Connecting the Demons

How connection choices of a Horde implementation affect Demon prediction capabilities.

Author:
Jasper van der Waa
jaspervanderwaa@gmail.com

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

August 23, 2013

Bachelor Thesis
Artificial Intelligence
Faculty of Social Sciences

Radboud University Nijmegen

Abstract

The reinforcement learning framework Horde, developed by Sutton et al. [9], is a network of Demons that processes sensorimotor data to general knowledge about the world. These Demons can be connected to each other and to data-streams from specific sensors. This paper will focus on how and if the capability of Demons to learn general knowledge is affected by different numbers of connections with both other Demons and sensors. Several experiments and tests where done and analyzed to map these effects and to provide insight in how these effects arose.

Keywords: Artificial Intelligence, value function approximation, temporal difference learning, reinforcement learning, predictions, prediction error, pendulum environment, parallel processing, off-policy learning, network connections, knowledge representation, Horde Architecture, GQ(λ), general value functions

Contents

Chapter 1

Introduction

Developing an entity that can learn general knowledge about the world while moving around in that same world, is a main research topic in the field of Artificial Intelligence. In 2011 Sutton and his colleagues developed the Horde Architecture with this purpose in mind [9]. This state of the art architecture is able to learn general knowledge about the world. An entity that acts through the use of an Horde Architecture is able to learn how to behave in its environment. But, more importantly, while learning this behavior it acquires more knowledge about its environment without any prior knowledge. An implementation of the Horde Architecture is able to do this because it learns knowledge in a way similar to animals; the architecture first learns new basic knowledge about the world, then it uses this knowledge to learn more complex and abstract things.

A Horde Architecture is a network which consists out of many, sometimes even thousands of nodes. Each node is called a Demon, hence the name Horde (of Demons). This network, the Horde implementation, receives a constant stream of sensorimotor data. This raw data stream consists of information that comes directly from the entity's sensors about its internal processes and its environment. While other machine learning approaches use data-mining techniques to alter or reinterpret this raw data before actual learning, Horde is able to learn from this raw data directly. It does this through its Demons; each Demon is responsible for the interpretation of its own input. The input of the Demon represents the knowledge it has access to and the interpretation of it represents a small piece of knowledge learned from the world. This way a Demon that is connected to specific sensors can learn new things about the sensorimotor data received from those sensors. But a Demon can also receive input from other Demons. Such a Demon is able to interpret the knowledge learned by others to learn more complex and abstract knowledge about the world.

To clarify, imagine an entity with a Horde implementation. This entity learns to find the treasure inside a maze filled with traps. But the sensors of the entity cannot see far into the darkness of the maze and it knows nothing about the maze except that there is a treasure somewhere. Some Demons inside the Horde network can for example learn that when the floor ends in darkness before the entity, it might stand in front of a cliff. Others may learn that when the entity falls of a cliff it can no longer move. Now with the use of this information another Demon can learn that it is bad to move forward when the entity is in front of a cliff, since getting stuck prevents it from finding the treasure. This example shows how Demons inside a Horde implementation can work together to learn complex knowledge, starting with no knowledge at all.

The fact that the Horde Architecture is able to learn complex and abstract knowledge about a world without prior knowledge makes it very general applicable and versatile. This makes a Horde Architecture a very suitable machine learning technique for environments of which little knowledge exists. Think of applying a Horde Architecture in an exploration robot for example. For this, one only needs to create a network of Demons and define what Demon is responsible for learning what potential knowledge, with no restriction to the number of Demons inside the network. After defining the network the

Horde implementation can be released at the environment; where each Demons learns new knowledge in parallel about the world and how to behave in that same world.

Related work underlines this generality of the Horde Architecture. Sutton et. al first tested the Horde Architecture in a robot to experiment if the architecture was able to predict if future sensor values became within a certain range. They also tested if the robot was capable of predicting the time needed to come to a full stop on different terrains [9]. Planting tested the capability of the Horde Architecture to solve the problem in a basic simulated MDP[6]. The Horde Architecture was also used in predicting the intended movement of a subject's arm based on sensory data from an EEG. These predictions where used to help a prosthetic arm to learn and act faster on commands received from the subject through the EEG [5].

However, a Demon needs to be connected to specific data streams from sensors, other Demons or both, as these connections define what the Demon learns about the world. If a Demon is created with the purpose to learn some important piece of knowledge, one still need to define what information the Demon needs to learn that piece of knowledge. In other words; you need to define the connections within the network. But before you can make decisions about these connections, it helps to know how these decisions might affect the capability of the Demon to learn what you want it to learn. One of these decisions you then face is; how many connections does my Demon need? This decision plays the key role in this thesis, hence the research question for this thesis is:

Does varying (the number of) connections between Demons and sensors affect the capability of the Demons learning the knowledge they should learn?

We answer this question by testing hundreds of different instantiations of the Horde Architecture. All of these instantiations differ mainly in the number of connections between Demons and between Demons and the data-streams from sensors. Every experiment has two phases. The first phase is the learning phase, in which every Demon is allowed to learn the knowledge it should provide. The second phase is the test phase, in this phase we analyze how closely the knowledge provided by the Demons is to the actual knowledge they should have learned. The experiments are all done in one specific environment and with a limited number of Demons inside each instantiation.

This thesis begins with explaining the needed background knowledge to understand Horde together with the notation used throughout this thesis. This is followed by an in depth explanation of how the Horde Architecture and its Demons actually work. Then the methods and approaches that where used are introduced, including a detailed description of the used environment. After that the experimental setup and variables are explained followed by a chapter with the collected results. We finish with the main conclusions from these results.

Chapter 2

Background

The Horde Architecture uses different machine learning techniques. Here we introduce and explain these and give an overview of the notation used in this thesis.

2.1 A Machine Learning Environment and its Agent

A machine or system that learns and acts, called an agent, does so in a certain environment [2][7]. An implementation of the Horde Architecture is such an agent. The environment of an agent includes all of its surroundings in which it can act and perceive. Part of a defined machine learning environment are two important terms; the state of the environment as perceived by the agent and the possible actions the agent can perform [8]. The state exists of a certain set of variables that define the perceived properties of the surroundings of an agent. The actions is are possible interactions with the environment the agent is capable of. These interactions change the properties of the current state and results in a new state. To summarize; a machine learning agent learns and acts inside an environment which is a defined surrounding that changes from one state into the other based upon the actions performed by the agent.

Machine learning environments also have a certain goal. Recall the example from Chapter 1 of an agent searching for treasure in a maze; the maze and the traps inside it are the surroundings of the agent, what the agent perceives inside the maze is the current state and the possible actions might include walking, turn a corner, opening a door and so on. The goal is to find the treasure. The agent needs to learn what actions to perform at what times to achieve this goal.

2.2 Reinforcement Learning

A Horde implementation is a reinforcement learning agent. reinforcement learning is an important technique of machine learning which is inspired on how biological beings learn; trying to maximize some sort of reward through trial-and-error. A reinforcement learning agent explores its surroundings to discover the actions that give the greatest rewards. It is also capable of learning what action sequences maximizes this reward in the end.

A reinforcement learning technique is not so much a definition for an agent, but more of how the interaction between environment and the agent is defined [8]. Recall from the previous section that an agent has access to some properties of the environment, known as the state. reinforcement learning is the approach of defining the environment in such a way that the agent receives a certain reward in every state. Through this reward the environment tells the agent how desirable it is to be in that state and allows the agent to learn an action sequence that maximizes the sum of all of these received rewards.

In the environment of the treasure seeker for example, we can define a slightly negative reward to states that show empty hallways. This way the agent can learn that visiting those states only is not desirable. A state that shows that the agent fell into a spiked trap can give a large negative reward, as it is very undesirable to get impaled. But in contrast, finding the treasure offers a great positive reward. Rewards like these can be presented to the agent in a simple numeral way; a high positive number represents a high reward where a negative number represents a negative reward. To learn how rewarding a certain action sequence is, a reinforcement learning agent has to learn to estimate the sum of rewards received in future states, which are determined by the action sequence. The agent can then use these expectations to alter this sequence, so that the estimated sum of rewards increases.

2.3 Notation and definitions

Here we introduce a list of the used notations and definitions to prevent any ambiguity and to give a clear overview of the used terms. The notations and definitions used in this thesis are mostly the same to those used by Sutton and his colleagues in their book [8], the paper about the Horde Architecture [9] and the paper about the used learning algorithm [3]. Any deviation from their notations is to prevent ambiguity between two or more terms.

$\mathbf{N} : \mathbf{x} \times \mathbf{y} \rightarrow \mathbf{domain}$ The notation used to indicate a function over values from certain distributions or domains into another distribution. N is the name of this distribution, x and y are the distributions from which the input values are and $domain$ denotes the domain of the outcome of N.

Goal A state in an environment which is the one the agent should learn to reach.

Episode The period from the start of the agent until its termination. Some environments have states that can cause the agent to terminate. Environments with such states are called episodical because those can have multiple episodes. Environments that have no terminal states are called non-episodic and the agent in such an environment will never terminate.

Pi With **Pi** we mean the numeral value of the mathematical variable π.

s A state as introduced in Section 2.1.

a An action as introduced in Section 2.1.

S The set of all possible states.

A The set of all possible actions.

$\phi(s)$ The feature vector that gives a representation of the state s. The vector is a list of properties or features of a state. There is a unique feature vector for every state s in S.

t The current time-step. A time-step is the time in which an agent decides what action it should take and performs that action.

s_t The state known to the agent at time-step t

a_t The action taken at time-step t

r $r : S \times A \times S \rightarrow \mathbb{R}$ is the transient reward function. This function gives a reward in every state, called the transient reward. $r(s_t, a_t, s_{t+1})$ means the reward that is received when action a_t is performed in state s_t resulting in the new state s_{t+1}. The transient reward received can be denoted r_t, that corresponds to $r(s_{t-1}, a_{t-1}, s_t)$.

z $z : S \to \mathbb{R}$ is the function that gives the reward when the agent comes in a terminal state, it is called the terminal reward function. $z(s)$ means the terminal reward received in s. The terminal reward can be denoted z_t, that corresponds $z(s_t)$.

β $\beta : S \to (0, 1]$ is the termination function. This function gives the probability that the state is a terminal state. $\beta(s)$ is the probability of the state s to be a terminal state. The termination probability can also be denoted β_t, that corresponds to $\beta(s_t)$.

γ $\gamma \in [0, 1]$ is the discount factor. The purpose of this constant value is to scale any expected future rewards. If $\gamma = 0$ it means that the agent takes no future rewards into account when estimating the sum of future rewards. If $\gamma = 1$, this means that future expected rewards are not discounted at all.

π $\pi : S \to A$ is a deterministic policy. A policy is a rule that says what action will be taken in what state. For example $\pi(s) = a$ means that in state s the action a will be performed. A policy can also be stochastic; $\pi : S \times A \to [0, 1]$. Which is a policy that gives the probability of an action to be chosen in the given state. So $\pi(s, a)$ gives a certain probability that action a is chosen in state s. The policy determines the sequence of actions and therefore also the sequence of visited states.

$\pi^\star$ The optimal policy. This is a policy that always gives the best possible action that maximizes the reward. In other words; the optimal policy gives the solution to the problem of the environment that results in the maximum sum of rewards.

R_t $R : S \times A \times S \to \mathbb{R}$ is the total reward the agent receives in a state at time-step t. This function is explained in detail in Section 2.4.

V, V^π $V : S \to \mathbb{R}$ is a state value function. The state value function is explained in depth in Section 2.4. $V^\pi(s)$ is a certain value that defines the agent's expected sum of future rewards in state s when following policy π.

Q, Q^π $Q : S \times A \to \mathbb{R}$ is a state action value function. The difference with the state value function is that the state action value function makes a distinction between the possible actions in states. The value $Q^\pi(s, a)$ is the expected sum of future rewards for the agent when action a is performed in state s and when following π.

$V^\star$ The optimal state value function. This state value function gives the maximum expected sum of future rewards for every given state.

V_t Is the approximated state value function that an agent has learned up to time-step t. The agent tries to approximate $V^\star$ with this value function. The use of this value function is explained in Section 2.4 and how the agent approximates it is explained in Section 2.6.

$Q^\star$ Is the optimal state action value function.

Q_t Is the approximated state action value function that an agent has learned up to time-set t.

2.4 State Value Functions

A state value function is a function of a state that defines how preferable it is for the agent to be in that state [8]. This preference or state value is defined as the total reward an agent expects to receive from that state on. This total reward is defined as a sum of discounted rewards received in the future. Discounting occurs so that immediate rewards weigh more then future rewards.

A state value function is denoted as V where $V^\pi(s)$ is the expected discounted sum of future rewards when following π from state s. This sum can be formally defined by the equation 2.1.

$$V^\pi(s) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid s_t = s \right\} \tag{2.1}$$

Here E_π indicates that it is an expectation and $s_t = s$ means that this expectation is given with the state s as the starting point. R_t is the total reward received at time-step t and its formal description is shown in the equation 2.2. The other variables are defined in the Notations Section 2.3.

$$R_{t+1} = R(s_t, a, s_{t+1}) = r(s_t, a, s_{t+1}) + \beta(s_{t+1}) \cdot z(s_{t+1}) \tag{2.2}$$

The total reward received at a time-step as shown in Equation 2.2 is defined by the received transient reward given by r and the received terminal reward given by z scaled by the probability that the new state s_{t+1} is a terminal state given by β. For example, if this probability is 0.75 in this state, it means that the agent terminates in 75% of the cases in this state. Therefore the agent receives on average 75% of the terminal reward.

A variation on the state value function is the state action value function defined as Q^π. The state action value $Q^\pi(s, a)$ gives the agent's preference to execute the action a in the state s.

A reinforcement learning agent has its own value function to determine the preference (the expected discounted sum of rewards) of a state given the current policy. However the agent does not know the optimal value function $V^\star$ from the start; it has to learn to approximate that function, by using a technique called function approximation that is discussed in detail in Section 2.6. This learned value function known to the agent at time-step t is denoted V_t. The closer the values from V_t become to those of $V^\star$, the more effective the changes are that the agent can make to the policy in terms of total received rewards.

Therefore, improving V_t over time through learning is a good approach of how a reinforcement learning agent can learn to maximize its received rewards. The agent can change the policy by altering the actions given in certain states in such a way that the estimated sum of rewards given by V_t increases. This eliminates the point of following the entire new policy to see if it is actually better than the old one., since the value function can already give an expectation of the total reward received at the end of the policy.

2.5 Temporal Difference Learning

Humans constantly learn to make a guess based upon other guesses and this is the key for understanding the reinforcement learning method temporal difference learning (TD learning). A TD approach finds the current state value based upon the next state value, a process called bootstrapping [8]. This is possible since the value of the current state equals the received reward in that state plus the estimated rewards of all following states [7]. The Equation 2.3 gives a formal description of how this can be achieved by using the formal description of a value function as stated by Equation 2.1.

$$\begin{aligned} V^\pi(s) &= E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid s_t = s \right\} \\ &= E_\pi \left\{ R_t + \gamma \sum_{k=1}^{\infty} \gamma^k R_{t+k} \mid s_t = s \right\} \end{aligned}$$

$$= \quad E_\pi \{R_t + \gamma V^\pi(s_{t+1}) \mid s_t = s\} \tag{2.3}$$

The TD learning algorithm updates the known value function so that the values given by this function become more accurate. The learning in a TD algorithm is based upon calculating the difference between the predicted state value as calculated by Equation 2.3 and the actual state value. This temporal difference error can be described by:

$$\delta(s_t) = R_t + \gamma V^\pi(s_{t+1}) - V^\pi(s) \tag{2.4}$$

This calculation is based on the fact that the difference between the estimated state value of the current state and the discounted estimate made in the next state should be equal to the reward received in the current state. TD learning algorithms use this error in a temporal difference update step. This step updates the way the value function forms its state value according to the TD error and some learning parameter alpha. The result is V_t. More formally:

$$V_t = V_t(s_t) + \alpha \cdot \delta(s_t) \tag{2.5}$$

A reinforcement learning agent that uses TD Learning has the advantages of reinforcement learning as described in Section 2.2. But TD learning also allows online learning: The Horde agent does not have to follow the entire policy before it can adjust a state value. This is because a TD algorithm makes use of bootstrapping and the temporal difference update step. This update step allows an algorithm to change the made expectation of a state value directly after an action.

2.6 Function Approximation and Gradient Descent Methods

In this section we will explain function approximation [8]. We also describe a group of methods called gradient descent methods that apply this technique.

The first step of function approximation is to experience exemplar values from the value function the agent wants to learn. Then the agent tries to generalize from those examples to approximate the value function behind these examples [8]. Function approximation and the methods based upon it, use supervised learning; the approximated function is adjusted based upon the difference between the values from the approximated function and the experienced values from the actual function.

Gradient descent methods apply function approximation to learn to improve V_t through TD learning. These methods use a weight vector θ_t with a fixed length that represents V^π [8]. This, in combination with the feature vector ϕ, allows V_t to be described as a linear function of θ_t and ϕ_s as shown in Equation 2.6. Here V_t is the expected scaled sum of future rewards by the approximated value function at t, ϕ_s is the feature vector of state s and θ_t is the parameter vector at time-step t, both vectors with a fixed length of n.

$$V_t(s) = {\phi_s}^\top \theta_t = \sum_{i=1}^{n} [\phi_s(i) \cdot \theta_t(i)] \tag{2.6}$$

Gradient descent methods can now use this function and the principle of TD learning to adjust the weights in θ to give a better approximation of the actual value function. They do this by changing the gradient of $V_t(s)$ in the opposite direction of the made TD error. This way V_t becomes a closer match to V^π as the error decreases.

The advantage of function approximation is that the weight vector θ and feature vector ϕ substitute all state values from V. This way only the weight vector θ has to be stored to find the state value of every state. This results in far less memory usage than when every state value has to be stored, especially when there a lot of states possible.

2.7 Off-policy Learning

Until now we discussed that an agent can learn a value function and adjust a policy according to that value function while behaving as defined by that same policy, a method called on-policy learning. There is however a second method, called off-policy learning. With this method the agent learns a value function and a policy the same way as in on-policy learning but with the difference that the agent behaves according to a different policy. In other words; an agent learns one policy, called the target policy or π_{target}, while acting according to a different policy, called the behavior policy or π_{beh}.

This is a complex form of reinforcement learning as the sequence of received rewards are not directly related to the policy and value function the agent tries to learn. An agent can only learn the value function and the target policy when the actions of π_{beh} are similar to those of π_{target} for some states or sequence of states. When this is the case, the reward R can be related to the target policy and its value function.

Therefore, if the target policy and behavior policy are completely different, they will never overlap and the agent will not be able to learn an accurate value function. But off-policy learning has some great advantages if the two policies do overlap. The first advantage is that an agent can learn to approximate a value function belonging to a different policy. The second advantage is that an agent can learn a policy while it acts according to a different policy without the need to try out the learned policy. Both of these advantages are important to the Horde Architecture as we will discuss in the next chapter.

Chapter 3

The Horde Architecture

3.1 Introduction

An implementation of the Horde Architecture [9], the Horde agent, exists of a network of agents called Demons. Every Demon inside this network is a stand-alone reinforcement learning agent with its own value function. The state values from this value function represent predictions about the environment and can be used to learn policies to achieve certain (sub)goals. These predictions and policies represent the general knowledge about the environment learned by the Horde agent. Such predictions can be seen as the answers on specific questions about the environment. Think of questions as; "*How long will it take before I hit a wall?*" or "*When will I stand still if I brake now?*". A Demon that also learns a policy, answers questions about what actions should be taken to reach a certain goal. I this case, think of questions as "*What is the best policy to follow so that I do not hit a wall?*" or the bigger question; "*What is the best policy to follow so that I reach the goal state of the environment?*". The answers to such questions, the learned policies, can be used by the Horde agent to determine its behavior inside the environment. All Demons give such answers by interpreting the data received from their connections inside the network with their value functions.

As shown in Figure 3.1 Demons can be connected to features and other Demons. A feature represents the single continuous raw data-stream from a specific sensor. Connections with features gives a Demon access to that data-stream in every state. Connections with other Demons give access to the predictions made by those Demons. These kind of connections allow a Demon to make predictions based upon the predictions from other Demons. In general; it allows the Horde agent to interpret the knowledge from Demons, their learned policies and predictions, into more complex and abstract knowledge with the use of more Demons. The Horde agent in Figure 3.1 shows a relatively small network of Demons. But a Horde agent has no restriction on the size of this network and can contain thousands of Demons that all give small bits of knowledge about the environment.

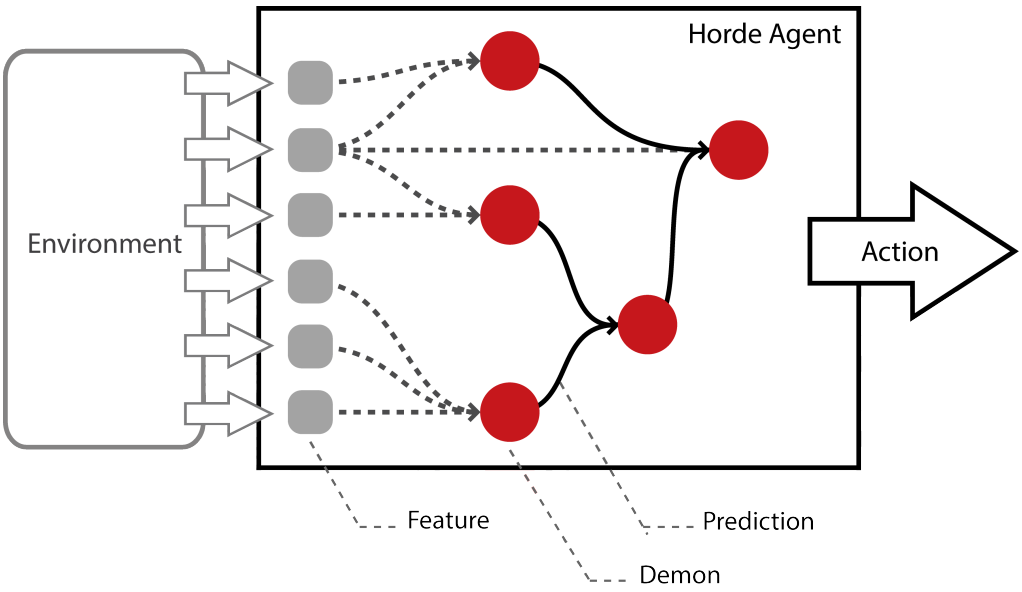


Figure 3.1: *An example of a Horde Agent. From left to right; the environment, the features representing the data-streams from different sensors, followed by the actual network of Demons and their connections and finally the the performed action given by the network.*

In Section 3.2 of this chapter we will discuss how the value functions used by Demons are different from the regular value functions explained earlier. Followed by Section 3.3 that discuss what a Demon is capable of in more detail. We conclude with Section 3.4 in which we discuss the methods and techniques that are needed to construct an actual Horde agent.

3.2 General Value Functions

In Chapter 2 we explained how value functions work. Recall that a value function gives a value based on a state or state action pair that defines some preference for an agent to be in that state or to take that action in that state. These preferences or values are the expectations of future discounted rewards the agent will receive according to some policy. Those received rewards where defined by the transient and terminal reward functions of the reinforcement learning environment such that the preferences given by the value function where related to the agent's surroundings. But it would not be beneficial to give this value function to all the Demons inside the network of a Horde agent. Because we want Demons to make different predictions about the environment, perhaps even form policies that tell how to reach certain sub-goals that are not defined by the environment. Therefore the value functions of Demons must be defined, in some way, independently from the environment.

To allow such a more general usability of value functions, Sutton et al. defined a variation on the regular value functions called a General Value Function, or GVF in short[9]. The idea behind the GVF is that all value functions depend on four parameters; the policy π, the transient and terminal reward functions r and z and finally the termination function β. The regular value function only receives one parameter as input; the policy π and the other three functions, the r, z and β, are the functions defined by the environment and do not need any variation. A GVF receives all these four parameters as input. Therefore a GVF is defined as $v^{\pi,r,z,\beta}$ and the state action GVF as $q^{\pi,r,z,\beta}$. This makes a GVF a lot more general applicable. By allowing r, z and β to be parameters of the value function as well we can define what the GVF will predict, independently from the reward functions and termination functions that belong to the actual environment.

A GVF still works the same as a regular value-function, but with three additional functions as in-

put. Let us define the state GVF $v^{\pi,r,z,\beta}$ as in Equation 3.1, with the use of the definition of a value function given by the Equations **2.1** and 2.2:

$$v^{\pi,r,z,\beta}(s,a) = E_{\pi,r,z,\beta}\left\{\sum_{k=0}^{\infty}\gamma^k\left[r(s_t,a,s_{t+1}) + \beta(s_{t+1})\cdot z(s_{t+1})_{t+k+1}\right] \mid s_t = s\right\} \tag{3.1}$$

A GVF will give a discounted sum of rewards according to the rewards provided by r and z. In Equation 3.1 we notice that these two reward functions are responsible for what this sum indicates. This sum indicates the preference of the agent to be in a state in the regular value function. However, by allowing r and z to give rewards other than those from the environment, this sum can indicate other knowledge. The termination function β as a parameter is a more substantive change, because this function has an influence on the flow of state transitions [9] as it can cause termination. We cannot allow every Demon to actually terminate the entire Horde agent if its specific termination function says so. Therefore we will simply not allow β to cause termination but still allow it to act like a termination function inside the GVF. This means that the termination function acts more as a function that defines how many states the Demon and its GVF will take into account when giving a value [4]. Lastly, the policy parameter still acts in the same way; it still defines the sequence of visited states over which the GVF should estimate the state value.

In Section 3.1 we mentioned that a value function can be seen as a question about the environment and the answer is the value it gives. To relate back to this way of thinking we can see the four parameters of a GVF as the *question functions* as they define what question is "asked" by the GVF. To clarify how the four parameters define a question, let us look at an example of a simple game. In this game the terminal rewards are $z = 1$ for winning and $z = -1$ for losing. The game has no transient rewards ($r = 0$) and β defines whether the game is won ($\beta = 1$), lost ($\beta = 1$) or should continue ($\beta = 0$). These functions are defined by the environment (the rules of the game) that an agent can use to learn how to win. But imagine that we want the agent to learn to predict how many time-steps the game will last given a certain policy. To rephrase that as a question; "How many time-steps will it take before termination occurs when following this policy?". To formulate this question in terms of the question functions, we can define them as following: $r = 1$, $z = 0$ and because we need to know when the actual agent will terminate, β is equal to that of the game itself. If we take a discount factor of $\gamma = 1$ and these reward functions, every transition means an undiscounted plus one to the total reward. This results in a GVF that counts the amount of transitions or time-steps until termination in the form of the expected sum of rewards. With the use of a function approximation technique this GVF can learn to give more accurate values, or in other words; it can learn to give more accurate knowledge about the environment.

3.3 Demons

In Section 3.1 we mentioned two different types of Demons; Demons that only make predictions about the environment and Demons that use their predictions to adjust their own policies to reach a (sub)goal. These Demons are called Prediction and Control Demons respectively. Both types have their own GVFs which they learn to adjust such that it approximates the ideal GVF more closely. But a Prediction Demon does not use its own predictions, whereas a Control Demon uses the predictions from its GVF to improve its policy. Both types also form their state representations in the same way. They use the variables defined by their input connections to form their current state.

The amount of questions that can be defined by a GVF inside a Demon are tremendous. A GVF can formulate a simple question as "*How many time-steps until termination?*" with four easily defined question functions. But a GVF can also ask questions as "*What will the activity be of this feature for a certain amount of time-steps when following this policy?*". These kind of questions are more complex, but can still be defined by letting r give rewards equal to that of the data values from the feature the

Demons wants to predict, β can be defined such that the GVF will take as many time-steps into account as desired [4]. However, even more complex questions can be defined, as we can also let the reward or value given by r, z or β depend on the value or TD error belonging to another GVF. This way a Demon can, for example, predict the activity of the predictions made by another Demon. This can be done in a similar fashion as predicting the future activity of a feature, but here r gives rewards equal to the predictions made by this other Demon. The same can be done for z and β, allowing a Demon to make complex and abstract predictions. The variations are almost endless, but not all are equally useful, important or learnable.

The answers on these described questions are predictions about the environments and can be learned and made by Prediction Demons. But Control Demons can also have the same GVFs as a Prediction Demon. A Control Demon differs from a Prediction Demon, because a Control Demon will adjust its given policy according to this prediction because it will handle the prediction as a reward that it tries to maximize. A Control Demon can therefore be used by the Horde agent to create a policy that it can follow, for example a policy aimed at avoiding obstacles. But Prediction Demons can also use these adjusted policies to make their own predictions. This way a couple of Prediction Demons can show how the future environment will look like if the Horde agent should follow that learned policy. Again, the variations are almost endless, but not all knowledge can be learned or is important for the Horde agent to know.

3.4 Methods and Techniques Needed

The Horde architecture described until now is very general and has almost no restrictions. But to allow this generality and let a Horde agent actually function, a couple of methods and techniques are needed. First of all any implementation of the Horde Architecture requires the developer to think in reinforcement learning terms. One cannot create a Horde agent without understanding how an agent learns from the reinforcement concept of rewards.

The second requirement is off-policy learning as described in Section 2.7. This is because every Demon inside a network of (potential) thousands of Demons can have its own unique policy. To make any real-time learning possible for a Horde agent all of these Demons need to learn in an off-policy manner where the policy of a Demon is the target policy and the policy the agent is following is the behavior policy [9]. This prevents the need for the Horde agent to actually perform all of its Demon's policies. However with off-policy learning there has to be some overlap between the target and behavior policy. So Demons that have a target policy with actions that are never performed by the behavior policy will not learn useful knowledge.

A Horde implementation also requires all Demons to learn through the use of a Temporal Difference learning technique. This is similar to the reasons why Demons need to learn in an off-policy manner. Because the Horde agent cannot be allowed to follow every policy to the very end to see if the given values where correct. With TD-learning this is not needed.

Lastly the GVFs of all the Demons inside a Horde architecture need to learn how to give better predictions, otherwise the Horde agent will not learn any knowledge at all. Therefore the values given by GVFs (the predictions) need to be adapted with the use of function approximation. This allows the GVFs to learn to approximate the optimal GVFs. But function approximation techniques need the use of a feature vector, therefore Demons have to form their received inputs from their connections as a feature vector.

Although these methods and techniques do exist, it is still very hard to learn in an off-policy and temporal difference manner with the use of function approximation. Therefore it is not guaranteed that a Demon can learn to answer every possible question that can be formulated, in theory, by a GVF.

Chapter 4

Methods and Approach

4.1 Introduction

In Chapter **1** we introduced the research question: *Does varying (the number of) connections between Demons and sensors affect the capability of the Demons learning the knowledge they should learn?* In this chapter we will discuss and argue the methods and general approach used to create and test the different implementations of the Horde Architecture that we used to find an answer on this research question.

First the environment is discussed, in which the implementations of the Horde Architecture, the Horde agents, will learn and act. This section is followed by a description of the behavior policy used by all the Horde agents. Next is a detailed description of what knowledge the Demons will try to learn. This is followed by a description of the measure used to determine how the capabilities of Demons to learn general knowledge are affected. After that we explain and argue the methods used to form connections between Demons. Next we give a clear overview of how our Horde agents are defined and work in general. We conclude with a short description of the learning algorithm used by every Demon.

4.2 The Pendulum Environment

The Demons in a Horde architecture will learn general knowledge based on the data-streams from a specific environment. Therefore, the environment must be carefully considered as the environment may limit or effect the outcome of the experiments. First, the environment needs a raw and constant data-stream, preferably without noise. Although a Horde agent is probably capable of handling the noise from a raw data-stream, this ensures that we can rule out noise as a potential cause of a decrease in the prediction capabilities of Demons. Secondly, the environment should allow the formation of interesting and useful GVFs. For these reasons we chose for the Pendulum Environment; an environment with two constant data streams from two environmental features, each with clear patterns.

Figure 4.1 shows an exemplar state of the Pendulum Environment. Here you can see a floating joint in the middle with a solid bar attached to it with a small weight at the end. The possible actions are different torque values that affect this small weight in either clockwise or counterclockwise direction, shown as the dotted force vector. This environment contains a non-episodic problem that an agent has to solve. The goal is to get the pendulum in an upright vertical position and keep it there by balancing it in that position while gravity affects the bar and weight. This gravitational force constantly tries to pull the pendulum in a downwards vertical position and is shown as the dashed force vector.

The pendulum has another physical property; it can build up an angular momentum. This means that the pendulum can overshoot a desired position if its momentum is large enough. A consequence of this is that when the gravitational force, the torque or both are directed in the opposite direction of

swinging pendulum, the momentum decreases. The opposite is also true; when the swing direction of the pendulum is equal to the direction of the gravitational pull, torque or both, an increase in momentum will occur.

The Pendulum Environment is not the most complex environment there is as it is non-episodical, has a limited amount of states and two environmental features. There are other machine learning algorithms than Horde that can solve the problem in this environment. But we can ask some interesting questions about this environment in the form of GVFs which are discussed in Section 4.4. This environment also allows the calculation of how accurate the learned knowledge from a Demon is, as explained in Section 4.7.

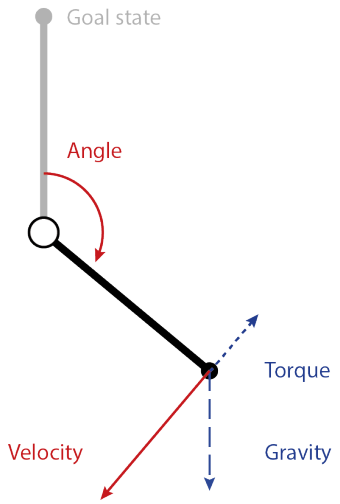


Figure 4.1: *The Pendulum environment. The environment contains a solid bar attached to a floating joint that is affected by gravitational forces (the dashed vector) and the torque (the dotted vector). The goal is to balance the pendulum in a vertical position, as shown by the light gray state. The angle of the current position of the bar relative to this goal state is one feature, the second is the velocity vector of the pendulum's tip that is the result of the torque and gravitational force.*

4.2.1 The features

The Pendulum Environment has two features that define the current state of the pendulum. The first feature is the smallest angle of the current position of the pendulum relative to the upwards goal state. The second is the velocity of the tip of the pendulum. The angle is indicated with Θ and the velocity with ν.

The angle is expressed in radians ranging from $-\mathbf{Pi}$ to $\mathbf{Pi}$. Figure 4.2 shows four possible values of this angle. The first example shows that the goal state has an angle of $\Theta = 0$. The second and third examples show that the sign of the angle defines the side the pendulum is on. In the fourth example the angle has theoretically two values, since the pendulum is at both sides with an angle of $180°$ (or $\mathbf{Pi}$ in radians). The actual value given to this state, $\mathbf{Pi}$ or $-\mathbf{Pi}$, depends on the direction of the pendulum; $\Theta = -\mathbf{Pi}$ for the counterclockwise and $\Theta = \mathbf{Pi}$ for the clockwise direction.

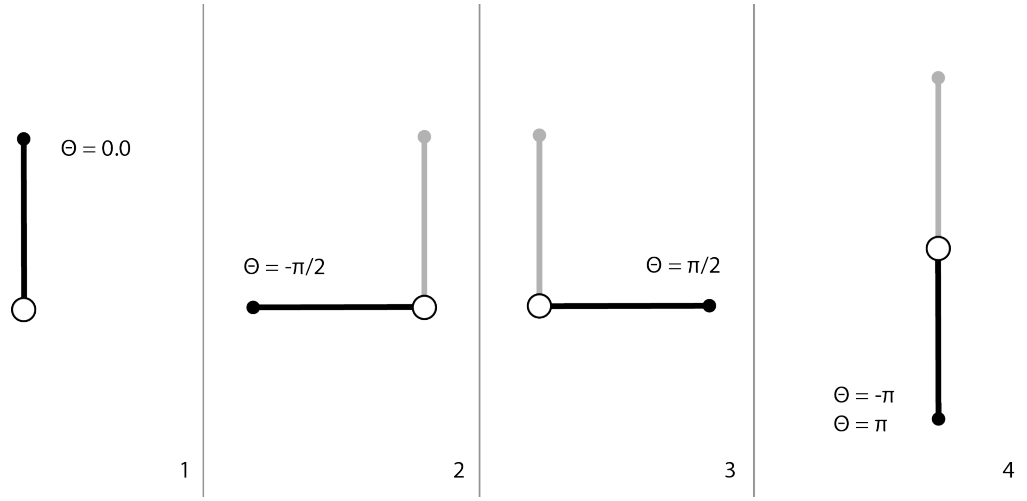


Figure 4.2: *Four possible values of the angle feature and the position of the pendulum belonging to that angle. In the vertical downward position the angle has two theoretical values (for the reason why, see text).*

The velocity of the pendulum is a feature with a complex update function. This is because of the different physical properties that define the velocity, such as the gravitational pull, the length and mass of the pendulum and the acceleration. The velocity vector depends on the combination of the torque and gravitational vector, as shown in figure 4.1. This can be done because these two forces determine the acceleration of the pendulum which can, in turn, be used to calculate the velocity. This vector is changed every update interval in terms of length (the velocity) and direction.

The two features have recognizable patterns without noise that makes it easier for demons to learn these patterns. An example of these patterns can be seen in figure 4.3. These patterns have some irregularities that might be classified as noise. But these irregularities are in fact the small influences of the chosen actions on both the velocity and angle. For example in the velocity pattern one can see that larger velocities are obtained. This increase is caused because the chosen actions help increase the velocity. A similar effect can be seen in the angle pattern; greater angles are reached over time meaning that the pendulum swings further with every swing. Both mean an increase of angular momentum and that the pendulum gets closer to its goal state instead of noise in the data-streams.

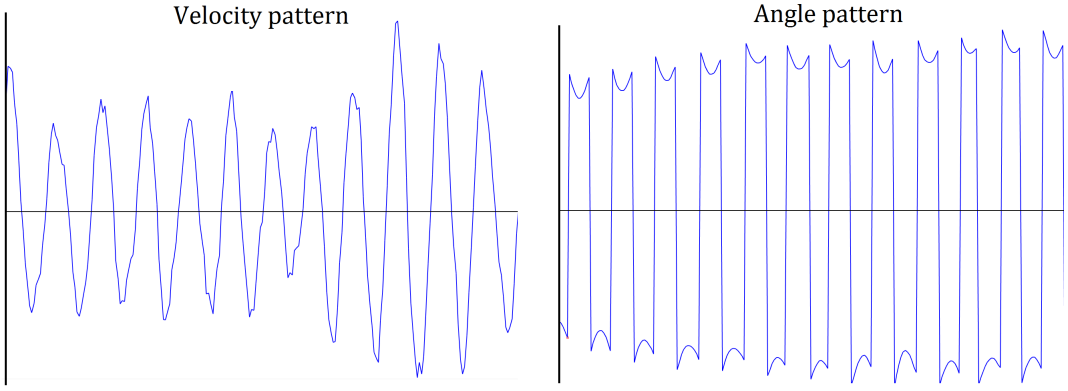


Figure 4.3: *An example of the patterns of the two features. Left the velocity and right the angle of the pendulum. The sudden changes in the angle from negative to positive (or vice versa) indicate that the pendulum switched from the left side to the right side (or from right to left).*

Both features are updated every 10 milliseconds. This means that each millisecond the state of the pendulum changes. The choice of the one millisecond interval is to make the pendulum to swing

seemingly fluent without causing a large increase of computational cost.

4.2.2 Possible Actions

The domain of all actions in this Pendulum Environment can be described as $A = [-2; 2]$. This means that there is an infinite amount of actions in A. But the learning algorithm used for the Demons needs a finite amount of actions, the reason why will be explained in Section 4.8. For this reason we limited the amount of actions for a Demon to 100 actions by generalizing the performed actions, which can still be any action from A, into one of these 100 actions. This is done by dividing the range $[-2, 2]$ into 100 sub-ranges adjacent to each other and by finding the range in which the performed action falls. Then, the generalized action is either the minimum or maximum of that range, whichever is closer. For example if the performed action is -0.11, we will generalize this to the action-0.12. Because this action falls in the sub-range $[-0.12; -0.08]$ and is closer to -0.12 than to -0.08.

As mentioned before, these actions represent the torque of the pendulum. Therefore the action is not the actual length or direction of the velocity vector as shown in figure 4.1, but is part of it. The pendulum can, for example, still move if the torque is zero due to the gravitational pull and/or angular momentum, the velocity vector merely shows the end result of these forces.

The possible actions, the torque values, allow an agent to decrease or increase the angular momentum. But these actions do not allow the pendulum to reach its goal state on its own. For this it needs to build up an angular momentum by swinging back and forth. With each swing the momentum and velocity increases and the pendulum will get higher until it reaches the goal state. At that moment the challenge is to keep the pendulum at that position and prevent it from overshooting. Due to this property the problem is challenging and makes the value function that forms the policies that can solve this problem discontinuous. Because, if the velocity gets below a certain threshold while the goal state has not yet been reached, a different sequence of actions has to be performed to acquire a new momentum.

4.2.3 Description of the value function of the problem

The solution to the balancing problem inside the Pendulum Environment can be defined as a policy learned from the optimal value function that gives the actual state action value $Q^\star(s, a)$ for every state s and possible action a. The agent that learns to solve the problem has to learn to approximate this optimal value function. The agent can then alter the policy π . This policy will be the optimal policy $\pi^\star$ as it is formed by the optimal state value function $Q^\star(s, a)$.

But before an agent can try to solve this problem an adequate reward function must be defined. Recall that the goal state is a vertical upward position of the pendulum. This means that the reward at that position should be the greatest, while the reward should be the smallest when the pendulum is as far as possible from the goal state. These rewards should be depending on either Θ or v, or both as these describe the state of the pendulum. This reward function should also give an equal reward for every mirrored state of the pendulum. For example if the pendulum has an angle of $\frac{\pi}{2}$ the received reward should be equal to the reward if the angle was $-\frac{\pi}{2}$. This way the agent will have an equal preference for the pendulum to be on either side of the goal state.
Following these four constraints, a simple reward function arises as shown in 4.1. The cosine of the angle will satisfy all constraints since it depends on the angle. This reward function will give 1.0 as a reward when the pendulum is in the goal state ($\Theta = 0$) and -1.0 as the smallest reward when the pendulum is as far as possible from the goal state ($\Theta = \pi$ or $\Theta = -\pi$). It also satisfies the fourth constraint, since $cos(x) = cos(-x)\ \forall x \in (-\pi, \pi)$.

$$r(s_t) \ = \ cos(\Theta_t) \tag{4.1}$$

Due to the fact that the problem is non-episodical it does not need a terminal reward function nor a termination function. Therefore the value or utility function of the entire problem can be described as in equation 4.2 where $r(s)$ is as described in equation 4.1. This means that the total reward of the environment can be $-\infty$ if the angle of the pendulum never exceeds $\frac{-\pi}{2}$ or falls below $\frac{\pi}{2}$ but also that the total reward can be ∞ if the angle stays between $\frac{-\pi}{2}$ and $\frac{\pi}{2}$. Therefore a machine learning algorithm that tries to solve the problems needs a kind of discounting for the rewards given by $r(s)$ otherwise state action values will reach infinite values..

$$Q^{\pi}(s,a) = E[\sum_{k=t+1}^{\infty} r(s_k)] \tag{4.2}$$

4.3 The Behavior Policy

Demons inside a Horde agent are responsible for giving general knowledge about the entire environment of the agent. In other words; Demons should be able to give adequate predictions about the environment in every state of this environment. Because if the Horde agent, including its Demons, only visits a subset of states we cannot trust the knowledge given by every Demon to be actual general knowledge about the entire environment. Therefore the Horde agent has to visit as many states as possible of the Pendulum Environment. But due to the nature of the pendulum, following a random policy will not work as the pendulum needs to have different and deliberately created angular momentums to reach all states. However, as mentioned in Section 4.2, there are several different algorithms that are capable of learning a solution for this problem. While a machine learning algorithm learns such a solution the pendulum makes multiple circles with different speeds, therefore visiting more states than it would if the agent would perform only random actions. Thus, using a policy from such a learning algorithm as the behavior policy for the Horde architecture will make sure that the architecture visits all states.

Such an algorithm can be seen as independent from the Horde agent whose only task is to improve and follow its policy according to the problem. The Horde agent can then handle the actions given by this policy as the actions from any other behavior policy. The agent that learns this behavior policy will be indicated as the Behavior Agent in this setting. The Behavior Agent acts as how a Control Demon would act inside the network of Demons, but with the difference that the Behavior Agent is not part of the network and cannot have connections with other Demons or features. The action provided by the Behavior Agent is performed after the learning process of both the Behavior and Horde agent. This way we guarantee that both agents learn from the same states and actions.

However the use of such a behavior policy means that some actions are less often performed than others. In Section **3.4** we mentioned that a GVF and its Demon can only learn adequate predictions if their is an overlap between the target policy and the behavior policy. However due to the nature of the problem inside the Pendulum Environment it is to be expected that all actions will be performed at certain times. Both large and small torque values are needed to either increase the angular momentum fast enough or to make small adjustments for balancing the pendulum in its goal state.

4.4 The Chosen GVFs

The workings of GVFs allow very different questions to be asked by Demons. If too much of these different Demons are implemented in the network of a Horde agent and all of these Demons are able to form connections, it would become very difficult to maintain an overview of the network and the function of each Demon. Furthermore, a large variation in questions increases the likelihood for a Demon to have connections with Demons that give information that is useless for that Demon in learning

how to answer its own question.

This risk needs to be minimized so that how well a Demon learns to answer its question only depends on how well it can learn from the answers on related questions (connections between Demons). To do this, the amount of different questions will be limited to a set of similar questions that are based on this basic question:

What will the activity be of this feature if the Horde agent would follow this policy, while constantly looking this far into the future?

There are still a lot of variations on this basic question but all Demons with such a question will try to learn to predict the future activity of a certain feature if the agent would follow a certain policy. But the variations are now limited to what feature and policy it would use and how far it will look into the future. Therefore Demons are more likely to connect to Demons that give useful information. This limitation on the amount of different GVFs also results in a clear overview of the networks. These questions are also not too complex to comprehend what their answers mean. This makes it easier to understand why a Demon would perform better or worse than other Demons. Finally, the answers on such questions are interesting as it can be important to know how the data-stream from a feature will react to a specific policy.

Recall from Section **3.2** that to formalize such a question into a GVF we need to define the four question functions. The basic question shown above exists out of three parts and each part allows limited variations of one of the four question functions. Here you can see these parts and the question function that formalizes that part:

What will the activity be of this feature $\Longrightarrow r$

if the agent would follow this policy $\Longrightarrow \pi_{target}$

while constantly looking this far into the future? $\Longrightarrow \beta$

The reward function r defines what the GVF will predict, so the rewards given by r will need to represent the data from the feature that belongs to that specific question. With the Pendulum Environment we have two features; Θ and v. The two reward functions for these features can be defined as $r(s_t) = \bar{\Theta}_t$ or $r(s_t) = \bar{v}_t$, where $\bar{\Theta}_t$ and $\bar{v}_t$ indicate the normalized angle and velocity, respectively, on time-step t. These reward functions ensure that GVFs with these functions will predict the activity of the feature stated by their question. The terminal reward function z is defined as $z(s) = 0$ for all states as these questions do not define any terminal reward. The third question function, the termination function β, will determine how far the Demon will look into the future when predicting a value. The question states that this probability is a constant. Therefore β will give the same value in every state and can be defined as $\beta(s) = c$ in all states, where c is some predefined constant from the range $[0, 1)$. In other words, the Demon will look a constant amount of time-steps ahead when predicting the activity of the feature [4]. The policy π_{target} is also a predefined constant that can be the same as the behavior policy or any other policy. These other policies are restricted to policies that randomly choose actions with equal probabilities out of all possible actions or a subset of these actions. How these limited variations of the question functions are used to form a GVF will be explained in depth in the next chapter.

This basic question is closely related to the questions the (nexting [4]) Demons answered in the Horde agent developed by Sutton et. al in their experiments with the Critterbot [4]. In addition to these questions, they also used Demons that predicted if a certain feature value got within a specified range or not. Predicting continuous values seems more difficult than predicting bits. But it is unlikely that a Demon will not be able to learn to predict such continuous values since the environmental features of the Pendulum Environment are without noise and contain patterns closely related to the performed actions.

4.5 The Connections

To determine if the prediction capabilities of Demons are affected by the number of connections they receive as input, we first need to create these connections. The connections between Demons are created randomly and the connections between Demons and environmental features are predefined in each experiment. Creating the connections between Demons randomly can be seen as detrimental because it bears the risk of creating a useless connection such as stated in Section 4.4. But because all Demons try to learn similar knowledge about the environment, as discussed in that same section, this risk is minimized. An advantage of creating random connections, instead of predefining them, is that it allows a greater variation in different Horde Networks with the same amount of Demons and number of connections. Another advantage is that randomization has no bias towards creating connections that seem to be useless but may in fact be very useful.

These random connections are generated according to a few constraints. The first constraint is that Demons are not allowed to form connections with themselves. The second constraint is that Demons are not allowed to connect with other Demons in circles. An example of such a circular connection is shown in Figure 4.4. Such connections are not desired because they create mutual dependencies. In this example Demon A needs the prediction made by Demon C to make its own prediction. But Demon C needs the prediction of Demon B that in turn needs the prediction made by Demon A. A similar situation occurs when Demons are connected to themselves; to make their prediction they need their own prediction.

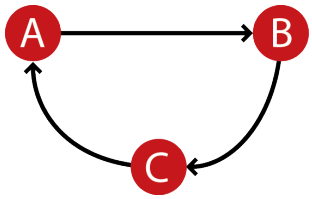


Figure 4.4: *An example of circular connections between Demons.*

To maintain these two important constraints all Demons are randomly distributed over a certain amount of layers, followed by the formation of random connections between Demons based upon these layers. Both constraints are met if a Demon can only receive input from a Demon in a previous layer, because a single Demon cannot be in two layers at the same time and if all Demons are only connected to Demons in previous layers this will prevent any circular connections.

Figure 4.5 shows an example of a Horde Network where all the Demons are connected according to their distribution over layers. This network has a total of seven Demons, two features, three layers and one feature layer. Every Demon in this network is allowed to have two input connections with other Demons. With the input connections of a Demon we mean the act of forwarding the prediction from a different Demon to that Demon. In Figure 4.5 these input connections are indicated by arrows, where "*a*" is an example of an input connection of Demon 5 with Demon 1. Noticeable is that all Demons except for Demon 1 through 3 are connected to two other Demons, because these Demons are in the first layer with no previous layers with Demons they can connect to. Another noticeable fact is that Demons are not restricted to have input connections with Demons only in the previous layer; input connections can be randomly formed with all the Demons in all previous layers.

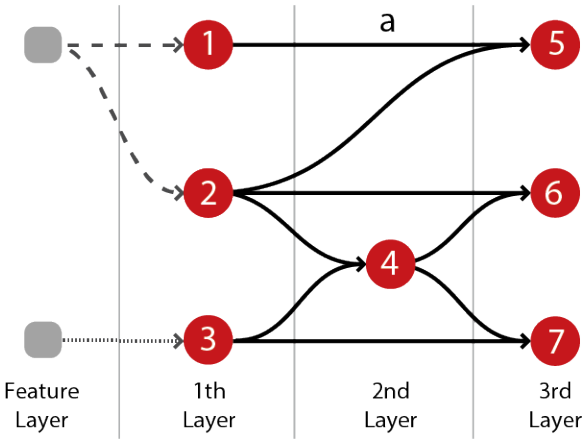


Figure 4.5: *An example of how Demons can be distributed in layers inside the network of a Horde agent. The network contains seven Demons, two features and four layers including the layer with the environmental features. Demons in all layers except for the first layer have two input connections.*

The distribution of Demons over all layers is done randomly, even the number of layers is chosen randomly. This is done with a few constraints. The constraint on determining the number of layers is that there should be at least two layers if Demons are allowed to connect with each other. When this is not allowed the number of layers is always one, as there is no reason to have more than one layer in this case. The first constraint of randomly distributing the Demons over all layers is that the first layer should at least contain an equal amount of Demons as the maximum allowed number of connections between Demons. To clarify, in Figure 4.5 this means that the first layer should at least contain two Demons since Demons have a maximum of two input connections. This is done to make sure that the Demons in the second layer can have the maximum specified number of input connections. The second constraint is that every layer should at least contain one Demon with a maximum of all the Demons minus the minimum amount needed to fill the first layer. These constraints makes sure that Demons do not form mutual dependencies and that all Demons, except for those in the first layer, can have the maximum of allowed input connections.

A result of these constraints is that if the number of allowed connections between Demons increases, the average number of Demons inside the first layer also increases. This results in a lower average of Demons that can actually form that number of connections. This is something that has to be taken into account when setting the maximum of allowed connections between Demons.

4.6 An Overview

Until now we discussed the different methods used to create and define the environment of the Horde agents and their behavior, GVFs and connections. This section will give a clear and general overview how these different methods form the entire reinforcement learning agent and environment interaction. Figure 4.6 shows this overview with an exemplar Horde agent and network.
Figure 4.6 shows on the far left the Pendulum Environment with the pendulum and its angle Θ and velocity v, These two variables represent two continuous data-streams that determine the value of their representations in the Horde network for each environmental state; the angle feature node and the velocity feature node. The two data-streams also act as the input of the Behavior Agent whose task is to learn a policy that solves the problem inside the Pendulum Environment. To ensure that both the Horde agent and the Behavior agent learn in every state from the same action, this action a is determined independent from both agents and is only executed when both agent are done learning from the current state. This action a from π_{beh} is also given to the Horde agent as the action from the behavior policy for off-policy learning.

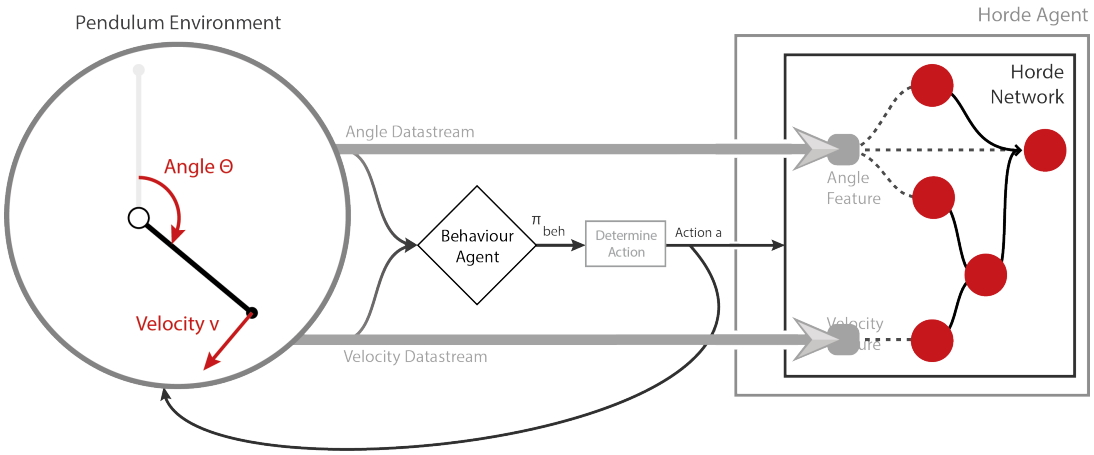


Figure 4.6: *An overview of the flow of data, knowledge and actions between the environment, the Behavior Agent and the Horde agent as an example of an implementation of the Horde Architecture.*

Before the Demons inside the Horde agent start to learn, all Demons make their predictions based upon this current environmental state. The Demons inside the first layer predict first, followed by those in the second layer and so on, until all Demons have made their predictions. All of these predictions and the two feature values form a list of values. When this list is complete the Demons begin to learn in parallel where every Demon access the feature values and/or predictions inside the list according to their connections inside the network. For example, if a Demon must make and learn its prediction based upon the prediction of one other Demon and one feature it will access those two values inside the list and forms its own state with them.

4.7 Assessing the Prediction Capabilities of a Demon

The research question requires a measure that determines how well the knowledge learned by a Demon matches the knowledge it should have learned. This measure needs to be suitable to compare Demons with each other based on how well they learned such knowledge. Therefore we chose this measurement to be the difference between the learned prediction provided by the Demon, the learned knowledge, and what this prediction should have been, the actual knowledge.

This difference is the prediction error made by a Demon and is indicated by $\Delta_D(s_t, a_t)$ where D is a specific Demon and s and a are the state and action in which the error was made. The Equation 4.3 gives a formal description of this error.

$$\Delta_D(s_t, a_t) = Pr(D) - Obs^D(s_t, a_t) \tag{4.3}$$

In this equation $Pr(D)$ means the prediction made by the Demon D and $Obs^D(s_t, a_t)$ is the actual observed value that the Demon D should have predicted in state s_t. The prediction made by the Demon is the value given by the GVF as defined by the Equation **3.1** and can therefore be described as following:

$$Pr(D) = q^{\pi,r,z,\beta}(s,a) \; = E_{\pi,r,z,\beta}\left\{\sum_{k=0}^{\infty}\gamma^k\left[r(s_t,a,s_{t+1}) + \beta(s_{t+1})\cdot z(s_{t+1})_{t+k+1}\right] \mid s_t = s\right\} \tag{4.4}$$

The formal definition of the actual observed value $Obs^D(s,a)$ is similar to that of $Pr(D)$. The difference is that $Obs^D(s_t, a_t)$ is not an expectation of the scaled sum of received rewards, but the actual

received scaled sum. This definition is given by Equation 4.5, in this equation the sub-scripted D of the question functions means that these are values the question functions from Demon D would give in that state.

$$Obs^D(s_t, a_t) = \sum_{k=0}^{n} \gamma^k \left[r_D(s_t, a, s_{t+1}) + \beta_D(s_{t+1}) \cdot z_D(s_{t+1})_{t+k+1} \right] \tag{4.5}$$

Another difference between the definition of $Obs^D(s_t, a_t)$ and $Pr(D)$ is that the observed value does not take all (infinite) future states into account; it only looks as far as n states. This is desired because to calculate the exact sum of scaled rewards these future states need to be visited to observe the actual received rewards. This calculation of n is shown in Equation 4.6 and is based on the logarithmic relation between the discount factor γ and the desired precision of this sum. The smaller the precision variable, the larger n will be and the more precise the prediction error gets.

$$n = \frac{log(precision)}{log(\gamma)} \tag{4.6}$$

The prediction error is a suitable measure to evaluate the accuracy of the knowledge a Demon learned because it shows how far off the made prediction was to what it supposed to be. In terms of GVFs; the prediction error $\Delta_D(s_t, a_t)$ of Demon D in state s_t and performed action a_t is equal to the state action value $q^{\pi,r,z,\beta}(s_t, a_t)$ minus the state action value given by the optimal GVF $q^\star$. The observed value is equal to the value given by the optimal GVF $q^\star$ because both give the actual sum of scaled rewards.

By calculating the prediction error for every Demon in each Horde agent we can determine how the Demons inside the network of such an agent perform on average in learning general knowledge. With this average prediction error we can compare how well Horde agents with different numbers of connections learn general knowledge about the environment.

4.8 The Learning Algorithm GQ(λ)

Demons use the $GQ(\lambda)$ algorithm to learn to approximate their GVFs. This algorithm is developed by Sutton et. al and is ideal for learning how to approximate GVFs [3]. This is because $GQ(\lambda)$ meets all the requirements to use it as a learning algorithm for Demons; it uses temporal difference learning, feature vectors and it learns in an off policy manner. Sutton et. al used this algorithm in their own implementations of the Horde Architecture because of its generality and its effective off-policy learning [9]. For the same reasons this algorithm is used in our Demons.

This section will not explain the explicit workings of $GQ(\lambda)$ as this is explained in detail in the paper about $GQ(\lambda)$ [3]. Instead, this section provides a general description of the algorithm and how it functions within our Horde agents.

The first step of the algorithm is the calculation of the average expected next feature vector, indicated by $\bar{\phi}_{t+1}$. This is done by summing the multiplication of the probability of each action a to be executed in state s_t by the feature vector that defines s_t. This is why the infinite number of actions possible in the Pendulum Environment are generalized to a finite number. Formally defined by the following equation:

$$\bar{\phi}_{t+1} = \sum_a \pi(a, s_{t+1}) \phi(a, s_{t+1}) \tag{4.7}$$

The feature vector $\phi(a, s_t)$ indicates the state action feature vector that works the same as a regular feature vector $\phi(s_t)$, with the difference that the state action feature vector makes the distinction between the feature values that are caused by a certain action. The combination of these state action

feature vectors form the entire feature vector $\phi(s_t)$. How this is done is explained in detail in Section 4.2 of the thesis from Planting [6].

The next step in the algorithm is calculating the temporal difference error between the current and the next state:

$$\delta_t = r_{t+1} + \beta_{t+1} z_{t+1} + (1 - \beta_{t+1}) \theta_t^\top \bar{\phi}_{t+1} - \theta_t^\top \phi_t \tag{4.8}$$

Noticeable in this equation is that there is no discount factor γ. This is because the discounting can be done with through the choice of the termination function β as it can discount both the terminal reward and future estimated rewards. Therefore the termination function can be seen as the function that determines how far a Demon will look into the future when learning the scaled sum of rewards. For example; if $\forall s \in S,\ \beta(s) = 1$, this means that in Equation 4.8 the terminal reward is not discounted and the future rewards are nullified, resulting in a Demon that only takes the current received rewards into account and that ignores all other future rewards. This is on a par with the meaning of β in general; it gives the probability that the agent terminates for the given state.

The final step of the algorithm is the updating of its three weight vectors, θ, w and e. The weight vector θ is the final product of the algorithm and is the same weight vector as discussed in Section 2.6. The weight vector w acts as the update vector for θ and the vector e is for the eligibility traces. The use of eligibility traces is an extension on the calculation of the temporal difference error. These traces allow a temporal difference algorithm to not only update the value of states according to the value of the next state, but according to the following n states where n depends on the lambda parameter [6]. More information about the workings and theory of eligibility traces can be found in Chapter 7 of [8].

Chapter 5

Experiments and Settings

5.1 Introduction

The previous chapter explained how the prediction error can be used to determine how well a Demon has learned general knowledge. In this chapter we will discuss how this prediction error is used to compare how well different Horde agents with different connection properties are able to learn general knowledge.

In total a number of seven different experiments were done. Every experiment exists out of several rounds each with twenty Horde agents that are all created according to the experimental parameters specific to that round and experiment. An overview of the structure of each experiment is shown in Table 5.1.

Experiment	Number of Rounds	NR. of Horde agents per round	Training Time-steps	Demons	Demon tests	Time-steps Demon tests
1	12	20	150 000	2	64	100
2	12	20	150 000	2	64	100
3	7	20	150 000	16	64	100
4	6	20	150 000	16	64	100
5	7	20	150 000	16	64	100
6	6	20	150 000	16	64	100
7	7	20	150 000	16	64	100

Table 5.1: *This table gives an overview of the general setup of each experiment.*

This table shows for each experiment the amount of rounds and the number of different Horde agents created, trained and tested in each round. Every Horde agent in every round will have new random connections between Demons (if allowed). The next column gives the total amount of time-steps during training, after this amount of time-steps the Behavior Agent solved the balancing problem in the Pendulum Environment. The table also shows the amount of different Demons used in each Horde agent. After this training phase of the Horde agent and the Behavior Agent comes the test phase. In this phase every Demon of the Horde agent will be tested separately in a freshly created environment in which the behavior policy is equal to the target policy of the Demon being tested. This is the only way to observe the actual received rewards with that target policy needed to calculate the prediction error of a Demon. The last two columns of the table show the number of times a Demon is tested and the time-steps during such a test. When a new Demon test begins, the pendulum inside the Pendulum Environment will be set at a new random position.

The difference between experiments three to seven lies in the properties of the Horde networks used. The rounds within each of these experiments have the same parameters with the difference in the

number of allowed random connections between Demons. The Demons of the Horde networks in the first two experiments do not have any connections with other Demons. The rounds in these two experiments differ in the Demons they have; each round has two or four Demons with different questions they try to answer about the environment. In Section 5.2 we discuss the parameters on which the different experiments are based and in Section 5.3 the different experiments will be explained in more detail.

We will not compare the prediction error of every Demon in every Horde agent with those of the other Horde agents, but instead we will calculate the average made prediction error of every round. This gives us a measurement that we use to compare the different rounds with each other within and between every experiment. We will be using three more measurements based upon the prediction error to help indicate the overall prediction capability of all Demons inside a round. The first two are the number of Demons that make extreme errors and the number of Demons that get unstable. A Demon makes extreme errors if its average prediction error is ten times larger than the value it should have predicted, for reference these Demons are called Extreme Demons. A Demon is classified as an Unstable Demon if it did not converge into giving some sort of prediction, such Demons will have an infinite prediction error. The third measurement is the average prediction errors without the Extreme and Unstable Demons.

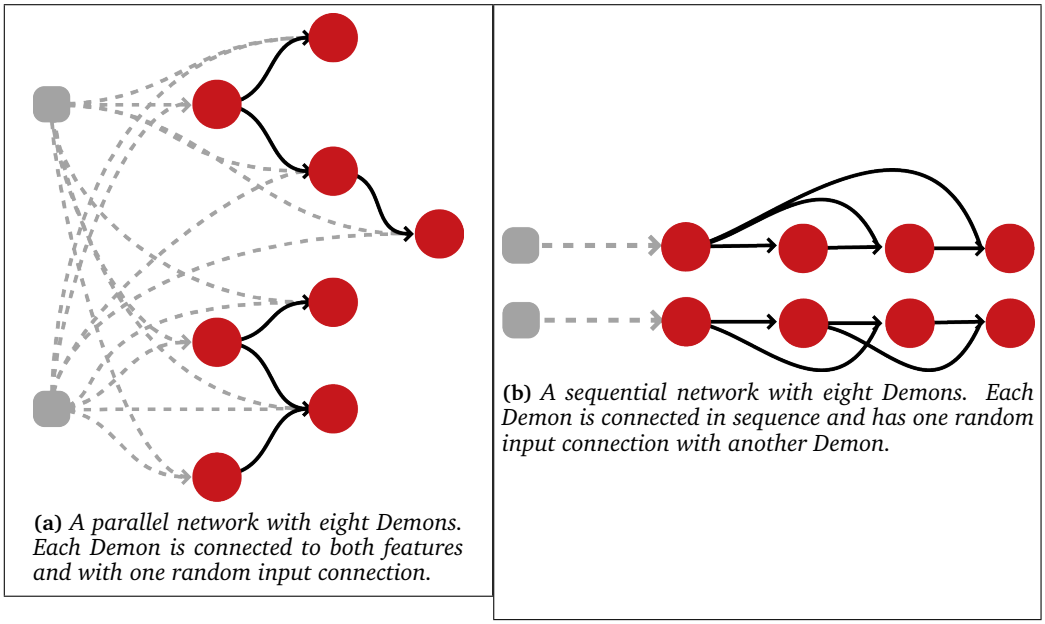
5.2 Parameters

The parameters used in the different rounds and experiments can be divided into three groups; the parameters that change between experiments and rounds and that define the network properties of a Horde agent, the specific question functions of the Demons and finally the constant parameters for every experiment.

5.2.1 Horde Network Parameters

The variation between rounds and (mostly) the experiments are based on four parameters:

- **Number of feature connections**. A Demon can be connected to the feature of which it tries to predict the future activity (one feature connection) or to both features of the Pendulum Environment (two feature connections).
- **Type of connections between Demons.** This parameter sets the type of connections between Demons formed at the creation of a Horde agent. The first type of connections, referred to as *biased connections*, force a Horde agent to only create connections between Demons that predict the activity of the same feature. The other type, referred to as *unbiased connections*, indicate that connections between all available Demons are possible.
- **Number of random connections between Demons.** This parameter states how many random input connections a Demon is allowed to have from other Demons. If this is zero there is no type of connections defined.
- **The network type.** This parameter influences how Demons are connected to each other and the features. Their are two types; the parallel network and the sequential network. An example of a parallel network is shown in Figure 5.1a and an example of a sequential network in Figure 5.1b. The parallel network is a network with no further constraints on the possible connections others than those discussed in Section 4.5. The sequential network however has three additional constraints; only one Demon per layer is allowed, all Demons are connected to the Demon in the previous layer except for the first Demon which is connected to a feature, all Demons in a sequence predict the activity of the same feature and the first Demon is connected to this feature. This sequential network is to find out how well a Demon is capable of learning its own prediction based upon the prediction from another Demon which is based on another prediction, and so on.



(a) *A parallel network with eight Demons. Each Demon is connected to both features and with one random input connection.*

(b) *A sequential network with eight Demons. Each Demon is connected in sequence and has one random input connection with another Demon.*

Figure 5.1: *An example of a parallel and sequential network*

5.2.2 Question Functions

Tables 5.2 and 5.3 show an overview of the question functions for every used Demon. The sixteen Demons in Table 5.2 are used in all experiments the sixteen Demons in Table 5.3 are only used in the first and second experiment. The second column gives the target policy of every Demon. The target policy is referred to as the range of the uniform distribution. The target policy randomly selects an action out according to this uniform distribution. Table 5.3 also shows eight Demons that use the behavior policy as their target policy. The second column refers to what normalized feature value is given at each time-step as the transient reward for that Demon, in other words; it defines what feature activity it predicts.. The terminal reward is zero for every state and the termination probability is a constant.

To clarify, let us look at *Demon 1*. The target policy for this Demon is randomly picking a torque value in a counterclockwise direction, its transient reward is the normalized angle in that time-step. The terminal reward is zero as explained in Section 4.4 and the termination probability is one for every state, which means that the Demon will try to predict only the current angle. To formalize this into the question of the GVF : "What will the angle be at this time-step if the agent should only pick random torque values between -2.0 and 0.0 according to a uniform distribution?".

Demon	π_{target}	r_t	z_t	β_t
1	$[-2, 0]$	$\bar{\Theta}_t$	0.0	1.0
2	$[0, 2]$	$\bar{\Theta}_t$	0.0	1.0
3	$[-2, 0]$	$\bar{\Theta}_t$	0.0	0.8
4	$[0, 2]$	$\bar{\Theta}_t$	0.0	0.8
5	$[-2, 0]$	$\bar{\Theta}_t$	0.0	0.6
6	$[0, 2]$	$\bar{\Theta}_t$	0.0	0.6
7	$[-2, 0]$	$\bar{\Theta}_t$	0.0	0.4
8	$[0, 2]$	$\bar{\Theta}_t$	0.0	0.4
9	$[-2, 0]$	$\bar{v}_t$	0.0	1.0
10	$[0, 2]$	$\bar{v}_t$	0.0	1.0
11	$[-2, 0]$	$\bar{v}_t$	0.0	0.8
12	$[0, 2]$	$\bar{v}_t$	0.0	0.8
13	$[-2, 0]$	$\bar{v}_t$	0.0	0.6
14	$[0, 2]$	$\bar{v}_t$	0.0	0.6
15	$[-2, 0]$	$\bar{v}_t$	0.0	0.4
16	$[0, 2]$	$\bar{v}_t$	0.0	0.4

Table 5.2: *An overview of the question functions of every Demon used in all experiments.*

Demon	π_{target}	r_t	z_t	β_t
17	π_{beh}	$\bar{\Theta}_t$	0.0	1.0
18	π_{beh}	$\bar{\Theta}_t$	0.0	0.8
19	π_{beh}	$\bar{\Theta}_t$	0.0	0.6
20	π_{beh}	$\bar{\Theta}_t$	0.0	0.4
21	π_{beh}	$\bar{v}_t$	0.0	1.0
22	π_{beh}	$\bar{v}_t$	0.0	0.8
23	π_{beh}	$\bar{v}_t$	0.0	0.6
24	π_{beh}	$\bar{v}_t$	0.0	0.4
25	$[-2, 2]$	$\bar{\Theta}_t$	0.0	1.0
26	$[-2, 2]$	$\bar{\Theta}_t$	0.0	0.8
27	$[-2, 2]$	$\bar{\Theta}_t$	0.0	0.6
28	$[-2, 2]$	$\bar{\Theta}_t$	0.0	0.4
29	$[-2, 2]$	$\bar{v}_t$	0.0	1.0
30	$[-2, 2]$	$\bar{v}_t$	0.0	0.8
31	$[-2, 2]$	$\bar{v}_t$	0.0	0.6
32	$[-2, 2]$	$\bar{v}_t$	0.0	0.4

Table 5.3: *An overview of the question functions of the additional Demons used in the first and second experiment.*

5.2.3 Constant Parameters

The learning parameters for the Demons were kept constant throughout every experiment to enable stable learning of the entire network[4]. An overview of these parameters including the precision used to calculate the prediction error is in Table 5.4.

Learning parameters	
α_θ	0.1
α_w	0.001
λ	0.7
$Precision$	0.01

Table 5.4: *Constant parameters used by all Demons.*

5.3 The Experiments

The seven experiments all test the average prediction error made in every round and the number of Extreme and Unstable Demons. The aim of these experiments is to find the effect of different network properties (number connections, type of connections and type of network) on the average prediction error on these networks caused by their Demons.

The experiments will variate in the number of connections between Demons and the two features and in the number of connections a Demon can have with other Demons. Within these connections a distinction is made between biased and unbiased connections as described in Section 5.2.1. Combinations of these variations will be tested in the different experiments.

These experiments combined will give an overview of the effects of different numbers of connections on the capability of a Demon to learn the general knowledge it should learn.

5.3.1 First and Second Experiment

The main aim of the first and second experiment was to see how the prediction capabilities of the Demons are affected by the made choices regarding the question functions (transient and terminal reward functions, target policy and termination function) as shown in Table 5.2. This is done to see how specific Demons are capable of learning their knowledge. The data from these two experiments can give an insight in how Demons with different termination probabilities and target policies relate to each other in terms of their prediction error. The difference between these two experiments is that the first experiment tests these effects when Demons are connected to one feature. The second experiment tests the same but with Demons connected to both features.

The first four rounds of both experiments look at the effect of an increase in the termination probability of the Demons when their target policy is equal to the behavior policy from the Behavior Agent. The next four rounds also look at the effect of this increase but now their target policy picks random actions from uniform distributions, all with the range $[-2, 2]$. The last four rounds each contain two Demons per feature, each with a target policy that picks random actions from uniform distributions with either the range $[-2, 0]$ or $[0, 2]$. These four rounds also test the effect of the increase of the termination probability of the four Demons in each round.

An overview of the parameters used in the first and second experiment are shown in the Tables 5.5 and 5.6, respectively. These tables also show what Demons are used in every round, these numbers agree with those from Table 5.2 and 5.3.

Experiment 1				
Round	Network type	Demons used	NR. feature connections	NR. Input connections
1	Parallel	17, 21	1	0
2	Parallel	18, 22	1	0
3	Parallel	19, 23	1	0
4	Parallel	20, 24	1	0
5	Parallel	25, 29	1	0
6	Parallel	26, 30	1	0
7	Parallel	27, 31	1	0
8	Parallel	28, 32	1	0
9	Parallel	1,2,9,10	1	0
10	Parallel	3,4,11,12	1	0
11	Parallel	5,6,13,14	1	0
12	Parallel	7,8,15,16	1	0

Table 5.5: Parameter settings of the rounds in the first experiment.

Experiment 2				
Round	Network type	Demons used	NR. feature connections	NR. Input connections
1	Parallel	17, 21	2	0
2	Parallel	18, 22	2	0
3	Parallel	19, 23	2	0
4	Parallel	20, 24	2	0
5	Parallel	25, 29	2	0
6	Parallel	26, 30	2	0
7	Parallel	27, 31	2	0
8	Parallel	28, 32	2	0
9	Parallel	1,2,9,10	2	0
10	Parallel	3,4,11,12	2	0
11	Parallel	5,6,13,14	2	0
12	Parallel	7,8,15,16	2	0

Table 5.6: Parameter settings of the rounds in the second experiment.

5.3.2 Experiments three to six

The experiments three to six test how the prediction errors of Demons are affected with different numbers of connections between Demons and features, between Demons that predict the activity of the same feature and between all available Demons.

The Horde agents in these experiments all use the parallel network type and the Demons from Table 5.2. The Demons in the experiments three and four are all connected to one feature, those in experiments five and six to both features. The difference between experiment three and four is that experiment three only allows biased connections and experiment four unbiased connections. The same difference exists between experiment five and six.

Table 5.7 shows the parameter settings for the experiments three through six. The experiments four and six do not have a round with no input connections between Demons as this data would coincide with the first round in in experiment three and seven, respectively.

These experiments test the different combinations between Demons with one or two feature connections, with zero to six biased connections and with zero to six unbiased connections. Combined, these experiments can give an overview of the effects caused by these different numbers and types of connections.

Experiment 3			
Round	Feature conn.	Input conn.	Conn. types
1	1	0	Biased
2	1	1	Biased
3	1	2	Biased
4	1	3	Biased
5	1	4	Biased
6	1	5	Biased
7	1	6	Biased

(a)

Experiment 4			
Round	Feature conn.	Input conn.	Conn. types
1	1	1	Unbiased
2	1	2	Unbiased
3	1	3	Unbiased
4	1	4	Unbiased
5	1	5	Unbiased
6	1	6	Unbiased

(b)

Experiment 5			
Round	Feature conn.	Input conn.	Conn. types
1	2	0	Biased
2	2	1	Biased
3	2	2	Biased
4	2	3	Biased
5	2	4	Biased
6	2	5	Biased
7	2	6	Biased

(c)

Experiment 6			
Round	Feature conn.	Input conn.	Conn. types
1	2	1	Unbiased
2	2	2	Unbiased
3	2	3	Unbiased
4	2	4	Unbiased
5	2	5	Unbiased
6	2	6	Unbiased

(d)

Table 5.7: *Parameter settings of experiments three to six. All experiments use the parallel network type.*

5.3.3 Experiment seven

The final experiment tests the effect on the average prediction error when Demons are connected in sequences, with varying numbers of connections a Demon has with those Demons in front of it in the sequence. This experiment can show the effect on the average prediction error when Demons have to make predictions based upon the predictions from other Demons that based their predictions on those

from other Demons, and so on.

The Horde agents in this experiment only use the sequential network type. The Demons are the same as those from experiments three to six; those from Table 5.2. Table 5.8 shows the parameter settings of this experiment.

Experiment 7		
Round	Input conn.	Conn. types
1	0	Biased
2	1	Biased
3	2	Biased
4	3	Biased
5	4	Biased
6	5	Biased
7	6	Biased

Table 5.8: Parameter settings of experiment 7. This experiment uses the sequential network type, therefore the number of features connections parameter is ignored.

Chapter 6

Results

This chapter gives a summary of the results of each experiment separately and argue these results. These summaries will exist out of the mean, standard deviation of both the prediction error of all Demons and of all Demons excluding the Extreme and Unstable Demons. It will also show the average number of both the Extreme and Unstable Demons found in every round, together with the standard deviation and standard error of the mean (SEM). All standard errors will be calculated with a sample size 40, equal to the number of Horde agents tested.

In Appendix 7 is the complete summary of all the 40 Horde agents found for every experiment, appendix 7 shows for each experiment two generated Horde networks as examples.

6.1 First Experiment

Both the first and second experiment tested the effects of the choice in different target policies (π_{target}) and constant termination probabilities (β) for Demons. Table 6.1 shows the results of the first experiment. The complete error is the prediction error with all Demons, the incomplete error with only the Demons that are not classified as either Extreme or Unstable. When their are no Extreme or Unstable Demons, the values of the complete and incomplete errors are the same.

Round	Complete Error		Incomplete Error		NR. Extreme Demons			NR. Unstable Demons		
	Average	St. Dev.	Average	St. Dev.	Average	St. Dev.	SEM	Average	St. Dev.	SEM
1:$\beta = 1.0$	0.2257	0	0.2257	0	0	0	0	0	0	0
2:$\beta = 0.8$	-	-	-	-	1	0	0	1	0	0
3:$\beta = 0.6$	-	-	-	-	0	0	0	2	0	0
4:$\beta = 0.4$	-	-	-	-	0	0	0	2	0	0
5:$\beta = 1.0$	0.2306	0.0021	0.2306	0.0021	0	0	0	0	0	0
6:$\beta = 0.8$	0.6153	0.0047	0.6153	0.0047	0	0	0	0	0	0
7:$\beta = 0.6$	0.6382	0.0048	0.6382	0.0048	0	0	0	0	0	0
8:$\beta = 0.4$	0.6444	0.0047	0.6444	0.0047	0	0	0	0	0	0
9:$\beta = 1.0$	0.2246	0.0015	0.2246	0.0015	0	0	0	0	0	0
10:$\beta = 0.8$	0.6181	0.0028	0.6181	0.0028	0	0	0	0	0	0
11:$\beta = 0.6$	0.6406	0.0022	0.6406	0.0022	0	0	0	0	0	0
12:$\beta = 0.4$	0.6448	0.0020	0.6448	0.0020	0	0	0	0	0	0

Table 6.1: *Summary of the results from the first experiment. The first column shows, besides the round, also the termination function used by all Demons in that round. The Demons in round one to four als used the behaviour policy as their target policy, round five to eight used a uniform distrubution with range* $[-2; 2]$ *to randomly select actions. The Demons in round nine to twelve used a uniform distrubution either with the range* $[-2; 0]$ or $[0; 2]$.

Round one through four tested Demons with a target policy equal to the learned behavior policy and with decreasing termination probabilities. To clarify the termination functions used by the Demons are shown in Table 6.1 for each round number. In this table we can see that a decrease in the termination probability results in an increase of the number of Extreme and Unstable Demons.

Rounds five through eight are tests with the same increase in the termination probability but the target policies of these Demons are with uniform distributions with a range from $[-2, 2]$. Rounds nine through twelve uses two different ranges instead; $[-2, 0]$ and $[0, 2]$. These rounds show a surprising similarity in the average prediction errors as can be seen in Figure 6.1 (a plot of these averages for these two sets of rounds). Table 6.1 also shows a lower standard deviation of of round five compared to round nine, round six compared to ten, round seven compared to eleven and round eight compared to twelve.

These results show that Demons are quite capable of learning to predict the activity of the angle and velocity feature when their target policy picks random actions according to a uniform distribution. The next finding is that the prediction error increase if the termination probability increases, Figure 6.1 visualizes this increase. This relates to the intuitive thought that predicting the activity over a longer future period is more difficult than predicting the current activity. We also see that Demons are not capable of learning their knowledge when their target policy is equal to the learned behavior policy and they need to predict future values as well. An explanation for this can be that this policy is too complex (due to its discontinuous nature) for the Demons to learn off-policy, and can better be learned on-policy.

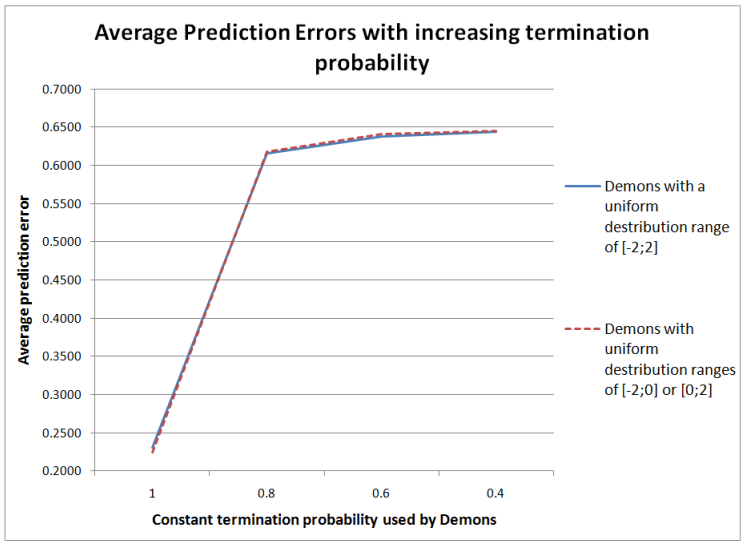


Figure 6.1: *A plot of the average prediction errors made in rounds five to eight (solid line) and nine to twelve (dashed line). This plot shows that the averages are almost the same in these two sets of rounds and that there is an increase of the average prediction error related to the decrease of the termination probability.*

Another interesting finding was found when we studied the differences between the Demons that predicted the activity of the angle and those that predicted the activity of the velocity; Demons that predicted the angle made larger prediction errors but where also the Demons that consequently where classified as the Extreme Demons in round 2. This could mean that although the Demons predicting the angle make larger errors, they are less likely to become Unstable when a complex target policy, such as the discontinuous behavior policy, is used.

6.2 Second Experiment

The second experiment was very similar to the first experiment, with the difference that in this experiment all Demons where connected to both features. Table 6.2 shows the results of this experiment with very similar increases in the prediction errors, although these errors are all larger than those in the first experiment. Also noticeable is that all Demons with the learned behavior policy as their target policy get unstable as soon as their termination probability decreases.

What we can conclude from this experiment is the same as we did in the first experiment, with the addition that connecting Demons to both feature results in an overall increase of the prediction error and caused the Demons that once where Extreme Demons in the second round to become Unstable Demons.

Round	Complete Error		Incomplete Error		NR. Extreme Demons			NR. Unstable Demons		
	Average	St. Dev.	Average	St. Dev.	Average	St. Dev.	SEM	Average	St. Dev.	SEM
1:$\beta = 1.0$	0.3229	0	0.3229	0	0	0	0	0	0	0
2:$\beta = 0.8$	-	-	-	-	0	0	0	2	0	0
3:$\beta = 0.6$	-	-	-	-	0	0	0	2	0	0
4:$\beta = 0.4$	-	-	-	-	0	0	0	2	0	0
5:$\beta = 1.0$	0.3280	0.0022	0.3280	0.0022	0	0	0	0	0	0
6:$\beta = 0.8$	0.6461	0.0045	0.6461	0.0045	0	0	0	0	0	0
7:$\beta = 0.6$	0.6664	0.0050	0.6664	0.0050	0	0	0	0	0	0
8:$\beta = 0.4$	0.6733	0.0047	0.6733	0.0047	0	0	0	0	0	0
9:$\beta = 1.0$	0.3327	0.0020	0.3327	0.0020	0	0	0	0	0	0
10:$\beta = 0.8$	0.6554	0.0024	0.6554	0.0024	0	0	0	0	0	0
11:$\beta = 0.6$	0.6769	0.0028	0.6769	0.0028	0	0	0	0	0	0
12:$\beta = 0.4$	0.6814	0.0030	0.6814	0.0030	0	0	0	0	0	0

Table 6.2: *Summary of the results from the second experiment. . The first column shows, besides the round, also the termination function used by all Demons in that round. The Demons in round one to four als used the behaviour policy as their target policy, round five to eight used a uniform distrubution with range* $[-2; 2]$ *to randomly select actions. The Demons in round nine to twelve used a uniform distrubution either with the range* $[-2; 0]$ or $[0; 2]$.

6.3 Third Experiment

The third experiment tested Horde agents with a parallel network type, with Demons connected to one feature and an increase in biased connections between Demons (round one with zero up to six in round seven). Table 6.3 shows the results of this experiment.

Round	Complete Error		Incomplete Error		NR. Extreme Demons			NR. Unstable Demons		
	Average	St. Dev.	Average	St. Dev.	Average	St. Dev.	SEM	Average	St. Dev.	SEM
1	0.5324	0.0010	0.5324	0.0010	0	0	0	0	0	0
2	-	-	0.5521	0.0225	0	0	0	2.225	2.4230	0.3831
3	-	-	0.5546	0.0270	0	0	0	2.55	2.1715	0.3433
4	-	-	0.5495	0.0287	0	0	0	2.725	1.4848	0.2348
5	-	-	0.5670	0.0363	0	0	0	2.725	1.3395	0.2118
6	-	-	0.5564	0.0274	0	0	0	2.425	0.9578	0.1514
7	-	-	0.5556	0.0290	0	0	0	1.925	0.2667	0.0422

Table 6.3: *Summary of the results from the third experiment.*

These results show that allowing biased connections between Demons results in some Demons being classified as Unstable and no Extreme Demons. It also shows more variation in the average incomplete

prediction error within the Horde agents (larger standard deviation) but with not much change in the actual average.

Figure 6.2 shows that the number of unstable Demons increases from round two to five, to decrease again in the sixth and seventh round. This initial increase can be caused by the increase in the number of connections between Demons. The decrease in the number of Unstable Demons can be caused by a lower variation in Horde networks as the number Demons that have to be in the first layer increases as the number of connections between Demons increases (as explained at the end of Section 4.5).

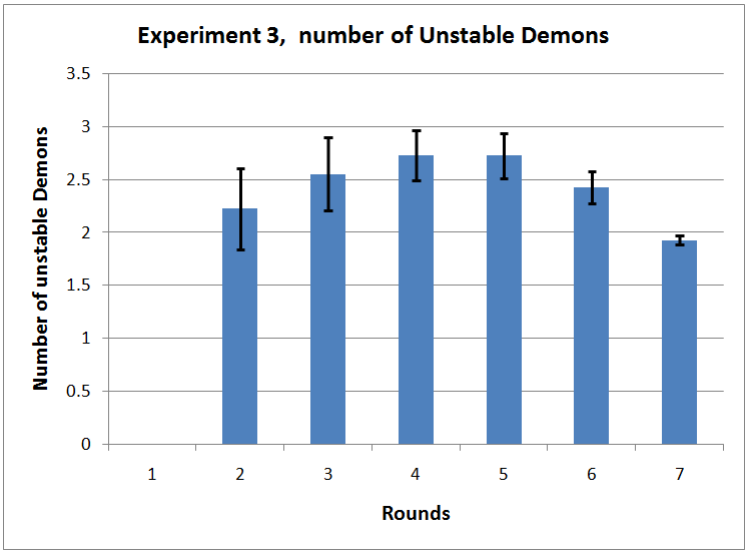


Figure 6.2: *Box plot of the average number of Unstable Demons in the different rounds of the third experiment with the standard error as the error bars.*

The absence of Unstable Demons in round one shows that the Unstable Demons do not converge because of their connections with other Demons. This conclusion is strengthened by the decrease in the standard deviation and SEM between rounds two to seven and the fact that the amount of Demons that can form the specified number of connections decreases (as explained at the end of Section 4.5). Since, as this amount decreases, there is also less room for variation in the average number of Unstable Demons.

This is enough evidence to conclude that by allowing additional random and biased connections between Demons, the result is a greater variation in the prediction errors made by Demons and causes some Demons to never converge.

6.4 Fourth Experiment

The fourth experiment is very similar to the previous experiment and so are the results (shown in Table 6.4). In this experiment unbiased connections where allowed, instead of biased ones as in the third experiment.

As the results for this experiment are almost similar to that of the third experiment, we can conclude the same things. The only difference is that here we see an increase in the average numbers of Unstable Demons as compared to those found in the third experiment. This is why we can add the conclusion that allowing unbiased connections results in an increase of the average number Unstable Demons if the Demons do diverge because of their connections with other Demons..

Round	Complete Error		Incomplete Error		NR. Extreme Demons			NR. Unstable Demons		
	Average	St. Dev.	Average	St. Dev.	Average	St. Dev.	SEM	Average	St. Dev.	SEM
1	0.5324	0.0010	0.5324	0.0010	0	0	0	0	0	0
2	-	-	0.5531	0.0213	0	0	0	1.475	3.3740	0.5335
3	-	-	0.5651	0.0746	0	0	0	3.8	5.0444	0.7976
4	-	-	0.5543	0.0439	0	0	0	3.15	4.6550	0.7360
5	-	-	0.5433	0.0443	0	0	0	5.45	4.6847	0.7407
6	-	-	0.5437	0.367	0	0	0	5.3	3.2122	0.5079
7	-	-	0.5367	0.0388	0	0	0	4.85	3.2467	0.5133

Table 6.4: *Summary of the results from the fourth experiment.*

Here we also notice an increase in the average number of Unstable Demons. Because of this we plotted these averages and their SEM again, as shown in Figure 6.3. Although this increase is a bit more irregular as the one found in the third experiment, it is still possible to conclude that there is a general increase in the average number of Unstable Demons related to the number of connections between Demons.

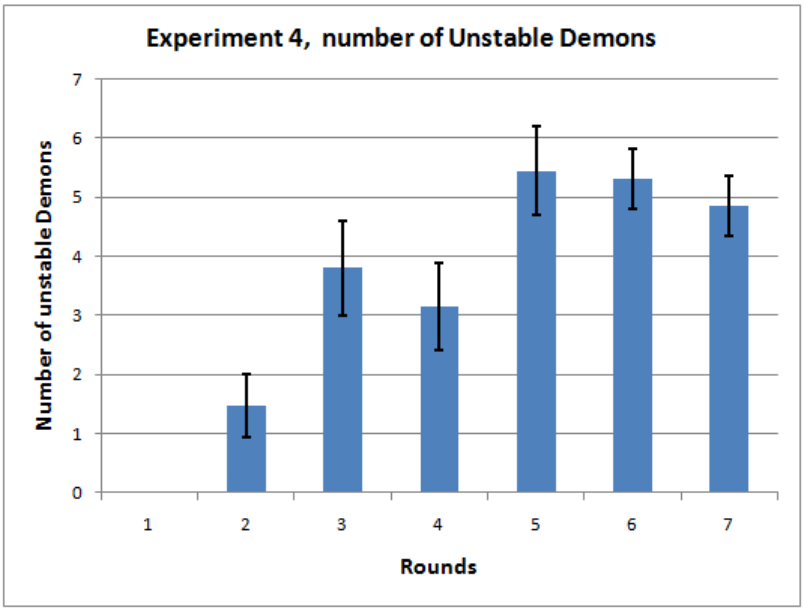


Figure 6.3: *Box plot of the average number of Unstable Demons in the different rounds of the fourth experiment with the standard error as the error bars.*

6.5 Fifth and Sixth Experiment

The fifth and sixth experiment both tested different Horde networks in the same way as the third and fourth experiment did; the rounds in both experiments allow increasingly more connections between Demons where the fifth experiment only allows biased connections and the sixth allows unbiased connections. Both experiments, however, connect the Demons with both features instead of one.

The results of these two experiments are shown in Table 6.5 and 6.6. The findings from these results are similar as those found in the previous two experiments, even with the same differences between the two. These findings where: The results show almost the same average prediction errors in the rounds with the same number of allowed connections. We also see that the standard deviations of these averages in the rounds are larger than those found in the first round with zero connections

between Demons. Finally we see that allowing connections between Demons causes some Demons to become Unstable and that this number increase when the connections are unbiased.

From these findings we can conclude the same thing as we did before; the connections between Demons cause some Demons to diverge, which increases when connections can be made with all available Demons. Such unbiased connections also cause a greater variation in the average prediction error made by the stable Demons.

Round	Complete Error		Incomplete Error		NR. Extreme Demons			NR. Unstable Demons		
	Average	St. Dev.	Average	St. Dev.	Average	St. Dev.	SEM	Average	St. Dev.	SEM
1	0.5871	0.0013	0.5871	0.0013	0	0	0	0	0	0
2	-	-	0.5912	0.0163	0	0	0	1.6	2.2049	0.3486
3	-	-	0.5974	0.0237	0	0	0	2.525	2.1482	0.3397
4	-	-	0.6047	0.0309	0	0	0	2.45	2.0248	0.3202
5	-	-	0.6051	0.0262	0	0	0	2.975	1.7466	0.2762
6	-	-	0.6075	0.0322	0	0	0	2.7	1.5392	0.2434
7	-	-	0.6019	0.0236	0	0	0	2.6	1.1503	0.1819

Table 6.5: *Summary of the results from the fifth experiment.*

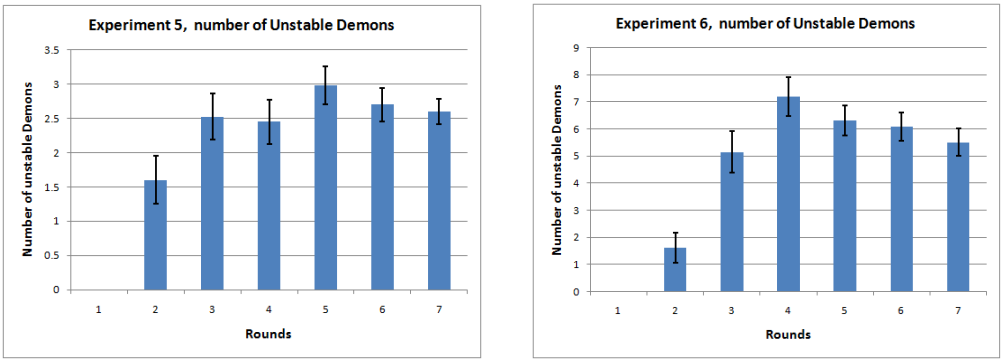
Round	Complete Error		Incomplete Error		NR. Extreme Demons			NR. Unstable Demons		
	Average	St. Dev.	Average	St. Dev.	Average	St. Dev.	SEM	Average	St. Dev.	SEM
1	0.5871	0.0013	0.5871	0.0013	0	0	0	0	0	0
2	-	-	0.6011	0.0243	0	0	0	1.625	3.5423	0.5601
3	-	-	0.6170	0.0486	0	0	0	5.15	4.8228	0.7625
4	-	-	0.6016	0.0546	0	0	0	7.2	4.5075	0.7127
5	-	-	0.5948	0.0373	0.05	0.3162	0.05	6.325	3.4891	0.5517
6	-	-	0.5918	0.0429	0	0	0	6.075	3.3158	0.5243
7	-	-	0.5899	0.0396	0.025	0.1581	0.025	5.5	3.2185	0.5089

Table 6.6: *Summary of the results from the sixth experiment.*

There are two differences between this data and those of the previous two experiments. The first is that we see a clear increase all average predictions errors from these two experiments as compared to the third and fourth experiment. This is consistent with the difference found between the first and second experiment; connecting Demons to both features causes a slight but clear increase in the average prediction errors made by those Demons. The second difference is that in the rounds five and seven of Table 6.6 (the sixth experiment) there where a few Extreme Demons. However, the SEM is equal to these averages. This means that these averages might just as well be zero and it could still be statistical significant. Therefore no sound conclusions can be made from this finding.

Figures 6.4a and 6.4b show box plots of the average number of Unstable Demons of the fifth and sixth experiment, respectively. Figure 6.4a shows that the increasing number of connections between Demons causes an increase in the average number of Unstable Demons. Figure 6.4b shows this increase as well. The possible decrease can be caused by the increasing amount of Demons in the first layer of all the generated Horde networks, as explained in Section 4.5.

With these findings we can conclude that even one additional connection between Demons causes some Demons to diverge, and the number of diverged Demons increases as the number of such connections increases.



(a) (b)

Figure 6.4: *Box plots of the average number of Unstable Demons in the different rounds of the fifth (a) and sixth (b) experiment with the standard error as the error bars.*

6.6 Seventh Experiment

The purpose of this final experiment was to see how Demon react when they are connected in sequence, in a so called sequence type network as described in Section 5.2.1. With the results of this experiment we can conclude if Demons are capable of learning the knowledge they should learn from the predictions of Demons that predict very similar knowledge. This can give us some insight in why some Demons diverge in the previous experiments.

Table 6.7 shows the results of this final experiment. The differences that are immediately noticed compared to the other experiments is that the amount of Unstable Demons is significantly less but with similar standard deviations. The standard errors are large compared to the found average, this means that the actual average can be very different. This is probably also why the sixth round has no Unstable Demons.

Round	Complete Error		Incomplete Error		NR. Extreme Demons			NR. Unstable Demons		
	Average	St. Dev.	Average	St. Dev.	Average	St. Dev.	SEM	Average	St. Dev.	SEM
1	0.3263	0.0566	0.3263	0.0566	0	0	0	0	0	0
2	-	-	0.3683	0.0611	0	0	0	0.575	2.2175	0.3506
3	-	-	0.4031	0.0741	0	0	0	0.425	1.5172	0.2399
4	-	-	0.4121	0.0624	0	0	0	0.875	2.5938	0.4101
5	-	-	0.4247	0.0598	0	0	0	0.9	2.6292	0.4157
6	0.4161	0.0683	0.4161	0.0683	0	0	0	0	0	0
7	-	-	0.4298	0.0722	0	0	0	0.3	1.3243	0.2094

Table 6.7: *Summary of the results from the fourth experiment.*

Figure 6.5 shows a plot of the average number of Unstable Demons found in each round and the error bars are the standard deviations of the averages. This plot clearly shows that there is a lot of variation between the average prediction errors of each Horde agent in every round. Knowing this, the only conclusion that can be made is that allowing biased connections between Demons in a sequential network type can cause a few Demons to never converge.

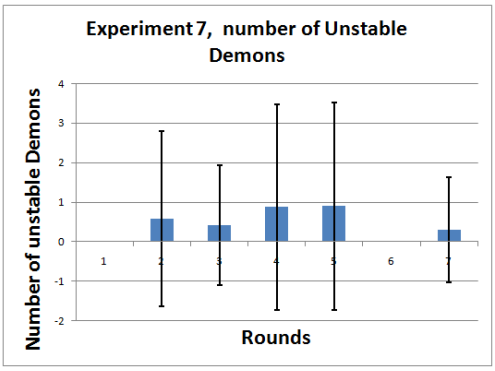


Figure 6.5

Figure 6.6: *Box plots of the average number of Unstable Demons in the different rounds of the seventh experiment. In a) the error bars are the standard error, showing the possible variation in these averages compared to the statistical population.*

Another difference between these results and those of the other experiments is that the average prediction errors of these Horde agents are smaller. To show this we plotted the average prediction errors of all rounds of experiment three to seven, which can be seen in Figure 6.7. Here we see that the average prediction errors of this experiment is less than those of the other experiments. This graph also shows the smaller difference between the experiments with one feature connection (third and fourth experiment) and the experiments with two feature connections (fifth and sixth experiment)

Finally, this plots shows that the average prediction error of this experiment increases as the number of allowed extra biased connections increases. From this we can conclude that Demons further in the sequence perform worse when they are connected to more Demons earlier in the sequence.

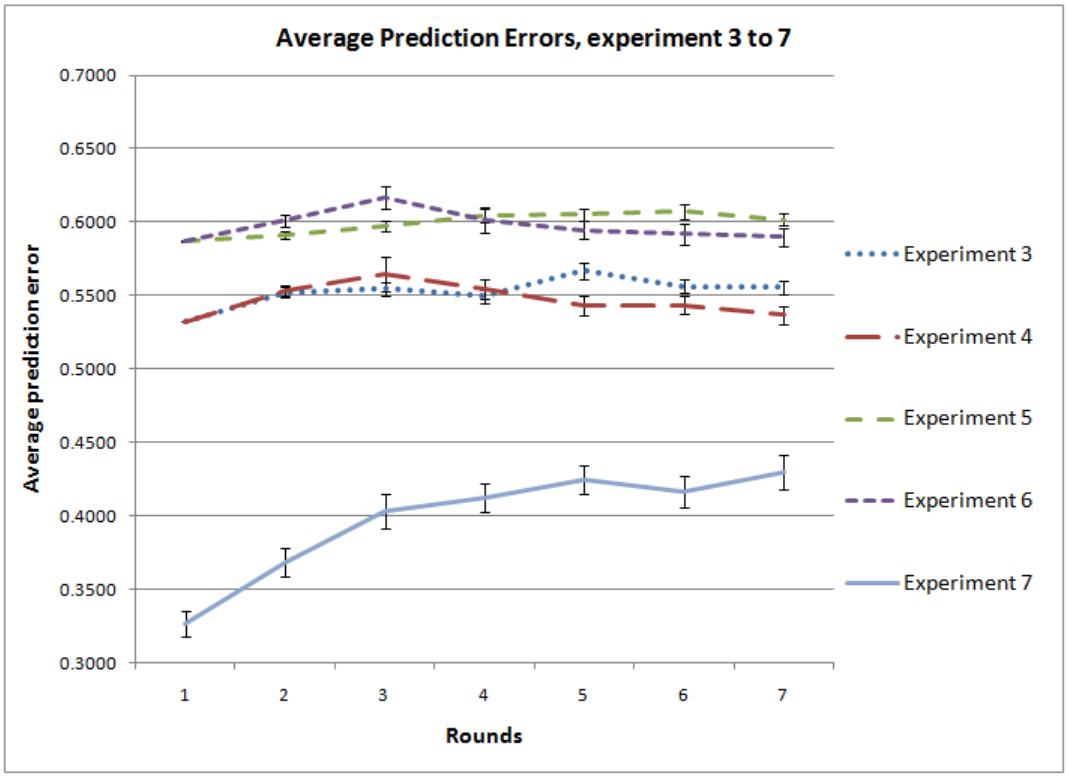


Figure 6.7: The average prediction errors of each round of the experiments three to seven. The error bars show the *standard error*.

6.7 Overview

Below we give a list of findings and conclusions drawn from the data shown above and from the data from the specific created Horde agents and their networks.

- Demons with a constant termination probability of 0.8 and with a target policy equal to the learned Behavior policy will either make extreme errors or diverge. Demons with a constant termination probability of 0.6 or lower and with a target policy equal to the learned Behavior policy will diverge. This happens no matter if Demons are connected to one or both features. The reason for this might be that the Behavior policy is discontinuous and therefore too complex.
- Demons with a target policy that chooses random actions according to a uniform distribution with a range of $[-2, 2]$ will make, on average, the same prediction errors as the Demons with a target policy with distribution ranges of $[-2, 0]$ or $[0, 2]$, but the variations in these averages declines.
- Demons connected to only one feature, the one of which they predict the activity, make on average less prediction errors than those Demons that are connected to both features. A reason for this can be that, although the two features are correlated, this correlation might be too complex for the Demons to learn and use to improve their predictions. Therefore the extra data from the additional feature connections is rendered useless for the Demon.
- Allowing connections between Demons causes a greater variation in the average prediction errors made by those Demons.
- In Horde networks where Demons have connections to other Demons, some of these Demons will diverge. The number of Demons that will diverge increases when the connections between

Demons are not only restricted to Demons that predict the activity of the same feature, but can be formed between all available Demons.

- The number of Demons that diverge increases as the number of connections between Demons increases.
- Demons in networks of the sequence type (learning their predictions based upon other predictions) perform on average better than Demons inside networks of the parallel type. The number of Demons that diverge because of extra biased connections between Demons is also far less and with similar variations.
- Demons connected further in the sequences of a sequence type network are likely to perform worse than those more at the beginning of the sequence.
- The Demons further in the sequence of a sequential network perform worse as the number connections with Demons earlier in the sequence increases. There are no signs that the amount of Demons that diverge also increases when the number of extra biased connections increases (in a sequential network).

Although these findings and conclusions show some interesting changes in Demons with different connections properties, the results are too general to conclude why some Demons diverge and why those in a sequence network perform better.

Chapter 7

Conclusion

In this thesis we created several instantiations of the Horde Architecture in the Pendulum Environment. Within these different instantiations we connected the Demons with each other and the sensory data streams from two features according to several properties. This was done in several experiments to see if there was an effect on the prediction capabilities of the Demons, and if so; what this effect was. The research question was; *Does varying (the number of) connections between Demons and sensors affect the capability of the Demons learning the knowledge they should learn?*.

To answer this research question; yes, varying in the number of connections between Demons and features does have an affect on how well these Demons can learn the knowledge they should learn. To be more specific, increasing the number of connections between similar Prediction Demons and features results in an increase of the average prediction error made by Demons. Allowing Demons to connect to other Demons that predict the same feature activity but over different numbers of time-steps, results in the divergence of several Demons. The increase of such connections result in an increase of the Demons that diverge. The final conclusion we can make, is that connecting Demons that predict the same feature activity in sequences, results in an decrease of the average prediction error.

The results found, show a clear influence of connections on the prediction errors made by Demons, although the experiments were specific for the Pendulum Environment. The experiments also tested the effect of connections on Prediction Demons and not on Control Demons as well. However, the results still provide insight in how connections between Demons can affect the entire Horde agent. From these experiments we now know that their is a noticeable and measurable effect. Especially surprising is that we now know that the Prediction Demons connected in sequences outperform the Prediction Demons that were connected in parallel within these experiments.

Future research can focus on improving and exploring the generality of the architecture. Exemplar studies for this are; exploring the theoretical boundaries of GVFs (both in complexity as learnability), improving the RL techniques that Horde requires (such as off-policy and TD learning), improving the definition of GVFs to further increase their general applications and/or decrease their dependencies on multiple parameters. The focus of future research may also lie in expanding the intuitive understanding of the capabilities of the Horde Architecture and its applications for future users of this architecture. Some example studies to do so can be; exploring and defining new and different questions asked by GVFs, applying the architecture in different (unknown) environments and experimenting with different techniques to define the data streams both from sensors and Demons. The contribution of a more intuitive approach to the Horde Architecture can help this architecture to become a reliable and understood tool for future users that need an RL agent capable of predicting and learning general knowledge.

Overall, the done research gives insight in how Prediction Demons are affected by different connections properties. The results also contribute to the understanding of the Horde architecture, as they

underline the intuitive thoughts about the effect of the connections within a Horde network.

Bibliography

[1] Thomas Degris. RLPark software package, http://rlpark.github.io/index.html, 2012.

[2] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. *Intelligent Agents III, Springer*, 3:21–36, 1997.

[3] Hamid Reza Maei and Richard S. Sutton. GQ(lambda): A general gradient algorithm for remporal-difference prediction learning with eligibility traces. In *Proceedings of the Third Conference on Artificial General Intelligence*, 2010.

[4] Jospeh Modayil, Adam White, Patrick M. Pilarski, and Richard S. Sutton. Acquiring diverse predictive knowledge in real time by temporal-difference learning. In International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems, 2012.

[5] P.M. Pilarski, M.R. Dawson, T. Degris, Jason P. Carey, K. Ming Chan, Jacqueline S. Heber, and Richard S. Sutton. Adaptive artificial limbs. *IEEE Robotics & Automation Magazine*, 2013.

[6] Thomas Planting. Unleashing the horde; a modern reinforcement learning architecture reimplemented, August 2012.

[7] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition edition, 2005.

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, 1998.

[9] Richard S. Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M. Pilarski, Adam White, and Doina Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, 10:761–768, May 2011.

[10] M. Wiering and M. van Otterlo. *Reinforcement Learning: State of The Art*. Springer, 2012.

Appendix A

Complete results of all experiments.

Round	Complete average			Incomplete average			Average extr. error		
	mean	std. err.	SEM	mean	std. err.	SEM	mean	std. err.	SEM
1	0.2257	0.0	0.0	0.2257	0	0	0	0	0
2	-	-	-	0	0	0	1.26E+128	0	0
3	-	-	-	0	0	0	0	0	0
4	-	-	-	0	0	0	0	0	0
5	0.2306	0.0021	0.003	0.2306	0.0021	0.003	0	0	0
6	0.6153	0.0047	0.007	0.6153	0.0047	0.007	0	0	0
7	0.6382	0.0048	0.008	0.6382	0.0048	0.008	0	0	0
8	0.6444	0.0047	0.007	0.6444	0.0047	0.007	0	0	0
9	0.2246	0.0025	0.002	0.2246	0.0025	0.002	0	0	0
10	0.6181	0.0028	0.005	0.6181	0.0028	0.005	0	0	0
11	0.6406	0.0022	0.003	0.06406	0.0022	0.003	0	0	0
12	0.6448	0.0020	0.003	0.6448	0.0020	0.003	0	0	0

Round	NR. Extreme			NR. Unstable		
	mean	std. err.	SEM	mean	std. err.	SEM
1	0	0	0	0	0	0
2	1	0	0	1	0	0
3	0	0	0	2	0	0
4	0	0	0	2	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
11	0	0	0	0	0	0
12	0	0	0	0	0	0

Table 7.1: Experiment 1

Round	Complete average			Incomplete average			Average extr. error		
	mean	std. err.	SEM	mean	std. err.	SEM	mean	std. err.	SEM
1	0.3229	0	0	0.3229	0	0	0	0	0
2	-	-	-	-	-	-	0	0	0
3	-	-	-	-	-	-	0	0	0
4	-	-	-	-	-	-	0	0	0
5	0.3280	0.0022	0.0004	0.3280	0.0022	0.0004	0	0	0
6	0.6461	0.0045	0.0007	0.6461	0.0045	0.0007	0	0	0
7	0.6664	0.0050	0.0008	0.6664	0.0050	0.0008	0	0	0
8	0.6733	0.0047	0.0007	0.6733	0.0047	0.0007	0	0	0
9	0.3327	0.0020	0.0003	0.3327	0.0020	0.0003	0	0	0
10	0.6554	0.0024	0.0004	0.6554	0.0024	0.0004	0	0	0
11	0.6769	0.0028	0.0004	0.6769	0.0028	0.0004	0	0	0
12	0.6814	0.0030	0.0005	0.6814	0.0030	0.0005	0	0	0

Round	NR. Extreme			NR. Unstable		
	mean	std. err.	SEM	mean	std. err.	SEM
1	0	0	0	0	0	0
2	0	0	0	2	0	0
3	0	0	0	2	0	0
4	0	0	0	2	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
11	0	0	0	0	0	0
12	0	0	0	0	0	0

Table 7.2: Experiment 2

Round	Complete average			Incomplete average			Average extr. error		
	mean	std. err.	SEM	mean	std. err.	SEM	mean	std. err.	SEM
1	0.5324	0.0010	0.002	0.5324	0.0010	0.0002	0	0	0
2	-	-	-	0.5521	0.225	0.0046	0	0	0
3	-	-	-	0.5546	0.0270	0.0043	0	0	0
4	-	-	-	0.5495	0.0287	0.0045	0	0	0
5	-	-	-	0.5670	0.0363	0.0057	0	0	0
6	-	-	-	0.5564	0.0274	0.0043	0	0	0
7	-	-	-	0.5556	0.0290	0.0046	0	0	0

Round	NR. Extreme			NR. Unstable		
	mean	std. err.	SEM	mean	std. err.	SEM
1	0	0	0	0	0	0
2	0	0	0	2.2250	2.4230	0.3831
3	0	0	0	2.5500	2.1715	0.3433
4	0	0	0	2.7250	1.4848	0.2348
5	0	0	0	2.7250	1.3395	0.2118
6	0	0	0	2.4250	0.9578	0.1514
7	0	0	0	1.9250	0.2667	0.0422

Table 7.3: Experiment 3

Round	Complete average			Incomplete average			Average extr. error		
	mean	std. err.	SEM	mean	std. err.	SEM	mean	std. err.	SEM
1	-	-	-	0.5531	0.0213	0.0034	0	0	0
2	-	-	-	0.5651	0.0746	0.0118	0	0	0
3	-	-	-	0.5543	0.0439	0.0069	0	0	0
4	-	-	-	0.5433	0.0443	0.0070	0	0	0
5	-	-	-	0.5437	0.367	0.0058	0	0	0
6	-	-	-	0.5367	0.0388	0.0061	0	0	0

Round	NR. Extreme			NR. Unstable		
	mean	std. err.	SEM	mean	std. err.	SEM
1	0	0	0	0	0	0
2	0	0	0	1.475	3.3740	0.5335
3	0	0	0	3.8	5.0444	0.7976
4	0	0	0	3.15	4.6550	0.7360
5	0	0	0	5.45	4.6847	0.7407
6	0	0	0	5.3	3.2122	0.5079
7	0	0	0	4.85	3.2467	0.5133

Table 7.4: Experiment 4

Round	Complete average			Incomplete average			Average extr. error		
	mean	std. err.	SEM	mean	std. err.	SEM	mean	std. err.	SEM
1	0.5871	0.0013	0.0002	0.5871	0.0013	0.0002	0	0	0
2	-	-	-	0.5912	0.0163	0.0026	0	0	0
3	-	-	-	0.5974	0.0237	0.0037	0	0	0
4	-	-	-	0.6047	0.0309	0.0049	0	0	0
5	-	-	-	0.6051	0.0262	0.0041	0	0	0
6	-	-	-	0.6075	0.0322	0.0051	0	0	0
7	-	-	-	0.6019	0.0236	0.0037	0	0	0

Round	NR. Extreme			NR. Unstable		
	mean	std. err.	SEM	mean	std. err.	SEM
1	0	0	0	0	0	0
2	0	0	0	1.6	2.2049	0.3486
3	0	0	0	2.525	2.1482	0.3397
4	0	0	0	2.45	2.0248	0.3202
5	0	0	0	2.975	1.7466	0.2762
6	0	0	0	2.7	1.5392	0.2434
7	0	0	0	2.6	1.1503	0.1819

Table 7.5: Experiment 5

Round	Complete average			Incomplete average			Average extr. error		
	mean	std. err.	SEM	mean	std. err.	SEM	mean	std. err.	SEM
1	-	-	-	0.6011	0.0243	0.0038	0	0	0
2	-	-	-	0.6170	0.0486	0.0077	0	0	0
3	-	-	-	0.6016	0.0546	0.0086	0	0	0
4	-	-	-	0.5948	0.0373	0.0059	2.22E+119	1.40E+120	2.22E+119
5	-	-	-	0.5918	0.0429	0.0068	0	0	0
6	-	-	-	0.5899	0.0396	0.0063	7.6973	48.6190	7.6973

Round	NR. Extreme			NR. Unstable		
	mean	std. err.	SEM	mean	std. err.	SEM
1	0	0	0	0	0	0
2	0	0	0	1.625	3.5423	0.5601
3	0	0	0	5.15	4.8228	0.7625
4	0	0	0	7.2	4.5075	0.7127
5	0.05	0.3162	0.05	6.325	3.4891	0.5517
6	0	0	0	6.075	3.3158	0.5243
7	0.025	0.1581	0.025	5.5	3.2185	0.5089

Table 7.6: Experiment 6

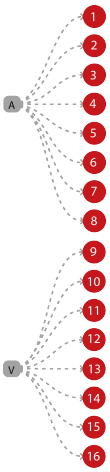
Round	Complete average			Incomplete average			Average extr. error		
	mean	std. err.	SEM	mean	std. err.	SEM	mean	std. err.	SEM
1	0.3263	0.0566	0.0090	0.3263	0.0566	0.0090	0	0	0
2	-	-	-	0.3683	0.0611	0.0097	0	0	0
3	-	-	-	0.4031	0.0741	0.0117	0	0	0
4	-	-	-	0.4121	0.0624	0.0099	0	0	0
5	-	-	-	0.4247	0.0598	0.0095	0	0	0
6	0.4161	0.0683	0.0108	0.4161	0.0683	0.0108	0	0	0
7	-	-	-	0.4298	0.0722	0.114	0	0	0

Round	NR. Extreme			NR. Unstable		
	mean	std. err.	SEM	mean	std. err.	SEM
1	0	0	0	0	0	0
2	0	0	0	0.575	2.2175	0.3506
3	0	0	0	0.425	1.5172	0.2399
4	0	0	0	0.875	2.5938	0.4101
5	0	0	0	0.9	2.6292	0.4157
6	0	0	0	0	0	0
7	0	0	0	0.3	1.3243	0.2094

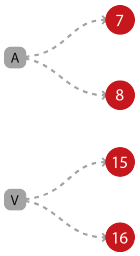
Table 7.7: Experiment 1

Appendix B

For every experiment two Horde networks are shown that where randomly generated, one network for the first round and one for the last round of the experiment.

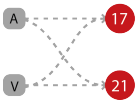


(a) First round.

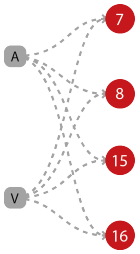


(b) Last round.

Figure 7.1: Experiment 1

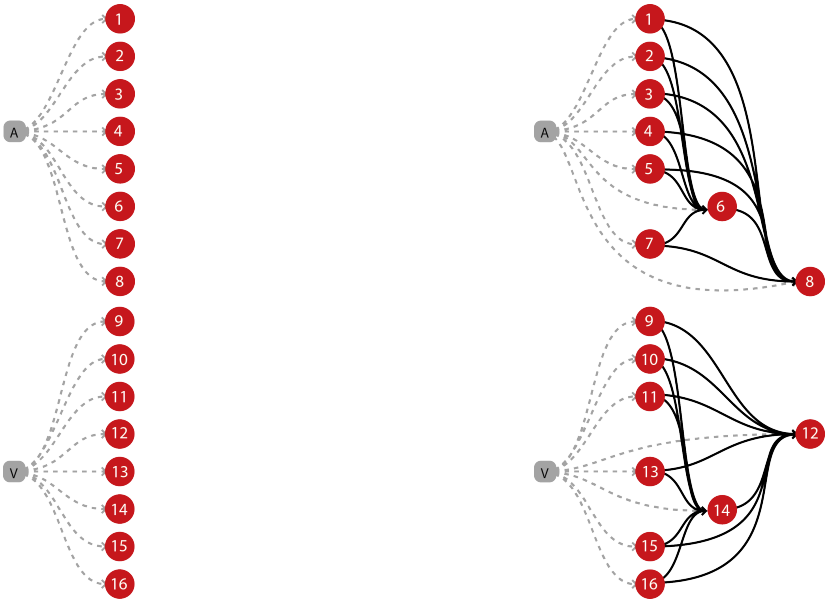


(a) First round.



(b) Last round.

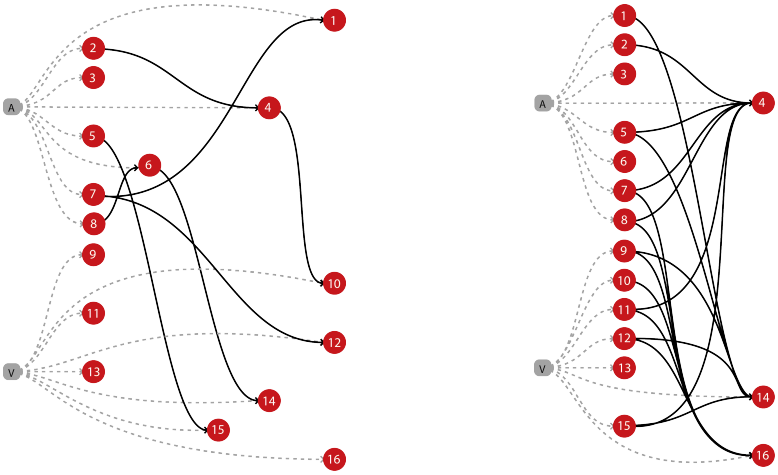
Figure 7.2: Experiment 2



(a) First round.

(b) Last round.

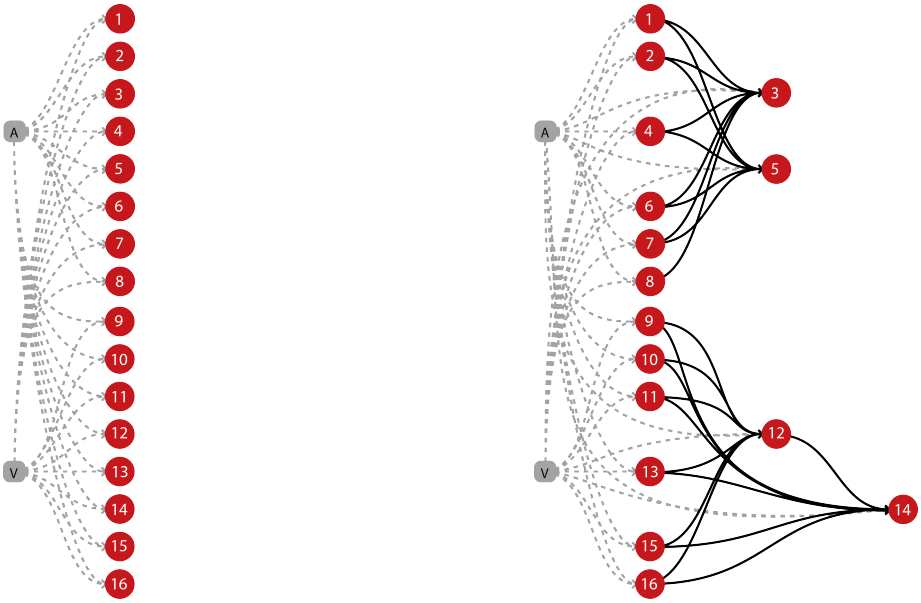
Figure 7.3: Experiment 3



(a) First round.

(b) Last round.

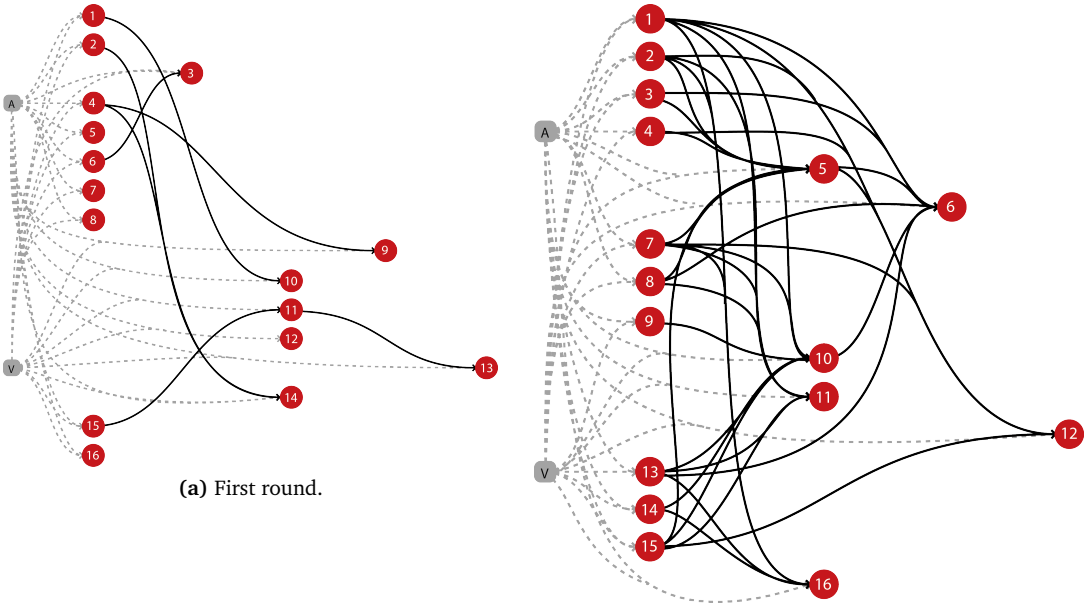
Figure 7.4: Experiment 4



(a) First round.

(b) Last round.

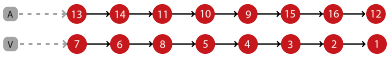
Figure 7.5: Experiment 5



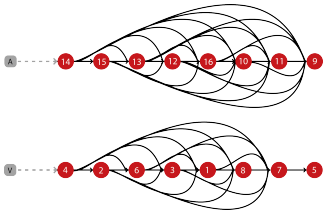
(a) First round.

(b) Last round.

Figure 7.6: Experiment 6



(a) First round.



(b) Last round.

Figure 7.7: Experiment 7