents the projection of the sphered signal on to the direction of the smallest variance.
As it can be seen the execution phase does not contain computationally expensive calculations therefore the performance of this phase is pretty good. It should also be mentioned that the memory consumption of this code is by far not as high as during the training phase, this means we can use multi-threading to further increase the performance of this algorithm.

4.2 Hierarchical SFA networks for high dimensional data

This chapter describes how to create a so called 'Hierarchical SFA network'. The problem with high dimensional data like in image processing is the handling of the very large amount of data. Even with modern computers it is not possible to perform a feature extraction (like the SFA) in one single shot. For example a RGB image with a resolution of 300×300 pixels consumes 270.000 slots of an array, for feature extraction algorithms it is recommended to use precise data types like float or double (common types for programming languages like C/C++, C# or Matlab), on 64-bit machine this means every single slot of the above array reserves 8 bytes. This means for one single image we use 2.160.000 bytes. Since we need between 100.000 and 150.000 images during training, with all the processing like expansion, eigenvalue, eigenvector calculations and so on, we can not train the algorithm with one single instance of a SFA, because of the limitations of the computer where the algorithm is executed on. One of the possible solutions is to create a 'Hierarchical SFA network' [5]. The network processes only small chunks of an image for example a region of 10×10 pixels, this is the so called 'receptive field' of the SFA. Using this technique it is much easier to perform a feature extraction on almost any resolution of an input image sequence. Sadly it is not sufficient to only use the SFA/SFA2 algorithm in the calculation nodes, this would lead to incorrect features or even worse 'singular matrix' errors, so some pre-processing as well as some post-processing is necessary to get reasonable results. In the following the post- and pre-processing algorithms are explained.
As already mentioned one node does not only contain one algorithm like the SFA, instead it contains up to six different algorithms to perform the correct feature extraction of the image chunk (receptive-field).

1. Additive-Noise: This step generates Additive-Gaussian-Noise. This means that noise is added to the input signal, which helps to avoid problems like 'singular matrix' errors during eigenvalue calculation. This error can occur if the pixels within a receptive-field have the same values, adding random noise to these values reduces that chance to run in this problem. The noise can be adjusted by changing the mean and deviation of it.

2. Linear-Slow-Feature-Analysis: Please refer to 4.1.2.

3. Quadratic-Slow-Feature-Analysis: Please refer to 4.1.2.

4. Clipping: This algorithm limits the input signal to specified maximum and minimum value. So for example if a input signal has a peak with the value of 10 and the maximum value given to the algorithm is 4, the peak is reduced to 4. The same is true for minimum values.

5. Independent Component Analysis (ICA): This algorithm is most likely used in the final step of the hierarchy, this means it is a so called post-processing step, which allows to reduce the feature space to really important and independent features. This means all features appear only once after the algorithm was executed.

4.3 Simple 2D hierarchical SFA network

In this chapter a proof of concept of the presented algorithms and structures is given. The objective is to extract the relevant features of a simple 2-D maze (see 4.9) The maze has a size of 60×60 pixels, the green pixel in the upper left corner is the agent, the two gray boxes are considered to be walls, which means that they are obstacles to the agent. The red pixel is the destination which should be reached by the agent, for the feature extraction it is not important how to reach the finish point, this is explained in the model learner chapter under 3. The agent explores the maze randomly, every pixel the agent moves generates one image, this means for example when the agent moves 10 steps to the right, 10 images are generated. For the first test 100.000 images have been generated, this seems to be a large number, but the SFA algorithm does not perform very well with fewer images. The generated images of this maze are then saved as simple binary images which reduces the calculation effort, that's why the hierarchical SFA output can be calculated pretty quickly and adaptations to the algorithm itself and to the parameters of the algorithm can be done easily. The structure of the hierarchical network, that was used to perform these test can be seen in figure 4.18.
Quite a lot of parameters needs to be adapted to find a solution that produces features of high quality. Luckily there exists very good papers like [25], which can help a lot to find the correct network structure, the Gaussian-noise and the clipping parameters.
As reference for the completed implementation the 'Modular toolkit for Data Processing' [27] was used. The structure shown in this thesis is based on this toolkit and tries to use the same structure for the nodes and the network.

4.3.1 Node structure

As already mentioned earlier in this chapter the SFA network is assembled using the so called SFA-Nodes. Each SFA-Node contains different algorithms which are executed sequential on a section of the image (receptive field). The node structure which was used to extract the features of the simple 2-D maze contains the following algorithms:

4.3.2 Network structure

The network contains four layers see figure 4.18, the first layer directly maps to input image, the last layer (4) provides the extracted features which are then used for further processing.

1. Layer 1: As already shown (see figure 4.9), the input images has 60×60 pixels, the receptive field has a size of 8×8 pixels with an overlap of 4 pixels and a channel dimension of 1. The channel dimension for example is used to distinguish between binary images which have only one color information (0 or 1) 'channel dimension = 1' or rgb images which have three color information 'channel dimension = 3'. The output of the first layer contains 32 features, these output serves as input to the next layer.

2. Layer 2: Due to the dimensional reduction of layer 1 the input images has the size of 14×14 pixels. The receptive field of this layer has as size of 4×4 with an overlap of 2 pixels, the channel dimension is set to 32 since the output of layer 1 gives us 32 features. The output of this layer provides again 32 features.

3. Layer 3: The images size now is reduced to 6×6 pixels. The receptive field size of this layer has now 4×4 pixels with an overlap of 2 pixels. The channel dimension is again set to 32 since the output of the previous layer contains 32 features. As the previous layer the output provides 32 features.

4. Layer 4: The final layer of the network, gets an input of size 2×2 pixels which results in a receptive field size of 2×2 and no overlap. The output of the final layer contains 64 features. This should be sufficient for further processing like ICA, to provide good data for model based algorithms.

4.3.3 Results

In this section some training and test results are presented and explained, methods like linear regression were used to generate these results, for this purpose Matlab was used. All the relevant algorithms and graphical tools are directly integrated and available. In the following we distinguish between 32 features and 64 features generated in the last layer of the network. The number of features is the most important parameter, since the difference can be significant. For example it may be sufficient to use 32 features and the test results confirm this, in a later step where a discretization of the feature space is necessary, the process simply fails because too few features were used. Such problems are very unfortunate since a lot of time is wasted, but there is no 100% reliable measure to be sure that the generated data are good enough for further processing. Therefore the decision was made of using too many features even when the test process was successfully with fewer features. It will be also shown that the noise parameters are crucial for good test results, the advantage in finding the noise parameters compared to the number of features is that a wrong noise parameter can be found during the test process. To generate the training data the agent performs random exploration of the maze (see figure 4.9), this was already mentioned, for evaluation of the test data the agent performs a systematic walk through the maze (see figure 4.9). In our case this means the agent moves from left to right and from top to bottom. Using this method we can ensure that every point of the maze has been visited which is very important to evaluate the trained SFA network. To perform the tests the linear regression algorithm was used (see listing 4.6), this procedure is simple and fast. As input to the algorithm the path of the agent through the maze is required and the extracted features calculated by the SFA network as well. Then the linear regression is calculated and the output is compared to the real path of the agent through the maze. The results are plotted separately for the x and y coordinates, the closer the two lines (real path and path calculated by the SFA network) are, the better the result, this can also be measured via the mean square error (MSE).

4.3.4 Test on the Small Feature Set

In this section the final layer (4) of the SFA network produces 32 features, all the predecessor layers also generate 32 features. The first plot (see figure 4.10), shows the evaluation of the training data in x-direction. This means the training path in x-direction of the agent is compared to the path calculated by the SFA network also in x-direction. To keep the plot simple and clear only the first 1000 time frames are shown in this plot. The second plot (see figure 4.11), shows the evaluation of the training data in y-direction. In this plot as we also only show 1000 frames. As it can be seen the results line up nicely which means the parameters of the Additive-Noise algorithm and the number of features used in all four layers are well chosen.

The node and network structure stays untouched, so it's the same as for the training section. The difference is that the input data are much smaller than for the training phase. During the test phase only 2820 time frames are used compared to the 100.000 during the training phase. The result for the x-coordinate can be seen in figure 4.12, this shows that the test results are not as good as expected the spikes between time frame 50 and 75 is very high and not only this single spike can be seen, we have a lot of them going to time frame 600. This indicates that something is wrong with the network (maybe too less features or not sufficient enough training data). For the y-coordinate see figure 4.13, the situation is slightly better, but miss classifications can also bee seen during the first time frames.

4.3.5 Test on the Large Feature Set

Based on our observations about the test results in the previous section some changes have been made to the SFA network. In this section the last layer (4) of the SFA network produces 64 instead of 32 features, all the predecessor layers stay the same and generate 32 features. All the input data also stay the same. In figure 4.14, we can see small improvements over the 32 features version, the calculated path of the SFA network follows the real path of the agent much closer than in the 32 feature version. The same is true for y-coordinate, see figure 4.15, so this network structure now looks much more promising than the one in the previous section.

As before the node and network structure stays untouched, so it's the same as for the training section. The test set contains 2820 images or time frames. The result for the x-coordinate can be seen in figure 4.16, the spikes are smaller than before and even the huge spike between time frame 50 and 75 was reduced by 50% the classification was improved so the SFA network follows the path of the agent much better than in the 32 feature version. The same is true for the y-coordinate which can be seen in figure 4.17, the path calculated by the SFA network is almost the same as the real path which was taken by the agent.

4.3.6 Additive Noise

Usually these parameters can be adapted easier than the network topology. Choosing the wrong parameters for the Gaussian noise usually can be seen during the training phase when a singular matrix error occurs, or later during training data evaluation when the linear regression fails. An example of an incorrectly selected noise value effects the evaluation can can be found in the 3D section of this thesis.

Matlab Code

Listing 4.1: Simple reference implementation of the SFA algorithm

```
1  clear all;
2  close all;
3  clc;
4
5  % 1. Input signal:
6  t = 0:0.01:2*pi;
7  x1w = sin(t) + power(cos(11*t),2);
8  x2w = cos(11*t);
9  figure; plot(x2w, x1w);
10 title('SFA Mixed input signal');
11 xlabel('time');
12 ylabel('amplidude');
13
14 % 2. Input signal normalization:
15 meanx1w = mean(x1w);
16 meanx2w = mean(x2w);
17 x1 = (x1w - meanx1w)/(sqrt(mean(power((x1w-meanx1w),2))));
18 x2 = (x2w - meanx2w)/(sqrt(mean(power((x2w-meanx2w),2))));
19 figure; plot(x2, x1);
20 title('SFA Normalized input signal');
21 xlabel('time');
22 ylabel('amplidude');
23
24 % 3. Non-linear expansion:
25 hw = [x1; x2; x2.*x2];
26 figure; plot3(hw(2,:),hw(3,:),hw(1,:));
```

```
27 title('Non-linear expansion');
28 xlabel('HW1(t)');
29 ylabel('HW2(t)');
30 zlabel('HW3(t)');
31
32
33 % 4. Sphering
34 for i=1:size(hw,1)
35  hw(i,:) = hw(i,:)-mean(hw(i,:));
36 end
37 C = cov(hw');
38 [V,D] = eig(C);
39 S = inv(sqrt(D))*V';
40 z = S*hw;
41 figure; plot3(z(2,:),z(3,:),z(1,:));
42 title('Sphered signal');
43 xlabel('Z1(t)');
44 ylabel('Z2(t)');
45 zlabel('Z3(t)');
46
47 % 5. Temporal variation
48 z_dot = z(:, 2:end) - z(:, 1:end - 1);
49 figure; plot3(z_dot(2, :), z_dot(3, :), z_dot(1, :));
50 title('Time derived signal');
51 xlabel('Z1(t)');
52 ylabel('Z2(t)');
53 zlabel('Z3(t)');
54
55 % 6. The slowest signal is extracted
56 C = cov(z_dot');
57 [V,D] = eig(C)
58 w = V(:,1);
59 g = w'*z;
60 figure; plot(g)
61 title('Output signal');
62 xlabel('t');
63 ylabel('y(t)');
```

Listing 4.2: Linear SFA train reference implementation

```
1 function [ eigen_values, eigen_vectors ] = LinearSFATrain(input)
2
3 n = size(input, 2);
4 sfa_range = 1:n;
5
6 % 1. Calculate the covariance matrix of the input
7 covariance_matrix = cov(input);
8
9 % 2. Perform time differentation of the imput signal
10 filtered_input = filter([1 -1], 1, input);
11 filtered_input = filtered_input(2:size(filtered_input, 1), :);
12 differentiated_input_signal = filtered_input' * filtered_input;
13
14 % 3. Determine the eigen-values and eigen-vectors
15 [v, d] = eig(differentiated_input_signal, covariance_matrix, 'qz');
16
17 eigen_values = diag(d);
18 [values, idx] = sort(eigen_values);
19 eigen_values = values;
20 eigen_vectors = v(:, idx(sfa_range))';
21
22
```

```
23 % 4. Normalize eigen-vectors between [-1, 1]
24 [rows, ~] = size(eigen_vectors);
25 colMax = max(abs(eigen_vectors), [], 1);
26 eigen_vectors = eigen_vectors./repmat(colMax, rows, 1);
27 end
```

Listing 4.3: Linear SFA execute reference implementation

```
1 function [ y ] = LinearSFAExecute(input, eigen_vectors)
2
3 mean_of_input = sum(input) / size(input, 1);
4 normalized_input = input - repmat(mean_of_input, size(input, 1), 1);
5 y = normalized_input * eigen_vectors';
6
7 end
```

Listing 4.4: Quadratic SFA train reference implementation

```
1 function [eigen_values, eigen_vectors, whitening_matrix, mean_of_input ,
      mean_of_expanded_input] = QuadraticSFATrain(input)
2
3 n = xp_dim(size(input,2));
4 sfa_range = 1:n;
5
6 % 1. Calculate the covariance matrix of the input
7 covariance_matrix = cov(input);
8
9 % 2. Perform principal component analysis & whitening
10 [whitening_matrix, ~, ~, ~] = PCA(covariance_matrix);
11
12 % 3. Perform expansion
13 mean_of_input = sum(input) / size(input, 1);
14 expanded_input = input - repmat(mean_of_input, size(input, 1), 1);
15 temp = expanded_input * whitening_matrix';
16 expanded_input = QuadraticExpansion(temp);%cat(2, temp, temp.^4);
17
18 % 4. Calculate the covariance matrix of the expanded input
19 expaned_input_covariance_matrix = cov(expanded_input);
20 mean_of_expanded_input = sum(expanded_input) / size(expanded_input, 1);
21
22 % 5. perform time differentation of the imput signal
23 filtered_input = filter([1 -1], 1, expanded_input);
24 filtered_input = filtered_input(2:size(filtered_input, 1), :);
25 differentiated_input_signal = filtered_input' * filtered_input;
26
27 % 6. Get the eigen-values and eigen-vectors
28 [v, d] = eig(differentiated_input_signal, expaned_input_covariance_matrix);
29
30 eigen_values = diag(d);
31 [values, idx] = sort(eigen_values);
32 eigen_values = values;
33 eigen_vectors = v(:, idx(sfa_range))';
34
35 % 7. Normalize eigen-vectors between [-1, 1]
36 [rows, ~] = size(eigen_vectors);
37 colMax = max(abs(eigen_vectors), [], 1);
38 eigen_vectors = eigen_vectors./repmat(colMax, rows, 1);
39 end
```

Listing 4.5: Quadratic SFA execute reference implementation

```
function [ y ] = QuadraticSFAExecute(input, eigen_vectors, whitening_matrix,
    mean_of_input ,mean_of_expanded_input)

normalized_input = (input - repmat(mean_of_input, size(input, 1), 1)) *
    whitening_matrix';
expanded_input = QuadraticExpansion(normalized_input);
normalized_expanded_input = expanded_input - repmat(mean_of_expanded_input, size(
    expanded_input, 1), 1);
y = normalized_expanded_input * eigen_vectors';

end
```

Listing 4.6: Linear Regression, to evaluate the SFA network

```
clear all;
close all;
clc;

timeFrames = 3255;

trainPath = textread('TestPath_2D.txt');
trainData = textread('TestOutput_2D_64_C++.txt');

trainData = trainData(1:timeFrames,1:64);
trainPath = trainPath(1:timeFrames,1:2);

SFA_train = [trainData, ones(size(trainData,1),1)];
wx = pinv(SFA_train)*trainPath(:,1);
wy = pinv(SFA_train)*trainPath(:,2);

x_t = trainPath(:,1);
y_t = trainPath(:,2);

x_pred = SFA_train*wx;
y_pred = SFA_train*wy;

mse_x = mean(mean((x_pred-x_t).^2))
mse_y = mean(mean((y_pred-y_t).^2))

figure; plot(x_pred(1:timeFrames));
hold on; plot(x_t(1:timeFrames),'r')

figure; plot(y_pred(1:timeFrames))
hold on; plot(y_t(1:timeFrames),'r')
```

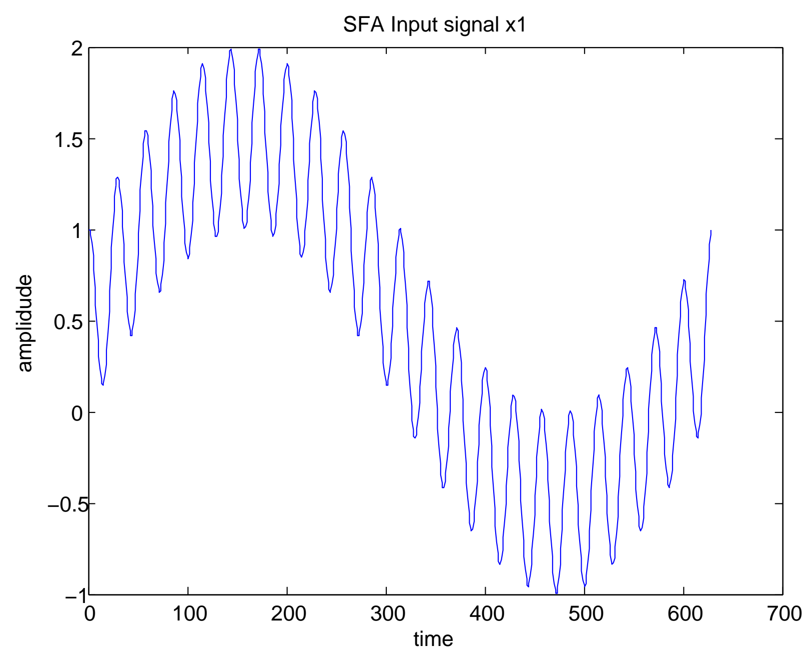

Abbildung 4.1: The first input signal.

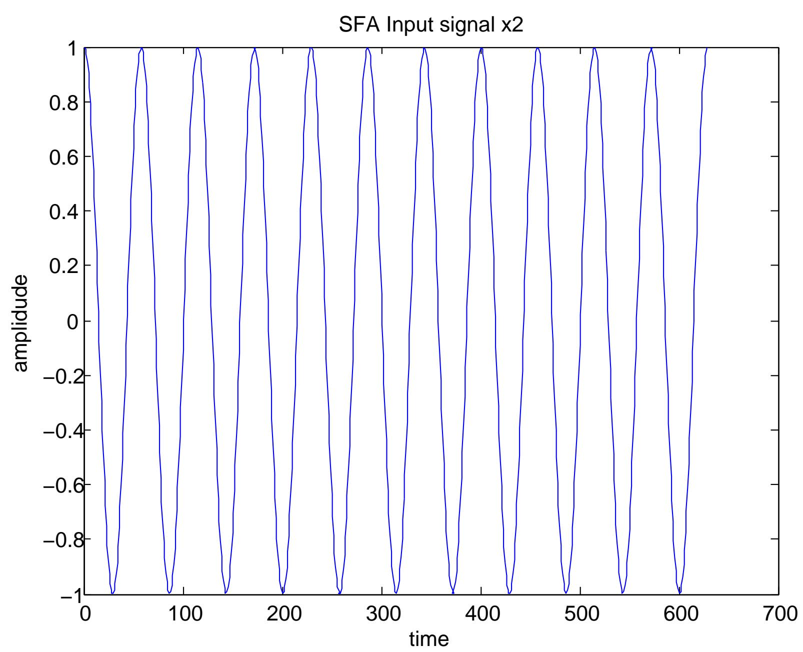

Abbildung 4.2: The second input signal.

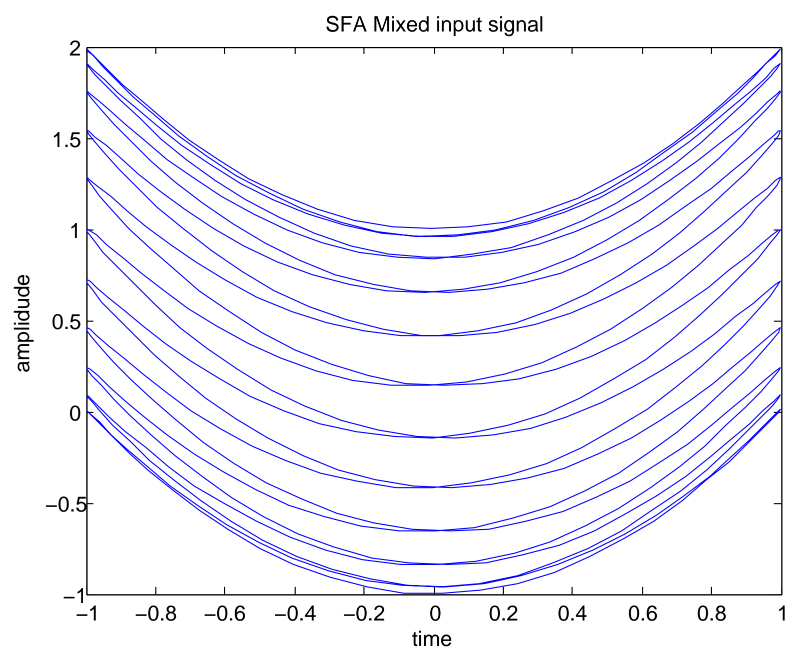

Abbildung 4.3: The mixed input, which is then used by the algorithm.

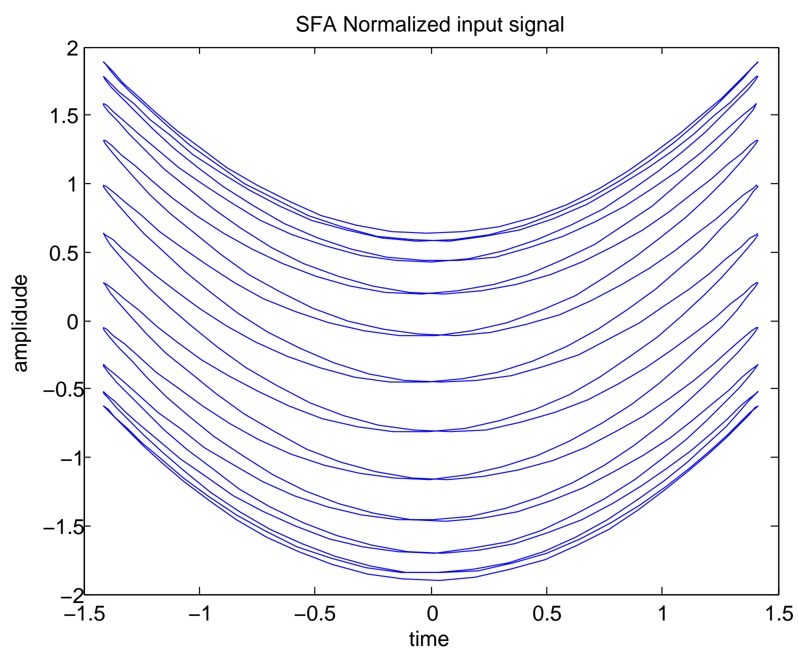

Abbildung 4.4: The normalized version of the input signal.

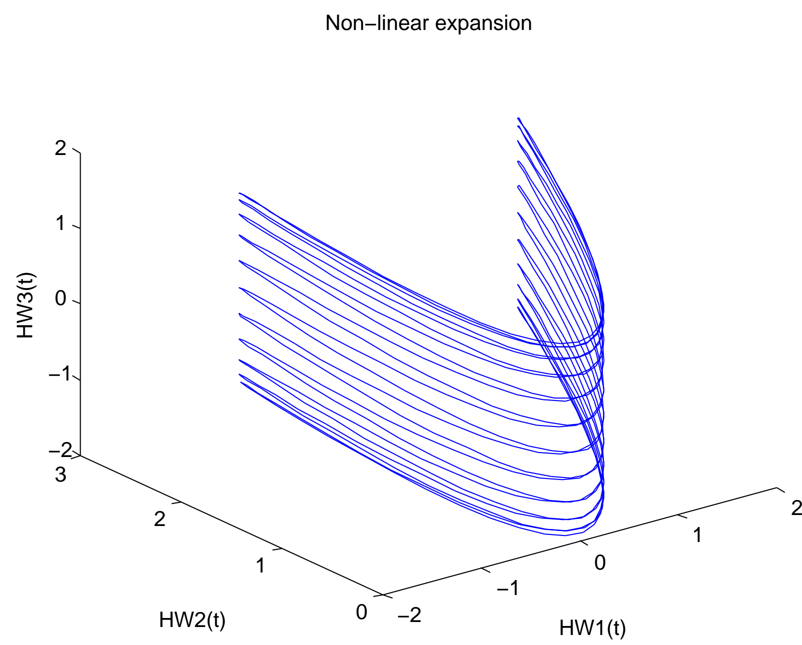

Abbildung 4.5: The non-linear expansion of the signal.

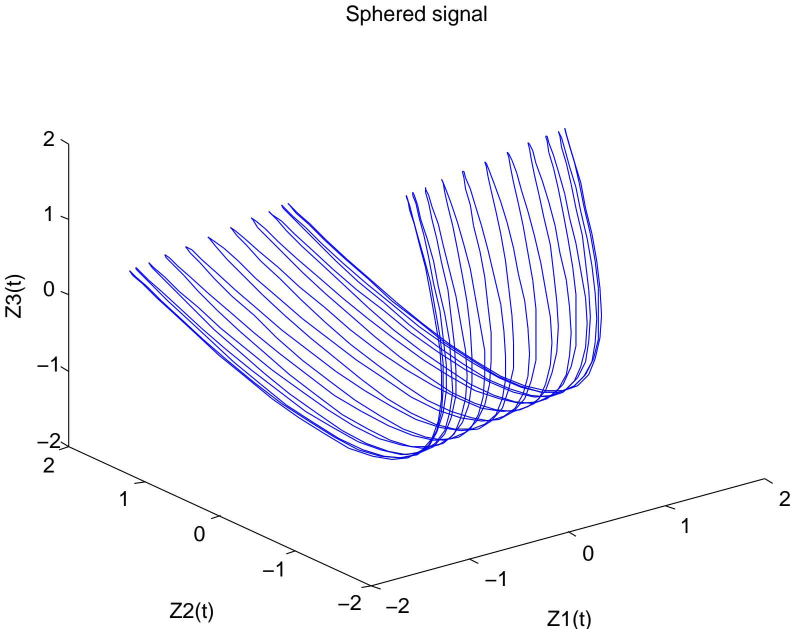

Abbildung 4.6: The sphered signal.

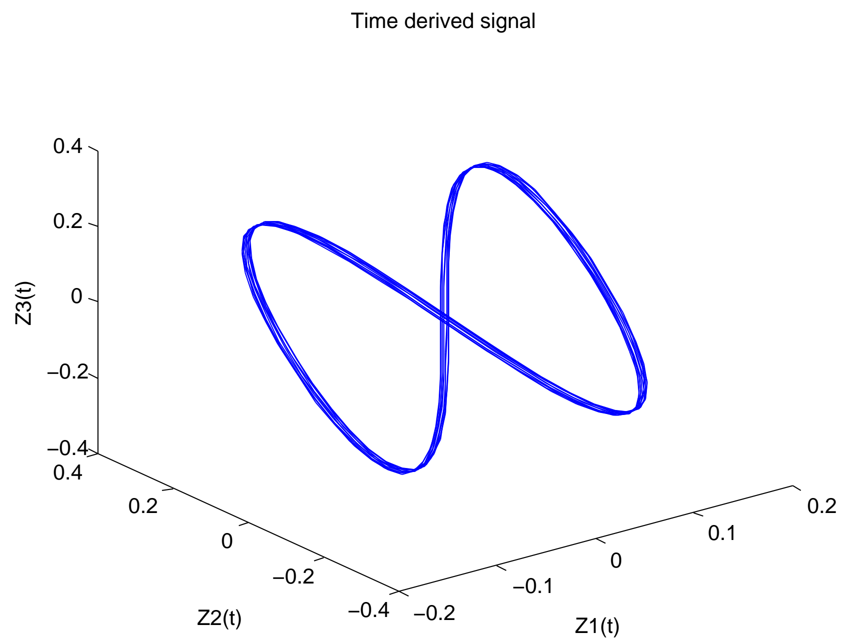

Abbildung 4.7: The time derivative of the sphered signal.

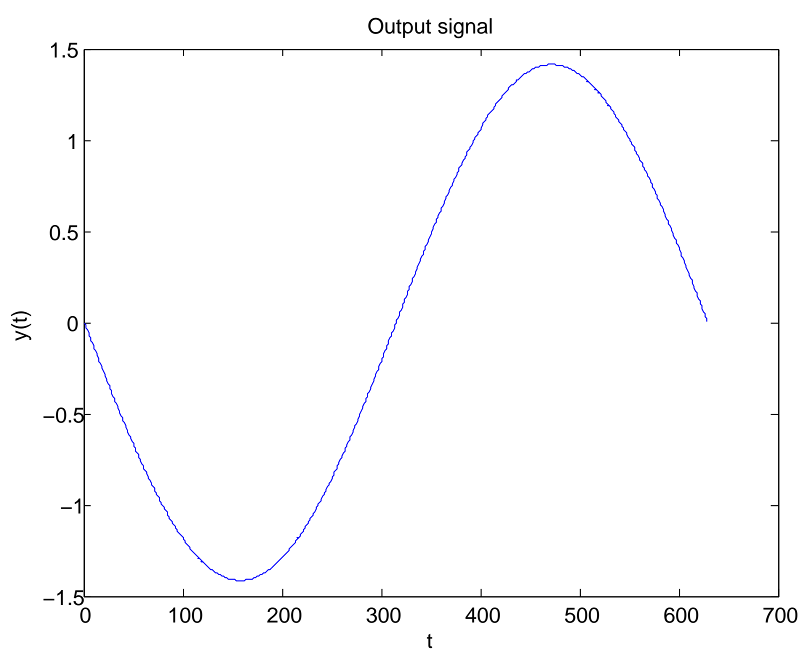

Abbildung 4.8: The slowest varying output signal.

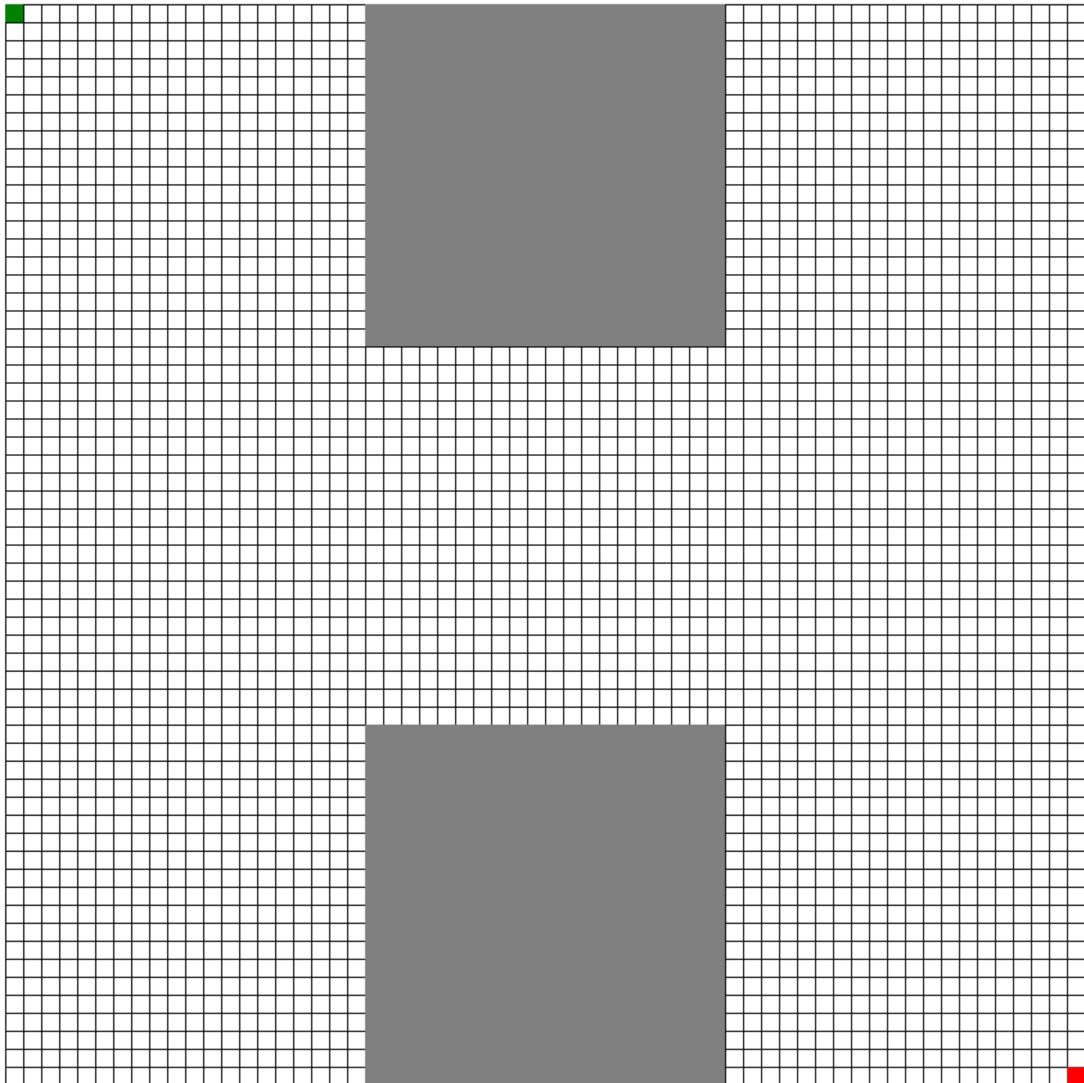

Abbildung 4.9: A simple maze (60x60)

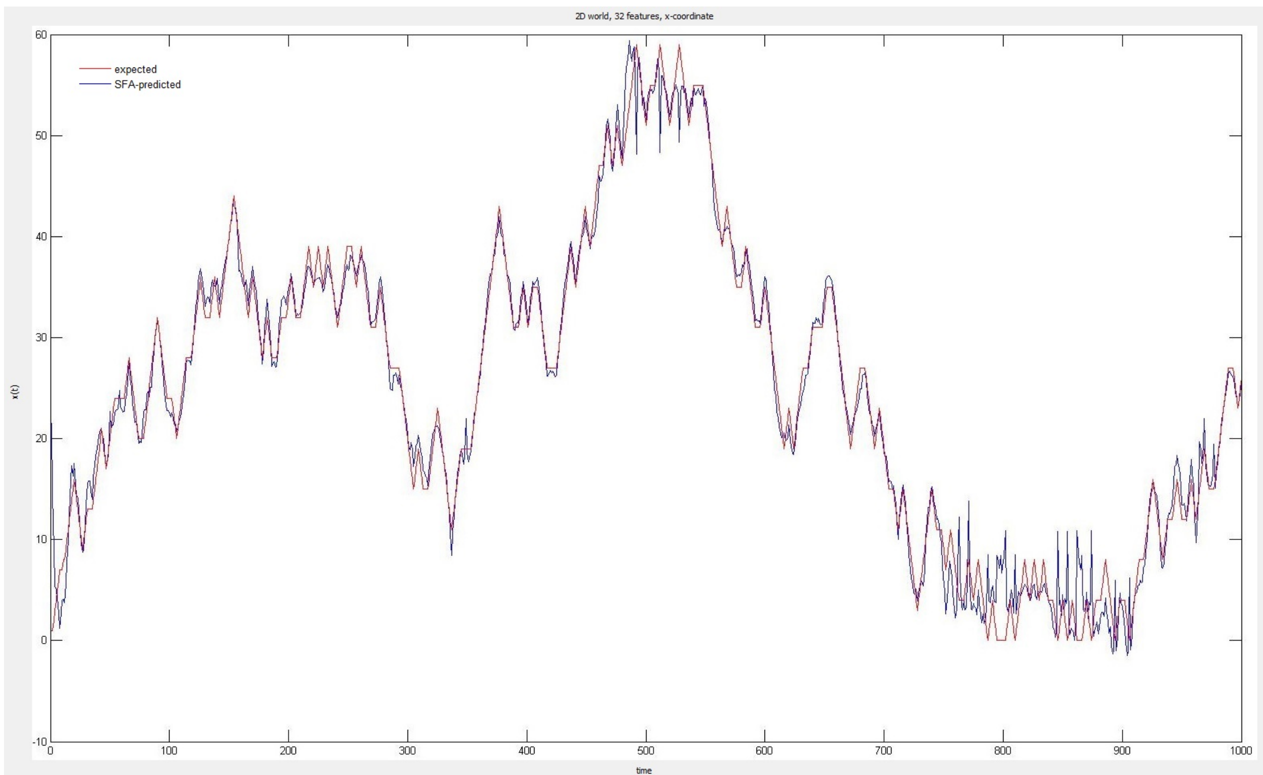

Abbildung 4.10: Comparison between real and SFA extracted path of the agent in x-direction on the trainings set.

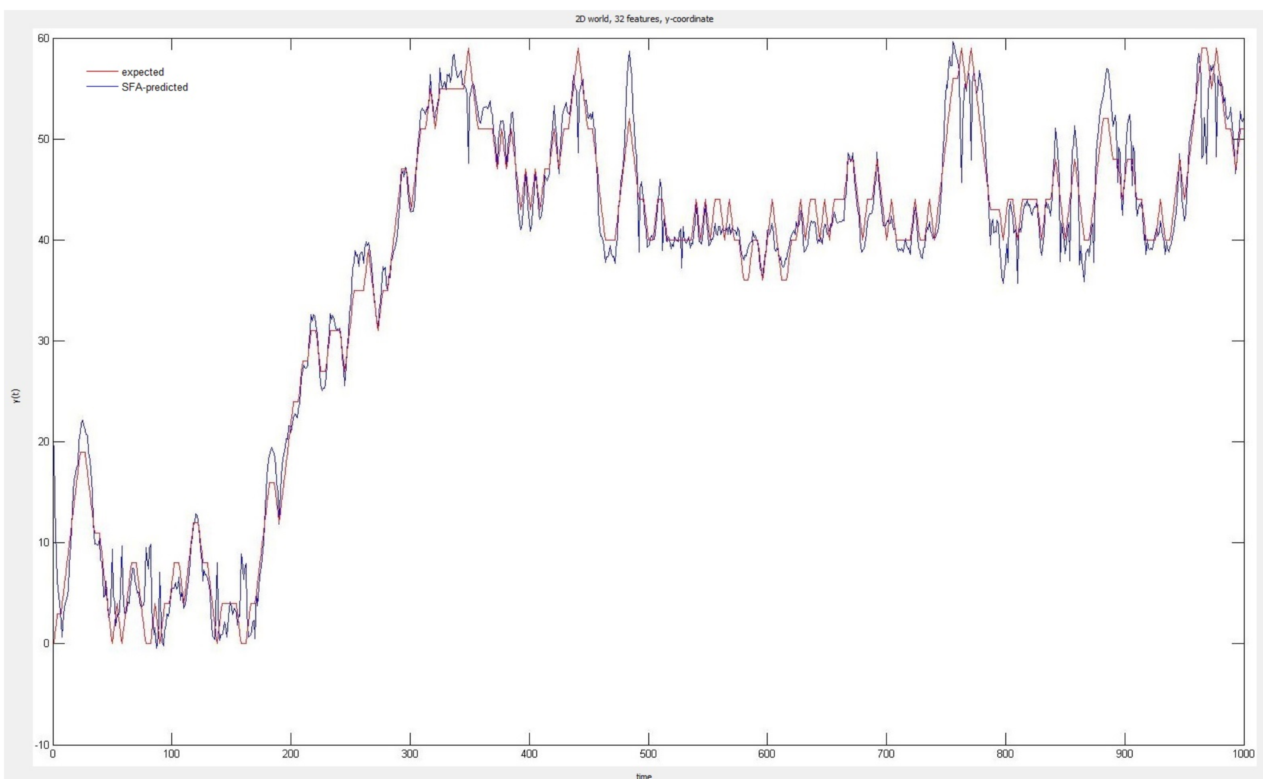

Abbildung 4.11: Comparison between real and SFA extracted path of the agent in y-direction on the trainings set.

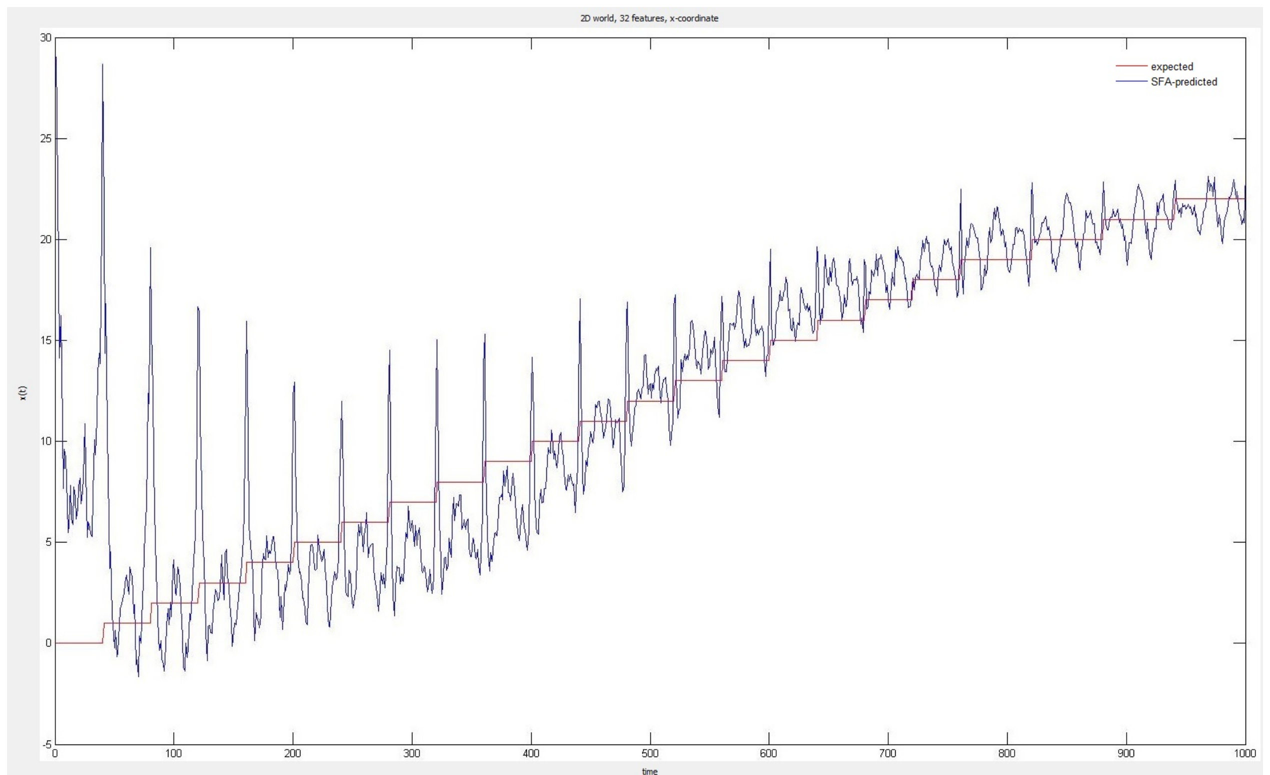

Abbildung 4.12: Comparison between real and SFA extracted path of the agent in x-direction on the test set.

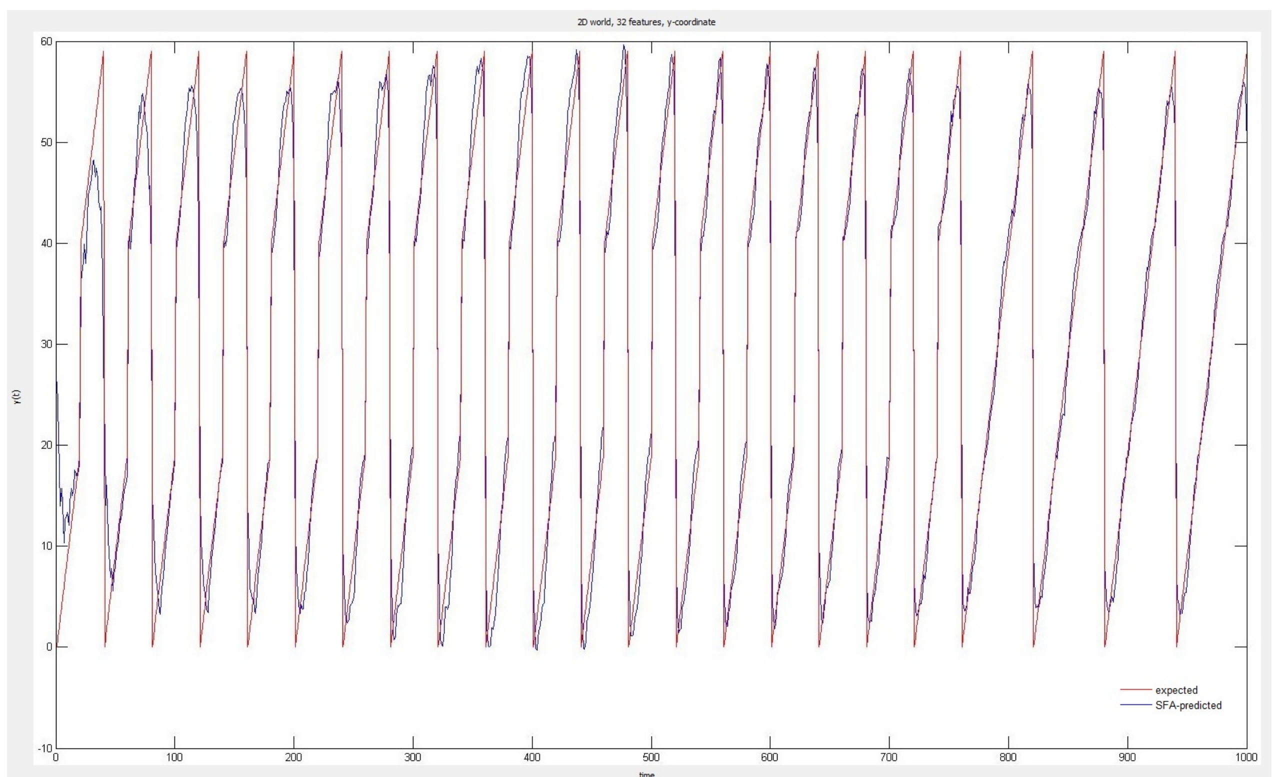

Abbildung 4.13: Comparison between real and SFA extracted path of the agent in y-direction on the test set.

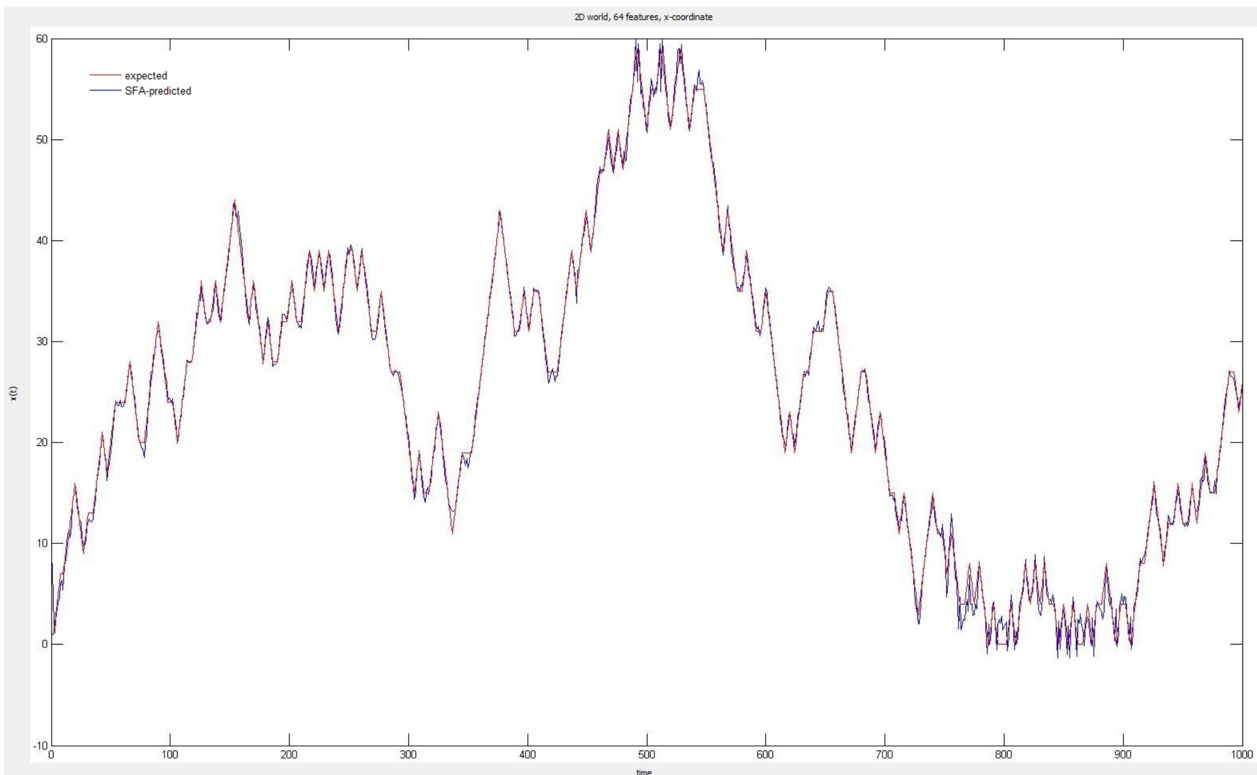

Abbildung 4.14: Comparison between real and SFA extracted path of the agent in x-direction on the trainings set.

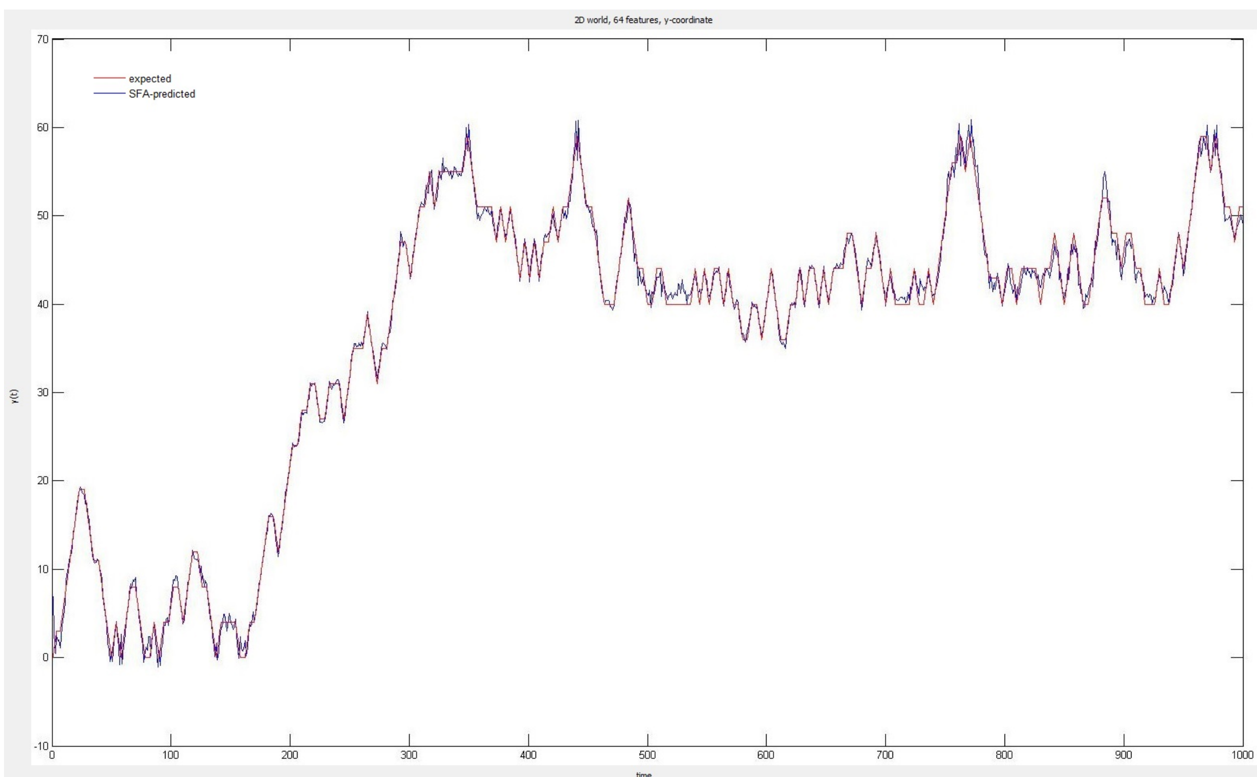

Abbildung 4.15: Comparison between real and SFA extracted path of the agent in y-direction on the trainings set.

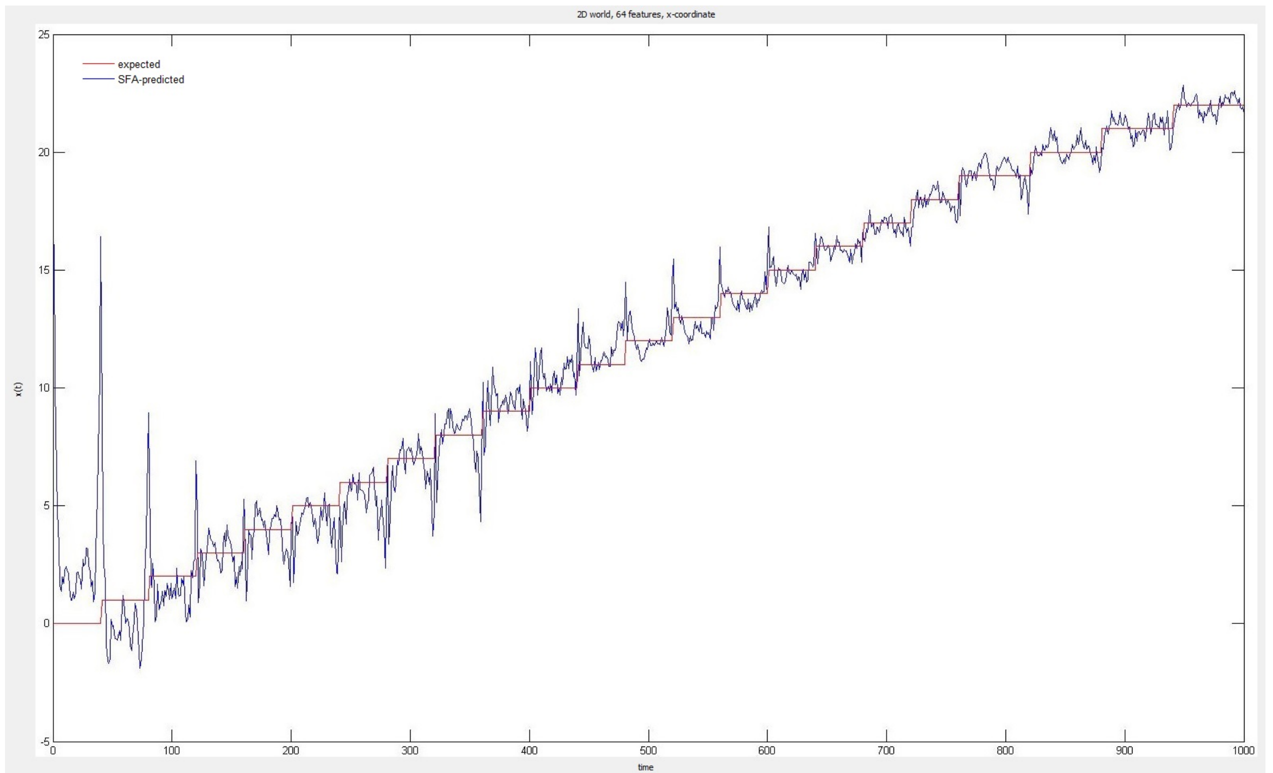

Abbildung 4.16: Comparison between real and SFA extracted path of the agent in x-direction on the test set.

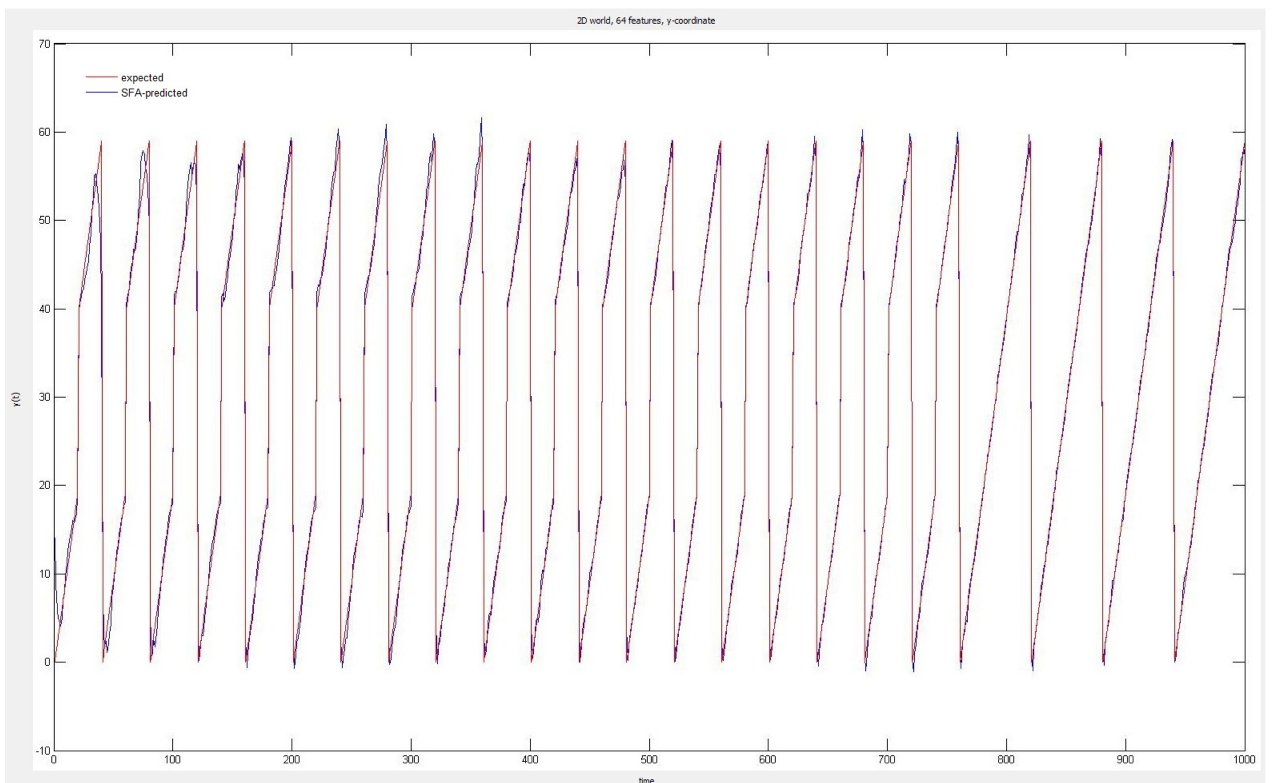

Abbildung 4.17: Ccomparison between real and SFA extracted path of the agent in y-direction on the test set.

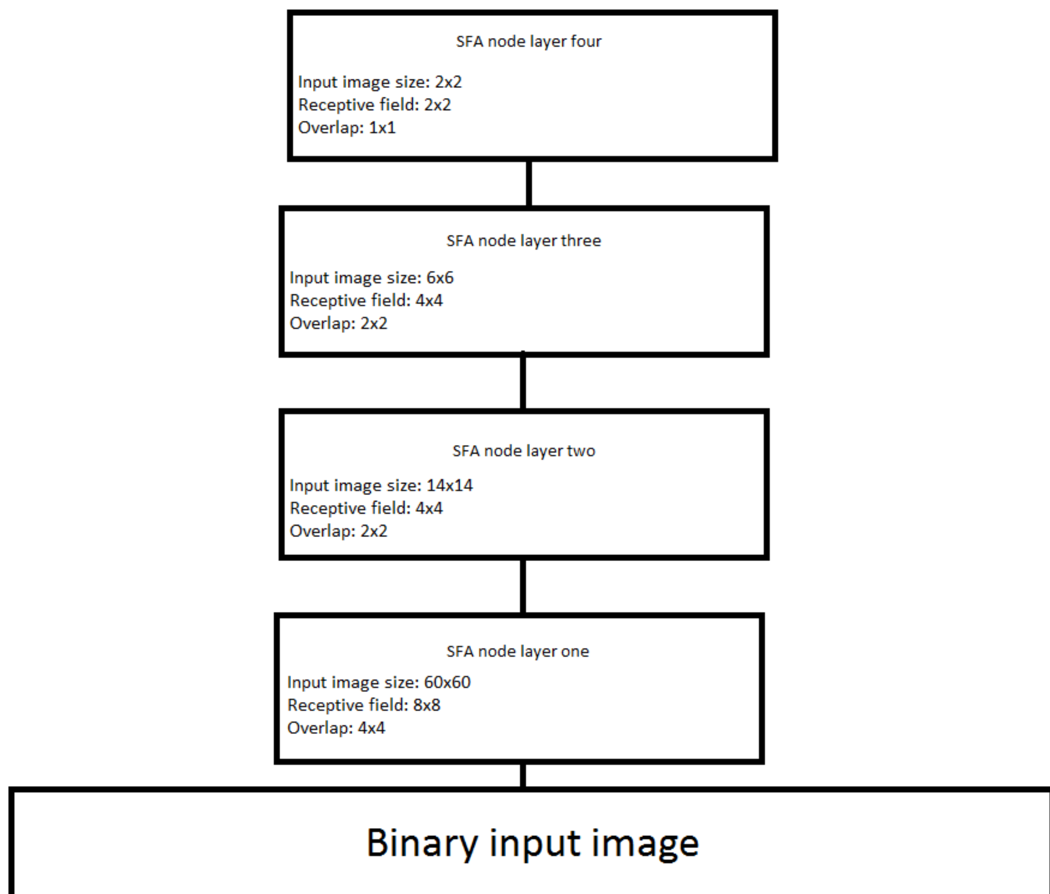

Abbildung 4.18: The SFA network used for the 2D environment

4.4 3D hierarchical SFA network

In this section the real use case of our feature extraction system is described, in the previous section simple binary images were used as input for the feature extraction system. In real life situations binary images are pretty rare. To evaluate if the system can handle more complex image structures a 3D OpenGL environment was implemented. This 3D system simulates a first person view (see figure 4.19) like it is known from popular ego shooter games (Doom, Quake, ...). To explain how this was exactly done is not part of this thesis, but we invite the interested reader to have a look at the attached source code. It should be mentioned here that a good knowledge of C++11/14 and Qt-5 is expected to understand what the code is doing. As in the simple 2D environment, the agent explores the 3D world randomly and for each step the agent performs an image is generated and saved for later processing. In the basic 2D example 100.000 images for the training phase were enough, in a 3D environment 200.000 combined images were necessary to get good results. A combined image (see figure 4.20) is an image which is assembled from four images, this means the agent performs a movement in one direction and rotates around the z-axis in 90°steps, so for a complete turn four images are produced. Next one combined image is generated out of the four images. This is a very important step, without doing this the feature extraction is not able to extract the precise location of the agent. This technique comes from image processing, for example in robotics this method is used to exactly determine the position of the robot in a 3D environment. In the paper [5] this method is described. The difference to this thesis is that the author of the paper suggested to produce images with a 320°view instead of 360°which is used here. The decision to use 360°views was made during experiments, since 360°views lead to better results than the suggestion made in the paper. This means for 200.000 combined trainings images 800.000 single images need to be rendered, which is quite an effort and requires several hours of computation. The same number of parameters need to be adapted as for the 2D version of the feature extraction system, also the node structure (see figure 4.31) stays the same. The only differences are the size of the receptive fields and the overlapping size of the first Linear-SFA nodes in the structure.

4.4.1 3D SFA nodes

Five different nodes are used to get the feature extractions system running. These nodes are the same as in the simple 2D experiment.

1. Additive-Noise: That is the first component in the node. To avoid problems like singular matrix errors, some noise is added to the input signal. The parameters of the algorithm are the 'mean' and the 'standard deviation', which were chosen to be the same as in the simple 2D version.

2. Linear-SFA: This algorithm is the first step to reduce the high dimensional input, to a output of 32 features.

3. Quadratic-SFA: In this step a quadratic expansion of the incoming data is mapped with a basis of the space of polynomials with degree up to two. In addition to the original data, all quadratic combinations are added to the data block.

4. Linear-SFA: A second Linear-SFA is applied to the expanded data. The output of this step is equivalent to the output of a SFA in the space of polynomials up to degree two.

5. Clipping: In this step the clipping algorithm is performed, the min parameter is set to -4 and the max parameter is set to +4. This procedure removes the extreme values that can occur on data due to the high influence the quadratic functions for large values.

4.4.2 Network structure

The network contains four layers see figure 4.31. The first layer directly maps to the input image, the final layer (4) provides the extracted features which are then used for further processing.

1. Layer 1: As already shown (see figure 4.20), the input images have 360×40 pixels, the receptive field has a size of 15×15 pixels with an overlap of 5×5 pixels and a channel dimension of 3. The channel dimension is used to distinguish between binary images which have only one color information (0 or 1) 'channel dimension = 1' or RGB images which have three color information 'channel dimension = 3'. The output of the first layer contains 32 features, this output serves as input to the next layer.

2. Layer 2: Due to the dimensionality reduction of layer 1 the input images have the size of 70×6 pixels. The receptive field of this layer has as size of 4×4 with an overlap of 2 pixels, the channel dimension is set to 32 since the output of layer 1 provides 32 features. The output of this layer again generates 32 features.

3. Layer 3: The images size now is reduced to 32×2 pixels. The receptive field size of this layer has now 2×2 pixels with an overlap of 2 pixels. The channel dimension is again set to 32 since the output of the previous layer contains 32 features. As the previous layer the output provides 32 features.

4. Layer 4: The final layer of the network, gets an input of size 17×1 pixels which results in a receptive field size of 17×1 and no overlap. The output of the final layer contains 64 features. This should be sufficient for further processing like ICA, to provide good input data for model based algorithms.

4.4.3 Results

In this section some training and test results are presented and explained. Methods like linear regression were used to generate these results. For this purpose Matlab was used since all the relevant algorithms and graphical tools are directly integrated and available. In the following we distinguish between 32 features and 64 features generated in the final layer of the network. For evaluating the training data the agent performs random exploration of the 3D world (see figure 4.20). For producing the test data, the agent performs a systematic walk through the 3D environment (see figure 4.20). This means the agent moves from left to right and from top to bottom. To perform the tests the linear regression algorithm was used (see figure 4.6). As input for the algorithm the path of the agent through the world and the extracted features calculated by the SFA network are necessary. Then the linear regression is calculated and the output is compared to the real path of the agent. The results are plotted separately for the x and y coordinates. The closer the two lines (real path and path calculated by the SFA network) are, the better the result. This can be quantified by using the mean square error (MSE).

4.4.4 Test on the Small Feature Set

In this section the final layer (4) of the SFA network produces 32 features, and all the predecessor layers also generate 32 features. The first plot (see figure 4.21), shows the evaluation of the training data in x-direction. This means the training path in x-direction of the agent is compared to the path calculated by the SFA network also in x-direction. To keep the plot simple and clear only the first 1000 time frames are shown. The second plot (see figure 4.22), shows the evaluation of the training data in y-direction. In this plot we also only show the first 1000 time frames. As it can be seen, the curves align nicely which means the parameters of the Additive-Noise algorithm and the number of features used in all four layers are well chosen.

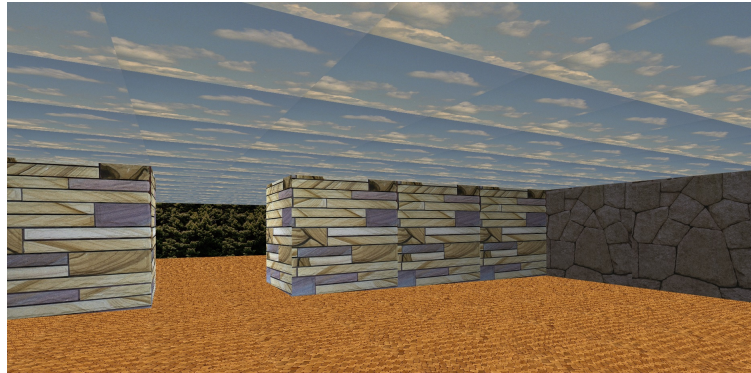

Abbildung 4.19: A first person view of the simulated 3D environment.

Abbildung 4.20: A combined image of the 3D environment.

The node and network structure stays untouched, so it's the same as for the training section. The difference is that the input data are smaller than for the training phase. During the test phase only 4248 time frames are used compared to the 200.000 during the training phase. The result for the x-coordinate can be seen in figure 4.23. The outcome is not bad, only the spike starting at time frame 0 are pretty high. For the y-coordinates see 4.24, the situation is better, considering only the y results, it would be fine to stick to 32 feature for this experiment.

4.4.5 Test on the Large Feature Set

Because of the test results in the previous section, especially the x coordinate plot, the number of features in the final layer (4) has been increased to 64 for this experiment. All the predecessor layers stay the same and generate 32 features. The input data also stay the same. In figure 4.25, we can observe improvements compared to the 32 features version. The calculated path of the SFA network follows the real path of the agent much more closely than in the 32 feature version. The same is true for y-coordinate (see figure 4.26).

As before the node and network structure stays untouched, so it's the same as for the training section. The test set contains 4248 images or time frames. The result for the x-coordinate can be seen in figure 4.27. Compared to the 32 feature version, the SFA system has improved in following the real path of the agent. The same observation is true for the y-coordinate which can be seen in figure 4.28.

4.4.6 Additive Noise

Same conditions for selecting this value hold as for the 2D version. An example of how an incorrectly selected noise value affects the evaluation can be seen in figure 4.29 and in figure 4.30. Especially the plot for the x-coordinate shows large spikes. Compared to the version with correctly selected noise parameters this plot is completely wrong, and does not follow the path of the agent at all. The y-coordinate plot shows a very noisy behaviour especially at the top and bottom edges of the graph but spikes can also be seen at the bottom at time frame 700 and 950. This example shows how only small changes to one single parameter can destroy the complete feature extraction system. It also shows that building and adapting a feature extraction system based on SFA, for real life situations is a very difficult and impractical approach, luckily there are other much more powerful feature extraction algorithms, like deep learning networks [24] that can be used in such real world scenarios.

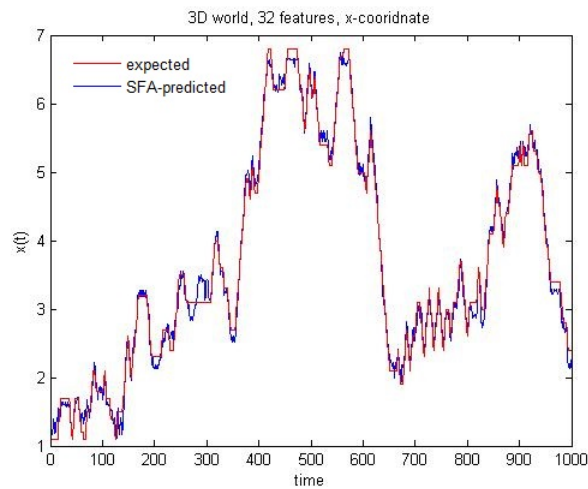

Abbildung 4.21: Comparison between real (red line) and SFA extracted path (blue line) of the agent in x-direction on the trainings set in a 3D environment.

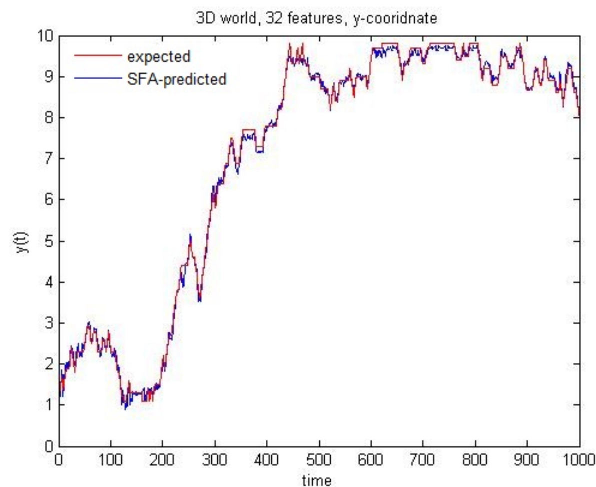

Abbildung 4.22: Comparison between real (red line) and SFA extracted path (blue line) of the agent in y-direction on the trainings set in a 3D environment.

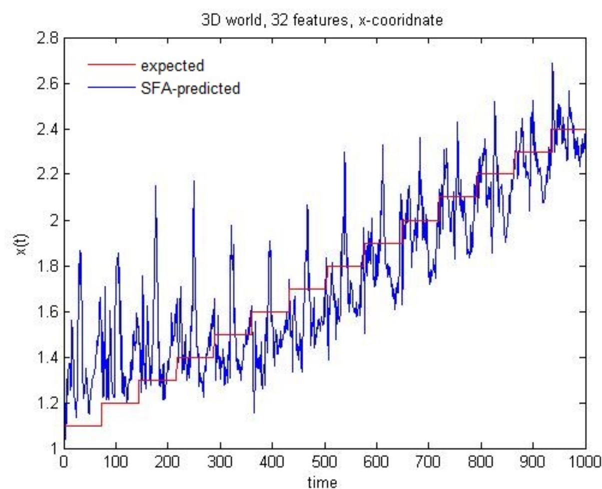

Abbildung 4.23: Comparison between real (red line) and SFA extracted path (blue line) of the agent in x-direction on the test set in a 3D environment.

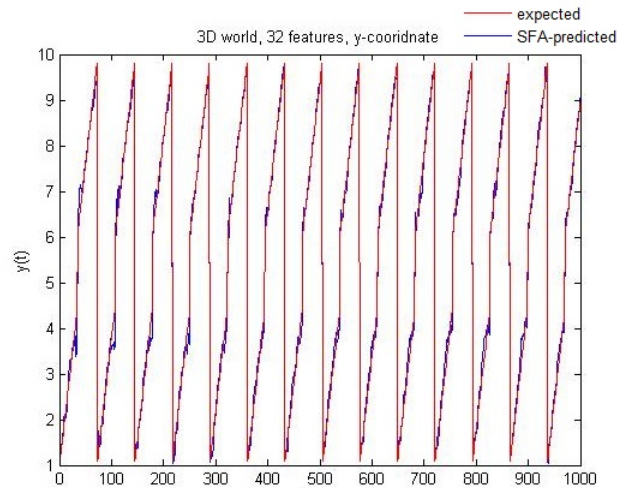

Abbildung 4.24: Comparison between real (red line) and SFA extracted path (blue line) of the agent in y-direction on the test set in a 3D environment.

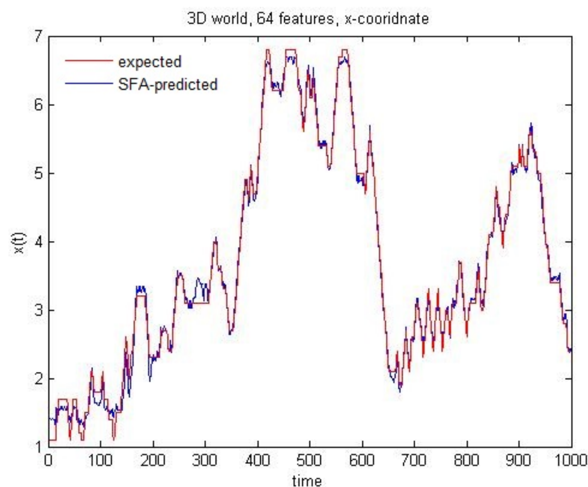

Abbildung 4.25: Comparison between real (red line) and SFA extracted path (blue line) of the agent in x-direction on the trainings set in a 3D environment.

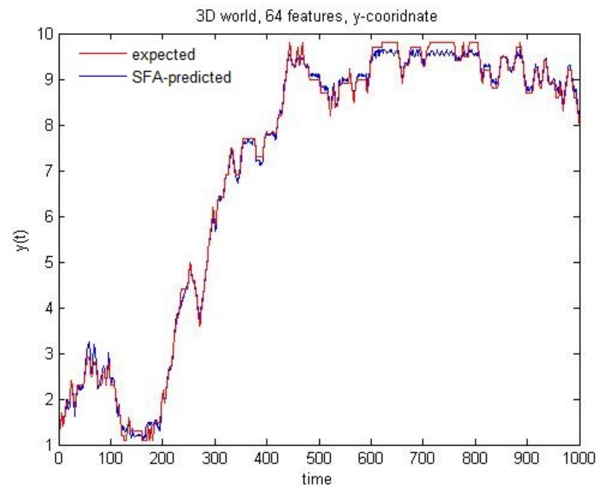

Abbildung 4.26: Comparison between real (red line) and SFA extracted path (blue line) of the agent in y-direction on the trainings set in a 3D environment.

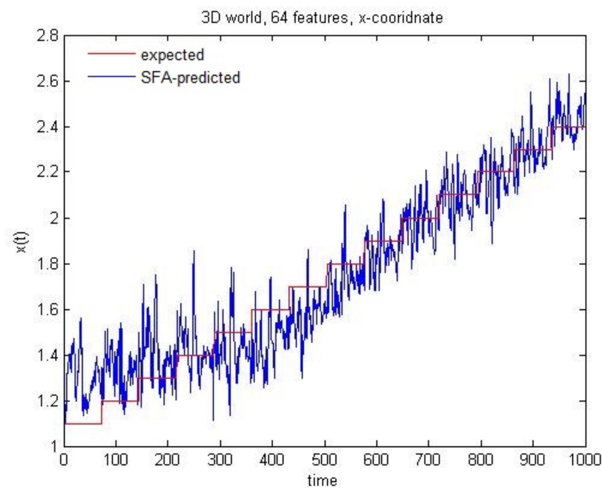

Abbildung 4.27: Comparison between real (red line) and SFA extracted path (blue line) of the agent in x-direction on the test set in a 3D environment.

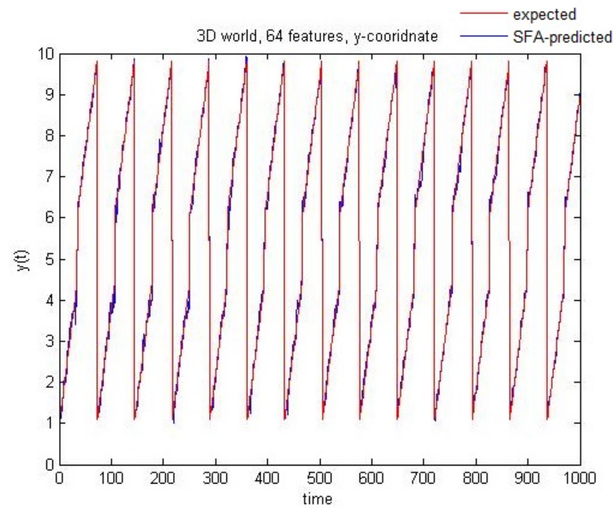

Abbildung 4.28: Comparison between real (red line) and SFA extracted path (blue line) of the agent in y-direction on the test set in a 3D environment.

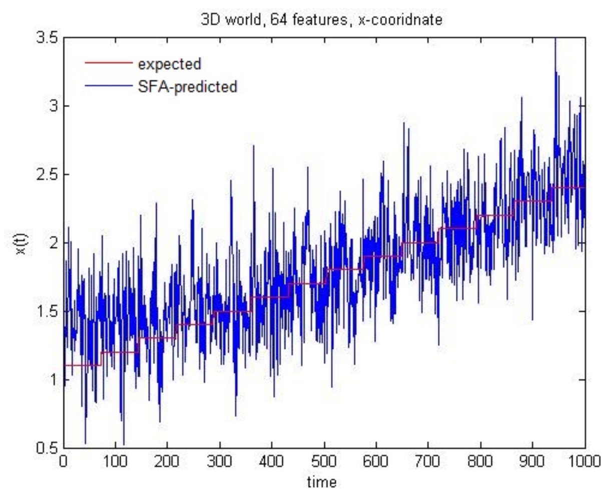

Abbildung 4.29: Comparison between real (red line) and SFA extracted path (blue line) of the agent in x-direction on the test set in a 3D environment, wrong selected additive noise.

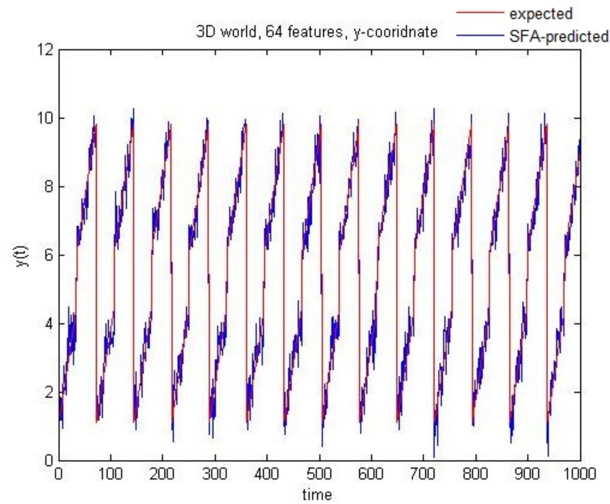

Abbildung 4.30: Comparison between real (red line) and SFA extracted path (blue line) of the agent in y-direction on the test set in a 3D environment, wrong selected additive noise.

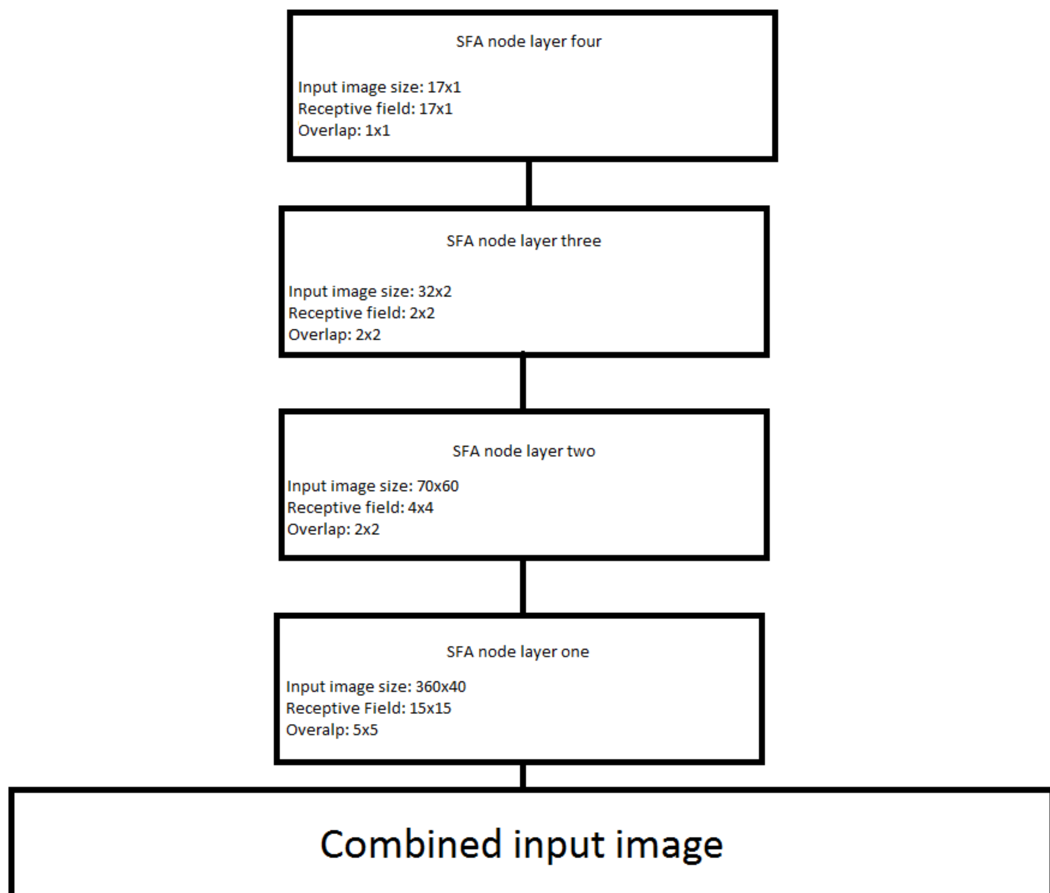

Abbildung 4.31: The SFA network used for the 3D environment

Kapitel 5

Model Based Learning Implementation

This chapter provides implementation details for the different methods used to generate a model out of an environment. Moreover, we outline the exploration of the environment using model learning algorithms. Furthermore all the relevant data structures are presented, along with an explanation how all information is kept up-to-date. The final part of this chapter describes how all the different components are put together to finally generate a complete model based learning system. The programming language used to implement the above explained structures is C++11/14. This advanced version of C++ was a good fit for the complex algorithms. Additionally this programming language provides some kind of platform independence and very good performance compared to other high level languages. All the used algorithms and their pseudo code can be found in the theoretical section of this thesis. It should be mentioned that no source code is added or explained in this thesis, a high level view is presented instead. Class and flow diagrams are used to show how the different components work together. For a deeper understanding on how something specific was implemented one can take a look to the attached source code. The model based learning structure is build out of three main modules: the environment, the model and the planning module. All these components communicate with each other during the learning phase. The communication itself and the three modules are explained in the following. After the implementation section a data analysis section follows, some experiments with different environments are shown as well as the differences in performance between Value-Iteration and UCT algorithms. Then we present an experiment where the reward state (terminal state) is dynamically changed during execution. We then compare the results between Value-Iteration and UCT. In the final section some possible future work is discussed which can improve the performance of the overall system.

5.1 From Features to Environment

In this section it is shown how to transform features generated by the SFA algorithm to discrete states to represent the environment. The first step after the SFA is to perform an ICA (as already explained in chapter 4). The algorithm calculates a set of independent or unique features (states) which are used by the environment class to create a discrete map of states see figure 5.1. The algorithm to discretize a feature space is pretty simple, the following input data are required:

- The ICA output, which contains in the columns the features and in the rows the different time frames.
- The positions of the agent during the SFA test phase, this means the position of the agent at a specific point in time.
- The start state coordinates.
- The terminal state coordinates.

With this information in place the algorithm iterates through all positions the agent visited. The method first obtains the feature output at the specific point in time (for example 64 features at position 10) the so called feature vector. Then the index of the feature with largest value in the feature vector is determined. This is a state or a part of a state in the discretized matrix. Then the current position of the agent is compared to the start respectively terminal state, and if one of those matches the state is marked as start or terminal state. The pseudo code can be seen below.

```
1  timeIdx = 0;
2  discretizedMatrix = createMatrix(rows, columns);
3
4  foreach (position in agentPositions)
5  {
6          featureVector = icaInput(timeIdx++);
7          maxIdx = max_element_idx(featureVector);
8
9          discretizedMatrix(position.x, position.y) = maxIdx;
10
11         if (position == startStateCoordinate)
12                 startState = position;
13
14         if (position == terminalStateCoordinate)
15                 terminalState = position;
16 }
```

As it can be seen in the pseudo code the algorithm is simple and can be used for any number of features and image sizes.
This is not the only reason why the so called 'Environment' class exists it also helps the model based agent or planner to explore its environment in a way that the agent at any point in time exactly knows where it is and what happens when it performs a certain action in the environment. To provide the agent with this information it is necessary to implement some methods that allow the agent to access the information it needs. The following methods are used to establish a communication between the agent an the environment:

- CurrentState: The state in which the agent is located at the moment.
- Apply: When an action was selected by the agent this function applies this action to the environment and returns the reward.
- IsTerminal: Returns true when the agent reached the terminal state otherwise false.

- Reset: Sets the start position of the agent on some random state, this method is used for random exploration of the environment.

The above mentioned methods are the most important ones. There are a few more but these are mainly used for the graphical representation of the environment.

5.2 The R-Max Model

The R-Max algorithm generates a model of the real world. The 'Environment' class represents the 'real' world. The R-Max model has no direct connection to the environment, it gets the required information from the agent. The agent is the only instance that directly communicates with the environment. How exactly this works is explained in the agent section of this chapter. As already mentioned in the introduction of this chapter pseudo code does not always fit the real world needs, exactly this is the case for the R-Max algorithm. The real implementation splits the algorithm in an update phase, where the agent provides its experience form the real world to the model and the 'get action' phase where the model provides the agent with information on how to plan the next action. Internally the algorithm uses some data structures to keep information about the model, for example an associative container (map) is used to store information of each and every state. This is done by using a struct named 'StateInfo' which stores information like number of visits for each action, an outgoing transitions count to each state reached with a specific action, and so on. The following methods are used to establish a communication between the agent an the model:

- UpdateWithExperience: This method is called when the agent performed an action on the environment. The method updates all the different counts, checks if the model has changed and determines if the state is 'known'. The return value indicates whether the model was changed by the update or not. The parameters of the method are:
  - CurrentState: The state where the agent is located now.
  - Action: The action the agent performed from the last state.
  - NextState: The state where the agent goes next.
  - Reward: The reward the agent received when performing the action.
  - IsTerminal: Determines whether the current state is a terminal state or not.
- GetStateActionInfo: This method is called by the agent to get information about the current state. The method calculates transition probabilities, the reward predictions and the terminal probabilities. The parameters of the method are:
  - State: The current state (position) of the agent.
  - Action: The action that was performed by the agent to reach the current state.
  - StateActionInfo: The parameter is a pointer to a struct which contains all relevant information about the current state (is the state already know, the reward, the terminal probability and the transition probabilities for each action).

The above methods are the most important ones, they are only used by the agent (planner) to update the model and get information of the current state of the model. Other methods exists but they are not important to operate the algorithm, they are only used to get information out of the model for graphical representations.
A description of the above mentioned methods in pseudo code now follows, to give a idea what exactly is done by these methods:

```
bool UpdateWithExperience(
        int currentState,
        Action action,
        int nextState,
        double reward,
        bool isTerminal)
{
        currentStateInfo = stateData[lastState];

        % If the state is already know -> model has not changed
        if (currentStateInfo->known[action])
                return false;

         % Update visit count for action just executed
        currentStateInfo->visits[action]++;

        % Update termination count
        if (isTerminal)
                currentStateInfo->terminations[action]++;

        % Update reward sum for this action
        currentStateInfo->Rsum[action] += reward;

        % Update transition count for outcome that occurred
        % First get transitions for all actions
        transitionCounts = currentStateInfo->outCounts[currentState];

        % Only update state transition counts for non-terminal transitions
        if (!isTerminal)
                transitionCounts[action]++;

        % Check if current state becomes known
        if (!currentStateInfo->known[action] &&
            currentStateInfo->visits[action] >= numVisits)
            {
                currentStateInfo->known[action] = true;
            }

        % Model changed
        return true;
}
```

```
void getStateActionInfo(
        int state,
        Action action,
        StateActionInfo *stateActionInfo)
{
        info->transitionProbabilities.clear();
        currentStateInfo = getInfo(state);

        % Unknown state
        if (currentStateInfo->visits[action] == 0)
        {
                % Set StateActionInfo to defaul values
                return;
        }

        % Calculate transition probabilities
        foreach (outCount in outCounts)
        {
                T(s, a, s') = C(s, a, s') / C(s, a);
        }

```

```
22          R(s, a) = Rsum(s, a) / C(s, a);
23          TermProb = Term(s, a) / C(s, a);
24  }
```

$C(s, a)$... number of times an action a was taken from state s.
$C(s, a, s')$... number of times that each next state s' was reached form (s, a).
$T(s, a, s')$... probability of outcome s'.

5.3 The Value Iteration Algorithm

Value-Iteration is a so called planning algorithm. It plans the optimal policy or action from one state to another. All the details on how to find the optimal policy and how the algorithm works can be found in the theoretical section of this thesis. In grid world based problems normally one cell represents exactly one state, which is not true in the real world problem. In the real world problem one state can be spread out across multiple cells therefore it is important to define a so called step-width for the agent. When the agent performs an action it moves, for example, three cells to the left direction instead of a single cell. The Value-Iteration algorithm communicates with the model (R-Max) as well as with the environment (the real world). The model provides the agent with the necessary information on how to decide which action to take from the current state, the environment provides the reward for the chosen action. After each action the model gets updated by the agent with the real world experience. In this section the communication between the agent and the model is described. The communication between agent and environment is described in the section 5.5. Internally the Value-Iteration algorithm uses some structures to keep the information about each and every state. It also keeps a pointer to the model class to access and update the model. The agent (ValueIteration class) provides the following methods to perform its calculations:

- PlanOnNewModel: This method is called whenever the model (R-Max) is updated with new experiences, so for example a new state was added during the last exploration step. The method then calls the following two private methods:
  - UpdateStateActionFromModel: This method accesses the underlying model (in this case R-Max) to get all necessary information of the current state and the current action.
  - CreatePolicy: This method performs the value-iteration algorithm. The method itself calls some helper functions to perform all calculations, which will be described by the pseudo code.
- UpdateModelWithExperience: This method is called after the agent performed an action on the real environment (the Apply() method of the Environment class was called). All information collected by the agent in the previous step is stored in the model. The parameters of the method are:
  - LastState: The state of the agent before applying an action.
  - Action: The action from the previous state to the current state.
  - CurrentState: The state in which the agent is now.
  - Reward: The reward the agent received for moving from the previous state to current state.
  - IsTerminal: Determines if the current state is a terminal state or not.
- GetBestAction: This method is called by the model based agent class see section 5.5, whenever the agent enters a new state this method is called to determine which action to perform to get the maximum reward. The parameter of the method is:
  - State: The current state of the agent.

The listed methods are the most important ones, there exist other public methods like load() and save(), which allow the user of the class to load respectively save the learned parameters from or to a file. These additional methods are not explained in this thesis, because they are not relevant for the functionality of the algorithm. The interested reader is invited to take a look at the attached source code. In the following a description of the above mentioned methods in pseudo code follows. The intention is to provide the reader with an understanding of what the methods exactly do and how they do it.
The first method to be described is 'PlanOnNewModel':

```
void PlanOnNewModel()
{
        % Update model info
        UpdateStateActionFromModel();

        % Run value iteration algorithm
        CreatePolicy();
}
```

As already mentioned above this method simply calls two private methods to get the job done.
The private method 'UpdateStateActionFromModel' will be explained next:

```
void UpdateStateActionFromModel()
{
        % Get information's of the state
        stateInfo = stateData[previousState];

        % Update stateInfo - get state action info from the model
        model->GetStateActionInfo(
                previousState,
                previousAction,
                stateInfo->modelInfo[previousAction]);
}
```

The above method is used to update the internal data structure of the Value-Iteration algorithm. Information like visit counts and Q-Values are stored in the model, these information are used by the Value-Iteration algorithm to plan the next action.
Now the method 'CreatePolicy', which performs the value iteration algorithm is explained:

```
void CreatePolicy()
{
        while (maxError > MinError)
        {
                % Loop over all states
                foreach(state in states)
                {
                        % Execute the Value-Iteration algorithm
                        % The return value contain the error
                        errorValue = PerformValueIterationAlgorithm(state);

                        if (errorValue > maxError)
                                maxError = errorValue;
                }
        }
}
```

The 'CreatePolicy()' method maintains an error value which is returned by the Value-Iteration algorithm to decide when to leave the update loop. Beside that it's a very simple method which iterates over all available states in the state-space and performs calculations on each and every state. The calculations performed by the method 'PerformValueIterationAlgorithm' are explained next:

```
double PerformValueIterationAlgorithm(int state)
{
```

```
3         % Get information of the state
4         stateInfo = stateData[state];
5
6         % Loop over all available actions
7         for (action in actions)
8         {
9                 % Get the model data for the current state and action
10                modelInfo = stateInfo->modelInfo[action];
11
12                % Calculate the Q-Value for the current state
13                qValue = CalculateTransitionProbabilities(
14                        currentState,
15                        stateInfo,
16                        modelInfo);
17
18                % Calculate the difference between new and old Q-Value
19                error = abs(stateInfo->qValues[action] - qValue);
20
21                % Update the state information with the new Q-Value
22                stateInfo->qValues[action] = qValue;
23        }
24
25        % Return the error
26        return error;
27 }
```

The 'PerformValueIterationAlgorithm' method simply iterates over all actions for the current state and calculates a new Q-Value according to the updated information in the model. Then the absolute difference between the old Q-Value and the new Q-Value is calculated (error) and returned by the method. The helper function 'CalculateTransitionProbabilities' determines the new Q-Value and is explained next:

```
1  double CalculateTransitionProbabilities(
2          int currentState,
3          StateInfo *stateInfo,
4          StateActionInfo *modelInfo)
5  {
6          % Q = R + discounted value of the next state
7          % First part of the Q-Value calculation -> the R part
8          newQ = modelInfo->reward;
9          probabilitySum = modelInfo->termProbability;
10
11         % For all next states, add discounted value appropriately.
12         % Loop through next state's that are in this state-actions list.
13         for (transition in modelInfo->transitions)
14         {
15                 % The probability must be between ->
16                 % 0 < transitionProbability < 1
17                 transitionProbability = (
18                         1.0 - modelInfo->termProbability) *
19                         modelInfo->transitionProbabilities[nextState];
20
21                 % Used for plausibility check
22                 probabilitySum += transitionProbability;
23
24                 % Update q values for any next states
25                 nextStateInfo = stateData[next];
26
27                 % Get maximum Q-Value
28                 maxQValue = max_element(nextStateInfo->qValues);
29
30                 % Calculate the new Q-Value
31                 newQ += (gamma * transitionProbability * maxQValue);
```

```
32        }
33
34        % probabilitySum must be 1
35        if (probabilitySum < 0.9999 || probabilitySum > 1.0001)
36                return some error condition
37
38        return newQ;
39 }
```

The above method is the last helper method in the process of calculating all relevant data for the Value-Iteration algorithm. It should be mentioned that not all parameters of the method are used in the pseudo code description, since the intention of pseudo code is to keep things as simple as possible.

```
1  bool UpdateModelWithExperience(
2          int lastState,
3          Action action,
4          int currentState,
5          double reward,
6          bool isTerminal)
7  {
8          % Get information of the state
9          stateInfo = stateData[lastState];
10
11         % Update the state visit count
12         stateInfo->visits[action]++;
13
14         % Create the experience class and fill it with all necessary information
15         Experience experience(lastState, action, reward, currentState, isTerminal
               );
16
17         % Update the model
18         return model->updateWithExperience(experience);
19 }
```

The method 'UpdateModelWithExperience', updates the model with the information from the previous action the agent performed. As it can be seen in the above pseudo code the 'Experience' struct is filled with the information from the agent's previous step, and then the model is updated with the new experience. The model returns a simple boolean to inform the caller whether the model has changed. If the model has changed the method 'PlanOnNewModel' gets called which triggers a recalculation of the QValues and state transitions.

```
1  Action GetBestAction(int state)
2  {
3          % Get information of the state
4          stateInfo = stateData[state];
5
6          % Get Q values
7          qValues = currentStateInfo->qValues;
8
9          % Choose an action
10         auto maxQValueIdx = max_element_idx(qValues.begin(), qValues.end());
11
12         return (Action)(maxQValueIdx)
13 }
```

The above method determines the best action from the current state. To achieve that, the maximum Q-Value for the current state is chosen, which is then mapped to an action. Each index of the Q-Value array represents an action. In our implementation, for example, the index two means 'Up'. The caller receives the best action which is applied to the environment, which is done by the class 'ModelBaseAgent' and is explained later in this section.

5.4 The Upper-Confidence-Bound applied to trees Algorithm

Upper-Confidence-Bound applied to trees (UCT) is quite similar to the Value-Iteration algorithm, but instead of an array, which is used by the Value-Iteration algorithm to keep track of all the calculated value functions for each state, the UCT algorithm uses a search tree to calculate the best possible action based on the states that the agent most likely visits soon. An explanation on how this works can be found in the theoretical part of this thesis. In this section we focus on how the algorithm was implemented and how it differs from the pseudo code version presented in the theory section. The communication between model, planner and environment stays the same as for the Value-Iteration algorithm, but to make this section self-contained the communication between the different modules is explained again. The UCT algorithm communicates with the model (R-Max) as well as with the environment (the real world). The model provides the agent with the necessary information on how to decide which action to perform from the current state. The environment then provides the reward for the chosen action and after this the model gets updated with the real world experience. In this section the communication between the agent and the model is explained, the communication between agent and environment is described in the section 5.5. Internally the UCT algorithm uses a tree data structure (std::map) to keep the information about each state and it also keeps a pointer to the model class to access and update the model. Additionally a pointer to the previous state is maintained, which keeps all the information of the state where the agent came form. The agent (UCT class) provides the following methods to perform its calculations:

- PlanOnNewModel: This method is called whenever the model (R-Max) is updated with new experiences, for example when a new state was added during the last exploration step. The method then calls the following two private methods:
  - ResetUCTCounts: This method accesses the internal search tree. In this tree each and every state is stored. The method iterates over all these states and resets their statistic back to default values.
  - UpdateStateActionFromModel: This method updates the previous state data structure. The parameters of the method are:
    - PreviousState: The state before the current state.
    - PreviousAction: The action performed to enter the current state.
    - PreviousStateInfo: A pointer to data structure which keeps all the information of the previous state.
- UpdateModelWithExperience: This method is called after the agent performed an action on the real environment (the Apply() method of the Environment class was called). All information collected by the agent in the previous step is stored in the model. The parameters of the method are:
  - LastState: The state of the agent before applying an action.
  - Action: The action from the previous state to the current state.
  - CurrentState: The state in which the agent is now.
  - Reward: The reward the agent received for moving from last state to current state.
  - IsTerminal: Determines if the current state is a terminal state or not.
- GetBestAction: This method is called by the model based agent class see section 5.5. Whenever the agent enters a new state this method is called to determine which action to perform to get the maximum reward. The parameter of the method is:
  - State: The current state of the agent.

  The method then calls the following private method:

- UctSearch: This method is the UCT algorithm itself. It performs all the steps that are outlined in the pseudo code of the theoretical section. The parameters of the method are:
  - CurrentState: The current state.
  - Depth: The search depth of the algorithm, i.e., how deep the tree can get when performing the search.

  The method then calls the two private methods SelectUCTAction and SimulateNextState:
  - SelectUCTAction: This method selects the best action depending on the current state. The parameter of the method is:
    - StateInfo: Statistics of the current state.
  - SimulateNextState: This method calculates, depending on some statistics, the next state to be visited in the MC search tree. The parameters of this method are:
    - CurrentState: The current state in the search tree.
    - StateInfo: The statistics of the current state.
    - Action: The action to perform from the current state to the next state
    - Reward: A pointer to the reward, the method returns the reward that can be expected when performing the passed in action to the next state.
    - Terminal: A pointer that informs the caller if the next state is terminal or not.

The above mentioned methods are the most important ones, but there are other public methods like load() and save(), which allow the user of the class to load respectively save the learned parameters from or to a file. These additional methods are not explained in this thesis, because they are not relevant for the functionality of the algorithm. The interested reader is invited to take a look at the attached source code. Next, we explain the mentioned methods using pseudo code. The intention is to provide the reader with an understanding of what the methods exactly do and how they do it.
The first method to describe is 'PlanOnNewModel':

```
1 void PlanOnNewModel()
2 {
3         % Reset visit counts, qValues....
4         ResetUCTCounts();
5
6         % Update the previous state information
7         UpdateStateActionFromModel(previousState, previousAction,
              previousStateInfo);
8 }
```

As already mentioned above this method simply calls two private methods.
The next method to describe is 'ResetUCTCounts'

```
1  void ResetUCTCounts()
2  {
3          % Iterate over all states in the tree
4          foreach(state in states)
5          {
6                  % Get information about the current state
7                  stateInfo = getStateInfo(state);
8
9                  % If number of visits larger than some default values
10                 % Set the visits count back to default
11                 if (stateInfo->visits > (minVisits * numberOfActions))
12                         stateInfo->visits = minVisits * numberOfActions;
13
14                 % Iterate over all actions
15                 foreach(action in actions)
16                 {
```

```
17                          % If number of visits for the current
18                          % action are larger then some default value.
19                          % Set back to default value.
20                          if (stateInfo->uctActions[action] > minVisits)
21                                  stateInfo->uctActions[action] = minVisits;
22                  }
23          }
24  }
```

The above method resets all statistics maintained by the search tree. The 'minVisits' constant can be set by the user of the class.
Now the description of the method 'UpdateStateActionFromModel' follows

```
1   void UpdateStateActionFromModel(
2           int previousState,
3           Action previousAction,
4           StateInfo* previusStateInfo)
5   {
6           % Get model information from the previous action
7           *modelInfo = &prevStateInfo->historyModel[previousAction];
8
9           % Update the state information with data from the model
10          model->getStateActionInfo(previousState, previousAction, modelInfo);
11  }
```

The above method first gets all the information from the previous state, which depends on performed action. Next, this information gets updated with data from the model, since the variable 'modelInfo' is a pointer to the data which are stored in the 'historyModel' and can be used for the next tree search step.
The next method to explain is 'UpdateModelWithExperience'

```
1   bool UpdateModelWithExperience(
2           int lastState,
3           Action action,
4           int currentState,
5           double reward,
6           bool isTerminal)
7   {
8           % Keep track about the state and action history
9           previousState = lastState;
10          previousAction = lastAction;
11
12          % Get information of the state
13          stateInfo = stateData[lastState];
14
15          % Create the experience class and fill it with all necessary information
16          Experience experience(lastState, action, reward, currentState, isTerminal
                );
17
18          % Update the model
19          return model->updateWithExperience(experience);
20  }
```

The method 'UpdateModelWithExperience', updates the model with the information from the last action the agent performed. In the pseudo code above the global variables 'previousState' and 'previousAction' are set to valid values, which is necessary since the algorithm needs information about the past. After this the 'Experience' struct is filled with the information from the agent's previous step, and then the model is updated with the new experience. The method returns a boolean to inform the caller whether the model has changed. If the model has changed the method 'PlanOnNewModel' is called which triggers an update on the underlying model as well as a reset of the statistics used by the UCT algorithm.
As next method 'GetBestAction' is explained:

```
Action GetBestAction(int state)
{
        % Perform a specific number of MC rollouts
        for (rollout to maxRollouts)
                UctSearch(state, 0)

        % Get information about the current state
        currentStateInfo = stateData[state];

        % Get maximum QValue index
        maxQValueIdx = max_element_idx(currentStateInfo->qValues);

        % Return the action
        return (Action)(maxQValueIdx);
}
```

The above method first performs a user specified number of Monte Carlo (MC) roll-outs. This means the algorithm performs a roll-out, starting from the agent's current position or state to a user defined depth. At each state an action is selected which leads to the next state one level deeper in the search tree. After reaching the maximum number of roll-outs or after reaching a terminal state, the values of the taken actions are updated recursively towards the start state. Then the state information of the current state is used to determine the optimal action from the current state to the next one. Depending on the number of states and the selected number of roll-outs this process can take quite a long time. There are methods to traverse multiple search trees at the same time in parallel, but these methods are difficult to implement and that's why they where not used in this thesis.
As next method 'UctSearch' is explained:

```
double UctSearch(currentState, depth)
{
        % Maximum depth reached -> nothing do to here
        if (depth > maxDepth)
                return maxQValue of current state

        % Call helper function 'SelectUCTAction'
        action = SelectUCTAction(currentStateInfo);

        % Call helper function 'SimulateNextState'
        nextState = SimulateNextState(
                currentState,
                currentStateInfo,
                action,
                reward
                terminal);

        % Terminal state reached?
        if (terminal)
        {
                currentState->qValue[action] =
                        learnRate *
                        (reward - currentState->qValue[action]);

                currentState->visits++;
                currentState->uctActions[action]++;

                return reward;
        }

        % Terminal state not reached
        % Recursively calculate QValue
        newQ = reward + UctSearch(nextState, depth + 1);

```

```
35          % Update QValue and visits statistics
36          currentState->qValue[action] =
37                  learnRate * newQ +
38                  (1 - learnRate) * currentState->qValue[action];
39
40          currentState->visits++;
41          currentState->uctActions[action]++;
42
43          return newQ;
44  }
```

The above function performs the UCT roll-out. At first the method checks if the maximum search tree depth defined by the user was reached. If so the maximum QValue of the current state is returned. If the maximum depth has not been reached yet the helper function 'SelectUCTAction' is called. This function determines the best action from the current state to next one. Then the helper function 'SimulateNext-State' is called to calculate the transition probabilities from the current state to the next one using the action provided by the method 'SelectUCTAction'. If the function 'SimulateNextState' determines that a terminal state was reached, the Q-Value of the state, the statistics for the visit and the action counts are updated. If no terminal state was reached the method 'UctSearch' is called recursively with the 'next State' value and the incremented 'depth' as parameters. Finally, as already explained for the case a terminal state was reached, the Q-Value and the statistics for visit and action counts are updated.
Now the helper method 'SelectUCTAction' is explained:

```
1   Action SelecteUCTAction(stateInfo)
2   {
3           foreach(action in actions)
4           {
5                   qValues[action] = qValues[action] + rewardBound *
6                           2 * sqrt(log(visits) / uctActions[action];
7           }
8
9           return maxQValueIdx;
10  }
```

The method 'SelecteUCTAction' iterates through all available actions and calculates the Q-Value for each action using the formula:
$\mathrm{argmax}_{a'}(Q^{d}(s, a') + 2 * r_{range} * \sqrt{\log(c(s, d)/c(s, a', d))})$.
Then the action with the maximum Q-Value is returned.
Next we explain the helper method 'SimulateNextState'

```
1   int SimulateNextState(
2           currentState,
3           stateInfo,
4           action,
5           reward,
6           terminal)
7   {
8           % Create a random uniform distribution
9           uniform_real_distribution uniformDistribution
10
11          terminal = uniformDistribution() < terminalProbability;
12
13          if (terminal)
14                  return currentState;
15
16          randomProbability = uniformDistribution();
17
18          % Iterate over all transition probabilities
19          foreach (probability in probabilities)
20          {
```

```
21                  probabilitySum += probability;
22
23                  if (randomProbability < probabilitySum)
24                  {
25                          nextState = probability->state:
26                          break;
27                  }
28          }
29
30          return nextState;
31  }
```

First a uniform random distribution object is created. The constructor is called without any parameters, so the distribution covers the range 0 to 1. In the case that the simulation determines that the current state is not terminal, the algorithm loops through all transition probabilities of the current state. Again a random uniform distribution is generated to determine which state to visit next. The probabilities container which is used in the 'foreach' loop contains a so called key-value pair, the 'key' is the state and the 'value' is the probability of entering the state.

The outlined methods or functions represent the UCT algorithm implementation. The pseudo code description of the different methods show a very high level point of view, but hopefully provide the reader with enough understanding on how the algorithm works and help to understand the real implementation of the attached source code.

5.5 Model Based Agent

In this section of the document the 'ModelBasedAgent' class is described and explained. This class combines the model, the planning algorithms, and the real world (environment) to learn a model of the real world and to act on the learned model to find a way from the start state to the terminal state. The class can be seen as a container where all necessary parts to create a model based learning system are registered. Using a small example we illustrate how the different parts of the 'ModelBasedAgent' interoperate and how the user can set up experiments. To get an overview on how the different components work together please have a look to figure 5.2. As illustrated in figure 5.2, first of all the feature extraction (in this case SFA) is performed, which yields a user defined number of output features (already explained in the SFA sections). Then these features are used the create a discretized state space, this is done by the 'Environment' class. Next the state space is used by the model based agent to provide information like rewards, next states and so on. It can be also seen that the model as well as the planner directly communicate with the model based agent, which is necessary to keep all the data up to date. Now a pseudo code section illustrates how the model based agent communicates with the different modules. The class provides three public methods to the caller, which are sufficient to perform all operations that are necessary to drive a model based learning system.

- FirstAction: This method is the entry point for each model learning phase. The method is called only once, at the beginning of the process. The method returns the best action to take from the start state to the next state, and it takes a single parameter:
  - StartState: The start state of the agent.
- NextAction: This method usually gets executed in a loop. It returns the best action from the current state. The method calls a function to update the model based on the experiences in the real world. The methods takes two parameters:
  - Reward: The reward returned from the environment.

  - CurrentState: The current state of the agent.

- LastAction: This function is called after the terminal state was reached. The method updates the underlying model with the experiences of the last move to the terminal state. The method takes one parameter:
  - Reward: The reward returned from the environment when entering the terminal state.

The first method to describe is ’FirstAction’:

```
Action firstAction(int startState)
{
        % Call the planners method 'getBestAction'
        action = planner->getBestAction(startState);

        % Set global members for later processing
        previousState = startState;
        previousAction = action;

        % Return the selected action
        return action;
}
```

The above method uses the access to the planner (ValueIteration or UCT) class to retrieve the best action for the current state. Then the function sets some global variables for internal use, and it returns the selected action to the caller.
The following pseudo code describes the method ’NextAction’:

```
Action NextAction(double reward, int currentState)
{
        % Update the underlying model
        modelChanged = planner->UpdateModelWithExperience(
        previousState,
        previousAction,
        currentState,
        reward,
        false);

        % Update the model?
        if (modelChanged)
                planner->PlanOnNewModel();

        % New Action
        action = planner->getBestAction(currentState)

        % Set global members for later processing
        previousState = startState;
        previousAction = action;

        % Return the action
        return action;
}
```

The function above is used to retrieve the best action from the current state. The method has two parameters the ’reward’ and the ’currenState’, which are passed directly from the ’Environment’ class. First the function updates the underlying model, this is done by calling the function ’UpdateModelWithExperience’ which is provided by the planner. That way the model always gets the information of the real world, while it is explored by the agent. If the model has changed in the previous update step (for example if a new state was explored), the planner needs to consider this, which is why the function ’PlanOnNewModel’ is called. The planner (or agent) then tries to determine the best action for the current state, and

returns it to the caller. All the functions called by the above method are explained in the planning and model sub-sections of this chapter.
The next method is 'LastAction':

```
void LastAction(double reward)
{
        % Update the underlying model
        modelChanged = planner->UpdateModelWithExperience(
        previousState,
        previousAction,
        currentState,
        reward,
        false);

        % Update the model?
        if (modelChanged)
                planner->PlanOnNewModel();
}
```

The purpose of the above method is to update the underlying model with the experience of the agent after it found the terminal state. The function uses the same mechanism for updating the model as the previous one.
The final step is now to show how the class is used to perform a simulation run. A helper function which is created by the user of the class can be used to put all the components together, alternatively a method by the 'ModelBasedAgent' class itself can be used. In this case a helper function created by the caller was used, since it's easier to adapt parameters and it makes the model learning process a bit clearer to the reader, since the process is not hidden inside the class.

```
void PerformSimulation()
{
        % Get the start state.
        currentState = environment.GetState();
        action = agent.FirstAction(currentState);
        reward = environment.Apply(action);

        % Terminal state reached?
        while(!environment.IsTerminal() && steps < maxSteps)
        {
                currentState = environment.GetState();
                action = agent.NextAction(currentState);
                reward = environment.Apply(action);
                ++steps;
        }

    % Terminal/last state
        if (environment.isTerminal())
        {
                agent.LastAction(reward);
        }
}
```

The above function shows how to use all the described classes and methods to perform a simulation run for a model based learning system. The function first gets the start state from the environment, then the agent calculates the best action for the start state, the action is then applied to the environment calling the 'Apply' method. The 'Apply' method returns the reward the agent received by performing a certain action on the real environment. The while loop performs these steps until as the terminal state is reached or a user defined 'maxSteps' value is reached. This step limit avoids the system for hanging in an infinite loop forever in the case the terminal state cannot be reached. The final step is to update the model with the reward gathered from the terminal state, but only in the case the terminal state was reached.

5.6 Experiments and Results

In this section we perform a number of experiments and discuss our findings. The series of experiments starts with the theoretical example we already explained in the feature extraction section of the thesis. This simple 2D example is used to verify if the algorithms work properly. This phase can be seen as a proof of concept of the used methods and the implementation. Then an artificial 3D world is used to prove that the implemented algorithms can be used in a real environment. The implementation of the 3D environment is only described from a high level point of view since the generation of a 3D world in OpenGL is not part of the thesis. For the main part of the experiments the so called two room problem was chosen (explained in the feature extraction practical section). For the 3D environment a version with a smaller door was used to see the performance difference between the standard and the advanced version. Later an experiment with a moving terminal state is tested, to evaluate if the algorithms are able to handle this. All experiments are performed twice, using the two different planning algorithms Value-Iteration and UCT. The performance differences of the two algorithms are shown, and we conclude our experiment by discussing which algorithm performs best in the different situations.

5.6.1 2D Environment

In this section the proof of concept version of the two room problem is presented. This version of the environment is constructed in a way that all the used algorithms should perform reasonably well and the terminal state is found in every situation. The input to the feature extraction system can be found in the practical section of the SFA: see figure 4.9. The green rectangle shows the agent's start state and red rectangle shows the agent's terminal state. As it can be seen in the plot, it's a simple two room problem and the connection between the two rooms is pretty large, which should simplify the job of the algorithms to find their way to the terminal state. The images are saved in binary format. This means the pixel of the current position of the agent is one '1', all the other pixels are set to zero '0'. After the feature extraction process a 64 large feature space is available. This feature space needs to be discretized, which is done by the 'Environment' class described earlier in this chapter. The result can be seen in figure 5.1. As it can be seen there are exactly 64 discretized states available, some of them are ambiguous, but the differnt algorithms should be able to handle this problem. The planner (or agent) now needs to find its way through the different states. This is achieved by the reward system. This means if the agent performs a step (or action) in one direction it receives a reward of -1. If the agent hits the wall it receives a reward of -2. When the agent finds the terminal state it receives a reward of 0. This is a so called negative reward system. In all experiments the agent is able to perform four different actions (up, right, down and left). As already mentioned in all the different reference papers one step of the agent represents exactly one state, this is not the case here. We decided to perform three steps per movement, which leads to pretty good results and also shows that the agent is able to find his way even in very large states.

Value-Iteration Results

In this subsection the results of the Value-Iteration algorithm are presented. The following table lists the different parameters to the algorithm. Please see table 5.1 for the results of the experiments. It should be mentioned here that the Value-Iteration algorithm supports one additional parameter which is not listed in the table. The so called discount factor, which is set to 0.99, and stays for all experiments the same. By setting this value we tell the algorithm that we want to maximize the future rewards. As it can be seen in the table the differences between the experiments (or lines) is not as significant as expected. For example the first two lines of the table use a lot of iterations during the train phase, but the 'StepsToTerminal' value in the last column does not improve so significantly that the effort of increasing the train value by the factor of eight (comparing the first and the fifth line), is worth the effort. As visible in the first column, a state is usually considered as 'known' after 50 visits. For some cases 25 visits where sufficient

but with 50 visits the results where much more stable. That's quit a good value, especially when the agent operates in large state space environments, since reducing the value reduces the computational effort. To increase the value of the 'MaxSteps' does not make a lot of sense, since the maximum number of states is 64 even lower values than 100 should work fine, which is confirmed by the fourth experiment. It is also interesting that the algorithm independently of the parameters always finds the optimal or nearly optimal path to the terminal state.

In the next experiment the Q-Values for the different directions are calculated and plotted.

- Q-Values of direction left, please refer to figure 5.3.
- Q-Values of direction up, please refer to figure 5.4.
- Q-Values of direction right, please refer to figure 5.5.
- Q-Values of direction down, please refer to figure 5.6.

It can be seen in the plots that they are pretty similar, this is the case because of very small differences in the Q-Values for the different directions. The values most likely differ at the third of fourth position behind the decimal point, but for the algorithm these small differences are enough to decide which action to take. By taking a closer look at the plots, a trend to right bottom of the plot can be seen, this is exactly what can be expected since the terminal state is exactly located in the right bottom corner of the plot.

The final experiment in the Value-Iteration section is to get the maximum Q-Value of each and every state, which represents the best action taking form the current to the next state. A table is shown that represents the agent's path from the start to the terminal state, please see table 5.3. Finally a plot is presented which shows a direction vector in each state, this should lead to a path from the start to the terminal state, please see figure 5.7.

UCT Results

In this subsection the results of the UCT algorithm are presented. The following table lists the different parameters to the algorithm. A huge disadvantage of the UCT method is the runtime, since this algorithm is based on recursive calls. The execution time compared to the Value-Iteration method is much higher for the same parameters. The problem is that the UCT algorithm explores the state space more precise than Value-Iteration algorithm. The exploration can be controlled by the learning-rate parameter, the higher the value the less exploration. During the different experiments it surfaced that for our problem a small value provides better results. The decision was made to select the value 0.01 and keep it for all experiments. Please refer to table 5.2 for the results of the experiments. As it can be seen in the table, the results of the different runs are more or less the same, there is no difference in the results between 40000 iterations and 2750 iterations. It should be noted that the Value-Iteration algorithm is able to find the terminal state with fewer iterations than the UCT method (line 12 in table 5.1 for Value-Iteration and line 9 in table 5.2 for UCT). Both algorithms need 50 visits to consider a state as known which means that this value represents the smallest possible number of visits, which enables the agent to find the terminal state. This value is more determined by the underlying model, than by the model finding algorithms. When comparing Value-Iteration with UCT, Value-Iteration is clearly the preferred way to go in this setup. Value-Iteration requires fewer iterations to find the terminal state, the execution time is much lower than UCT's and Value-Iteration is a iterative algorithm and not a recursive one. All the above results and findings are based on the simple 2D maze. The image data used to represent this 2D maze are binary, which means the feature extraction methods produce a very clean separated state space. This can be an advantage for Value-Iteration method. In the following a more realistic 3D world is used, maybe these scenario changes the findings which have been made in this chapter.

NumberOfVisits	NumberOfActions	NumberOfEpisodes	MaxSteps	StepsToTerminal
1000	4	40000	1000	36
500	4	20000	500	36
100	4	10000	100	37
100	4	10000	70	37
50	4	5000	100	37
50	4	5000	70	37
25	4	5000	70	37
50	4	4500	70	37
25	4	4500	70	37
50	4	2500	70	37
25	4	2500	70	Not reached
50	4	1000	70	37
25	4	1000	70	Not reached
50	4	500	70	Not reached

Tabelle 5.1: The different parameters of the Value-Iteration algorithm. The last column of the table shows the performance of the algorithm depending on the parameters.
NumberOfVisits: How many visits are necessary to consider a state as known.
NumberOfActions: The number of actions (left, up, right and down).
NumberOfEpisodes: Can be considered as train phase, how often should the algorithm be executed on the state space.
MaxSteps: The maximum number of steps from the start to the terminal state.
StepToTerminal: Performance measurement, how many steps does the algorithm need to reach the terminal state.

In the next experiment the Q-Values for the different directions are calculated and plotted.

- Q-Values of direction left, please refer to figure 5.8.
- Q-Values of direction up, please refer to figure 5.9.
- Q-Values of direction right, please refer to figure 5.10.
- Q-Values of direction down, please refer to figure 5.11.

It can be seen that the plots differ much more than the plots of Value-Iteration. The absolute difference between the Q-Values of all action is higher than the differences calculated for the Value-Iteration method. For example in figure 5.8 it can be seen on the left room, that it makes absolutely no sense to take the left action, since the terminal state is on the right. In contrast it can be seen in figure 5.10, to perform action right in the left room makes perfect sense, only for the states that are directly located to the wall, it can be seen that action right is not the correct choice, since a hit to the wall is punished with a -2 reward.

The final experiment in the UCT section is to get the maximum Q-Value of each state, which corresponds to the best action taking from the current to the next state. A table is shown that represents the the agent's path from the start to the terminal state, please see table 5.4. Finally a plot is presented which shows a direction vector in each state, this should lead to a path from the start to the terminal state, please see figure 5.12.

5.6.2 3D Environment, Two Room problem

In this section a more realistic scenario is used to test the performance of the model based learning system. As already mentioned in the feature extraction practical section, an 3D OpenGL generated en-

NumberOfVisits	NumberOfActions	NumberOfEpisodes	MaxSteps	StepsToTerminal
1000	4	40000	1000	37
500	4	20000	500	37
100	4	10000	100	37
50	4	5000	100	37
50	4	5000	70	37
50	4	4500	70	36
50	4	3500	70	37
50	4	3000	70	37
50	4	2750	70	37
50	4	2500	70	Not reached
25	4	5000	70	37
25	4	4500	70	Not reached

Tabelle 5.2: The different parameters of the UCT algorithm. The last column of the table shows the performance of the algorithm depending on the parameters.
NumberOfVisits: How many visits are necessary to consider a state as known.
NumberOfActions: The number of actions (left, up, right and down).
NumberOfEpisodes: Can be considered as train phase, how often should the algorithm be executed on the state space.
MaxSteps: The maximum number of steps from the start to the terminal state.
StepToTerminal: Performance measurement, how many steps does the algorithm need to reach the terminal state.

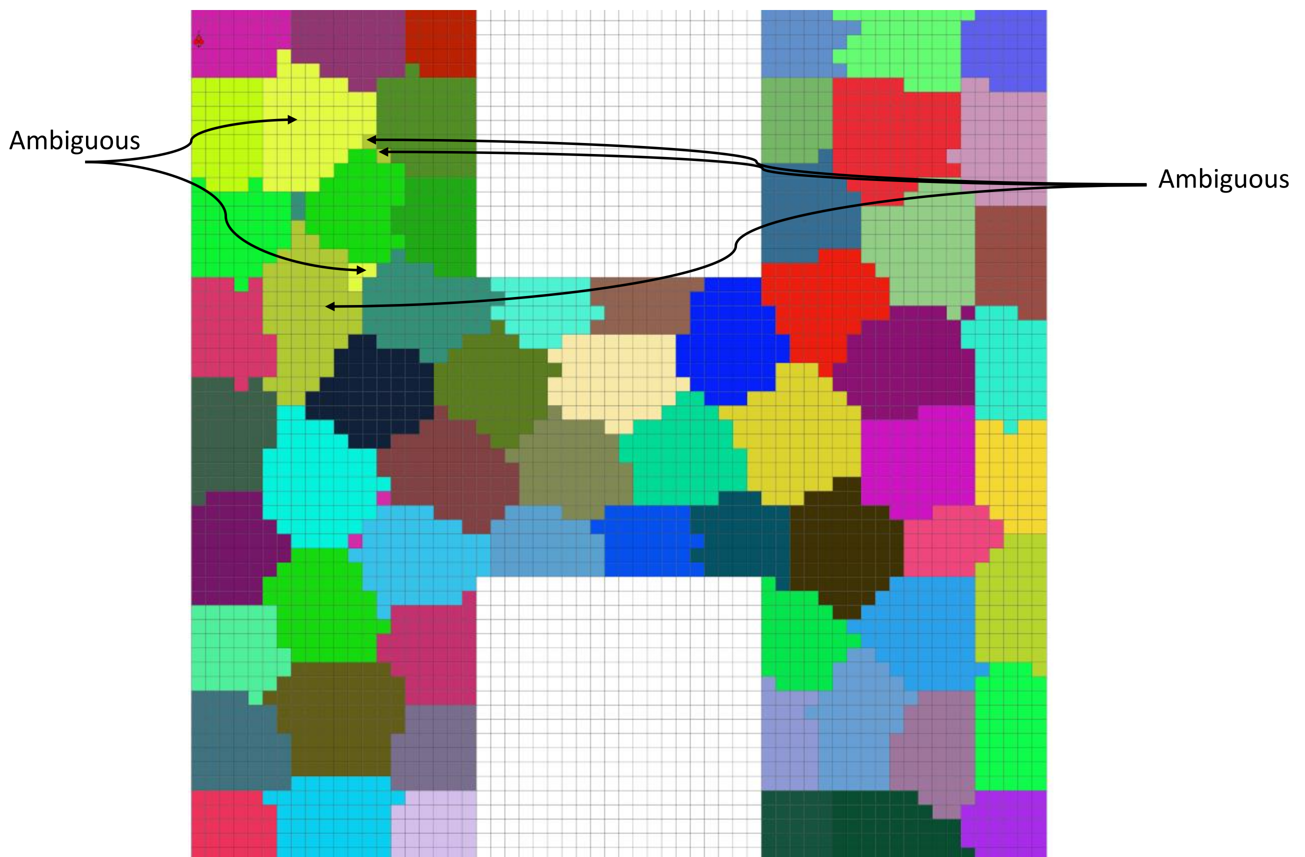

Abbildung 5.1: The discretized state space of the simple 2D two room problem. Each colored shape represents one state. We also show an example for an ambiguous states.

State	Q-Left	Q-Right	Q-Up	Q-Down	Action
11	-31.4234	-30.5706	-31.7274	-30.2933	Down
11	-31.4234	-30.5706	-31.7274	-30.2933	Down
38	-30.925	-29.6499	-30.5897	-29.8019	Right
38	-30.925	-29.6499	-30.5897	-29.8019	Right
48	-29.7116	-28.8952	-29.7173	-28.5135	Down
48	-29.7116	-28.8952	-29.7173	-28.5135	Down
48	-29.7116	-28.8952	-29.7173	-28.5135	Down
18	-29.1643	-27.9821	-29.4087	-28.225	Right
61	-27.9262	-26.5824	-27.3141	-27.6254	Right
61	-27.9262	-26.5824	-27.3141	-27.6254	Right
40	-26.7166	-27.1971	-27.4441	-25.8638	Down
59	-25.7389	-24.5087	-26.1854	-24.36	Down
59	-25.7389	-24.5087	-26.1854	-24.36	Down
20	-25.7411	-24.0545	-25.2844	-24.6619	Right
55	-23.5448	-21.8476	-23.4235	-22.2545	Right
55	-23.5448	-21.8476	-23.4235	-22.2545	Right
55	-23.5448	-21.8476	-23.4235	-22.2545	Right
25	-21.5254	-20.2276	-21.283	-20.3177	Right
25	-21.5254	-20.2276	-21.283	-20.3177	Right
25	-21.5254	-20.2276	-21.283	-20.3177	Right
58	-20.076	-18.522	-19.5551	-18.4357	Down
58	-20.076	-18.522	-19.5551	-18.4357	Down
37	-19.3703	-17.6292	-18.9553	-17.5109	Down
37	-19.3703	-17.6292	-18.9553	-17.5109	Down
8	-16.4301	-14.5474	-16.6001	-15.2918	Right
8	-16.4301	-14.5474	-16.6001	-15.2918	Right
57	-13.9803	-12.2669	-14.0973	-12.2089	Down
57	-13.9803	-12.2669	-14.0973	-12.2089	Down
19	-12.2708	-10.9611	-12.884	-10.856	Down
19	-12.2708	-10.9611	-12.884	-10.856	Down
60	-11.1732	-9.45067	-10.9618	-9.55183	Right
62	-10.1415	-8.21746	-9.86225	-8.42557	Right
62	-10.1415	-8.21746	-9.86225	-8.42557	Right
45	-8.00643	-5.56179	-7.99474	-6.33354	Right
45	-8.00643	-5.56179	-7.99474	-6.33354	Right
3	-6.51228	-5.71372	-6.48607	-4.11445	Down
3	-6.51228	-5.71372	-6.48607	-4.11445	Down
3	-6.51228	-5.71372	-6.48607	-4.11445	Down

Tabelle 5.3: The current state of the agent, the Q-Values (calculated by the Value-Iteration algorithm) for each action and the selected action in the state.

State	Q-Left	Q-Right	Q-Up	Q-Down	Action
11	-52.7719	-48.305	-48.6411	-34.9494	Down
11	-52.7719	-48.305	-48.6411	-34.9494	Down
38	-45.5855	-34.3515	-42.6424	-38.677	Right
38	-45.5855	-34.3515	-42.6424	-38.677	Right
48	-48.6529	-48.3843	-48.7664	-31.041	Down
48	-48.6529	-48.3843	-48.7664	-31.041	Down
48	-48.6529	-48.3843	-48.7664	-31.041	Down
18	-45.7466	-32.0732	-39.3385	-36.0233	Right
61	-43.9362	-29.4498	-43.6385	-45.2208	Right
61	-43.9362	-29.4498	-43.6385	-45.2208	Right
40	-35.4065	-44.5903	-43.4532	-28.2769	Down
59	-39.3007	-38.879	-40.1988	-28.2524	Down
59	-39.3007	-38.879	-40.1988	-28.2524	Down
20	-38.3297	-29.4198	-37.1129	-32.5523	Right
55	-35.8879	-24.4549	-36.2118	-30.0549	Right
55	-35.8879	-24.4549	-36.2118	-30.0549	Right
55	-35.8879	-24.4549	-36.2118	-30.0549	Right
25	-34.5633	-26.2622	-33.476	-24.0556	Down
25	-34.5633	-26.2622	-33.476	-24.0556	Down
53	-33.2424	-27.7707	-32.6924	-21.2243	Down
53	-33.2424	-27.7707	-32.6924	-21.2243	Down
22	-33.1675	-19.9209	-33	-35.0255	Right
5	-29.3997	-17.2767	-27.7963	-31.6041	Right
5	-29.3997	-17.2767	-27.7963	-31.6041	Right
8	-27.4565	-16.7121	-27.2437	-29.9018	Right
8	-27.4565	-16.7121	-27.2437	-29.9018	Right
57	-24.0737	-12.5562	-23.9108	-17.3104	Right
57	-24.0737	-12.5562	-23.9108	-17.3104	Right
57	-24.0737	-12.5562	-23.9108	-17.3104	Right
52	-18.841	-13.311	-19.2579	-9.70704	Down
52	-18.841	-13.311	-19.2579	-9.70704	Down
28	-20.2079	-14.1007	-21.8276	-9.31729	Down
28	-20.2079	-14.1007	-21.8276	-9.31729	Down
45	-12.2276	-1.25022	-12.1266	-7.6499	Right
45	-12.2276	-1.25022	-12.1266	-7.6499	Right
3	-14.4983	-19.3478	-14.6594	-0.0079222	Down
3	-14.4983	-19.3478	-14.6594	-0.0079222	Down
3	-14.4983	-19.3478	-14.6594	-0.0079222	Down

Tabelle 5.4: The current state of the agent, the Q-Values (calculated by the UCT algorithm) for each action and the selected action in the state.

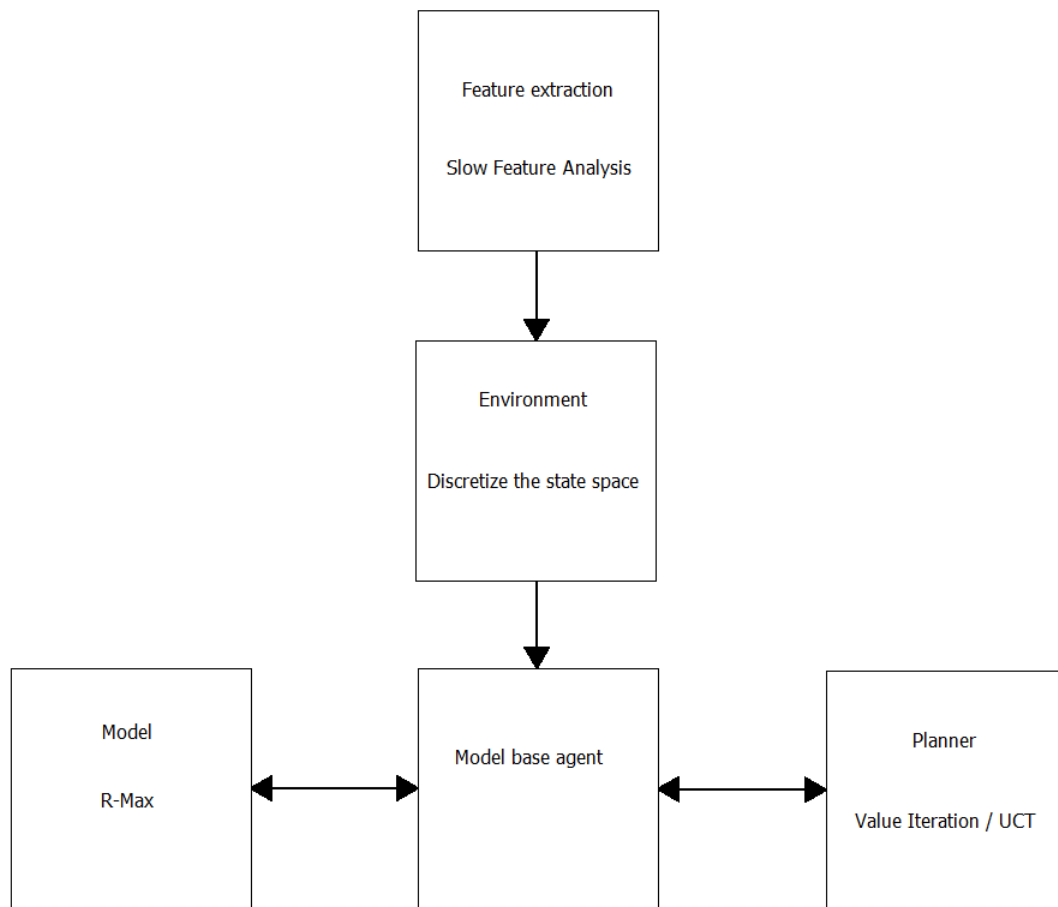

Abbildung 5.2: Overall architecture of the implemented model based learning system

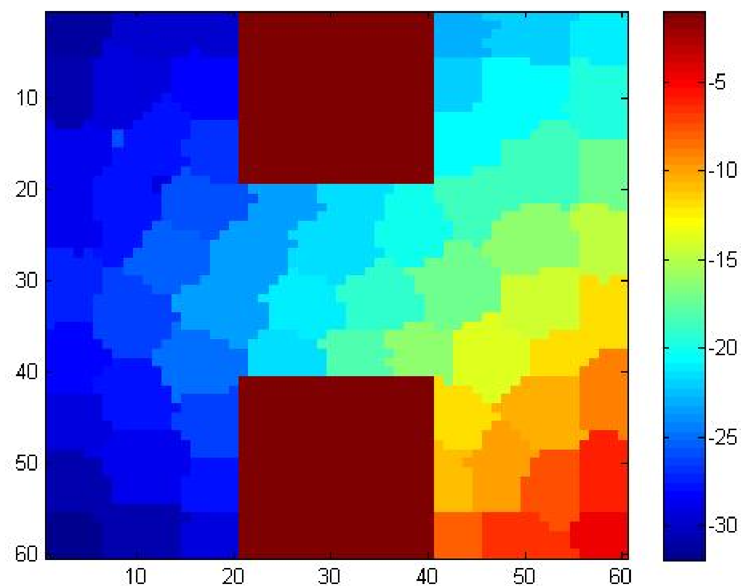

Abbildung 5.3: The Q-Values in left direction after trainings phase for the Value-Iteration algorithm. Each colored shape represents one state.

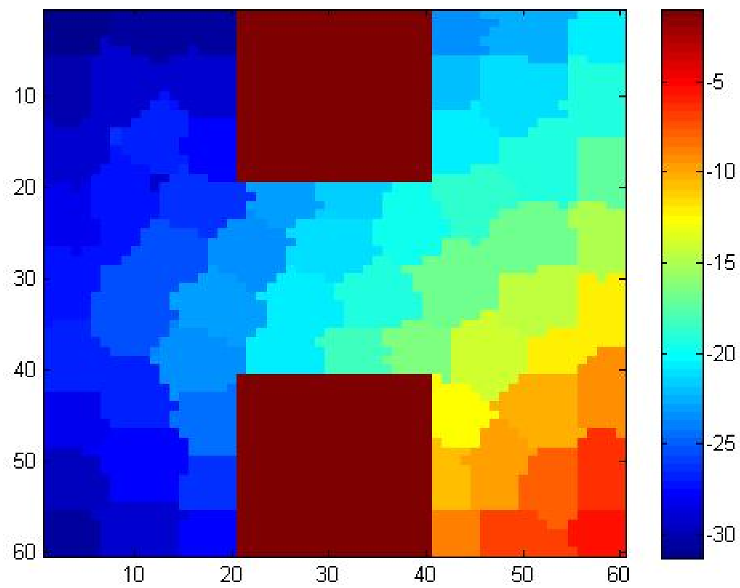

Abbildung 5.4: The Q-Values in up direction after trainings phase for the Value-Iteration algorithm. Each colored shape represents one state.

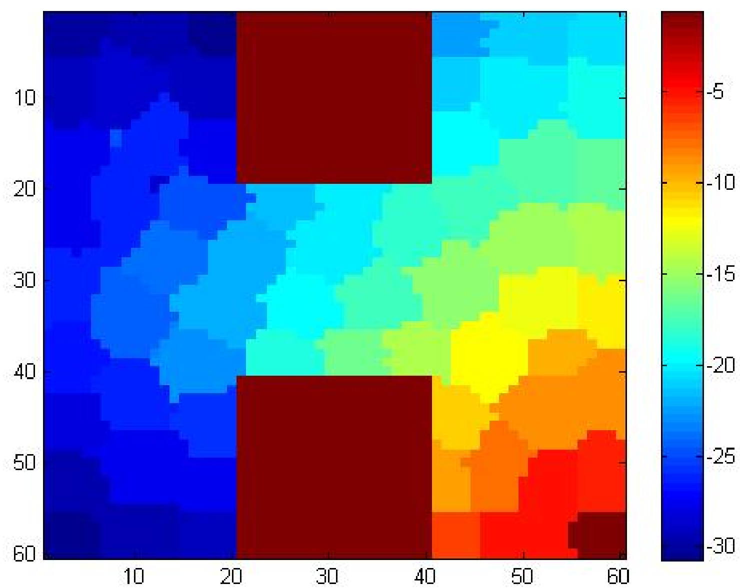

Abbildung 5.5: The Q-Values in right direction after trainings phase for the Value-Iteration algorithm. Each colored shape represents one state.

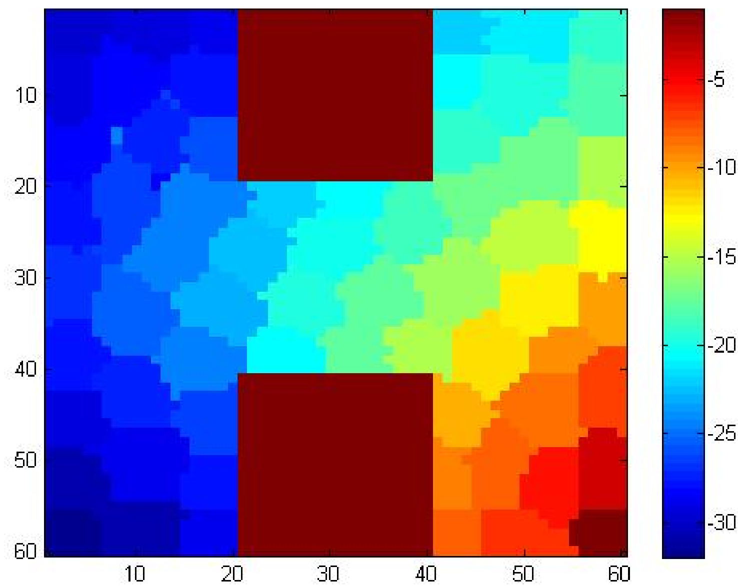

Abbildung 5.6: The Q-Values in down direction after trainings phase for the Value-Iteration algorithm. Each colored shape represents one state.

vironment is used to create a 3D version of the two room problem. The state space differs significantly from the artificial 2D version, in the sense that the state space itself is smaller (less than 64 features) but the states are larger, which means that the number of pixels covered by one state is lager compared to the 2D version. In the following the performance of the two algorithms will be compared, and the question will be answered if the two algorithms are even able to find a path in this difficult environment. The feature extraction process again provides 64 features, which are discretized by the 'Environment' class described earlier in this chapter. As it can be seen in figure 5.13 even if the feature extraction system was configured to provide 64 features. Fewer features were generated, which means that the information produced by 3D environment does not change so much from one point in time to the next. So the information stays more or less the same, that's why fewer features are necessary to represent them. The difficulty for the model learning algorithms is now to find an optimal path in an environment with fewer states, but these states are larger compared to the 2D version.

Value Iteration Results

In this subsection the results of the Value-Iteration algorithm are presented. The following tables show the different parameters to the algorithm and how the different parameters influence the performance of the algorithm. Please see table 5.5 for the results of the experiments. The number of visits column is not very surprising, 50 visits are necessary to create an accurate model of the environment, just like for the 2D example. The main difference between 2D and 3D environment can be seen in the number of episodes column. For the 2D world it was sufficient to perform 2500 episodes to generate a model which is able to find the path to the terminal state. In contrast 10000 steps were necessary in the 3D world. This result seems to be unexpected because of the relative small size of the state space (64 states in the 2D world and only 17 in the 3D world). The reason for the large number of episodes compared to the 2D environment, is the expansion of the different states, so for example in the 3D world the start state (4) contains 134 pixels, which is pretty large compared to 33 pixels in the 2D world. To produce comparable results the decision was made to not change the step width of the agent across all experiments, hence the agent performs three steps per action in the 2D world as well as in the 3D world. However, the perfor-

Abbildung 5.7: The Q-Value vectors, calculated by Value-Iteration algorithm, beginning at the start state, following these vectors the agent finally reaches the terminal state. Each colored shape represents one state.

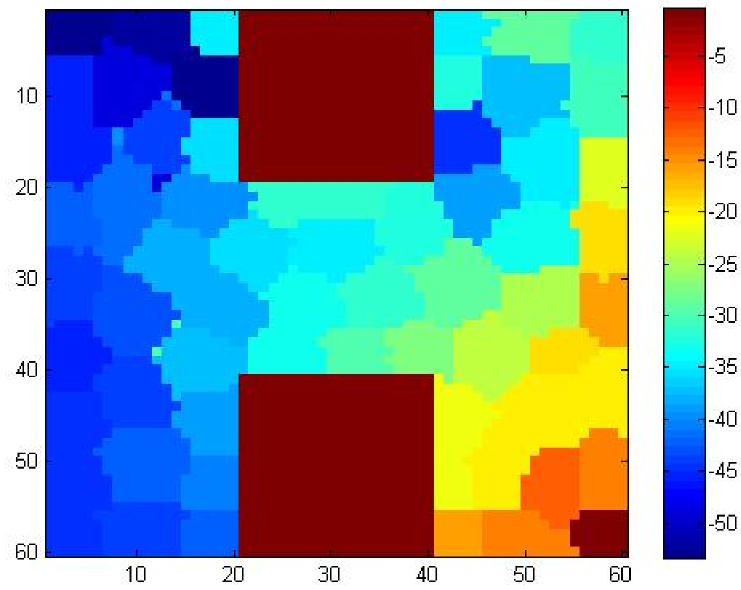

Abbildung 5.8: The Q-Values in left direction after trainings phase for the UCT algorithm. Each colored shape represents one state.

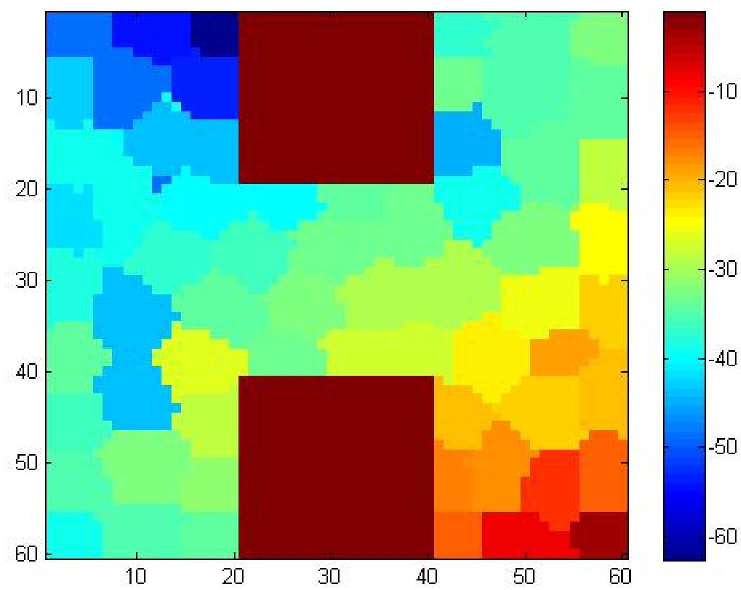

Abbildung 5.9: The Q-Values in up direction after trainings phase for the UCT algorithm. Each colored shape represents one state.

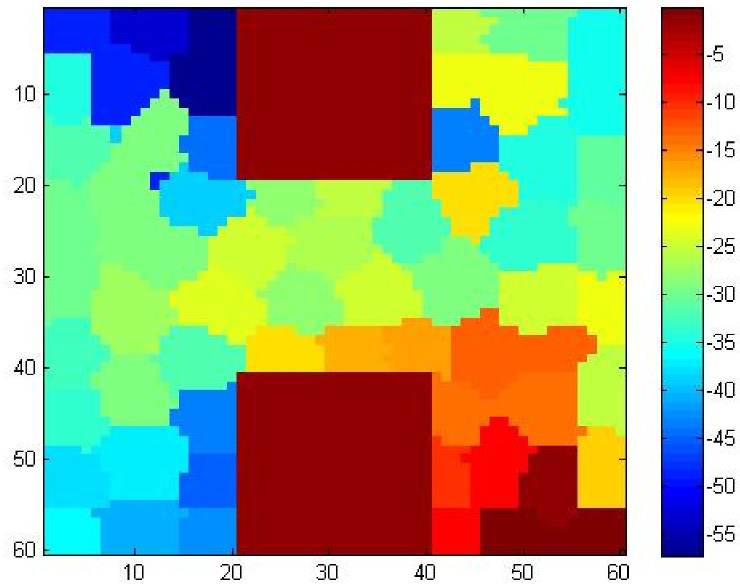

Abbildung 5.10: The Q-Values in right direction after trainings phase for the UCT algorithm. Each colored shape represents one state.

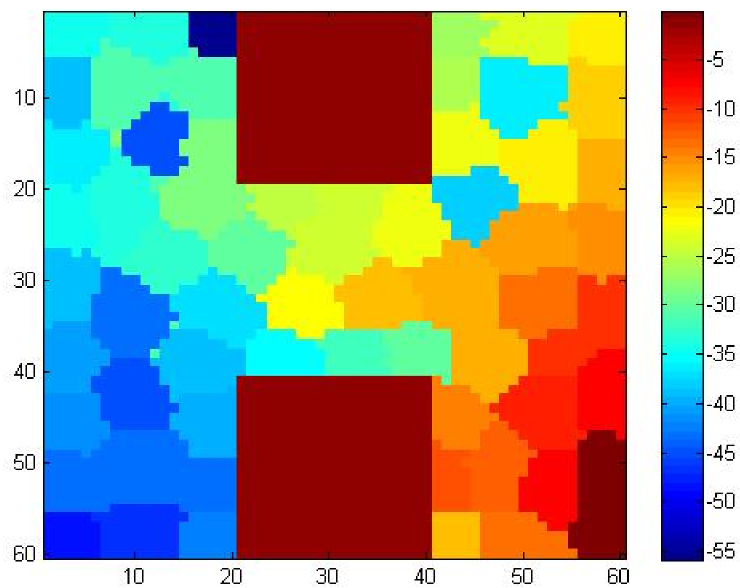

Abbildung 5.11: The Q-Values in down direction after trainings phase for the UCT algorithm. Each colored shape represents one state.

Abbildung 5.12: The Q-Value vectors, calculated by UCT algorithm, beginning at the start state, following these vectors the agent finally reaches the terminal state. Each colored shape represents one state.

mance of the algorithm is quite impressive: only 10000 steps are necessary, in a difficult state space, to produce very good results, which means the good performance of the Value-Iteration algorithm observed in the 2D world can also be seen in the 3D world.

In the next experiment the Q-Values for the different directions are calculated and plotted.

- Q-Values of direction left, please refer to figure 5.14.
- Q-Values of direction up, please refer to figure 5.15.
- Q-Values of direction right, please refer to figure 5.16.
- Q-Values of direction down, please refer to figure 5.17.

As already observed in the 2D experiment the Q-Values for the different actions only have small absolute differences. The values differ most likely at the third or fourth position behind the decimal point, but for the algorithm these small differences are sufficient to decide which action to take. By taking a closer look to the plots, a trend to the right bottom corner of the plot can be seen, which is exactly what can be expected since the terminal state is exactly located in the right bottom corner of the plot.

The final experiment in the Value-Iteration section is to get the maximum Q-Value of every state, which represents the best action form the current to the next state. A table is shown that represents the path of the agent from the start to the terminal state, please see table 5.6. Finally a plot is presented which shows a direction vector in each state, this should lead to a path from the start to the terminal state, please see figure 5.18.

UCT Results

In this subsection the results of the UCT algorithm are presented. Based on our 2D results, the expectation for the UCT algorithm is that the performance is worse than for the Value-Iteration. The only open question is: how will the UCT algorithm perform in a state space with only a few large states. It will be interesting to see whether UCT algorithm is able to outperform the Value-Iteration method in that case. Please see table 5.7 for the results of the experiments. It is interesting to see that the Value-Iteration algorithm needs at least 10000 epochs (or iterations) to find the solution, compared to UCT algorithm which only needs 6500 epochs to find the way from start to terminal state. Our results indicate that the UCT method is more convenient for state spaces with fewer large states (in the sense of number of pixels used by one state) than the Value-Iteration algorithm. However, the problem remains that the UCT algorithm is a recursive approach and as already mentioned the execution time compared to iterative methods (like Value-Iteration) is much higher. In other words the UCT method needs for 6500 episodes: 1950 sec or 32.5 min, where the Value-Iteration algorithm only requires 17 sec to execute 10000 episodes.

In the next experiment the Q-Values for the different directions are calculated and plotted.

- Q-Values of direction left, please refer to figure 5.19.
- Q-Values of direction up, please refer to figure 5.20.
- Q-Values of direction right, please refer to figure 5.21.
- Q-Values of direction down, please refer to figure 5.22.

In the 2D section of this chapter we discussed that the UCT method produces Q-Values, which differ much more than the Q-Values generated by Value-Iteration. This observation is also true for the 3D environment. So for example in figure 5.19, it can be seen that there is no reason for the algorithm to take

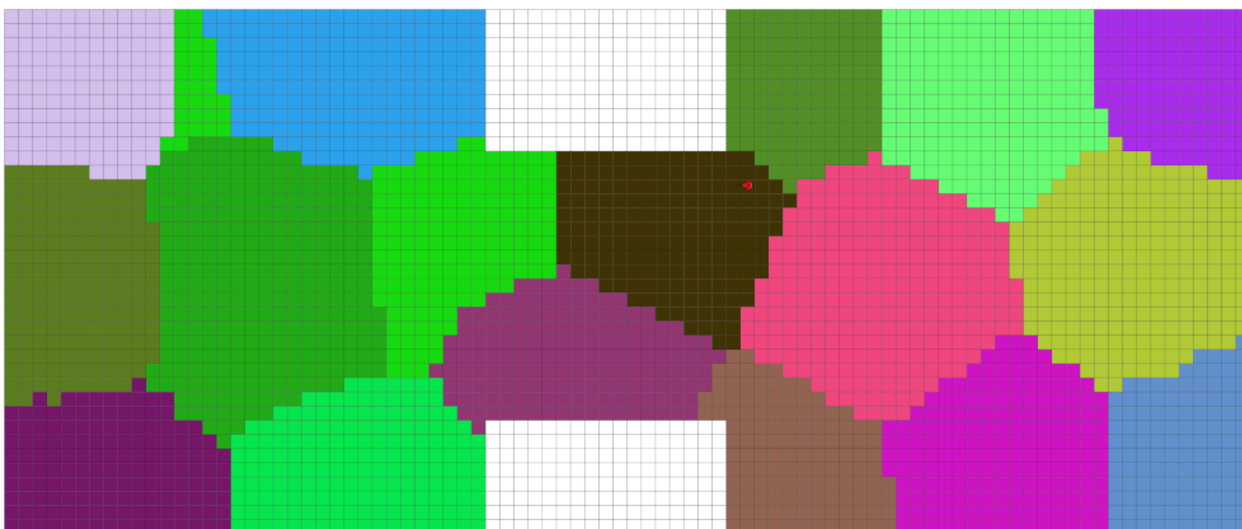

Abbildung 5.13: The discretized state space of the 3D two room problem. Each colored shape represents one state.

the left action on the left side of the room. In contrast in figure 5.21, it can be seen that taking the right action on the left hand side of the room makes sense, since the terminal state is on the right bottom in the right side of the room. For states that are directly located next to the wall, it can be seen that taking the right action is not the preferred action, since hitting the wall is punished with a -2 reward.

The final experiment in the UCT section is to get the maximum Q-Value of each state. This value represents the best action form the current to the next state. table 5.8 illustrates the agent's path from the start to the terminal state. Finally a plot is presented which shows a direction vector in each state, this should yield a path from the start to the terminal state, please see figure 5.23.

5.6.3 3D Environment, Two Room Problem With Smaller Passage

The following experiment is a slightly modified version of the 3D two room problem discussed in the previous section. In contrast to the normal two room problem the overall environment is larger, and the passage between the two rooms is narrower. Due to limitations of the OpenGL image generation system, the wall segments always have the same size. It was necessary to make the overall environment larger, to make the passage smaller. Otherwise we would have seen two separated rooms with no passage. One consequence of these changes is that the agent needs more steps to reach the terminal state. The main question is: does the narrow passage between the two rooms impact the performance of the agent, or is it even possible for the agent to reach the terminal state? In the following these questions will be answered, using the two approaches Value-Iteration and UCT. As usual the performance of the two algorithms is compared and plots and tables are used to visualize the results. The SFA again provides 64 features and this feature space needs to be discretized, which is done by the 'Environment' class described earlier in this chapter. Figure: 5.24 illustrates that even if the feature extraction system was configured to provide 64 features, fewer features are generated, as we already discussed in the previous section. As usual the Value-Iteration method is the one to start with.

Value-Iteration Results

In this section the results of the Value-Iteration algorithm are presented, the following table lists the different parameters to the algorithm and how the different parameters influence the performance of the algorithm. Please see table: 5.9 for the results of the experiments. The most important observation regar-

NumberOfVisits	NumberOfActions	NumberOfEpisodes	MaxSteps	StepsToTerminal
1000	4	40000	1000	34
500	4	20000	500	34
100	4	10000	100	34
100	4	10000	70	34
50	4	5000	70	Not reached
50	4	7500	70	Not reached
50	4	10000	50	34
25	4	10000	50	Not reached

Tabelle 5.5: The different parameters of the Value-Iteration algorithm in a 3D two room environment. The last column of the table shows the performance of the algorithm depending on the parameters. The agent performs three steps per action.
NumberOfVisits: How many visits are necessary to consider a state as known.
NumberOfActions: The number of actions (left, up, right and down).
NumberOfEpisodes: Can be considered as train phase, how often should the algorithm be executed on the state space.
MaxSteps: The maximum number of steps from the start to the terminal state.
StepToTerminal: Performance measurement, how many steps does the algorithm need to reach the terminal state.

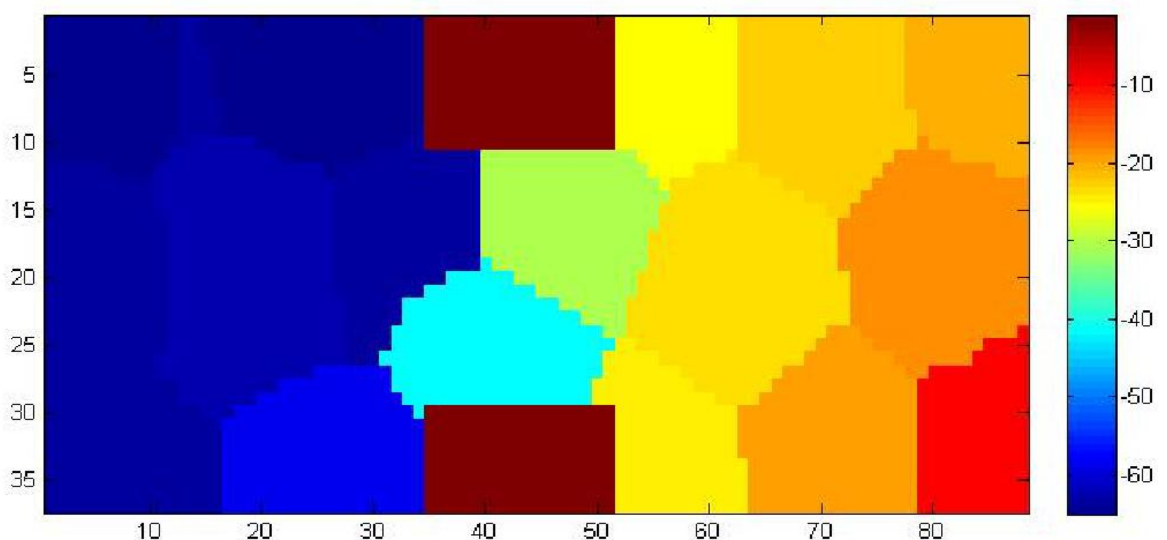

Abbildung 5.14: The Q-Values in left direction after trainings phase for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

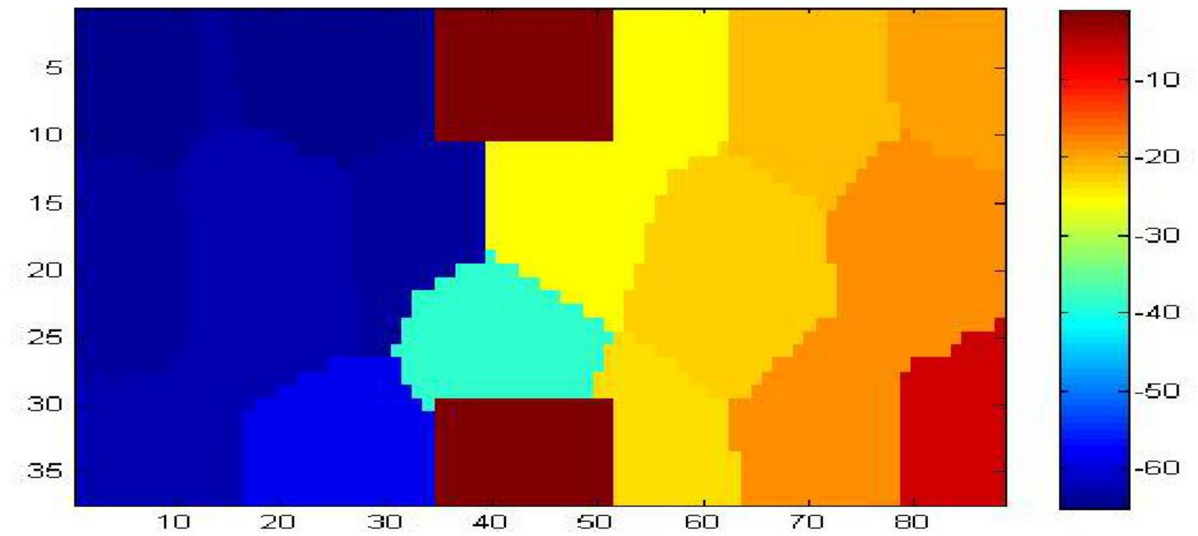

Abbildung 5.15: The Q-Values in up direction after trainings phase for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

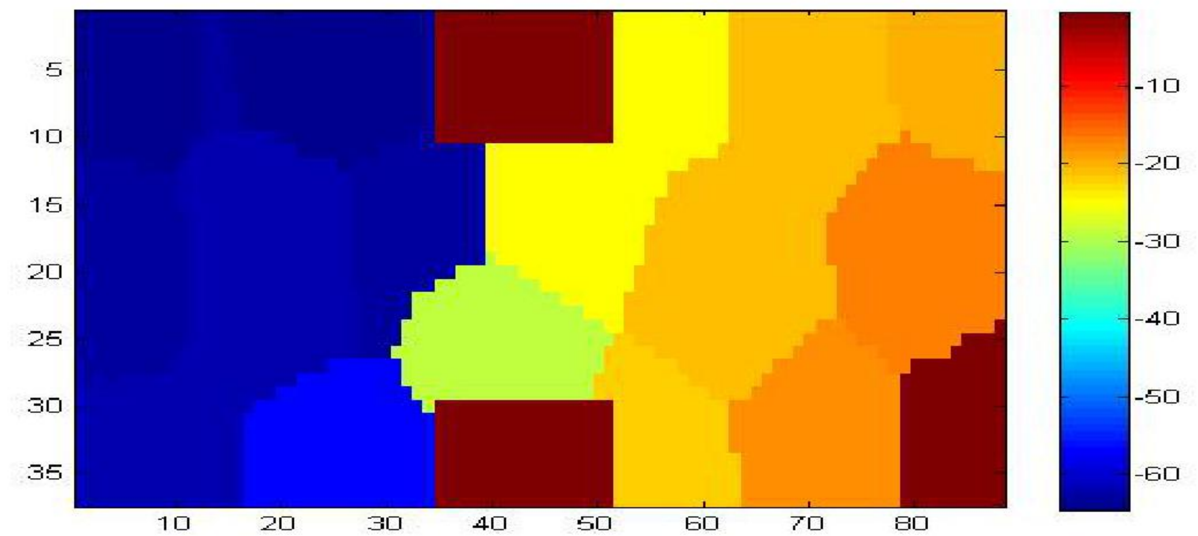

Abbildung 5.16: The Q-Values in right direction after trainings phase for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

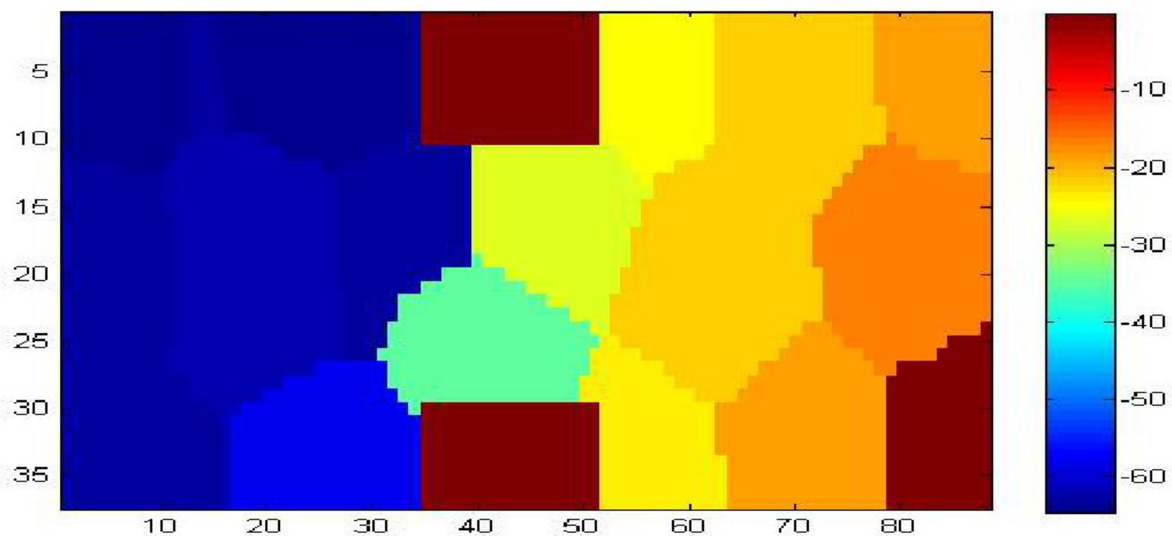

Abbildung 5.17: The Q-Values in down direction after trainings phase for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

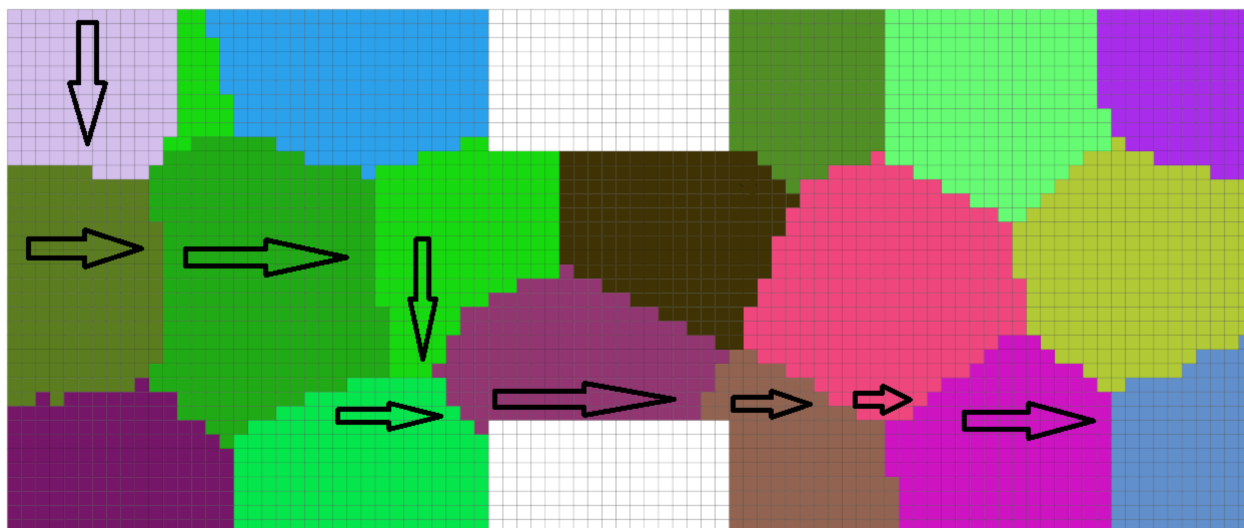

Abbildung 5.18: The discretized state space of the 3D two room problem and the Q-Value direction vectors. Each colored shape represents one state.

State	Q-Left	Q-Right	Q-Up	Q-Down	Action
4	-65.0699	-64.5594	-65.1099	-64.5352	Down
4	-65.0699	-64.5594	-65.1099	-64.5352	Down
4	-65.0699	-64.5594	-65.1099	-64.5352	Down
4	-65.0699	-64.5594	-65.1099	-64.5352	Down
55	-63.7058	-63.1574	-63.8259	-63.4648	Right
55	-63.7058	-63.1574	-63.8259	-63.4648	Right
55	-63.7058	-63.1574	-63.8259	-63.4648	Right
55	-63.7058	-63.1574	-63.8259	-63.4648	Right
40	-62.8383	-61.9384	-62.6737	-62.2499	Right
40	-62.8383	-61.9384	-62.6737	-62.2499	Right
40	-62.8383	-61.9384	-62.6737	-62.2499	Right
40	-62.8383	-61.9384	-62.6737	-62.2499	Right
40	-62.8383	-61.9384	-62.6737	-62.2499	Right
61	-63.6803	-62.9199	-63.7264	-62.8662	Down
61	-63.6803	-62.9199	-63.7264	-62.8662	Down
61	-63.6803	-62.9199	-63.7264	-62.8662	Down
61	-63.6803	-62.9199	-63.7264	-62.8662	Down
61	-63.6803	-62.9199	-63.7264	-62.8662	Down
19	-58.5149	-57.1015	-58.1675	-57.7705	Right
19	-58.5149	-57.1015	-58.1675	-57.7705	Right
29	-41.8817	-29.375	-38.7228	-35.2022	Right
29	-41.8817	-29.375	-38.7228	-35.2022	Right
29	-41.8817	-29.375	-38.7228	-35.2022	Right
29	-41.8817	-29.375	-38.7228	-35.2022	Right
29	-41.8817	-29.375	-38.7228	-35.2022	Right
29	-41.8817	-29.375	-38.7228	-35.2022	Right
23	-24.2909	-22.1191	-23.6713	-23.3179	Right
23	-24.2909	-22.1191	-23.6713	-23.3179	Right
23	-24.2909	-22.1191	-23.6713	-23.3179	Right
52	-23.5992	-20.923	-22.7599	-21.5981	Right
52	-23.5992	-20.923	-22.7599	-21.5981	Right
41	-19.8421	-17.696	-18.7469	-18.7991	Right
41	-19.8421	-17.696	-18.7469	-18.7991	Right
41	-19.8421	-17.696	-18.7469	-18.7991	Right
41	-19.8421	-17.696	-18.7469	-18.7991	Right

Tabelle 5.6: The current state of the agent, the Q-Values (calculated by the Value-Iteration algorithm) for each action and the selected action in the state.

NumberOfVisits	NumberOfActions	NumberOfEpisodes	MaxSteps	StepsToTerminal
1000	4	40000	1000	34
500	4	20000	90	34
100	4	10000	90	35
100	4	10000	70	35
50	4	7500	70	34
50	4	6500	50	34
50	4	5000	70	Not reached

Tabelle 5.7: The different parameters of the UCT algorithm in a 3D two room environment. The last column of the table shows the performance of the algorithm depending on the parameters. The agent performs three steps per action.
NumberOfVisits: How many visits are necessary to consider a state as known.
NumberOfActions: The number of actions (left, up, right and down).
NumberOfEpisodes: Can be considered as train phase, how often should the algorithm be executed on the state space.
MaxSteps: The maximum number of steps from the start to the terminal state.
StepToTerminal: Performance measurement, how many steps does the algorithm need to reach the terminal state.

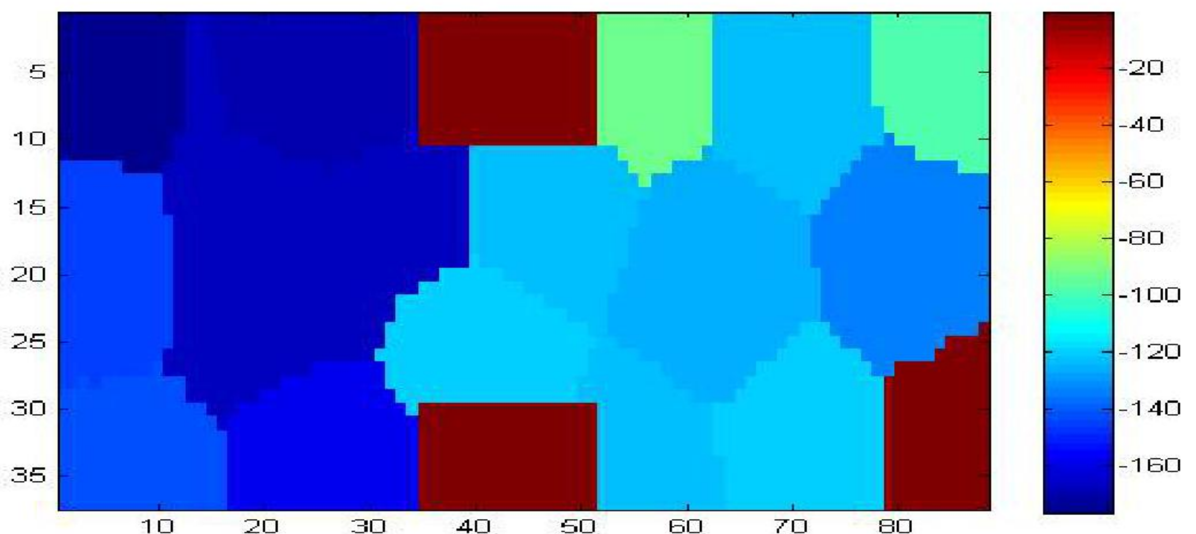

Abbildung 5.19: The Q-Values in left direction after trainings phase for the UCT algorithm in a 3D environment. Each colored shape represents one state.

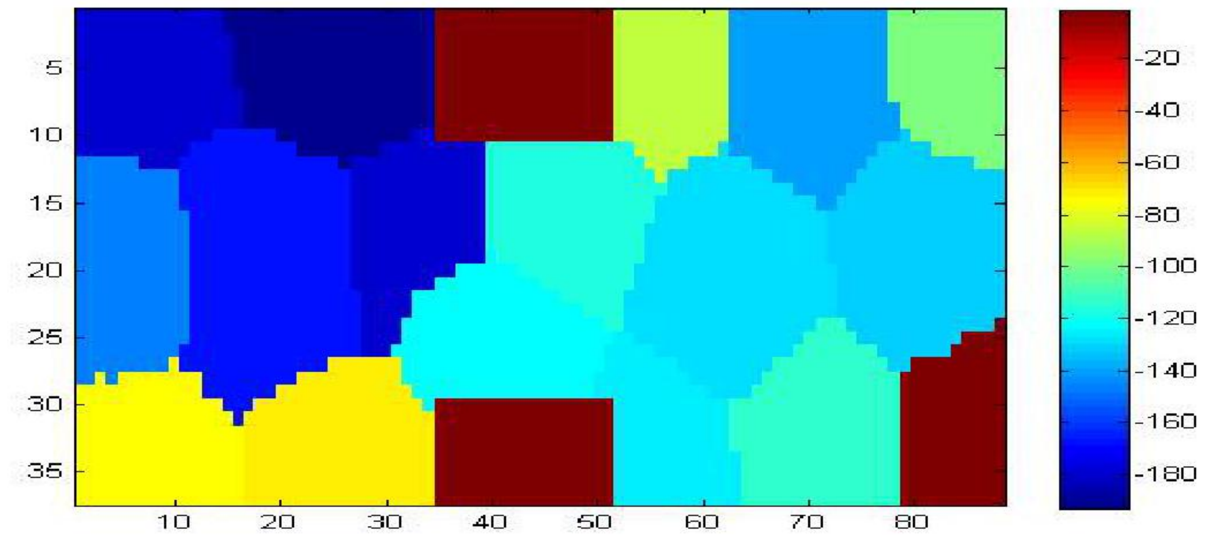

Abbildung 5.20: The Q-Values in up direction after trainings phase for the UCT algorithm in a 3D environment. Each colored shape represents one state.

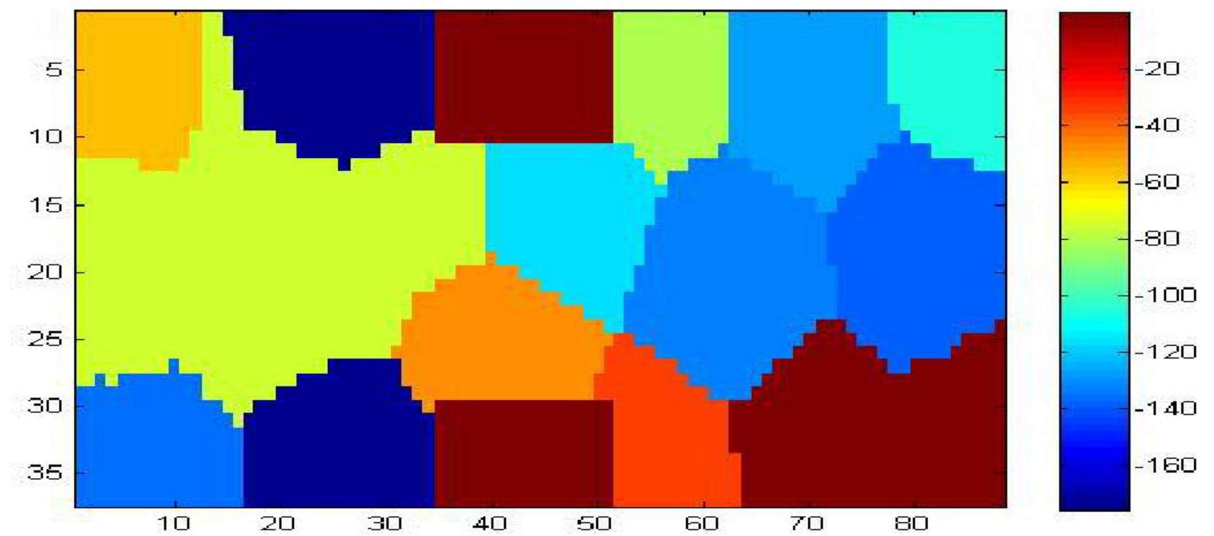

Abbildung 5.21: The Q-Values in right direction after trainings phase for the UCT algorithm in a 3D environment. Each colored shape represents one state.

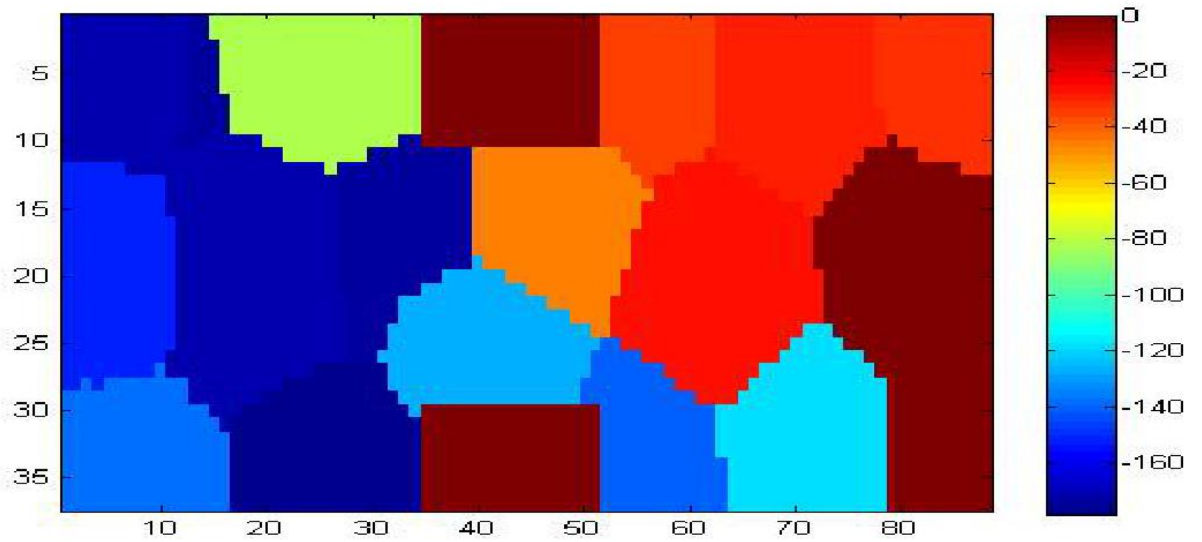

Abbildung 5.22: The Q-Values in down direction after trainings phase for the UCT algorithm in a 3D environment. Each colored shape represents one state.

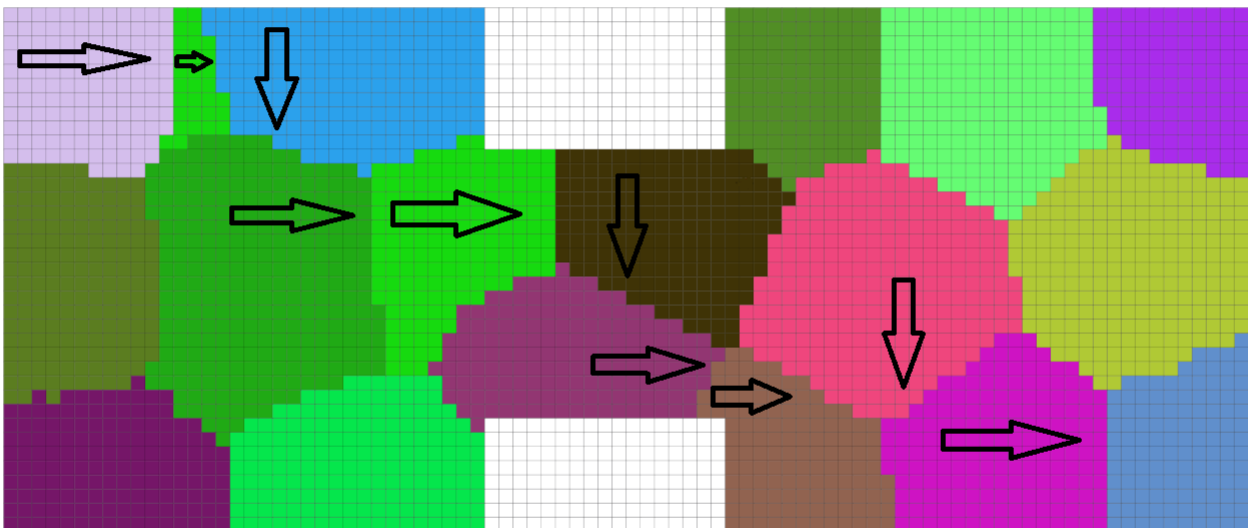

Abbildung 5.23: The discretized state space of the 3D two room problem and the Q-Value direction vectors. Each colored shape represents one state.

State	Q-Left	Q-Right	Q-Up	Q-Down	Action
4	-176.786	-57.0736	-180.125	-170.365	Right
4	-176.786	-57.0736	-180.125	-170.365	Right
4	-176.786	-57.0736	-180.125	-170.365	Right
4	-176.786	-57.0736	-180.125	-170.365	Right
61	-167.608	-76.7687	-178.374	-173.87	Right
28	-170.341	-175.475	-193.089	-81.6427	Down
28	-170.341	-175.475	-193.089	-81.6427	Down
61	-167.608	-76.7687	-178.374	-173.87	Right
28	-170.341	-175.475	-193.089	-81.6427	Down
40	-166.466	-74.3111	-168.636	-171.423	Right
28	-170.341	-175.475	-193.089	-81.6427	Down
40	-166.466	-74.3111	-168.636	-171.423	Right
40	-166.466	-74.3111	-168.636	-171.423	Right
61	-167.608	-76.7687	-178.374	-173.87	Right
61	-167.608	-76.7687	-178.374	-173.87	Right
61	-167.608	-76.7687	-178.374	-173.87	Right
61	-167.608	-76.7687	-178.374	-173.87	Right
57	-123.319	-115.718	-115.76	-44.7235	Down
57	-123.319	-115.718	-115.76	-44.7235	Down
29	-119.338	-49.0702	-121.521	-126.362	Right
57	-123.319	-115.718	-115.76	-44.7235	Down
29	-119.338	-49.0702	-121.521	-126.362	Right
29	-119.338	-49.0702	-121.521	-126.362	Right
57	-123.319	-115.718	-115.76	-44.7235	Down
29	-119.338	-49.0702	-121.521	-126.362	Right
23	-123.463	-35.1656	-125.034	-142.071	Right
52	-124.892	-132.724	-128.609	-25.8431	Down
23	-123.463	-35.1656	-125.034	-142.071	Right
23	-123.463	-35.1656	-125.034	-142.071	Right
52	-124.892	-132.724	-128.609	-25.8431	Down
23	-123.463	-35.1656	-125.034	-142.071	Right
41	-119.768	-0.599535	-112.482	-119.07	Right
41	-119.768	-0.599535	-112.482	-119.07	Right
41	-119.768	-0.599535	-112.482	-119.07	Right
41	-119.768	-0.599535	-112.482	-119.07	Right
41	-119.768	-0.599535	-112.482	-119.07	Right

Tabelle 5.8: The current state of the agent, the Q-Values (calculated by the UCT algorithm) for each action and the selected action in the state.

ding table 5.9 is that the algorithm is able to find the terminal state. The performance is comparable to the original 3D two room problem, except that the algorithm requires more steps to reach the the terminal state. However, the reason for this is not the smaller passage, it's the larger environment. As already mentioned the results are comparable to the previous two room problem, which is not very surprising since the problem to solve does not differ so much from the previous one. One important observation is that the algorithm is able to handle ambiguous states very well, even if they are pretty large.

In the next experiment the Q-Values for the different directions are calculated and plotted.

- Q-Values of direction left, please refer to figure 5.25.
- Q-Values of direction up, please refer to figure 5.26.
- Q-Values of direction right, please refer to figure 5.27.
- Q-Values of direction down, please refer to figure 5.28.

The results of the Q-Value plots reflect the observations that have been made for the two room problem in the previous section. Small differences in the Q-Value plots, but these small differences are enough to find the path from the start to the terminal state. For a more detailed discussion of the results please refer to the Value-Iteration section.

The final experiment in the Value-Iteration section is to get the maximum Q-Value of each and every state, which represents the best action form the current to the next state. A table is shown that represents agent's path from the start to the terminal state, please see table 5.10. Finally a plot is presented which shows a direction vector in each state, this should lead to a path from the start to the terminal state, please see figure 5.29.

UCT Results

This experiment showed that the UCT algorithm simply was not able to find the terminal state. A lot of different parameters were tested even absurd search tree depths like 1000 but none of them were successful. A problem with UCT method and testing a lot of different parameters is the time consumption of the algorithm. Experiments with quite a lot of iterations or high search tree depths easily take 10 hours or more, which is not very practical. The reasons why the method is not able to find the terminal state is may be that one of the states is ambiguous or that the states are too large, but both of the problems already occurred in earlier experiments, where UCT algorithm successfully found the terminal state. In the previous two room problem we observed that the UCT method behaves better in situations where the state space is small but the single states are larger, but only if there were no ambiguous states present. Maybe this is the reason for the algorithm's failure in this example. However the conclusion is that the overall performance of the Value-Iteration method seems to be better than the performance of the UCT method. Aside from the fact that the Value-Iteration algorithm was able to find the terminal state in every experiment, it also was the much faster method (no experiment took more than 30 minutes, even these with an absurdly high number of iterations), and the task to find good parameters was much easier compared to UCT algorithm. The overall outcome of all these experiments is that the combination Value-Iteration algorithm and R-Max model fit best for solving the different two room problems of this thesis.

5.6.4 3D Environment, Two Room problem with movable terminal state

In this section we first train on the standard 3D two room problem, after the terminal state is found, the terminal is moved to a different state. The question now is: Is the algorithm able to find the new terminal

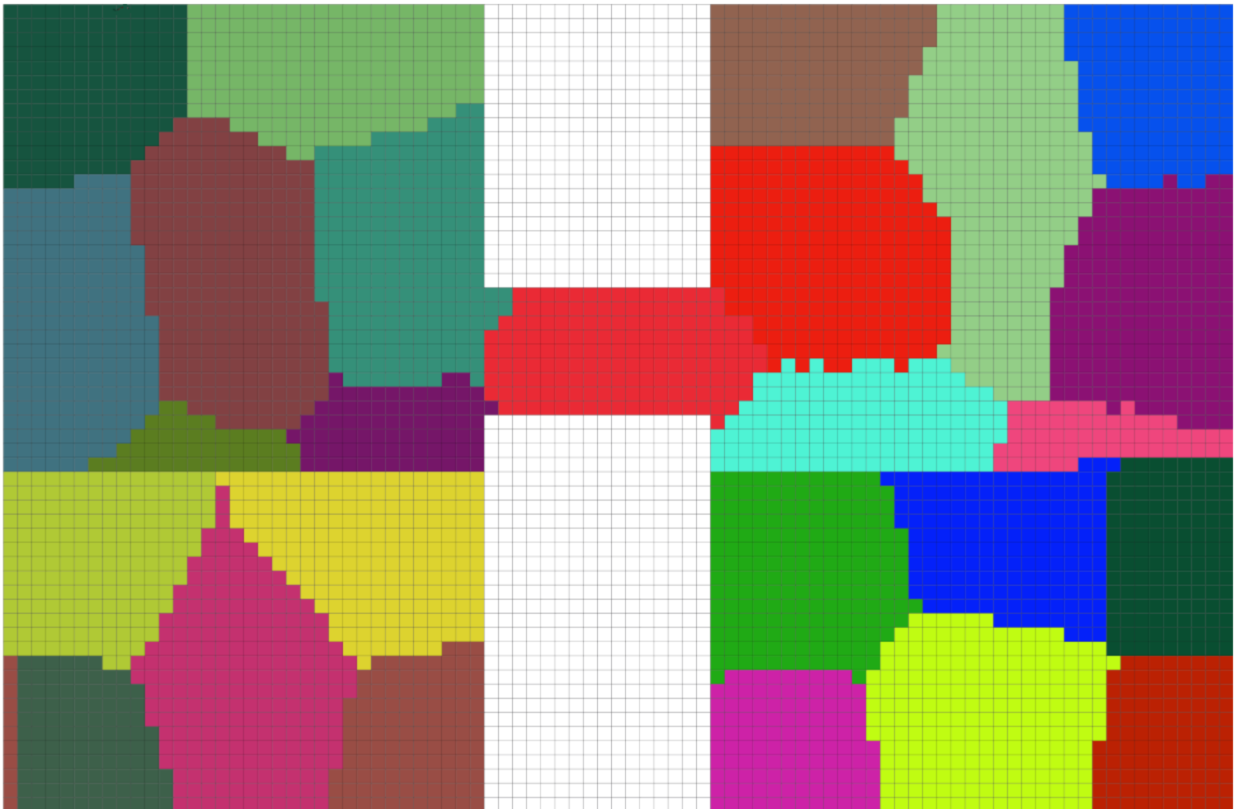

Abbildung 5.24: The discretized state space of the 3D two room problem with smaller passage. Each colored shape represents one state.

NumberOfVisits	NumberOfActions	NumberOfEpisodes	MaxSteps	StepsToTerminal
2000	4	60000	1000	41
1000	4	40000	500	41
500	4	20000	350	41
100	4	10000	350	41
50	4	10000	250	41
50	4	10000	150	41
50	4	10000	100	Not reached
50	4	7500	150	41
50	4	5000	150	Not reached
25	4	7500	150	Not reached

Tabelle 5.9: The different parameters of the Value-Iteration algorithm. The last column of the table shows the performance of the algorithm depending on the parameters.
NumberOfVisits: How many visits are necessary to consider a state as known.
NumberOfActions: The number of actions (left, up, right and down).
NumberOfEpisodes: Can be considered as train phase, how often should the algorithm be executed on the state space.
MaxSteps: The maximum number of steps from the start to the terminal state.
StepToTerminal: Performance measurement, how many steps does the algorithm need to reach the terminal state.

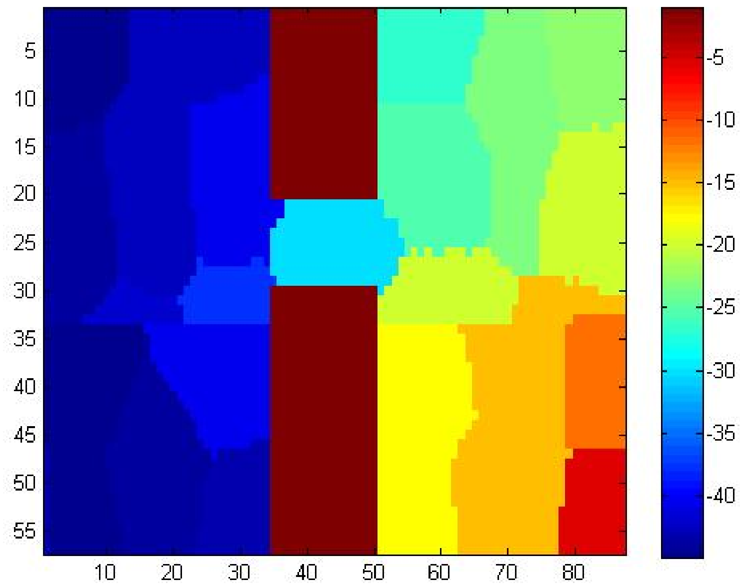

Abbildung 5.25: The Q-Values in left direction after trainings phase for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

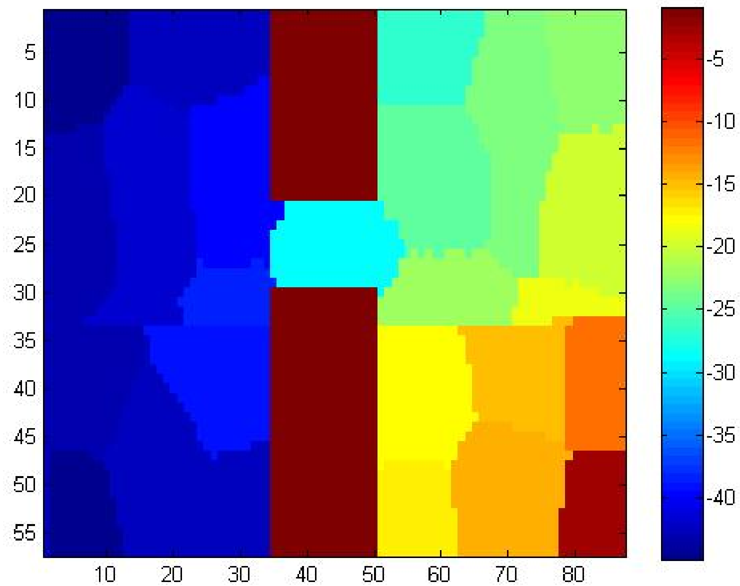

Abbildung 5.26: The Q-Values in up direction after trainings phase for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

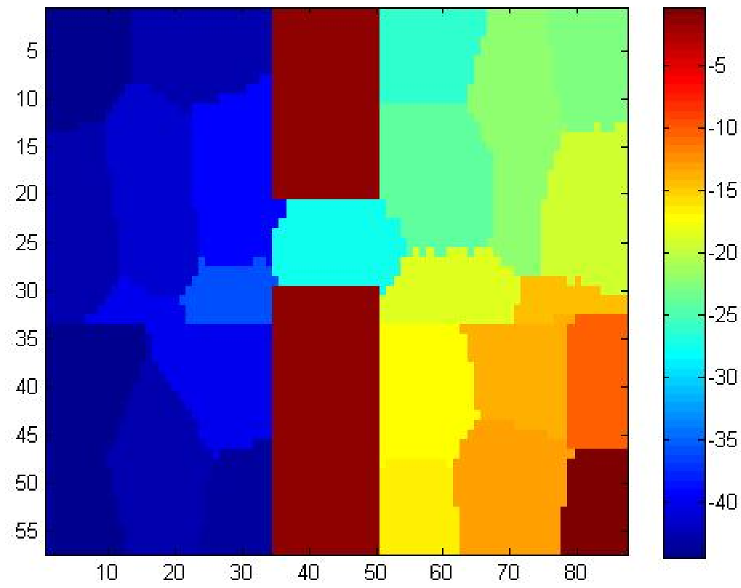

Abbildung 5.27: The Q-Values in right direction after trainings phase for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

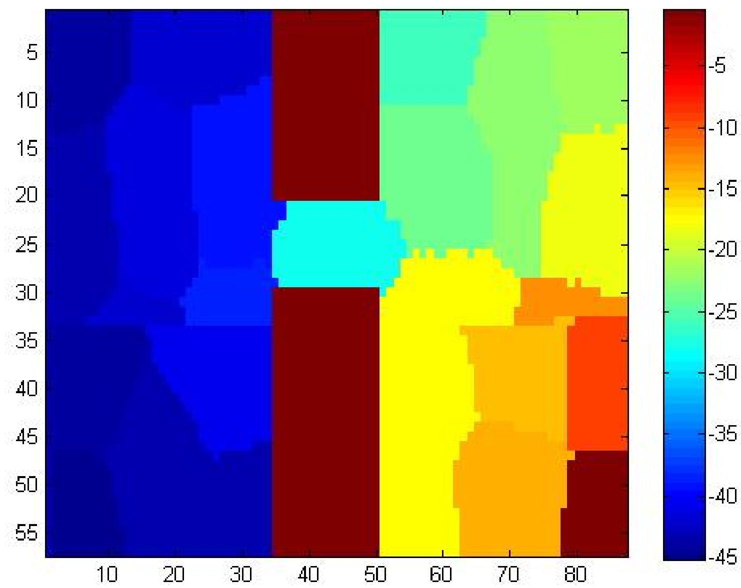

Abbildung 5.28: The Q-Values in down direction after trainings phase for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

State	Q-Left	Q-Right	Q-Up	Q-Down	Action
10	-44.9324	-44.1206	-44.9424	-44.182	Right
10	-44.9324	-44.1206	-44.9424	-44.182	Right
10	-44.9324	-44.1206	-44.9424	-44.182	Right
10	-44.9324	-44.1206	-44.9424	-44.182	Right
10	-44.9324	-44.1206	-44.9424	-44.182	Right
30	-42.6643	-42.3558	-42.755	-41.8445	Down
30	-42.6643	-42.3558	-42.755	-41.8445	Down
30	-42.6643	-42.3558	-42.755	-41.8445	Down
42	-42.2162	-41.2058	-42.0225	-41.5538	Right
30	-42.6643	-42.3558	-42.755	-41.8445	Down
42	-42.2162	-41.2058	-42.0225	-41.5538	Right
42	-42.2162	-41.2058	-42.0225	-41.5538	Right
59	-40.2316	-38.9601	-40.1095	-38.9535	Down
59	-40.2316	-38.9601	-40.1095	-38.9535	Down
59	-40.2316	-38.9601	-40.1095	-38.9535	Down
59	-40.2316	-38.9601	-40.1095	-38.9535	Down
59	-40.2316	-38.9601	-40.1095	-38.9535	Down
16	-37.7407	-35.925	-38.1956	-38.2203	Right
16	-37.7407	-35.925	-38.1956	-38.2203	Right
16	-37.7407	-35.925	-38.1956	-38.2203	Right
16	-37.7407	-35.925	-38.1956	-38.2203	Right
31	-29.9563	-27.617	-28.8193	-28.3391	Right
31	-29.9563	-27.617	-28.8193	-28.3391	Right
31	-29.9563	-27.617	-28.8193	-28.3391	Right
31	-29.9563	-27.617	-28.8193	-28.3391	Right
31	-29.9563	-27.617	-28.8193	-28.3391	Right
31	-29.9563	-27.617	-28.8193	-28.3391	Right
13	-19.7823	-18.5192	-21.6426	-17.7701	Down
13	-19.7823	-18.5192	-21.6426	-17.7701	Down
40	-18.1304	-17.0969	-17.9954	-17.6703	Right
40	-18.1304	-17.0969	-17.9954	-17.6703	Right
40	-18.1304	-17.0969	-17.9954	-17.6703	Right
58	-15.2636	-13.6749	-15.2058	-14.473	Right
58	-15.2636	-13.6749	-15.2058	-14.473	Right
58	-15.2636	-13.6749	-15.2058	-14.473	Right
58	-15.2636	-13.6749	-15.2058	-14.473	Right
58	-15.2636	-13.6749	-15.2058	-14.473	Right
33	-11.7452	-10.4358	-11.5325	-9.19171	Down
33	-11.7452	-10.4358	-11.5325	-9.19171	Down
33	-11.7452	-10.4358	-11.5325	-9.19171	Down
33	-11.7452	-10.4358	-11.5325	-9.19171	Down
33	-11.7452	-10.4358	-11.5325	-9.19171	Down

Tabelle 5.10: The current state of the agent, the Q-Values (calculated by the Value-Iteration algorithm) for each action and the selected action in the state.

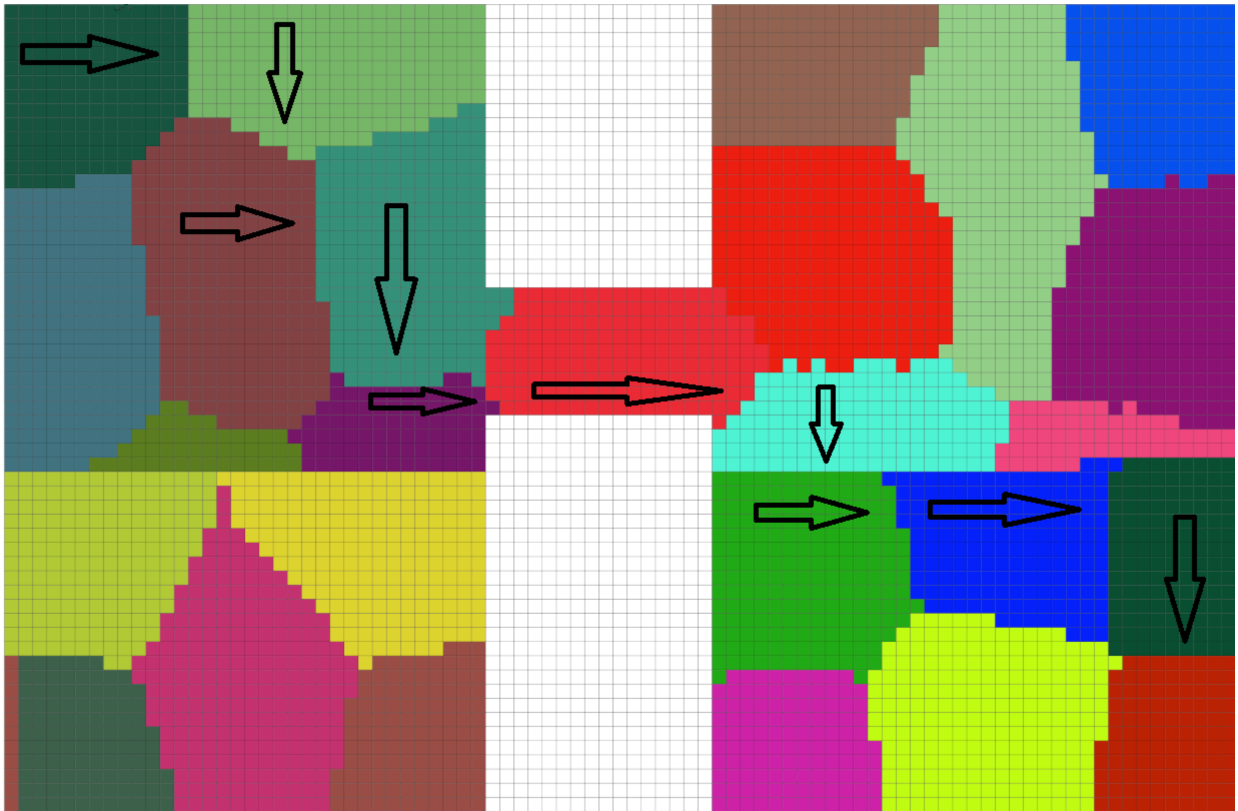

Abbildung 5.29: The discretized state space of the 3D two room problem and the Q-Value direction vectors. Each colored shape represents one state.

state, and if so, how many iterations are necessary to find it. We first outline how the model algorithm has to be changed to account for this new requirement:

```
void SetAllStatesToUnknown(int additionalVisits)
{
        % Loop through all states
        foreach (state in states)
        {
                state.known = false;
        }

        % Increase the visits count
        numberOfVisits += additionalVisits;
}
```

As it can be seen in the above code, a new method was introduced, namely 'SetAllStatesToUnknown'. The method has one parameter the 'additionalVisits', which is selected by the caller and determines how many additional visits are necessary to treat a state as known. The method first, rests all states to 'unknown', otherwise the algorithm would consider all the states as known and no Q-Value updates would be performed, which in turn would mean that the new terminal state would never be reached. Next, the 'numberOfVisits' counter is increased, this variable is used to determine when a state should be considered as known. For example if the user sets the 'numberOfVisits' variable to 50, the algorithm treats a state as known after 50 visits. By increasing this number the algorithm is able to extend the exploration phase, which means the possibility to find the new terminal state is pretty good. Another possibility would be to set the 'visits' counter maintained for each and every state back to zero, but this some how fakes an incorrect behaviour, that's why we decided to increase the 'numberOfVisits' count instead.
Now it is time for some results of the Value-Iteration algorithm, in the following a table is shown see 5.11,

NumberOfEpisodes	StepsToTerminal
5000	31
2500	31
1000	31
500	31
250	Not reached

Tabelle 5.11: The settings for the most important parameter for this task and the performance of the algorithm.
NumberOfEpisodes: Can be considered as train phase, how often should the algorithm be executed on the state space.
StepToTerminal: Performance measurement, how many steps does the algorithm need to reach the new terminal state.

that contains the 'NumberOfIterations' and the 'StepsToTerminal' columns. These are the only adapted parameters, all the other parameters stay the same as for the first terminal state. It can be seen that the algorithm is indeed able to find the new terminal state. The same is true as for the first terminal state, when the algorithm is able to find the terminal state then the result is nearly optimal, no matter how many iterations were performed.

In the next experiment the Q-Values for the different directions are calculated and plotted. These plots show the Q-Values after the terminal was changed to a different state.

- Q-Values of direction left, please refer to figure 5.30.
- Q-Values of direction up, please refer to figure 5.31.
- Q-Values of direction right, please refer to figure 5.32.
- Q-Values of direction down, please refer to figure 5.33.

In plot 5.30, we can see that it does not make sense to go to the left, marked with dark blue. In contrast it makes perfect sense to go to the right see plot 5.32, which is illustrated by the slightly paler shade of blue.

The final experiment in the Value-Iteration section is to get the maximum Q-Value of every state. This value represents the best action form the current to the next state. A table is shown that represents the agent's path from the start to the new terminal state, please see table 5.12. Finally a plot is presented which shows a direction vector in each state, this should lead to a path from the start to the new terminal state, please see figure 5.34.

5.6.5 Important note to number of features

One of the central parameters of these models is the number of features extracted in the SFA step. Even if the problem was already discussed in the practical section of the SFA, it again popped up in this chapter, especially when it comes to discretization of the state space. During the experiments we observed that after the ICA was performed on the state space, the number of states was reduced significantly. This could indicate that the number of features in the feature extraction process is too high and could be decreased. However, a quick experiment indicates that the discretization impacts the result significantly. Plot 5.35 depicts, the state space of the 3D world with small passage after discretization, where only 32 features were extracted. It can be seen that the state space is ambiguous and a lot of small states appeared. The consequence of this is, that the model learning system is not able to find a path from the start to the terminal state. We conclude that it makes sense to extract more features, even if the ICA drops a few of them.

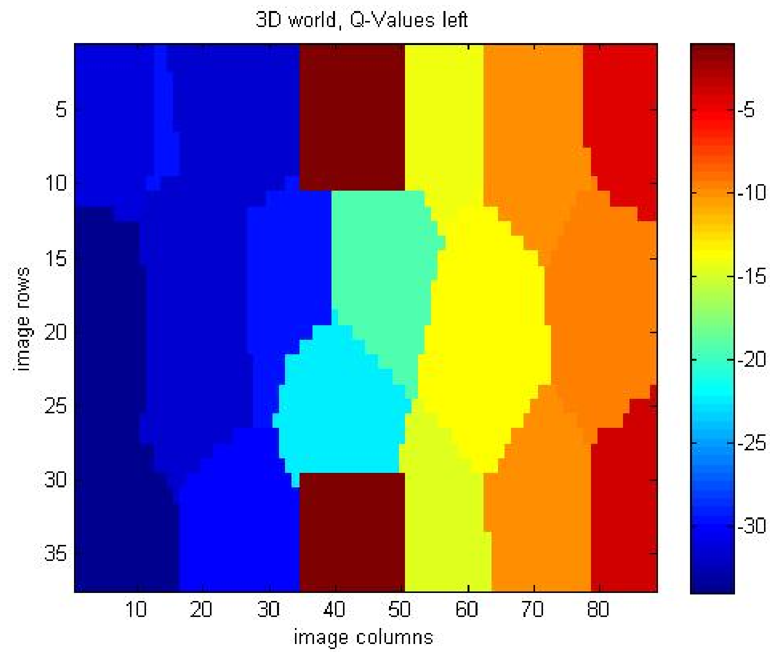

Abbildung 5.30: The Q-Values in left direction after terminal state was changed for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

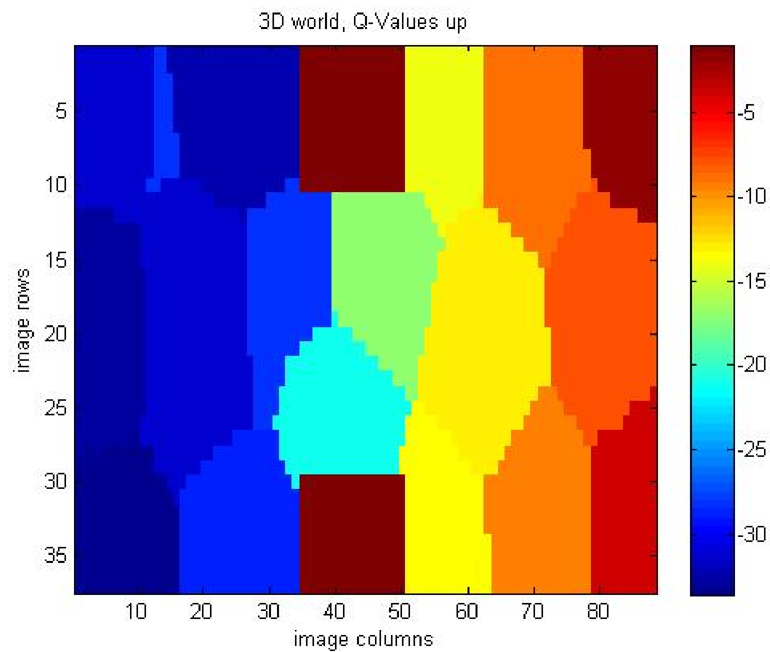

Abbildung 5.31: The Q-Values in up direction after terminal state was changed for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

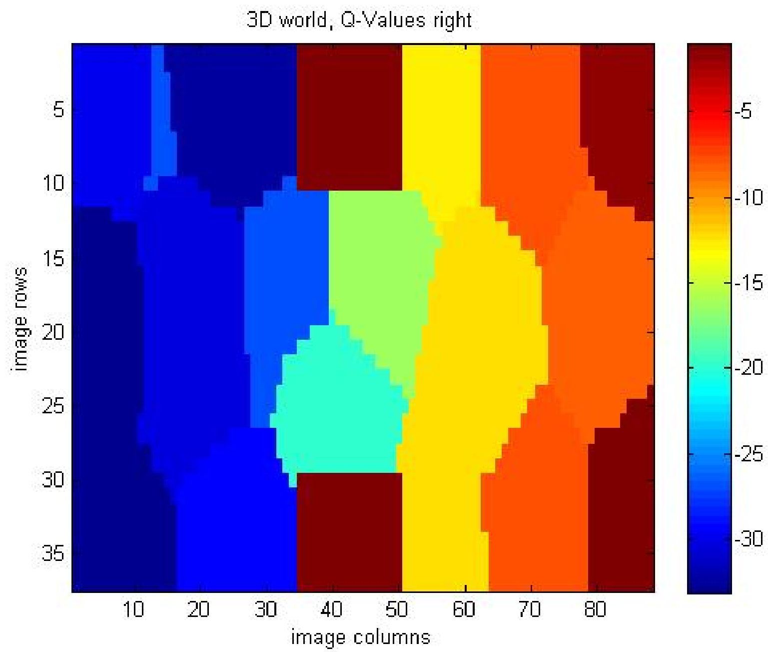

Abbildung 5.32: The Q-Values in right direction after terminal state was changed for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

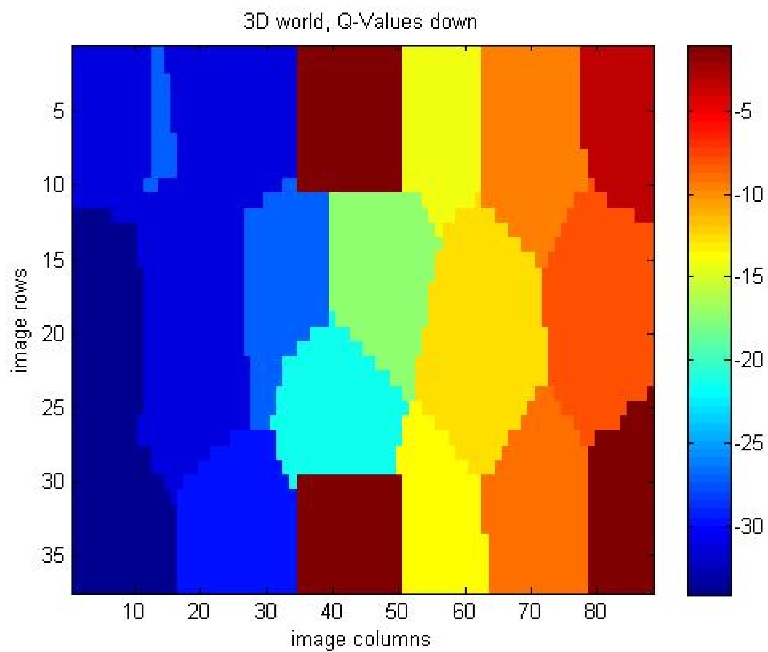

Abbildung 5.33: The Q-Values in down direction after terminal state was changed for the Value-Iteration algorithm in a 3D environment. Each colored shape represents one state.

State	Q-Left	Q-Right	Q-Up	Q-Down	Action
4	-31.0097	-30.1007	-31.0847	-31.3726	Right
4	-31.0097	-30.1007	-31.0847	-31.3726	Right
4	-31.0097	-30.1007	-31.0847	-31.3726	Right
4	-31.0097	-30.1007	-31.0847	-31.3726	Right
28	-31.6011	-32.2261	-32.4498	-31.419	Down
28	-31.6011	-32.2261	-32.4498	-31.419	Down
61	-29.5614	-26.6882	-28.458	-27.2938	Right
28	-31.6011	-32.2261	-32.4498	-31.419	Down
40	-31.7423	-30.5262	-31.2163	-31.1403	Right
28	-31.6011	-32.2261	-32.4498	-31.419	Down
40	-31.7423	-30.5262	-31.2163	-31.1403	Right
40	-31.7423	-30.5262	-31.2163	-31.1403	Right
61	-29.5614	-26.6882	-28.458	-27.2938	Right
61	-29.5614	-26.6882	-28.458	-27.2938	Right
61	-29.5614	-26.6882	-28.458	-27.2938	Right
61	-29.5614	-26.6882	-28.458	-27.2938	Right
57	-19.4599	-16.4675	-17.2113	-17.531	Right
57	-19.4599	-16.4675	-17.2113	-17.531	Right
57	-19.4599	-16.4675	-17.2113	-17.531	Right
57	-19.4599	-16.4675	-17.2113	-17.531	Right
57	-19.4599	-16.4675	-17.2113	-17.531	Right
57	-19.4599	-16.4675	-17.2113	-17.531	Right
52	-13.8549	-12.0452	-12.7355	-12.6899	Right
52	-13.8549	-12.0452	-12.7355	-12.6899	Right
52	-13.8549	-12.0452	-12.7355	-12.6899	Right
44	-9.9055	-7.76167	-8.79869	-9.35722	Right
44	-9.9055	-7.76167	-8.79869	-9.35722	Right
44	-9.9055	-7.76167	-8.79869	-9.35722	Right
35	-9.49864	-8.25061	-7.88041	-8.22505	Up
44	-9.9055	-7.76167	-8.79869	-9.35722	Right
35	-9.49864	-8.25061	-7.88041	-8.22505	Up

Tabelle 5.12: The current state of the agent, the Q-Values (calculated by the Value-Iteration algorithm) for each action and the selected action in the state.

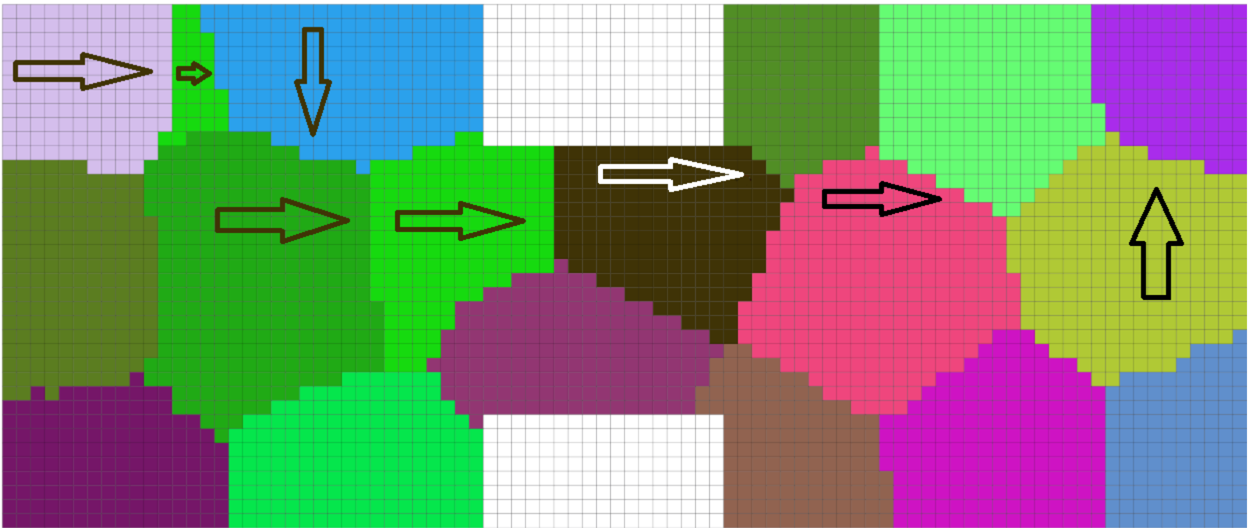

Abbildung 5.34: The discretized state space of the 3D two room problem and the Q-Value direction vectors. Each colored shape represents one state.

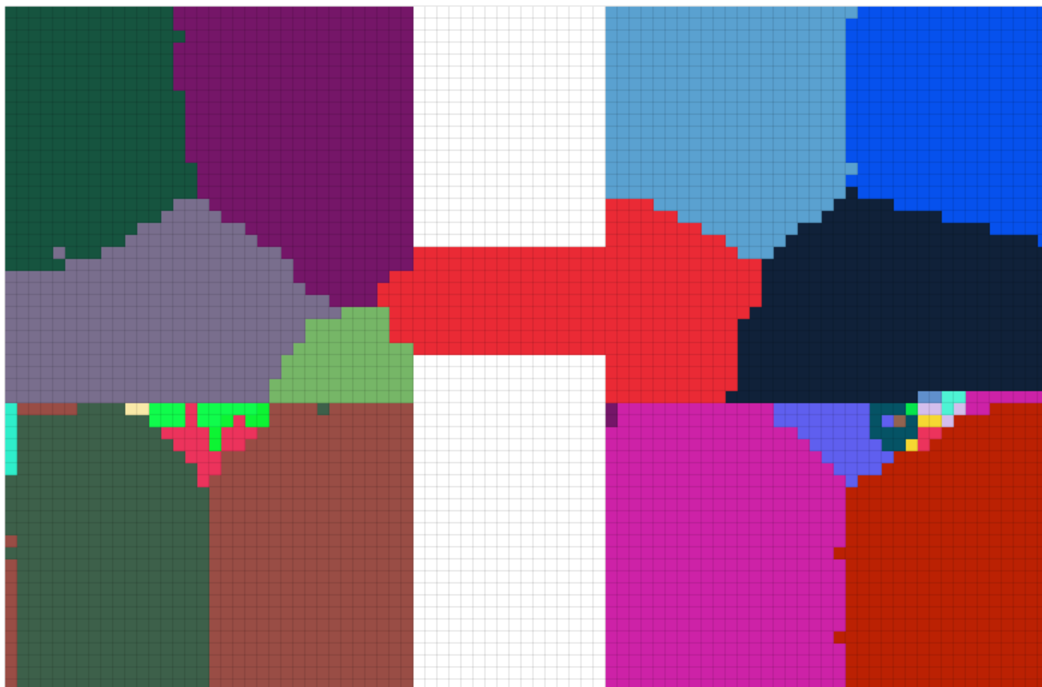

Abbildung 5.35: The discretized state space of the 3D two room problem and the Q-Value direction vectors. Each colored shape represents one state.

Kapitel 6

Outlook

6.1 General Trends

In computer science especially in the field of machine learning, the approach of using unsupervised methods, like the Slow Feature Analysis for feature extraction used in this thesis is very popular. These methods are a step in the right direction to simulate the human learning behaviour in much situations, so for example to recognize when the noise of a car sounds strange, for a human this is a unsupervised learning process. A lot of other examples exist, and for science it's the only way to simulate biological behaviour accurate. But one question stays open: To build a complete self learning and self organizing computer system, is the way to archive this, trying to simulate biological behaviour, or should science try to find other solutions for this?
The same question arises for reinforcement learning, it makes perfectly sense to use this approach to simulate the human learn process and a lot of these methods and algorithms provide great results in doing so. All these algorithms rely on statistics and probability theory, so if only one of these statistic measurements fail the outcome of these methods most likely is completely wrong, and finding the mistake is very hard. So for scenarios like this, its easier to use supervised methods, because finding a problem in such a system is easier compared to unsupervised or reinforcement learning systems.
It's hard to say what the future of artificial intelligence will bring, but it's clear that both supervised and unsupervised systems will be heavily used, and maybe it will be a combination of both systems, what finally brings the success.

6.2 Ideas for Future Work

A lot suggestions for improvements have been made in the previous chapters of this thesis, but some of these will be outlined here, because these seem to be the most important for future work.
The 'Slow Feature Analysis' system which is used to extract important features from the input data, for example. One major drawback of the method is that all image data need to be available for the training of the system. In this thesis between 100.000 and 200.000 training images were used. This is a huge amount of data and not each computer has enough memory to handle this. Even after the system was trained successfully, the computational resources to execute a SFA network are quit high. In days were it is very popular to use embedded systems like smart phones, which still have limited resources compared to modern desktop computers, it would be very hard to execute the SFA algorithm on such devices. Incremental Slow Feature (ISFA) Analysis [26] could be a solution for the above mentioned problems, but this is out of the scope of this thesis.
The 'UCT' algorithm suffers from comparable problems. The difference is that the algorithm does not have problems due to memory consumption, the problem here is the execution time. As described in the

previous chapters the UCT method is based on recursive programming. This is a problem, since recursive algorithms are slower than their iterative counterparts. A solution would be to use of dynamic programming techniques to transform UCT to a iterative algorithm, but due to the use of a decision tree, it's not clear if this potential improvement is practicable.

The above mentioned two points are the most important ones, and an improvement would increase the chance of using these algorithms not only in theoretical scenarios, but also in real world applications.

Kapitel 7

Concluding Remarks

In this section the results of the whole thesis is shortly summarized, the focus is on the results of the following chapters: 'Feature Extraction Practical Section' 4 and 'Model Based Learning Practical Section' 5.
In the SFA practical section, the main goal was to find a system which is able to handle all kinds of different inputs. This goal was reached since the feature extraction system is able to generate useful features on simple binary images as well as on real world RGB images. The implementation provides a good performance compared to the 'Modular toolkit for Data Processing (MDP)' [27] as well as a platform independent implementation since C++ was used. The user of the implementation is able to combine different algorithms like Linear-SFA, Quadratic-SFA, Additive-Noise,... in one single SFA-Node. This node then can be used in a hierarchical SFA-Network, this feature is the same as the one provided by the MDP toolkit. The results in section 4 proof that the implementation works. A drawback of the implementation is that the memory consumption is pretty high, an improvement on the implementation or the algorithm itself for a better performance would be welcome.
In the model based learning practical section, the goal was to implement a model based learning system which is able to find a path from a user defined start state to a also user defined terminal state. The state space is generated out of the features extracted by the SFA method, then two different planning algorithms (Value-Iteration and UCT) were used to find the path. As the name already says, it is a model based learning system, hence the planner algorithm (Value-Iteration or UCT) uses a model to make its decisions. For this case the so called R-Max algorithm was implemented to generate such a model. The results in the section 5 proof that the methods work. The referenced chapter also confirms the Value-Iteration is the better performing algorithm, not in the sense that it finds the better way, but the overall performance is better. This means that Value-Iteration executes faster than UCT. The reason for this is that UCT is a recursive algorithm while Value-Iteration is iterative. The other reason why Value-Iteration outperforms UCT is that Value-Iteration was able to find a path in each experiment, while UCT was not. So the conclusion is that when building a model based learning system for tasks similar to those considered in this thesis, the best choice is to use R-Max model in combination with Value-Iteration planning algorithm.

Literaturverzeichnis

[1] Christopher M. Bishop, *Pattern Recognition an Machine Learning*, Springer-Science+Business Media, 2006.

[2] Jonathon Shlens, *A Tutorial on Principal Component Analysis*, Google Research, Mountain View, CA 94043, April 7, 2014, Version 3.02.

[3] Aapo Hyvärinen and Erkki Oja, *Independent Component Analysis: Algorithms and Applications*, Neural Networks Research Centre Helsinki University of Technology, P.O. Box 5400, FIN-02015 HUT, Finland

[4] Laurenz Wiskott and Terrence Sejnowski, *Slow Feature Analysis: Unsupervised Learning of Invariances*, Computational Neurobiology Laboratory

[5] Mathias Franzius, Henning Sprekeler, Laurenz Wiskott, *Slowness and Sparseness Lead to Place, Head-Direction, and Spatial-View Cells*, Institute for Theoretical Biology, Humboldt-Universität zu Berlin, Germany

[6] Richer S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction, Second edition*, The MIT Press Cambridge, Messachusetts, London, England

[7] Todd Hester and Peter Stone, *Learning and Using Models*, Department of Computer Science, The University of Texas at Austin

[8] Becker, S. and Hinton, G. E., *A self-organizing neural network that discovers surfaces in random-dot stereograms*, Department of Computer Science, The University of Texas at Austin

[9] Kakade S (2003), *On the sample complexity of reinforcement learning*, PhD thesis, University College London

[10] Erwin Kreyszig, *Advanced Engineering Mathematics*, Ohio State Univerity Columbus, Ohio, John Wiley and Sons, INC.

[11] David Poole, Alan Mackworth, *Artificial Intelligence Foundations of Computational Agents*, `http://artint.info/html/ArtInt.html`, Accessed at: 16.06.2015.

[12] Brafman R, Tennenholtz M (2001), *R-Max - a general polynomial time algorithm for near-optimal reinforcement learning*, Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)

[13] Levente Kocsis and Csabe Szepesvari *Bandit bases Mote-Carlo Planning* Computer and Automation Research Institute of the Hungarian Academy of Scien ces, Kende u. 13-17, 1111 Budapest, Hungary

[14] Auer P, Cesa-Bianchi N, Fischer P. *Finite-time analysis of the multiarmed bandit problem* Machine Learning 47(2):235-256

[15] Hester T, Quinlan M, Stone P (2011), *A real-time model-based reinforcement learning architecture for robot control*, ArXiv e-prints 11051749

[16] Sutton R (1990), *Integrated architectures for learning, planning, and reacting based on approximating dynamic programming*, Proceedings of the Seventh International Conference on Machine Learning (ICML)

[17] Sutton R (1991), *Dyna, an integrated architecture for learning, planning, and reacting.*, SIGART Bulletin 2(4):160-163

[18] Moore A, Atkenson C (1993), *Prioritized sweeping: Reinforcement learning with less data and less real time*, Machine Learning 13: 103-130

[19] Silver D, Sutton R, Müller M (2008) *Sample-based learning and search with permanent and transient memories.*, Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML)

[20] *Qt-Developers*, `http://www.qt.io/developers`

[21] *Eigen C++ template library for linear algebra*, `http://eigen.tuxfamily.org/index.php?title=Main_Page`

[22] *Intel Threading Building Blocks*, `https://software.intel.com/en-us/intel-tbb`

[23] *Sample mean and covaraince*, `https://en.wikipedia.org/wiki/Sample_mean_and_covariance`

[24] *Deep Learning, An MIT Press bock* `http://www.deeplearningbook.org/`

[25] Robert Legenstein, Niko Wilbert, Laurenz Wiskott (2010) *Reinforcement Learning on Slow Features of High-Dimensional Input Streams*, Institute for Theoretical Computer Science, Graz University of Technology, Graz, Austria

[26] Varun Raj Kompella, Matthew Luciw, and Jürgen Schmidhuber *Incremental Slow Feature Analysis*, IIDSIA, Galleria 2 Manno-Lugano 6928, Switzerland

[27] *Modular toolkit for Data Processing*, `http://mdp-toolkit.sourceforge.net/`