Generalization in Reinforcement Learning

by

Wouter Josemans
Born February 4th, 1988 in Leeuwarden, The Netherlands

Under supervision of Dr. Christos Dimitrakakis & Dr. Shimon Whiteson

A thesis submitted in partial fulfillment for the Bachelor’s Degree Kunstmatige Intelligentie

August 2009
Abstract

In this paper we evaluate two Temporal Difference Reinforcement Learning methods on several different tasks to see how well these methods generalize. The tasks were modeled as Markov Decision Processes with a continuous observation space and a discrete action space. Function approximation was done using linear gradient descent with RBFs as features. The tasks were taken from the Polyathlon domain of the 2009 Reinforcement Learning Competition. It was found that the more sophisticated method generalized better, but both methods were sensitive to changes in parameters, so there is a lot of room for improvement.

Acknowledgements

I would like to express my gratitude to my supervisors, Dr. Christos Dimitrakakis and Dr. Shimon Whiteson, who have sacrificed time and effort to help me complete this thesis. Many thanks also to the coordinator of the project, dr. Bert Bredeweg, and his teaching assistant Wouter Beek. Finally, I’d like to thank my fellow students for inspiring and supporting me during the process.
# Contents

Abstract i

Acknowledgements i

1 Introduction 1

1.1 Background ........................................ 1
1.2 Problem Sketch .................................... 2
1.3 Research Question ................................. 3
1.4 Description of Paper ............................. 3

2 Methodology 4

2.1 Background ......................................... 4
2.2 Related Work ........................................ 8
2.3 Algorithms ......................................... 9

3 Experiments & Results 13

3.1 Preliminary Experiments .......................... 13
3.2 Experiment setup .................................. 14
3.3 Results of Experiments ........................... 17

4 Conclusion & Discussion 21

4.1 Conclusion ........................................... 21
4.2 Discussion ........................................... 22
4.3 Future work ........................................ 23

Bibliography 24
Chapter 1

Introduction

1.1 Background

Learning is an important aspect of Artificial Intelligence. In fact, while there is a lot of
debate on what constitutes and intelligent agent, most people agree that such an agent
should at least have the ability to learn. It is impossible to specify all knowledge that
an intelligent agent will need in nontrivial real world tasks, so the agent would need
the ability to learn to cope with this lack of knowledge. Reinforcement Learning (RL),
having its roots in behavioral psychology, is one approach to learning how to behave.
The intuition behind it is rather simple, but the effectiveness of Reinforcement Learning
has been shown in practice many times[1, 2, 3]. Reinforcement Learning agents can learn
how to behave with no prior information about the world. This is achieved through the
mathematical equivalent of reward and punishment; after every action taken by the
agent in a certain state, numerical feedback is given indicating the valence of the action
in that state (and by extension, the actions taken in previous states). In Reinforcement
Learning, the agent tries to gather as much reward as possible.

While great efforts have been made in developing algorithms that learn efficiently in a
specific type of problem, efforts that aim at developing a general agent that is applicable
to many different Reinforcement Learning problems have been scarce. Often, we can
increase the performance of our agent by making certain assumptions about the nature
of the problem. However, if we want do develop an agent that can learn a wider range of
problems, such assumption can no longer be made (performance tends to drop sharply if
the assumption is not valid). For example, one such assumption would be that the effect
of an action does not depend on the state the agent is in, such as in a grid world where
a certain action always moves the agent one cell to the right. If such an assumption is
violated, the agent will most likely not be able to find the best way to behave, so these assumptions would have to be dropped in a general agent.

1.2 Problem Sketch

The problem addressed in this paper is one that has been (among other problems) the focus of the annual Reinforcement Learning Competition\(^1\). This competition is organized by field experts and enthusiasts, and approaches several Reinforcement Learning problems in a competitive, empirically oriented way. Participants can enter in different domains, each tackling a different RL problem. The participant’s performance is measured in an objective way and compared to that of other participants. One of the competition domains is the Polyathlon domain, which addresses the same problem that is examined in this paper. In the Polyathlon domain, several different Reinforcement Learning problems are presented to the agent, and the agent needs to perform well in all of these problems.

There are various degrees to the Reinforcement Learning problem, ranging from relatively easy problems to extremely hard ones. An easier degree of the problem would be one where the agent has an accurate model both of the rewards it will receive and of the actions it can take (i.e., what the world will look like after an action has been taken). In this case, the agent can find an optimal way to behave without even interacting with the world. Of course, it can still be very hard to find the optimal way to behave if there is an infinite number of actions that can be chosen from.

Harder degrees of this problem are problems where the agent has a model of neither actions nor rewards, and has to learn how to behave by interacting with the world. Still, there are several Reinforcement Learning algorithms specifically aimed at these kinds of problems that have been proved to work well in practice\[^1\]. In this case the observability of the world presents other difficulties; if the observations are noisy (incorrect) or incomplete, the problem gets harder still.

In the Polyathlon domain of the Reinforcement Learning competition and in this paper we deal with various instantiations of the problem where we do not have a model of actions and rewards, and we assume that the observations we get by interacting with the world are not noisy. We also assume that the observations give us enough information to uniquely identify what state we are in. Observations in our case are represented by a numerical vector. What the dimensions of the observation space represent is unknown; some of them could be completely unrelated to the task at hand. Not knowing anything about the problem posits other limitations; as we do not know in advance when the task will end or under what conditions it will end, it is very hard to find good heuristics

\(^1\)See http://2009.rl-competition.org/ for more information.
on spending our resources. Resources can be anything from computation time to fuel, but in our case the most precious resource is an interaction with the world, so we have to choose wisely how we want to spend this resource. In some cases, we may want to explore all possible action right until the end of the task, in other cases we may want to settle for a suboptimal solution and gather as much reward as possible.

1.3 Research Question

In this paper we investigate two algorithms. These algorithms were tested on a variety of problems from the 2009 Reinforcement Learning Competition, in order to find out how well they generalize across tasks. Most methods have quite a few parameters that will allow them to perform a lot better on a specific problem if these parameters are tuned to this problem. In other situations, the introduction of a problem-specific parameter could boost performance greatly. For instance, a parameter that represents the decay of another parameter might work extremely well for one problem, but may hurt performance on another. We want to find out how which of the methods we tested is best, and find out how much influence the parameter values of these methods have on performance.

1.4 Description of Paper

The next chapter contains some background on the theory behind Reinforcement Learning, and describes the methodology that is used. We discuss related work in this chapter. Pseudo code for the algorithms is listed as well. Chapter 3 describes the experiments that were performed to evaluate the performance of the algorithms, and the results that were obtained. Finally, in Chapter 4 the conclusions are presented, and we reflect on our approach.
Chapter 2

Methodology

2.1 Background

2.1.1 Reinforcement Learning & Markov Decision Processes

The field of Reinforcement Learning deals with finding algorithms that solve sequential decision problems. In a Reinforcement Learning task, there is an agent that tries to maximize the cumulative reward it receives from the environment. Reinforcement Learning tasks are usually defined as a Markov Decision Process, or MDP. An MDP is characterized by a tuple $M = \langle S, A, P, R \rangle$ The agent has a set of actions $A$ from which it can choose. At each state $s \in S$, the agent can choose an action. What the consequences of this action are is determined by the transition function $P$. This function indicates the probability of going from one state to another after executing action $a$: so $P_a(s')$ indicates the probability of transitioning from state $s$ to state $s'$ under action $a$.

The Markov Property states that this transition depends only on $s$ and $a$, not on any states visited (or actions executed) previously. After an action is executed, the agent will receive a reward: $r \in R$, $R : S \rightarrow \mathbb{R}$. $R_a$ indicates the expected reward after executing action $a$ in state $s$. The agent has to learn what the best action is in every state, i.e. for each state find the action that will yield the most reward in the long run. The return of a state describes how much reward the agent is expected to accumulate until the end of the episode when starting from this state. Tasks can be episodic, in which case the task ends after a finite number of time steps (after which the task may or may not be repeated), or continuous, in which case the task never ends. In both cases, time steps are indicated by $t$. Episodic tasks can be completed by the agent attaining its goal (such as a specific state), running out of resources or simply running out of time.
2.1.2 Value Functions & Policies

How the agent acts in a state is determined by the policy \( \pi : S \rightarrow A \), so the agent’s goal is to find the optimal policy. The optimal policy is the policy that will yield the most reward in the long run. A value function \( V_\pi^t \) indicates how much reward an agent can expect to receive when following some policy \( \pi \) from either a state or state-action pair (the state-action value function is usually called \( Q_\pi^t \)) at time step \( t \). The value function is characterized by the following equation (the Bellman Equation[4]):

\[
V_\pi^t(s) = R_\pi^t(s) + \gamma \sum_{s' \in S} P_\pi^t(s)(s') V_\pi^t(s')
\] (2.1)

In this formula, \( \gamma \) indicates a discount factor, ensuring the existence of the value function in non-episodic tasks. The higher the value of a state, the better it is for an agent to be in that state. During the reinforcement learning process, the value function is adjusted to match the experiences of the agent and the goal is to find the optimal value function, which is the value function that describes the expected cumulative reward from every state most accurately. Behaving greedily (taking the action with the expected highest value) with respect to the optimal value function corresponds to following the optimal policy. The optimal value function is characterized by the Bellman Optimality Equation:

\[
V^*(s) = \max_a \sum_{s'} P^a_s(s') \left[ R^a_s(s') + \gamma V^*(s') \right]
\] (2.2)

The optimal value function is approached by the process of generalized policy iteration, or GPI. The agent starts out with some value function and behaves greedily with respect to this function. At the next step, the agent can updates the value function with respect to the policy it followed at the previous step. Now the agent has a slightly different value function (that is a step closer to the optimal value function), and behaves greedily with respect to this function. This process continues until it stabilizes. When the policy is completely greedy with respect to its own value function, the Bellman Optimality Equation holds.

2.1.3 Exploration versus Exploitation

Exhaustively searching the state space is intractable for most problems and even impossible in cases where the solution space is continuous. The agent may not have the resources to explore indefinitely, so sometimes it might be desirable to stick with a suboptimal solution and gather as much reward as possible. Finding the balance between obtaining as much reward as possible and sufficiently exploring the solution space is known as the exploration versus exploitation problem. Exploration can be achieved in
several ways. One way is to sometimes pick a random action instead of the greedy action. In this case we talk about an $\epsilon$-greedy policy, $\epsilon$ being the probability that a random action is chosen. Another way is to overestimate the value of unexplored states, which means that the greedy action would often take the agent to unexplored states. It is sometimes desirable to maintain two policies instead of one, where one is used to select actions for the agent (the behavior policy) and the other (the estimation policy) actually reflects the estimate of the current value function. Methods that use two different policies are called off-policy methods, whereas methods that use one policy for behavior and estimation are called on-policy methods. An obvious advantage of off-policy methods is that the agent can learn even if it does not have control over the actions picked. Off-policy methods, however, have the risk of diverging towards infinity instead of converging to some (sub-)optimal policy[5] when used in combination with linear function approximation, unless special care is taken[6].

2.1.4 Generalization and Function Approximation

In a continuous observation space, it is impossible for the agent to visit every state and store the value for this state in a table. This is why the value function needs to be approximated. There are many ways to approximate a function, but some ways have more convenient properties than others[7]. For instance, using a method for function approximation that is guaranteed to converge to a sub-optimal solution might be preferable over using a method that might either find a better solution or not find a solution at all (divergence).

One way to approximate the value function that is used a lot in RL is linear gradient descent. In linear gradient descent, the value function is represented as follows:

$$V(s) = \tilde{\theta}_t^T \tilde{\phi}_s = \sum_{i=1}^{n} \theta_t(i) \phi_s(i) \tag{2.3}$$

In linear gradient descent, the value function is assumed to be linear in some parameter vector $\theta$ if the state is described by some feature vector $\phi$. This feature vector is how we generalize over the states. For example, in a 2D world where the agent observes x,y coordinates, the features might be cells in a grid laid over this 2D world with a feature value of 1 for the feature corresponding to the cell that the agent is in, and 0 for the other features. The greater the size of the cell, the greater the amount of generalization over the state space. The parameter vector $\theta$ is adjusted after every sample to reduce the Mean Squared Error (MSE) for that sample:

$$\tilde{\theta}_{t+1} = \tilde{\theta}_t - \frac{1}{2}\alpha \nabla_{\tilde{\theta}_t} \left[V^\pi(s_t) - V_t(s_t)\right]^2 \tag{2.4}$$
For the features, a commonly used method are Radial Basis Functions (RBFs). RBFs are continuous-valued, so the feature values will be in the interval \([0,1]\), reflecting the degree of “presence” of the feature. RBFs are functions (typically Gaussian) that have some center \(c_i\) and some width \(\sigma_i\). The feature value is calculated based on the distance of the measuring point to the center, and the width of the RBF. A wider RBF means that there is a wider areas of generalization. In Figure 2.1 an example of a one-dimensional grid of RBFs is shown. In this case, there are three RBFs, with their centers at \(x = -2\), \(x = 0\) and \(x = 2\) respectively. If the agent were at \(x = 0\), then the feature value for the middle RBF would be high (1) while the values of the left and right RBFs would be lower (about 0.45, in this case). If the parameter for the width of the RBFs had been higher, then those values would also have been higher.

2.1.5 Temporal Difference Learning

Problems in which a model of the world is not given can only be solved by interacting with the world. Temporal Difference (TD) methods are methods that interact with the world in order to learn, requiring no prior knowledge. The advantage of TD methods over other methods (such as Monte Carlo methods) is that they will update the value function during the episode, by bootstrapping the value for a state off the subsequent states. In TD methods, therefore, we base our estimate on another estimate. Other methods do not bootstrap. This means that they are usually a bit more robust against
violations of the Markov property, but they also tend to learn more slowly, because they only update their value function at the end of an episode. The TD update is as follows:

\[ V(s) = V(s) + \alpha [r + \gamma V(s') - V(s)] \] (2.5)

The \( \alpha \) parameter (a value in the interval \([0,1]\)) represents the learning rate of the agent, or how much we want to adjust the value for state \( s \) with respect to the latest experience. Especially in function approximation it is not wise to choose too big a learning rate, because completely matching all samples is usually not possible in function approximation.

### 2.2 Related Work

Generalization in Reinforcement Learning is a hard problem that requires knowledge of a great part of the field. See [8] for a very informative, though slightly dated, introduction in the field of Reinforcement Learning. Closely related to the problem of generalization is the area of function approximation, since in the continuous case we will have to extrapolate our value function from a limited number of samples. The subject is shortly introduced in [8], but the collection of papers [9] provides a useful overview as well.

In [10], an approach is used that was considered for implementation. A multilayer perceptron is used to approximate the value function, minimizing MSE over not just the most recent sample but over a collection of samples. The agent does not need to have a model of the world. Multilayer Perceptrons are an attractive way to represent the value function, as they can approximate non-linear functions. The approach was rejected, however, because of the additional complexity it introduces. In linear function approximation, the local minimum in Mean Squared Error is the same as the global minimum, which greatly simplifies exploration of the solution space. As shown by [11], the MSE of linear function approximation converges to a parameter vector with the following properties:

\[ \text{MSE}(\theta_\infty) \leq \frac{1 - \gamma \lambda}{1 - \gamma} \text{MSE}(\theta^*) \] (2.6)

Where \( \theta^* \) is the optimal parameter vector. In other words, a simple linear method would be sufficient for the experiments we wanted to conduct.

In [12], an algorithm called AMBI is suggested. AMBI uses a lazy form of function approximation, storing samples in a database and interpolating the value of a state from neighboring points. The algorithm also approximates a model by the same principle; effects of actions close to state \( s \) should be similar to the effects of the action in state \( s \). Lazy function approximation was not considered at first, due to the relatively high dimensionality of our observation space. It would take a lot of samples to create a reliable
estimate for all states, in fact, this number is exponential in the number of dimensions. Since all of these samples need to be stored and sorted, it would add a great amount of computational complexity. Whether lazy function approximation is better in high dimensional observation spaces than e.g. RBF linear gradient descent is an interesting question, but beyond the scope of this paper.

In last year’s Competition, the winner used a TD(\(\lambda\)) algorithm with KNN function approximation. No description of this method was published, but it proved quite successful during evaluation. However, in last year’s competition, the agent had an observation space of only 4 dimensions, which made the KNN algorithm a lot more attractive.

## 2.3 Algorithms

We have implemented and evaluated 2 algorithms. Both algorithms are on-policy Temporal Difference methods that use linear gradient descent function approximation.

### 2.3.1 Benchmark Algorithm

The benchmark algorithm is a basic implementation of the on-policy TD method[13], commonly referred to as the Sarsa algorithm[14]. Function approximation is done using linear gradient descent and using Radial Basis Functions as features. The RBFs are laid out in a uniform grid over the observation space, so the complexity in that regard is \(k^n\), \(k\) being the number of grid cells per dimension and \(n\) being the number of dimensions. The number of RBFs is therefore exponential in the number of dimensions, so if the observation space is high dimensional, the number of grid cells per dimension needs to be very low in order to keep the algorithm computationally tractable. Because RBFs are continuous, states are still described uniquely even if the number of RBFs is low. There is a caveat, however: if the RBFs are not wide enough, then the values for some features may be too small to distinguish from zero in a floating point representation. In this case, perceptual aliasing could occur. On the other hand, choosing a higher width for the RBFs means that there is a high degree of generalization, which might cause instability in the value function and slow down learning.

In Algorithm 1 the pseudo code for the first algorithm is listed. The \(\vec{e}\) vector is an eligibility trace[15], which greatly improves learning in tasks with delayed reward. In tasks with delayed reward it is a sequence of actions rather than a single action that will produce a (high) reward. In normal TD methods, it will take a while for the value to propagate along this sequence, but if eligibility traces are used this will be a lot faster.
An eligibility trace $\vec{e}_t$ gives a numerical indication of what states were visited before time $t$. The sample at time $t$ influences not only the value of the state that the agent is in at time $t$, but also the states in the eligibility trace, weighted by the eligibility value. Eligibility decays after every time step by a factor of $\gamma; \gamma \in [0, 1]$. TD($\lambda$) is the basic TD algorithm enhanced with eligibility traces. Exploration in this algorithm is done using an $\epsilon$ value; with a probability of $\epsilon$ a random action is chosen.

Algorithm 1: Algorithm 1 pseudo code

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Data:</strong> Parameters $\alpha$, $\gamma$ and $\epsilon$</td>
</tr>
<tr>
<td>2</td>
<td>initialize $\vec{\theta}$ arbitrarily</td>
</tr>
<tr>
<td>3</td>
<td><strong>repeat</strong></td>
</tr>
<tr>
<td>4</td>
<td>$\vec{e} = \vec{0}$</td>
</tr>
<tr>
<td>5</td>
<td>Observe initial state $s$, pick action $a$</td>
</tr>
<tr>
<td>6</td>
<td>$\vec{\phi}_s = \text{feature values at state } s$</td>
</tr>
<tr>
<td>7</td>
<td><strong>while</strong> State not terminal <strong>do</strong></td>
</tr>
<tr>
<td>8</td>
<td>$\vec{e} = \vec{e} + \vec{\phi}_{s'}$ with an upper bound of 1 for each component</td>
</tr>
<tr>
<td>9</td>
<td>Take action $a$, Observe reward $r$ and next state $s'$</td>
</tr>
<tr>
<td>10</td>
<td>$\vec{\phi}_{s'} = \text{feature values at state } s'$</td>
</tr>
<tr>
<td>11</td>
<td><strong>for</strong> $b \in A$ <strong>do</strong></td>
</tr>
<tr>
<td>12</td>
<td>$Q[b] \leftarrow \vec{\phi}_{s'}^T \vec{\theta}$</td>
</tr>
<tr>
<td>13</td>
<td><strong>end for</strong></td>
</tr>
<tr>
<td>14</td>
<td>$a' = \arg \max_{b \in A} Q[b]$</td>
</tr>
<tr>
<td>15</td>
<td>With probability $\epsilon$: replace $a'$ with random action</td>
</tr>
<tr>
<td>16</td>
<td>$\delta = r + \gamma Q(s', a') - Q(s, a)$</td>
</tr>
<tr>
<td>17</td>
<td>$\vec{\theta} = \vec{\theta} + \alpha \delta \vec{e}$</td>
</tr>
<tr>
<td>18</td>
<td>$a = a', s = s'$</td>
</tr>
<tr>
<td>19</td>
<td>$\vec{e} = \gamma \lambda \vec{e}$</td>
</tr>
<tr>
<td>20</td>
<td><strong>end while</strong></td>
</tr>
<tr>
<td>21</td>
<td>$\delta = r - Q(s, a)$</td>
</tr>
<tr>
<td>22</td>
<td>$\vec{\theta} = \vec{\theta} + \alpha \delta \vec{e}$</td>
</tr>
<tr>
<td>23</td>
<td><strong>until</strong> End of Task</td>
</tr>
</tbody>
</table>

### 2.3.2 Improved Algorithm

A disadvantage of the algorithm listed above is that it does not use its experiences very efficiently. In situations where we care little about computational complexity, the most expensive resource would be interactions with the real world, especially if we do not have a model of the world and cannot reason predictively. In Algorithm 1, each sample is considered just once and then forgotten about, its only contribution being a small
change in the parameter vector. Linear gradient descent only tries to minimize the Mean Squared Error for one sample at a time. In order to use our data more effectively, an experience replay method can be used. In experience replay, a number of samples are retained in a database for some time, all of which are used in updating the parameter vector. Instead of adjusting the parameter vector only to the most recent sample, every sample in the database is used. This means that the parameter vector will be adjusted with respect to \( n \) samples at each time step, which should make the value function less susceptible to distortion by outliers. That is, if there is a single sample that does not very well match the current estimate of the value function, the value function does not change very drastically, so the overall MSE over all samples will likely be lower.
Algorithm 2 Algorithm 2 Pseudo code

1: **Data:** Parameters $\alpha$, $\gamma$, $\epsilon$ and $\kappa$
2: initialize $\vec{\theta}$ arbitrarily
3: **repeat**
4: Clear sample database
5: $\vec{e} = \vec{0}$
6: Observe initial state $s$, pick action $a$
7: $\vec{\phi}_s = \text{feature values at state } s$
8: **while** State not terminal **do**
9: $\vec{e} = \vec{e} + \vec{\phi}_{s'}$ with an upper bound of 1 for each component
10: Take action $a$, Observe reward $r$ and next state $s'$
11: Add $(s, a, r, s')$ sample to database
12: **if** Database size $> \kappa$ **then**
13: Remove oldest sample from database
14: **end if**
15: $\vec{\phi}_{s'} = \text{feature values at state } s'$
16: **for** $b \in \mathcal{A}$ **do**
17: $Q[b] \leftarrow \vec{\phi}_{s'}^T \vec{\theta}$
18: **end for**
19: $a' = \arg \max_{b \in \mathcal{A}} Q[b]$  
20: With probability $\epsilon$: replace $a'$ with random action
21: $\delta = r + \gamma Q(s', a') - Q(s, a)$
22: $\vec{\theta} = \vec{\theta} + \alpha \delta \vec{e}$
23: $a = a'$, $s = s'$
24: $\vec{e} = \gamma \lambda \vec{e}$
25: **for all** Sample $x$ in database **do**
26: $\delta = x_r + \gamma Q(x_{s'}, x_{a'}) - Q(x_s, x_a)$
27: $\vec{\theta} = \vec{\theta} + \alpha \delta \vec{\phi}_{xs}$
28: **end for**
29: **end while**
30: $\delta = r - Q(s, a)$
31: $\vec{\theta} = \vec{\theta} + \alpha \delta \vec{e}$
32: **until** End of Task
Chapter 3

Experiments & Results

3.1 Preliminary Experiments

In order to test the algorithm and observe that Algorithm 2 really is an improvement on Algorithm 1, some experiments were conducted on a specific, typical Reinforcement Learning task. The task is the Mountain Car Problem, see Figure 3.1. The agent controls the car. Its goal is to reach the goal state at the top of the mountain, but the car does not have enough power to drive up the mountain. Instead, the car will have to build up speed by going left and right. The agent will receive a reward of -1 at every time step, so the longer the agent takes to reach the goal, the less reward it will get. The agent has 2 continuous observation dimensions: its speed and its position. It can take three actions: full throttle forward, full throttle backward or no throttle.

Both algorithm were tested on this problem. Learning rate was set to $\alpha = 0.01$ for both algorithms, and $\epsilon$ to 0.1. For Algorithm 1, $\lambda$ was set to 0.9 and for Algorithm 2, $\kappa$ was set to 20. 10 RBFs per dimension were used. Every algorithm was run for 500 episodes, and the results were averaged over 30 different trials. The agent starts in the same spot every episode. In Figure 3.2, the results are listed. It is clear that both algorithms

Figure 3.1: The Mountain Car Problem
do indeed learn a good policy after interacting with the MDP for a while. It is also clear that Algorithm 2 outperforms Algorithm 1: the former has found a good policy after about 100 episodes, while the latter reaches a comparable policy only at the end of the 500 episodes. Algorithm 2 takes about 80 time steps to reach the goal after it has learned the best solution. Results vary slightly around 80, likely due to the $\epsilon$ value of 0.1. Normally, after a good policy is found this value is set to 0, making the policy purely greedy. An alternative is decaying $\epsilon$ over time. However, it is hard to find a good heuristic to do this in a general sense, and introducing yet another parameter only adds to the complexity of the problem.

3.2 Experiment setup

To thoroughly test the algorithms discussed previously, we evaluated them on the Reinforcement Learning Competition’s Polyathlon domain training MDPs. There are 6 training MDPs, but what kind of problem they represent is unknown. A proper general agent should be able to learn in situations that were not foreseen during design. There were some specifications that the agent should comply with. These were:
Table 3.1: Parameters for Algorithm 1

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Parameter Description</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>λ</td>
<td>Eligibility Decay</td>
<td>0.9</td>
<td>0.8</td>
<td>0.7</td>
</tr>
<tr>
<td>α</td>
<td>Learning Rate</td>
<td>0.1</td>
<td>0.01</td>
<td>0.001</td>
</tr>
<tr>
<td>ε</td>
<td>Random Action Probability</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
</tr>
</tbody>
</table>

- The agent will have 6 continuous valued observation dimension, with every value in the interval [0,1]
- All MDPs are stationary, they do not change over time
- The observations are Markov, but may contain superfluous information
- The agent will be able to take 6 discrete actions
- The rewards received are in the interval [-1,1]
- Every task will run for 100000 (one hundred thousand) steps

A six-dimensional observation space is not usually encountered in canonical Reinforcement Learning problems, but such dimensionalities are common in real-world applications such as robot car or helicopter navigation. This relatively high dimensionality in the observation space presents a computational challenge, however, the specification does not penalize for computational costs. In real world situations, it usually does not matter if an agent takes a while to learn a good policy. Once the policy has been learned, the computational costs are very low: all the agent has to do is consult its policy. Of course, if the problem is non-stationary (it changes over time), the agent will have to keep learning. In such situations, computational costs could become an issue.

Each algorithm has a number of parameters. We ran a series of experiments, testing both algorithms on 6 MDPs. We did this for several different parameter settings. The parameters we used are listed in Table 3.1, along with the value used. Every permutation of these values was tried. These values were chosen in preliminary experiments, showing them to be in an interesting interval where variation in their values would show a noticeable effect in performance.

We did the same for the second algorithm, but because this algorithm does not use eligibility traces we have left out the $\lambda$ parameter. Instead we use the $\kappa$ parameter, which represents the number of samples replayed at each time step. The parameters along with their settings are listed in Table 3.2.

Experiments were run in 50000 steps instead of the prescribed 100000. 50000 steps takes only half the time to run, and preliminary experiments showed that the difference in
Table 3.2: Parameters for Algorithm 2

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Parameter Description</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa$</td>
<td>Samples Replayed</td>
<td>5</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Learning Rate</td>
<td>0.1</td>
<td>0.01</td>
<td>0.001</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Random Action Probability</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
</tr>
</tbody>
</table>

...performance had become more than apparent after 50000 steps. Episodes were cut off if they took longer than 10000 steps, sometimes restarting helps the agent find the goal state in cases of sparse or delayed reward. The number of RBFs per dimension was kept at a constant of 5; so there were $5^6 = 15625$ RBFs in total.
3.2.1 Implementation Details

The interface between agent and MDP is formulated in RL-Glue[16], a software framework for Reinforcement Learning tasks. The MDPs used in the experiments are the training MDPs included in the competition software, found on the Reinforcement Learning Competition website\(^1\). The interface specifies functions for the agent called by the MDP, of which the most important one is \texttt{agent\_step}. This function is passed two arguments: the state the agent currently observes and the reward it has received after the previous action. The function returns the action that the agent has decided to take in the current state. This function is called at every time step until the end of the episode. At the start and the end of the episode, special functions are called that signal the agent that the episode is starting or ending. The MDP software can simply start a new episode then, or it can end the task and terminate the MDP and agent programs. After every episode, the reward accumulated during that episode was stored to measure performance. The MDP itself provides the discount factor \(\gamma\). It also passes the agent information about the task through a Task Specification, which contains settings such as the number of observation dimensions, the nature of these dimensions (continuous or discrete) and the number of action dimensions.

The agent was implemented in Java. Experiments were run concurrently on different computers, as it took 20 minutes to 2 hours to run the set of MDPs with each parameter configuration, depending on the algorithm and the parameter values. There were 27 experiments to run for each algorithm, so doing these experiments sequentially took quite a while. Especially experiments with a high \(\kappa\) took a long time to run; this was due to a design decision not to store the feature values for every sample. Java does not handle high space complexity well, terminating the virtual machine of it is out of memory. This is not convenient thing to happen in a series of automated experiments that takes quite a while to complete. This is why the feature value vector for each sample has to be calculated during replay.

3.3 Results of Experiments

The performance of the agent was measured as the reward accumulated over the 50000 steps that the MDP was run for. In Figure 3.3 some results are listed. The results listed were obtained with the parameter setting \(\alpha = 0.1, \epsilon = 0.2, \kappa = 15\) and \(\lambda = 0.9\). These particular results were averaged over 20 trials.

\(^1\)http://2009.rl-competition.org/software/15-rl-competition-2009.tar.gz
Chapter 3. Experiments & Results

Figure 3.3: Results on 6 MDPs. Blue is Algorithm 1, red is Algorithm. Listed in order, left to right, top to bottom. 2
The results listed here were chosen because they were among the best results. Especially the Algorithm 1 often produced the results where the agent did not appear to have learned anything at all. With too low an $\alpha$ (learning rate), both algorithm did not improve their performance at all in MDPs 2 and 3. It was only with the highest value of $\alpha$ that we tried that good results were obtained. If too many samples were used for replaying in the Algorithm 2, performance tended to drop as well. Performance seemed pretty good at $\kappa = 15$, but $\kappa = 30$ was not any better (worse on some problems) and took a lot more computation time. Results for Algorithm 1 were best with a $\lambda$ of 0.9. The value of $\epsilon$ did not seem to have a lot of effect on performance. During the learning process, it does not make a lot of difference if a random action is taken with a probability of 0.2 or 0.3. If a good policy has been found, it is preferable not to take random actions, but in the experiments we conducted this situation did not occur often, as the agent was still in the process of learning a good policy. The graphs in Figure 3.3 give the impression that Algorithm 2 is a bit better, and if we look at the total accumulated reward per MDP, as listed in Table 3.3, then we see that in most MDPs, Algorithm 2 has indeed gathered the most reward.

Finally, we ran the Mountain Car experiment one more time, with the algorithms using the parameters that gave the best results on the 6 MDPs ($\alpha = 0.1, \epsilon = 0.2, \kappa = 15$ and $\lambda = 0.9$). The results are shown in Figure 3.4. It is obviously a completely different
Table 3.3: Cumulative reward per MDP. Highest score is underlined per MDP.

<table>
<thead>
<tr>
<th>MDP0</th>
<th>MDP1</th>
<th>MDP2</th>
<th>MDP3</th>
<th>MDP4</th>
<th>MDP5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 1</td>
<td>35417</td>
<td>-25822</td>
<td>-53944</td>
<td>-49794</td>
<td>-132502</td>
</tr>
<tr>
<td>Algorithm 2</td>
<td>35421</td>
<td>-24180</td>
<td>-50512</td>
<td>-43026</td>
<td>-132217</td>
</tr>
</tbody>
</table>

The first 100 episodes performance is about the same, although at its best it is still twice as bad as the solutions found in the preliminary experiments. After about 100 episodes, Algorithm 2 never again succeeds in finding the goal (episodes are reset after 50000 steps, though the reward for those episodes is shown as -5000 for scaling purposes). Algorithm 2 gets unstable after about 130 episodes.
Chapter 4

Conclusion & Discussion

4.1 Conclusion

From the results of the experiments, we can conclude that the Algorithm that uses Experience Replay is more general than the basic algorithm. It outperformed the basic algorithm at almost every task. Learning was achieved in most of the tasks, though the learning was not always very fast. Also, both methods were extremely sensitive to the setting of their parameters. Although a good setting was found that seemed to work pretty well on most of the MDPs that we tested on, it is unlikely that this setting will work well on other problems. For example, when using these parameters to test the algorithms on our implementation of the Mountain Car problem from the preliminary experiments, performance was not very good compared to the results from the initial experiment. It is our belief that the parameter values influence performance a great deal. It is very improbable that a fixed value for the parameters will produce good results on all MDPs. An agent designed to learn on different types of problems should therefore have some mechanism of adjusting the parameters.

Surprising was that performance of Algorithm 2 did not keep increasing with the number of samples replayed. Too many samples seemed to even hurt performance. A possible explanation for this is found in the fact that we learn on-policy. After each step, the value function is altered slightly, which may cause the greedy action in a state to change. If we replay too many samples that are no longer “valid” according to our behavior policy, then the parameter vector could be distorted by these samples. This could especially be a problem at the start of the learning process, where the behavior policy tends to change a lot in a short time.
4.2 Discussion

It is clear that the algorithms described in this paper do not generalize very well. Although Algorithm 2 was an improvement on Algorithm 1, performance was not great. The approach we took has a couple of serious disadvantages, and these disadvantages proved to be fatal in the experiments. The following issues have become apparent:

- RBFs do not seem to be the ideal choice for function approximation. Rather, a lazy nearest neighbor function approximation approach may be preferable. The disadvantage of RBFs is the high computational cost in observation spaces with lots of dimensions. Because of this high cost, we were forced to go for a low number of RBFs (5), whereas in initial experiments a higher number of RBFs seemed to improve performance radically. The RBFs are spread out in a uniform grid over the observation space, which is not ideal. Some parts of the observation space may never be visited by the agent, so having RBFs in those regions is wasteful. A K-Nearest Neighbor approximation, on the other hand, will tend to be more accurate in areas that are visited often, and no calculations are wasted on parts of the space where the agent never comes. Additionally, the parameter vector that is kept for RBFs tries to model the value function over the whole observation space, whereas a KNN algorithm only approximates the value function locally. Still, KNN also brings a lot of computational overhead; the neighborhood of a point needs to be calculated or stored, both operations are very tricky to implement efficiently.

- The approach has a lot of parameters. As shown in the experiments, parameter values can have a big impact on the performance on a task. These findings are supported by [17]. An approach that does not rely as much on such parameters would be more robust. Another solution would be a good heuristic for the agent to select these parameters; if the agent can find a way to decide by itself what the values of the parameters should be, then that would be a big step forward.

- The approach does not use a model. Since our most precious resource are the samples we obtain by interacting with the world, we want to make as much of that resource as possible. By updating the value function, we use the information gained from the reward. However, the we have observed a transition from one state to another under a certain action. This is information that can be used to approximate a model, and a model allows us to do planning. In planning, we can learn without interacting with the world, by reasoning e.g. about how to get to a state with a high value. Exploration can also be done much more efficiently if a model of the transitions is known. For example, if a model is known, the agent can give bonuses to actions that take it to an unexplored state.
It would seem that because of these disadvantages, there is little hope for this method as a means of generalization over Reinforcement Learning tasks. Especially RBF function approximation proved not to scale very well to higher dimensions, and the naive approach to parameter choice were fatal to performance and need to be improved upon.

4.3 Future work

One important simplification that has been made in the Polyathlon domain is the Markovian nature of the problems. Assuming that the observations are Markovian is not realistic in real-world applications. If we are dealing with actual robots with physical sensors, there is almost always noise or uncertainty in the observations. In this case, we have a Partially Observable Markov Decision Process, or POMDP. POMDPs add another layer of complexity to the problem, but a general agent must be able to deal with noisy observations. Another simplification is the discrete nature of the action space. If the action space is continuous instead, the problem gets that much harder. However, it is yet another addition that would make the agent more general. Perhaps a more powerful way of function approximation is required as well, linear gradient descent cannot approximate non-linear functions. If the value function happens to be non-linear, the agent will therefore never learn a good approximation for it. As has become apparent in the experiments, it is also vital to have the right values for the method’s parameters. Future research could focus on the agent learning to choose its parameters; this could even be modeled by an MDP.
Bibliography


