CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Multi Agent Reinforcement Learning

Master's thesis in Mathematical Statistics

Rikard Isaksson

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Thesis for the degree of Master of Science

Multi Agent Reinforcement Learning

RIKARD ISAKSSON

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURGH
Gothenburgh, Sweden 2019

Multi Agent Reinforcement Learning
RIKARD ISAKSSON

Supervisors: Jonas Hellgren, Volvo GTT, VA
Marina Axelsson-Fisk, Department of Mathematical Sciences
Examiner: Johan Jonasson, Department of Mathematical Sciences

Master's Thesis 2019
Department of Mathematical Sciences
Chalmers University of Technology
University of Gothenburgh
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Abstract

Machine learning and artificial intelligence has been a hot topic the last few years, thanks to improved computational power the machine learning framework can now be applied to larger data sets. Reinforcement learning is a group of machine learning algorithms where one does not know the correct answer in advance, much like unsupervised learning. However, in contrast to unsupervised learning, the quality of a decision can be measured as a number. By trail and error a program can learn to find the optimal decisions to take based on this measure. The reinforcement learning framework has shown to find solutions to complex problems in confined game environments and control systems such as balancing tasks and bipedal walking. With reinforcement learning, usable solutions or strategies have been found to many problems which in theory could be solved to optimality but in practice are intractable. The success with reinforcement learning in games such as Chess, Backgammon and Go are examples of such strategies [11].

A problem with reinforcement learning in general is the so called curse of dimensionality. As the problem gets more complex, it naturally takes longer for the program to learn and the computational time often grows quickly with the complexity of the problem. The issue with scalability translates to reinforcement learning systems with multiple agents and new issues arise concerning the learning in terms of stability of a solution.

In this thesis we present three algorithms which attempts to tackle the issue with stability of solutions in systems with cooperating or competing agents. The algorithms minimax Q, Nash Q and win or learn fast are presented and implemented on a set of selected problems and the algortihms performance is discussed. We also discuss the scalability and make an attempt at interpreting the assumptions in these algorithms in order to draw conclusions about their applicability to real world problems.

Contents

Acknowledgments

I would like to express my sincere gratitude for everyone who has supported me throughout this thesis and some of you deserve a special mention. First, I would like to thank Marina Axelsson-Fisk, at Chalmers University of Technology, for taking time to follow my work continuously, guiding me through this thesis and supporting my decisions.

I am also grateful to Jonas Hellgren, at Volvo Group Trucks Technology, for giving me the opportunity to write this thesis, supporting me with his advice and the resources necessary to complete my thesis. I would also like to thank all my colleges at Volvo M1:7 for who got me installed and kept me company during my time at Volvo.

Finally, I must express my gratitude to my family and my partner for the continuous support and encouragement they have given me throughout my studies. This thesis would certainly not have been possible without them.

Background

Automated system are becoming larger and more complex, the problem of organising and adapting large systems are in need of more versatile solutions. Real time planning and scaling of methods used today are facing problems with large automated systems. New methods with a decentralised approach which attempts to break down the problem in smaller pieces are raising in popularity to distribute computational power and dependency throughout a system. If a system can continue to operate at a sub optimal level even though parts of it are disabled and still perform close to optimal when operating at full capacity, it would make a system more robust to disturbances. The reinforcement learning framework can be broken down to a decentralised model naturally by letting parts of the system act and learn independently.

Multi agent reinforcement learning has raised in popularity and some methods recently developed show promising results. One advantage of multi agent reinforcement learning is that the units can solve task given to them without the need for detailed instructions from a central control tower. This can relieve the need for high performance communication and active monitoring of each working unit in the system. Potential usage areas for this type of decentralised control system could be in surveilance or search and rescue missions. In a search and rescue mission small units can work independently and cooperate to search large areas and report back to an operator once the target is found. With todays advancements in image analysis and classification the mission can be almost fully automated. Another area of application is logistics system where each individual unit could potentially accept and execute a mission from a mission planner, without details on which path to take.

Objectives

This thesis aims to survey the field of multi agent reinforcement learning and some of the methods used in multi agent reinforcement learning. The goal is to review some of the strengths and weaknesses introduced with modeling reinforcement learning problems as multi agent systems. A number of selected problems will be implemented, compared and discussed. The problems are formulated with inspiration form litterature and are considered to be benchmark problems in reinforcement learning.

The concepts used in multi agent reinforcement learning builds on the theory and concepts used in single agents reinforcement learning. Therefore, the theory on single agent reinforcement learning will be presented as well as some of the difficulties that arise in the single agent reinforcement learning framework. One objective of the thesis is to see how these difficulties translate and behave in the multi agent framework but also to investigate if any new difficulties arise.

1 Introduction

Robotics and automation is being implemented more widely and the need for more versatile systems is growing. To program a machine or system to solve a specific task, such as navigating along a predefined path, can be a challenge in itself and the program is often only good for navigating this known path. In contrast, with the reinforcement learning framework one can create programs that can solve an entire class of problems. The goal is to teach a machine to navigate any of the paths in search for a given goal. When the goal is found, the machine should also learn to find the shortest path to this goal. By letting the program explore the problem and measure the quality of an attempt to solve the task, one can have the program optimise the solution.

In applications today complex autonomous units need to work where communication is limited and take decisions on their own. Examples of applications are in the survey industry, search and rescue missions or discovery missions. Several units could spread out and explore while communicating with each other to complete the mission efficiently, without the need for detailed instructions from a human controller. This would let one person search through enormous areas with the help of multiple drones that report back only if they find something of interest.

Machine learning is often separated into three categories, depending on how the machine learns. There is supervised learning and unsupervised learning with reinforcement learning somewhere in between. In supervised learning, data is fed to the machine and the correct answer is already known during learning. The machine adapts its model to the correct answer in an attempt to approximate a function that maps the input data to the correct answers. In unsupervised learning the correct answers are not known beforehand. When the machine is fed with data the machine tries to cluster the data based on some algorithm, trying to classify each data point to some group by distinguishing patterns in the input data.

Reinforcement learning is a category of machine learning in between supervised and unsupervised learning. The idea originates from how animals learn when receiving feedback as a result of some decision. Often there is no known best decision for each situation so supervised learning does not work while at the same time there is some idea of which direction to go. If there is a way to measure the quality of a decision one can instead compare decisions with each other to find the optimal decision for each given situation. In reinforcement learning the feedback gained is a real number, which is used to evaluate the decisions. The machine is referred to as an agent whose objective is to evaluate the feedback gained in order to improve on its strategy to select the optimal decision in each situation.

The theory in reinforcement learning is built up around concepts from different disciplines. This thesis aims to present the core concepts in reinforcement learning, first for single agent systems and then for systems with multiple agents. Some methods for solving problems with reinforcement learning for single agent systems and for multi agent systems will also be presented. A few selected problems will then be solved to compare the methods to some extent.

In the second chapter we present the definitions and measures used for evaluation in the reinforcement learning algorithms. The third chapter is an introduction to single agent reinforcement learning and a brief discussion about the difficulties these methods face. The forth chapter is the main chapter in this thesis where the main algorithms and methods for multi agent systems are presented. The methods presented in chapter four are also implemented in chapter five. Chapter six contains a brief presentation of how the results are measured. Lastly the results are presented and discussed together with some ideas for future work in chapter seven to ten.

2 Mathematical concepts

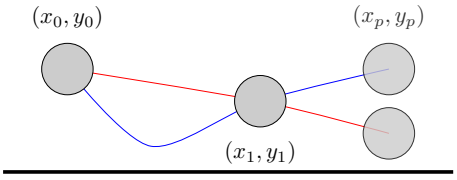
In this section we introduce some of the definitions used to model a reinforcement learning problem. We first discuss Markov chains and Markov decision processes. Following these definitions, some theory regarding solutions of Markov decision processes and the methods that serve as a foundation to reinforcement learning are presented. With a solution to a Markov decision process one refers to a probability distribution over decision, or a sequence of deterministic decisions. An optimal solution is a probability distribution over decisions which results in the highest possible total reward for a simulation. The reader is assumed to be familiar with Markov processes, basic probability theory and standard methods for optimisation. Should one be unfamiliar with expectations, conditional expectations, stochastic processes or probability distributions a good resource for rehersing these concepts can be found in [4].

2.1 Markov decision processes

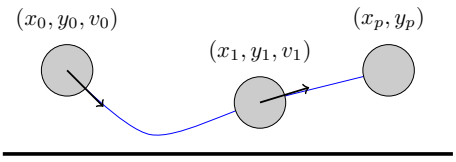
The environment in a reinforcement learning problem is often formulated in terms of a Markov process. A Markov process is characterized by the memory less property that states that the future of a stochastic process is independent of its past, given the present. The Markov property is convenient in implementation since the input to the model will be of the same type and dimension in each iteration. The input to the model, or the representation of the environment for a given time, is called a state. To utilize the Markov property it is important that the representation of the environment contains enough information to distinguish between different states. If the representation of the environment is not detailed enough, it can be hard to distinguish different states from each other.

To illustrate this, consider a model of a bouncing ball as illustrated in Figure 1. Knowing the position of the ball at a given time would not be enough for predictions on where the ball will be in the next time step. One needs to know the previous position as well in order to estimate the trajectory of the ball. If one also has a vector giving the current velocity and its direction, one has the information required for making predictions. In reinforcement learning, a sufficient and compact description of the environment which possesses the Markov property can be a tricky thing to formulate.

Even though some examples used in this report have continuous variables for time and state, these will be modeled as discrete. This is done by making a countable partition on the set of states or the set of time. In this report we will only consider stochastic processes that are discrete in time and state space. A stochastic process which possesses the Markov property and is discrete in time and state is called a Markov chain.



(a) State include positions only, even if one knows the previous position, the prediction may not be certain.



(b) State includes position and current speed with direction. Information is sufficient for unambiguous prediction.

Figure 1: Comparison of different representation of states.

Extending the Markov chain definition we can define a Markov decision process as essentially a Markov chain with some additional properties for decision making, one of which is the reward

function. Another property included in the Markov decision process is a set of actions for each state. Given a state, the agent can select an action from the set of actions and then receive a reward based on the reward function. The reward function maps each state and action pair to a numeric reward, the reward gained is used by an agent to evaluate the action taken.

Definition 2.1 (Markov Decision Process) *A Markov decision process X_t, $t \in T$, for some index set T, is a collection $\{S, A, P, r, \gamma\}$ where*

S**:** *is the state space, containing all possible settings of the environment.*

A**:** *is the action space, containing all possible decisions for each state.*

$P : S \times A \times S \to [0, 1]$**:** *is the transition probability of jumping to some state s', given some state s and action a.*

$\gamma_t : T \to [0, 1]$**:** *is the discount factor, weighting the rewards impact on a decision. The discount factor may or may not be constant for each $t \in T$.*

$r : S \times A \times S \to \mathbb{R}$**:** *is the reward function, mapping each given combination of state s, action a and the next state s' to a real number.*

When solving a Markov decision process one wants to find a policy that describes the optimal decision to make in each state. The policy is a function $\pi : S \times A \to [0, 1]$ that for each state and action pair gives a probability for selecting that action, in the given state. Thus, by the law of total probability, it holds that $\sum_{a \in A} \pi(s, a) = 1$ for each state $s \in S$. The rewards are used to update the policy that the agent is using to make decisions. The policy is an agents probability distribution over actions, which is not the same as the transition probabilities. The transition probabilities is the probability of the next state, given the current state and the selected action. Often the transition probability is deterministic such that for any given state and action pair, the probability is 1 for some following state and 0 for any other following state. To illustrate the difference between policy and the transition probabilities, consider one takes the action to flip a coin. The outcome of the action is the result of the coin flip, that is which side of the coin that is facing up. A deterministic transition probability would be to pick the side in advance and lay the coin flat. When the transition probaiblity is not one (flipping the coin), the result of the agents action is not always the same in the given state. In order to evaluate its decision, the agent need to consider the expected reward from the two possible results, heads or tails. That is, the agent not only has to estimate the reward of an action but also the transition probability. Deterministic transition probabilities makes it easier for the agent to find an optimal policy in each state since the agent does not have to estimate the transition probabilities. The result of an action will always be the same.

We denote the immediate reward at time $t + 1$ as r_{t+1}, this is an observation of a reward or an outcome. The function $r : S \times A \times S \to \mathbb{R}$ depends on the policy and the transition probabilities. Given a state X_t and an action Y_t, the reward gained depends on the resulting state X_{t+1}. To ease the notation we will drop the X_{t+1} from the argument so that $r(X_t, Y_t) = r(X_t, Y_t, X_{t+1})$. When the immediate reward is denoted by $r(X_{t+1}, Y_{t+1})$, it is to clarify the dependence on the policy and the transition probabilities [6]. When the immediate reward is denoted r_{t+1} or $r(s, a, s')$, it is considered an observation. Given a state s, an action a and a next state s', the reward function $r(s, a, s')$ is a given number. Otherwise it is random and will be denoted with random variables as $r(X, Y)$ where X is the next state and Y is the action taken in the state.

The agent in a Markov decision process has as its objective to maximise its expected future reward by finding a policy that produces as large expected future rewards as possible. For some time window $\{0, \ldots, N\}$ we have that the sum of future rewards is defined as $\sum_{k=0}^{N} \gamma^k r(X_{t+k+1}, Y_{t+k+1})$.

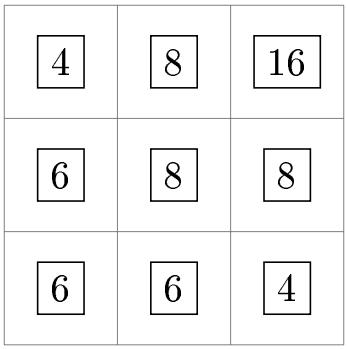


Figure 2: Values of different positions propagate through the grid. Here the state in the top right corner has a reward of 16 while the rest have a reward of 0. Given a uniform policy with two possible action (move up or right). The value function for each state is influenced by the policy and the reward in other states. In this example the discount factor is 1 and the time horizon is one step only, the value is then the mean value of the possible next positions.

The sum of future rewards can be expressed in different ways depending on the discount factor γ and the time window N. For finite N, the future rewards is said to be of **finite horizon**. If one considers the limit as $N \to \infty$, the future rewards is said to be of **infinite horizon**. For infinite horizon future rewards it is often assumed that $\gamma < 1$ to give some guarantee that the sum converges. With $\gamma \in (0, 1)$, the importance of rewards which are distant in time are scaled down gradually. The scaled down future rewards are often referred to as the discounted future rewards.

In order to evaluate a given policy one can consider a function which maps each state to some value, called the value function. The value function is defined in terms of the future rewards from a given state, which then depends on the policy and the transition probabilities. Thus, the value of one state is influenced by neighbouring states. By changing the policy in a state, an agent can control by how much to let neighbouring states influence a given state. This is due to the value function dependence on the future rewards which in turn depends on future actions and states, this is illustrated in Figure 2.

Definition 2.2 (Value Function) *Given a Markov decision process with state space S and $s \in S$. The value function $V_\pi(s)$ for an agent with policy π, is defined as the expectation*

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{N} \gamma^k r(X_{t+k}, Y_{t+k}) | X_t = s \right].$$

One can also evaluate the policy by the state-action value function which is the expected reward given some state s and an action a.

Definition 2.3 (Action value function) *Given a Markov decision process with state space S and action space A with $s \in S$ and $a \in A$. The action value function $Q_\pi(s, a)$ for an agent with policy π, at an arbitrary time t is defined as the expectation*

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{N} \gamma^k r(X_{t+k}, Y_{t+k}) | X_t = s, Y_t = a \right]$$

In some sense, the action value function is the value of the next state, given an action a. Although more accurately the action value function is the value of some action, given a state. Using the

action value function, one can construct a probability measure over the actions available in one state that reflects the quality of each action. One can also pick the action that has the highest value for one state of the action value function. When using this latter, deterministic, rule for selecting action, it is required to let the agent explore the state space by deviating from the deterministic rule with some probability. Often this is done by selecting an action on random. Otherwise the agent will never select actions that have a low action value and will never investigate solutions which include this state action pair.

2.2 Solving Markov Decision Processes

A solution to a Markov decision process is a policy or a sequence of decisions that gives the highest reward. For a path finding problem, going from point A to point B in some environment, a solution would be what is often called the shortest path in optimisation. The shortest path does not have to be the shortest in terms of distance, as the term suggests, it could for example be the path which consumes the least amount of fuel or the fastest route. It depends on how the rewards, or the objective function, is formulated.

Using the state value function one can derive a recursive relation, using backward induction. The idea is to consider the last decision made in the process and walk backwards through the chain of decisions. One can express the value of the second last state in terms of the last state and continue in this fashion one gets an expression of each state in terms of the neighbouring states. This concept is used in many popular algorithms for solving an MDP and belongs to a family of algorithms called dynamic programming.

The following recursive relation of the value function is called the Bellman equations, introduced in [1]. Using infinite horizon value functions one can consider the sum of future rewards from the next state s_{t+1} by letting $r(X_{t+1}, Y_{t+1})$ be outside of the summation sign. Factoring out γ and using linearity of the expectation we get the value function of the next state as one term. The other term in the expression is the expectation of $r(X_{t+1}, Y_{t+1})$. Conditioning on the two probability distributions P and π to expand $\mathbb{E}[r(X_{t+1}, Y_{t+1})]$ we can factor out π, P and r. The end result is an expression of the state value function in terms of the values of neighbouring states.

$$
\begin{aligned}
V_\pi(s) &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r(X_{t+k+1}, Y_{t+k+1}) \Big| X_t = s \right] \\
&= \mathbb{E}_\pi \left[r(X_{t+1}, Y_{t+1}) + \gamma \sum_{k=0}^{\infty} \gamma^k r(X_{t+k+2}, Y_{t+k+2})) \Big| X_t = s \right] \\
&= \sum_{a \in A} \pi(s,a) \sum_{s' \in S} P(s,a,s') r(s,a,s') + \gamma \, \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r(X_{t+k+2}, Y_{t+k+2}) \Big| X_{t+1} = s' \right] \qquad (1) \\
&= \sum_{a \in A} \pi(s,a) \sum_{s' \in S} P(s,a,s') \left(r(s,a,s') + \gamma \, \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r(X_{t+k+2}, Y_{t+k+2}) | X_{t+1} = s' \right] \right) \\
&= \sum_{a \in A} \pi(s,a) \sum_{s' \in S} P(s,a,s') \left(r(s,a,s') + \gamma V_\pi(s') \right)
\end{aligned}
$$

The recursion has a unique solution for each policy π. To optimise an MDP, one needs to find a policy π such that the corresponding value function $V_\pi(s) \geq V_{\pi'}(s)$ for each $s \in S$ and for each other policy π'. A value function with this property will henceforth be refered to as an **optimal value function**. The corresponding policy will be refered to as an **optimal policy**.

Methods for finding an optimal policy iteratively are called value iteration and policy iteration methods, introduced by [1]. In similar fashion, one can derive a recursive relation for the action value function $Q_\pi(s,a)$.

$$Q_\pi(s,a) = \sum_{s' \in S} P(s,a,s') \left(r(s,a,s') + \gamma \sum_{a' \in A} \pi(s',a') Q_\pi(s',a') \right)$$

The policy for decision making can be updated using the value function which in turn can be estimated iteratively, using the reward function. Often the transition probabilities are deterministic, meaning that for some following state s_1 the transition probability $P(s,a,s_1) = 1$. For any other following state s_2, by the law of total probability, we have $P(s,a,s_2) = 0$. When P is deterministic, the implementation of value iteration and policy iteration is simplified since one only has to consider one of the terms in the sum over states. The following algorithm is guaranteed to converge in limit[12] and in implementation of the algorithm, some type of criterion for when to stop is needed. In most literature the stopping criterion is set $|V^{i+1} - V^i| < \delta$ for some small but arbitrary real number δ. The starting point for V is also set arbitrarily, often to the zero vector or a vector of ones. Since the value function and the action value function both depends on the rewards in neighbouring states, it would seem reasonable to initiate them to the most common discounted reward in the system.

Algorithm 1: Value iteration

Data: Initialise $V(s)$ as arbitrary real numbers for each state s
Result: Optimal value function
while *Stop criterion not true* **do**
  for $s \in S$ **do**
    $V^{i+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s,a,s')[r(s,a,s') + \gamma V^i(s')]$
  i++

Value iteration converges in limit to the optimal value function which can be used to derive an optimal solution to the problem. In policy iteration, presented in Algorithm 2, ones solves a system of linear equations and then updates the policy for each state. If the policy is considered unchanged in one iteration, based on δ, the algorithm terminates.

Algorithm 2: Policy iteration

Data: Initialise $\pi(s)$ as uniform distribution over actions
Initialise $V(s)$ as arbitrary numbers for each state s
Result: Optimal policy function
while *Stopping criterion not true* **do**
  solve system of linear equations
  $V^{i+1}(s) = \sum_a \pi(s,a) \sum_{s' \in S} \gamma P(s,a,s')(r(s,a,s') + \gamma V^i(s')$
  i++
  for $s \in S$ **do**
    Select the action with highest value
    $\pi(s,a) = \begin{cases} 1 \text{ if } a = \operatorname{argmax}_{a \in A} \sum_{s' \in S} \gamma P(s,a,s')(r(s,a,s') + \gamma V^i(s')) \\ 0 \text{ else} \end{cases}$

The above assignment of the policy might seem cumbersome but in practice it is very simple. The idea is to set probability one for the action which yields the highest value, based on the value function that was updated in the same iteration with respect to i. If the state and action

space is finite, so is the number of possible deterministic policies. Together with the fact that policy iteration improves the policy in each iteration, the computation time is bounded since there are finite possibilities to consider. However, in policy iteration one needs to solve a system of linear equations which makes policy iteration slower then value iteration. One could also use value iteration combined with policy iteration to approximate the solution to the system of linear equations. It is not clear which of value iteration and policy iteration is the best method. Both Algorithms 1 and 2 have inspired to the iterative methods used in reinforcement learning [12].

2.3 Q-Tables and the curse of dimensionality

If the set of actions and the set of states are both finite, the action value function can be expressed as a table called a Q-table. A Q-table is then a table where each row represents a state in which the system can be. The row consists of values for each possible action which the agent can take when in this state. The Q-table is then the action value function in Definition 2.3 and gives a measure of how good an action is, in a given state. These values can be estimated using simulation, for an estimate to be reliable each state-action pair must be observed a sufficient number of times. This is one of the main drawbacks with reinforcement learning since it quickly becomes intractable to maintain the Q-table as the size of the problem increases. Often the size of the state space grows exponentially when properties are added to the model.

One of the main factors which increases the state space is the number features present in the model. A feature in the model is a dynamic object in the state representation, meaning it has to be represented in more than one way. If an object does not have more then one representation in the state space, we will not consider the object a feature. As an example suppose we have a vehicle in traffic. Suppose further that a traffic lights crossing has the states green light or red light. This is combined with the state of the vehicle, say the speed of the vehicle, in the model environment representation. Now if one adds another traffic light to the state space the two traffic lights can have 2^2 settings. Each traffic light setting also need each vehicle speed to represent a state of the system. Thus the state space doubles in size for each traffic light added to the system. Discrete algorithms and methods that utilize the action value function or the state value function to optimize the policy are called tabular methods. Tabular methods suffer greatly from the fact that the state space grows exponentially as more agents or features are introduced. In this thesis, to be able to use tabular methods which are easier to visualise and comprehend, the problems selected are small and simple in their nature. Non-tabular methods can be very effective in managing larger problems but these non-tabular methods will mearly be discussed briefly.

When the state space becomes too large for tabular methods to effectively manage the data required, one can attempt to use non-tabular methods instead. Non-tabular methods often rely on approximations of the action value function or value function. Popular approaches are to use neural networks to classify the states into actions to take in a given state. One can either approximate each action separately for a state with a function $f : S \times A \to \mathbb{R}$ or approximate the policy directly using a function $g : S \to \mathbb{R}^{|A|}$. To capture longer sequences of actions there is often a recurrent neural network with for example a long short-term memory network included in the architecture as well. Combining convolutional layer for classification with a recurrent layer to include dependence in time is considered the state of the art in reinforcement learning today. One of these methods is called Asynchronous Actor-Critic Agents which is abbreviated with A3C.

Tabular compared to non-tabular methods only differ in the way that the state space is interpreted for or by the agent. In tabular methods the agent has access to the full representation of the environment. This gives a more accurate description of the environment, compared to when the action value function is approximated, for the agent to base its decisions on. When the action value function is approximated, some information is lost in the representation of a the states. The non-tabular methods can potentially learn faster, since the size of the state space is in some sense

reduced. Tabular methods, on the other hand, have more accurate data to rely on if one decides to analyse how or why an agent made a specific decision in some state.

3 Single Agent reinforcement learning

In reinforcement learning, the agent is simulated in an environment where it can make decisions and observe the reward from an action taken in a given state. After each decision, the agents receives a reward and moves to a new state according to the underlying Markov decision process. By keeping track of the rewards received for each state-action pair, the idea is that the agent learns to make decisions which yield the highest possible total reward. The learning, training or optimisation of the policy can be done in each time step or separately in batches. When training using batches, data with paired up state, action and reward are saved in a batch to be processed in a later stage. Training the agent using such a batch is called batch learning or offline learning. Training in each iteration while the agent is acting upon the environment is called online learning.

In this section a few games are introduced which are simple and intuitive. The intention is to illustrate some of the core difficulties in reinforcement learning but also to touch upon the potentials of these methods. The n-armed bandit addresses the problem with non-stationary environment. Since the algorithms estimates the values of the state-action pairs through iteration, for the estimate to be accurate there must be a fixed value for the agent to estimate. If the value changes in time, that is if the assumption about the process being stationary is violated to some extent, the estimate might be far from accurate.

The grid world game addresses the problems associated with scalability, grid world game is also the main example problem which will be studied further. The grid world settings has the advantage that it is very versatile and many problems can be formulated as a variation of the grid world. The grid world has the potential to contain most of the difficulties that a reinforcement learning problem can have with stationarity, scalability and non-uniqueness of solutions [5].

3.1 The n-armed bandit

Reinforcement learning is built upon stationary Markov chains. For the agent to learn which decision is best for each state, it must assume that any previously attempted decisions in that state will have the same result. Otherwise the agent cannot compare the two decisions made. When the underlying environment is non-stationary, the agent cannot rely on previous estimates. The violation of stationarity is an important factor in multi agent systems since as the agent train, an effect is that the transition probabilities for the other agents changes. A classic example on how non-stationary environments can cause issues is the non-stationary n-armed bandit game.

In the stationary n-armed bandit game, the agent is to choose from n arms on a slot machine. The different arms have different expected returns and the objective of the agent is to find and focus on selecting the best arm. Selecting an arm is an action, so this game has one state and n actions. When the agent has found the arm with the highest expected return, it can focus on selecting that arm as often as possible to maximise its total reward. The rewards from each arm could be modelled to follow some probability distribution with different expected values for each arm. The learning process of the agent is then to explore the n different arms and estimate the expected values of each arm. After having observed the return of each arm a sufficient number of times, the agent can start to focus on the arm with the highest estimated expected return. By the law of large numbers this will in the end be the optimal strategy, given that the agent eventually picks one arm that it then selects in each iteration.

This is not the case if the environment is non-stationary, i.e. if the expected return for each arm changes with time. In this situation, the agent has to keep exploring to stay updated on the changes in the expected returns for each arm. The requirement to stay updated and keep exploring greatly inhibits the convergence rate for many algorithms and may even make the agent fail completely. Figure 3 illustrates how the expected return could be realised as a stochastic process for a two-armed bandit game.

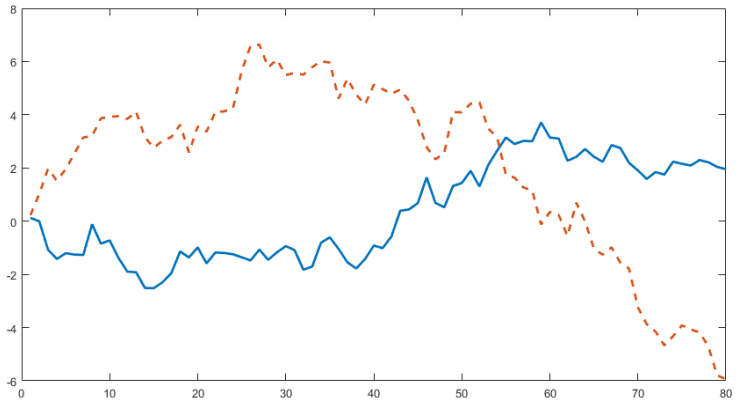


Figure 3: Simulation of Stochastic processes representing the means of the different arms in the non-stationary n-armed bandit problem. One can see that the arm represented by the dashed line is the best alternative, but only for a limited period of time.

In the case with the n-armed bandit where the agent must keep exploring, the agent can never pick one arm that it believes is the best and stop testing the other arms, this can end up in a total reward which is far from the potential maximum. The reason to why this concept is important in multi agent systems is that the optimal actions for different agents will likely depend on other agents actions in the system. Other agents in the system will also adapt and learn which could change the optimal action over time. Some strategies which looked promising early on might have been effectively countered by other agents or it might not fit well with the strategy for the team. Likewise, actions which initially seemed poor might be optimal later on, but the agent may have allready adapted the policy so that this action has low probability to be selected. It is therefore important to keep exploring strategies or re-learn new agents in a system that has adapted to a specific strategy. One method to avoid this problem is to train one agent at a time and exclude other agents, or keep their strategies fixed while one agent is training.

3.2 The grid-world game

The grid-world game is based on a discrete set of positions which are structured in a grid. In the standard settings of the game, the agent is placed in the grid and the goal for the agent is to find its way to the terminal state called the goal. This is the simplest variation of the grid-world which could also be viewed as the classical optimisation problem of finding the shortest path or the path with the lowest cost. One can extend the problem by letting some states in the grid have other properties, such as a wall which is a state that the agent cannot reach or move to. Another position is a pit which slows the agent down and reduces the reward gained by some amount. The grid-world setting is highly versatile and customisable to be used as a model for many problems. For example, any path finding optimisation problem formulated as a connected graph could be viewed as a grid-world problem. Possibly, one has to allow for the agent to move in more then 4 directions to fit graphs with more then 4 edges to a node.

In the standard formulation of the grid-world setting, the only termination of the simulation is when the goal is reached. The reward gained though the path is a measure of the quality of a path and this measure is used to update the probability of selecting a similar path in future simulations. This way the agent explores the states and adapts its policy with the hope of finding and adapting to an optimal solution to the problem of finding a path through the grid.

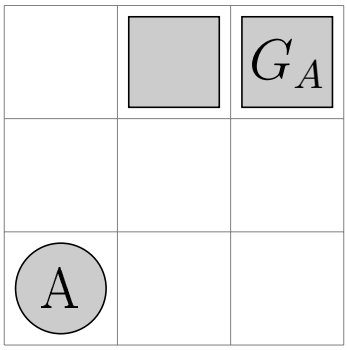


Figure 4: An illustration of a 3×3 grid-world game. Here A represents the agent, and G_A represents the goal for agent A. Reaching the goal yields a higher reward and terminates the simulation. The grey position represents a position which cannot be occupied by the agent.

3.3 Gridworld, default settings

Throughout this report, if not stated otherwise, the following settings will be used for each agent. The actions available to the agent is to stay put, move up, down, left or right. There are five actions in total and each time step yields a reward of -1 apart from when the agent reaches the goal. Reaching the goal yields a reward of 10. If there are multiple agents in a system, the terminal state is defined to be when all agents is at their respective goal positions. Moving outside the boundaries of the grid or into a wall will have the agent bounce back to its previous position but still takes up the time of making a step. In multi agent system the agents can collide with each other, collisions gives a reward of -2 for both agents and both agents are returned to their previous positions.

Due to the structure of the grid-world problem, many symmetries arise and there is often more then one optimal policy for one problem setting. An example of two selected optimal solutions for a small grid-world problem is illustrated in Figure 5.

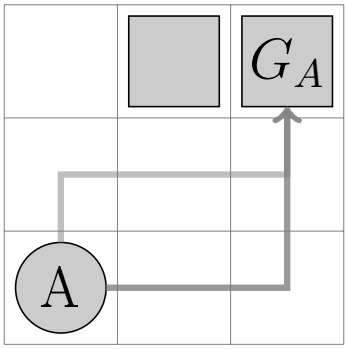


Figure 5: Example of two optimal paths for an agent in a 3×3 grid-world setting.

3.4 Solving Markov decision processes with reinforcement learning

Two algorithms which are commonly used in reinforcement learning are Q-learning and Temporal difference learning [12]. Both methods share the same idea and build on the idea of policy or value iteration. The agent takes an action a in state s according to some policy, moves to a new state s' and receives a reward. Once in the next state, the agent updates the value of the previous state-action pair (s, a) using the reward and the estimated value of the state-action pair $\max_a Q(s', a)$. Repeating the process, the rewards gained in some state propagate to neighbouring states and influence their values.

The intuitive idea of both temporal difference learning and Q-learning is that the agent takes a step forward, observes and evaluates the current state and then updates the value of the previous state-action combination. This means that the system is simulated using actions based on the current estimates to search for solutions. Temporal difference learning is related to Monte Carlo methods, one of the differences is that in temporal difference learning one uses the estimated model in the updates of the value function. In Monte Carlo one would simulate a full episode and then update the value function for each state while in temporal difference learning this is done while an episode is running, using the current estimates. The way of updating the value function is a form of bootstrapping as the agent uses the current estimates, or the previous knowledge, when learning. There is also a hyper parameter involved denoted α often referred to as the learning rate. One can interpret the expression below, $\alpha(r + \gamma V^i(s') - V^i(s))$, as a step in one direction where α gives the step length.

Algorithm 3: Temporal Difference learning

Data: Initialise V(s) as arbitrary real number for each s
Data: π, chosen policy to be evaluated
for *Number of training episodes to complete* **do**
    Start in initial state s
    while *Not at terminal state* **and** *Maximum number of steps not reached* **do**
        Choose action a based on current policy
        take action a and move to next state s'
        Receive reward r
        $V^{i+1}(s) \leftarrow V^i(s) + \alpha(r + \gamma V^i(s') - V^i(s))$
        $s \leftarrow s'$
        $i++$

Algorithm 4: Q-learning

Data: Initialise Q(s,a) as arbitrary real numbers for each state s
for *Number of training episodes to complete* **do**
    Start in initial state s
    for $s \in S$ **do**
        $a \leftarrow \text{argmax}_a\, Q(s, a)$,
        $\pi(s, a) \leftarrow 1$
    while *Not at terminal state* **and** *Maximum number of steps not reached* **do**
        Choose action a using policy $\pi(s)$
        or Choose action a at random if exploring
        Take action a and move to next state s'
        Receive reward r
        Update state-action value according to:
        $Q^{i+1}(s, a) \leftarrow Q^i(s, a) + \alpha(r + \gamma \max_{a'} Q^i(s', a') - Q^i(s, a))$
        $s \leftarrow s'$
        $i++$

3.5 Exploration vs. exploitation

When training of an agent starts it is important that the agent explores the solution set sufficiently. Exploring means that the agent moves to uncharted land or that the agent sometimes disregards its policy and selects an action at random. The exploration is not only important initially, but to get accurate estimates of the values of each state it is important to keep exploring. In problems which are not stationary the exploration is especially important, that is if any of the state values might change over time. This can happend if the transition probabilities change or if the reward function change with time.

There are multiple ways to implement exploration, the most common and straight forward way is to let the agent select action uniformly with some probability. Often this probability is initially high and decays as the agent trains. Other methods use probability distributions for the policy of an agent that maintain non-zero probability for each action. This way it is always possible for the agent to select each action and evetually it will explore each state-action combination.

4 Multi agent reinforcement learning

Systems with multiple agents are based on the same idea as single agent reinforcement learning. The difference being that one considers the joint action among multiple agents, since the agents all effect the environment and each other. Instead of looking at an action a for one agent one considers the joint actions $a_1, \ldots, a_n$ for n agents. To keep the notation simple and avoid multidimensional matrices, we will limit ourselves to systems with two agents. The ideas presented in this section extend to systems with any number of agents, apart from the minimax Q algorithm. The fact that each agent acts independently and influences the quality of other agents decisions is precisely what poses problems in multi agent systems. The environment is no longer stationary and a state-action pair which gave a high reward previously might be a terrible choice given another agent's decision.

There are several reason as to why one would want to split up one agent into several decision making entities. Restrictions in communication and data management between parts of an agent could be one motivation as to why this separation is required. It could also be that the problem changes with time and it is not feasible to redo the full calculations required to find a new feasible solution. If the agents could act independently and solve small tasks on their own without instruction from a central control structure, one can relax the need of high performance communication.

4.1 Stochastic games

For two agent systems one can represent the reward structure as a matrix where the actions for two agents, (a_1, a_2) are indexes in the matrix. Here a_1 is the action for agent one and a_2 is the action for agent two. These matrices are in game theory called matrix games [8]. This extends naturally to multidimensional matrices but for simplicity of presentation we shall only consider two player games. Matrix games give an overview of the reward for one step in a Markov decision process. The matrices are organised so that for each combination of actions, the corresponding element in the matrix gives the rewards for each agent. Say a matrix game is given as the matrix

$$\begin{bmatrix} (1,-2) & (-1,2) \\ (-2,1) & (2,-1) \end{bmatrix}.$$

Given the actions for each agent one can find the reward for this combination of actions in the matrix representing the matrix game. Suppose the agents both takes action one, then the element $(1, 1)$ indicates that agents one gains a reward of one while agent two gains a reward of minus two. Some example matrices for matrix games are presented in Table 1. Matrix games can be classified into the categories zero-sum games, general sum games and cooperative games. Examples of each game is presented in Table 1. The game in Table 1a is a game of matching pennies, it is a zero-sum game meaning that the reward matrices for each agent sum up to the zero matrix. The game in table 1c is a general sum game, there is no fixed structure for how the rewards relate to each other. The last game, Table 1b, is a cooperative game where the rewards for each agent share the same sign in each action pair.

Table 1: Example reward structures for different classes of games. Table 1a shows a zero-sum game where the rewards for each agent sums up to zero in each pair of actions. Table 1b shows a cooperative game where the signs for the rewards of each agent share the same sign. Table 1c shows a general sum game where the rewards does not relate to each other in a structured way.

(a)

	$a_2 = 1$	$a_2 = 2$
$a_1 = 1$	(1, -1)	(-1, 1)
$a_1 = 2$	(-1,1)	(1,-1)

(b)

	$a_2 = 1$	$a_2 = 2$
$a_1 = 1$	(0, 0)	(1, 1)
$a_1 = 2$	(-1,-1)	(2, 2)

(c)

	$a_2 = 1$	$a_2 = 2$
$a_1 = 1$	(3, -5)	(2, 4)
$a_1 = 2$	(-5, 3)	(-2, 1)

When the action taken by the agent is stochastic, a matrix game is called a stochastic game. Much like a matrix game, the difference being that the actions are chosen with respect to some probability distribution. Littman [7] proposed a framework for multi agent reinforcement learning, based on stochastic games. By formulating a matrix game for each state one can use the matrix of rewards to find an optimal policy in each state. The matrix of rewards can be multiplied by the policy distributions by matrix-vector multiplication which gives the expected reward under a given policy. By finding the policy which maximizes the expected reward in a given state using optimisation, the idea is that this policy will be part of the policy over all states. By repeating the optimisation for each state, the optimal one-step policies will form the full, optimal policy for solving the problem.

4.2 The grid-world with multiple agents

Extending the grid-world problem into a multi agent system we consider a multi agent path finding problem. This is essentially the same as the grid-world problem but with more agents. This introduces another dimension to the state space and increases the complexity of the problem. The most significant property in the grid-world problem which affects the convergence speed is with no doubt when more features are added to the environment. A feature is a new type of object in the grid with properties distinct form the other types of objects in such a way that it should be uniquely represented in the state. For each agent added to the system, the agent will increase the number of possible configurations by a factor equal to the number of valid positions. Suppose we have a 2x2 grid where the agents can be in the same state, for a system with one agent we have 4 possible states. Adding one agent, the added agent can occupy any of the 3 remaining positions giving 12 possible states, disregarding any symmetries. Letting n be the number of positions and k the number of agents, then the general expression for the number of states is $k!\binom{n}{k} = \frac{n!}{(n-k)!}$. This is when we assume that each agent is unique so that two states where 2 agents have swapped places are distinct from each other. With non-unique agents the size of the state space decreases to $\binom{n}{k}$.

4.2.1 Multi agent path finding

Many variations of the grid-world game exist, in this report we focus on multi agent path finding and a game of tag. In the path finding formulation there are multiple agents in a grid and each agent has a goal position. The objective for the agents is for all agents to reach their goal positions in as few time steps as possible and preferably without colliding with each other. In the game of tag we consider two agents where one is the invader and the other is the defender. The defender has as its objective to come sufficiently close to the invader, if the defender gets sufficiently close the game will terminate and the defender gains a positive reward. The invader has as its objective to reach a given goal position. The game of tag is thus a variation of a path finding problem with the goal being mobile for one of the agents. The game of tag also differs in that the terminal state is when one of the agents reaches their goal and not all agents.

4.2.2 Multi agent game of tag

The other variation of the grid-world game that is implemented in this thesis is the game of tag. In the game of tag, one agent is chasing another agent. The evading agent have an objective to reach a goal or occupy some area in the grid. The other agent is to protect that area and catch the evading agent. The simplest variation of the game is equivalent to a game of matching pennies. In the game of matching pennies there are two agents and both agents are to choose between two actions. Agent one wins the game if the same action is selected and loses if the agents select different actions. The game of matching pennies is extensively studied in game theory and the Nash equilibrium strategy is for both agent to select action at random, that is by flipping a coin. In the grid-world formulation, consider a 2×2 grid where 2 agents start in opposite corners as illustrated in Figure 6, the evading agent A is to reach the position where the defending agent B starts. Each agent decides on one of two actions, going to the upper right position or going to the lower left position. In case they go to the same position, the defender wins. If they choose to go to different positions, the evader can enter its goal position safely and wins. That is, the two agents play a game with two actions. One agent benefits when they select the same actions, the other agent benefits when they select different actions. The full game only takes one step per trail. This is equivalent to the game of matching pennies which is a extensively studied game in game theory [8].

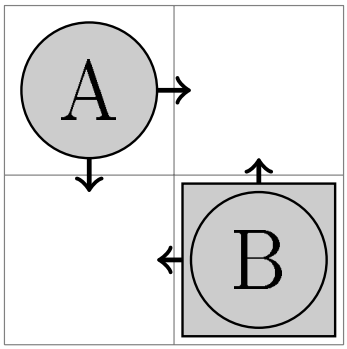


Figure 6: Game of matching pennies as a grid-world game. Agent B wants to go to the same cell as agent A for a positive reward. Agent A gains a negative reward for moving into the same cell as agent B.

4.3 Learning algorithms for multi agent systems

We shall now present a selection of algorithms tested in this thesis, algorithms was selected with inspiration form the work in [3]. Some algorithms utilize the information of all agent's actions to construct a matrix game based on the state-action function. In these algorithm the actions from each agent is considered jointly as one action. That is, if one agent selects action a_0 and the other agent selects action a_1, the joint action will be (a_0, a_1) which is one of the possible actions in the action value function $Q(s, a)$. We shall denote the action value function with the individual actions as $Q(s, a_0, a_1)$ instead of mapping each joint action to a new, discrete, set of actions. One can use the true state-action function or an estimate of the state-action function if it is initially unknown. The constructed matrix game gives a reward structure for each combination of actions chosen by the agents. By finding the optimal policies for a given state, the idea is that the full policy for each state will converge towards the optimal policy.

4.3.1 Minimax Q

The minimax Q algorithm is designed for two player zero-sum stochastic games, published in [7]. The idea is to minimize the opponents expected reward by finding an optimal policy to a linear optimisation problem. The algorithm relies on the expression $\max_{\pi_s} \min_{a_o} \sum_a Q(s, a, a_o)\pi_s(a)$ where a is the agents action and a_o is the opponents action. The idea is to maximize the policy with respect to the least favourable action that the other agent could choose. Suppose agent one is playing minimax Q, then the inner sum is a conditional expectation over the discounted rewards for agent one given that agent two selects action a_o. Minimax Q does not take into account the other agents policy, instead one assumes the worst. The algorithm is essentially the same as standard Q-learning apart from the term maximisation over the state-action value function being replaced by a minimax over the conditional expectation.

Algorithm 5: Minimax Q Learning

Data: Initialise $V(s)$ as arbitrary real number for each state s
Initialise the policy $\pi(s, a)$ as a uniform distribution over actions
Initialise the state-action value function $Q(s, a_s, a_o)$ as an arbitrary
real number for each state and joint action pair.
Start in initial state s, let $i = 0$
while *Not at terminal state* **and** *Maximum number of steps not reached* **do**
  Choose action a based on policy π
  take action a and move to next state s'
  Update Q-values for each agent:
  $Q^{i+1}(s, a) \leftarrow Q^i(s, a) + \alpha(r + \gamma V^i(s') - Q^i(s, a))$
  $V(s') \leftarrow \max_{\pi(s)} \min_{a_o} \sum_a Q(s, a, a_o)\pi(s, a)$
  $s \leftarrow s'$
  $i++$

The matrix containing the Q-values is built up by the estimated rewards, $r(X, Y)$ where X and Y are stochastic variables over the actions of agents one and two respectively. The objective function in the optimisation problem for a given state s is the minimum of $\sum_a Q(s, a, a_o)\pi(s, a) = \mathbb{E}[r(X, Y)|Y = a_o]$. So the objective is to maximise the conditional expectation, given that agent two has chosen the action which gives the smallest expectation. In other words, agent one tries to minimise the risk by maximising the expected reward of the worst outcome.

4.3.2 Nash Q

In Nash Q the goal is to compute an optimal strategy for each state in a hope to converge to the overall optimal strategy. The optimal strategy is computed with respect to a conditional expectation, using the estimated rewards. Hu and Wellman [5] concluded that under some assumptions, Nash Q is guaranteed to converge in general sum games. During learning, the game must maintain precisely one optimal strategy. Furthermore, when solving the quadratic programming problem, there must exist either a saddle point or a global optimum. These two conditions are often too strict since the estimated Q-values that they rely on change while training, and it is difficult to keep track of what happens to these Q-values.

The NashQ$[Q(s')]$ denotes a scalar derived from optimising the policies with respect to the future rewards for an agent. In a game with two agents, the future rewards for agent one are represented as a matrix where each element (i, j) in the matrix gives the estimated expected future reward given that agent one selects action i and agent two selects action j. The reward for agent two would be element (j, i) in the reward matrix of agent two. Given some state, let R denote the reward matrix for agent one, $\boldsymbol{\pi}$ denote the policy for agent one and τ the policy for agent two. Let there be n actions, so that R is an $n \times n$ matrix, $\boldsymbol{\pi}$ and $\boldsymbol{\tau}$ are n dimensional vectors. In a one-step game, the expression for NashQ$[Q(s')]$ simplifies to the expected reward for an agent, which is only natural since that is exactly what the agent wishes to maximise. In this expression the actions for agent one and two are denoted by the stochastic variables X and Y respectively.

$$
\begin{aligned}
\boldsymbol{\pi}^T R \boldsymbol{\tau} &= [\pi_1, \pi_2, \ldots, \pi_n] \begin{bmatrix} r(1,1) & \ldots & r(1,n) \\ \vdots & \ddots & \vdots \\ r(n,1) & \ldots & r(n,n) \end{bmatrix} \boldsymbol{\tau} \\
&= [\sum_{i=1}^{n} \pi_i r(i,1), \ldots, \sum_{i=1}^{n} \pi_i r(i,n)] \boldsymbol{\tau} \\
&= [\mathbb{E}[r(X,Y)|Y=1], \ldots, \mathbb{E}[r(X,Y)|Y=n]\, \boldsymbol{\tau} \\
&= \tau_1 \, \mathbb{E}[r(X,Y)|Y=1] + \cdots + \tau_n \, \mathbb{E}[r(X,Y)|Y=n] \\
&= \sum_{j=1}^{n} \mathbb{E}[r(X,Y)|Y=j] \tau_j \\
&= \mathbb{E}[r(X)]
\end{aligned}
$$

The simplification uses the definition of vector-matrix multiplication and the definition of the conditional expectation. For the last step one needs the law of total expectation to see that this is indeed the expected reward for agent one. The algorithm is presented in detail in Algorithm 6 where NashQ$[Q(s')]$ is the expectation in the quadratic form presented above. In an environment with multiple steps, the matrix R will not contain the rewards directly but instead contain the estimated action value function values.

Algorithm 6: Nash Q learning

Data: Initialise $Q_k^0(s,a)$ as arbitrary real numbers for each state-action pair (s,a) and for each agent k

Initialise $\pi_k(s)$ as uniform distribution for each state s and for each agent k

Start in initial state s, let $i = 0$

while *Not at terminal state* **and** *Maximum number of steps not reached* **do**

- Each agent k choose an action a based on $\pi_k(s)$
- Each agent takes action a and moves to next state s'
- Receive reward r
- Solve the optimisation problem given by $\text{NashQ}[Q_k^i(s')]$
- $\pi_k(s) \leftarrow \text{argmax}_{\pi_i(s)} \text{NashQ}[Q_k^i(s')]$
- $Q_k^{i+1}(s,a) \leftarrow Q_k^i(s,a) + \alpha(r + \gamma \text{NashQ}[Q_k^i(s')] - Q_k^i(s,a))$
- $s \leftarrow s'$
- $i++$

4.3.3 Win or learn Fast

Another class of learning methods tries to avoid solving the linear or quadratic programs as in minimax Q and Nash Q. In win or learn fast one tries to adapt the learning rate based on conditions derived from experience. Similar algorithms exists in numerous variations [2] but in theory they are not much different from each other.

The Win or Learn Fast algorithm is Q-learning with an adaptation so that when some condition is met, the agent changes its rule for updating the policy or the action values. Specifically, the learning rate is changed based on whether the agent thinks it is losing or winning. When the agent is winning, it will utilize regular Q-Learning. When the agent thinks it is losing, the agent will try to learn faster by increasing the learning rate.

Win or Learn fast tries to classify its current strategy as a losing strategy or a winning strategy. The agent then adapts its learning rate to slow learning if the agent is winning or fast learning if the agent is losing. This could be interpreted as an adaptive step length that takes smaller if the current solution is close to an optimal solution and longer if it is far from the optimal solution. As a means to decide upon whether the current strategy is winning or losing the average of previous strategies is used.

Algorithm 7: Win or learn fast Q learning

Data: Initialise $Q_k^0(s,a)$ as arbitrary real numbers for each state s and for each agent k

Start in initial state s, let $i = 0$

while *Not at terminal state* **and** *Maximum number of steps not reached* **do**

- Choose action a based on $Q^i(s,a)$
- take action a and move to next state s'
- Receive reward r
- Adjust learning rate for each agent k:
- **if** $\sum_{a \in A} \pi_k^i(s,a) Q_k^i(s,a) < \sum_{a \in A} \bar{\pi}_k(s,a) Q_k^i(s,a)$ **then**
  - $\alpha \leftarrow \alpha_l$
- **else**
  - $\alpha \leftarrow \alpha_w$
- $Q_k^{i+1}(s,a) \leftarrow Q_k^i(s,a) + \alpha(r + \gamma \max_{a'} Q_k^i(s',a') - Q_k^i(s,a))$
- $s \leftarrow s'$
- $i++$

5 Problems implemented

Using MATLAB a few selected problems were implemented and simulated to test the previously presented algorithms. The problems are inspired from H. Schwartz [10] which has also served as an initial inspiration to this thesis. The minimax Q algorithm Algorithm 5 is only applicable to zero-sum games and thus it was only tested for problems which statisfy this condition. Nash Q and win or learn fast algorithms Algorithms 6 and 7 are both applicable to general sum games, these two algorithms where tested zero-sum and general sum games.

5.1 Small game of tag

In the game of tag two agents play against each other. The small version is formulated as a zero sum game so that the minimax Q algorithm could be applied. The small game of tag is implemented as a 2×2 grid-world game and is illustrated in Figure 6. The game is equivalent to a game of matching pennies and has two agents and two actions with only one state. One could interpret the four possible terminal states as their own states in which case the game would have 5 states. It is however only one state that is of interest since the other four states have no possible actions associated with them. The four terminal states does however have a value and a reward associated with them. The defending agent wants to select the same action as the intruding agent. The intruding agent wants to select the action that the defending agent does not select. The optimal strategy for this game is for both agent to select their action with equal probabilities, 0.5 that is.

5.2 Path finding problems

A cooperative path finding problem as illustrated in Figure 5 was implemented with Nash Q and WolF algorithms. The game was implemented with the same settings as in [5]. In **??** a path finding problem with two agents and two goals are implemented as shown in Figure 7 where a few selected optimal strategies are also shown. The rewards are set to the default values, that is -1 for each time step taken and 10 for reaching the terminal state. To encourage agents to stay at their goal and wait for the other agent to finish, there is also a reward of $+1$ when standing at the goal position.

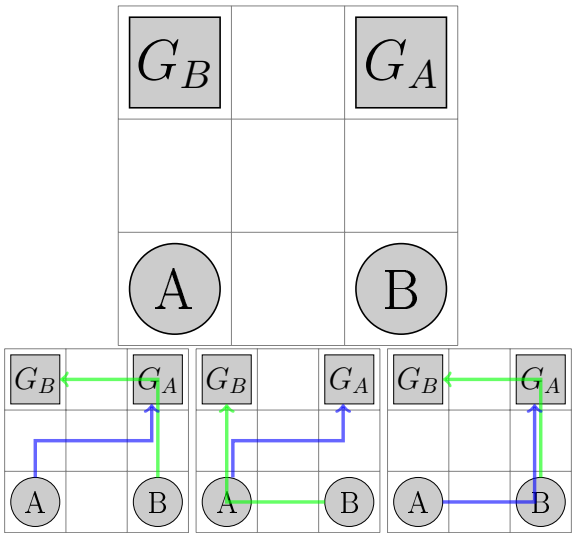


Figure 7: Selected optimal policies for the Gridworld problem used in Nash Q implementation.

5.3 Large game of tag

The 6×6 version of the game of tag was implemented using minimax Q, Nash Q and win or learn fast. The game start setup is illustrated in Figure 6. For minimax Q to be applicable, the reward structure was set to a zero-sum game. In the larger 6×6 version illustrated in Figure 8 one can formulate the rewards in a number of ways to motivate different strategies. The goal for the defending agent could be either to catch the intruding agent as fast as possible or the main goal could be to defend the terminating state which is the goal for intruding agent. Depending on how the rewards are formulated, the agents will adapt to different strategies, either it can wait near the goal for the intruding agent or chase the intruding agent across the grid.

If the defending or the intruding agent gets positive reward for letting the game continue and not pursuing the terminal state, the agent will stall the game by waiting for the other agent to pursue its goal. If both agents gain negative rewards for stalling the game, both agents will pursue their goals. If both agents gets a positive reward for stalling the game, both agents will learn to stall the game. The game was implemented with a zero-sum formulation for comparison of the three algorithms. The zero-sum formulation could be implemented either as positive versus negative reward for all states or as zero reward for all states except the terminal states. The large game of tag was also implemented with a general sum formulation where both agents receive negative feedback for stalling the game.

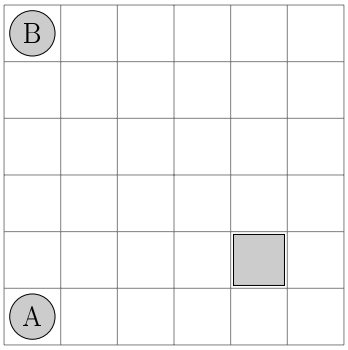


Figure 8: Intruder defender game, the goal for agent B is to reach the gray area while agent A tries to defend the same area.

6 Performance measure

In this chapter we will present and discuss a few different methods that have been used for evaluation of policies. The problem solving is focused around the use of Q-Tables and different ways of updating the tables based on the rewards. Some methods learn faster than others but also explore less which often makes them end up in a solution that is suboptimal. In the small problems that are considered in this report one can see an optimal solution to the problem just by looking at it. Such optimal solutions will be used as a reference to measure a methods performance.

6.1 Reference solution and comparison of methods

In reinforcement learning it is often the case that the optimal solution is not known or it is too cumbersome to compute. This makes it difficult to measure performance in some situations. Measures of performance in reinforcement learning is in general difficult to formalise and generalise. A popular measure is to compare with average human performance by looking at statistical data

over some population. For example when using reinforcement learning to play games, the agent can compete against human players and compare high score. Performance is then measured as a percentage compared to the top human players high scores. The same idea has been applied in image analysis, for example when x-ray images are to be diagnosed and classified by doctors. The program trains and computes its accuracy on how often it correctly classifies an x-ray image. The programs accuracy is then compared to professional doctors average accuracy.

In the simplest problems like the ones considered in this thesis, apart from the large version of the game of tag, the minimum number of time steps needed can be easily calculated by simply looking at the problem. Indexing the start position with (x, y) and the goal with (x', y'), the minimum number of steps is given by the Manhattan distance $|x-x'|+|y-y'|$. This assumes that there are no obstacles or other agents conflicting with the shortest possible path. In the problems implemented in this thesis, the Manhattan distance between points will in general be the minimum number of time steps needed. A feasible solution with the same number of steps as the Manhattan distance will exist in all problems considered. For a 3×3 Grid the number of steps needed to reach any position from any other position is at most 4. For a 10×10 grid the number of steps is 18 and so on. The optimal solutions can be used to measure performance by looking at how often the agent learns an optimal solution. We will consider the success rate which is a measure of how often an algorithm converges to an optimal solution. While the agents are training, the number of time steps needed to find a solution is registered and if the solution is optimal, the training has succeeded. If the agent does not reach an optimal solution before the training is finished, the training is considered a failure. The ratio successes/(successes + failures) is then the success rate.

In the large version of the game of tag it is harder to find the optimal solution and it depends on the reward structure for the two agents. Also the game is imbalanced since the defending agent has a winning strategy and has an upper hand. By standing near the goal the defending agent can prevent the evading agent from ever reaching reaching the terminal state. One can, however, formulate the rewards in a number of ways. One can give each agent negative reward for keeping the game alive, making it beneficial for both agents to end the simulation as early as possible. To formulate the game as a zero-sum game, one has to either give one agent positive reward and the other a negative reward or give both agents zero reward in each time step. What the optimal strategy turns out to be is based on the formulation of the rewards.

If both agents get zero reward for each time step, except for the terminal states, there is a winning strategy for the defending agent. If the defending agent stand close to the goal of the intruding agent, it would be impossible for the intruding agent to reach its goal. A well trained defending agent should then win in every match. If the defending agent get a positive reward in each time step the same strategy can be used without loss and a well trained defending agent should win in each match in this formulation as well. With the formulation where the defending agent gets a negative reward in each time step it will be motivated to end the match early. This formulation is in the intruding agent favour and a well trained intruding agent will win more often.

An illustration of how a policy can look is shown in Figure 9. To simplify the presentation of a policy, agent B is assumed to follow an optimal path and the policy for agent A is shown in each cell. The figures in black represents the states possible for agent A to occupy when both agent follow the policy shown. The grey states, even though it is possible for them to occur, they will not occur if the agents follow the specified policy.

7 Results

In this section we present results from our implementations and some comparative results for the different algorithms and problems discussed throughout the thesis. A table with a summary of the

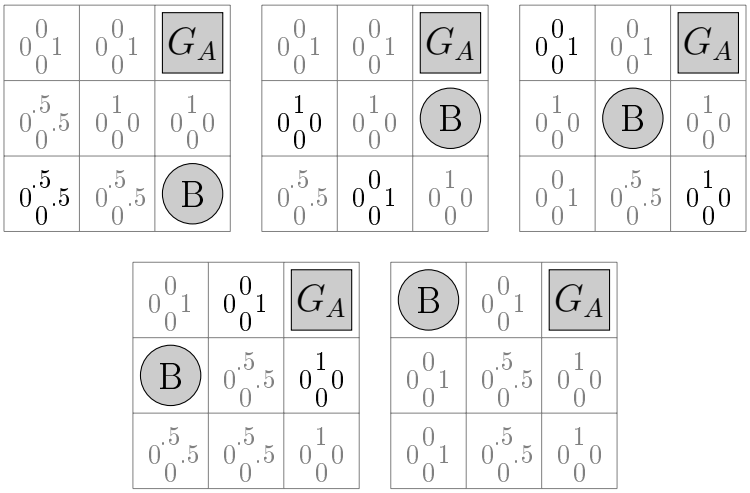


Figure 9: Optimal policy for agent A when agent B is following a selected optimal path. In each cell are the probabilities for moving in each direction indicated by relative position from the centre of the cell (up, down, left or right). Policies coloured in grey will not be included in the optimal policy since the states they represent are not included in the optimal solution. As can be seen, there exists two equally good options in the first, bottom left, state for agent A.

quantitative results are shown in Table 2.

Table 2: Best observed success rate for the different algorithms in different problems. 2×2 refers to the 2×2 game of tag, 3×3 a refers to the $3x3$ path finding problem with 2 actions. 3×3b refers to the 3×3 path finding problem with 4 actions. 6×6a refers to the 6×6 game of tag where the defending agent wants to stall the game. 6×6b refers to the 6×6 game of tag where the invading agent wants to stall the game. In the problems 6×6a and b the success rate is measured as the ratio of wins for the defending agent.

Problem	2×2	3×3a	3×3b	6×6a	6×6b
Minimax Q	100%	-	-	93%	100%
Nash Q	100%	100%	75%	100%	100%
WolF	100%	70%	50%	97.9%	67.4%

7.1 Small game of tag

Minimax Q showed to be robust and converges for almost any learning parameters or starting conditions chosen for the small game of tag. Convergence for selected parameter settings are shown in Figure 10. Minimax Q is limited however since the algorithm is only applicable to zero-sum games.

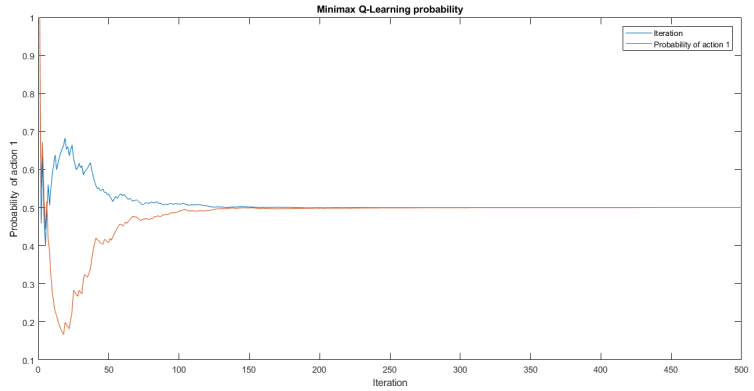


Figure 10: Results from the small game of tag with minimax Q learning. Both agents quickly converge to the optimal policy where they select any action with probability 0.5.

With Nash Q one solves the optimisation problem for the one step matrix game and the algorithm finds the optimal strategy in one iteration with 100% success rate. The problem only has one state and the state satisfies the assumptions for the Nash Q algorithm. Thus, one can formulate the reward matrix for this state and solve optimisation problem which will give the optimal policies for both agents.

Win of learn fast converges towards an optimal policy but does not solve for one exactly like the Nash Q and minimax Q. However, with small step length it quickly learns a close to optimal policy and stays there, as seen in Figure 11.

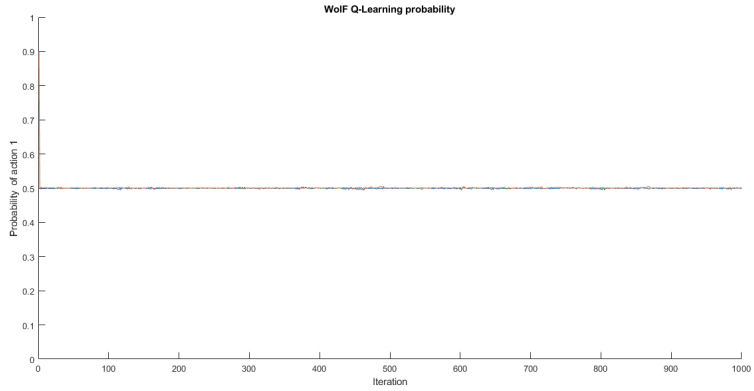


Figure 11: Results from the small game of tag with win or learn fast Q learning. Both agents quickly converge to the optimal policy where they select any action with probability 0.5.

7.2 Path finding game

In the implementation of the Nash Q algorithm to the grid-world problem shown in Figure 7, there was issues with convergence, conclusive with the results of Hu and Wellman [5]. Performance was measured in success rate which is defined as the proportion of trails which converged to an optimal solution. In the original formulation with five possible actions, (stay, up, right, left, down), in each state the algorithm rarely found an optimal solution. After making restrictions to the action space as done in [5], the convergence increased first to 75% and then to 100%. When removing the

option to stay in the action space the success rate increased to 75%. After removing all actions not present in the optional solutions there was a success rate of 100% even though the problem still violates the required conditions for the algorithm. Moving down and staying still was excluded completely as well as moving left or right depending on the agent's optimal path.

Even though the assumptions are still violated with the limited action space, the algorithm is still consistently converging to an optimal solution. The violations seen in this problem is the existence of multiple optimal policies. In the Nash Q algorithm one does not have control over which policy the agents choose if there are multiple options. In the 3×3 path finding problem the agents have to coordinate so that one of the agents moves up and the other moves to the middle, there is no such coordination in the Nash Q algorithm.

When training the agent's using the win or learn fast algorithm it is notable how large of an effect the training parameters have on the algorithms behaviour. Good settings of training parameters seem to be problem specific and finding good choices could lead to extensive work. The training itself is fast compared to Nash Q and minimax Q. However, investigating the learning rate, the discount factor and the criteria for winning or losing could take time still and there are no guarantees for convergence to an optimal solution. The process of testing different parameter settings takes up a significant time even for these small problems and potentially there are no good choices of parameters.

7.3 Large game of tag

In the large game of tag in a 6×6 grid-world it is harder to define an optimal policy for the problem. However, with some of the rewards structures used there is a winning strategy for the defending agent. The defending agent can wait near the goal for the intruding agent to get nearby, it will be almost impossible for the intruding agent to reach its goal if the defending agent is guarding. For the defending agent to benefit from this strategy it must not gain negative reward for stalling the game. If the defending agent gets negative reward for stalling, the accumulated negative reward will negate the positive reward gained for catching the intruding agent. The success rate in Table 2 is given as the proportion of wins for the defending agent.

8 Discussion

The algorithms presented are some of the first and simplest ones which are adapted for multi agent systems. Nash Q and minimax Q are also the only algorithms found which have any guaranties of finding an optimal solution as long as the required assumptions are met. The assumptions required in Nash Q are very strict and also nearly impossible to validate since the assumptions depend on the estimated state-action values. These values change as the agent is learning, making it hard to maintain the assumptions. In minimax Q the assumption about the game being a zero-sum game limits the algorithm to a smaller set of problems. In this section we discuss the results and draw conclusions about the algorithms that have been presented and tested. Some ideas for future work are also discussed.

8.1 Minimax Q

Minimax Q was only tested for the two versions of game of tag with a zero-sum reward structure. The algorithm shows very promising results and is very robust in both problems. The learning

parameters or starting conditions, that is how the estimates are initialised, seems to have very low impact on the algorithms performance. The algorithm is only valid for systems with two agents or systems where the agents can be categorized into two teams. Some variations of minimax Q which are team based exists and essentially they treat a team as one single agent. In minimax Q one has to solve a linear system of equations in each iteration. The system of equations grows quadratically with respect to the number of actions available. When more agents are introduced an action is interpreted as the joint action of all agents in that team. This makes the problem scale poorly when more actions or agents are added to the system due to the linear program taking up too much computational power. Minimax Q is certainly a valid candidate for small problems or in problems where the computational time is not an important factor. However, problems of this size can often be solved with a deterministic optimisation algorithm. Thus, there seems to be no good use for the minimax Q algorithm.

8.2 Nash Q

From tests of the Nash Q algorithm it is clear that when the assumptions are violated, there are issues with convergence, which is of course expected. Nash Q has a guarantee of convergence for a very strict set of problems. For example in problems where multiple optimal solutions exist there is no guarantee for convergence. The Nash Q algorithm does however perform better than promised by the required assumptions and solves the problem reliably even though the assumptions are violated to some extent, as they are in the path finding problem. In the problems considered in this thesis there exists symmetric optimal solutions that should causes issues for the Nash Q algorithm since one of the required assumptions is that there should be only one optimal policy in each stage game. However, even if the problem have precisely one optimal policy, the algorithms uses the estimated Q-values for finding the optimal policies for each state. One cannot guarantee that the intermediate, estimated, Q-values do not violate the required assumptions. Furthermore, considering the restrictive assumptions it seems unlikely that the intermediate Q-values does not violate the assumptions.

The Nash Q algorithm needs information about the other agents policies, this can be gained from observing the other agent's reward and estimating their policies or by sharing its policy with others.

If one seeks to use a multi agent framework for reinforcement learning to solve route optimisation and scheduling problems where on-line updates of the paths might be necessary due to changes in the environment these methods are highly intractable for large systems. The data management required for storing the massive amounts of data required for any problem of reasonable size is also intractable.

8.3 Equilibrium strategies

In multi agent reinforcement learning, much inspired from game theory, one tries to find a so called equilibrium strategy. In this thesis we have called them optimal strategies since they are the optimal solutions to the optimisation problems which defines them. With optimal we mean that it is the policy which maximises the value function for each agent. There is however reason to question the assumptions made in this approach. It is assumed that the opponents are rational in the sense that the opponents too are trying to maximize their value functions, if they do not then the optimal solution found might not be optimal at all. It is also assumed that the reward function is sensible and well defined, the reward are in many problems engineered in a way to promote specific behaviour. In some situations the rewards can translate directly to something real, such as in a scheduling or a logistics problem where the rewards translate directly into moved goods or time taken to complete a set of tasks.

8.4 Win or learn fast

Win or learn fast is an implementation of regular Q-learning where one ignores that the assumption of stationarity is violated. For some problems this naturally has larger impact than for others. A modification to Q-learning is made to make the agent try to re-train in some situations to avoid getting stuck with a suboptimal strategy. The advantage with Win or learn fast is that the training is very fast. The algorithm only needs to calculate one scalar product in addition to the calculations done in Q-learning. Due to its speed, one can test a large set of candidate parameters to find suitable parameter settings which work for the specified problem. However, since there are no guarantees for convergence, the process of searching through parameters might not yield a solution to the problem at all.

9 Conclusions

The ideas presented in this thesis rely on optimisation problems to solve the intermediate optimal policy for each state. We have shown that these methods scale badly with the increasing size of the problem. Moreover the minimax Q algorithm is very limited in which problems it is applicable to and the same holds for Nash Q. For Nash Q it is also the case that even if the algorithm is applicable to the problem in its final form, there is no way to guarantee that the algorithm is applicable to the intermediate problems defined by the estimated state-action value function. This makes the algorithm unreliable.

It is also debatable whether it is justifiable to solve for the intermediate optimal policies. It may not be reasonable to assume that the opponents or any other dynamic objects in the problem are rational as is assumed when choosing the optimal intermediate strategies. However, when solving a problem through simulation this could be engineered to hold.

10 Further research

Finding algorithms which can guarantee convergence to optimal strategies and scales well with an increase of the problem would be fantastic. This is often a trade-off one has to do in optimisation problems. Either one uses a heuristic solution to find a suboptimal solution quickly, or one uses a computationally complex algorithm which guarantees convergence to an optimal solution. One advantage with optimisation problems is that one can often compare different solutions to see which is best. This makes people lean towards using fast heuristics which could find a solution quickly and one can verify its quality.

10.1 Partial observable states and decentralization

In reinforcement learning, a problem that occurs frequently is the issue of large state spaces. When the state space becomes infeasible to manage one can either try to find a more compact representation or try to exclude information that can be considered as irrelevant. Sometimes, information is excluded as a result of the model, some information is simply unknown. An important distinction that makes an observation partial is if two or more states in the model are represented in the same way. Reinforcement learning problems with this property are called partially observable and the framework used to represent the environment is called partially observable Markov decision processes. As an example of a partially observable state representation, suppose we model the

state of a vehicle as the nearby surroundings only and suppose the vehicle is in a very long tunnel, Illustrated in Figure 12. In the environment each state is unique but for the vehicle they are all the same. In this case, the vehicle interprets each position in the tunnel as equal but in the environment it is not. In other words, when a mapping from the environment to the set of states is not one-to-one, the Markov decision process is said to have partial observations.

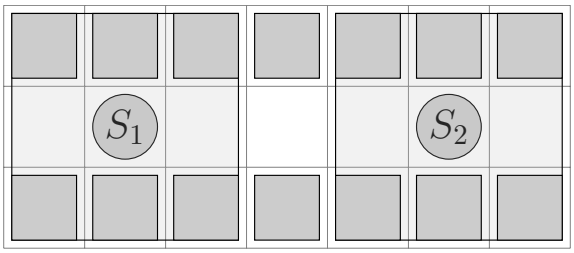


Figure 12: Visualisation of partially observable states in a Markov decision process. S_1 and S_2 are equal but do not represent the same state in the environment.

With partial observability, one can for example exclude all other agents in a system or only consider nearby agents. One could exclude any information which is not of importance for the agent's decision and hopefully the agent will still be able to learn an optimal solution. By clever representation of the state space the agents can become more dynamic and applicable to a broader class of problems. In the implementations considered in this thesis, the agents train for one specific problem only and can not handle different scenarios efficiently. Instead the agent would have to re-train and perhaps adapt learning parameter to successfully solve modified problems.

10.2 Partial observations in the grid-world

In the grid-world an idea for partial observations is to let the agents have a vision distance. In Figure 13 one can see a visualization of the partial observation space. An observation for the agent consists of the nearby environment and a unit vector pointing out the direction in which the agent can find the goal. With this formulation of the state space, the agent is independent of the size and shape of the grid-world. The agent can train itself in finding the goal and when trained, the agent has the potential to find solutions in multiple settings. It does not matter where the goal is, as long as it can be reached. If the agent has learned to use the unit vector to find the goal,it can navigate to the goal no matter the starting position of goal position. This formulation suggested is also well suited for applying in an image analysis framework, using convolutional neural networks to classify the different states. The idea is implemented in [9] with some promising results, in particular the versatility and scalability of this method is impressive.

10.2.1 Position balansing problem

Consider a looped route with several vehicles which are moving around the loop with some common reference speed v_r. The objective for the vehicles is to maintain an even distance between them. In other words one wants to minimize the variance in the set of distances between vehicles. If there are any random disturbances along the route such as traffic, hills or other obstacles, the distance between the vehicles will be disturbed. The vehicles must adapt their speed to the current state to maintain an even flow of vehicles. This can be done by adding an adjustment to the reference speed so that $v_1 = v_r + v_i$ where v_1 is the speed of agent 1, adjusted by v_i.

If implemented as a single agent reinforcement learning problem, one way is to let the state space be the position of each vehicle in the route. A problem with this formulation is that it quickly

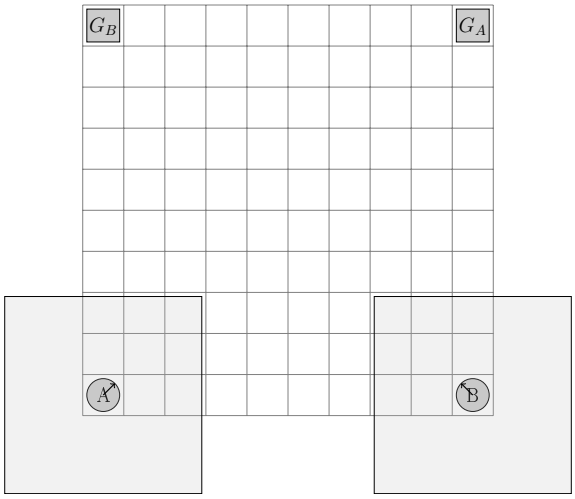


Figure 13: Visualisation of decentralized agents observation space. A and B represents agents with their observable environment as transparent. An indicator for the direction in which it can find its goal is also included.

grows very large as new agents are added. If one has a discrete set of positions, the number of combinations in which you can place k agents in n positions is $k!\binom{n}{k}$. With 5 agents and 40 positions the state space is already over half a million in size. The problem with scalability is a common issue for reinforcement learning problems and for multi agent reinforcement learning in particular.

One can decentralize the problem by letting the state space be only the distance between the vehicles, this would remove one dimension of the state space. This way the agents can work independently to maintain a gap to the next vehicle. With this simplified representation of the state space the number of states is only the number of agents, times the number of possible lengths between vehicles. With 5 agents and 40 possible positions, assuming that two agents cannot occupy the same position, this is only $35 \cdot 5 = 175$ possible states. Compared to over half a million states, this is a significant reduction. A Markov decision process with a state space that is simplified in this fashion is called a partially observable Markov decision process. The partial observations arise when multiple states in the process are represented in the same way.

An advantage of the partial observation of states is that the same partial observation could make sense in other problems. In the fully centralized approach, as soon as the size of the problem or the number of agents changes, the centralized solution might not be usable at all. The representation of a state or state-action pair changes and can no longer be mapped by the state value or state-action value function. The decentralized solution will still be able to maintain the same distance and will find a solution to the problem still, even though the solution might be suboptimal.

One of the disadvantages with the decentralized approach is that the agents can not take into account any naturally occurring disturbances. Traffic or inclines/declines in the route could disturb the distances, but only for a limited time, since these will affect all vehicles equally. For example if one vehicle reaches an uphill incline and slows down, the vehicle behind it will possibly catch up. When the first agent reaches the top of the hill and speeds up again, the gap between them will naturally return to its original state. In a sense there is a natural speed profile which the vehicles follow throughout the route. Information to recognize these types of patterns is not present in the partially observable representation of the state space and the agents will likely not learn to effectively handle such situations.

10.2.2 Sequential training with partial observations

In [9] the training is done in sequences. Initially one can formulate a training environment where the agent learns one specific task. Then the agent is proficient in this one task, one can let it train on another task or add another stage to the task. This is not possible when the agent has full observability since any change to one state or a change in the size of the state will make the agent's representation of the state invalid. With the partial observations as illustrated in Figure 13 one can train the agent in finding the goal, wherever it is in the grid. After that one could add another agent to the system and let them train at passing each other. If also using a neural network as in [9], the information gained when training one task propagate to the solution of another task in a more seamless way compared to when using tables. This is due to the structure of the neural network. If one changes a single element of an input vector to a neural network, only this column of the matrix of weights are effected. This is not the case with tabular methods where a change to one element of the input vector will change the whole row of the table. To make information of previous training valid in the modified state, one has to manually account for these situations and transfer what has been learned to the modified state.

10.3 Reducing the observation space

One of the biggest problems with multi agent reinforcement learning is keeping the state space or observation space small enough. Methods for keeping the state space small involve clever representations of the states and careful selection of the information to include. Other methods involve discretising the state space and testing which splits of the state space work the best.

Using different function approximations to estimate or classify a state is one approach to limit the amount of data one has to store. Popular methods involve approximations using neural networks such as a convolutional neural networks, recurrent neural networks or long short term memory networks. Convolutional layers together with long short term memory layers have been shown to perform very well in reinforcement learning. The idea is to classify the input vector using convolutional layers and then use a recurrent layer to recognize patterns in the sequence of classes.

There are several suggestions on how to reduce the size of the domain for the action-state function. Sutton [12] suggests tile coding as an option. The idea is to organise the state space in layers, one can discretise each layer and have the layers overlap. The result is a finer discretisation of the state space with a more compact representation. The idea is similar to that of a convolutional neural network where one uses a filter to map blocks of a matrix to a smaller block or a real number.

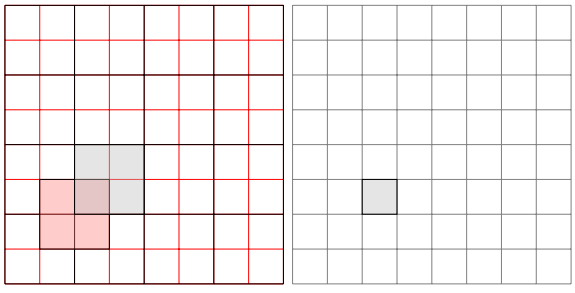


Figure 14: Exampe of tile coding, active areas represented as opaque areas. To the left a tile coding representation with two indicator functions with 4×4 elements, mapping each position to a state by combining the indication from the two discretisations. To the right, a finer grid giving the same accuracy but using a 8×8 grid. The tile coding implementation only require 32 elements while the one layered approach requires 64 elements.

As an example, consider a map over some terrain with coordinates as numbers with 3 decimals. One can discretise the coordinates to integers but this might give a coarse representation of the map, instead of using every 0.5 step in the coordinates mapping as a discretisation one can have a second layer which is shifted. By decreasing the step size to 0.5 the state space would require 4 times the data to represent the map. With the overlapping layer we receive the same accuracy with only twice the data. The idea is illustrated in Figure 14

References

[1] Richard Ernest Bellman. *Dynamic Programming*. Dover Publications, Inc., New York, NY, USA, 2003.

[2] Michael Bowling and Manuela Veloso. Rational and convergent learning in stochastic games, 2001.

[3] Lucian Busoniu, Robert Babuska, and Bart De Schutter. *Multi-agent Reinforcement Learning: An Overview*, volume 310, pages 183–221. Springer, 07 2010.

[4] G.R. Grimmett and D.R. Stirzaker. *Probability and random processes*, volume 80. Oxford university press, 2001.

[5] Junling Hu and Michael P. Wellman. Nash Q-learning for general-sum stochastic games. *Journal of Machine Learning Research*, 2004.

[6] Peter Karkus, David Hsu, and Wee Sun Lee. Qmdp-net: Deep learning for planning under partial observability, 2017.

[7] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. *Machine Learning Proceedings 1994*, pages 157–163, 1994.

[8] Hans Peters. *Game Theory. : A Multi-Leveled Approach.* Springer Texts in Business and Economics. Berlin, Heidelberg : Springer Berlin Heidelberg : Imprint: Springer, 2015., 2015.

[9] Guillaume Sartoretti, Justin Kerr, Yunfei Shi, Glenn Wagner, T. K. Satish Kumar, Sven Koenig, and Howie Choset. PRIMAL: pathfinding via reinforcement and imitation multi-agent learning. *CoRR*, abs/1809.03531, 2018.

[10] Howard M Schwartz. *Multi-Agent Machine Learning, A Reinforcement Approach*. Wiley, 2014.

[11] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

[13] Till Tantau. *The TikZ and PGF Packages.*