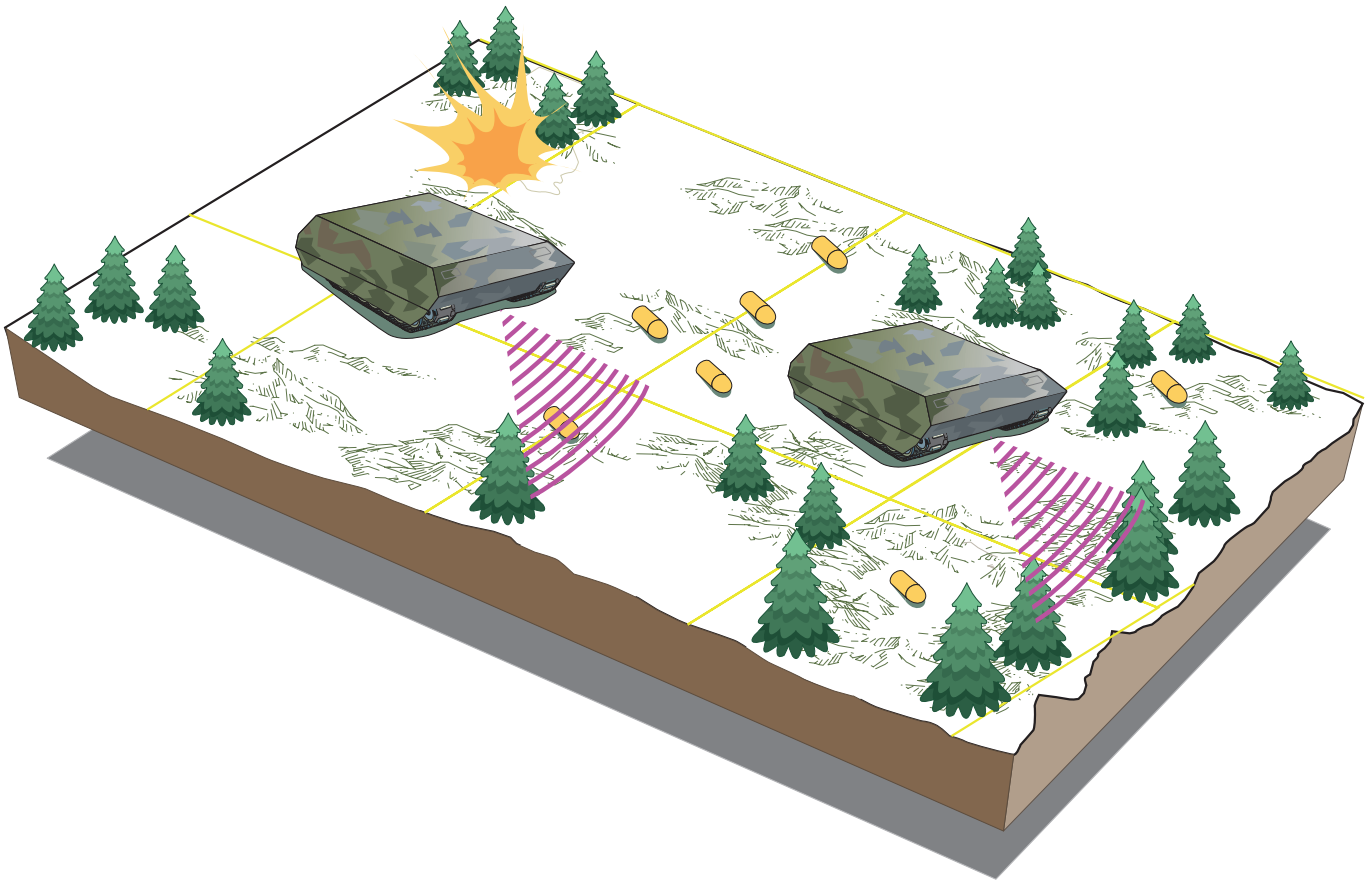
FOI-R--0386--SE
February 2002
1650-1942

Scientific report

Daniel Martin

Learning to Cooperate in a Search Mission via Policy Search



System Technology Division
SE-172 90 STOCKHOLM
Sweden

Swedish Defence Research Agency
System Technology Division
SE-172 90 STOCKHOLM
Sweden

FOI-R--0386--SE
February 2002
1650-1942

Scientific report

Daniel Martin

Learning to Cooperate in a Search Mission via Policy Search

Issuing organization	Report number, ISRN	Report type
Swedish Defence Research Agency System Technology Division SE-172 90 STOCKHOLM Sweden	FOI-R--0386--SE	Scientific report
	Research area code: Combat	
	Month year: February 2002	Project no.: E6003
	Customers code: Contracted Research	
	Sub area code: Weapons and Protection	

Author/s (editor/s)	Project manager
Daniel Martin	Peter Alvå
	Approved by: Monica Dahlén
	Scientifically and technically responsible: Martin Hagström

Report title

Learning to Cooperate in a Search Mission via Policy Search

Abstract

The dangers of and the time needed when clearing an area from unexploded ordnance can be reduced by a system consisting of unmanned, autonomous robots. The system will need less time when more than one robot cooperate to search the area.

The reinforcement learning algorithm GPOMDP is evaluated for the specific case of finding a decision rule that, given a map and the robot's position on the map, enables the robot to automatically choose between different possible actions. The actions lead to a near optimal path through an area where some parts need to be searched. A neural network is used as a function approximator to store and improve the decision rule, and also to find actions according to it. The problem is expanded to include two robots using the same decision rule, distributed in a sense that the robots pick actions according to their own perception of the surroundings and independent of the other robot's action. To achieve cooperation between the robots, they are trained to maximise a shared reward that is equal to the sum of individual rewards that are given according to the consequences of the robots' actions.

When using the learnt policy to search the largest of the experiment's areas, two robots that have been trained with a shared reward use 70% of the time that one optimal robot would need, while two agents that have been trained with their individual rewards need 88%.

Keywords

reinforcement learning, policy search, collaborating control, neural networks

Further bibliographic information	Language
	English
ISSN: 1650-1942	Pages: 50
Distribution: By sendlist	Price: Acc. to pricelist Security classification: Unclassified

Utgivare	Rapportnummer, ISRN	Klassificering
Totalförsvarets forskningsinstitut Avdelningen för Systemteknik SE-172 90 STOCKHOLM Sweden	FOI-R--0386--SE	Vetenskaplig rapport
	Forskningsområde: Bekämpning	
	Månad, år: Februari 2002	Projektnummer: E6003
	Verksamhetsgren: Uppdragsfinansierad verksamhet	
	Delområde: VVS med styrda vapen	
Författare/redaktör: Daniel Martin	Projektledare: Peter Alvå	
	Godkänd av: Monica Dahlén	
	Tekniskt och/eller vetenskapligt ansvarig: Martin Hagström	

Rapportens titel

Inlärning av samarbete under sökuppdrag genom policysökning

Sammanfattning

Farorna vid och tiden som krävs för att rensa ett område från ammunition och artillerigranater som inte har exploderat kan minskas genom att använda ett system bestående av obemannade och självgående robotar. Systemet behöver mindre tid när flera robotar samverkar under avsökningen av området.
Reinforcement learning-algoritmen GPOMDP utvärderas för att hitta en beslutsregel som möjliggör att, givet en karta och robotens position på kartan, automatiskt välja bland flera möjliga handlingar. Handlingarna leder till en nära optimal väg genom ett område där vissa delar behöver sökas av. Ett neuronnät används som funktionsapproximator för att lagra och förbättra beslutsregeln samt att hitta handlingar som följer den. Problemet expanderas till att innehålla två robotar som använder samma beslutsregel, distribuerad så att robotarna väljer handlingar efter sin egen rumsuppfattning oberoende av den andra robotens handling. För att åstadkomma samverkan mellan robotarna är de tränade att maximera en delad belöning, bestående av summan av individuella belöningar som har utdelats utifrån konsekvenserna av robotarnas handlingar.
När den inträngade policyn används för att söka av det största området som använts under experimenten behöver två robotar tränade med gemensamma belöningar 70% av tiden som en ensam optimal robot skulle behöva, medan två robotar som har tränats med individuella belöningar behöver 88%.

Nyckelord

reinforcement learning, policysökning, samverkande styrning, neuronnät

Övriga bibliografiska uppgifter	Språk
	Engelska
ISSN: 1650-1942	Antal sidor: 50
Distribution: Enligt missiv	Pris: Enligt prislista Sekretess: Öppen

Contents

Chapter 1

Introduction

This report is a master's thesis in numerical analysis written at the Swedish Defence Research Agency (FOI), Systems Technology Division. The thesis is the final part of a master's degree in mechanical engineering at the Royal Institute of Technology (KTH), Stockholm, Sweden. Supervisors are Antonios Fokas at FOI and Peter Raicevic at KTH. Examiner is Anders Lansner.

1.1 Background

The goals of FOI's project 'Guidance of Collaborating Missiles' are to develop technology and methods for coordinated and autonomous control of cooperating missiles, and to find possible tactical benefits and drawbacks with collaborating missile systems. One important part of the project is to study robust and optimal real time control algorithms.

The number and importance of unmanned vehicles is expected to increase in tomorrow's army. Unmanned vehicles both need 'low level'-control algorithms that controls the different functions of the vehicle, such as the rotation of its wheels, and 'high level'-control that decides which actions the vehicle is going to take, such as which path the vehicle will follow.

FOI's project studies missiles, but will use an experimental platform with ground vehicles to illustrate the coordinated control. The algorithm studied in this thesis can be used in ground as well as aerial vehicles and to enable a trouble free implemenation into the experimental platform, a scenario with unmanned ground vehicles is used as background during the thesis.

This thesis studies high level-control and assumes vehicles that have certain defined actions, such as go left and go right. The objective is to find control rules, or behaviours, that efficiently uses these actions to solve the problem dealt with.

Earlier work at FOI include finding vehicles' optimal flight paths [1].

One problem with previous algorithms has been the fast increase in complexity when expanding a problem to more details and to more agents. This thesis studies algorithms where the complexity does not grow as fast.

The algorithms studied are using reinforcement learning, presented in section 3.1, which is one of several methods to calculate optimal solutions to given problems. Within reinforcement learning, a special class of algorithms that use gradient descent-search in the policy function's parameter space is studied. This class of algorithms, often referred to as policy search-algorithms, has some qualities that are useful as the problems become more complex.

1.2 Purpose of the Thesis

The objective of the master's thesis is to conduct a survey of policy search algorithms, to choose an algorithm to study in detail and to evaluate it empirically. The thesis evaluates a single vehicle system and then expands it to a multi vehicle system with a few specific limitations.

The evaluation includes a Matlab implementation of an example scenario and an evaluation of the performance of the solution found versus the computational cost to find the solution. A more theoretical understanding of the algorithm is also desired to enable correct implementation of the algorithm and expansion into more than one vehicle.

In the multi vehicle section of the thesis it is specified that all vehicles should use the same control rule, often referred to as policy, but with different perception of the outside world. It is also specified that the improvement of the vehicles' ability to cooperate and strive towards a common goal when using a global reward function should be studied.

1.3 How to Read This Thesis

A number of military applications where reinforcement learning could be used to find an efficient solution are discussed in chapter 2. A more detailed example scenario is described and used as background during the implementation of the algorithm.

Technical background to reinforcement learning, function approximation and policy search methods is presented in chapter 3. A survey is made of different policy search algorithms. The presentation is meant as an introduction and is needed to understand the technical parts of the thesis. In the references, there are a number of papers and books that can be read for more in-depth knowledge.

In chapter 4, the algorithm that is evaluated is presented. The chapter also discusses how the policy is represented and how the solution is expanded into more agents. This chapter provides a detailed view of how the problem is solved.

The simulations made and the result from the simulations are presented in chapter 5. This chapter is difficult to understand without reading chapter 4 first.

The conclusions on how the algorithm performs are presented in chapter 6 and recommendations on continued work are given in chapter 7.

Chapter 2

Tactical Background

The number and importance of unmanned vehicles is expected to increase in tomorrow's army. So far, much of the focus has been on unmanned aerial vehicles (UAV), but unmanned ground vehicles (UGV) will probably play an important role and have a variety of applications.

2.1 Current Use of Unmanned Vehicles

The current UGV applications are often remotely controlled standard tanks with operators guiding. For example the US Army used UGVs for mine clearing in Bosnia. In Sweden, tests have been performed using the S103 tank with a remote control kit [6].

Controlling UGVs by remote is problematic due to a number of reasons. The high capacity communication link needed for control is difficult to maintain in a combat situation and the operator have difficulties assessing the UGVs situation and thereby getting stuck in situations where a manned vehicle would not have any problem.

2.2 Need For Autonomous Vehicles

In the future, a wide variety of UGVs with different sizes, numbers and applications are expected. The U.S. Department of Defence's Joint Robotics Program (JRP) is developing UGVs that fall into five main categories: reconnaissance, surveillance and target acquisition; military operations in urban terrain; explosive ordnance disposal; physical security; and countermine operations [8].

Many of the applications mentioned demand many robots that are more or less autonomous to be efficient. First and foremost they have to avoid obstacles and penetrate terrain by themselves (a subject not covered in this thesis). They will also need the ability to take tactical decision and cooperate with other UGVs while doing so. These vehicles will be 'tele-supervised', that is they will only need the operator to occasionally provide command and the ultimate goal is for one operator to control a group of autonomous vehicles [6].

In this thesis an algorithm is evaluated that teaches the robots how to behave in different situations to efficiently perform the mission and generalise this behaviour to deal with unexpected situations. The algorithm is meant to work on a high level, telling the robot where to go next more than telling it how to avoid an obstacle.

In the next section an example scenario will be presented. The scenario will work as an illustration for the technical discussion following in the rest of the report.

2.3 Example scenario: Eliminating Hazardous Ordnance

An infantry troop has advanced and regained control of a territory. The area ahead used to be an enemy fortification and was bombarded with artillery. In the area, hundreds of small unexploded ordnance items (UXO) are believed to be scattered. Two different kinds of UXO is around, small grenades designed to injure personnel, which is used as an area-denial weapon and ammunition that have failed to function as intended.

The area needs to be cleared for two purposes, to clear the danger of people moving around in the area getting hurt and to allow usage of available facilities in the area. Troops could be used to clear out the UXO, but it is dangerous and time consuming.

Instead, small, cheap robots (deminers) in large numbers are used to pick up and dispose the ordnance. The deminers need to be cheap since accidents with explosive ammunition is likely. For them to be cheap, they use simple sensors and control algorithms that need to have all the ordnance items pinpointed on the map.

More expensive robots with advanced sensors (scouts) are used to pinpoint all the items of ordnance before the deminers are used. Since they are expensive, the scouts are instructed to only mark the location of ordnance and not to get too close.

The scouts need to be fed with information about the terrain and where the ordnance is likely to be located (could be made from information about terrain and where the bombardment have been focused). An operator controls the robots, confirms the robots' decisions and supervises the operation.

A system like this is developed within the U.S. Joint Robotics Program (JRP). The system is called Basic UXO Gathering System (BUGS). One of the suggested solutions is having deminers that are directed by the previously mentioned scouts. The scouts will be developed from the existing Remote Controlled Reconnaissance Monitor (RECORM) vehicle.

The robots are to either place a counter-charge on the target or pick it up and deposit it at a common collection point. The goal is full autonomy but an operator will probably be needed for functions such as visual verification of detected targets [15, pages 29-33].

Chapter 3

Technical Background

The simulated robots will use *reinforcement learning* to solve the search problem as efficiently as possible. A *neural network* will store and implement the controller. The controller translates the robots sensations and the map into a benefitial action.

This chapter will give a short introduction to reinforcement learning and neural networks that will be needed to understand the rest of the thesis.

3.1 Introduction to Reinforcement Learning

In reinforcement learning, the decision maker learns how to maximise a numerical reward signal by picking the most rewarding action for different situations. The learner explores the possible actions to learn their corresponding rewards by trying them in a safe, simulated environment. [13]

In reinforcement learning the world is divided into the *agent*, the learner and decision-maker, and the *environment*, the world around the agent. The agent and the environment interact with each other. Contrary to many other optimisation methods, the dynamics of the environment does not necessarily need to be known by the agent. The environment can be seen as a black box that is fed with an action, $u \in \mathcal{U}$, by the agent and then return a new state, $i \in \mathcal{S}$, and a reward signal, r, see figure 3.1.

The dynamics of the environment can be defined by the two parameters $\mathcal{P}^u_{ii'}$, probability for state i' when in state i and taking action u, and $\mathcal{R}^u_{ii'}$, expected immediate reward for the same transition. To learn about the environment, the agent is trained in a simulated environment. Numerical rewards or penalties will be given for each step depending on what improvement it has lead to. By iteration and exploration of different actions and states, the agent learns about the environment and when the simulation has iterated to infinity, the agent will have learned $\mathcal{P}^u_{ii'}$ and $\mathcal{R}^u_{ii'}$.

Not needing an explicit model for the environment, causes both advantages, such as making it easier to model complex non-linear dynamics, and disadvantages, such as needing many iteration before a good result is achieved. An important problem that reinforcement learning has to deal with is the balance between exploring actions whose properties are unknown since they have not been explored before and exploiting knowledge that the agent already has learnt.

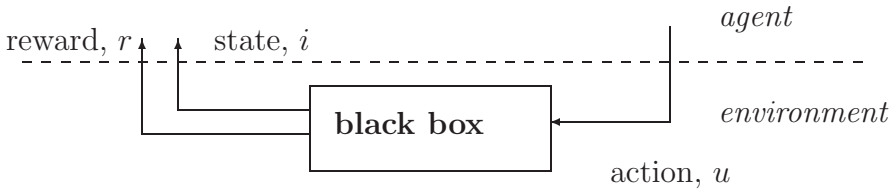


Figure 3.1: The agent and the environment interact in reinforcement learning, but the agent does not necessarily know the dynamics of the environment.

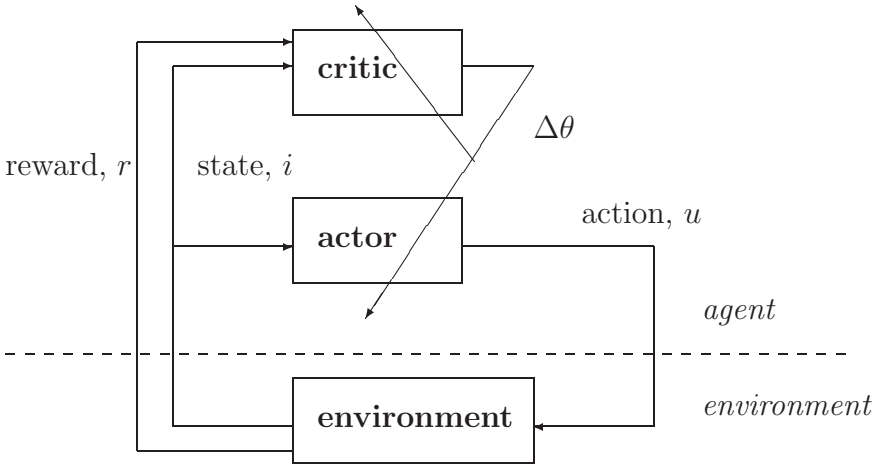


Figure 3.2: The actor and the critics are introduced to structure the agent.

3.1.1 Concepts The reward signal is central in reinforcement learning, the algorithm tries to find the solution to the problem that maximises the expected sum of future rewards, also known as the *expected discounted return*, equation 3.1. The rewards can be discounted using the parameter β to prioritise the better-known short-term rewards to the long term ones.

$$R_t = r_{t+1} + \beta r_{t+2} + \beta^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \beta^k r_{t+k+1} \tag{3.1}$$

The agent used in reinforcement learning is divided in different parts that each has an important task in structuring and solving the problem. The actor, figure 3.2, is used to map any state i to an action u. The actor is usually made up by a *policy* function, $\mu(i,u)$, that returns the probability for picking action u given state i, equation 3.2, and a function that picks an action randomly according to the probabilities given by μ.

A table where each action for each state is stored or a parameterised function can for example make up the policy function.

$$\mu(i,u) = P(u_t = u | i_t = i) \tag{3.2}$$

A critic, figure 3.2, is also introduced to evaluate the policy and to structure the rewards given by the environment. The critic usually stores the return, R_t, in one way or the other. The stored experience from the training is used to improve the actor's policy.

The return is traditionally stored in one of two ways. The *state-value function*, $J^\mu(i)$, is defined as the expected return when in state i and following policy $\mu(i,u)$ (see equation 3.3). The *action-value function*, $Q^\mu(i,u)$, is defined as the expected return when in state i taking action u and following policy $\mu(i,u)$, equation 3.4.

$$J^\mu(i) = E_\mu\{R_t | i_t = i\} = E_\mu\left\{\sum_{k=0}^{\infty} \beta^k r_{t+k+1} | i_t = i\right\}, \tag{3.3}$$

$$Q^\mu(i,u) = E_\mu\{R_t | i_t = i, u_t = u\} = E_\mu\left\{\sum_{k=0}^{\infty} \beta^k r_{t+k+1} | i_t = i, u_t = u\right\}. \tag{3.4}$$

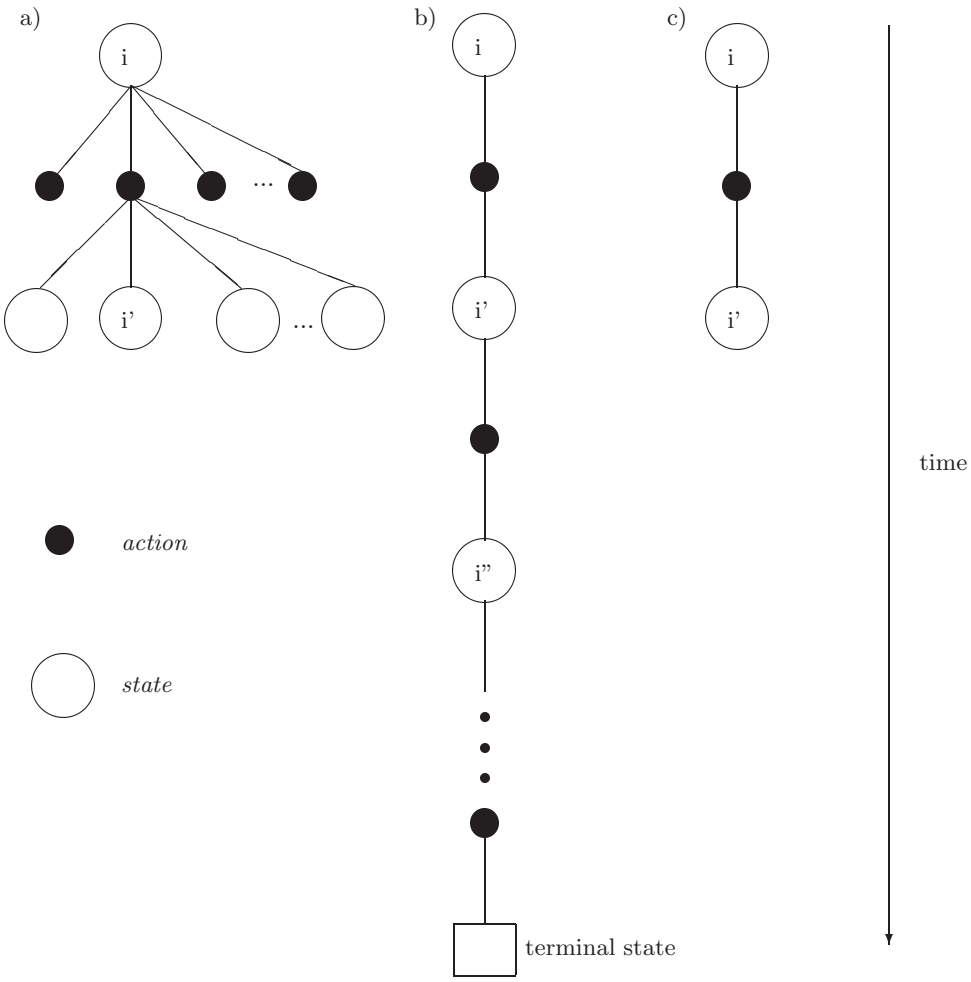


Figure 3.3: The backup diagram shows which states affect the value of state i for a) dynamic programming, b) Monte Carlo methods, c) temporal-difference learning [13].

3.1.2 Estimating the Expected Return In reinforcement learning there are three basic ways to estimate the expected return. The three ways all have different advantages but the primarily used one is probably temporal-difference learning.

The term *dynamic programming* here refers to methods that use knowledge of the dynamics of the environment to estimate the value function. The value of the state value function for state i, is updated by weighting the value of the state value function for the possible next states, i', depending on the likelihood of i' in concern of the current policy and the dynamics of the environment [13, page 91], see equation 3.5.

$$J^{\mu}(i) = E_{\mu}\{r_{t+1} + \beta J^{\mu}(i_{t+1}) | i_t = i\} = \sum_{u} \mu(i,u) \sum_{i'} \mathcal{P}^{u}_{ii'}[\mathcal{R}^{u}_{ii'} + \beta J^{\mu}(i')], \qquad (3.5)$$

Notice that the estimated value for state i is based on the estimated value for state i'. This is common in reinforcement learning and is called *boot strapping*. The backup diagram in figure 3.3a shows how every possible state in the next time step affects the value of state i.

Monte Carlo methods estimate value functions and discover optimal policies without having knowledge of the environment's dynamics, $\mathcal{P}^{u}_{ii'}$ and $\mathcal{R}^{u}_{ii'}$. The expected return is calculated when an episode is completely simulated, by summing all the rewards

received. This is averaged to the experience of previous episodes, see equation 3.6.

$$J^{\mu}(i) = E_{\mu}\{r_{t+1} + \beta r_{t+2} + \beta^2 r_{t+3} + \cdots = \lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n} \left[\sum_{k=0}^{\infty} \beta^k r_{t+k+1} \right] \tag{3.6}$$

where the index i denotes the episode number. This return will depend on the random result of each action selection and environment reaction, and if simulated an infinite amount of times the value function is supposed to converge to the expected values. The backup diagram in figure 3.3b shows that each consecutive step affects the value of state i.

Temporal-difference (TD) learning is a combination of the ideas of dynamic programming and Monte Carlo methods. Similar to Monte Carlo methods, no mathematical model of the environment is needed and similar to dynamic programming, TD bootstraps, using estimates of the next state-value to estimate the current state-value. The backup diagram in figure 3.3c shows that only state i' affect the value of state i.

3.1.3 Improving the Policy The improvement of the policy and the calculation of the value function are iterative processes. First, an arbitrary policy is chosen and used to calculate a value function. The value function is used to choose a new, improved policy. There are many different algorithms, but in each state they all seek an action that will maximise the expected return.

$$\mu(i, u) = \arg\max_{u} \left[\sum_{i'} \mathcal{P}^{u}_{ii'} \left[\mathcal{R}^{u}_{ii'} + \beta J^{\mu}(i') \right] \right]$$

Note that when the action space is limited, selecting the action that in the scope of the value function would maximise the expected value is simple. Still, care has to be taken to make sure that the other actions are properly explored as well. Usually the probability to select an action is not set to one to secure exploration.

3.1.4 Classes of Algorithms Reinforcement learning algorithms can be divided into three important classes.

- *Actor-critic* The agent has separate functions for the policy and the value estimation. The value function is used to improve the policy function, while the policy is used to pick an action depending on the state.
- *Actor-only* The agent only keeps a policy function. The policy function is updated from Monte Carlo runs of the simulated agent-environment interaction. When the policy has been updated, the rewards given in previous runs are discarded.
- *Critic-only* The agent only keeps a value function and do not store a separate policy function. Actions are chosen by picking the action that, according to the value function, returns the highest expected return. A small change in the value function can cause large changes in the policy when more than one action in a state have similar expected values. Critic-only methods have long been the most common reinforcement learning approach, examples are Q-learning [10] and SARSA(λ) [13].

Actor-critic and actor-only algorithms are often called policy search methods since the policy parameter space is searched for a policy that will maximise the expected discounted return. The policy function is often a parameterised, continuous function. The gradient of the policy function's expected return with respect to its parameters

is calculated. To improve the policy, a small step is taken in the direction of this gradient.

This thesis focuses on actor-only and actor-critic methods. Lately these classes have received more focus since they have been shown to converge to a near optimum policy even when function approximation is used to reduce the needed memory and training of the agent (function approximation is introduced in section 3.2).

3.2 Function Approximation

One way of organising the value function estimates are by storing them in a table. Although intuitive, this will only work for small problems with a limited amount of states and actions. When using a relatively small map with one hundred squares (a 10x10 map) and only two possible values, 0 and 1, in each square, more than 2^{100} states are possible. That is, even when solving a small discrete problem, the amount of states that needs to be explored and the size of the tables that needs to be stored are too large.

When modelling a continuous problem, the number of states is infinite. For large or continuous problem the need for parameter functions to approximate the tables is obvious.

The idea of *function approximation* is to approximate the value or the policy function as a parameterised function. Typically, the number of parameters is much less than the number of states. Using a function that tries to represent the previously mentioned tables is the first, intuitive approach. A better approach is to set up a vector of features, y_i, each representing an important part of the state. The function is then represented as a combination of these features. The combination can for example be linear, equation 3.7, or in the shape of a neural network where the parameters, θ_t, sets the influence of each feature.

$$J_t(i, \theta_t) = \theta_t^T y_i = \sum_{i=1}^{n} \theta_t(i) y_i(i) \tag{3.7}$$

Since there is generally no vector θ of limited size that gets all the states and values correctly, the strategy is to minimise the value prediction error from the observed samples. A class of learning methods for function approximation suiting reinforcement learning well is gradient-descent methods. These methods adjust the parameter vector by a small amount in the direction that will reduce the error the most, i.e. the negative gradient direction.

$$\theta_{t+1} = \theta_t - \frac{1}{2}\alpha \nabla_{\theta_t} [J^\mu(i_t) - J_t(i_t, \theta_t)]^2 = \theta_t - \alpha [J^\mu(i_t) - J_t(i_t, \theta_t)] \nabla_{\theta_t} J_t(i_t, \theta_t)$$

Where α is a positive step-size parameter and $\nabla_{\theta_t} f(\theta_t)$ is the gradient of any function f with respect to θ_t [13, pages 197-198].

3.2.1 Neural Networks Like Fourier series, *neural networks* are parameterised approximation of functions. The exactness of the approximation is a function of how many parameters are used, how well the parameters are chosen and how complicated the original function is.

In a single layer neural network, figure 3.4, each value, y_i, in the input vector is multiplied by a parameter $\theta_{j,i}$. These multiplied values are added together with a bias, b_j, to form the node internal activity v_j. The node activity is used as input to a non-linear *activation function*, $\phi(\cdot)$, such as a sigmoidal or exponential function to form the values y_j in the output vector.

$$\left.\begin{array}{l} y_j = \phi_1(v_j) \\ v_j = \sum_i \theta_{j,i} y_i + b_j \end{array}\right\} \Rightarrow y_j = \phi_1 \left(\sum_i \theta_{j,i} y_i + b_j \right) \tag{3.8}$$

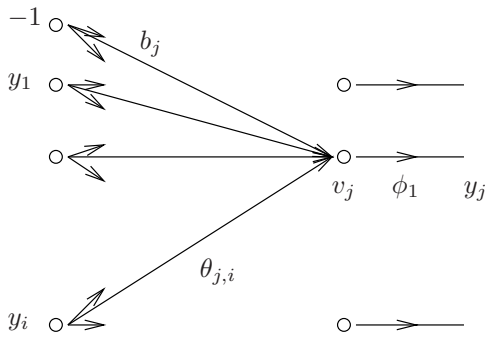


Figure 3.4: Each value, y_i, in the input vector is multiplied by a parameter $\theta_{j,i}$ and then added together with a bias, b_j, to form the output vector of a single layer neural network.

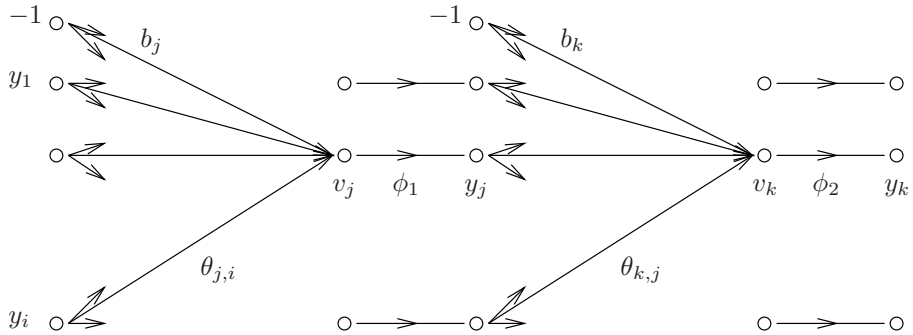


Figure 3.5: By using multi-layer neural networks, any arbitrary function can be approximated arbitrarily well.

If the neural network is to approximate a complicated function, the neural network has to consist of more than one layer, figure 3.5. When using more than one layer, the function is called *multi-layer neural networks*. In such a case, the output vector, y_j, of the single neural network is now a middle layer of the network, often called a hidden layer. The values in this vector are added in the same fashion as with the single layer.

$$y_k = \phi_2 \left(\sum_j \theta_{k,j} y_j + b_k \right) \tag{3.9}$$

The advantage of using a non-linear function is exploited when using multi-layer neural networks. If the input values had been linearly added, there had been no point using a second layer since the same effect could have been accomplished by a trivial recalculation of the parameter values, $\theta_{j,i}$. For non-linear functions this is not possible.

Adjusting a Neural Network A common method for tuning the neural network's weights is called *back-propagation*. It is a recursive method where each iteration goes as follows: The parameter when starting iteration $t+1$ are $\theta^t_{j,i}$ and $\theta^t_{k,j}$ and the biases are b^t_j and b^t_k. The output values y^{t+1}_k are calculated by equation 3.8 and 3.9. These values are used in a target function, $\delta(\theta^t_{j,i}, \theta^t_{k,j}, y^{t+1}_i)$, that the back-propagation will strive to minimise or maximise.

To optimise $\delta(\theta^t_{j,i}, \theta^t_{k,j}, y^{t+1}_i)$, the target function's partial derivatives with respect

to the parameter is used to form the gradient $\nabla_{\theta^t}\delta(\theta^t, y_i^{t+1})$, where all the parameter $\theta_{j,i}^t$ and $\theta_{k,j}^t$ are denoted by θ^t.

To improve the parameters, a small step γ is taken in the direction of the gradient (negative gradient if the function is to be minimised): $\theta^{t+1} = \theta^t + \gamma\nabla_{\theta^t}\delta(\theta^t, y_i^{t+1})$.

To find the gradient, $\nabla_{\theta^t}\delta(\theta^t, y_i^{t+1})$, the partial derivatives of the target function with respect to each parameter needs to be calculated. Using the chain rule to back-propagate the derivative through the network does this. The partial derivatives with respect to the parameters closest to the output are the easiest to find,

$$\frac{\partial\delta(\theta^t, y_i^{t+1})}{\partial\theta_{k,j}} = \frac{\partial\delta(\theta^t, y_i^{t+1})}{\partial y_k}\frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial\theta_{k,j}} \tag{3.10}$$

When calculating the partial derivative for the target function with respect to the parameters in a hidden layer, the effect of the parameter to each outer layer node that affect the target function have to be taken into account. The parameter $\theta_{j,i}$ affect the node j and the effect that node j has on all the nodes in the k-layers is summated and the chain rule gives

$$\frac{\partial\delta(\theta^t, y_i^{t+1})}{\partial\theta_{j,i}} = \sum_k\left[\frac{\partial\delta(\theta^t, y_i^{t+1})}{\partial y_k}\frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial y_j}\right]\frac{\partial y_j}{\partial v_j}\frac{\partial v_j}{\partial\theta_{j,i}} \tag{3.11}$$

The back-propagation algorithm fits reinforcement learning well because of it's iterative behaviour and that it does not fit the parameters all the way to the maximum/minimum of δ in one iteration. The gradient ascent search method will balance the different inputs and find parameters that approximate the target function as closely as possible with the given number of parameters.

3.2.2 Generalisation Using function approximation will allow the agent to *generalise* between similar states. The generalisation comes from using a continuous function to encode the action decision in each state. A non linear interpolation between learned points in the state space produces reasonable actions for similar inputs. The selection of features, used to describe the state can reduce the time needed for the learning process. Good feature selection allows the agent to generalise from an action in one state to another closely related state.

To illustrate how the feature selection affects the agent's ability to generalise between similar states, consider the case described in section 2.3. To simplify the problem and to represent the world surrounding the agent, a table representing a simple map is created. An unknown area in the world is shadowed in the table while an already explored area is not shadowed. Now, consider using the map as to represent the state that the agent will use to take decisions. The value of each square will then be a feature used to represent the state. Although each state is uniquely represented, and the agent has all the information it needs to take a decision, this approach is problematic from a generalisation point of view. If the map would be displaced, for example if the agent and each known and unknown square would be moved one step to the left, figure 3.6, almost every feature would be changed. Almost every square on the map changes from known to unknown.

However, from the agent's point of view, the new, displaced state is exactly the same as the non-displaced. A state representation centered on the agent would obviously solve this problem since the optimal behaviour of the agent is to keep acting as the displacement had never happened.

In [1], Axelson solves this by using a sensor that translates the global map to a local state, see figure 3.7. The sensor sums the amount of unknown map squares in each sensor area, creating an approximation of the global state, centralised around the agent. Since the sensor areas are larger further away from the agent, the parts of the global map close to the agent are more detailed.

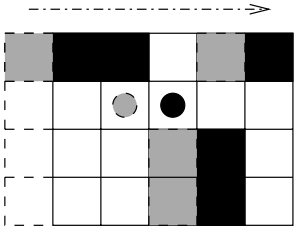


Figure 3.6: If the map would be displaced, for example if the agent and each known and unknown square would be moved one step to the left, almost every feature would be changed. The grey squares are map squares that needs to be searched before the displacement and the black squares needs to be searched after. The circles represent the agent before and after the displacement.

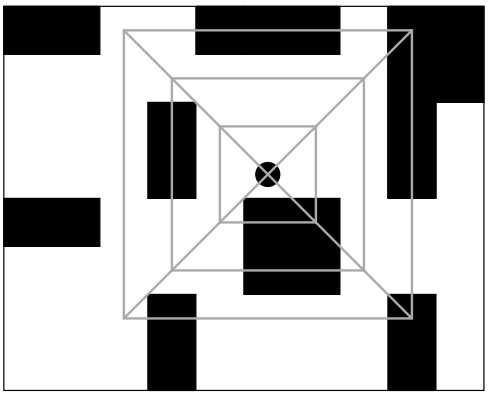


Figure 3.7: The local sensor averages the parts of the map so as data close to the agent is more detailed than data far away.

When using the local state to choose an action, a displacement in the global map would cause the centering of the local view to be displaced as well, ending up with the same local view as before the shift.

3.3 Policy Search Methods

Below is a brief presentation of some of the policy search algorithms that exist. The algorithm that is chosen to be investigated is presented in detail in section 4.1.

William's REINFORCE method [16] uses no value function to learn and store the expected return from an action or state. Instead, the method runs one complete episode until termination and uses the return as an approximation of J^{μ}. This gives an unbiased estimation of the return but since the method has not stored any previous experience, chance can create large variance in the return.

The policy function is updated in the direction of the gradient, estimated from the return estimation. Since the return estimation has a large variance, the gradient also shows large variance. The estimation is however unbiased, i.e. the average gradient is in the correct direction. Unfortunately, the large variance in performance makes the method converge slowly.

Sutton et al. have developed a modification of the REINFORCE method [14]. The form the gradient is written on was changed to make it suitable to store the experience in the form of a state-value or action-value function. Using a value function reduces the variance in performance gradient estimates that are problematic in the REINFORCE algorithm. The method is shown to converge to a locally optimal policy using an arbitrary differentiable function approximation.

Konda and Tsitsiklis present a similar method in which the critic uses TD learning with a linear approximation architecture and where the actor is updated in the approximate gradient direction [9]. In the rest of the text, this method and that of Sutton et al. will be considered similar and called PIFA (Policy Iteration with Function Approximation).

The Action Transition Policy Gradient (ATPG) algorithm presented in [7] is similar to the REINFORCE method in a sense that it does not use function approximation to represent the action-value function Q. Instead it uses a selective process to choose direct samples of Q to reduce the variance in the performance gradient estimates. The ATPG algorithm is compared by experiments to the PIFA and the REINFORCE methods and is shown to converge with order of magnitude fewer iterations. It is also shown that the ATPG converges to an optimal policy when the policy and action value function follows piece-wise continuity conditions.

In [2, 5, 3] an algorithm called GPOMDP is introduced. The algorithm is shown to optimise the performance of the policy in a *partially observable Markov decision process*. As the REINFORCE and the ATPG algorithms, GPOMDP do not store a value function. In [3], the GPOMDP algorithm is also used in a conjugate-gradient procedure that might find local optimas faster than using gradient ascent.

Chapter 4

Solution

4.1 Algorithm Evaluated - GPOMDP

4.1.1 Why GPOMDP? Algorithms that use a separate policy function, often called policy search algorithms, have been increasingly popular within reinforcement learning. Some policy search algorithms have been proven [14] to converge to local optimum even when using function approximation instead of tables to represent the policy. Since the use of function approximation is needed when studying larger problems, this convergence property is very important.

Within the policy search class, the actor-only model has one large advantage. When using a global reward function and a distributed actor, actor-only methods can be expanded into more than one agent with relatively small changes [11]. The global reward function used in this thesis is presented in section 4.5.2 and the distributed actor in section 4.5.1.

Grudic and Ungar presented an actor-only algorithm in [7]. Since it is actor-only it does not store a value function, but takes one sample episode to decide how the policy function is to be improved. After the policy function is improved, the data from the episode is thrown away. Grudic and Ungar show that this algorithm not only converges when using function approximation but it also converges faster than the traditional actor-critic algorithm and the PIFA algorithm mentioned in section 3.3.

GPOMDP, an algorithm developed by Baxter and Bartlett [2, 5, 3], is also an actor-only algorithm. I have not seen a speed comparison between GPOMDP to Grudic's and Ungar's algorithm, but GPOMDP models the environment as a partial observable Markov decision process (POMDP) instead of a Markov decision process.

It is important that the algorithm converges for POMDPs when using the sensor presented in section 3.2.2 since the sensor cannot discriminate between every possible global state, i.e. more than one global state can look exactly the same to the agent.

The mathematical analysis of GPOMDP is clear and more developed than that of Grudic and Ungar, which combined with Baxter's good reputation in reinforcement learning suggest that GPOMDP is a promising algorithm.

4.1.2 Mathematical Background Most of the information in this section and section 4.1.3 can be found elsewhere [3, 2, 5]. These articles are recommended for formal proofs and a more detailed explanation of GPOMDP.

The formal framework used to describe the agent's interaction with the environment is a partial observable Markov decision process, POMDP. A POMDP iterates in the following way: At time t, the environment is in state i_t. The agent receives an observation of the environment, $y(i_t)$, according to a probability distribution $v_y(i)$. From the observation, the agent decides to take an action u_t according to the policy $\mu_{u_t}(\theta, y_t)$, where θ are the parameters in a parametric policy function. Depending on the action, the environment returns a new state i_{t+1} according to the probability $p_{ii'}(u_t)$. The Markov chain corresponding to θ has state transition matrix $P(\theta) = [p_{ii'}(\theta)]$ given by $p_{ii'}(\theta) = \boldsymbol{E}_{y\sim v(i)}\boldsymbol{E}_{u\sim\mu(\theta,y)}p_{ii'}(u)$. When the POMDP has reached its final state i_N, the process is finished.

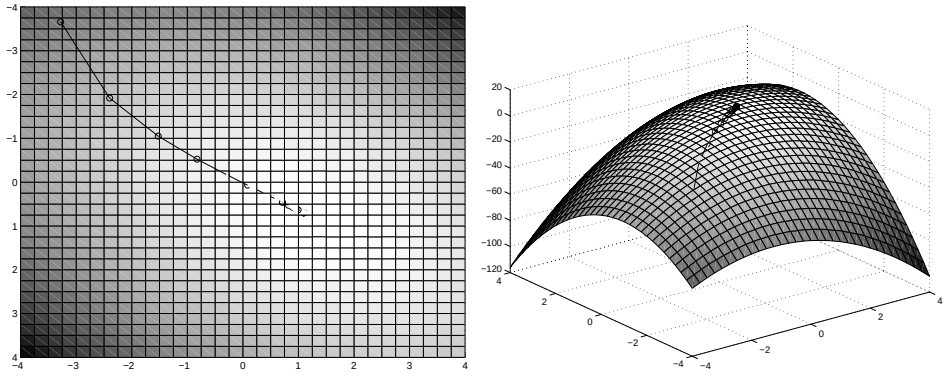


Figure 4.1: Two different views of a gradient ascent search. Gradient ascent finds a maximum by recursively taking small steps in the gradient direction.

Assumption 1
From now on it is assumed that each $P(\theta)$ has a unique stationary distribution $\pi(\theta) := [\pi(\theta,1),\ldots,\pi(\theta,n)]^T$ satisfying the balance equations $\pi(\theta)^T P(\theta) = \pi(\theta)^T$. The magnitudes of rewards, $|r(i)|$, are uniformly bounded by $R < \infty$ for all states i ([3, Assumption 1]).

The goal of the method is to find a θ that maximises

$$\eta_\beta(\theta) := \sum_{i=1}^{n} \pi(\theta,i) J_\beta(\theta,i) = \pi^T J_\beta$$

where

$$J_\beta(\theta,i) := \lim_{T\to\infty} \tfrac{1}{T} E_\theta \left[\sum_{t=0}^{T} \beta^t r(i_t) | i_0 = i \right] = \{\text{the value of each state } i \in \mathcal{S}\},$$

and $\eta_\beta(\theta)$ is called the expected discounted reward. $\eta_\beta(\theta)$ can be thought of as the reward that can be expected for the next step averaged over each state in the POMDP.

A recursive method for finding the maximum of a function is gradient ascent, which goes as follows: A starting parameter vector θ_0 is chosen. The gradient at θ_0 is calculated and a small step is taken in the direction of the gradient, $\theta_1 \leftarrow \theta_0 + \gamma \nabla \eta(\theta_0)$. This is repeated until the gradient is small enough and the maximum is considered reached. An illustration of gradient ascent for a two-element parameter vector is shown in figure 4.1. This two-dimensional search can easily be expanded to a more complex policy function with more than two parameters.

There are two important limitations to the gradient ascent search: The policy needs to be differentiable and a local optimum can be found and picked instead of the global optimum. Using a differentiable neural network to calculate a stochastic policy solves the first limitation but the second remains a problem.

According to [2, Theorem 1]

$$\eta_\beta(\theta) = \frac{\eta(\theta)}{1-\beta},$$

where $\eta(\theta)$ is the average reward. In other words, it is equivalent to maximise the expected discounted reward and to maximise the expected reward.

The long-term expected reward

$$\eta(\theta) = \lim_{T\to\infty} \frac{1}{T} E_\theta \left[\sum_{t=1}^{T} r(i_t) \right],$$

is independent of the starting state i_0 and equal to

$$\eta(\theta) = \sum_{i=1}^{n} \pi(\theta, i) r(i) = \pi' r.$$

The gradient of the expected reward with respect to the parameters θ can be calculated exactly by solving

$$\nabla\eta = \nabla\pi^T r = \pi^T \nabla P [I - P + e\pi^T]^{-1} r, \tag{4.1}$$

where the dependence on θ has been left out. Equation 4.1 should be read as one equation for each $\frac{\partial}{\partial\theta_i}$. However, solving this equation system is only possible when having a small amount of states and a better way is to find an approximation of the gradient. Equation 4.1 can be rewritten as

$$\nabla\eta = (1 - \beta)\nabla\pi^T J_\beta + \beta\pi^T \nabla P J_\beta, \tag{4.2}$$

where J_β is the discounted value of state i. It can be shown that when β approaches 1, the first term in equation 4.2 becomes negligible. That is, when β approaches 1,

$$\nabla_\beta\eta := \beta\pi^T \nabla P J_\beta, \tag{4.3}$$

is a good approximation of the gradient $\nabla\eta$. It can also be shown that β does not need to be close to 1 if $1/(1 - \beta)$ is large compared to the mixing time, i.e. the time constant determining how quickly the underlying Markov chain converges to it's stationary distribution.

4.1.3 GPOMDP - The Algorithm GPOMDP is an algorithm that uses one sample path of the POMDP to approximate $\nabla_\beta\eta$. $\nabla_\beta\eta$ is used to approximate the gradient of η since $\nabla\eta$ is unknown in general. $\nabla_\beta\eta$ can be an arbitrary good approximation of $\eta(\theta)$ by choosing β with respect to the mixing time of the underlying Markov chain [2].

To understand how GPOMDP (algorithm 1) works, first note that the approximation of $\nabla_\beta\eta$ after T steps in the POMDP, Δ_T, approaches $\nabla_\beta\eta$ as $T \to \infty$ since [2, Theorem 6]

$$\nabla_\beta\eta = \pi^T \nabla P J_\beta = \sum_{i,j,y,u} EZ^T,$$

where

$$Z^T := \chi_i(X_t)\chi_j(X_{t+1})\chi_u(U_t)\chi_y(Y_t)\frac{\nabla\mu_u(\theta, y)}{\mu_u(\theta, y)} J(t+1), \tag{4.4}$$

$$J(t+1) = \sum_{s=t+1}^{\infty} \beta^{s-t-1} r(X_s),$$

and $\chi_i(\cdot)$ denotes the indicator function for state i

$$\chi_i(X_t) := \begin{cases} 1 & \text{if } X_t = i, \\ 0 & \text{otherwise.} \end{cases}$$

The letters i and j denote the possible states and X_t is the state at time t, u denotes the possible actions and U_t is the action taken at time t. y represents the possible observations and Y_t is the observation at time t.

Z^T in equation 4.4 equals zero except for when all the $\chi_i(\cdot)$ functions are one. In a simulation of a POMDP, the indicator functions are only one for the actions and states that occur. Because of this, the sum of $\frac{\nabla\mu_u(\theta,y)}{\mu_u(\theta,y)} J(t+1)$ for the states

and actions that occur during a non-biased simulation will converge to $\nabla_\beta \eta$ as the simulation iterates to infinity.

$$\lim_{T\to\infty} \frac{1}{T} \sum_{t=0}^{T} \frac{\nabla \mu_{u_t}(\theta, y_t)}{\mu_{u_t}(\theta, y_t)} J(t+1) \to \sum_{i,j,y,u} EZ^T = \nabla_\beta \eta. \tag{4.5}$$

GPOMDP offers a clever way of adding up the $\frac{\nabla \mu_u(\theta,y)}{\mu_u(\theta,y)} J(t+1)$ during a simulation of a POMDP and approximating $\nabla_\beta \eta$ without storing all the rewards, states and actions found in the process. To understand how GPOMDP finds the average, note that each term in left hand side of equation 4.5 can be rewritten as

$$\begin{array}{ll}
t=0 & \frac{\nabla \mu_{u_0}(\theta,y_0)}{\mu_{u_0}(\theta,y_0)} J(1) = \frac{\nabla \mu_{u_0}(\theta,y_0)}{\mu_{u_0}(\theta,y_0)} (r_1 + \beta J(2)) = \frac{\nabla \mu_{u_0}(\theta,y_0)}{\mu_{u_0}(\theta,y_0)} (r_1 + \beta r_2 + \beta^2 J(3)) = \cdots \\
t=1 & \frac{\nabla \mu_{u_1}(\theta,y_1)}{\mu_{u_1}(\theta,y_1)} J(2) = \frac{\nabla \mu_{u_1}(\theta,y_1)}{\mu_{u_1}(\theta,y_1)} (r_2 + \beta J(3)) = \frac{\nabla \mu_{u_1}(\theta,y_1)}{\mu_{u_1}(\theta,y_1)} (r_2 + \beta r_3 + \beta^2 J(4)) = \cdots \\
\vdots & \\
t=n & \frac{\nabla \mu_{u_n}(\theta,y_n)}{\mu_{u_n}(\theta,y_n)} J(n+1) = \frac{\nabla \mu_{u_n}(\theta,y_n)}{\mu_{u_n}(\theta,y_n)} (r_{n+1} + \beta J(n+2)) = \\
 & = \frac{\nabla \mu_{u_n}(\theta,y_n)}{\mu_{u_n}(\theta,y_n)} (r_{n+1} + \beta r_{n+2} + \beta^2 J(n+3)) = \cdots
\end{array}$$

Now, collect all terms in the left-hand side of equation 4.5 that should be multiplied with each reward.

$$\begin{aligned}
\tfrac{1}{T} \sum_{t=0}^{T} \tfrac{\nabla \mu_{u_t}(\theta,y_t)}{\mu_{u_t}(\theta,y_t)} J(t+1) = \tfrac{1}{T} \Bigg[& \underbrace{\frac{\nabla \mu_{u_0}(\theta,y_0)}{\mu_{u_0}(\theta,y_0)}}_{z_1} r_1 + \dots \\
\dots + & \underbrace{\left(\beta \frac{\nabla \mu_{u_0}(\theta,y_0)}{\mu_{u_0}(\theta,y_0)} + \frac{\nabla \mu_{u_1}(\theta,y_1)}{\mu_{u_1}(\theta,y_1)} \right)}_{z_2} r_2 + \dots \\
\dots + & \underbrace{\left(\beta^2 \frac{\nabla \mu_{u_0}(\theta,y_0)}{\mu_{u_0}(\theta,y_0)} + \beta \frac{\nabla \mu_{u_1}(\theta,y_1)}{\mu_{u_1}(\theta,y_1)} + \frac{\nabla \mu_{u_2}(\theta,y_2)}{\mu_{u_2}(\theta,y_2)} \right)}_{z_3} r_3 + \cdots \Bigg]
\end{aligned} \tag{4.6}$$

Note the relation

$$z_{t+1} = \beta z_t + \frac{\nabla \mu_{u_t}(\theta, y_t)}{\mu_{u_t}(\theta, y_t)},$$

which is used in the GPOMDP algorithm to reduce the amount of data needed to be stored to approximate the gradient.

The GPOMDP algorithm will only work properly if assumption 2 and assumption 1 are satisfied.

Assumption 2
The derivatives $\frac{\partial \mu_u(\theta,y)}{\partial \theta_k}$ *exist for all* $u \in \mathcal{U}, y \in \mathcal{Y}$ *and* $\theta \in \Re^K$. *The ratios*

$$\left[\frac{\left| \frac{\partial \mu_u(\theta,y)}{\partial \theta_k} \right|}{\mu_u(\theta, y)} \right]_{y=1\dots M; u=1\dots N; k=1\dots K}$$

are uniformly bounded by $B < \infty$ *for all* $\theta \in \Re^K$ *([3, Assumption 5]).*

There is one parameter that needs to be set in GPOMDP, the discount factor $\beta \in [0, 1)$. Setting it is a trade-off between having a low variance (β close to 0) and having a low bias (β close to 1).

Algorithm 1 The GPOMDP algorithm [3, Algorithm 1]

1: Given:

- Parameterised class of randomised policies $\{\mu(\theta,\cdot) : \theta \in \Re^K\}$ satisfying Assumption 2.
- Partially observable Markov decision process which when controlled by the randomised policies $\mu(\theta,\cdot)$ corresponds to a parameterised class of Markov chains satisfying Assumption 1.
- $\beta \in [0,1)$
- Arbitrary (unknown) starting state i_0
- Observation sequence $y_0, y_1, ...$ generated by the POMDP with controls $u_0, u_1, ...$ generated randomly according to $\mu(\theta, y_t)$
- Bounded reward sequence $r(i_0), r(i_1), ...$ where $i_0, i_1, ...$ is the (hidden) sequence of states of the Markov decision process.

2: Set $z_0 = 0$ and $\Delta_0 = 0$ ($z_0, \Delta_0 \in \Re^K$).
3: **for** each observation y_t, control u_t, and subsequent reward $r(i_{t+1})$ **do**
4: $\quad z_{t+1} = \beta z_t + \frac{\nabla \mu_{u_t}(\theta, y_t)}{\mu_{u_t}(\theta, y_t)}$
5: $\quad \Delta_{t+1} = \Delta_t + \frac{1}{t+1}[r(i_{t+1}) z_{t+1} - \Delta_t]$
6: **end for**

4.1.4 Conjugate Gradient Search In a gradient ascent search, taking a small step in the direction of the gradient changes the parameter values. When making a conjugate gradient search, a clever combination between the previous search direction and the gradient is used as search direction. Due to its better use of earlier data, conjugate gradient converges faster to a maximum than gradient ascent. For a more detailed description of the conjugate gradient method, we recommend [12].

The algorithm CONJPOMDP, algorithm 2, uses the Polak-Ribiere conjugate gradient algorithm that is designed to use noisy and biased data of the gradient, for example provided by GPOMDP. When the quadratic norm of the gradient is small enough, the algorithm terminates.

In this thesis, CONJPOMDP will be using GPOMDP to approximate the gradient $\nabla_\beta \eta(\theta)$ but any well-working method of approximation would do. In the algorithm, the symbol $\cdot$ is used to denote inner products.

Instead of just taking a small step in the search direction, the conjugate gradient method often finds a maximum in the search direction by doing a line search. The algorithm GSEARCH, algorithm 3, can be used to bracket this maximum and finding a step size while only using the gradient information that GPOMDP can produce. By using GSEARCH, the convergence time of CONJPOMDP might be reduced.

The algorithm GSEARCH works by projecting the gradient on the search direction, figure 4.2. If the projection is positive, the point in which the gradient is taken is in an upward slope in the search direction, if the projection is negative, there is a downward slope.

At the starting point, θ_0, there is an upward slope if the gradient has been correctly estimated. GSEARCH then tries the initial step size s_0 and calculates the gradient at the new point $\theta_1 \to \theta_0 + s_0\theta_0$.

If the new gradient's projection on the search direction is positive, the point θ_1 is in an upward slope and the maximum has not been reached. Since this means that the maximum point is beyond θ_1 the step size is doubled and the gradient is calculated at the new point $\theta_2 \to \theta_0 + 2s_0\theta_0$. This is repeated until a point, θ_N, is found where

Algorithm 2 CONJPOMDP (GRAD,θ,s_0,ϵ)$\leftarrow \Re^K$ [3, Algorithm 2]

Given:

- GRAD: $\Re^K \leftarrow \Re^K$: an estimate of the gradient of the objective function to be maximised.
- Starting parameters $\theta \in \Re^K$
- Initial step size $s_0 > 0$.
- Gradient resolution ϵ

$g = h =$GRAD(θ)
while $\|g\|^2 \geq \epsilon$ **do**
  GSEARCH(GRAD, θ, h, s_0, ϵ)
  $\Delta =$GRAD(θ)
  $\gamma = (\Delta - g) \cdot \Delta / \|g\|^2$
  $h = \Delta + \gamma h$
  if $h \cdot \Delta < 0$ **then**
    $h = \Delta$
  end if
  $g = \Delta$
end while
return θ

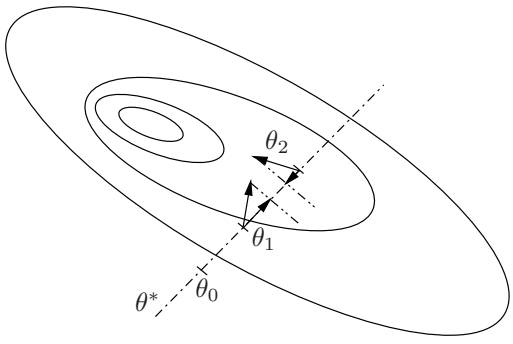


Figure 4.2: By projecting the gradient on the search direction, GSEARCH is able to find a maximum along the search direction given to GSEARCH.

Algorithm 3 GSEARCH (GRAD,θ_0,θ^*,s,ϵ)$\leftarrow \Re^K$ [3, Algorithm 3]

Given:

- GRAD: $\Re^K \leftarrow \Re^K$: gradient estimate
- Starting parameters $\theta_0 \in \Re^K$
- Search direction $\theta^* \in \Re^K$ with $\text{GRAD}(\theta_0) \cdot \theta^* > 0$
- Initial step size $s > 0$
- Inner product resolution $\epsilon \geq 0$

$\theta = \theta_0 + s\theta^*$
$\Delta =$GRAD(θ)
if $\Delta \cdot \theta^* < 0$ **then**
  Step back to bracket the maximum
  repeat
    $s_+ = s, p_+ = \Delta \cdot \theta^*, s = s/2$
    $\theta = \theta_0 + s\theta^*$
    $\Delta =$GRAD(θ)
  until $\Delta \cdot \theta^* > -\epsilon$
  $s_- = s$
  $p_- = \Delta \cdot \theta^*$
else
  Step forward to bracket the maximum
  repeat
    $s_- = s, p_- = \Delta \cdot \theta^*, s = 2s$
    $\theta = \theta_0 + s\theta^*$
    $\Delta =$GRAD(θ)
  until $\Delta \cdot \theta^* < \epsilon$
  $s_+ = s$
  $p_+ = \Delta \cdot \theta^*$
end if
if $p_- > 0$ and $p_+ < 0$ **then**
  $s = \frac{s_- p_+ - s_+ p_-}{p_+ - p_-}$
else
  $s = \frac{s_- + s_+}{2}$
end if
return $\theta_0 + s\theta^*$

the projection is negative. The maximum is located between θ_{N-1} and θ_N. By using the gradient values in these two points to interpolate, a good approximation of the maximum is found.

If the gradient's projection in point θ_1 is negative, the point θ_1 is in a downward slope and the maximum is somewhere between θ_1 and θ_0. To bracket this maximum, the step size, s_0, is halved and a new gradient is calculated at the point $\theta_2 \rightarrow \theta_0 + s_0/2\theta_0$. This is repeated until a point, θ_N, has been found where the gradient projection on the search direction is positive. The maximum is located between θ_{N-1} and θ_N and the maximum is approximated by the same fashion as before.

There are two issues with using GSEARCH: GSEARCH calls GPOMDP a few times which is a computationally expensive function and if no maximum is found, GSEARCH can return high values that seriously overshoots the target.

4.2 Structure of the Solution

The problem that will be used as an illustration of the GPOMDP algorithm is the example scenario in section 2.3. To simplify the problem and to represent the world surrounding the agent, a grid that divides the map into a network of equal-sized squares is created. A table with an entry for each square in the grid is stored. Squares where UXOs are believed to be scattered and which have not been searched are marked with 1. Already searched squares and squares that have not been bombarded with artillery are not interesting to the robots and are marked with 0,2 to separate the world outside the map from the parts of the world included.

For a square to be considered searched, a robot has to travel through it. Once the robot has travelled through the square, all the grenades are considered discovered and the square is marked with 0,2.

The robots have four possible actions. Go up, left, down and right. Each action will lead to the expected next state and reward. If a robot chooses an action that makes it leave the map, a penalty is given and the robot's position is not updated, i.e. the robot is still at the same position as it was before it chose to leave the map.

4.3 Sensor

To improve the agent's ability to generalise and to reduce the input to the neural network, the agent is using a local sensor similar to the one described in section 3.2.2. The sensor divides the map in areas centered around the agent. The values of the map squares within each area are averaged, figure 4.3. The average is used as a representation of the map within that area. Together the averages from each sensor area represents the entire map, but with a lower resolution and centered around the agent.

$$y_i = \frac{\sum_{x \in A_i} \text{map values}(x)}{\text{number of map squares in sensor area}}$$

The averages y_i will be the input to the policy function.

The map squares that lie on the diagonal lines of the sensor are divided in half by two sensor areas. Each of these areas receive half of the divided map square and the mean value calculation is modified accordingly.

All sensor areas with the same distance to the agent forms a sensor layer where each area cover the same number of map squares. The number of squares covered by a layer is decided by the layer length, l_n, figure 4.4 . Different number of layers and layer lengths have been used during the experiments to compensate for larger maps. The areas are numbered as in figure 4.4.

The disadvantage with the sensor is that it does not fully discriminate between all the possible states. For example, consider a state where the map square to the left

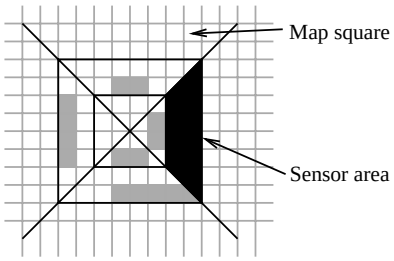


Figure 4.3: The sensor adds the values of the map square within each sensor area. The average values from each area gives a representation of the map centered around the agent.

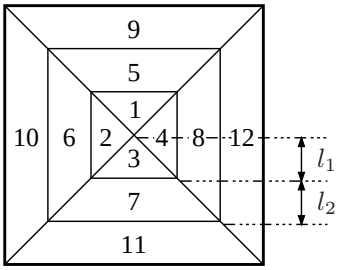


Figure 4.4: Each sensor layer has length l_n and each sensor area is numbered as in the figure.

of the agent is unknown and the squares above and under the unknown square are known. The sensor picture of this state looks the same as the picture of a state where the map square to the left is known but the squares above and under is unknown. The lack of discrimation between states can cause the variance to increase, slowing down the learning process and limiting the performance of the agent.

4.4 Policy

The agents are using stochastic policies with neural networks that translate the sensor values to probabilities to choose each action. The input to the neural network is the value in each sensor area, y_i, and the output are four values, y_k, that are divided with $\sum_k y_k$ to make sure that the sum of probabilities is one. The input vector is normalised to length 1.

4.4.1 Neural Network The neural network consists of two layers as in figure 3.5. The output layer has four nodes (one for each action) and the number of hidden nodes can be varied between experiments.

The hidden layer uses the sigmoidal activation function $y_j = \phi(v_j) = \tanh(v_j)$ which has the maximum value 1 and minimum value -1. The output layer uses the exponential activation function $y_k = \phi(v_k) = e^{v_k}$.

When using the activation functions above, the probability to pick action u_l becomes

$$\mu_{u_l}(\theta, y_i) = \frac{y_l}{\sum_k y_k}, \tag{4.7}$$

where

$$y_k = e^{v_k},$$

$$v_k = \sum_j \theta_{k,j} y_j + b_k,$$

$$y_j = \tanh(v_j),$$

$$v_j = \sum_i \theta_{i,j} y_i + b_j.$$

4.4.2 Finding $\nabla\mu/\mu$ In algorithm 1, the gradient of the policy function with respect to its parameters needs to be calculated. To find the gradient, the partial derivatives of the policy with respect to its parameters need to be found. In equation 3.8 and 3.9, these derivatives for a two layer neural network were derived. The partial derivatives with respect to the output layer parameters are

$$\frac{\partial \mu_{u_l}(\theta, y_i)}{\partial \theta_{k,j}} = \frac{\partial \mu_{u_l}(\theta, y_i)}{\partial y_k} \frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial \theta_{k,j}}$$

The partial derivatives are easy to derive using the definitions in section 4.4.1.

$$\mu_{u_l}(\theta, y_i) = \frac{y_l}{\sum_k y_k} \Rightarrow \frac{\partial \mu_{u_l}(\theta, y_i)}{\partial y_k} = \begin{cases} \frac{\sum_k y_k - y_l}{(\sum_k y_k)^2} & l = k, \\ \frac{-y_l}{(\sum_k y_k)^2} & l \neq k. \end{cases}$$

$$y_k = \phi_2(v_k) = e^{v_k} \Rightarrow \frac{\partial y_k}{\partial v_k} = e^{v_k} = y_k$$

$$v_k = \sum_j \theta_{k,j} y_j + b_k \Rightarrow \frac{\partial v_k}{\partial \theta_{k,j}} = y_j$$

$$\frac{\partial \mu_{u_l}(\theta, y_i)}{\partial \theta_{k,j}} = \begin{cases} \frac{\sum_k y_k - y_l}{(\sum_k y_k)^2} y_l y_j & l = k, \\ \frac{-y_l}{(\sum_k y_k)^2} y_k y_j & l \neq k. \end{cases} \tag{4.8}$$

The derivatives with respect to the hidden layer parameters are

$$\frac{\partial \mu_{u_l}(\theta, y_i)}{\partial \theta_{j,i}} = \sum_k \left[\frac{\partial \mu_{u_l}(\theta, y_i)}{\partial y_k} \frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial y_j} \right] \frac{\partial y_j}{\partial v_j} \frac{\partial v_j}{\partial \theta_{j,i}}$$

The partial derivatives that have not yet been derived are easy to derive using the definitions in section 4.4.1.

$$v_k = \sum_j \theta_{k,j} y_j + b_k \Rightarrow \frac{\partial v_k}{\partial y_j} = \theta_{k,j}$$

$$y_j = \phi_1(v_j) = \tanh(v_j) \Rightarrow \frac{\partial y_j}{\partial v_j} = 1 - \tanh(v_j)^2 = 1 - y_j^2$$

$$v_j = \sum_i \theta_{j,i} y_i + b_j \Rightarrow \frac{\partial v_j}{\partial \theta_{j,i}} = y_i$$

$$\frac{\partial \mu_{u_l}(\theta, y_i)}{\partial \theta_{j,i}} = \left((\sum_k y_k - y_l)\theta_{l,j} + \sum_{k \neq l} [-y_k \theta_{k,j}] \right) \frac{(1 - y_j^2) y_l y_i}{(\sum_k y_k)^2} \tag{4.9}$$

In GPOMDP, the gradient should be divided by the policy function $\mu_{u_l}(\theta, y_i)$. Dividing equation 4.8 and 4.9 with the policy returns the demanded $\frac{\nabla \mu_{u_l}(\theta, y_l)}{\mu_{u_l}(\theta, y_l)}$. The elements of the gradient for parameters of the outer layer,

$$\frac{\frac{\partial \mu_{u_l}(\theta, y_i)}{\partial \theta_{k,j}}}{\mu_{u_l}(\theta, y_i)} = \begin{cases} \frac{\sum_k y_k - y_l}{\sum_k y_k} y_j & l = k, \\ \frac{-y_k}{\sum_k y_k} y_j & l \neq k. \end{cases} \tag{4.10}$$

For the hidden layer

$$\frac{\frac{\partial \mu_{u_l}(\theta,y_i)}{\partial \theta_{j,i}}}{\mu_{u_l}(\theta,y_i)} = \left((\sum_k y_k - y_l)\theta_{l,j} + \sum_{k \neq l} [-y_k \theta_{k,j}] \right) \frac{(1-y_j^2)y_i}{\sum_k y_k} \tag{4.11}$$

According to assumption 2, equations 4.10 and 4.11 have to be bounded by $B < \infty$ for all $\theta \in \Re^K$ for GPOMDP to work properly. Since

$$\sum_k y_k - y_l < \sum_k y_k,$$
$$y_l < \sum_k y_k,$$
$$y_j = \tanh(v_j) \in]-1, 1[,$$

equation 4.10 is bounded. Since

$$\frac{\frac{\partial \mu_{u_l}(\theta,y_i)}{\partial \theta_{j,i}}}{\mu_{u_l}(\theta,y_i)} = \frac{(\sum_k y_k - y_l)\theta_{l,j}(1-y_j^2)y_i}{\sum_k y_k} + \frac{\sum_{k \neq l}[y_k \theta_{k,j}](1-y_j^2)y_i}{\sum_k y_k},$$
$$\sum_k y_k - y_l < \sum_k y_k,$$
$$y_l < \sum_k y_k,$$
$$y_j = \tanh(v_j) \in]-1, 1[\text{ and all } y_i \text{ are bounded.}$$

4.5 More Than One Agent

How the agents are to interact is not set when using reinforcement learning as an optimisation method. As in all reinforcement learning the agents have to learn the necessary cooperation in order to solve a problem as efficiently as possible.

The agents are given information about the other agents and use this information as part of the input to their policy functions. The learning is implemented in the same way as for the single agent case, but now the agents have a state space where the other agents are included.

Although they change their behaviour as they change their policies, the other agents' interaction with the environment is seen as part of the environment during the learning process.

4.5.1 Distributed Common Actor In a multi robot setting, the agent can either be central, taking decisions for all the different robots or distributed where each robot has its own agent that uses sensor information and its policy to pick an action.

One of the objectives of this thesis is to investigate a case where each agent uses the same distributed actor. Each agent will use the same parameters in their neural network, but will come to different decisions since they have different sensor data as input into the policy function.

A distributed actor is more robust in a sense that an agent can take a decision without having contact with a central agent. The actor can be made simpler since the complexity in an agent's policy does not increase as fast with respect to the number of agents as for a central agent.

Distributed actors need special care when there are problems that demand synchronisation between the different agents' actions. For example, if two cars face each other head on, it is important that the drivers both turn in the same direction, otherwise they will have an accident. Either this problem has to be solved by communication or by having a rule such as using right-hand traffic. The agents are supposed to spontaneously learn the social conventions needed during the training phase in multi agent reinforcement learning.

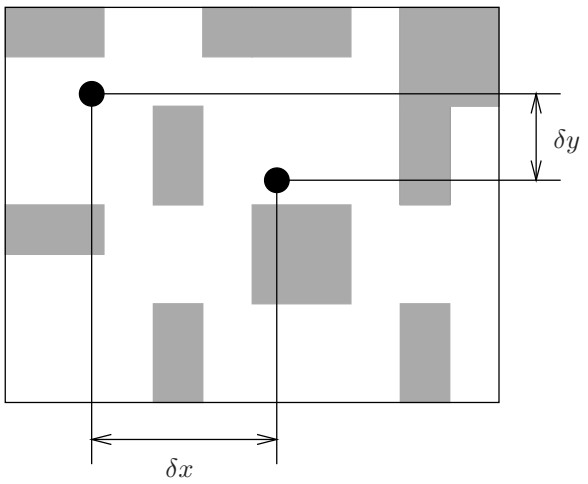


Figure 4.5: The distances δx and δy can be used to locate the other agents.

The benefits when all agents use the same actor function are that only one policy function has to be found and the solution is expandable in a sense that the actor only needs to be installed into an added agent.

One problem with having the same policy function for all agents is when two agents are at the same position with the same sensor input. Two agents with identical policy functions and identical sensor input will produce the same policy. If not special care is taken, the agents will have trouble finding decisions that separate them from each other. By communication or by having special rules for these situations, this problem might be solved.

Note that the problem with two agents in the same position is a problem that might not occur in reality. The agents would collide if they were in the same position and the best way to solve the problem might be to penalise collisions between the agents during learning.

4.5.2 Global Reward Function In the example problem, the agents' common goal is to search the area as quickly as possible. To make sure that the agent strive towards this common goal, the agents uses the sum of all agents' rewards as their reward function.

$$r_t = \sum_n r_{t,n}$$

From now on the index n represent each agent. If each agent was given it's own reward $r_{t,n}$, a semi-optimal situation where each agent tries to search as many areas as possible and not uses the advantage of the other agents could arise.

The common reward function will probably cause a larger variance to the reward since not only the agent's own, but also all the other agents' actions influence the rewards given.

4.5.3 Sensing the Other Agents The agents need to know where the other agents are located. A simple implementation is the distance δx and δy, figure 4.5. With this implementation, two extra features are needed per other agent.

If an agent is lost, or if another agent is allowed into the search, a new policy function with more parameters is needed when using the distance implementation. This counteracts the goal of having an implementation that easily can be expanded to more agents.

A suggested improvement is using a similar sensor as in section 4.3, but instead of averaging the amount of unsearched areas in each sensor area, the amount of agents per map entry is averaged.

With this implementation one policy is enough no matter how many agents that are added or removed. Adding or removing agents from the problem only increases or decreases the values in each sensor area. The number of features stay the same and the same policy function can be used if the agent has been trained for similar situations. The policy function can probably also generalise between a problem with ten and a problem with eleven agents cooperating to solve a problem.

In this thesis, only agents that use the distance to each other as sensor input have been implemented.

4.5.4 GPOMDP in a Multi Agent Setting Suppose each agent, $n \in [1, \ldots, N]$, have their own parameter set θ_n and observation of the environment y_n. As previously mentioned, each agent decides on its own action according to its policy $\mu_{u_n}(\theta_n, y_n)$. If all the agents receive the same reward signal, GPOMDP can be applied to the POMDP obtained when concatenating the observations, controls and parameters into single vectors $y = [y_1, \ldots, y_N]$, $u = [u_1, \ldots, u_N]$ and $\theta = [\theta_1, \ldots, \theta_N]$ respectively. However, an identical result would be received if GPOMDP would be applied to each agent independently and then concatenating the results $\Delta = [\Delta_1, \ldots, \Delta_N]$ [4, section 7.1].

The result is an algorithm where the agents adjust their parameters according to gradient estimates that have been calculated independently for each agent. The agents have no explicit communication, but they still strive to maximise the global reward function.

In the case considered in this thesis, all the agents have the same policy parameters, θ. Independent of which agent GPOMDP is applied to, the algorithm tries to estimate the same gradient. To make good use of the data, GPOMDP is simultaneously applied to each agent in the simulations during the learning process. The mean gradient is then calculated and used to update the policy. The following shows that the implementation is simple in the GPOMDP algorithm. The implementation is used during the multi agent simulations in this thesis.

$$\Delta_T = \frac{1}{N}\sum_{n=1}^{N}\Delta_{T,n} = \frac{1}{N}\sum_{n=1}^{N}\left[\sum_{t=0}^{T-1}\frac{\nabla_{u_{t,n}}\mu_n(\theta,y_{t,n})}{\mu_n(\theta,y_{t,n})}\sum_{s=t+1}^{T}\beta^{s-t-1}r_t\right],$$

where N is the number of agents used during the simulations.

The part of the equation inside the brackets could be written as equation 4.6, in other words, row 5 in algorithm 1 now becomes

$$\begin{gathered}\Delta_{t+1} = \frac{1}{N}\sum_{n=1}^{N}\left[\Delta_{t,n} + \frac{1}{t+1}\left[r_{t+1}z_{t+1,n} - \Delta_{t,n}\right]\right] = \\ \frac{1}{N}\sum_{n=1}^{N}\Delta_{t,n} + \frac{1}{t+1}\left[\frac{1}{N}\sum_{n=1}^{N}r_{t+1}z_{t+1,n} - \frac{1}{N}\sum_{n=1}^{N}\Delta_{t,n}\right] = \\ \Delta_t + \frac{1}{t+1}\left[r_{t+1}\frac{1}{N}\sum_{n=1}^{N}z_{t+1,n} - \Delta_t\right]. \qquad (4.12)\end{gathered}$$

Hence, it is necessary to keep a separate z for each agent, but not a separate Δ in the GPOMDP algorithm.

Chapter 5

Experiments and Results

The experiments were conducted in a simulated MATLAB environment where the solution presented in chapter 4 was implemented. First the algorithm was tested, using only one agent to search the map, then the experiments were expanded to include more agents.

The studies showed that the complexity of the problem grew as the size of the map and the complexity of the pattern to be searched on the map grew. Three different maps, figure 5.1 , of different sizes were used to see how the implementation handled problems with different complexity.

The maps are referred to by their sizes, the left map is called 3x3, the middle 6x6 and the right 8x8. When the results are presented, three figures are displayed, one for each map and in the order of figure 5.1.

5.1 Single Agent

During the single agent part of the experiments, some parameters were held constant to make the simulations comparable. The discount factor β was set to 0,95 and the initial step size of the line search s_0 was set to 0,01.

The number of hidden nodes used in the experiments for each map was respectively 12, 24 and 48.

The rewards given to the agent were

stepping out of the map	-1,0
taking a step	-0,1
searching an unknown area	0,8
reaching the terminal state	5,0

CONJPOMDP was terminated either if $\|\Delta\|^2$ was smaller than 10^{-4} or if a maximum of ten thousand policy updates had been conducted.

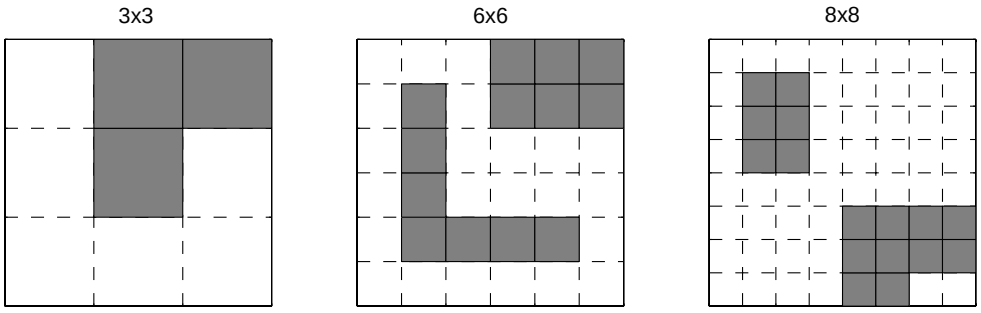


Figure 5.1: Three different maps of different sizes were used in the experiments to see how the implementation handled problems with different complexity.

If no step size had been found after four iterations in GSEARCH, no step was taken in the search direction. This prevented serious overshooting of the maximum discussed in for example [5].

Different sensors were used for the different maps. The layer most distant to the agent was always given a length that made sure that the sum of layer lengths were equal to the largest side length of the map minus one.

For the 3x3 map, the agent used a sensor with two layers, the inner layer had length l_1 of 1 and the outer layer length is decided by the previous rule. For the 6x6 map, a three-layer sensor with lengths 1 and 2 is used and for the 8x8 map, a four-layer sensor with lengths 1, 1 and 2 is used.

Ten independent simulations where the CONJPOMDP algorithm was used to find a policy were conducted for each experiment. This was done to reduce the effects on the result caused by random events during the learning process. The results shown in this chapter are averages over all ten simulations.

To evaluate the policy found, the final policy of each of the ten simulations was used in the same environment as during the learning process. For each policy, the agent searched the map a thousand times with random starting points. The rewards collected and steps needed to reach the final state were stored for comparison.

If some of the ten final policies were a lot worse than the others, these policies were discarded and the number of bad policies was noted. For the other policies, the average discounted reward collected $\eta_\beta(\theta)$, the average steps $s(\theta)$, the standard deviation of the average steps $\sigma(\theta)$ and the average number of policy updates needed to find the policy was used as a measurement of how well the algorithm performed.

The average steps were divided by the average steps taken before reaching the terminal state using an optimal policy (denoted by *% from optimal* in the result tables). The minimum possible steps to reach the terminal state were calculated for each map position, p, when p is used as a starting point. The average minimum for the entire map was then used as the average steps needed for the optimal policy. The maps have the optimal average steps (in order): 3,1111, 15,1111 and 20,125.

The second measurement is how the algorithm behaved during the learning process. For each of the simulations, the average amount of steps needed to reach the terminal state was stored before each time the policy was updated and used to create a trend plot of how the policy had improved.

To create a more readable plot, each point was not included in the plot. A distance between the points, d, on the x-axis was chosen and values at each $x_n = d/2+nd, n \in [0..N]$ were used in the plot. For each point x_n, the average steps is calculated by taking the mean value of each stored average step, $\widehat{s}$, between $\widehat{s_{nd-d/2+1}}$ and $\widehat{s_{nd+d/2}}$.

To find the standard deviation, the average steps are first filtered

$$s_{t+1} = s_t + \alpha(\widehat{s_{t+1}} - s_t),$$

where s_t is the filtered average amount of steps for policy update t and α is a constant (set to 0,1). The standard deviation at each x_n is then found by

$$\sigma_n = \sqrt{\frac{1}{d}\sum_{k=nd-d/2+1}^{nd+d/2}(\widehat{s_k} - s_k)^2}.$$

The mean values and the standard deviation is averaged over the ten simulations and normalised by

$$\text{normalised average steps} = \frac{\text{optimal steps} - \text{average steps}}{\text{steps for random policy} - \text{optimal steps}} + 1,$$

$$\text{normalised standard deviation} = \frac{\text{standard deviation}}{\text{steps for random policy} - \text{optimal steps}},$$

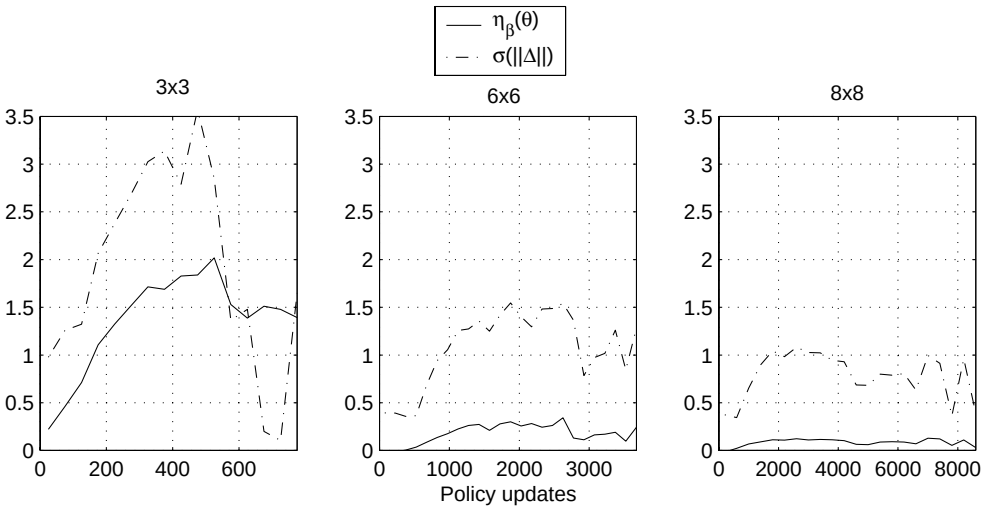


Figure 5.2: The standard deviation of the norm of the gradient approximation and the average reward as a function of the number of policy updates when not repeating the Markov decision process. Note that the variance in the gradient approximation grows as the policy improves.

where the steps for a random policy is the average steps before the first policy update in all the simulations conducted for the different algorithms.

The normalised average steps enable simple comparisons between different maps and different variations of the algorithm where one is an optimal and zero is a random policy.

The normalised average steps and standard deviation are plotted, for example in figure 5.4 , where the middle line is the average steps, the bars represent the standard deviation and the x-axis is how many times the policy has been updated in the CONJPOMDP algorithm.

5.1.1 Non-Repeating Markov Decision Process The first experiment implements the algorithm as presented in [2 , 5 , 3]. The gradient approximation of GPOMDP is considered good enough and GPOMDP is stopped when the terminal state of the POMDP has been reached, i.e. the entire map has been searched, or when a maximum number of steps has been taken.

As the policy improves, the variance in the gradient approximation, Δ , increases, see figure 5.2. The Markov process examined becomes shorter as the policy improves and fewer steps are needed to reach the terminal state. This causes the gradient approximations to become very noisy. In some cases only two steps are taken in the POMDP before the policy is updated.

The large random component in the gradient approximation makes the algorithm unstable and reduces the chances to find an optimal policy.

map size	$s(\theta)$	*% from optimal*	$\sigma(\theta)$	$\eta_\beta(\theta)$	*bad policies*	*policy updates*
3x3	3,5116	13	1,1	6,0622	3	454
6x6	18,5897	23	6,1	7,6643	6	2 638
8x8	35,2840	75	5,8	5,9194	9	6 767

5.1.2 Repeating the Markov Decision Process To reduce the variance of the gradient approximations, the POMDP repeats the search procedure until the maximum amount of steps has been reached. Each time the agent reaches the terminal

state, the estimate of the gradient, Δ, is accumulated and a new episode starts with the original map restored and with the agent in a new starting position.

The POMDP will more direct satisfy assumption 1 if the POMDP is restarted. An MDP will only have a stationary distribution if it is ergodic, i.e. irreducible and aperiodic. It is only irreducible if it, with a probability one, will return to each state when iterating an infinite amount of times. Certainly the POMDP for the non-repeating Markov decision process can return to the same state the next time GPOMDP is called and by this, the ergodic property will be stored into the neural network, but in a less direct way.

Modification of GPOMDP for Repeating Markov Processes During the derivation of equation 4.6, it was shown that each term in the gradient estimation

$$\Delta_T = \frac{1}{T}\sum_{t=0}^{T}\frac{\nabla\mu_{u_t}(\theta, y_t)}{\mu_{u_t}(\theta, y_t)}J(t+1),$$

can (for $t = n$) be rewritten as

$$\begin{gathered}\frac{\nabla\mu_{u_n}(\theta,y_n)}{\mu_{u_n}(\theta,y_n)}J(n+1) = \frac{\nabla\mu_{u_n}(\theta,y_n)}{\mu_{u_n}(\theta,y_n)}(r_{n+1}+\beta J(n+2)) = \\ = \frac{\nabla\mu_{u_n}(\theta,y_n)}{\mu_{u_n}(\theta,y_n)}(r_{n+1}+\beta r_{n+2}+\beta^2 J(n+3)) = \cdots .\end{gathered}$$

In equation 4.6 it was shown that this sum lead to the relation

$$z_{t+1} = \beta z_t + \frac{\nabla\mu_{u_t}(\theta, y_t)}{\mu_{u_t}(\theta, y_t)}.$$

When the terminal state occurs, unbiased Monte Carlo estimates of the value functions $J(t)$ have been found. If the POMDP is restarted, it is important that the rewards following the restart not are added to the value functions for states occurring before the restart.

To avoid adding too many rewards to the value function, the gradient approximations for each repetition of the POMDP are added separately. If A is the number of times the POMDP is restarted, a is the episode number and T_a is the steps taken in episode a, then

$$\Delta_T = \frac{1}{T}\sum_{a=1}^{A}\left[\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta, i_{t,a})}{\mu_{u_{t,a}}(\theta, y_{t,a})}\sum_{s=t+1}^{T_a}\beta^{s-t-1}r(i_{s,a}).\right] \tag{5.1}$$

In a similar fashion to the derivation of equation 4.6, the GPOMDP approximation of the gradient is

$$\begin{aligned}\Delta_T = \tfrac{1}{T}\Bigg[&\underbrace{\frac{\nabla\mu_{u_{0,1}}(\theta, y_{0,1})}{\mu_{u_{0,1}}(\theta, y_{0,1})}}_{z_{1,1}} r_{1,1} + \underbrace{\left(\beta\frac{\nabla\mu_{u_{0,1}}(\theta, y_{0,1})}{\mu_{u_{0,1}}(\theta, y_{0,1})} + \frac{\nabla\mu_{u_{1,1}}(\theta, y_{1,1})}{\mu_{u_{1,1}}(\theta, y_{1,1})}\right)}_{z_{2,1}} r_{2,1} + \dots \\ &\dots + \underbrace{\left(\beta^{T_1-1}\frac{\nabla\mu_{u_{0,1}}(\theta, y_{0,1})}{\mu_{u_{0,1}}(\theta, y_{0,1})} + \dots + \frac{\nabla\mu_{u_{T_1-1,1}}(\theta, y_{T_1-1,1})}{\mu_{u_{T_1-1,1}}(\theta, y_{T_1-1,1})}\right)}_{z_{T_1,1}} r_{T_1,1} + \\ &+ \underbrace{\frac{\nabla\mu_{u_{0,2}}(\theta, y_{0,2})}{\mu_{u_{0,2}}(\theta, y_{0,2})}}_{z_{1,2}} r_{1,2} + \cdots\Bigg].\end{aligned} \tag{5.2}$$

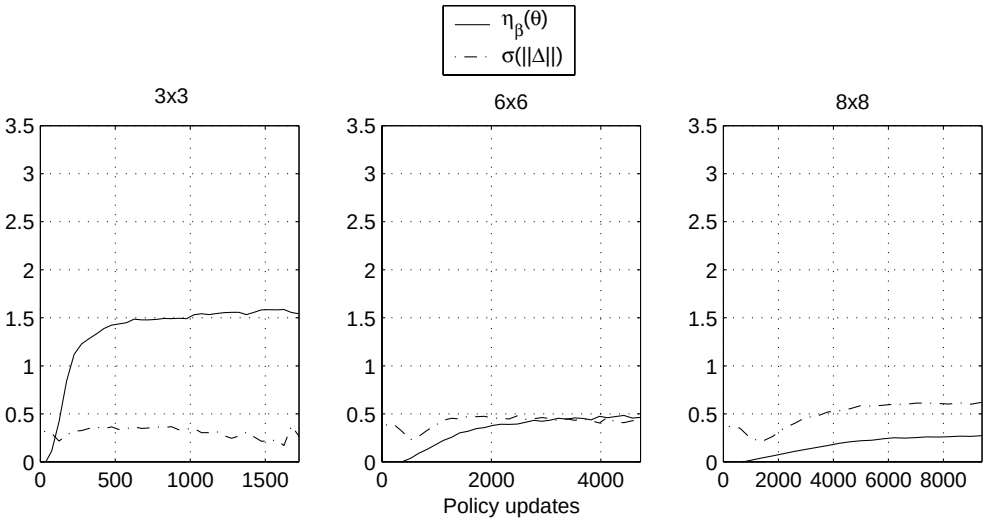


Figure 5.3: The standard deviation of the norm of the gradient approximation and the average reward as a function of the number of policy updates when repeating the Markov decision process. Note that the variance in the gradient approximation has been reduced and does not grow as the policy improves.

Equation 5.2 implies that z has to be set to zero every time the POMDP is restarted to receive unbiased estimates of the value functions.

Result with Repeating POMDP During the simulations for this and all the following single agent experiments, a hundred steps were taken in the POMDP before the policy was updated. The number of steps was not increased with increasing map size and problem complexity.

The reason for repeating the POMDP was to decrease the variance in the gradient approximations, Δ. This was also achieved, which can be observed in figure 5.3.

The average steps taken in each episode of GPOMDP is plotted in figure 5.4 as described in section 5.1.

Note that the variance is much lower in the left figure, which illustrates the experiments with the 3x3 map. The larger problems have increasing complexity and the variance can cause the algorithm to loose a well working policy for a bad one. In these figures, it is not possible to see that the large variance can cause the algorithm to choose very bad policies, especially when the policy parameters reach high values.

Bad, almost deterministic policies are especially dangerous. Any almost deterministic policy, even a bad one, can cause CONJPOMDP to terminate, with the risk of returning a bad policy as a result.

When an almost deterministic policy has been found, a change in the parameters in the direction towards the likely action will not cause a significant difference and hence $\nabla\mu_{u_t}(\theta, y)$ will become small. If the agent due to the stochastic policy picks an action which is less likely, the gradient will grow, especially because of that $\nabla\mu_{u_t}(\theta, y)$ should be divided by $\mu_{u_t}(\theta, y)$. But if the policy is close to deterministic, no unlikely action might be taken during a GPOMDP run and, if so, the gradient approximation returned will be very small. The criteria used for an optimum and to terminate CONJPOMDP is a small gradient and hence CONJPOMDP can return any almost deterministic policy found.

In figure 5.5, the norm of the gradient approximation is plotted in the same plot as the average reward during the learning phase. The norm of the gradient grows as

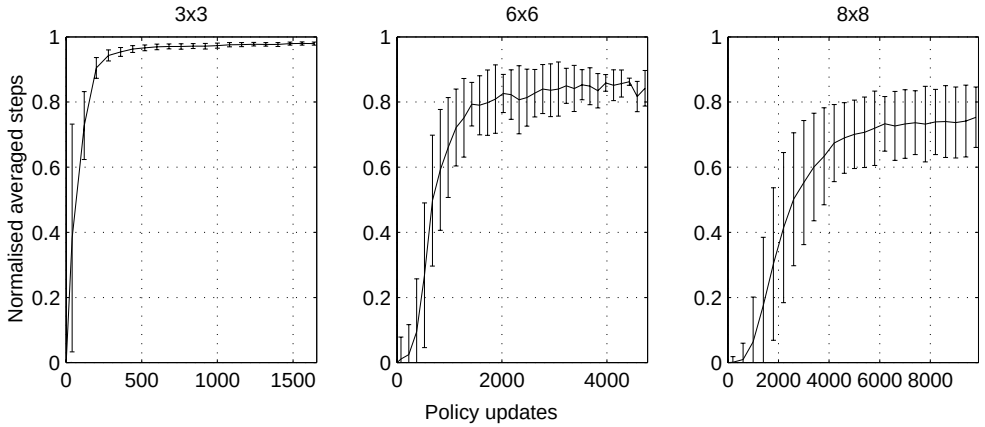


Figure 5.4: Convergence behaviour for the CONJPOMDP algorithm when the POMDP is repeated during GPOMDP run, plotted as described in section 5.1.

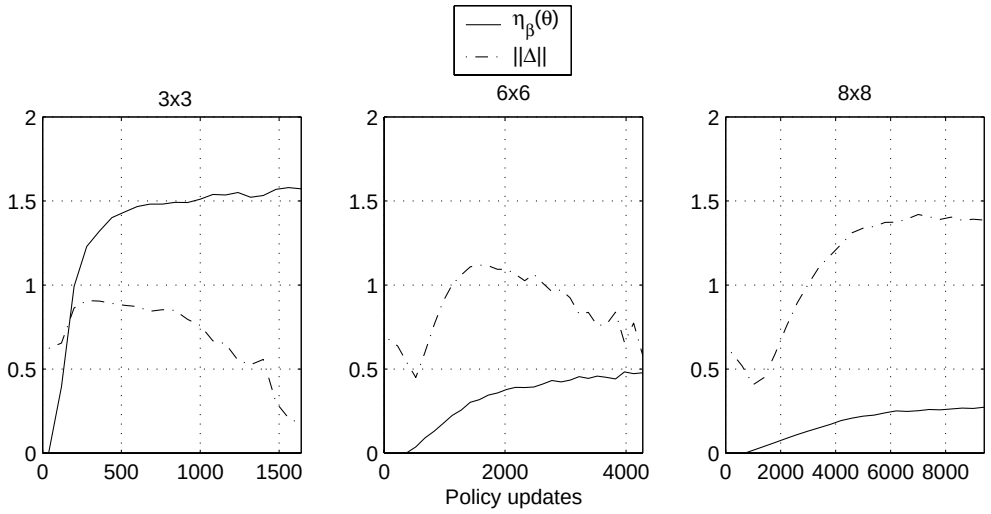


Figure 5.5: The average reward received by the agent and the norm of the gradient approximation as a function of the number of policy updates when repeating the POMDP. Note that the norm is the smallest at the start of the algorithm in the larger map cases.

the policy improves and the gradient does not become small until very close to the optimum point.

A large value of the gradient approximation norm combined with noisy gradient approximations is dangerous. The gradient approximation norm determines the size of the step taken in the parameter space when the policy is updated. As the parameters of the neural network grows, almost deterministic policies become more likely and it is more likely to find a random, almost deterministic policy. A step size that is reduced when the parameters grow might be needed.

map size	$s(\theta)$	*% from optimal*	$\sigma(\theta)$	$\eta_\beta(\theta)$	*bad policies*	*policy updates*
3x3	3,1905	3	0,8	6,1743	0	1 360
6x6	15,9797	6	3,1	8,3064	0	4 183
8x8	26,5576	32	7,0	7,3429	0	10 000

In the 8x8 case, the policy has been updated the maximum number of times allowed. The calculations were not stopped since a small gradient had been reached, but since a predetermined number had been reached. This might be the reason for the large deterioration of the result for the large map.

5.1.3 Modifications of the Reward Functions The reward function has a large effect on the gradient estimation since it is an important part of equation 5.1. A badly chosen reward function increases the size of the gradient estimation and reduces the stability of the conjugate gradient search. If the absolute value of the reward given to the agent grows when the policy improves, the norm of the gradient approximation will grow.

An attempt to reduce the growth of the gradient norm is to subtract the average reward to the rewards given in the learning phase.

$$\Delta_T = \frac{1}{T}\sum_{a=1}^{A}\left[\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta, i_{t,a})}{\mu_{u_{t,a}}(\theta, y_{t,a})}\sum_{s=t+1}^{T_a}\left[\beta^{s-t-1}r(i_{s,a}) - \eta_\beta(\theta)\right]\right], \tag{5.3}$$

where $\eta_\beta(\theta)$ is the discounted reward per step. The change in the gradient approximation will keep the mean reward at zero and make sure that $\frac{\nabla\mu_{u_t}(\theta,y_t)}{\mu_{u_t}(\theta,y_t)}$ applies the major effect on the gradient approximation norm and the size of change in the policy.

Subtracting the Average Reward from the Previous GPOMDP Run A simple method of implementing the modification of the reward function is to calculate $\eta_\beta(\theta)$ during a GPOMDP run and using it as an approximation of $\eta_\beta(\theta)$ for the next run. Even though the policy and $\eta_\beta(\theta)$ has changed, the change is small and most of the time, so is the error in this approximation. However, due to the non-linear property of the environment and the non-linear property of the policy function, a small change in the policy weights can cause large changes in $\eta_\beta(\theta)$.

In figure 5.6 it is shown that subtracting the average reward reduces the growth of the gradient approximation norm.

The average steps taken in each episode of GPOMDP is plotted in figure 5.7 as described in section 5.1.

map size	$s(\theta)$	*% from optimal*	$\sigma(\theta)$	$\eta_\beta(\theta)$	*bad policies*	*policy updates*
3x3	3,1265	0,5	0,8	6,1986	0	640
6x6	15,6208	3	1,5	8,3485	1	3 117
8x8	23,4456	17	5,0	7,7796	1	10 000

As in section 5.1.2, the maximum number of policy updates were reached in the 8x8 case and the result was highly deteriorated.

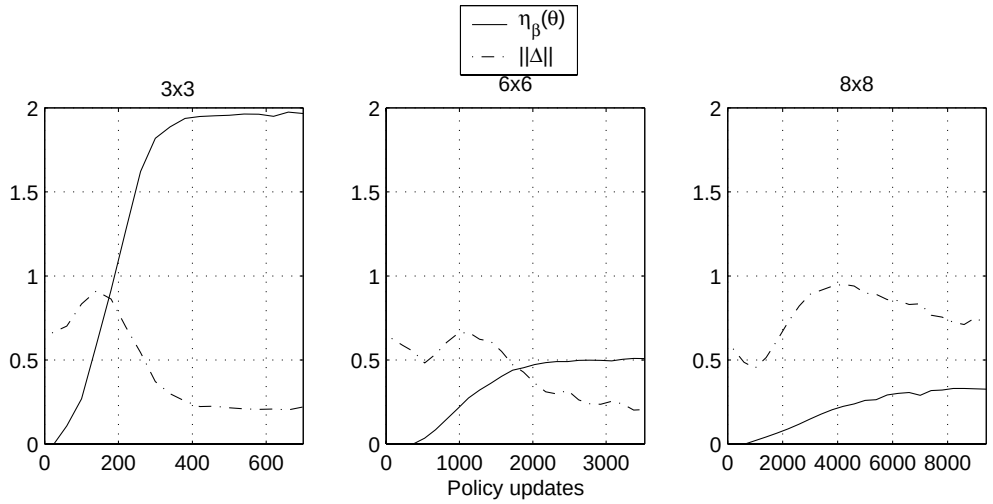


Figure 5.6: The average reward received by the agent and the gradient approximation norm as a function of the number of policy updates when subtracting the average reward. Note that the norm of the gradient approximation is reduced as the policy improves.

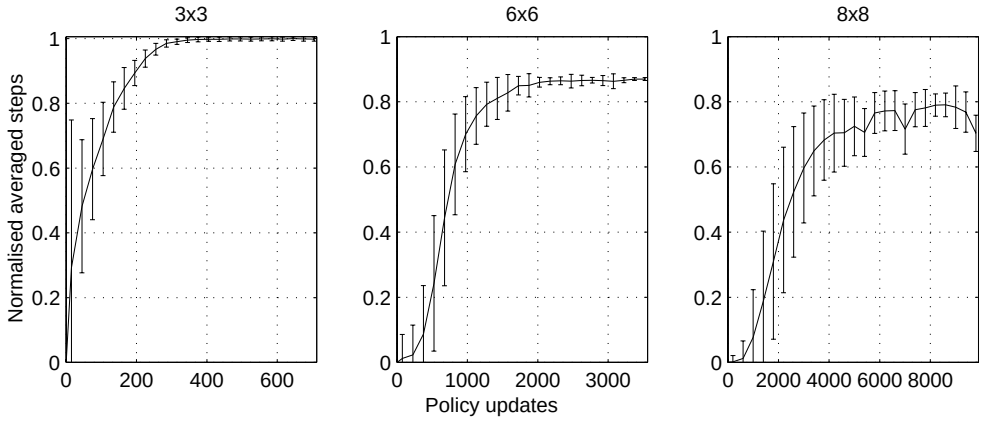


Figure 5.7: The progress of the agent's learning when subtracting the average reward from the previous GPOMDP run, plotted as described in section 5.1.

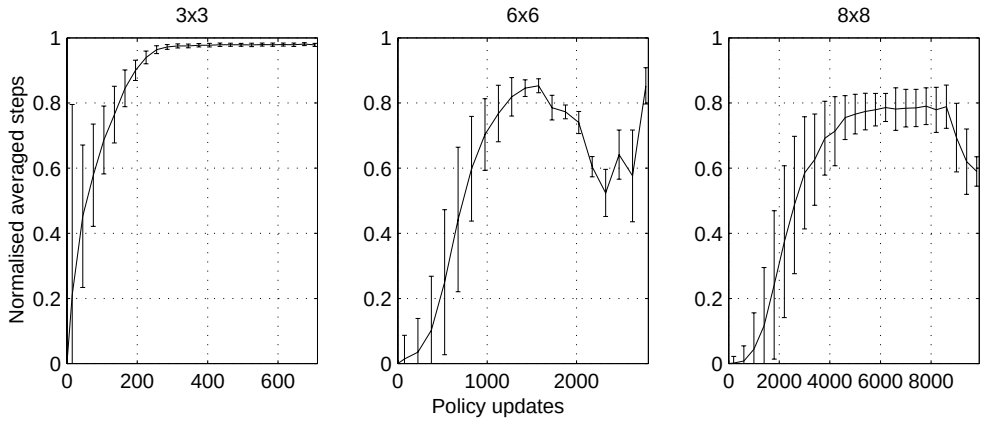


Figure 5.8: The progress of the agent's learning when subtracting the average reward calculated during the same GPOMDP run, plotted as described in section 5.1.

Subtracting the Non-Biased Average Reward Using the average reward of the previous policy as an estimate of $\eta_\beta(\theta)$ causes a bias on the gradient estimation. This can be removed by subtracting the average reward of the current policy.

One obstacle is that $\eta_\beta(\theta)$ is subtracted in each iteration in equation 5.3, but not known until the last iteration of the GPOMDP run. A modification where $\eta_\beta(\theta)$ is not subtracted until after the last iteration of GPOMDP is derived, starting from equation 5.3.

$$\Delta_T = \frac{1}{T}\sum_{a=1}^{A}\left[\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta,i_{t,a})}{\mu_{u_{t,a}}(\theta,y_{t,a})}\sum_{s=t+1}^{T_a}\left[\beta^{s-t-1}r(i_{s,a})-\eta_\beta(\theta)\right]\right] =$$
$$= \frac{1}{T}\sum_{a=1}^{A}\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta,i_{t,a})}{\mu_{u_{t,a}}(\theta,y_{t,a})}\sum_{s=t+1}^{T_a}\beta^{s-t-1}r(i_{s,a}) - \frac{1}{T}\sum_{a=1}^{A}\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta,i_{t,a})}{\mu_{u_{t,a}}(\theta,y_{t,a})}\sum_{s=t+1}^{T_a}\eta_\beta(\theta).$$

Focusing on the right hand side of the previous equation

$$\frac{1}{T}\sum_{a=1}^{A}\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta,i_{t,a})}{\mu_{u_{t,a}}(\theta,y_{t,a})}\sum_{s=t+1}^{T_a}\eta_\beta(\theta) = \{\eta_\beta(\theta)\text{ is a constant} =\}$$
$$= \frac{1}{T}\sum_{a=1}^{A}\sum_{t=0}^{T_a-1}\eta_\beta(\theta)\frac{\nabla\mu_{u_{t,a}}(\theta,i_{t,a})}{\mu_{u_{t,a}}(\theta,y_{t,a})}(T_a-t) =$$
$$= \frac{1}{T}\eta_\beta(\theta)\sum_{a=1}^{A}\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta,i_{t,a})}{\mu_{u_{t,a}}(\theta,y_{t,a})}(T_a-t) =$$
$$= \frac{1}{T}\eta_\beta(\theta)\sum_{a=1}^{A}T_a\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta,i_{t,a})}{\mu_{u_{t,a}}(\theta,y_{t,a})} - \frac{1}{T}\eta_\beta(\theta)\sum_{a=1}^{A}\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta,i_{t,a})}{\mu_{u_{t,a}}(\theta,y_{t,a})}t,$$

$$\Delta_T = \frac{1}{T}\sum_{a=1}^{A}\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta,i_{t,a})}{\mu_{u_{t,a}}(\theta,y_{t,a})}\sum_{s=t+1}^{T_a}\beta^{s-t-1}r(i_{s,a}) -$$
$$-\frac{1}{T}\eta_\beta(\theta)\sum_{a=1}^{A}T_a\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta,i_{t,a})}{\mu_{u_{t,a}}(\theta,y_{t,a})} + \frac{1}{T}\eta_\beta(\theta)\sum_{a=1}^{A}\sum_{t=0}^{T_a-1}\frac{\nabla\mu_{u_{t,a}}(\theta,i_{t,a})}{\mu_{u_{t,a}}(\theta,y_{t,a})}t. \tag{5.4}$$

This equation can be implemented, storing four extra vectors in the GPOMDP algorithm.

The average steps taken in each episode of GPOMDP is plotted in figure 5.8.

map size	$s(\theta)$	*% from optimal*	$\sigma(\theta)$	$\eta_\beta(\theta)$	*bad policies*	*policy updates*
3x3	3,1466	1	0,8	6,1955	0	618
6x6	15,8327	5	2,4	8,3343	0	2 491
8x8	22,4580	12	3,6	7,9653	3	10 000

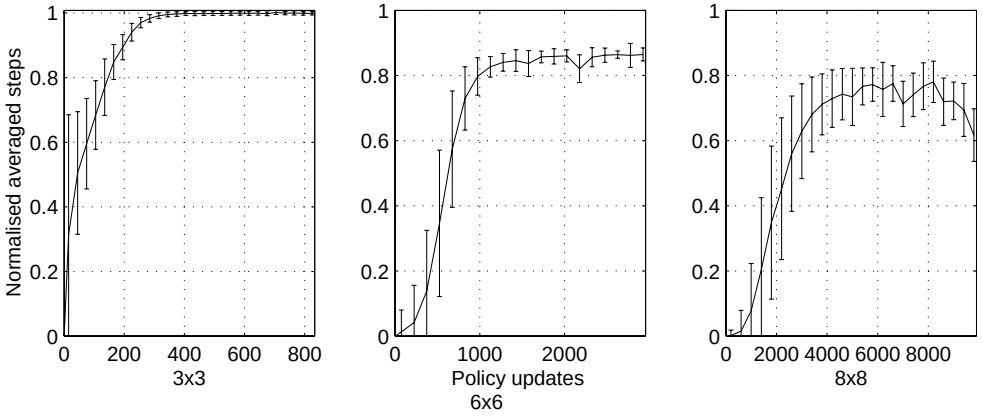


Figure 5.9: The progress of the agent's learning when subtracting the average reward from the previous runs of GPOMDP with a low pass filter, plotted as described in section 5.1.

As before, the maximum number of policy updates were reached in the 8x8 case and the result was highly deteriorated.

Subtracting the Average Reward with Low Pass Filter Using the approximation of $\eta_\beta(\theta)$ from one GPOMDP run can cause a noisy $\eta_\beta(\theta)$. A solution with bias but with less noise is using an approximation of $\eta_\beta(\theta)$ averaged over the previous GPOMDP runs. To prioritise rewards given to policies close to the current, a low pass filtered reward is subtracted.

$$\eta_\beta(\theta_{t+1}) = \eta_\beta(\theta_t) + \alpha \left(\widehat{\eta_\beta(\theta_{t+1})} - \eta_\beta(\theta_t) \right),$$

where $\widehat{\eta_\beta(\theta_{t+1})}$ is the average reward per step in GPOMDP iteration $t+1$ and the constant α is set to 0,1.

The average steps taken in each episode of GPOMDP is plotted in figure 5.9 as described in section 5.1.

map size	$s(\theta)$	*% from optimal*	$\sigma(\theta)$	$\eta_\beta(\theta)$	*bad policies*	*policy updates*
3x3	3,1255	0,5	0,8	6,1972	0	677
6x6	15,9047	5	2,7	8,3622	1	2 864
8x8	23,0316	14	4,0	7,8475	3	10 000

As before, the maximum number of policy updates were reached in the 8x8 case and the result was highly deteriorated.

5.1.4 Summary of Single Agent Experiments The experiments have shown that the algorithm can find well working policies in the single agent case. The best experiments have come to a mean policy that takes an average of 1%, 5% and 12% more steps to search the map than an optimal policy.

For the largest, 8x8, map, the search for an optimal policy was stopped since the maximum number of policy updates was reached and not since a small gradient was found. This might be the reason for the large deterioration in result with increasing map size.

More reliable convergence and better solutions for the larger problems might be found by changing β or by taking more steps in the Markov decision process before the policy is updated. However, the time it takes for the agent to find an optimal

3x3 map						
optimal steps: 3.1111		*% from*			*bad*	*policy*
algorithm	$s(\theta)$	*optimal*	$\sigma(\theta)$	$\eta_\beta(\theta)$	*policies*	*updates*
not repeated MDP	3,5116	13	1,1	6,0622	3	454
repeated MDP	3,1905	3	0,8	6,1743	0	1 360
subtracting previous $\eta_\beta(\theta)$	3,1265	0,5	0,8	6,1986	0	640
subtracting current $\eta_\beta(\theta)$	3,1466	1	0,8	6,1955	0	618
subtracting filtered $\eta_\beta(\theta)$	3,1255	0,5	0,8	6,1972	0	677
6x6 map						
optimal steps: 15.1111		*% from*			*bad*	*policy*
algorithm	$s(\theta)$	*optimal*	$\sigma(\theta)$	$\eta_\beta(\theta)$	*policies*	*updates*
not repeated MDP	18,5897	23	6,1	7,6643	6	2 638
repeated MDP	15,9797	6	3,1	8,3064	0	4 183
subtracting previous $\eta_\beta(\theta)$	15,6208	3	1,5	8,3485	1	3 117
subtracting current $\eta_\beta(\theta)$	15,8327	5	2,4	8,3343	0	2 491
subtracting filtered $\eta_\beta(\theta)$	15,9047	5	2,7	8,3622	1	2 864
8x8 map						
optimal steps: 20.125		*% from*			*bad*	*policy*
algorithm	$s(\theta)$	*optimal*	$\sigma(\theta)$	$\eta_\beta(\theta)$	*policies*	*updates*
not repeated MDP	35,2840	75	5,8	5,9194	9	6 767
repeated MDP	26,5576	32	7,0	7,3429	0	10 000
subtracting previous $\eta_\beta(\theta)$	23,4456	17	5,0	7,7796	1	10 000
subtracting current $\eta_\beta(\theta)$	22,4580	12	3,6	7,9653	3	10 000
subtracting filtered $\eta_\beta(\theta)$	23,0316	14	4,0	7,8475	3	10 000

Table 5.1: Summary of the experiments with single agents.

policy is already long for these small problems. If the algorithm is to find solutions to continuous or more complex problems, an improvement in speed has to be found.

A comparison between the different modifications tried in this section is presented in table 5.1. It shows that repeating the Markov decision process before updating the policy improves the performance of the policy found, but subtracting the average reward only improves the solution slightly or not at all. The needed policy updates decreases when the average reward is subtracted but more bad policies are found.

One way of reducing the time and improve the stability might be to change the termination criteria for CONJPOMDP. The criterion with a small gradient norm is time costly and can be triggered for bad policies. A criterion where either a reached goal average reward or a converged reward function terminates the algorithm might improve the algorithms reliability and reduce the computational cost and still produce well working policies.

5.2 Two Agents

During the experiments with two agents, the same algorithm as in section 5.1.2 was used, i.e. the Markov decision process was repeated but no average reward was subtracted. Subtracting the average reward does not improve the algorithm significantly and repeating the POMDP has for example been done by Baxter and Bartlett [5, section 4.2] and should provide an interesting comparison.

The parameters, sensors and maps of the single agent simulations were held constant, section 5.1, with the following exceptions: The number of hidden nodes for each map was respectively 16, 32 and 64. The steps taken in the GPOMDP algorithm before the policy is updated is for each map 100, 200 and 400.

The results accomplished are shown in tables but $\eta_\beta(\theta)$ is not comparable between each experiment and is not shown. The average number of steps needed is a better

measurement since it is comparable between experiments and what is supposed to be minimised in the scenario. The average number of steps is divided by the optimal average number of steps for a single agent as a measurement of the time saved when using two agents instead of one. Note that the value used, $s(\theta)$, is actually time steps, i.e. one step in the two agent case means two agents taking one step each. The measurement is denoted by *% of single agent* in the result tables.

A new column, indicating the total amount of steps taken by the agents during the learning phase, is added to the result table. The number, denoted by *steps during learning*, is a measurement of the amount of calculations needed to find the policy and is used during the discussion of how calculation effort increases with increasing map size.

The average number of steps during the learning phase is plotted as discussed in section 5.1. However, the normalisation is changed to

$$\text{normalised average steps} = \frac{0,5 \cdot \text{optimal steps for single agent} - \text{average steps}}{0,5 \cdot \text{optimal steps for single agent}} + 1.$$

The normalisation results in a plot where the value one represents 50% of the average number of steps that an optimal single agent would need to search the map and zero represents 100%. The standard deviation is normalised in the same fashion.

5.2.1 Terminating the Conjugate Gradient Search In algorithm 2 it is suggested that the conjugated gradient search should be iterated until $\|\Delta\|^2$ is small enough. When the norm is small enough, a local optimum is assumed and this is accepted as the policy.

During the simulations, it seems that long before the norm of the gradient decreases, the reward function has almost reached it's maximum. As the agent gets closer to it's maximum, the algorithm becomes more unstable and since CONJPOMDP terminates when the gradient is small, bad policies might be returned, see page 33.

A second criterion for terminating the search is when a chosen amount of steps have been taken. However, there is a chance that after the chosen number of steps, variance has caused a temporary bad policy.

A third criterion is to see if the average reward between updates of the policies has converged. If the changes in the mean average reward have been small during the last policy updates, the algorithm terminates independent of the norm of the gradient.

During the experiments with two agents, all three criteria have been used to terminate CONJPOMDP. The first one fulfilled, usually a converged reward function, terminates the algorithm.

5.2.2 Two Independent Agents During the first simulation the agents have no knowledge of each other except that the other agent's actions influence the map of the problem. The agents are given the rewards caused by their own actions. In other words, these are two single agents trained in an environment where two robots searches the area.

The convergence behaviour is plotted in figure 5.10.

map size	$s(\theta)$	*% of single agent*	$\sigma(\theta)$	*bad policies*	*policy updates*	*steps during learning*
3x3	2.1972	71	0.82836	0	2 858	804 965
6x6	12.8587	85	3.8241	0	2 065	1 048 960
8x8	18.1709	90	6.2436	0	4 307	3 019 980

5.2.3 Agents Receiving a Local Reward In the second experiment, the other agent's position relative to the agent is added to the policy function's input. Both

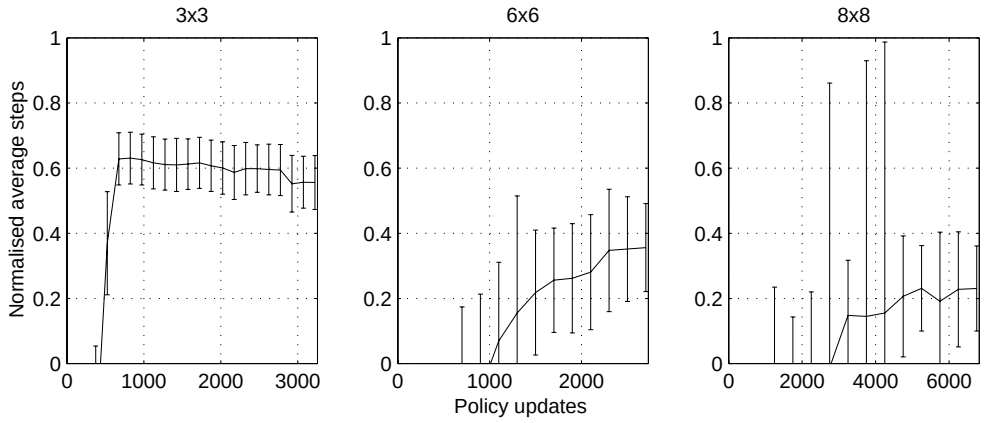


Figure 5.10: The progress of the agent's learning when two independent agents search the area, plotted as described in section 5.2.

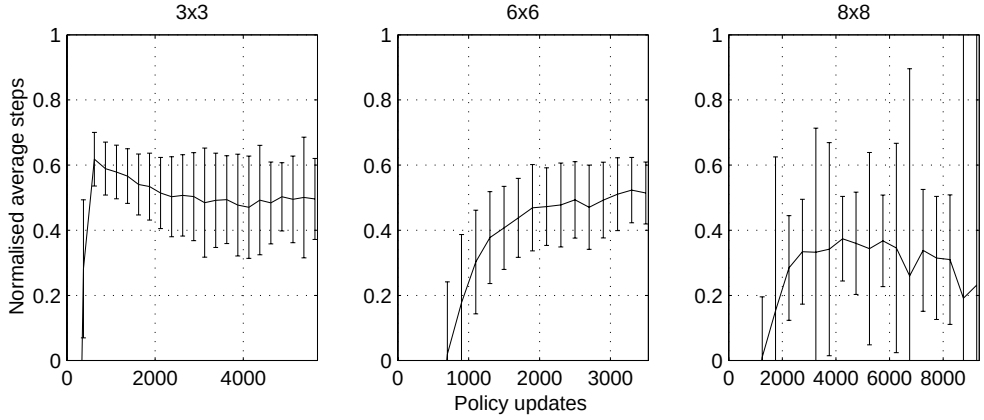


Figure 5.11: The progress of the agent's learning when only local rewards are given to the agents, plotted as described in section 5.2.

agents still receive reward for their own actions. The agent that searches the last area is rewarded with the terminal state reward. If two agents search the same area at the same time, they share the given reward.

There is a risk that the agent will not strive towards a common goal but will try to counteract the other agent so that they have the chance of getting as much reward as possible for themselves.

The convergence behaviour is plotted in figure 5.11.

map size	$s(\theta)$	*% of single agent*	$\sigma(\theta)$	*bad policies*	*policy updates*	*steps during learning*
3x3	2.2743	73	1.082	0	3 576	954 040
6x6	11.4599	76	3.3501	0	2 643	1 330 340
8x8	17.754	88	5.831	0	8 650	6 041 020

5.2.4 Agents Receiving a Global Reward In the third experiment, the input to the policy function is the same as in section 5.2.3, but the rewards given to the agents are added together to a global reward, see section 4.5.2. The agents will try to maximise the total reward and could perform actions that are not optimal for

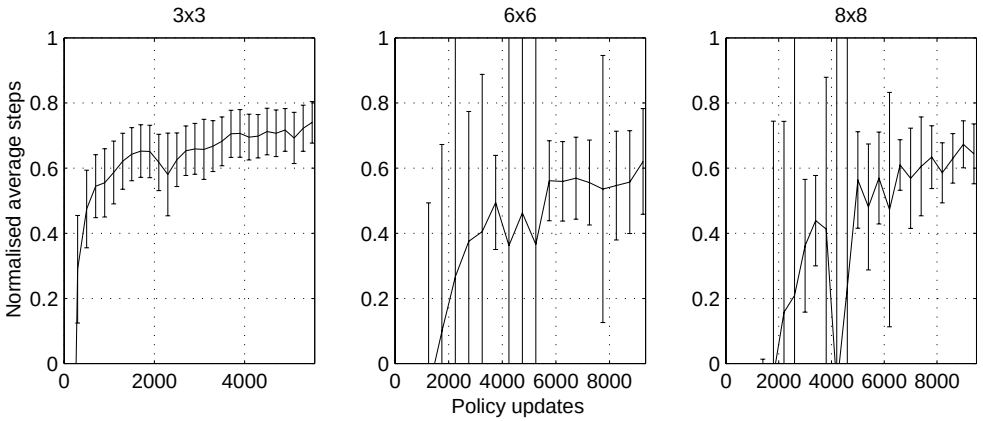


Figure 5.12: The progress of the agent's learning when global rewards are given to the agents, plotted as described in section 5.2.

themselves but might be optimal for the group.

The rewards are independent of who that performs the successful (or not successful) action. The variation will probably be larger since a good action can receive negative reward if the other agent at same time picks a bad action.

The convergence behaviour is plotted in figure 5.12.

map size	$s(\theta)$	*% of single agent*	$\sigma(\theta)$	*bad policies*	*policy updates*	*steps during learning*
3x3	2.0239	65	0.77097	0	4 080	1 018 780
6x6	11.1330	74	3.0432	0	5 050	2 576 142
8x8	14.6862	73	4.9414	0	5 290	3 691 220

5.2.5 Combination of Global and Local Reward To reduce the variation when using the global reward and to still achieve a cooperating behaviour, the agents are given a combination of global and local reward. For each of the agent's actions, it is given the subsequent reward. When the terminal state has been reached, both agents receive a terminal state reward. The input to the policy function includes the other agent's relative position as in section 5.2.3

If both agents are given the terminal state reward, they will strive to search the map as quickly as possible since the discounting factor causes the final reward to become less valuable for each step. To maximise the value function the agents need to search the map as quickly as possible.

Since both agents receive a local reward for their own action, the variation will be reduced. An action will lead to its consequence except for when the terminal state has been reached.

The convergence behaviour is plotted in figure 5.13.

map size	$s(\theta)$	*% of single agent*	$\sigma(\theta)$	*bad policies*	*policy updates*	*steps during learning*
3x3	2.0134	64	0.75913	0	2 083	536 870
6x6	10.6546	71	3.5539	0	4 537	2 265 100
8x8	14.1348	70	3.5884	0	4 403	3 081 620

5.2.6 Scalability of the Algorithm An important measurement of the algorithm is how well it handles increasing problem complexity. How much more calculations are needed when a large problem is to be solved compared to a small one?

To provide a measurement, three different map sizes have been used during the experiments and the number of squares in each map is compared to the calculation

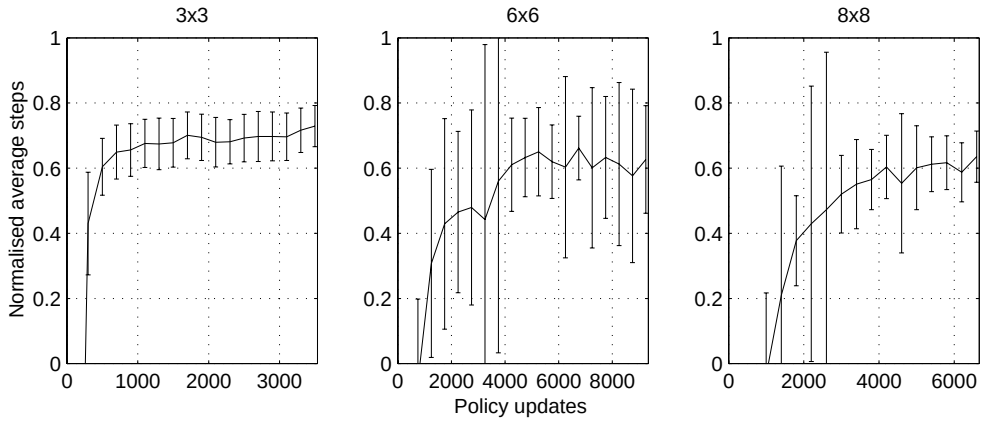


Figure 5.13: The progress of the agent's learning when given a combination of the global and the local reward, plotted as described in section 5.2.

effort needed to find the solution. The total number of steps taken in the map during the learning phase is used as a computer independent measurement of the calculations needed to find an efficient solution.

The number of steps taken in the map during the learning phase, t, is divided by the number of squares in each map, m and by m^2 to study if the calculation effort grows linearly or quadratic.

	Two single agents			*Local reward*		
m	t	t/m	t/m^2	t	t/m	t/m^2
9	804 965	89 441	9 938	954 040	106 000	11 778
36	1 048 960	29 138	809	1 330 340	36 954	94 391
64	3 019 980	47 187	737	6 041 020	94 391	1 475
	Global reward			*Combined reward*		
m	t	t/m	t/m^2	t	t/m	t/m^2
9	1 018 780	113 200	12 580	536 870	59 652	6 628
36	2 576 142	71 559	1 988	2 265 100	62 919	1 748
64	3 691 220	57 675	901	3 081 620	48 150	752

The values implies that the calculations needed to find an efficient policy is linearly dependant on the number of squares in the map that is to be searched. However, the amount of data is limited and other factors, such as the number of hidden nodes and the number of steps in the map before the policy is updated, could affect the result.

5.2.7 Summary of the Two Agent Experiments The experiments with two agents have shown that the agents perform better when informed of the other agent's location. Two agents that did not have information of the other fulfilled the search mission, for each map respectively, using 71%, 85% and 90% of the steps that an optimal single agent would need. Note that one step means each agent taking one step. Agents that had information about the other did the same using 73%, 76% and 88% of the steps.

The experiments have also shown that the agents perform better when both agents strive to maximise a global reward function, instead of a local one. Two agents that received the sum of both agents' rewards fulfilled the mission, for each map respectively, using 65%, 74% and 73% of the steps that one agent would need.

To reduce the large variance that occurs when a global reward function was used during the learning phase, a combination of global and local rewards were used. The agents were given rewards for their own action, but both agents received the terminal state reward. Somewhat surprisingly, the policy found outperformed the policy found

Experiment	*3x3 map*	*6x6 map*	*8x8 map*
Optimal single agent	100	100	100
Two single agents	71	85	90
Local reward	73	76	88
Global reward	65	74	73
Combination reward	64	71	70

Table 5.2: A summary over all two agent experiments with a comparison to an optimal single agent search. The table values are the time steps needed to search the map using the resulting policies from each experiment divided by the time steps needed for an optimal single agent. The values have been called *% of single agent* during the two agent experiment.

with a global reward function. The agents fulfilled the mission using 64%, 71% and 70% of the steps that one agent would need, the performance of the resulting policies of each different experiment is summarized in table 5.2.

The anticipated result with the combination reward had been a quicker convergence but a slightly worse final policy. Why the result was reached has not been fully investigated but possible reasons are:

- A behaviour where the two agents repelled each other was noted in the global reward case. This behaviour lead to cases where a single unsearched square was in between the two agents but it took a while until an agent searched it.
- The training phase did not iterate until the norm of the gradient was small. Due to the lower variance of the local and global reward case, a better policy had been reached when the algorithm was terminated.
- The problem itself did not need a sophisticated coordination to be solved. The difference between receiving local and global rewards or only global rewards was not strong enough.
- Reaching the terminal state at the shortest amount of time was encouraged in both cases. Every step costs a penalty of -0.1 and the terminal state reward is discounted for each step until the terminal state is given.

Chapter 6

Conclusions

The thesis evaluated a reinforcement learning algorithm within the policy search class. In the evaluation, the algorithm was to produce a policy that guided agents to the shortest search path through a map.

The algorithm chosen, GPOMDP, was first evaluated in a single agent environment. Three different maps with different sizes were used during the evaluation to see how the algorithm handled increasing complexity. Different modifications to the algorithm were made to improve the algorithm's speed and stability. The best result was reached when taking a chosen number of steps in the search process before updating the policy and subtracting the average discounted rewards from the rewards received during the search. The average policy searched the map using 1%, 5% and 12% more steps than an optimal policy would (starting with the smallest map).

Hence, an optimal policy was not found, but for each map except the largest, the policy was near optimal. For the largest map, the learning algorithm was stopped due to time limitations instead of assumed optimality. This partly explains the large deterioration in result.

The thesis also evaluated how the algorithm could be expanded to two agents and if the algorithm could find an efficient policy for them. By giving the agents a global reward, i.e. a reward consisting of the sum of all the agents' instantaneous rewards, the agents were made to cooperate towards a common goal.

Four different experiments were conducted with agents having different properties: Independent agents with no knowledge of the other's position; agents with knowledge of the other's position, given rewards only based on their own actions; agents given the total rewards created by both agents; agents given rewards based on their own actions and the rewards given when the entire map was searched.

The best results were found with agents having the last set of properties, rewards given based on their own actions and a global terminal state reward. With the resulting policies, each respective map was searched using 64%, 71% and 70% of the steps that an optimal single agent would need. Note that one step means that each agent has taken a step. As a comparison, the policies found with agents that only were given rewards based on their own actions, searched the map using 73%, 76% and 88% of the steps that an optimal single agent would need.

During the two agent experiments, the amount of calculations seemed to increase linearly with respect to the number of squares in the search map. However, the amount of data is small and other factors could affect the calculation time.

The evaluation shows that the GPOMDP algorithm can find near optimal policies for the search problems explored. It also shows that a global reward function improves the performance of a two-agent solution.

The most important problems occurring in the simulations are slow convergence and problems with stability of the algorithm. Suggestions on how these problems can be solved in future work are given in chapter 7.

Chapter 7

Continued Work

7.1 Reinforcement Learning

Optimising the Algorithm There are a few parameters that can be set to optimise the GPOMDP algorithm's stability and performance. The effects of changing these parameters have only briefly been investigated during this project. Parameters that can be changed are for example:

- The discounting factor, β, sets the balance between a noisy or a biased gradient.
- The sensor layers' lengths, l_n, set the agent's resolution of the environment.
- The number of iterations in GPOMDP before the policy is updated sets the balance between a noisy gradient approximation and computational cost.
- The reward function influences the behaviour of the algorithm.
- The number of hidden nodes in the neural network sets the balance between the resolution of the agents' policies and the storing and computational cost.
- The initial step size, s_0, sets the balance between having an unstable or a slow algorithm.

Another area of interest is how to terminate the CONJPOMDP algorithm. Trying to reach a maximum of the policy function in the parameter space has some problems. The algorithm can become unstable when getting close to the maximum and it is a computationally costly target. In this thesis, the convergence of the expected discounted rewards or a maximal number of iterations have been used as examples of termination criteria.

Other Algorithms An equal comparison to other policy search algorithms would be beneficial. Examples of algorithms of interest include the PIFA algorithm [14], the ATPG algorithm [7] and the OLPOMDP [5].

Only minor changes of the Matlab code used in this thesis would be needed to implement the OLPOMDP algorithm.

Expanding the Problem By increasing the size of the map and the number of agents in the problem, the complexity would rise. This would increase the variance and set higher demands on the algorithm, but is needed to model more realistic problems. The final goal would be continuous action and state spaces.

7.2 Modelling of Multi Agent Systems

Sensing Other Agents The implementation presented here has used the distance to other agents to sense where the other agents are located. As discussed in section 4.5.3, a better solution might be to implement a sensor, similar to the one that filters the global map, that detects the other agents. The distance implementation

is problematic if an agent is lost or added. The change would improve the agents' ability to generalise problems with different amount of agents.

Limiting Knowledge In many applications for unmanned vehicles, the communication link between the agents is limited. The agents are not likely to have complete knowledge of what the other agents have explored and discovered.

If the knowledge about the map was limited but could be updated by special actions, an interesting scenario where the agents would have to communicate to solve the problem might occur.

Bibliography

[1] Lars Axelson. Reinforcement learning for missile control. Technical Report FOA-R–98-00916-314–SE, Swedish Defence Research Establishment (FOA), November 1998.

[2] Jonathan Baxter and Peter L. Bartlett. Direct gradient-based reinforcement learning: I. Gradient estimation algorithms. Technical report, Research School of Information Sciences and Engineering, Australian National University, July 1999. URL `http://citeseer.nj.nec.com/250374.html`.

[3] Jonathan Baxter and Peter L. Bartlett. Reinforcement learning in POMDP's via direct gradient ascent. In *Proc. 17th International Conf. on Machine Learning*, pages 41–48. Morgan Kaufmann, San Francisco, CA, 2000. URL `http://citeseer.nj.nec.com/baxter00reinforcement.html`.

[4] Jonathan Baxter and Peter L. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001. URL `http://www-2.cs.cmu.edu/afs/cs/project/jair/pub/volume15/baxter01a.pdf`.

[5] Jonathan Baxter, Lex Weaver, and Peter L. Bartlett. Direct gradient-based reinforcement learning: II. Gradient ascent algorithms and experiments. Technical report, Research School of Information Sciences and Engineering, Australian National University, July 1999. URL `http://citeseer.nj.nec.com/baxter99direct.html`.

[6] Erik Berglund. Unmanned ground vehicles. *Militärteknisk tidskrift*, 3:18–20, 2001.

[7] Gregory Z. Grudic and Lyle H. Ungar. Localizing policy gradient estimates to action transitions. In *Proc. 17th International Conf. on Machine Learning*, pages 343–350. Morgan Kaufmann, San Francisco, CA, 2000. URL `http://citeseer.nj.nec.com/grudic00localizing.html`.

[8] Mark Hewish. GI, robot. *Jane's International Defense Review*, pages 34–40, jan 2001.

[9] V. Konda and J. Tsitsiklis. Actor-critic algorithms, 2000. URL `http://citeseer.nj.nec.com/434910.html`.

[10] Martin Lauer and Martin Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *Proc. 17th International Conf. on Machine Learning*, pages 535–542. Morgan Kaufmann, San Francisco, CA, 2000. URL `http://citeseer.nj.nec.com/lauer00algorithm.html`.

[11] Leonid Peshkin, Kee-Eung Kim, Nicolas Meuleau, and Leslie Pack Kaelbling. Learning to cooperate via policy search. In *Proceedings of the Sixteenth International Conference on Uncertainty in Artificial Intelligence.*, 2000. URL `http://citeseer.nj.nec.com/peshkin00learning.html`.

[12] J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. URL `http://www.cs.cmu.edu/$\sim$quake-papers/painless-conjugate-gradient.ps`. August 1994.

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. The MIT Press, 1998.

[14] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, volume 12, pages 1057–1063. MIT Press, 2000. URL `ftp://ftp.cs.umass.edu/pub/anw/pub/sutton/SMSM-NIPS99-submitted.pdf`.

[15] *Joint Robotics Program Master Plan*. U.S. Department of Defence, 2001. URL `http://www.jointrobotics.com`.

[16] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992. URL `http://citeseer.nj.nec.com/williams92simple.html`.