Combining Reinforcement Learning and Feature Extraction

Bachelor-Thesis von David Sharma aus Offenbach am Main
November 2012

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Computer Science
Intelligent Autonomous Systems

Combining Reinforcement Learning and Feature Extraction

Vorgelegte Bachelor-Thesis von David Sharma aus Offenbach am Main

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten:

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den November 30, 2012

_____

(D. Sharma)

Abstract

In reinforcement learning, an agent interacts with its environment by taking actions and receiving rewards for the taken actions. However, if we want to apply reinforcement learning on tasks with high dimensional state-spaces, reinforcement learning becomes infeasible because the number of different states depends exponentially on the number of state variables.
In this thesis, we want to combine reinforcement learning with feature extraction to reduce the number of states. We use feature extraction to cluster the observations of the agent. The features are then defined as the similarities to these clusters. We will use the extracted features in linear function approximation to reduce the state space. We will demonstrate this approach on a discrete grid-world, which is composed of different patterns (e.g., arrows, circles). As the agent is not aware of the different symbols nor it knows the meanings of the symbols, the agent has to recognize the symbols and learn their meanings to make reasonable movements in the world.

Acknowledgements

I want to thank my supervisor, Gerhard Neumann for his patience and support. Even though, he had a lot of other stuff to do, he was always available to me and helped me a lot.

Contents

1 Introduction

The human brain is one of the most impressive and complex structures in our universe. With the connection and communication between the neurons, we are able to think, learn, memorize and remember just to mention a few things, the brain enables us to do. The by far most important characteristic not only of humans, but of all vertebrates is to learn and memorize what we have learned, to be able to reason about the things we have learned. Without learning, we would not be able to survive in our environment. The question now is, whether it is possible to make also machines learn. The fields of machine learning and artificial intelligence are dealing with this problem. There are different problems to solve for a machine. Supervised learning, unsupervised learning and reinforcement learning.

1.1 Supervised learning

In supervised learning, a teacher provides the agent with the correct answers for all training samples. Consider we have to solve the following problem: We are given a set of values X and their target values Y. Our task now is to find a function $f : X \to Y$, such that $f(x) = y, x \in X, y \in Y$. This is a supervised learning problem, because we have been provided with the target values Y.

1.2 Unsupervised learning

In unsupervised learning, we have to extract useful information from input data. There is no teacher that provides us with training samples. Here, the task is to find groups of similar examples, given only the input data. For example, consider we have a group of animals and we want to split them in different groups depending on their similarities. Now we need to check for similarities and put the animals with similar characteristics in the same group. This method is also called clustering which will play an important role in this thesis.

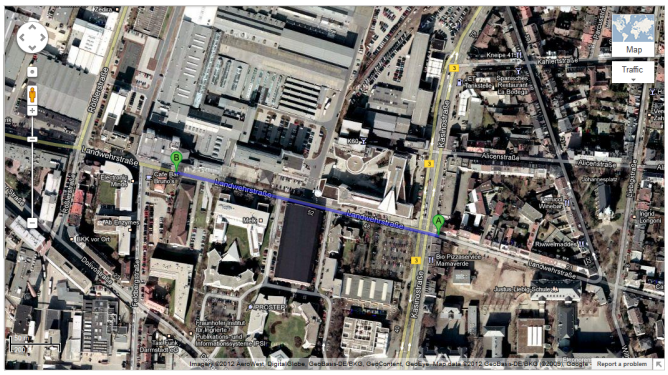
1.3 Reinforcement learning

In reinforcement learning, an agent interacts with its environment. For every action a_t the agent takes in a state s_t, it receives a reward from the environment and ends up in state s_{t+1}. Instead of having a supervisory that tells the agent what to do in every state, it gets feedback from the environment in form of rewards. The agent chooses actions according to a policy, which maps states to actions. The goal of the agent is to learn a policy that maximizes the long-term reward. In order to do that, the agent has to explore its environment and use its gained experience to improve the policy. For a more detailed introduction, we refer to Section 3.

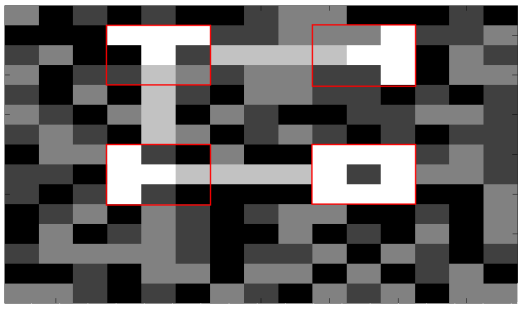
1.4 Motivation

In reinforcement learning, an agent needs to know its state. In complex environments a definition of a state is not given or very high-dimensional. The problem with high-dimensionality is that we end up in too many states. In theory, many RL algorithms require that each state is visited infinitely many times. This assumption makes RL infeasible for high-dimensional state spaces. We will use feature extraction to reduce the dimensionality for the reinforcement learning task. We want to use RL to learn on raw pixel images. Suppose we have a Google satellite image and we want the agent to walk from A to B. To get from A to B, the agent is only allowed to walk on streets. Paths or short cuts through any other terrain are now allowed. The difficulty is, that the agent does not know the concept of streets nor does it know, how a street looks like. The agent first needs to recognize that streets are an important feature and then it has to learn that it is only allowed to walk on the street to get from A to B. We accomplish this by combining two learning strategies, reinforcement learning and feature extraction. Every patch of the Google satellite image the agent sees, could be a street. Since an image consists of pixels and a pixel has different pixel values, we end up in too many states, as the number of different states grows exponentially with the number of state variables (e.g, pixels). This problem is also called curse of dimensionality.

To overcome this problem, we use unsupervised learning methods which generalize over the high-dimensional state space by performing dimensionality reduction. We reduce the number of states by grouping similar observations together in one cluster, such that, after the agent observes a new image, we just need to ask which cluster created that data point. In this thesis, we combine unsupervised learning with reinforcement learning, however, as a starting point, we use a much simpler world. Our world is composed of different patterns (e.g. arrows). Each pattern has a meaning, i.e., an arrow leads the agent to the next pattern. A pattern is a concatenation of pixels. A single pixel of the pattern can have different pixel values. Every patch the agent perceives is a possible state in the world. We compute the similarity of every cluster with each data point and then use these similarities as features in linear function approximation, to reduce the state space for reinforcement learning. With this approach, the agent is able to recognize patterns properly and makes reasonable movements in the world.



(a) Google satellite image ©Google



(b) Simplified world

Figure 1: A Google satellite image and our simplified world. The arrows and the goal symbol in our world are bordered in red. The arrows lead the agent to the goal. In the Google satellite image the agent should get form A to B by only using the streets marked in blue.

1.5 Related Work

There are several approaches in combining reinforcement learning with methods which generalize over the state space to reduce the amount of states. We will summarize a few approaches in this thesis. In [11], a kd-tree is used to partition the state space into smaller regions. The root node of the tree represents the entire state-space. The branches of the tree divide the space into two equally sized discrete sub-spaces halfway along the axis. The leaf-nodes contain the data. Each leaf node stores the Q-function for a small hyper-rectangular subset of the entire state space. The discrete areas of the continuous space that the leaf nodes cover are referred to as regions. This method is used to adaptively increase the resolution of a discretized Q-function based upon which region of the state-space is most important for the purposes of decision making. To guide the partition process, decision boundaries were used. This method is used to find improvements in policies at a higher resolution, whereas in areas of uniform policy there is no performance benefit for knowing that the policy is the same in twice as much detail. As a result, it was shown that Q-function representations for model-free reinforcement learning can be learned in simple continuous-state environments with very little initial information. The refining process allows fast initial learning and continually improving policies.
Another approach is given in [8]. Here, a reward-related low-dimensional state space is extracted by combining Input Correlation Learning (ICO) and Reinforcement Learning (RL). A property of ICO-learning is to quickly find a correlation between a state and an unwanted condition,i.e., learn anticipatory actions to avoid unwanted conditions. First, ICO is used to extract a low dimensional feature space to find a failure avoidance policy. Then, the extracted feature space is used as a prior for RL. In this work, ICO and RL complement each other in a way that the evaluative feedback from RL is needed for ICO and the reward related feature-extraction from ICO is needed for RL to achieve better task performance.
In [2], a model free learning algorithm is introduced, called LEAP (Learn Entities Adaptive Partitioning), that uses multiple overlapping partitions which are dynamically modified to learn near optimal policies with a small number of parameters. If an incoherence between the current action values and the actual rewards from the environment is detected, LEAP starts to generate new partitions. LEAP changes the structure of its function approximator on-line until the target values in macro states are sufficiently coherent. To overcome possible over-refinement, a pruning mechanism combines macro states without affecting the policy. Refinement and pruning in LEAP leads to a multi resolution state representation specialized only where it is actually needed. LEAP reduces the state space by using a set of learning

entities (LE). One LE is created for each state variable. It only maintains its state variable and ignores the others.
In [7] a hierarchical slow feature analysis (SFA) network is used to first preprocess raw high-dimensional input data and then a simple neural network is trained based on rewards. The SFA network extract the most slowly varying features in the input stream. It first extracts local features and then integrates them in more global and abstract features. The hierarchical model works such that, the first layer extracts slow features of small local image patches. The second layer extracts slow features of these features and so on. After the process of feature extraction, a neural network is trained by a reward based synaptic learning rule, that is related to policy gradient methods or a immediate reward is maximized, which results in a much simpler task.
In [6] conditional mutual information is used to measure the dependency between return and state feature sequences. The conditional mutual information has to be approximated and this is done with the least-squares method, which results in a computationally efficient feature selection procedure. To select the features, based on conditional mutual information, forward selection is used. A similar work is given in [4]. Here, reinforcement learning is also applied on raw pixel images as input state. The reinforcement learning algorithm they chose is known as fitted Q-iteration.

2 Feature Extraction - The EM-Algorithm

In this chapter, we describe how we use an unsupervised learning method, called the expectation-maximization (EM) algorithm to extract features and thereby perform a dimensionality reduction of our high-dimensional state space. Before we do that, we need to explain mixture models, which are essential to explain the EM-algorithm. After we explained mixture models, we first explain the EM-algorithm in general and afterwards, we apply the EM-algorithm to our problem.

2.1 Mixture Models

Consider we have a set of N data points and each data point is created from 1 of K components. Each component is represented by a probability distribution. If we have more than one component, we talk about a mixture distribution which corresponds to a mixture model [10]. Our data is drawn from a probabilistic model. The data is represented by patterns. A pattern consists of pixels and a pixel can have j possible pixel values. We can represent a single pixel as a J dimensional vector where the j-th value is 1 and all the other values are 0, describing that the pixel has the j-th pixel value. To model the distribution for a single pixel, we need a J-dimensional vector $\boldsymbol{\mu}$ that holds the probabilities that a single pixel can take a certain pixel value. Since a pixel can take 1 of J pixel values, we can use a multinomial distribution to represent the model. If we denote the probability of a pixel value $y_j = 1$ by the parameter $\boldsymbol{\mu_j}$, then the distribution of a single pixel $\mathbf{y}$ is given by

$$p(\mathbf{y}|\boldsymbol{\mu}) = \prod_{j=1}^{J} \mu_j^{y_j}.$$

If we want to represent a pattern, we can represent it in a matrix $\mathbf{Y}$, because a pattern consists of l pixels. The l-th column then represents the l-th pixel and the j-th row represents the j-th pixel value. The matrix $\mathbf{Y}$ is then given by

$$\mathbf{Y} = \begin{pmatrix} y_{11} & \cdots & y_{1l} \\ \vdots & \ddots & \vdots \\ y_{j1} & \cdots & y_{jl} \end{pmatrix}.$$

Since we draw our patterns from a probabilistic model, we need to model the distribution for a pattern. Again, we will model it as a matrix $\mathbf{M}$, where a column represents the distribution for a single pixel. The matrix $\mathbf{M}$ then holds the probabilities $\boldsymbol{\mu}$ that a pixel can take a certain pixel value, which is given by

$$\mathbf{M} = \begin{pmatrix} \mu_{11} & \cdots & \mu_{1l} \\ \vdots & \ddots & \vdots \\ \mu_{j1} & \cdots & \mu_{jl} \end{pmatrix}.$$

The matrix $\mathbf{M}$ is a parameter of the mixture component. Since the parameters in the matrix represent probabilities, the parameters $\boldsymbol{\mu_{jl}}$ must satisfy $\mu_{jl} \geq 0$ and the values of every column must sum to 1. If we want to represent the distribution for a whole pattern, we need to iterate over the matrices $\mathbf{Y}$ and $\mathbf{M}$ and so the distribution for a pattern $\mathbf{Y}$ is given by

$$p(\mathbf{Y}|\mathbf{M}) = \prod_{j=1}^{J} \prod_{l=1}^{L} \mu_{jl}^{y_{jl}}, \tag{1}$$

which corresponds to a mixture component. The mixture model is described by the sum over all mixture components weighted by the prior distribution $\mathbf{c}$ which is a K-dimensional vector. The prior distribution $\mathbf{c}$ is the other parameter of the mixture component. Mathematically, the mixture model can be written as

$$p(\mathbf{Y}) = \sum_{k=1}^{K} \overbrace{p(k)}^{c_k} p(\mathbf{Y}|\mathbf{M_k}).$$

2.2 EM-Algorithm

The EM-algorithm is an iterative method for finding maximum likelihood solutions for models with hidden variables [5], [9], [3]. For a mixture model, the hidden variable $\mathbf{z}$ is a K-dimensional vector, that describes the assignment for one data point to one of the mixture components. If we would know the value of $\mathbf{z}$ for every data point $\mathbf{x}$, the data set $\mathbf{\{X,Z\}}$ is complete. Here, $\mathbf{Z}$ is a matrix which holds the hidden variables for every data point. Every row in the matrix is a hidden variable for one data point. If we just know the observed data $\mathbf{X}$, the data set is incomplete. Now the EM algorithm iteratively tries to find the parameter $\boldsymbol{\theta}$ that maximizes $p(\mathbf{X}|\boldsymbol{\theta})$ by using the observed data $\mathbf{X}$. Note, that the EM-algorithm may only find a locally optimal solution, because the likelihood function $p(\mathbf{X}|\boldsymbol{\theta})$ has multiple peaks, it can not be guaranteed that it finds a global optimal solution. It is often easier to calculate the parameter vector $\boldsymbol{\theta}$ that maximizes log likelihood function. The log likelihood function is given by

$$\log p(\mathbf{X}|\boldsymbol{\theta}) = \log \left\{ \sum_{Z} p(\mathbf{X},\mathbf{Z}|\boldsymbol{\theta}) \right\}. \tag{2}$$

The sum over the latent variable $\mathbf{Z}$ appears inside the logarithm. The presence of the sum prevents the logarithm from acting directly on the joint distribution, resulting in complicated expressions for the maximum likelihood solution [1]. The EM-algorithm consists of two steps, the expectation step (E-step) and the maximization step (M-step). In the E-step, we fix the parameters $\boldsymbol{\theta} = \{\mathbf{M_{1:K}},\mathbf{c_{1:K}}\}$, to calculate the posterior distribution of the hidden variable. In the subsequent M-step, the algorithm then tries to maximize the expected complete data log-likelihood. Suppose we have a set of data points that were created by a mixture distribution. With the data points and our model parameters, we first try to find out which data point is created by which mixture component and make an assignment of a data point to a mixture component with a certain probability (responsibility of a mixture component for a data point). This step refers to the E-step. The subsequent M-step now takes the computed responsibilities and re-estimates the model parameters where each data point is weighted by the responsibility of the mixture component. This two steps are repeated until the algorithm converges to the parameters that best fits the data.

2.2.1 E-Step

Since the true value of the latent variable is unknown, we need to compute the posterior distribution $p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta})$. The posterior distribution $p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta})$ is given by

$$p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}) = \frac{p(\mathbf{Z},\mathbf{X}|\boldsymbol{\theta})}{p(\mathbf{X}|\boldsymbol{\theta})}.$$

We now discuss the E-step for our concrete problem. Since the E-step iterates over all data points and mixture components, we model our latent variables with a matrix $\mathbf{Z}$. Remember, that every data point has a latent variable describing which mixture component the data point is assigned to. Every row of $\mathbf{Z}$ holds the latent variable for a data point. The matrix $\mathbf{X_i}$ is our observed i-th data point. The matrix $\mathbf{M_k}$ is one parameter of the k-th mixture component, it holds the distribution for a whole pattern. The K-dimensional vector $\mathbf{c}$ is the prior distribution and is the other parameter of a mixture component. The parameter $\boldsymbol{\theta}$ then holds $\boldsymbol{\theta} = \{\mathbf{M}_{1:K}\mathbf{c}_{1:K}\}$. Now we are able to calculate the probability $\gamma(z_{ik})$, that the data point $\mathbf{X_i}$ was generated by the mixture component $\mathbf{M_k}$ with the following equation

$$\gamma(z_{ik}) = p(z_{ik} = 1|\mathbf{X_i},\boldsymbol{\theta}) = \frac{p(z_{ik} = 1,\mathbf{X_i}|\boldsymbol{\theta})}{p(\mathbf{X_i}|\mathbf{M}_{1:K})},$$

$$p(z_{ik} = 1,X_i|\boldsymbol{\theta}) = p(\mathbf{X_i}|\mathbf{M_k},z_{ik} = 1)\overbrace{p(z_{ik} = 1)}^{c_k},$$

$$p(X_i|M_{1:K}) = \sum_{y} p(z_{iy} = 1,X_i|\mathbf{M}_{1:K}).$$

The expected value is calculated by the posterior distribution $p(z_{ik} = 1|\mathbf{X_i}, \boldsymbol{\theta})$. The nominator is composed of the joint distribution $p(\mathbf{X_i}|\mathbf{M_k}, z_{ik} = 1)$ weighted by the prior distribution $p(z_{ik} = 1)$ and is divided by the marginal distribution $\sum_y p(z_{iy} = 1, \mathbf{X_i}|\mathbf{M_{1:K}})$. The likelihood of the k-th mixture component looks like Equation 1.

2.2.2 M-Step

After having computed the posterior distribution of the latent variables, we can now use the results from the E-step to try to find a parameter vector $\boldsymbol{\theta}$ that maximizes the expectation of the complete-data log likelihood function. The expectation is given by

$$\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{old}) = \sum_Z p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{old}) \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}). \tag{3}$$

Here, $\boldsymbol{\theta}^{old}$ always denotes the calculated $\boldsymbol{\theta}$ from the previous M-step. If we are in the first iteration of the EM-algorithm, then $\boldsymbol{\theta}^{old}$ was randomly initialized. The log-likelihood function is weighted by the posterior distribution from the previous E-step. By maximizing this function

$$\boldsymbol{\theta}^{new} = \arg\max_{\boldsymbol{\theta}} \mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{old}),$$

we obtain the new model parameters $\boldsymbol{\theta}^{new}$. First, we need to differentiate Equation 3 with respect to each model parameter. Subsequently, we set the resulting derivatives to zero and solve the equations with respect to each model parameter. Note that, in Equation 3 the logarithm now acts directly on the joint distribution, instead of acting on the marginal distribution as in Equation 2. Finally, we need to check for convergence of either the log likelihood or the parameter values, i.e., if the change of the parameters or the log likelihood function falls below a defined threshold, the convergence criterion is satisfied and the algorithm terminates. If the convergence criterion is not satisfied, we set $\boldsymbol{\theta}^{old}$ to $\boldsymbol{\theta}^{new}$ and return to the E-Step.
In the case of our probabilistic feature model, given the posterior distribution from the E-Step, we now want to re-estimate the model parameters $\boldsymbol{\theta} = \{\mathbf{M}_{1:K}, \mathbf{c_{1:K}}\}$. To do so, we need to maximize the expected complete data log-likelihood

$$\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{old}) = \sum_i \sum_k \gamma(z_{ik}) \log p(\mathbf{X_i}, z_{ik} = 1|\theta).$$

To obtain the new model parameters we apply the same steps as described above in the general case. After doing these three steps, we obtain the following new model parameters

$$\mathbf{M_k}^{new} = \frac{\sum_i \gamma(z_{ik}) X_i}{\sum \gamma(z_{ik})}, \tag{4}$$

$$\mathbf{c_k}^{new} = \frac{\sum \gamma(z_{ik})}{N}. \tag{5}$$

In Equation 4, the data points are weighted by the responsibilities. Here, we sum up the responsibilities from mixture component k for every data-point. To normalize the new $\mathbf{M_k}$, we need to divide the nominator by the sum of the responsibilities from the mixture component for every data-point. In Equation 5, we divide the sum of the responsibilities from the mixture component for every data-point by the effective number of data points. Equation 5 is the average of the responsibilities for the k-th mixture component.

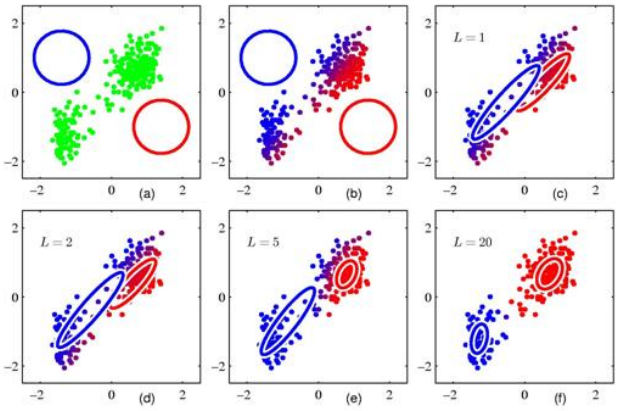


Figure 2: The EM-Algorithm with two mixtures of Gaussians (a). In (b) the E-step is applied to the data, by computing the responsibility of every mixture component for every data point. In (c) the M-step is applied by re-estimating the mixture components based on the responsibilities from the E step. In (d), (e) and (f) further iterations of the E- and M-step are applied until the algorithm converges to the components which best fit the data. The picture is taken from [1]. ©C.M. Bishop

3 Reinforcement Learning

After we have identified potential useful patterns, we now want to use the responsibilities (which measure the "similarity" to a data point) of the extracted patterns to reduce our state space with linear function approximation. The state-space is then defined as the "similarity" of the features to a data point. The goal of applying reinforcement learning on our extracted patterns is to know the meaning of a single pattern, to make reasonable movements in our world. The purpose of this chapter is first to explain the general idea of reinforcement learning and then to introduce Q-learning, which is a reinforcement learning approach. We first explain reinforcement- and Q-learning in general and then we apply Q-learning on our problem. In reinforcement learning, the agent learns by interacting with its environment. For every action the agent takes, it receives a reward from the environment. The agent is not told what action to take in a certain situation, but instead it must discover which actions yield the highest reward [14]. In order to learn how to act correctly in the environment, the agent has to explore how its actions affect the environment and then use its experience to produce the highest possible reward.
A reinforcement learning task is defined by a set of possible states **S**, a set of possible actions **A**, a state transition function $\delta\colon\ \mathrm{S}\times\mathrm{A}\times\mathrm{S}\to[0,1]$, a reward function $\mathrm{r}\colon\ \mathrm{S}\times\mathrm{A}\times\mathrm{S}\to\mathbb{R}$ and a policy $\pi\colon\ \mathrm{S}\times\mathrm{A}\to[0,1]$. We always assume our states to have the Markov property. A state has the Markov property if it is able to summarize all past information in the state at time t,i.e., the transition function and the reward function are independent of past states. A state is a Markov state, if and only if

$$P\{s_{t+1}=s',r_{t+1}=r|s_t,r_t,s_{t-1},a_{t-1},\ldots,r_1,s_0,a_0|\}=P\{s_{t+1}=s',r_{t+1}=r,s_t,a_t|\}. \tag{6}$$

A reinforcement learning task that satisfies the Markov property is called a Markov decision process, or MDP [14]. Equation 6 enables us to predict the next state and expected next reward given the current state and action. Given the state s and action a the probability for the next state s' is

$$P_{ss'}(a)=P\{s_{t+1}=s'|s_t=s,a_t=a\}.$$

Given the state s and action a together with any next state s' the expected value for the next reward is

$$r(s,a,s')=E\{r_{t+1}|s_t=s,a_t=a,s_{t+1}=s'\}.$$

The policy tells the agent what action a_t to take in the current state s_t. The main goal of the agent is to maximize its long-term reward specified by the following equation

$$R_t=\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}.$$

The parameter γ, $0\leq\gamma\leq 1$, is called the discount factor. It determines the importance of future rewards. If γ is zero, the agent only considers current rewards. If the factor approaches one, the influence of future rewards is increased.

3.1 Value Functions

Each policy has a value function. It is used to estimate the expected long-term reward for a given policy. The value function of a policy is defined of the expected long-term reward of the agent when starting in state s and following that policy thereafter. This function can be written as follows

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\}.$$

This value function is also called the state-value function. It tells the agent how much reward it can expect in the future, starting from state s and then following policy π, i.e., even though the current state has a low reward, it can have a high value, because the following states yield high rewards. The state-value function can also be written in a recursive manner. Given a stochastic policy π and a stochastic transition probability $P(s_{t+1} = s'|s_t = s,a_t = a)$ the equation can be written as

$$V^{\pi}(s) = \sum_{a}\pi(a|s)\sum_{s'}P(s'|s,a)(r(s,a,s') + \gamma V^{\pi}(s')).$$

This equation is called the Bellman equation for V^{π}. It expresses the relationship between the values of a state and the values of its successor states [14]. We now define another value-function. It is used to estimate the expected value of taking action a in state s and following policy π thereafter. This function can be written as follows

$$Q^{\pi}(s,a) = E_{\pi}\{R_t|s_t = s,a_t = a\} = E_{\pi}\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s,a_t = a\}.$$

This value-function is called the state-action-value function. This equation can be written in a recursive manner too and is given by

$$Q^{\pi}(s,a) = \sum_{s'}P(s'|s,a)(r(s,a,s') + \gamma V^{\pi}(s')).$$

The difference between the Q-function and the V-function is that we first take an action a in state s and then follow policy π, whereas with the V-function, we already would follow policy π from the beginning. If we use the value-function, we are bound to policy π, because we do not know the state-transition function. In order to learn a policy, we also need to evaluate actions, in order to do that, we need the state-action value function, the Q-function.

3.2 Optimal Value Functions

We have defined both value functions V^{π} and Q^{π} for a policy π in the previous section, but does this policy also gives us the highest possible long-term reward? Can we find a better policy π' that gives us higher long-term rewards? A policy π' is said to be better than policy π if and only if

$$V^{\pi'}(s) \geq V^{\pi}(s),$$

for all $s \in S$. If a policy is better than all the other possible policies, then this policy is said to be the optimal policy, denoted as π^*. Formally, the optimal policy can be written as

$$\pi^* = \max_{\pi} V^{\pi}(s),$$

for all $s \in S$. If the agent learned the best possible value for every state, it has learned an optimal value function denoted as V^*. The optimal value function can be written as

$$V^*(s) = \max_{\pi} V^{\pi}(s),$$

for all $s \in S$. Optimal policies also have optimal action-value functions, denoted as Q^*. The optimal action-value function can be written as

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a),$$

for all $s \in S$ and $a \in A$. We can now define the optimal state-value function. If the agent learned the optimal state-value function, then it always executes the best action in every state. The optimal value function is given by

$$V^*(s) = \max_a Q^*(s,a).$$

We now formulate the Bellman optimality equations for V^* and Q^*. It expresses that the expected return for the optimal value function under an optimal policy must equal the expected value for the best action from that state [14]. The Bellman optimality equation can be written as

$$V^*(s) = \max_{a \in A_s} \sum_{s'} P(s'|a,s)(r(s,a,s') + \gamma V^*(s')).$$

Similarly the Bellman optimality equation for the optimal Q-function looks like this

$$Q^*(s,a) = \sum_{s'} P(s'|a,s)(r(s,a,s') + \gamma \max_{a'} Q^*(s',a')).$$

3.3 Q-Learning

Q-Learning is a temporal difference learning [12] approach. Temporal difference learning combines two reinforcement learning strategies, called Monte Carlo methods and dynamic programming (DP). In the Monte-Carlo methods, the environment is unknown. Thus, the agent has to explore the environment by sampling state-transitions, actions and rewards according to a policy. In contrast to Monte-Carlo methods, DP needs a complete model of the environment. It needs to know the complete probability distributions of all possible transitions. To evaluate the V-function for every state, DP evaluates new estimates for a state, by using old estimates of every possible successor state. TD methods do not need a model of the environment (Monte Carlo) and it only uses old estimates of V to generate new estimates of V without need to wait for the final outcome (DP). This function can be written as follows

$$V(s_t) = V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)).$$

The parameter α is called the learning rate. If it is set to 0, the agent does not obtain any new information. If it is set to 1, the agent only considers the most recent information. The term in the brackets is called the temporal difference error. It is the difference between the estimate of the value of a state before (old value) and after (learned value) performing the action. In temporal difference learning there are two different policy control methods, off- and on policy control. One of them is Q-learning, an off-policy control method. The other method is called SARSA and is an on-policy control method. On-policy methods like SARSA evaluate and improve the same policy as they use, to make their decisions. Off-policy control methods like Q-learning use two different policies, one policy for behaviour and one policy for estimation, i.e., while the agent chooses actions according to policy π', it evaluates and improves the values for policy π. In SARSA, we estimate the value of the current policy (with exploration), however, as the value function defines the policy, the policy will change and also converge to the optimal policy. Q-learning estimates the value of the optimal policy (thus, no exploration). Both will almost find the same policy (except for special cases). SARSA is defined as follows

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)).$$

Similarly, Q-learning is defined as follows

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha(r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)). \tag{7}$$

The difference between SARSA and Q-learning is that SARSA updates its values using the action at time step $t+1$ according the policy it also evaluates and improves at the same time. Q-learning updates its values using the currently best action at time step $t+1$. In Q-learning algorithm [15], the following steps are repeated: The agent takes an action a and receives an immediate reward from the environment. Then the Q-value for state s after taking action a is updated with Equation 7.

We now want to apply reinforcement learning on our problem. To do that, we first need to reduce our large number of states. In theory, every state needs to be visited infinitely often to find an optimal policy. In the case of the high dimensional state space reinforcement learning becomes infeasible, therefore, to reduce the number of states, we need function approximation.

3.4 Linear Function Approximation

The goal of function approximation [13] is to approximate a value function by generalizing over a large state space. In this thesis, we want to approximate the state-action value function $Q(s,a)$ and this is done by using linear function approximation. Here, the states will be represented by a vector of features $\phi(s)$ and the state-action value function is a linear combination of features

$$Q(s,a) = \sum_{k=1}^{K} \xi_k^{(a)} \boldsymbol{\phi}_k(s) = \xi^{(a)} \boldsymbol{\phi}^T(s).$$

The parameter vector $\xi^{(a)}$ defines the weighting of each feature. What we need to achieve now is that the error between the estimated true action-value function and the approximated action-value function is as small as possible by modifying the parameter $\xi^{(a)}$. We do this with the gradient descent method. The gradient-descent method adjusts the parameter vector after each example and moves it in the direction of the decreasing error. We use a gradient based update rule which is given by

$$\xi^{(a)} = \xi^{(a)} + \alpha(r(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)) \frac{dQ(s,a)}{d\xi^{(a)}},$$

$$\xi^{(a)} = \xi^{(a)} + \alpha(r(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)) \boldsymbol{\phi}(s). \tag{8}$$

Now we are able to apply the Q-learning algorithm to our problem. The features will be determined by the EM-algorithm and we will describe that in the next section.

3.5 Epsilon-Greedy Policy (ϵ-greedy)

We still need to determine how to choose our actions. If we use a policy that always chooses the best action, also called the greedy policy, we are likely to miss better actions because of lack of exploring. Instead of acting greedy all the time, we act greedy most of the time, i.e., with a small probability of ϵ we choose an action at random. This policy is called the ϵ-greedy policy. In this policy the best action is selected with a probability of $1-\epsilon$

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{|A|} & a \neq a^* \\ \frac{\epsilon}{|A|} + (1-\epsilon) & a = a^* \end{cases}.$$

4 Combining the EM-Algorithm and Q-Learning

After having discussed the two algorithms we need, we now want to combine them to see, how reinforcement learning can interact with feature extraction. The algorithm begins with Q-Learning. The agent begins to walk through the world without any knowledge of how the world and its patterns look neither it knows the meaning of the patterns. The main purpose of the first iteration of the Q-Learning algorithm is to collect data for the EM-Algorithm, nevertheless, it also updates the state-action values. The EM Algorithm then uses the data to find useful patterns. The patterns, or clusters, that were calculated by the EM-Algorithm are now used by the Q-learning algorithm to calculate the responsibilities of the mixture components when the agent perceives a certain pattern and updates the state-action values. The responsibilities are used as features for linear function approximation. After every Q-Learning iteration, the EM-Algorithm tries to find more precise mixture components.

4.1 The World

The world is a $N \times N$ grid world. The combination of pixels can form patterns, (e.g. arrow, circles). The patterns are $M \times M$ patches of the world, with $M < N$. Each pattern has a meaning, e.g., an arrow to the left has the meaning that the agent should go to the left until it sees the next symbol. Since the agent is not aware of the different symbols nor the meaning of these symbols, we combine our two previously described algorithms, to recognize the patterns as symbols and then learn their meaning at the same time. A pixel is a random natural number from 1 to j. The number describes the value a pixel can take, e.g., a color. As symbols, we will use a goal symbol and four arrow symbols, left, right, up, down. The arrows lead the agent to the goal. The world is created as follows: First, the world is created randomly without containing any specific patterns. Then the goal symbol is created, then k steps from the goal symbol an arrow is added, which points to the goal symbol. The next symbol again is added k steps away from the last arrow and this arrow also points to the last created arrow. Optionally we are going to put some noise on the world. The agent is able

to walk in four directions, which are left, right, up and down and it is able to perceive a M × M patch of the world. For every step the agent takes, it receives a reward of -1. Only if the agent reaches the goal, it will receive a higher reward. We also implemented connections between the symbols, which can be seen as a "street". The connection starts at every arrow head of an arrow and ends at the next symbol. The intuition behind this is that, the agent can use the "street" as an orientation to the next symbol.

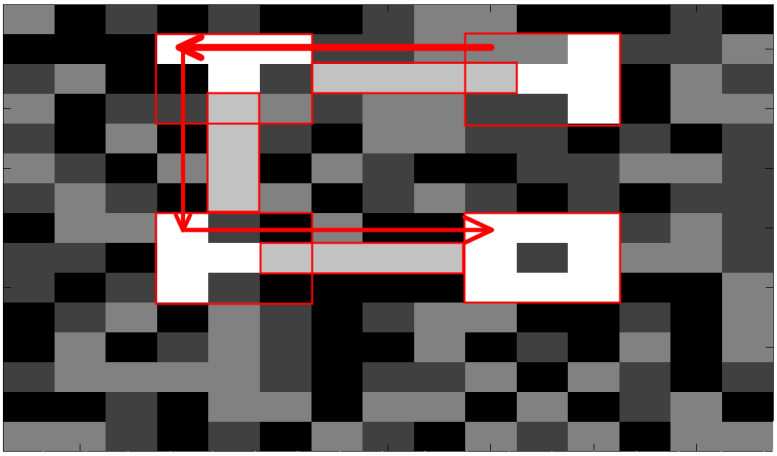
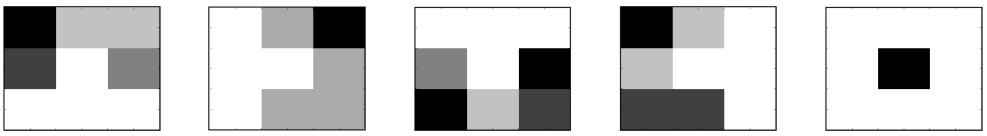


Figure 3: A 15×15 world. The arrows and the goal symbols are bordered. Also the streets, which connect the arrowheads with the next symbol are shown. The red arrows mark the path, the agent takes to get to the goal. The agent always starts at one of the arrow symbols. Keep in mind, that after an episode is finished, a new world is created which is not the same as previous world. The first symbol (goal) is placed arbitrarily, when the world is created. The next symbol is added k steps away from the first symbol, as described above.



(a) Up arrow (b) Right arrow (c) Down arrow (d) Left arrow (e) Goal symbol

Figure 4: Every arrow leads the agent to the next symbol. The agent follows the direction of the arrow, until it reaches the next symbol. If the agent reaches the goal, the episode ends and a new world is created.

(a) A patch of the world (b) A patch of the world (c) A patch of the world

Figure 5: Since the world consists of noise, the agent could see patches, which only consists of noise or patches which only show parts of the symbols, as shown in (a) which could be a left arrow.

4.2 Feature Vector and Memory

The by far most important element of our algorithm are the responsibilities, because they measure the "similarity" of a mixture component to the current input. They are the basis for identifying the patterns, when moving through the world. Since the symbols are placed with a certain distance to each other, the agent needs to memorize which symbol

it saw recently. Therefore, our feature vector $\phi(s)$ will also hold responsibilities of past time steps. We will keep the responsibilities of the last seen symbol in the memory, until the agent reaches the next symbol. Let ω denote the size of our memory and K denote the number of mixture components.

We evaluated two different representations for $\phi(s)$:

1. The feature Vector $\phi(s)$ is modelled such that it holds the responsibilities for the current time step t and the past time steps $t-\omega$ depending on how big we define our memory. A problem with this approach is, that the agent remembers the past time steps, but it does not really know where it came from, i.e., it does not know which actions it took to get to the next symbol and therefore it is not able to associate taken actions with last seen symbol. That is why we need to consider taken actions in the memory.

2. The feature Vector $\phi(s)$ is modelled such that it holds the responsibilities for the current time step t and the past time steps $t-\omega$, but this time our memory is an "action memory", i.e., depending on what action the agent chose at time step $t-1$, the responsibilities will be inserted in the memory for a certain action a. Let $\delta(s_t)$ be the responsibility vector for a current data point. If our action-memory should only remember the time step $t-1$ our $\phi(s)$ will look like this

$$\phi(s) = [\delta(s_t), \underbrace{\delta_1(s_{t-1}), \delta_2(s_{t-1}), \delta_3(s_{t-1}), \delta_4(s_{t-1})}_{\text{memory}}].$$

Here, $\delta_a(s_{t-1})$ is a K-dimensional vector holding the responsibilities for the last data point at time step $t-1$. Note that, at each time step only one memory for an action is active, depending on what action the agent chose at time step $t-1$, i.e., if action 1 was chosen at time step $t-1$, the responsibilities will be inserted in $\delta_1(s_{t-1})$. With every time step, the agent needs to remember, the size of $\phi(s)$ grows exponentially, because we need to consider every combination of actions in the past time steps

$$\text{size}(\phi(s)) = 4^{\omega-1}((\omega-1)) + K,\ K \geq 1,\ \omega \geq 1.$$

4.3 The Algorithm

We start with the Q-learning algorithm. The most important parameters for the Q-learning algorithm are the mixture components, or clusters of patterns, to compare them with the data, the agent perceives. The Q-function updates the policy and collects data which will be subsequently used by the EM-algorithm. Everything the agent perceives is being stored. The EM-algorithm now takes the data and performs the calculations of the E-step and the M-step, to find more precise mixture components and prior distributions. Its goal is to find parameters, which best fit the data. The Q learning algorithm uses the new calculated mixture components for calculating the feature vector and to update the policy. The user decides how often we iterate between the EM-algorithm and the Q learning algorithm.
When we start the Q-learning algorithm, the agent always starts on one of the arrow symbols. An episode consists of maximum i steps. An episode ends, if the agent reaches the goal in less than j steps, or if it exceeds the maximum of i steps. After every episode a new world is created and the agent starts on one of the arrow symbols again. A Q-learning iteration is one update of the policy. The user also initializes the number of Q-learning iterations and the steps the agent is allowed to take in one episode. Finally, the user needs to determine the number of mixture components, he wants to use for the algorithm. The distributions of the mixture components are initialized randomly and the prior distribution is initialized uniformly.

Algorithm 1 Feature Based QL Combined With EM: The Pseudocode describing how our algorithm works. First the mixture components are initialized. Then we enter the loops. A action is chosen according to the epsilon-greedy policy and then the agent makes a step. The agent now receives an observation and computes the responsibilities of the mixture components for the current observation. After that, our feature vector ϕ is updated and subsequently, we update our policy with our update rule. After Q learning is finished, the EM-algorithm is run.

```
// mixtureComponents : The clusters we use, to compare them with the data points
// o_t : observation at time step t
// γ : The responsibility data of every mixture component for a data point.
// φ : The feature vector.
// ξ^(a) : The parameter vector. There is a parameter vector for every action.
// j : number of iterations for the entire algorithm
// k : number of iterations for the Q-Learning part.

mixtureComponents = initMixtureComponents();
for j = 1 → J do
    for k = 1 → K do
        // Epsilon-greedy policy
        a_t ~ π(a_t|s_t)
        s_{t+1} ~ p(s_{t+1}|s_t,a_t)
        o_t = getObservation();
        γ = computeResponsibilities(o_t);
        φ = updateFeatures(γ);
        ξ^(a_t) = ξ^(a_t) + α(r(s_t,a_t,s_{t+1}) + γ max_{a_{t+1}} Q(s_{t+1},a_{t+1}) − Q(s_t,a_t))φ(s_t)
    end for

    Run EM-Algorithm
end for
```

5 Experiments

In this section, we present the results of our experiments. We will choose a 10×10 or a 15×15 world as an environment for the agent, with three arrow symbols and a goal symbol. The distance between two symbols and the allowed step size of the agent determines the size of our memory by the following equation

$$\text{sizeMemory} = \frac{\text{distanceSymbols}}{\text{stepSize}} - 1.$$

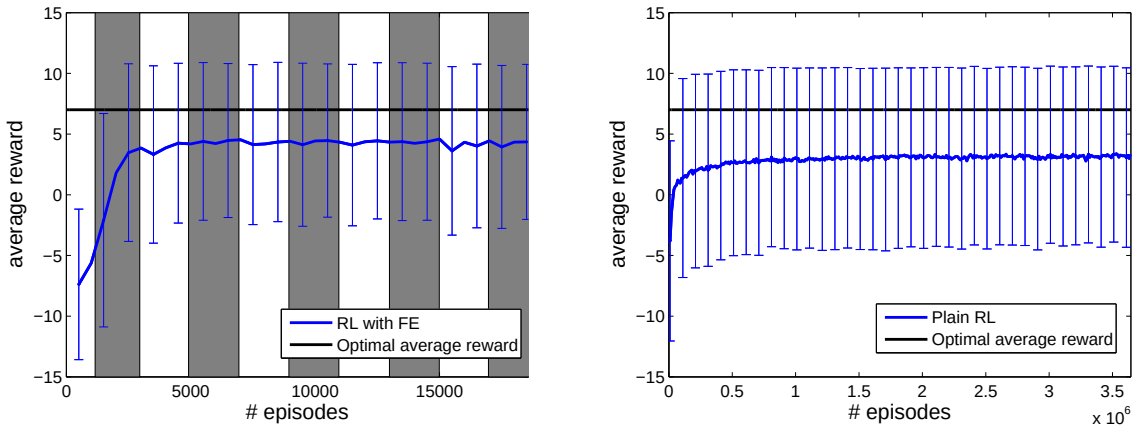
Every step size will count as one step taken in the world. An observation is always given by a 3×3 neighborhood of the agent. The distance between two symbols is determined by the steps the agent must take to the next symbol.
We will start the algorithm with Q-learning. The open parameters of the algorithm are the number of Q- learning iterations, the number of steps, the agent is allowed to walk in an episode and the number of mixture components we want to use. In the Q-learning algorithm, we choose our actions according to the ϵ-greedy policy. Every observation, the agent makes in the world is stored in a data container. We store a maximum of the 10000 most recent data points. If the agent exceeds a certain number of allowed steps in an episode or if it reaches the goal symbol, a new world is created randomly. Hence, we want to generalize between the worlds. After Q-learning is finished, the EM algorithm uses the data points to update the mixture components. In our first experiment, we want to compare plain reinforcement learning and our previously described algorithm (reinforcement learning with feature extraction). After we confirmed that our algorithm converges faster to an optimal policy than plain reinforcement learning, we want to evaluate our algorithm in different experiments. First, we evaluate how different numbers of mixture components can affect the learning speed and the quality of the policy. Then we need to look at how the agent performs, if it interacts with less and more complex worlds, using different sizes of memory. In our last experiment, we evaluate the different feature representations, we introduced in Section 4.2. In all experiments, we will use a noise of 10%, a learning rate of 0.1 and a discount factor of 0.95.

5.1 Comparison of plain reinforcement learning and reinforcement learning with feature extraction

We first compare plain reinforcement learning with our algorithm to show that plain reinforcement learning converges much slower to an optimal policy than our algorithm, using feature extraction to reduce the state space. In this experiment, we only use two pixel values to simplify the problem as much as possible for standard reinforcement learning. In plain reinforcement learning, a data point always includes the observations at time step t and time step $t-1$. Thus, the algorithm needs to maintain 2^{18} state-action value, 2^9 for time step t and 2^9 for time step $t-1$. For the initial state, we assume the data point at time step $t-1$ be the same as at time step t. The following table shows the used settings for both approaches.

	Plain RL	RL with FE
QL-iterations	10×10000	20 Million
Size of world	10×10	10×10
Distance between two symbols	6	6
Step size	3	3
Allowed steps in an episode	10	10
Size of memory	-	1
Number of mixture components	-	20

Table 1: In this experiment, we compare plain reinforcement learning approach with reinforcement learning combined with feature extraction. The table shows the used parameters for the experiment.



(a) RL with FE (b) Plain RL

Figure 6: The graph shows the results of the experiments, where we tested reinforcement learning with feature extraction and plain reinforcement learning. The results show, that reinforcement learning with feature extraction converges much faster than plain reinforcement learning. Every beginning and every end of the gray-shaded areas stands for the beginning of the EM-algorithm.

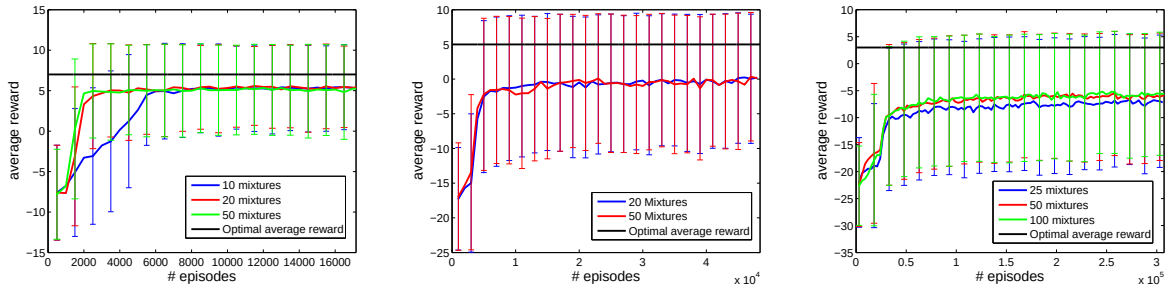
The results show that reinforcement learning with feature extraction learns much faster than plain reinforcement learning. This result is not surprising, because plain reinforcement learning needs to maintain much more state-action values (2^{18}) than reinforcement learning with EM which only has to maintain the state-action values for 20 mixture components. Reinforcement learning with EM converges after approximately 5000 episodes, whereas plain reinforcement learning needs approximately 1 million episodes to converge and even then, plain RL can not perform as good as RL with feature extraction.

5.2 Evaluation of number of mixture components

To see how the number of mixture components affects the learning performance of the agent, we test our approach with a different number of mixture components on a less complex and a more complex world. In the complex world, the agent needs to take 4 steps until it reaches the next symbol. In the less complex world, the agent only needs to take 2 steps until it reaches the next symbol. We evaluate whether the number of mixture components affects the learning speed or the quality of the learned policy.

	Little mixture components	Many mixture components
QL-iterations	10×10000	10×10000
Size of world	10×10, 15×15	10×10, 15×15
Distance between two symbols	4,6	4,6
Step size	3,2,1	3,2,1
Allowed steps in an episode	10,20,25	10,20,25
Size of memory	1,2,3	1,2,3
Number of mixture components	10,20,25	50,100

Table 2: In this experiment we evaluate, if the agent performs better with more mixture components. The table shows the parameters, we use in the experiments.



(a) Step size 3 (b) Step size 2 (c) Step size 1

Figure 7: The graphs show the results of the evaluation of the effect of a different number of mixture components. We tested on less and more complex worlds.

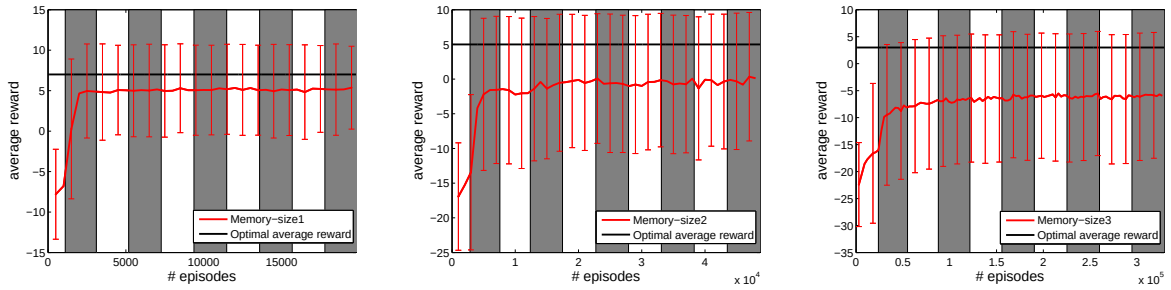
The results show that, with too few mixture components, as in Figure 7(a) with 10 mixture components, the agent needs about 5000 to 6000 episodes to converge to the highest average reward. With more mixture components, the agent was 3 times as fast to converge to the optimal policy. If we further increase the mixture components from 20 to 50, then the agent only learns slightly faster. That concludes that, for efficient learning, a certain number of mixture components is needed. Increasing the number of components does not yield any benefit. In more complex worlds a higher number of mixture components is needed to improve the quality of the learned policy, as we see it in Figure 7(c). Here, the agent learns slightly better policies with 50 mixture components than with 25 mixture components. If we exceed a certain number of mixture components, there also will be no improvement in the quality of the learned policy.

5.3 Evaluation of needed memory size

To simplify the problem and reduce the size of memory, we first tested with a step size of 3. Since the agent only needs 2 steps to the next symbol, we only need a memory size of 1. In the next experiments we decreased the step size to 2 and 1. Thus, we need a memory size of 2 and 3. We do that, because we want to see, how the agent performs if the distance to the next symbol becomes larger.

	Memory 1	Memory 2	Memory 3
QL-iterations	10×10000	10×50000	10×500000
Size of world	10×10	15×15	15×15
Distance between two symbols	6	6	4
Step size	3	2	1
Allowed steps in an episode	10	20	25
Size of memory	1	2	3
Number of mixture components	50	50	50

Table 3: In this experiment, we evaluated the performance of our algorithm if we increase the number of steps needed to reach the next symbol. The steps, the agent needs to take to the next symbol follows from the division of the distance between two symbols by the step size. The table shows the parameters that we used in this experiment.



(a) Memory-Size 1 (b) Memory-Size 2 (c) Memory-Size 3

Figure 8: The graphs show the results of the experiments, where we evaluated different memory sizes. The results show that the more complex a world becomes, it gets more difficult for the agent to learn the meaning of the symbols properly. The distance to the optimal average value becomes larger, the more complex the world gets. Every beginning and every end of the gray-shaded areas stands for the beginning of the EM-algorithm.

The results show that, the bigger the distance to the next symbol, the longer the agent needs to learn. Even then, it is very difficult for the agent to learn the meaning of every symbol properly. In Figure 8(a), the quality of the solution gets near the optimal average value. The agent also moves safely through the world, without making any mistakes. In Figure 8(b) and (c), we can also see that the distance of the average reward of the learned policy to the optimal value gets larger with an increasing distance of the symbols. If the agent moves in a more complex world, it is more likely to taking a wrong step due to exploration. Hence, the agent might get lost and does not find its way back to the actual path. Since the agent gets a reward of -1 for every step it takes, there is no indication for the agent that it is on the right way to the next symbol. Thus, it may get lost on its way to the next symbol, because it explores other actions. For an environment, where the agent only needs to take 2 steps to the next symbol, it learns the meaning of the symbols properly and moves safely through the world thereafter.

	With action memory	Without action memory
QL-iterations	10×50000	10×50000
Size of world	15×15	15×15
Distance between two symbols	6	6
Step size	2	2
Allowed steps in an episode	20	20
Size of memory	2	2
Number of mixture components	20	20

Table 4: In this experiment, we evaluated how the performance of the algorithm with different feature representations (action memory and no action memory). The table shows the parameters that we used in the experiments.

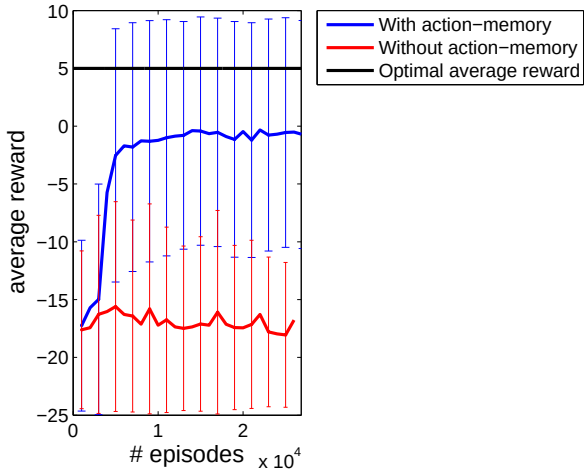


Figure 9: The plot shows the comparison of different feature representations. The results show, that without action memory, the agent is not able to learn.

In our initial experiments, we tested with standard memory without incorporating actions into the memory. However, if the agent gets lost due to exploration, it needs to be able to find its way back. This is only possible by incorporating the previously taken actions into our feature representation. A comparison of the standard memory for the worlds with 3 steps between the symbols can be seen in Figure 9. The results show that the agent performs much better with the implemented action memory in more complex environments. Without action memory, the agent is not able to learn, because it forgot where it came from. Its performance rather seems to get worse, because it quickly gets lost in the environment.

6 Conclusion and Future Work

In this thesis, we used the EM-algorithm to extract features and thereby define a state representation for the reinforcement learning algorithm in a high-dimensional state-space. At the same time, we reduced the state-space by using the responsibilities of extracted clusters in linear function approximation. The results of the experiments show that this approach significantly improves the performance of learning in comparison to plain reinforcement learning, where we have to maintain much more states. Unfortunately, this approach is limited to the steps an agent must take to the next symbol. For simple environments this approach works fine, but the longer the distance to the next symbol, the longer an agent needs to learn, how to act in certain situations and even then it is difficult for the agent to learn the meaning of a symbol properly. Another problem arises with the representation of our feature vector $\phi(s)$. It grows exponentially with every additional time step we want to memorize, because we need to consider every possible combination of actions taken in the past time steps. We showed that a memory without considering the actions is of no use, because the agent needs to know which actions it took to get to the current position, to associate an action with an arrow symbol. We also showed that for efficient learning a certain number of mixture components is needed. Increasing the number of components does not yield any benefit.

As future work, we want to optimize our feature vector. The exponential growth of ϕ with the memory size, if we want to

evaluate worlds where the distance between symbols becomes larger is not feasible. This optimization could be done by not just considering the data point at time step t, but also the data, we saw in the past. In the EM-algorithm, a data point would then consist of the current data and the past seen data. As a result we would need more mixture components, to cover all possible combinations of the data. Also, we would not need a memory in our feature vector $\phi(s)$, because the past time steps are covered in the data points. We also want to include rewards in the EM-algorithm, such that the EM-algorithm always knows, which clusters are important. The EM-algorithm would then prefer clusters with higher rewards. We also want to apply our presented approach on Google satellite images as described in the introduction. Here, we are facing different problems. First there are no arrows, which are leading the way to the goal. A street always looks the same. Since there will be higher noise on the satellite image, it may also get more difficult for the agent to recognize a street. Another problem are dead ends. Since the agent does not know how a dead end looks like, neither it knows its own position, it could end up in a loop, always trying out the street which leads to a dead end. There are other several problems for example, not just walking in a horizontal or vertical direction or much more noise in the observations, which leads to a more difficult pattern recognition.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc. Seacaucus, NJ, USA, 2006.

[2] Andrea Bonarini, Ro Lazaric, and Marcello Restelli. Reinforcement learning in complex environments through multiple adaptive partitions, 2007.

[3] Yihua Chen and Maya R. Gupta. Em demystified: An expectation-maximization tutorial. UWEE Technical Report UWEETR-2010-0002, Department of Electrical Engineering University of Washington, 2010.

[4] Ernst D., Marée R., and Wehenkel L. Reinforcement learning with raw pixel images as input state. In *International Workshop on Intelligent Computing in Analysis/Synthesis, volume 4153 of Lecutre Notes in Computer Science, pages 446-454*, 2006.

[5] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38, 1977.

[6] Hirotaka Hachiya and Masashi Sugiyama. Feature selection for reinforcement learning: Evaluating implicit state-reward dependency via conditional mutual information. In *ECML/PKDD (1)*, 2010.

[7] R. Legenstein, N. Wilbert, and L. Wiskott. Reinforcement learning on slow features of high-dimensional input streams. *PLoS Computational Biology*, 2010.

[8] Poramate Manoonpong, Florentin Woergoetter, and Jun Morimoto. Extraction of reward-related feature space using correlation-based and reward-based learning methods. In *ICONIP (1)'10*, pages 414–421, 2010.

[9] Radford Neal and Geoffrey E. Hinton. A view of the em algorithm that justifies incremental, sparse, and other variants. In *Learning in Graphical Models*, pages 355–368. Kluwer Academic Publishers, 1998.

[10] Petteri Nurmi. *Mixture Models*. Helsinki Institute for Information Technology. petteri.nurmi@cs.helsinki.fi.

[11] Stuart I. Reynolds. Adaptive resolution model-free reinforcement learning: Decision boundary partitioning. In *In Proc. 17th International Conf. on Machine Learning*, pages 783–790. Morgan Kaufmann, 2000.

[12] Richard S. Sutton. Learning to predict by the methods of temporal differences. In *MACHINE LEARNING*, 1988.

[13] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, 1996.

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT-Press, Cambridge,MA, 1998.

[15] Christopher J.C.H. Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, Psychology Department, 1989.