

Unleashing the Horde

A modern reinforcement learning architecture
reimplemented

Author:
Thomas Planting
3030938

Supervisors:
Dr. ir. Martijn van Otterlo
Dr. Ida Sprinkhuizen-Kuyper

August 31, 2012

Bachelor Thesis
Artificial Intelligence
Faculty of Social Sciences

Radboud University Nijmegen



Abstract

Sutton et al. (2011) proposed a new reinforcement learning architecture called Horde. This architecture is based on the parallel processing of sensorimotor data. We have implemented the Horde architecture in our own simulated environment. This environment is a simple continuous MDP. We have done this to investigate if the paper provides enough information to reconstruct the algorithm, which design choices that are not explicitly mentioned in the paper have to be made to reconstruct the architecture, and whether the algorithm generalizes well to a different simulation environment. Several tests were run and the results were analyzed.

Keywords: artificial intelligence, knowledge representation, real-time, reinforcement learning, off-policy learning, temporal difference learning, value function approximation, tile coding, general value functions, GQ(λ), predictions, parallel processing

Contents

1	Introduction	3
2	Notation and Background	5
2.1	Markov Decision Processes	5
2.2	Definitions	7
2.3	Exploration	8
2.4	Generalization	8
2.5	Eligibility traces	8
3	The Horde architecture	10
3.1	Introduction	10
3.2	Off-policy learning	11
3.3	General Value Functions	11
3.4	GQ(λ)	12
3.5	Demons	13
4	Implementation and design decisions	14
4.1	The environment	14
4.2	The features	15
4.3	The policies	18
4.4	The overall architecture	18
5	Experiments and Results	19
5.1	Introduction	19
5.2	Experiment 1	19
5.3	Experiment 2	21
5.4	Experiment 3	24
6	Conclusion	28
	References	30

Chapter 1

Introduction

The Horde architecture (Sutton et al., 2011) uses a new reinforcement learning technique based on the parallel off-policy unsupervised processing of sensorimotor data. The Horde Architecture is state-of-the-art. The paper in which this architecture was introduced was published in 2011. The main value of this architecture is the learning of general knowledge of the world from sensorimotor data. By making predictions bottom up and in parallel, Horde is able to learn without much specific prior knowledge. Horde learns how to interpret its environment while learning how to act in it at the same time. The Horde architecture consists of a single agent with potentially thousands of sub-agents, called demons. Each demon learns its own small piece of information about the world.

The Horde architecture learns from raw data. Many reinforcement learning algorithms use pre-determined classifiers for objects in the environment, like walls and cliffs. The Horde architecture is able to learn directly from a sensory stream. With a sensory stream we mean the stream of all sensory information an agent receives. Horde can process this by assigning different demons to different sensors and combinations of sensors. This way, each demon is responsible for interpreting its own small piece of data. By then making use of these small pieces of information, other demons will be able to solve more complex problems. For example, a couple of demons could predict that stench and soft ground mean that the agent is near a swamp. Another demon could use this information to learn that if it is in a swamp, it will get stuck. Yet another demon could learn that it is bad to get stuck. This way, the overall architecture will have learned to stay away from places with a lot of stench and soft ground. It has done this by using the predictions from individual demons to learn about its environment without using a hard coded classifier for swamps. This way, Horde is really able to learn from the ground up about its environment.

Being able to learn from the ground up is what makes the Horde architecture so general. Like biological beings, Horde is able to learn without much specific knowledge of the environment. This makes Horde very interesting for fields which pursue learning from unknown environments, like developmental robotics (Weng, 2004). The idea of learning in parallel also bears a resemblance to biological creatures. Take a baby, for instance, that moves its hand in seemingly random directions. While doing this, it learns all kinds of things about gravity, motor control in 3 dimensions, different muscle activations and all kinds of sensations accompanied with the movement. In much the same way (on an abstract level) Horde learns many different small pieces of information at the same time.

We implemented the architecture in our own simulated environment, which we based upon a simple example Markov Decision Process from Russell and Norvig (2003, Chapter 17). We did however make some adjustments on this MDP. Most notably, we changed it from a discrete grid world to a continuous environment, thereby allowing us to make tests in a more real world like situation. The goal of the thesis is to find out if the architecture still works in such a deliberately different environment, and at the same time to see what pitfalls have to be overcome and what major design choices have to be made that are not explicitly described in the paper of Sutton et al.

Our research questions are:

1: What major design choices not explicitly presented in the paper Horde: A Scalable Real-time Architecture for Learning Knowledge from Unsupervised Sensorimotor Interaction (Sutton et al, 2011) have to be taken to create a working implementation of it?

2: Will the Horde architecture work well in a different simulation?

First we will present the notation we will use in this thesis and the background knowledge that is required to understand the rest of the thesis. Then we will explain the Horde architecture in greater detail. After that we will outline our own implementation of the architecture, in which we will explicitly explain our design choices. Following that will be a description of the experiments we did, using different kinds of demons and question functions, and their results. We finish with the conclusion.

Chapter 2

Notation and Background

In this chapter we explain Markov Decision Processes, introduce the notation used in this thesis, and explain the concepts of exploration, generalization, and eligibility traces for reinforcement learning.

2.1 Markov Decision Processes

A reinforcement learning problem usually consists of an agent in a certain environment, which can be either simulated or real. It is the goal of the agent to learn for every part of the environment what action it has to take to maximize the reward. The agent learns by running training cycles, in which it notices the rewards it gets for doing certain actions or going to certain parts of the environment. Note, however, that in reinforcement learning the agent does not get direct feedback if an executed action is the right one, the only clue he has for that is the reward it gets over time.

The standard framework for reinforcement learning is a Markov Decision Process, or MDP. An MDP is a framework in which on each time step t , the agent is in a certain state, and it can choose any action that is available in that state. More formally, an MDP is a 4-tuple $\langle S, A, T, R \rangle$, where S is a finite set of states the agent can be in, A is a finite set of actions the agent can take, the transition function $T : S \times A \times S \rightarrow [0, 1]$ gives the probabilities in each state to end in another state by performing an action (that is $T(s, a, s')$ is the chance that performing action $a \in A$ in state $s \in S$ will result in state $s' \in S$), and $R : S \times A \times S \rightarrow \mathbb{R}$, such that $R(s, a, s')$ is the immediate reward that is received upon moving from state s to state s' with action a . If a Markov Decision Process is deterministic performing an action a in a state s always results in the same next state s' . This means every $T(s, a, s')$ is either 1 or 0. Note that, in both deterministic and non-deterministic (stochastic) MDPS, the values of T for all possible outcomes of performing any action a in any state s sum to one, or more formally:

$$\forall s \in S, \forall a \in A \left(\sum_{s' \in S} T(s, a, s') = 1 \right)$$

The reward R consists of two different types of rewards: a transient reward and a terminal reward. The transient reward $r : S \times A \times S \rightarrow \mathbb{R}$ is the reward an agent always gets. It is usually a low negative value, and it can be viewed as the cost of performing an action. The terminal reward $z : S \rightarrow \mathbb{R}$ is the extra reward that is received upon entering a certain terminal state, which is independent of the action that was taken to get there or of the state the agent came from. A terminal state is a state which terminates the process if the agent enters it. To determine if a state is terminal, a *termination function* $\gamma : S \rightarrow [0, 1]$ is used. For each state, the termination function determines the chance that state is **not** a terminal state. This is a bit counterintuitive, but it is the notation as provided in the original Horde paper (Sutton et al., 2011). As it is usually known which states are terminal, the termination function γ will often give a 1 or a 0 for each state.

A terminal reward is usually a high positive or negative value, and it can be viewed as the reward for reaching a goal (in the case of a positive value) or as failing (in the case of the negative value). Note that the total reward received upon moving from a state s to a state s' with an action a equals the sum of the transient reward and the terminal reward for that step. Written as an equation, this results in:

$$R(s, a, s') = r(s, a, s') + (1 - \gamma(s'))z(s')$$

The terminal reward is multiplied with $1 - \gamma(s')$ because that is the chance the state that was moved to is terminal. This means that if a state does have a chance to be terminal, the terminal reward can be set to its desired value and the state will still receive a terminal reward proportional to that chance, while all states that are not terminal (and thus have a γ of 1) will not receive a terminal reward.

A special property of MDPs is the so-called Markov Property, which means the next state s' depends only on the current state s and the chosen action a . In other words, the next state is conditionally independent of all previous states and actions, and the transition probabilities $T(s, a, s')$ only depend on the current state s and the actions a , and not on the history.

Now, for a simple example: we consider a 5×5 MDP (see Figure 2.1). The terminal states are the states with the x and y coordinates (4,1) and (1,5). State (4,1) has a terminal reward of +1 and state (1,5) has a terminal reward of -1. The transient reward is -0.01. The agent is in state (3,1) and chooses an action a which results in an 80 percent chance the agent will move up, a 10 percent chance it will move to the left and a 10 percent chance it will move to the right. Moving up will increase the y coordinate by 1, moving left will decrease the x coordinate by 1, and moving right will increase the x coordinate by 1.

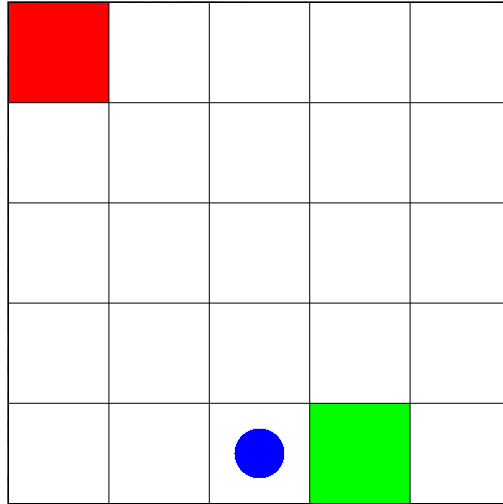


Figure 2.1: The MDP in this example

The resulting T , r , z and γ values will be as follows:

$$\begin{aligned} T((3,1), a, (3,2)) &= 0.8 & r((3,1), a, (3,2)) &= -0.01 & z(3,2) &= 0 & \gamma(3,2) &= 1 \\ T((3,1), a, (2,1)) &= 0.1 & r((3,1), a, (2,1)) &= -0.01 & z(2,1) &= 0 & \gamma(2,1) &= 1 \\ T((3,1), a, (4,1)) &= 0.1 & r((3,1), a, (4,1)) &= -0.01 & z(4,1) &= 1 & \gamma(4,1) &= 0 \end{aligned}$$

This results in the following R values:

$$\begin{aligned} R((3,1), a, (3,2)) &= r((3,1), a, (3,2)) + (1 - \gamma(3,2))z(3,2) = -0.01 + 0 * 0 = -0.01 \\ R((3,1), a, (2,1)) &= r((3,1), a, (2,1)) + (1 - \gamma(2,1))z(2,1) = -0.01 + 0 * 0 = -0.01 \\ R((3,1), a, (4,1)) &= r((3,1), a, (4,1)) + (1 - \gamma(4,1))z(4,1) = -0.01 + 1 * 1 = 0.99 \end{aligned}$$

2.2 Definitions

In Section 2.1 we introduced MDPs and some notation, in this section we give the complete notations we use in the rest of our thesis.

The base problem: The task of the agent as defined by the environment. Often, an environment has certain positive states and certain negative states. Learning the base problem means learning how to act in the environment the agent is in to maximise the reward.

- A is the set of all possible actions the agent can take. For example, in the environment we use in this thesis the actions are UP, DOWN, LEFT and RIGHT. A specific action out of A is usually denoted as a .
- S is the set of all possible states in the environment the agent can be in. A specific state out of S is usually denoted as s .
- t is the current time step. It is usually used in subscript to denote the value of something on a certain time. For example, s_t means the state the agent is in on time t .
- r $r : S \times A \times S \rightarrow \mathbb{R}$: is the transient reward. $r(s, a, s')$ is the reward the agent gets if it moves from state s to state s' with action a .
- z $z : S \rightarrow \mathbb{R}$ is the terminal reward. $z(s)$ is the reward the agent gets if it gets to state s if that is a terminal state.
- γ $\gamma : S \rightarrow [0, 1]$ is the termination function. This represents the chance this state is a terminal state.
- R $R : S \times A \times S \rightarrow \mathbb{R}$ is the total reward received. $R(s, a, s') = r(s, a, s') + (1 - \gamma(s'))z(s')$
- j $j \in [0, 1]$ is the discount factor. A discount factor is a measure of how important future rewards are. If j is low, future rewards are less important.
- π, π^* $\pi : S \times A \rightarrow [0, 1]$ is a probabilistic policy. This is a function that determines the probability of performing a certain action in a certain state. $\pi(s, a) = p$ means that the probability action a is chosen in state s is p . Note that

$$\forall s \in S (\sum_{a \in A} \pi(s, a) = 1)$$

The optimal policy, or π^* , is the policy that results in the highest expected reward. The goal of a reinforcement learning agent is to learn π^* .

- V^π $V^\pi : S \rightarrow \mathbb{R}$ is the state-value function. $V^\pi(s)$ is the expected return when starting in state s and following policy π thereafter. This is defined as:

$$V^\pi(s) = E^\pi \left(\sum_{k=0}^{\infty} j^k R_{t+k+1} \mid s_t = s \right)$$

where E^π denotes the expected value given that the agent follows policy π and R_t is the reward received on time t , which depends on the state and actions at that time step.

- Q^π $Q^\pi : S \times A \rightarrow \mathbb{R}$ is an action-value function, or Q-function. This is a function which represents the future reward of performing an action a in a state s , and thereafter following policy π . It works the same as a state-value function, except that it also has an action as input. It is defined as

$$Q^\pi(s, a) = E^\pi \left(\sum_{k=0}^{\infty} j^k R_{t+k+1} \mid s_t = s, a_t = a \right)$$

Policies are often determined by Q-functions. In each state, the probability of selecting an action $\pi(s, a)$ depends on the Q-value $Q(s, a)$ of that state and action.

2.3 Exploration

There are two possible motivations with which a reinforcement agent chooses actions during training: exploitation and exploration. Exploitation means choosing actions according to the probabilities of the learned policy π to maximize the reward based on what the agent has already learned. This makes sense because the agent expects to be performing those actions during the actual runs. Because of that, the value functions should reflect the rewards received when the agent follows the policy. However, an agent performing too much exploitation during training runs the risk of getting stuck in a local maximum. If it dismisses all actions that would take it off of the path it thinks is optimal early in the training, it may miss paths that are better in the long run. For that reason, a reinforcement learning agent also has to explore during the training. Exploration means trying out different actions to learn more about the effects of the actions. An agent must always make a tradeoff between exploration and exploitation during training.

2.4 Generalization

If a task has a relatively large amount of states and actions the current state action combination is often represented by a feature vector ϕ . Each state has certain features, and states who are similar in a certain way will have some similar features. This way, instead of having to learn a value function for every state action combination individually, an agent only has to learn a value function for each feature vector action combination. This drastically reduces the search space. These general expected values are represented by a weight vector θ , which is of the same length as ϕ . The expected value of a certain state s equals $\phi(s) \cdot \theta$, where $x \cdot y$ denotes the dot product (inproduct) of the vectors x and y . Basically, the weight vector assigns a value, which can be either positive or negative, to each feature. Depending on how strongly that feature is represented in the current state (usually a value between 1 and 0) the weight value has influence on the total expected value of the state. For more information about generalisation and function approximation see Sutton and Barto (1998, Chapter 8).

2.5 Eligibility traces

Another important aspect of reinforcement learning is the concept of eligibility traces, determined by a parameter $\lambda \in [0, 1]$. Very simply said, an eligibility trace parameter λ determines how much the expected values of states are adjusted by the information of the values of onward states. Let us say we have a trace of all the states an agent has been from the beginning to termination: s_t, s_{t+1}, s_{t+2} etc. If an agent moves from a state s_t to s_{t+1} , it is possible to calculate what the expected value of state s_t should be by the expected value of state s_{t+1} . If the value functions of different states (or different feature vectors) are consistent, the following formula should hold:

$$V^\pi(s_t) = R_{t+1} + j \cdot V^\pi(s_{t+1}) \quad (2.1)$$

This means the value for being in a state equals the expected next reward plus the discount factor times the expected value for being in the next state. Each time the agent moves from a state s_t to a state s_{t+1} , the expected value for state s_t is calculated using the above formula. The value of state s_t is then updated by a learning parameter α times the difference in the expected value from the next state and “current” expected value. More formally:

$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha(R_{t+1} + j \cdot V^\pi(s_{t+1}) - V^\pi(s_t)) \quad (2.2)$$

This is called a *temporal difference (TD)* update. Eligibility traces take this a step further by updating the value of states not only by the value of the next state, but by the values of *the next n states*, where n depends on λ . How n depends on λ is more of a mechanistic question than a theoretical one, and will not be discussed in this thesis. For more information on this subject see Sutton and Barto (1998, Chapter 7). Equation 2.2 could be seen as a special case of this, with $n = 1$. Recall from Equation 2.1

that the value of a state should equal the first reward plus the discounted value of the next state. This makes sense because $j \times V_\pi(s_{t+1})$ takes the place of all future rewards $jR_{t+2} + j^2R_{t+3} + j^3R_{t+4}$ etc. $j \cdot V_\pi(s_{t+1})$ represents the value from state s_{t+1} and onwards, after all. With $n = 2$, this simply goes one step further. With $n=2$ the value of a state equals the first reward plus the discounted second reward plus the twice discounted value of the state that will be reached in two steps. More formally:

$$V_\pi(s_t) = R_{t+1} + j \cdot R_{t+2} + j^2 \cdot V_\pi(s_{t+2}) \quad (2.3)$$

In this case $j^2 \cdot V_\pi(s_{t+2})$ takes the place of all future rewards after reaching the second state after s_t , which are $j^2R_{t+3} + j^3R_{t+4}$ etc. The more general formula is:

$$V_\pi^n(s_t) + j \cdot R_{t+1} + \dots + j^{n-1}R_{t+n} + j^n V_\pi(s_{t+n}) \quad (2.4)$$

Chapter 3

The Horde architecture

3.1 Introduction

The Horde architecture consists of a single agent with potentially thousands of sub-agents, or demons. Each demon is a mostly independent reinforcement learning agent with its own value function. Each demon learns from its own piece of information from the sensory stream the overall agent receives. Every demon is responsible for making its own prediction about the environment or for finding a good policy for a certain sub-problem. This subproblem does not have to be dependent on the base problem. Recall that the base problem is the task of the agent as defined by its environment. Finding a good policy for a sub-problem means learning how to solve a subproblem in a good way. A subproblem could be “do not fall off a cliff” or “do not bump against a wall”, for example. Making predictions could be seen as processing small sensorimotor data into larger, more useable predictions which are useful for filling in value functions. To get back to the example with the swamp in the introduction, the small data were the stench and the soft ground, and the demon could predict out of that whether it was near a swamp. In turn, the small data in this example are probably made out of even smaller data, etc.

In order to be able to learn from different information, and to learn different subproblems, Horde uses General Value Functions (GVFs). GVFs are value functions which receive additional inputs for the termination function, transient reward function and terminal reward function. Instead of copying these variables from the base problem, each GVF can have different values for these variables. Each demon has its own GVF which defines the prediction it has to make or the subproblem it has to solve. General Value Functions will be discussed in more detail in Section 3.3. The overall architecture of Horde is shown in Figure 3.1. This figure is based on a picture in the powerpoint slides of Sutton (2011).

The sensorimotor data of the environment is recoded into a feature vector, as explained in Section 2.4. Each horizontal line represents a single feature, and each vertical line represents a demon. Every demon is a full reinforcement learning agent estimating its own value function. Each demon learns its own weight vector θ , see Section 2.4. Each intersection between a horizontal and a vertical line in Figure 3.1 represents one of those weights. Depending on the weights and the features, the demons then each learn a prediction. After that, the demons can supply this information back to the recoder, and the predictions of the demons can be used in the feature list.

Learning proceeds in parallel by all demons simultaneously so as to extract the maximal training information from whatever actions are taken by the system as a whole. To be able to train multiple demons in parallel the demons have to be able to learn *off-policy*. Off-policy learning is learning a policy which is not the policy that is being followed during the training, and will be discussed more extensively in the next section. The reinforcement learning algorithm used by the individual demons is $GQ(\lambda)$ (Maei & Sutton, 2010). This algorithm has been chosen by the authors of the Horde architecture for both its generality and its effectiveness with off-policy learning.

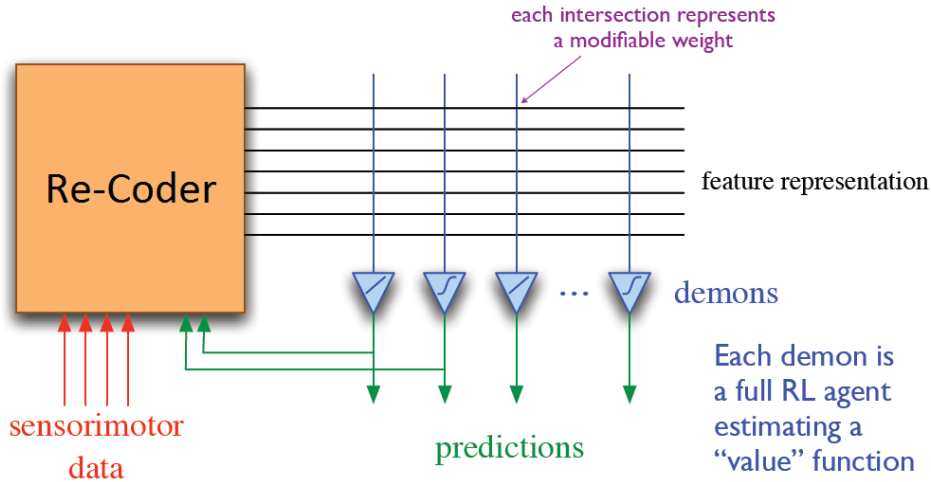


Figure 3.1: The Horde architecture. Adapted from Sutton (2011)

3.2 Off-policy learning

The Horde architecture is based on the parallel processing of multiple demons. Since the agent can only follow one policy at a time, this requires so called off-policy learning. What this means is that the *behaviour policy*, the policy the agent uses to choose his actions during the training cycles, does not have to be the same as the *target policy*, the policy the agent is learning. During test cycles, demons choose their behaviour in the same way as a conventional reinforcement learning agent does. The agent chooses its next states by the probabilities of the policy it is following, and updates its value function. Normally, however, this value function is then used to update the policy the agent is usually following if it is not exploring. With off-policy learning, this is not the case. The agent keeps updating its value function without it having any effect on the actual behaviour during training. The only thing the value function is used for is to generate the target policy, which is the policy the agent is learning. This means one way to act is being learned (the target policy) while behaving in a completely different way (the behaviour policy). This is necessary for Horde because Horde is based on parallel processing of different demons, which are trying to learn different things, while there is only one actual agent moving in the environment. This means each demon has its own target policy, but there is only one behaviour policy which can be executed. Therefore Horde requires off-policy learning.

3.3 General Value Functions

GVs are an essential part of the Horde architecture. The Horde architecture consists of many demons, where each demon learns its own task, formulated as a *question*. Recall that a regular action-value function $Q^\pi : S \times A \rightarrow \mathbb{R}$ is a function which approximates the value of performing an action a in a state s , and thereafter following policy π .

Apart from the direct inputs of the state the agent is in and the action it chooses, a regular action-value function is only considered dependent on the policy π . However, the function is also dependent on the transient reward r and the terminal reward z . These functions could just as well be used as inputs in the same way as π is. A normal value function uses the r and z from the base problem, but we could also define (Sutton et al. 2011) a more general value function: $q^{\pi r z} : S \times A \rightarrow \mathbb{R}$ which takes r and z as inputs. Furthermore, we can define a value function which also takes the termination function as input. This is a little bit less trivial than the previous step, because, unlike with the reward functions, termination usually interrupts the normal flow of state transitions and resets the agent to a starting position or starting state distribution. With a termination function supplied to a General Value Function, this is

simply omitted. The agent still only resets to its starting position when the real base problem terminates, but a general value function with a certain termination function which is different from the termination function of the base problem may be considered to be terminated at any time. Formally a General Value Function is defined as:

$$Q^{\pi r z \gamma} : S \times A \rightarrow \mathbb{R}$$

The functions π , r , z and γ , where π is the target policy, are considered the *question functions*. It depends on these functions what kind of question is asked of the reinforcement learning agent using the GVF. GVFs are so important for the Horde architecture because they allow all kinds of different questions to be asked from demons. A demon with the question functions from the base problem will simply learn the base problem, and a demon with different question functions could learn something completely else. For example, if we made a demon with the target policy that was learned by a demon which tried to solve the base problem, a transient reward of 1, a terminal reward of 0 and the termination function from the base problem, $V(s)$ would yield the expected number of steps from state s to termination if the agent follows the learned target policy.

3.4 GQ(λ)

The reinforcement learning algorithm used by the demons is GQ(λ). This algorithm was chosen by the authors of the Horde architecture for both its generality and its effectiveness with off-policy learning. This algorithm maintains three vectors for each GVF. The first one is the set of weights $\theta \in \mathbb{R}^n$. Recall that this is the set of weights that ultimately decides the expected value in a certain position, because the expected value of a certain state s equals $\phi(s) \cdot \theta$. The second vector is the eligibility trace vector $\sigma \in \mathbb{R}^n$. I won't go into much detail of it now, but this vector remembers how long ago the agent has been in each state, and uses those values to determine how much the expected value of each state should be adjusted, together with the eligibility trace variable λ . This has to do with the mechanistic view of eligibility traces (see Sutton and Barto (1998)). The third vector is a secondary set of weights $w \in \mathbb{R}^n$, which is used to update θ . All three vectors are initialised to 0 (θ can also be initialised arbitrarily).

On each time step, GQ(λ) calculates the expected next feature vector, by summing for each action the probability of that action with the features that would be reached by taking that action. More formally:

$$\bar{\phi}_t = \sum_a \pi(s_t, a) \phi(s_t, a)$$

GQ(λ) also calculates the improvement in expected reward from the last action each time step.

$$\delta_t = r_{t+1} + (1 - \gamma(s_{t+1}))z(s_{t+1}) + \gamma(s_{t+1})\theta \cdot \bar{\phi}_t - \theta \cdot \phi(s_t, a_t)$$

Each time step, GQ(λ) updates the three vectors using these variables:

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha(\delta_t e_t - \gamma(s_{t+1})(1 - \lambda)(w_t \cdot e_t) \bar{\phi}_{t+1}) \\ w_{t+1} &= w_t + \alpha(\delta_t e_t - (w_t \cdot \phi_t) \phi_{t+1}) \\ e_t &= \phi_t + \gamma_t \lambda_t \frac{\pi(s_t, a_t)}{b(s_t, a_t)} e_{t-1} \end{aligned}$$

where α is the learning parameter, ϕ is the target policy, b is the behaviour policy and λ is the eligibility trace parameter.

The approximation that will be found depends on the feature vector ϕ , the behavior policy b and the eligibility trace function λ . These three functions are referred to as the *answer functions*, and are constant in all experiments I ran. For more information on GQ(λ) see (Maei & Sutton, 2010).

3.5 Demons

The demons can be divided into two kinds: *control demons* and *prediction demons*.

Control demons are tasked with finding a target policy which is as close to being optimal as possible, while prediction demons make predictions about questions defined by the question functions about what happens if the demon follows the given target policy. As described by (Sutton et al., 2011) in the original Horde paper: "*Control demons can learn and represent how to achieve goals, whereas the knowledge in prediction demons is better thought of as declarative facts*".

Control demons learn how to represent and achieve goals by using the value function to create a good target policy π , and prediction demons use their value functions to make a *prediction policy*. A prediction policy assigns a prediction of the sum of discounted sensor values from a certain state and onwards, which represents the answer to the question asked with the question functions. Those sensor values are often represented in the transient and terminal reward functions r and z . For example, one kind of question that can be asked is "how many steps will it take from here to terminate?". Or "am I in the top left corner?" An enormous amount of different kinds of questions can be asked from prediction demons. In this way, prediction demons can use small pieces of information about the environment to make larger predictions.

Although demons often work independently of each other it is possible for demons to use information from other demons. One such way is that a prediction demon can use the target policy of a control demon. If we use the example of the demon which predicts the amount of time steps it takes to terminate while following the target policy of the demon which tries to solve the base problem, we can see that this prediction demon uses the target policy of another demon. This allows questions of the kind "*if I act like this control demon wants, what can I predict about this question?*"

It is also possible for prediction demons to use each other's GVs. This way one prediction demon can learn a concept such as "near a wall" as a maximal distance to a wall from a state, and another demon can learn *if I follow this policy, will I then be near a wall?*". It would not be possible to learn this using only one demon, because two different target policies are needed for this. The demon that learns the concept of near a wall has to use a target policy with which it walks in all directions, or else it can not know if a certain state is within a minimum distance to a wall (provided the agent does not have any long range sensors). The only way to know if a certain state is near a wall is to remember if the agent bumped against a wall once in this state after a certain amount of steps depending on the *minimal distance* and the *step size*. On the other hand, for the demon that learns if it will get near a wall while following a certain policy π , it is essential that the target policy is π , or it will predict something else. This way, by using the information learned by certain demons in other demons, Horde is able to learn things no individual demon could.

Control demons can also use the predictions from prediction demons to construct a better target policy. Since the value of a state is determined only by the features of that state, a normal reinforcement learning agent is not good in using previous information to determine its next action. If, for example, the value of an end state depends on whether a previous state is "red" or "blue", and this differs each cycle, a single demon will not be able to perform well in the environment. If a prediction demon is made to predict if this state is red or blue, and the control demon which makes the final policy uses this prediction as a feature, it will be able to perform much better.

Chapter 4

Implementation and design decisions

4.1 The environment

The environment we used is a simple Markov Decision Model (MDP) based on an example of (Russell & Norvig, 2003, Figure 21.1 on Page 765). This example is a simple grid world, which is confined by impassable walls at the sides. (see Figure 4.1). There are four types of states: empty, obstacle, positive end state and negative end state. An empty state is a state the agent can walk through but it will not terminate or yield any terminal reward. An obstacle is an object the agent will just bump up against if it tries to walk into it. A positive end state is a state which will terminate the learning cycle and yield a, usually high, positive reward. A negative end state is a state which will terminate and yields a, usually high, negative reward. The transient reward r is -0.01 . The agent has four actions from which it can choose: UP, DOWN, RIGHT and LEFT. All actions are always available to choose from, regardless of the state the agent is in.

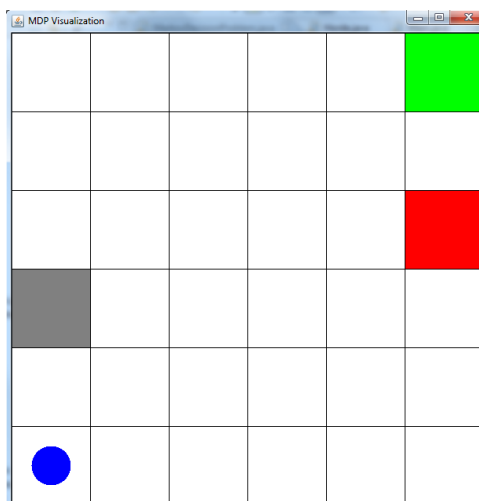


Figure 4.1: An example MDP of the implemented type. The blue circle is the agent, the grey square above the agent is an obstacle, the green square in the upper right corner is a terminal state with a positive reward, and the red square at the right side is a terminal state with a negative reward.

The main difference between this example and our own environment is that the environment we implemented is continuous instead of discrete, see Figure 4.2. In our implementation, there is no specified number of states, but the position of the agent is represented by its x and y position. The actions each

move the agent a certain predefined stepsize to their respective directions, instead of simply moving to the next discrete state. We consider this a more appropriate environment to test the Horde architecture, since it is based on making predictions and solving problems in the real world. The agent is point-sized and its position in the environment can be described by a tuple of an x position and a y position. In other words, $s_t = (x_t, y_t) \in \mathbb{R}^2$. We used a deterministic model for all tests. We did this because it is easier to predict what the outcomes of the experiments should be this way.

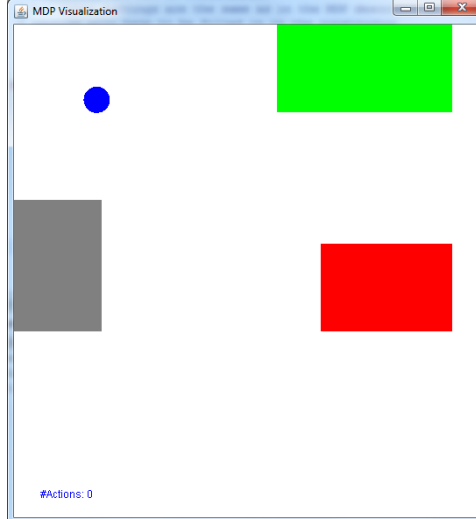


Figure 4.2: An example continuous MDP of the implemented environment. The agent’s true position is a point at the top left of the circle visualizing the agent. The rectangular regions act as terminal states and obstacles.

4.2 The features

We constructed the feature vector ϕ via a tile coding (Sutton & Barto, 1998). A tile coding is an effective way of generating features in a continuous environment. A tile coding algorithm divides the environment into several dimensions like columns, rows and diagonals, which are called *tilings*. Each of these tilings consists of a predefined number of parts, which are ordered according to their position. For example, a column tiling can consist of 5 columns, which are ordered from left to right. Each tiling yields a row of binary numbers, one for each part. Each of those numbers represents whether the agent currently occupies that part.

A visual example is shown in Figure 4.3. The field is split up in 5 columns. The position of the agent (the circle) is determined by its top left coordinate, since technically the agent is only a point and the top left coordinate is its position. Therefore, the agent is considered to be in the first column. This means the first number of the row is a 1, and the rest are zeroes. Therefore, the features of this tiling are: 10000.

One tiling only gives a little bit of information. The more tilings are used, the more potentially relevant information can be used. Normally, multiple tilings are used, and their features are concatenated into one large feature vector ϕ . If we apply a row tiling to this situation, we would get Figure 4.4. We can see the agent is considered to be in the fifth row. This makes the features of the row tiling 00001. To determine the features yielded by the column tiling and the row tiling together, we simply concatenate the features of the column tiling (10000) and the row tiling (00001). So, if we start with the column tiling (it does not matter which one goes first, as long as it is consistent), the total features of the agent’s position will be 1000000001.

In our implementation we used a column tiling, a row tiling and two diagonal tilings, one from the

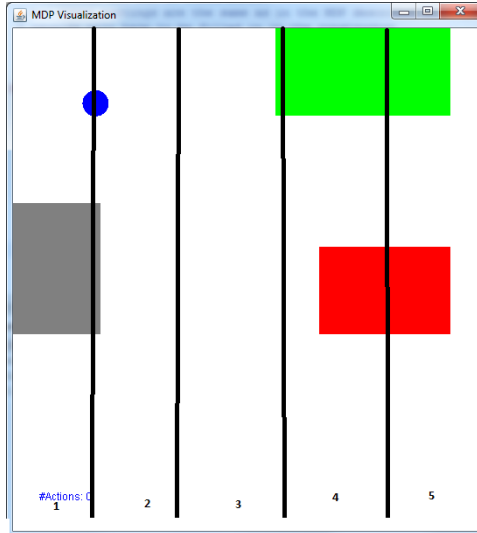


Figure 4.3: A column tiling being applied to an MDP

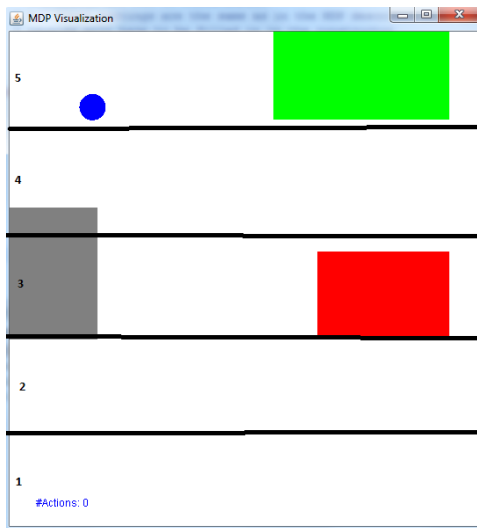


Figure 4.4: A row tiling being applied to an MDP

bottom left to top right, and one from the bottom right to the top left. The diagonal tilings divide the field in squares, and then look which squares lie on certain diagonals. The one from the bottom left to the top right is shown in Figure 4.5.

The agent, whose real position is at the top left of the drawn circle is considered to be in the fifth diagonal. The feature vector of this diagonal tiling is: 000010000. The diagonal from the bottom right to the top left works just the same, but in a different direction, see Figure 4.6

In this tiling the shown agent is considered to be in the ninth diagonal, so the features of this tiling are 000000001.

We chose to implement the diagonal tilings in this way, as opposed to simple diagonal lines over the entire MDP because the algorithm performed slightly better this way. We also used a bias, a single value which is always 1. A bias is useful because it is a feature that is independent of the x and y positions of the agent, but with a bias it is possible to learn an independent value to all states (through the weight vector θ). We represented state action combinations by making a separate state feature list for each action, and concatenating them to each

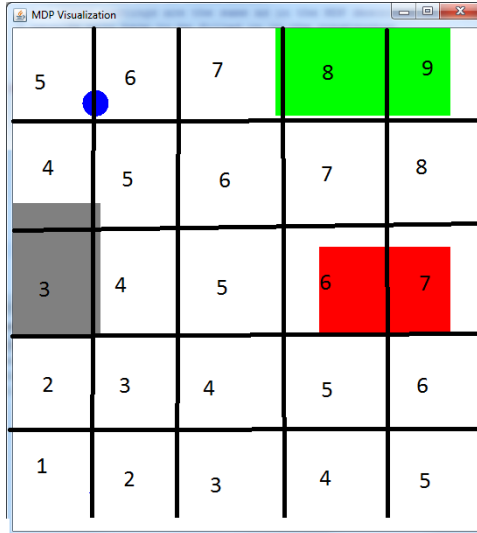


Figure 4.5: The diagonal tiling starting at the bottom left being applied to an MDP

other. Each action was given a number, and the n^{th} state feature list in the longer state action feature list, received the actual features of the current state, where n is the number of the action that was chosen. All other state feature lists were filled with zeros.

We will now come back to the simplified example of a row and a column vector of 5 parts each. This time, however, we will add a bias, and we will determine the features of all state action combinations. The state-feature list will remain the same, except that a bias will be added at the end of the list. So instead of 1000000001 the state-feature list is 10000000011. Suppose the action UP is given the number 1, DOWN is given 2, LEFT 3 and RIGHT 4. If we determine the features for the position in pictures 4.2-4.6 with the action UP, only the first state-feature list in the state-action feature list will get the actual features of the state, and the rest will be filled with zeros. This means the state-action feature lists for the different actions will be as shown in Table 4.1.

Action	Features			
UP	10000000011	00000000000	00000000000	00000000000
DOWN	00000000000	10000000011	00000000000	00000000000
LEFT	00000000000	00000000000	10000000011	00000000000
RIGHT	00000000000	00000000000	00000000000	10000000011

Table 4.1: The state-action features for the different actions

This way, the reinforcement learning algorithm can easily distinguish the state-action values of different actions in the same state. The Q-values of the different actions are fully dependent on the state the agent is in.

It is also straightforward to compute the expected next feature vector (as outlined in Section 3.4) this way. Suppose, for example, the situation that an agent reaches a certain state s and has an 80 percent chance to perform action 1 and a 20 percent chance to perform action 2 in that state. The first state-feature list in the expected next state-action feature list will be the state-feature list in state s with all numbers multiplied with 0.8, the second feature list will be the same but multiplied with 0.2 instead of 0.8, and the other feature lists will be all zeros.

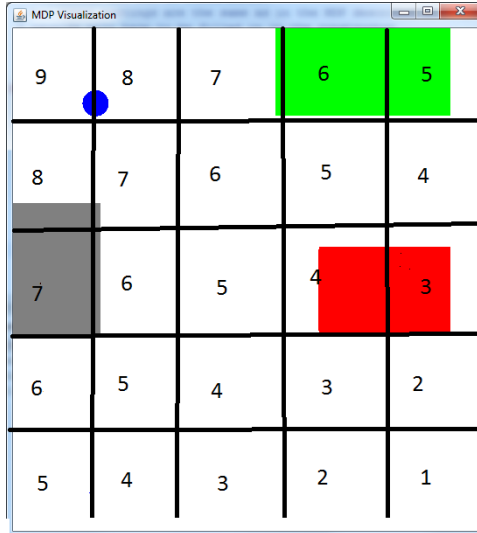


Figure 4.6: The diagonal tiling starting at the bottom right being applied to an MDP

4.3 The policies

Recall that a policy π is a mapping from a state-action combination to a value between 0 and 1, or $\pi : S \times A \rightarrow [0,1]$, where the result is the chance action a will be chosen while in state s . During exploitation, policies are based on the value functions. For each state, the action with the highest Q-value of that state gives the highest result in the policy. The formula we use for this makes use of the Boltzmann distribution:

$$\pi(s, a) = \exp(q(s, a)) / \sum_{a' \in A} (\exp(q(s, a')))$$

Note that this formula satisfies the constraint that all probabilities in a state sum to 1, or

$$\forall s \in S (\sum_{a \in A} \pi(s, a) = 1)$$

4.4 The overall architecture

All demons are constructed independently of each other, and are given their question functions upon creation. It is possible, however, to alter them during training. This way demons can use information from each other, as described in Section 3.5. Control demons are initialized without a given target policy since it is their goal to make an optimal target policy. Prediction demons are given a target policy upon creation, which is usually altered during training as to represent a target policy yielded by a control demon. This way prediction demons can answer questions of the form “if I do this, what will be the answer to that question?” The specific setups of the demons and their question functions define the problem that has to be solved. The setups for the experiments we ran will be explained in detail in Section 5.

Chapter 5

Experiments and Results

5.1 Introduction

In this chapter we will describe the experiments we have run with the Horde architecture, and their results. We used a deterministic model for all tests, because it is easier to predict the right outcomes that way. The term “base control demon” refers to the control demon which tries to solve the base problem. I use a base control demon for all tests, because prediction demons usually have to predict what happens *if* the demon follows the (near) optimal policy of the base control demon. The question functions for the base control demon are as follows:

The termination function γ of the base problem is 0 for end states (positive and negative) and 1 for all other states.

The transient reward function r is -0.01 for all transitions.

The terminal reward function z is 1 for positive end states, -1 for negative end states and 0 for all other states.

These question functions are summarized in Table 5.1

r	always -0.01
z	1 for positive end states, -1 for negative end states, 0 for all other states
γ	0 for end states, 1 for all other states

Table 5.1: The question functions for the base control demon in our experiments

Since it is a control demon, it makes its own greedy target policy π based on its value function. The discount factor is set to 1 in these experiments, which means future rewards are considered just as important as immediate rewards. This is because there are no unexpected terminations possible in the environment. Each state is either a terminal state or not. Because the path is guaranteed to continue if a non-terminal state is reached, there is not as much reason to consider possible future rewards less important. The tests were all run in a 500 by 500 world, with a stepsize of 100.

5.2 Experiment 1

The first experiment we ran was to predict the number of steps the agent has to take until termination if it follows the stochastic target policy yielded by the base control demon. The goal of this experiment was to see if the implemented architecture is able to make a simple prediction with a single prediction demon. We chose for this particular prediction because it is easy to actually simulate a great number of runs with an agent following the base control demon’s target policy. This way, we could measure

the actual average amount of steps the agent has to take until termination. This provided us with an objective way to measure the performance of the architecture. We used an MDP with an end state with a negative terminal reward at the right side of the field and about halfway to the top, an end state with a positive terminal reward at the top of the right side of the field, and an obstacle at about halfway to the top to the left side. The environment used is shown in Figure 5.1

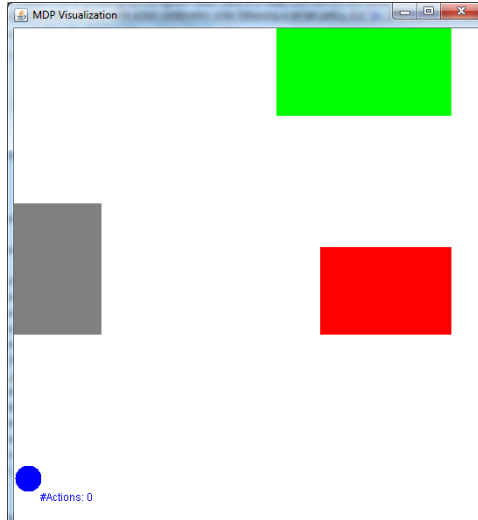


Figure 5.1: The MDP used in Experiment 1.

We ran this experiment by using two demons: a base control demon, which learned the base problem, and a prediction demon for the question asked. The question functions for the prediction demon were:

π : each step, the prediction demon copied the π from the control demon

r : always 1. Recall from Section 3.5 that a prediction policy assigns a prediction of the sum of discounted sensor values, which are often represented by the transient and terminal reward functions r and z . In this experiment, the sensor values are represented by the transient reward r . Because the discount factor is set to 1, the prediction policy will yield a prediction of the sum of the transient rewards received during a run. Because the transient reward is 1, the prediction policy will predict the amount of steps until termination.

z : always 0. For this problem it does not matter in which end state the agent comes, it only has to predict how long it takes to get there.

γ : the same as the base problem. The prediction demon terminates at the same time as the normal control demon.

These question functions are summarized in Table 5.2

π	copied from the control demon each step
r	always 1
z	always 0
γ	the same as the base problem

Table 5.2: The question functions for the prediction demon in Experiment 1.

We ran 10 trials, which each ended when either both demons had converged so much almost nothing was learned for a large sequence of actions in either demon, or when a maximum of 10,000 training cycles had been completed. Each trial ended when either the base problem had reached a terminal state, or a maximum of 100 actions had passed in that cycle. The results are an average of all 10 trials, and represent the prediction on the starting state 1,1: 30.7 steps.

We also simulated how many steps it actually took the agent to terminate from that position. We also

ran 10 trials for this, and each trial consists of 10.000 cycles of walking around following the target policy from that position. The average amount of steps we found is: 39.0. These results are summarized in Table 5.3

the prediction	30.7
the result from the actual simulation	39.0

Table 5.3: The results of Experiment 1.

The predicted result is a bit lower than the result from the actual simulation. Seeing how this is the average over 10 trials, it seems this demon underestimates the number of steps until termination a bit. As of present, we do not have an explanation for this.

5.3 Experiment 2

In the second experiment the demon has to predict whether the agent passes a certain critical area while following the stochastic target policy. The goal of this prediction is to see if the implemented architecture can make accurate predictions about the future path while following the base control demon’s policy. This experiment was run with the base control demon and a single prediction demon for the relevant question. The environment used was the same one as with the previous experiment, only with a certain “critical area” included. The prediction demon has to predict whether the agent will pass this critical area before it enters a terminal state. The agent follows a stochastic policy. The question functions for the prediction demon were:

π : each step, the prediction demon copied the π from the control demon.

r : always 0. The transient reward is irrelevant for this specific question.

z : 1 if the current position is part of the “critical” area, else 0. For this question, the expected sum of rewards is determined by the terminal reward. If a terminal reward is received during a run the sum of rewards is 1, and if not it is 0.

γ : 0 for all positions the γ of the base problem is 0, and also 0 for all x and y positions which fall within the critical area. Once the agent enters the critical area, the test run can be considered complete, since that is the very thing it set out to predict.

These question functions are summarized in Table 5.4

π	copied from the control demon each step
r	always 0
z	always 1 if current position is within critical area, 0 otherwise
γ	0 for all positions within an end state or within the critical area, 1 otherwise

Table 5.4: The question functions for the prediction demon in Experiment 2.

We ran several tests with this setup, each with a different critical area. We did this to investigate how the demon would react in different situations. We averaged over 10 trials with each situation, with the same approach to cycles and actions as in the previous experiment.

Our first test was with a critical area which covered the entire MDP, see Figure 5.2. This should be a really easy to learn problem, and we wanted to see if it could deal with this as well as it should

The expected result: 1

The actual result: 0.906

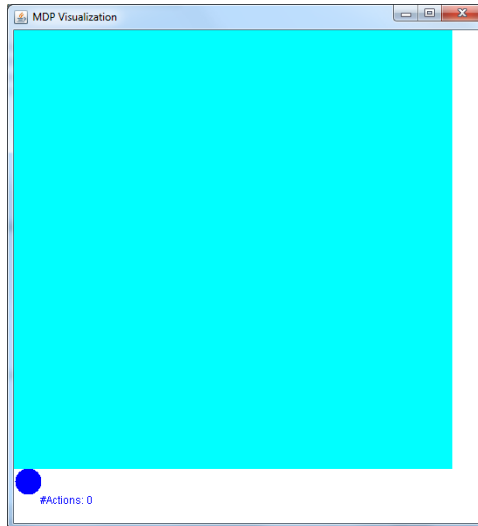


Figure 5.2: The MDP completely covered by a critical area. Note that the actual MDP is only the blue square. The position of the agent, which is its top left position, is the bottom left position of the MDP.

We had anticipated this result to be a bit higher. However, it is still fairly close to the expected result of 1.

In our second test only the area around the starting position of the agent was critical, see Figure 5.3. This should also be an easy problem.

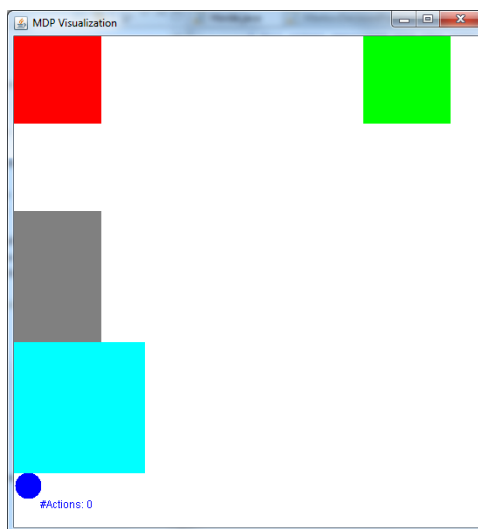


Figure 5.3: The MDP with a critical area around the agent's starting position

The expected result: 1
 The actual result: 0.957

This result is very close to the expected result, so it seems the implementation can deal with this situation.

In our third test, there is no critical area at all, see Figure 5.4. This should also be an easy problem,

and we wanted to see how well the architecture dealt with this kind of situation.

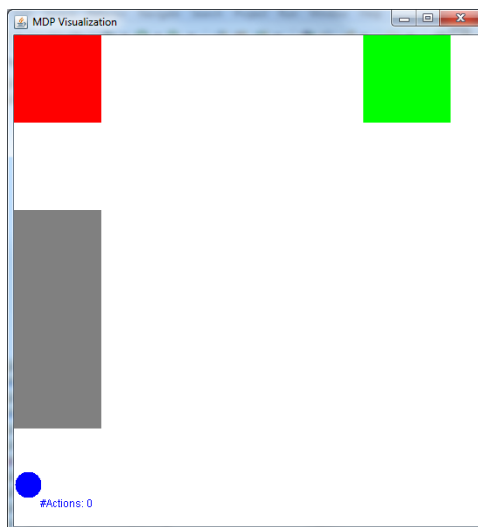


Figure 5.4: The MDP without a critical area.

The expected result: 0

The actual result: 0.00569

Once again, the result is very close to the expected result.

In our fourth test, there is a large critical area, see Figure 5.5. This area may very well be passed by the agent, but not necessarily. The prediction should probably lie somewhere between 0.8 and 1.

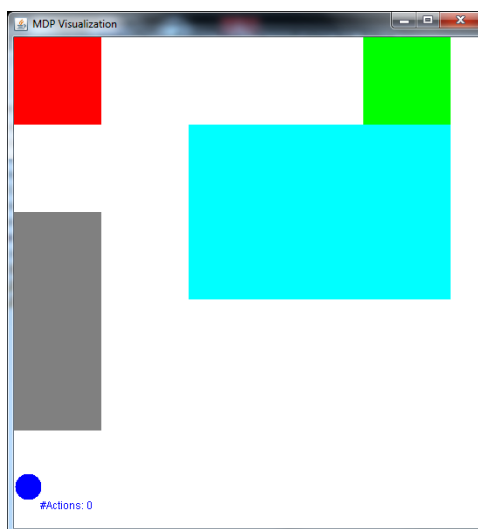


Figure 5.5: The MDP with a large critical area obviously in the optimal path.

The expected result: between 0.8 and 1

The actual result: 0.935

This prediction is a bit high, but not very much unlike our expectation. The fact it is higher than the prediction that was made when the entire MDP was covered with a critical area is strange though. We do not know how to explain this at this time.

In our fifth test, there is a small critical area, which is obviously not in the optimal path, see Figure 5.6. The prediction should be relatively low, not quite as close to 0 as without a critical area.

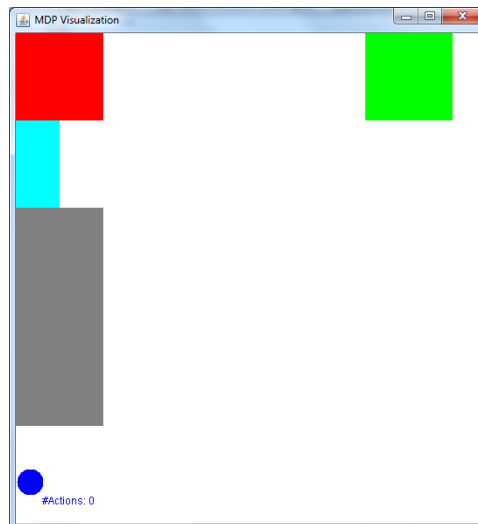


Figure 5.6: The MDP with a small critical area obviously not in the optimal path.

The expected result: relatively low
 The actual result: 0.201

This result is a bit higher than expected, but very well possible.

The results of all tests in Experiment 2 are summarized in Table 5.5

Critical Area:	Expected Result:	Actual Result:
entire environment	1	0.906
starting area	1	0.957
none	0	0.00569
large and in optimal path	between 0.8 and 1	0.934
small and not in optimal path	relatively low	0.201

Table 5.5: The results of Experiment 2

The results reflect our expectations reasonably accurate. In the test with the critical area which covered the entire MDP the result was a bit lower than expected. We not not know how to explain that at this time.

5.4 Experiment 3

In the third experiment, a prediction is made of the colour of a certain indicator area. This prediction is then used to improve the policy of the base control demon. We ran this experiment to see if we could actually use predictions to make the agent perform better in the base problem. The environment we used consists of an obstacle at the left of the field, an end state at the top left and an end state at the bottom right, and an “indicator” spot right next to the agent. The indicator can have two colours,

yellow and purple. If it is yellow, the end state at the bottom right of the field is a positive end state, and the one at the top left is a negative end state. If the indicator is purple, the end state at the bottom right is negative, and the one at the top left is positive. Each time this MDP was made (so also for each test cycle) the indicator was randomly set to either yellow or purple with equal probabilities. What sets this experiment apart from the other experiments is that the environment used here is only partially observable. This makes it very hard, if not impossible, to learn for the base control demon.

In all other experiments we tested whether a prediction could be made accurately. In this experiment, we tested whether a prediction can be used to actually improve the performance of a control demon. One difference between the tests in this experiment and those in the other experiments is that in this experiment, we tested with a deterministic policy. A deterministic policy is a policy which results in 1 for the action with the highest Q-value in the current state, and in 0 otherwise. We did this because the focus of this experiment is on whether the base control demon can use the information of a prediction demon to choose to which end state it tries to move. We wanted the resulting end state the agent moves to to be dependent only on how well it integrated the information of the prediction demon, and not on chance.

The environment of Experiment 3 is shown in Figure 5.7

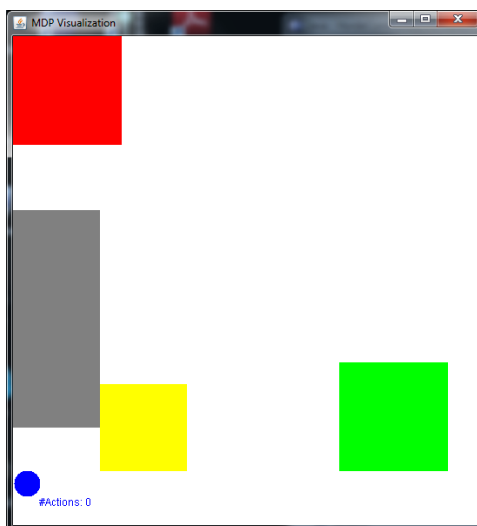


Figure 5.7: The environment used in Experiment 3. The squares in the corners are the end states, and the small square near the agent is the indicator area.

We first tested how well the normal, base control demon would do in this situation. We made a single, base control demon with the base question functions as described in the introduction of this chapter. We ran 100 trials, which each consisted of learning the problem with 10000 cycles and a test run. Each cycle had a maximum of 100 actions. We divided the possible results from the test runs in three categories: ended in positive end state, ended in negative end state, and did not reach an end state in a 100 steps. The last case means the agent kept walking against a wall or object, or kept repeating the same pattern.

The results are summarized in Table 5.6:

End State	Results:
Positive	35
Negative	28
None	37

Table 5.6: The results of the base control demon

For our second test we used two demons. The first of these was a prediction demon which has to predict

the colour of the indicator area. The second demon was a control demon which used all the base control question functions, but also used the prediction from the first demon as a feature. The question functions of the prediction demon in this test were:

π : a random target policy which is not changed during the training. Since the question asked of this demon has nothing to do with the actual behaviour of the agent, the target policy is irrelevant.

r : always 0. The transient reward is also irrelevant for this question.

z : 100 if the agent is in the indicator spot and it is yellow, -100 if the agent is in the indicator spot and it is purple. Always 0 if the agent is not in the indicator spot. The terminal reward represents the sensor values in this experiment. A high sum of rewards is a prediction the indicator spot is yellow, and a low sum of rewards is a prediction it is purple. The positive and negative terminal rewards are this high because this demon has to learn the colour of the indicator spot *for a single run*. Each cycle the colour changes, and so do the positions of the positive and negative end states. The task of this demon is to predict the colour of the indicator spot *this cycle*. This demon does not have 10000 cycles to learn like other demons, so the rewards have to be much higher.

γ : 0 if the agent is in the indicator spot, 1 otherwise. Only the indicator spot is a terminal state. It is the demon's task to predict that spot, all else is irrelevant to it.

The question functions for this prediction demon are summarized in Table 5.7

π	random
r	always 0
z	100 on yellow indicator, -100 on purple indicator, 0 otherwise
γ	0 on indicator, 1 otherwise

Table 5.7: The question functions for the prediction demon in Experiment 3.

This prediction demon only has a single feature, which is always 1. This is because the prediction of this demon should be fully independent of the current position of the agent, or of the action that is selected. Because it has only one feature, this demon also has a weight vector θ of length 1. This single number determines the prediction the demon will make, regardless of the current state or action.

As long as the agent has not yet entered the indicator spot the value function of this prediction demon gives a low negative or positive number. If the indicator state has been reached and it is yellow the demon will give a high positive number, and if the indicator has been reached and it is purple the demon will give a high negative number. Each action, a check is made to see if the result from the value function of the prediction demon is a high negative number, around 0 or a high positive number. This information is passed through to the control demon each step. The control demon uses a feature vector which consists of three different parts, one part for each possibility of the colour of the indicator (yellow, purple and unknown). This is done much in the same way as different parts of the feature vector are made for each action in our regular tile coding implementation. The total feature vector consists of three state action vectors put together. 2 of those three vectors consist of 0s and the part corresponding to the colour of the indicator as predicted by the prediction demon gets the actual state action values.

For example, if the normal state action feature vector is 00001000 (it is usually much longer), the feature vector if the indicator spot is unknown will be 00001000 00000000 00000000, the feature vector if the indicator spot is yellow will be: 00000000 00001000 00000000 and the feature vector if the indicator spot is purple will be 00000000 00000000 00001000. Each of those state action feature vectors still consists of four state feature vectors as explained in Section 4.2.

This way, the Q-values for state-action combinations are always dependent on the prediction of where the positive end state is. Each step during the training the prediction from the prediction demon is passed through to the control demon, thereby changing its feature vector. During the testing cycles, a prediction demon for the position of the positive end state was supplied. This demon learned for this

specific cycle (its weights vector was reset) where the positive end state was, and passed that information through to the control demons (which used the thetas it learned during the training). The results are shown in Table 5.8

End State	Results:
Positive	100
Negative	0
None	0

Table 5.8: The results of Experiment 3

These results clearly show the ability of different demons to use each others predictions to improve the performance of the architecture as a whole.

Chapter 6

Conclusion

The Horde architecture is a new reinforcement learning architecture based on parallel processing of sensorimotor data. By making use of General Value Functions, many very different questions can be answered at the same time. By using the answers from different sub-agents, called demons, all kinds of relational questions can be answered. By learning facts about the environment bottom up instead of using predetermined classifiers, Horde is grounded in sensorimotor data. We have reimplemented Horde using our own simulated continuous MDP environment. In this thesis we have explained the Horde architecture in detail, discussed our own implementation and vision of it, and run several tests. We have shown different demons can accurately learn completely different things in parallel. We have also shown that demons can use each others prediction to vastly improve their performance in a partially observable environment. The research questions we asked are:

- 1: What major design choices not explicitly presented in the paper Horde: A Scalable Real-time Architecture for Learning Knowledge from Unsupervised Sensorimotor Interaction (Sutton et al, 2011) have to be taken to create a working implementation of it?
- 2: Will the Horde architecture work well in a different simulation?

To which our answers are:

- 1: The first major design choice that has to be made to implement a working Horde implementation is the environment. Since the Horde architecture is so general, almost any environment will do. We chose for a continuous MDP with a point-sized agent, and the possible actions UP, DOWN, LEFT and RIGHT. This situation is very different to, for example, a situation with an agent with a specified size in both the x and the y direction and several motors which each direct its movement a bit. We chose for this environment mostly because it is very different from the example in the original Horde paper (Sutton et al, 2003). This way, we can answer the second research question (will the Horde architecture work well in a different simulation?) more convincingly. The second reason for this environment is that it is relatively simple to implement. This way we could focus more on the actual Horde architecture instead of the environment.

The second big decision we had to take was how to make the features. The paper by Sutton et al. briefly mentions they used a tile coding but nothing more specific. Using row, column and diagonal tilings is quite standard, as is using a bias. The way we integrated actions in the feature list is a bit stranger though. We made a separate state feature list for each action and concatenated them. For each action that was not chosen, the state feature list representing that action was filled with zeroes. This way, the action that is picked depends on the state the agent is in. We do by no means claim this is the only way to do this, but it is a design decision that had to be made.

The third and final big design choice we made was how to set up our experiments. The Horde architecture is only the basic framework on which all kinds of different set ups with all kinds of different demons can be made. We chose for two experiments with simple singular predictions of two very different kinds and for one experiment in which a demons prediction is used to improve the value function of another demon

(and therefore its policy). This is only a fraction of all the possibilities of the Horde architecture.

2: Although some results are a bit off, we think the results overall show the architecture is indeed viable in this simulation.

Possible future work includes experimenting with different setups of demons and with different environments. The Horde architecture is very broad. There is a great number of different possibilities for what kind of demons to make, what to predict, how to use demons to improve the predictions or behaviour of other demons etc. The Horde architecture is also very general. By learning directly from sensorimotor data, Horde is able to learn without much specific knowledge of the environment. It would be very interesting to see how well Horde performs in completely different environments. It would be especially interesting to implement Horde on actually physical robots. Being able to learn directly from a sensory stream, the Horde architecture should be especially suitable for dealing with the real world.

References

- Maei, H. R., & Sutton, R. S. (2010). GQ(λ): A general gradient algorithm for temporal difference prediction learning with eligibility traces. *Proceedings of the Third Conference on Artificial General Intelligence*, 1, 91-96.
- Russell, S., & Norvig, P. (2003). *Artificial intelligence: A modern approach* (Second ed.). Pearson Education Inc.
- Sutton, R. S. (2011). *Learning about sensorimotor data*. Retrieved from <http://webdocs.cs.ualberta.ca/~sutton/Talks/nips-granada.pdf>
- Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction* (R. S. Sutton & A. G. Barto, Eds.). MIT Press.
- Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., & White, A. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *Proc. of 10th int. conf. on autonomous agents and multiagent systems* (Vol. 1, p. 761-768).
- Weng, J. (2004). Developmental robotics: theory and experiments. *International Journal of Humanoid Robotics*, 1, 199-236.