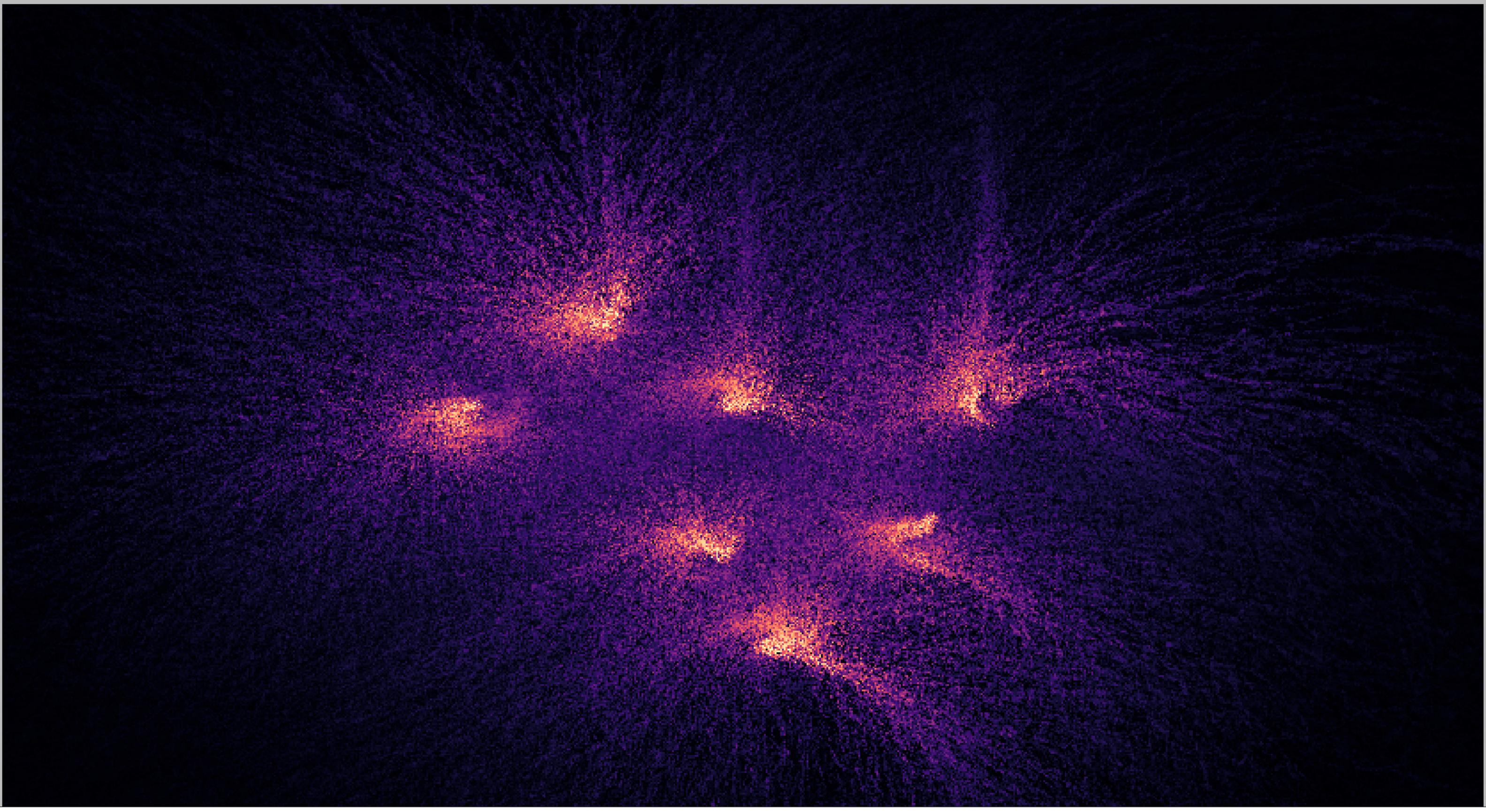
Faculty of Aerospace Engineering

A Multi-Agent Reinforcement Learning Approach to Air Traffic Control

Master of Science Thesis

Dennis van der Hoff

30 June 2020



TU Delft Delft University of Technology

Challenge the future

A Multi-Agent Reinforcement Learning Approach to Air Traffic Control

Master of Science Thesis

Master of Science Thesis

For obtaining the degree of Master of Science in Aerospace Engineering
at Delft University of Technology

Dennis van der Hoff

30 June 2020

Faculty of Aerospace Engineering · Delft University of Technology

Delft University Of Technology
Department Of
Control and Simulation

The undersigned hereby certify that they have read and recommend to the Faculty of Aerospace Engineering for acceptance a thesis entitled **"A Multi-Agent Reinforcement Learning Approach to Air Traffic Control"** by **Dennis van der Hoff** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 30 June 2020

Readers:

Dr. ir. J.M. Hoekstra

Dr. ir. J. Ellerbroek

Dr. ir. P.C. Roling

Acronyms

4DTBO	4D Trajectory based Optimization
AI	Artificial Intelligence
ATC	Air Traffic Control
ATF	Average Temperature Folding
CNN	Convolutional Neural Network
COMA	Counterfactual Multi-Agent Policy Gradients
CPU	Central Processing Unit
DDPG	Deep Deterministic Policy Gradient
Dec-MDP	Decentralized Markov Decision Process
Dec-POMDP	Decentralized Partially Observable Markov Decision Process
DLCQL	Deep Loosely Coupled Q-network
Double-DQN	Double Deep Q-network
DP	Dynamic Programming
DQN	Deep Q-Network
DRUQN	Deep Repeated Update Q-network
GPI	Generalized Policy Iteration
GPU	Graphicals Processing Unit
GRU	Gated Recurrent Unit
IAC	Independent Actor-critic
IQL	Independent Q-learning
KL	Kullback Leibler
LCQL	Loosely Coupled Q-learning
LoS	Loss of separation
LRN	Lenient Reward Network
LSTM	Long Short-term Memory
MADDPG	Multi-agent Deep Deterministic Policy Gradient
MARL	Multi-agent Reinforcement Learning

MC	Monte Carlo
MDP	Markov Decision Process
MLP	Multilayer Perceptron
MM	Minorization-maximization
MVP	Modified Voltage Potential
NF	Normalizing Flows
PBRS	Potential Based Reward Shaping
POMDP	Partially Observable Markov Decision Process
PPO	Proximal Policy Optimization
PS-TRPO	Policy Sharing Trust Region Optimization
RL	Reinforcement Learning
RLLIB	Reinforcement Learning Library
RNN	Recurrent Neural Network
RUQL	Repeated Update Q-learning
TD	Temporal Difference
TPU	Tensor Processing Unit
TRPO	Trust Region Policy Optimization
WDDQN	Weighted Double Deep Multi-agent Reinforcement Learning
WDQ	Weighted Double Q-learning

Contents

Part I

Preface

Preface

This document contains both the literature survey and final article of the thesis work surrounding the topic "A multi-agent reinforcement learning approach to air traffic control". The problem, as described in the title, is the application of reinforcement learning methods to the air traffic control problem. Reinforcement learning is a machine learning paradigm in which an agent, through trail and error, reinforces its own "behaviour" towards better performance. This performance is measured in the collected reward, which is given by the environment and received by the agent. An solution to the problem is found by optimizing a decision making model, called the policy. This decision making model is "trained" by collecting experience, experience in the form of state transitions, rewards and executed actions by the agent.

When I started working on this project there where a lot of blanks to be filled and trade-offs to be made. An example of such an initial trade-off was how to represent an agent in the air traffic control problem. Do I define centralized air traffic control towers as agents, or each individual aircraft as an agent? Even so, is a multi-agent approach even needed? A single "super" agent which decides for all aircraft simultaneously could also work. The advantages and disadvantages of these approaches are closely linked to other limitations as well, such as computational requirements. It is well known that deep learning methods consume lots of computational power. Reinforcement learning even more so, as it has to generate the data through simulations, which is subsequently used to train the model.

For example, when approaching the issue with the super agent as opposed to multiple simpler agents, this would require a significant higher computational demand. This agent would have an action space, i.e. the combination of all possible actions, that is significantly larger than that of multiple, simpler agents. However, the simpler agents could be well, to simple and not provide complex enough solutions.

Such trade-offs where common throughout the thesis work, and often there was no clear scientific basis or related work to fall back on. Most reinforcement learning problems are one of a kind, and have an unique environment and solution. Nevertheless, this "adventure into the unknown" made it an exciting problem to tackle. The in the end presented solution is one of many different approaches, balancing advantages versus disadvantages.

A lot of work went into the actual creating of the software platform to facilitate learning. The platform was proven to be capable of providing high data throughput, which enabled "quick" learning. Quick as in a sense that training results where achieved in about 34 hours. However, many previous iterations where slower, or had other problems. There have been training sessions where my desktop was running for about 8 days straight, in which after these 8 days the conclusion was; "No it's indeed not working as I hoped it would".

The scientific paper will contain the results of the thesis work, while the preliminary report can provide some more detailed background information on techniques used. However, there is no full match between information presented in the preliminary thesis and scientific paper, as techniques used gradually changed between the preliminary and final work.

Part II

Scientific Paper

A Multi-Agent Reinforcement Learning Approach to Air Traffic Control

D.E. van der Hoff
under supervision of J.Ellerbroek and J.M. Hoekstra
Delft University of Technology, Kluyverweg 1, the Netherlands

***Abstract*—Reinforcement learning has shown that, when combined with deep learning techniques, is able to provide solutions to complex and dynamic problems. Air traffic control is considered to be an problem of such nature, which bears the question to mind; Can reinforcement learning be used to solve the problem of air traffic control. This work explores the applicability of reinforcement learning to the air traffic control problem by setting up a distributed system for training and experience collection. The problem is formulated as a decentralized system. Each aircraft is modeled as an agent that uses local observations while being limited to heading changes only. During learning, information about the actions of surrounding agents are added. It is shown that for low air traffic density scenarios the model is able to provide collision avoidance and approach the correct runway under realistic limitations. However, due to the lack of global coordination and limited modeling of spatial relation between states this method is unable to solve more complex and higher air traffic density situations.**

I. Introduction

Air traffic control is a ground service established to provide guidance to airborne craft. Furthermore, air traffic controllers prevent collision and organize the flow of traffic as well as provide many information services. Due to the ever growing density of air traffic, optimized methods are needed to reduce the workload on the air traffic controllers as well as increasing the efficiency of the used airspace. An example of such a system is the concept of 4D-Trajectory Based Operations (4DTBO) [1]. In short, aircraft follow a predetermined, conflict free time-based contract that is known among all other aircraft. This method relies on the precision of models to be able to make accurate time based predictions. Uncertainties in model predictions are the result of stochastic phenomena, such as weather conditions and pilot errors. Also, the governing principles behind 4DTBO and other such methods are limited by the creativity of the author.

In search of a more exotic approach, and avoiding said limitations, model free self-learning algorithms could offer new insights and solutions. Reinforcement learning is such a self-learning algorithm. Reinforcement learning is based around the principle of trail and error and self reinforcement. An agent, which is an actor that can exert control through actions, exists within a certain environment. The definition of an agent can differ vastly depending on the problem formulation. With regards to the air traffic control scenario, this can be a centralized entity such as an air traffic control tower or an decentralized entity such as a single aircraft. What action an agent performs depends on its policy. The policy is the collection of learned behaviour, which translates a certain environmentally determined state into an action. The goal of reinforcement learning is to construct and improve this policy in such way that it maximizes the cumulative reward. Reward is a form of feedback from the environment, rewarding the agent positively for good behaviour, and penalizes bad behaviour. This interaction of state, action and subsequent reward is the bedrock of reinforcement learning. Modern reinforcement learning approaches utilize deep learning techniques to enable the application of reinforcement learning techniques on more complex problems. What sets deep reinforcement learning apart from other machine learning paradigms, is that deep reinforcement learning influences the underlying distribution that is being approximated. This in contrast to for example image classification, where the underlying distribution is stationary.

Recently reinforcement learning got notable attention due to the successes of the deep reinforcement learning agent AlphaGo [2] and OpenAI Five [3]. AlphaGo was able to defeat Lee Sedol in the classical boardgame of Go, and OpenAI Five attained superhuman performance beating the professional e-sports team OG in the video game DOTA. Sparked by the success of these agents this paper explores the application of deep reinforcement learning techniques to the problem of air traffic control. As impressive as the results obtained by AlphaGo and OpenAI Five are, a lot of computational power was required to find a solution. OpenAI Five, while being a far more expensive environment than the one used in this work, still required 180 years worth of experience each day [4], while taking about 10 months to achieve top performance.

Simulation of the airspace is provided by BlueSky, an air traffic management simulation platform. [5] BlueSky offers extendability, parallel computing and accurate models to correctly simulate aircraft dynamics making it suitable for (distributed) reinforcement learning. To facilitate the implementation of various state-of-the-art reinforcement learning algorithms as well as the handling of the various streams of data, RLlib [6] is used. RLlib is an open-source library for reinforcement learning, which offers a dedicated platform for multi-agent reinforcement learning problems as well as computational scalability.

This paper explores the applicability of reinforcement learning to the air traffic control problem, by using an approach of decentralized execution and centralized learning. The offered solution provides basic avoidance in low density situations and efficient navigation towards the goal states. However, due to the decentralized and un-cooperative setting used solutions diminish when air traffic density increases. Nevertheless, this work presents detailed insight into an approach to solve the air traffic control problem, while also providing an stepping stone to more global or advanced solutions.

II. Background information

A. Reinforcement Learning

The most commonly used mathematical framework for sequential decision making problems are *Markov Decision Processes*. A Markov Decision Process (MDP) is a classical formalization of sequential decision making. Each time step, the process is in a certain state, $s_t \in S$, the agent (the decision maker) chooses an action, $a_t \in A$, which transitions the agent into the next state, $s'_t \in S$. After the state transition, the agent receives a reward, $r_t \in R_a(s, s')$. Additionally, for each action there is the probability that a certain action leads to a certain state, formalized in the state transition probability, $P_a(s, s')$.

The goal of the agent is to maximize the cumulative received reward over the full trajectory, which is called the return as seen in equation 1. G_t is the return, γ is the discount factor, which balances the importance between future or immediate reward.

$$G_t = \sum_{t=0}^{\infty} \gamma^t r_t \tag{1}$$

However, the received reward depends on the action the agent takes in each state. This behaviour is called the *policy* of the agent. The policy is a mapping between state and action, and is defined as $\pi(a|s)$. The value function is defined as the *expected return* under policy π as seen in equation 2. $v_\pi(s)$ is the value function under policy π, G_t is the return and s the current state.

$$V_\pi(s) = \mathbb{E}[G_t | S_t = s] \tag{2}$$

Concurrently, the action-value function can be defined, which relies on both the state s and the action taken a as seen in equation 3.

$$Q_\pi(s, a) = \mathbb{E}[G_t | S_t = s, A_t = a] \tag{3}$$

An solution is reached by finding a policy π that maximizes the return from the start state. A method of finding this optimal policy is by iteratively estimating the value functions. An example of a value based solution method is *Q-learning*, which can be seen in equation 4. α is the step-size.

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \tag{4}$$

Q-learning iteratively updates the value function for each state-action pair. When the value function is fully converged, the optimal policy is simply a greedy policy, i.e. always taking the action that results in the highest value. However, this method requires multiple visitations of each state-action pair to be able to estimate the correct values for all states. For problems with a substantial state and action space it is unfeasible to attain convergence, and generalisation is required. This is where function approximation methods such as neural networks are used. This allows for generalisation between states and actions. One such combination of deep learning and reinforcement learning is called *Deep Q-learning* [7], where Q-values are approximated; $Q(s, a|\theta) \approx Q(s, a)$. The parameters θ are optimized by minimizing the loss function, as seen in equation 5.

$$L_i(\theta_i) = \mathbb{E}\big[r + \gamma \max_{a'} Q(s', a'|\theta_{i-1}) - Q(s, a|\theta_i)\big] \tag{5}$$

B. Policy Gradients

Opposite to value-based methods such as Q-learning, are *policy gradient* methods. Instead of solving the value function and extracting the policy from the converged value function, policy gradient methods directly adjust the parameters of the policy; $\pi(a|s;\theta) \approx \pi(a|s)$. This is done by formulating a loss, and using gradient decent methods to adjust the policy parameters. The fundamental algorithm underlying all other policy gradient methods is the *policy gradient theorem* [8], as seen in equation 6. The policy gradient theorem has an intuitive appeal.

$$\nabla J(\theta) = \mathbb{E}_\pi\big[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s, a)\big] \tag{6}$$

As can be see in equation 6, the gradient updates depend on the value of $Q^\pi(s, a)$. This gives notion to a school of algorithms called *actor-critic*. In this case, the actor represents the policy π_θ which is updated using equation 6. However, concurrently $Q^\pi(s, a)$ is estimated using function approximation; $Q^w(s, a) \approx Q^\pi(s, a)$. w are the model parameters.

When optimizing the policy using equation 6, excessively large policy updates should be avoided. Unbounded and large update steps often lead to instability, and weak convergence properties. Avoiding large policy updates is done by imposing an limit on the difference between the old and new policy with each update step. The method of limiting this update step is called *Proximal Policy optimization (PPO)* [9]. Schulman et al. define $r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}(a|s)}}$. Limiting the update steps is achieved by ensuring this ratio stays close to 1 during updates. This results in the rewritten policy gradient loss as seen in equation 7. ϵ is the step-size limit and $\hat{A}_t$ is the estimated advantage function.

$$L^{CLIP}(\theta) = \mathbb{E}_t\big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\big] \tag{7}$$

The classical definition of the advantage function is: $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$. The intuition behind the advantage function is that the action value is normalized relative to the overall value of the state.

Often, the loss is augmented with an additional *entropy* [10] term to improve exploration and smoother convergence. Entropy is defined as the average amount of information received from a probability distribution, which can be calculated as shown in equation 8. $H(X)$ is the entropy of the random variable X, x_i are possible outcomes and $P_X(x_i)$ being the probability of that outcome. Entropy is a measure of information; a low probability event carries more information compared to an event that has an high probability of occurring.

The addition of entropy increases the chance of reinforcing low probability actions, avoiding premature convergence to a subset of proven and more frequent occurring actions.

$$H(X) = -\sum_i P_X(x_i) \log_2 P_X(x_i) \tag{8}$$

C. Multi-Agent Reinforcement learning

Multi-agent reinforcement learning (MARL) consists of multiple agents interacting with the same environment, and each other. There are a few known "hurdles" within reinforcement learning, which are amplified when considering the multi-agent domain. An example is the *credit assignment problem*. In the single agent domain it is difficult to determine what actions in a certain sequence of states attributed positively or negatively to received reward. In the multi-agent settings, this effect is amplified because the received reward can be the result of an action the agent took, or the result due to the action of another agent.

This ambiguity of influence on the environment by either the agent itself or other agents, increases the *non-stationarity* of the environment. Furthermore, all agents evolve with each policy update which further increases non-stationarity.

Lastly, the system will likely be partially observable. It is not realistic for each agent to have full observability of the system; the observations space grows exponentially with each added agent when fully observable, exploding the state space. So partial observability is needed, meaning that the agent only considers its immediate surroundings. Luckily this is an realistic setting for the ATC environment. Aircraft are flying towards their far-off goal, while locally solving conflicts.

Recently a much used paradigm is that of *centralized training with decentralized excecution*.

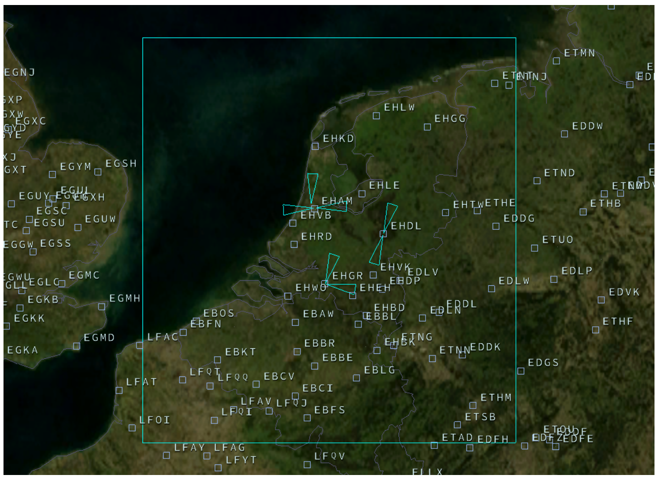


Fig. 1. An overview of the simulated BlueSky environment, showing the blue triangle goal states. The blue rectangle describes the area in which aircraft are generated.

The idea behind this approach is that when training, each agent has access to a more global state, while during execution agents rely on local observations only. Using extra state information during training has proven to increase convergence towards an solution. [11]

III. Problem Description

The goal of this research is to investigate the applicability of reinforcement learning to an air traffic control scenario. To maintain feasibility, the research scope is limited, which will be discussed in the next sections. First the environment used and limitation of the environment are discussed. After that, the action space is expanded upon while finishing with the definition of the reward structure.

A. Environment

The environment is simulated in BlueSky [5] and centered around the airspace of the Netherlands. There are a total of 7 goal states defined as an approximation of an ILS intercept beam; 3 runaways at EHAM, 2 runways at EHGR and 2 runways at EHGL. An overview can be found in figure 1. The goal state is reached when an aircraft is in the drawn triangle shaped area, while also approaching with the correct heading.

Each aircraft is of the same type, a "B747-800". The height of the aircraft is set to 5000 feet, and will stay constant as there are no height commands issued. Aircraft velocity is also kept constant, and is initialized at 250 knots. These values are chosen to approximate an landing approach.

A minimum horizontal separation is required between aircraft. If this minimum is exceeded, a *loss of separation* occurs. The minimum loss of separation distance is 5 nautical miles horizontally around the aircraft.

B. Observation space definition

The perspective taken towards the problem is that each individual aircraft makes decisions based on their local observations; each individual aircraft is an agent. The observation space is homogeneous for all agents. Due to this property, all agents share the same policy and subsequently use all their individual trajectories to improve this policy. The local observation space of each agent consists of roughly two parts, personal information and the information of n neighbouring aircraft. The agents personal information is given in the following tuple: s_p = (rwy_ID, lat, lon, hdg, rwy_dist, rwy_qdr). Table I provides an description for these states.

TABLE I
Overview of agent personal state information s_p

State	Description	Unit
rwyID	Runway goalstate identifier	int
lat	Current latitude	deg
lon	Current longitude	deg
hdg	Current heading	deg
rwy_dist	Distance to goal state	nm
wpt_qdr	Relative bearing towards the goal state	nm

The observation space is extended by considering the n closest agents. The full added state is a repetition of s_n, given

TABLE II
OVERVIEW OF NEIGHBOURING AGENT STATE INFORMATION s_n

State	Description	Unit
dist_plane	Distance to neighbouring agent	nm
qdr_plane	Relative bearing towards neighbouring agent	nm

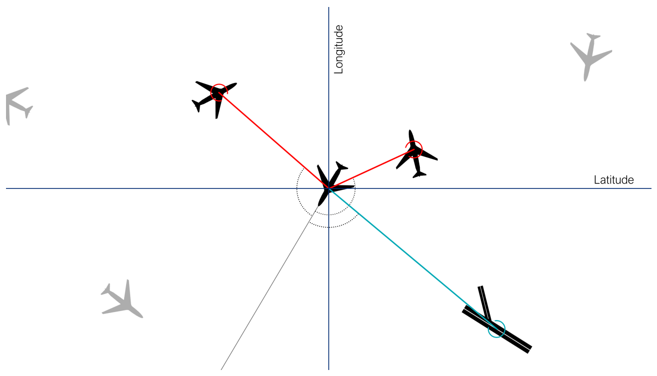


Fig. 2. Visual representation of the observation space of each agent. Currently taking into account two neighbouring aircraft.

in the following tuple $s_n = (\text{dist\_plane}, \text{qdr\_plane})$. Table II describes both state values.

This results in the complete local observation for each agent; $s_i = (s_p, s_1 ... s_n)$. s_n is ordered from closest to furthest away neighbouring agent before being combined with s_p. Figure 2 is an visual representation of the local observation space of each agent. In this example $n = 2$, meaning that two neighbouring aircraft being observed by the agent in the center.

C. Action space definition

The action space of each agent is limited to heading changes only. The heading changes are done relative to the current heading; like a steering wheel rather than absolute heading selection. Relative headings compress the action space significantly. Action space representation is discrete. Reason for this is two-fold; continuous actions have a higher resolution, but this high resolution is not needed for this scenario. Secondly, discrete actions allow for an easier multi-modal representation of the action space. Often Gaussian distributions are used to represent the continuous action space. However, equal probability of going either left or right cannot be properly captured by an unimodal representation such as a Gaussian. Discrete actions offer an simple solution. However, when using discrete action representation spatial information of the actions are lost. Spatial information refers to the ordering of values; an action of 10 degrees is more than 5 degrees, and less than 15 degrees. To restore this ordering, a *ordinal distribution network architecture* [12] is employed. The amount of discrete actions is 7, equally distributed on a bandwidth of $[-15, 15]$ degrees. Empirical results and reasoning to substantiate the different action space representations can be found in appendix A.

D. Reward signal definition

Sparse rewards are used, in contrast to reward shaping. Sparse rewards is an reward system that results in an zero reward signal for most transitions, while only receiving positive or negative rewards when reaching the goal state. Reward shaping on the other hand is a technique that ensures a reward signal is received for all transitions. Sparse rewards encourage more exotic behaviour, as there is less interference from the author. Reward shaping is difficult to correctly implement in a multi-agent multi-goal reinforcement learning setting, and often leads to sub-optimal solutions. These sub-optimal solutions are a result of the change in underlying solution equilibria. [13]

Using sparse rewards however often good explorations properties. This is inherently solved by the method of simulation, due to randomization of the initial state. Also, the generation of multiple agents covers an substantial amount of the state space in each episode. The reward for an loss of separation is minus one, while reaching the goal state results in a reward of one. To improve exploration around the goal states significantly, an penalty of negative zero point five is given to agents that move beyond 180 nautical miles away from their goal state.

IV. EXPERIMENT SETUP AND RESULTS

The problem is implemented as a distributed system. In this distributed system there is communication between BlueSky and a policy server. The policy server facilitates the deep learning models that are being optimized. The policy server receives the observation and reward, while returning an action. The action is the result of an forward pass over the policy model by using the observations as input, and receiving the corresponding action as output. Figure 3 shows an simplified overview of the interaction between BlueSky and the policy server.

BlueSky is used to sample observations and rewards as described in section III. Each agent takes into account 5 neighbouring aircraft. The policy server uses these trajectories to optimize the actor and critic. Both the actor and critic are

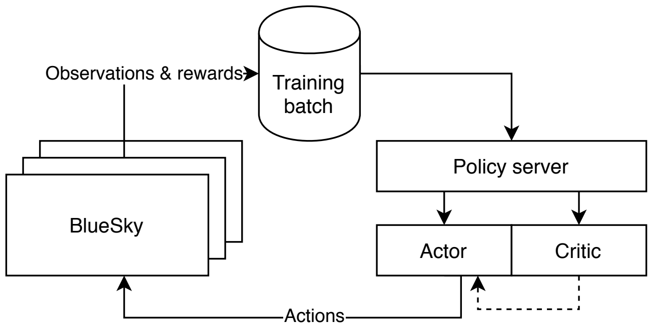


Fig. 3. Overview of the used distributed system. To the left the multiple threads of BlueSky are present, collecting experiences. These experiences are collected in a batch and when filled up are use to optimize the critic and actor. During experience collection, each time-step BlueSky queries the policy server on what action each agent should take depending on the given observations.

multilayered perceptrons with 2 hidden layers containing 256 neurons each. The actor uses local observations only, while the critic is a centralized critic. On top of the observation space previously defined, the critic is also presented with actions of neighbouring aircraft.

The clipped loss from *Proximal Policy Optimisation* [9] is used as loss function. Also, entropy regularization is included to improve exploration and avoid premature convergence to a subset of actions [10]. As gradient decent optimizer ADAM [14] is used. Both entropy regularization and the use of ADAM as optimizer are widely used for many different reinforcement learning problems. An empirical comparison is made to justify the use of these mechanisms, which can be found in appendix B. This appendix also includes other hyper-parameters used for the experiments. The output of the critic network is further processed by using *generalized advantage estimation* [15].

To evaluate and show the performance of the learned model two scenarios are used. The first scenario is an light scenario showing model capabilities in guiding aircraft to a single goal state. The second scenario involves all goal states and aircraft being guided towards these goal, as well as merging streams of traffic.

A. *Training*

During training each episode consisted of 25 aircraft, randomly spawned in an squared shaped area as seen in figure 1. Each aircraft is assigned a runway as goal state, with all runways being uniformly distributed amongst available aircraft. An action would be selected every 4 seconds. When an loss of separation between aircraft occurs or when the goal state is reached aircraft are removed from the simulation. Each episode has a maximum duration of 1500 time steps, or when all but 6 aircraft landed. The premature ending of the episode was done to avoid having the model learn how to deal with the padded values in the observation space. When fewer than 6 aircraft remain in the environment, the closest neighbours part of the observation space, s_n, is filled with dummy values. It was shown that eventually the model would be able to converge even with dummy values, but avoiding it all together significantly lowered the required training time.

Training was done by collecting experience with 22 parallel workers. Gradient updates where done after collecting an batch of 200,000 state transitions. This means that a variable number of episodes where collected before gradient updates where done. Training was finished after approximately $3.4x10^9$ time steps where used for training, which took about 32 hours.

Figure 4 shows the training results. Top left shows the mean reward collected per episode. The maximum attainable reward is 19. The top right figure shows the mean episode length. Bottom left shows the mean amount of losses of separation per episode. Bottom right shows the mean amount of aircraft that landed at the correct runway.

Training can be split up in roughly three phases; First phase is complete random behaviour which occurs at the start of training. Second phase is when an sub-optimal solution is found to the problem. During earlier training trails the model

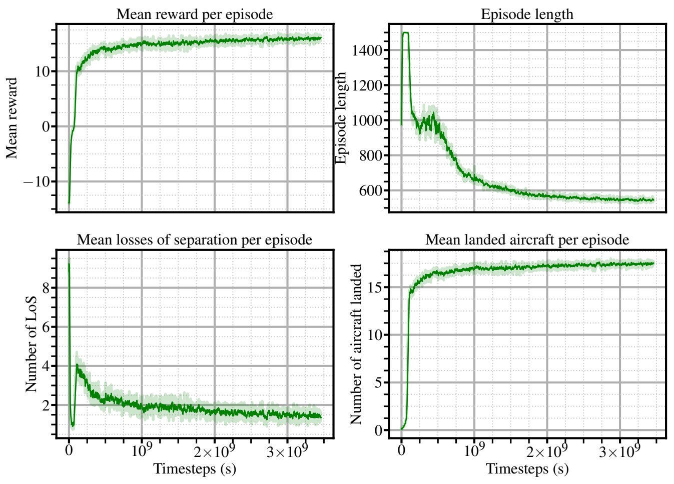


Fig. 4. Results of training for approximately $3.4x10^9$ time steps. Top left shows mean reward, top right shows episode length. Bottom left shows amount of losses of separation and bottom right shows number of landed aircraft.

would often be stuck at this sub-optimal solution due to lack of proper exploration. Third phase consists of fine tuning the found solution. The first and second phase are highlighted in figure 5.

As can be seen in figure 5 initially the aircraft behaves randomly. This behaviour results in an significant amount of losses of separation, which simultaneously results in a strong negative reward signal. Interesting to see is that also the episode length is relatively short because all aircraft get deleted. After the first phase, an sub-optimal solution is quickly found in which all agents avoid each other. This often resulted in flocking behaviour. Flocking means that all aircraft start following the same heading as the neighbouring aircraft, and move like a flock of birds. This behaviour ensures that

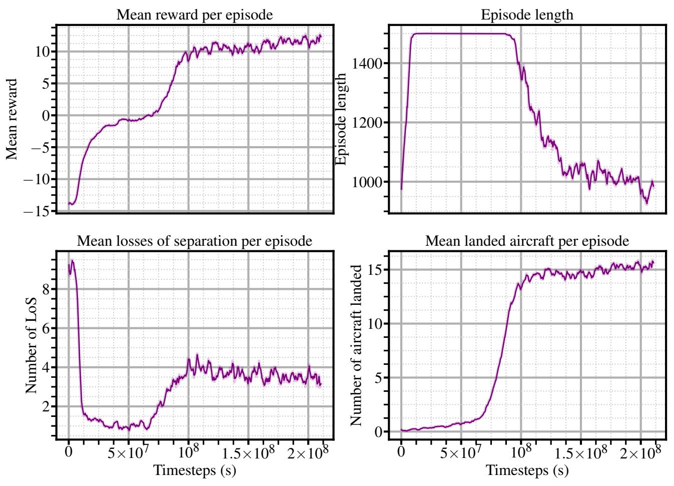


Fig. 5. Training results that highlight the first and second phase of the training. Top left shows mean reward, top right shows episode length. Bottom left shows amount of losses of separation and bottom right shows number of landed aircraft.

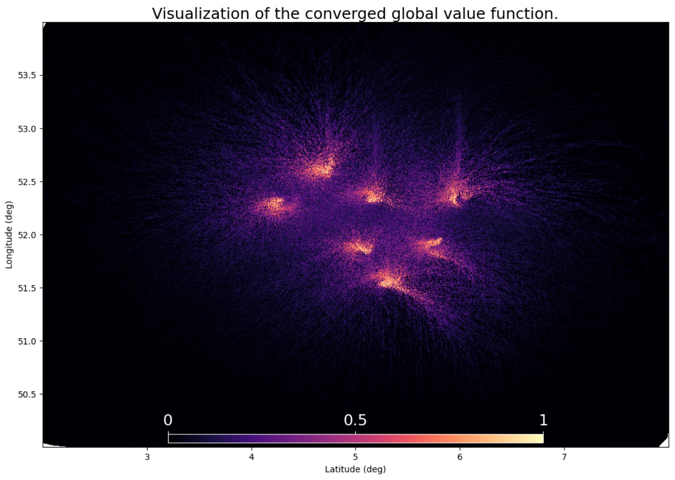


Fig. 6. Visualization of the converged value function. Note that this figure only highlights the global portion of the value function, ignoring interaction between agents.

there are a low amount of losses of separation, however there is also little gained positive reward.

After sufficient exploration the model is able locate the goal states which are then exploited, resulting in a quick ramp up of gained reward. However, this also introduces an increase in losses of separation as agents have to maneuver closer together around the goal states.

The fine tuning in the last phase significantly influences the mean episode length, having the model find more optimal routes to the goal states. Also an slight increase in mean reward, and decrease in losses of separation occur during this phase.

Figure 6 shows the converged global portion of the value function. The global portion aims at the part of the observation space which is not influenced by proximity and actions of other agents. As expected the states surrounding the goal states have an high value.

B. Evaluation

In this section insight is given in the model performance evaluated on two scenarios. The scenario will be evaluated on a set of two simple metrics; Number of aircraft landed and number of losses of separation, each accompanied with an overview of flown trajectories. Figure 7 shows the metrics for the two scenarios in question. Aircraft are randomly generated with an interval of approximately 60 seconds.

The first scenario shows no aircraft crashed, and all aircraft landed. As can be seen at the left hand side of figure 8, roughly two lanes where formed towards the runway. Interesting to see is that agents in the south-eastern lane have right of way when closing in to the runway. Whenever two aircraft would risk an loss of separation when simultaneously reaching the runway, the aircraft on the north-western lane would avoid collision and loop around whenever the other aircraft would land. This can be seen by the clockwise spiral that is made close to the runway.

The right hand side of figure 8 shows the second scenario. It can be seen that the model is able to distinguish the different goal states assigned to the aircraft, and is able to fly towards them. What is also evident is that for the goal states, similar to that in the first scenario, lane forming occurs. Also, the pattern of clockwise circles are present at most of the goal states. Looking at figure 7 it can be seen that losses of separation are increased in comparison to the first scenario. Most evidently, when the first aircraft started to land losses of separation occurred. A significant portion of losses of separation occurred at the west runway of EHAM; the blue ”tornado” of lines in the top center of figure 8. Aircraft would reach this goal state, but would come stuck in this clockwise ”tornado” that would fill up with more aircraft trying to reach the goal state. As each additional aircraft was increasingly interfering with other aircraft ability to approach the goal state correctly, this situation would eventually result in a significant portion of the losses of separation.

It can be seen that behaviour has developed in which the approach to the runway is often made by making clockwise turns. Another point of interest is that the right-of-way behaviour occurs at most of the runways entries. When two or more aircraft would approach the same runway while risking an loss of separation, the aircraft to the left of the aircraft that had right-of-way would avoid this situation by taking an ”outer ring” in the often seen clockwise approach.

V. Discussion

This section will discuss the results of the training, as well as the evaluation the performance on the scenario.

The multi-goal finding capabilities are highly depending on proper exploration of the environment. During training, an reward of minus zero point five was given to agents that would fly further away than 180 nautical miles from the goal state. In the early stages of the project, this penalty was not added, which led to extremely poor exploration properties as aircraft would settle for the sub-optimal flocking behaviour.

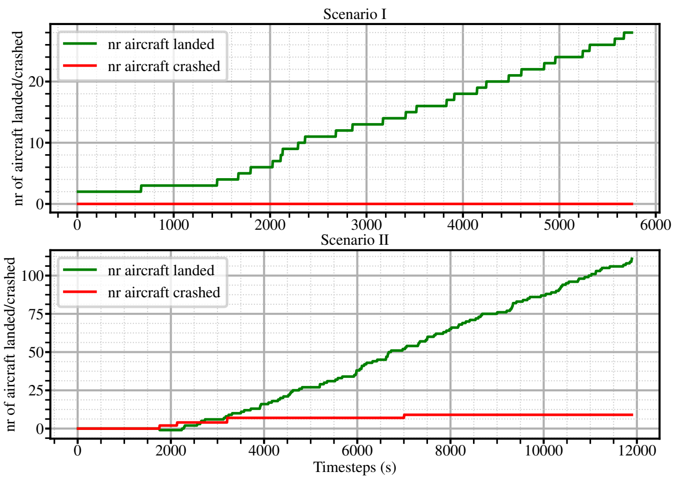


Fig. 7. Scenario results showing the amount of aircraft landed and crashed.

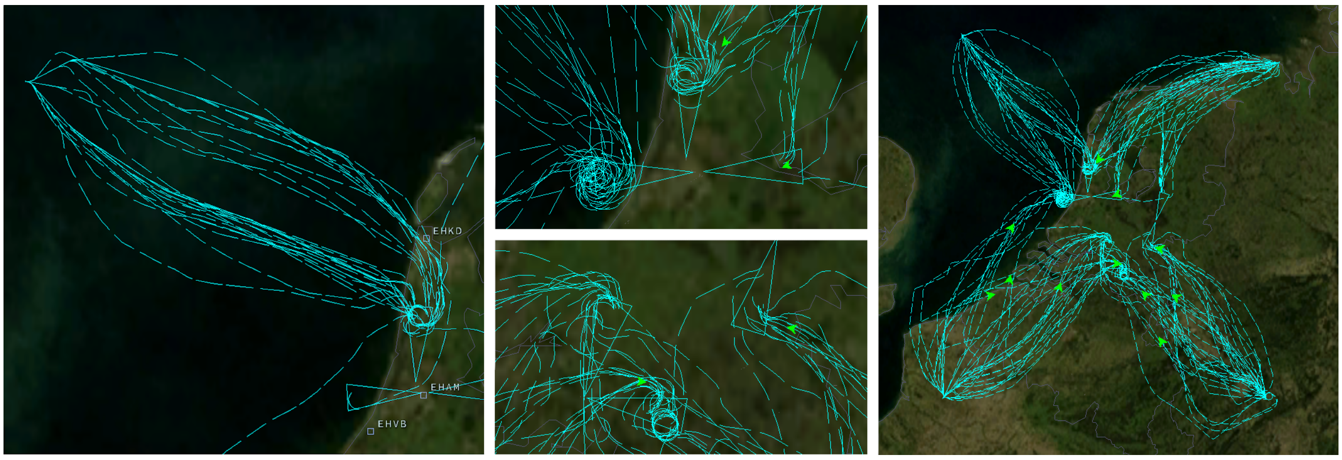


Fig. 8. The results of scenario I are shown on the left hand side. Scenario II results are shown in the centre, and right hand side of the figure. The centre two figures highlight the spiral like behaviour at the runways from scenario II.

However, after adding this penalty due to the many concurrent simulated scenarios goal states where discovered rapidly. This exploration through numbers also allows for the use of sparse rewards; no reward shaping is necessary as exploration is ensured through numbers and randomized initial states.

The maximum attainable reward is 19 each scenario, while the model converges to an mean reward of approximately 17. This is a result of the sub-optimal behaviour when there are to many aircraft approaching the runway at the same time, as seen in the scenarios. Episode length, without explicitly being rewarded, also decreased. This is a wanted result, because penalizing episode length would inhibit the finding of exotic solutions. For example, if an longer trajectory would be needed to solve an certain situation, the reward signal could be convoluted due to the reward depending on episode length. However, seen in the scenario examples the taken paths where sub-optimal, as a single straight lane would be sufficient for most approaches. This could be the result of no episode length reward, or other factors such as state definitions.

As seen in the scenarios the solution found works well under low density air traffic scenarios. The model is able to avoid losses of separation when the density is low, by using this right-of-way behaviour and lane forming. However, when the aircraft density increases, aircraft get stuck due to the lack of coordination. This was clearly seen in the ”tornado” behaviour at some runways. Each agent is exerting individualistic behaviour, which results in an sub-optimal solution. The reason for this lack of coordination and individualistic behaviour is two fold; The reward definition and state representation. The agents are all trained with the same policy, and rewarded positively for landing, and negatively for losses of separation. When agents collide, they both receive an negative reward. This results in sharing of negative reward amongst the agents. However, when a single agent lands only this agent gets an positive reward, and neighbouring agents that are queuing up to land do not receive any reward. Gaining no reward for neighbouring agents that land is unsensible; other agents landing makes room for another agent to approach the runway which directly benefits other agents.

As for the state representation, agents are only aware of their current state and do not keep track of a history of states. Also, the state itself does not include any information about possible trajectories. If agents would be able to model the resulting trajectory of a sequence of actions, more elegant solutions could emerge. For example, aircraft could disperse out of the tornado and reform an approach sequence that would allow for landing. However, with the current representation of the state this is unfeasible.

Another limitation imposed on the problem is that of limiting the action space to heading changes only. It is entirely probably that velocity changes are required to avoid certain deadlock scenario, or at least ease the finding of an solution. Offering velocity changes as a second action is interesting, as it makes the problem more difficult to solve by adding more possible combinations of actions. But on the other hand, velocity changes could offer an easy solution to certain conflict scenarios.

VI. Recommendations

First off, the expansion of the state representation. There are multiple ways of achieving this goal. An common method is the use of deep learning network architectures that are able to capture spatial relations in data. An recent and often used architecture components are that of *Long Short-Term Memory* [16] or *Gated Recurrent Unit* [17]. Both deep learning methods enable the learning of spatial features time series, which would allow the network to be trained on sequences of states and actions. Another method would be to collect multiple subsequent time steps, and present these as a single state. This is done for example in Deep Q-networks [10].

Including global rewards shared amongst agents incentives cooperation between agents. However, using global rewards creates noise on the reward signal, as it becomes increasingly difficult to determine what agent and what specific actions of each agent contributed towards the global reward. A solution to this is the use of a *counterfactual baseline* as seen in *Counterfactual Multi-agent Policy Gradients* [18]. The counterfactual baseline reduces noise in estimating the correct personal contribution towards the attained global reward.

Communication could also be key in solving the global coordination problem. Splitting up the problem and implementing actual air traffic control agents which communicate with the aircraft agents would create for a more realistic scenario, as well as an interesting problem. An example of communication between multiple agent is *Reinforced inter-agent Learning* [19]. Reinforced inter-agent learning presents a method of communication that is differentiable. This implies that the agent that got the communication message, in this case an aircraft, can provide "feedback" to the communicating agent, the air traffic control agent. It was already shown that the current distributed system allows for communication structures, as changing the goal state of an agent during simulation worked, and the agent changed course to this new goal.

What is shown is that the decentralized aircraft agents can control, avoid and solve simple problems based solely on local observation. A method to increase the global coordination an air traffic control agent could determine what runway and for example time slot a certain agent would receive. This implies that the observation space would be extended for each agent. The feasibility of assigning time slots to agents is closely related to the ability of the agents to predict and model it's own trajectories.

Currently, for training completely randomised starting locations are used for model learning. However, these scenarios are often unrealistically chaotic and possibly not include enough of the more realistic aircraft merging and sequencing scenarios. However, the complete random starting locations do benefit exploration. An suggestion would be to make an best of both worlds. Instead of spawning an set amount of agents, agents are randomly spawned over time. On top of the randomized scenarios, more controlled starting conditions can be included to highlight the more realistic occurrence of inbound traffic. The chaotic collection of experience is important for exploration, but also to avoid over-fitting of the model. Over-fitting would make the model "remember" what to do in each state, as opposed to generalise between states.

VII. Conclusion

This work explored the possibility of applying deep reinforcement multi-agent techniques on the problem of air traffic control. This is realised by implementing an decentralized multi-agent approach, where agents make decisions based on their local observations. Goal states where realistically defined, constraining both location and heading for the approach. Also collision avoidance is taken into account by penalizing losses of separation and proper observation space definition. Agents had limited control, only heading changes where taken into account. Action space representation was discrete, and an ordinal distribution network architecture was employed to retrieve the lost ordinal information.

To facilitate learning, BlueSky was employed in an distributed system which enabled training and large-scale experience collection. An actor-critic method is used which employed state-of-the-art mechanisms in combination with the distributed system to provide good training results. During training agents where trained on randomized scenario; starting location and goal states where randomly assigned. The evaluation scenarios presented a more realistic settings. During evaluation the deep learning models achieved good performance, especially when considering that the models where trained on randomized scenarios.

It was shown that the current approach to the problem was able to solve low density collision avoidance and multi-goal state solutions. The model showed an low level set of rules such as the right-of-way and lane forming to approach the problem. However, when air traffic density increased and problems that require coordination between agents emerged, the method was not able to resolve these issues flawlessly. The recommendations made should enable the model to capture spatial relations between observations, and posses the ability to plan ahead. On top of these requirements communication in some form is needed to tackle the coordination problem, as well as present global rewards to incentives cooperative behaviour between agents.

References

[1] H. Erzberger, *Transforming the NAS : The Next Generation Air Traffic Control System*, 2004, no. October.

[2] D. Kumaran, D. Hassabis, T. Graepel, M. Lai, D. Silver, M. Lanctot, L. Sifre, T. Hubert, K. Simonyan, I. Antonoglou, T. Lillicrap, J. Schrittwieser, and A. Guez, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[3] OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with Large Scale Deep Reinforcement Learning," 2019.

[4] OpenAI, "OpenAi Five," 2018. [Online]. Available: https://blog.openai.com/openai-five/

[5] J. M. Hoekstra and J. Ellerbroek, "BlueSky ATC Simulator Project: an Open Data and Open Source Approach," *7th International Conference on Research in Air Transportation*, 2016.

[6] E. Liang, R. Liaw, P. Moritz, R. Nishihara, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "RLlib: Abstractions for distributed reinforcement learning," *35th International Conference on Machine Learning, ICML 2018*, vol. 7, pp. 4768–4780, 2018.

[7] M. R. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, "Playing Atari with Deep Reinforcement Learning," *IJCAI International Joint Conference on Artificial Intelligence*, vol. 2016-Janua, pp. 2315–2321, 2016.

[8] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," 2020.

[9] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," 2017.

[10] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," vol. 48, 2016.

[11] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *Advances in Neural Information Processing Systems*, vol. 2017-Decem, pp. 6380–6391, 2017.

[12] Y. Tang and S. Agrawal, "Discretizing Continuous Action Space for On-Policy Optimization," 2019.

[13] S. Devlin, "Potential-Based Reward Shaping for Knowledge-Based, Multi-Agent Reinforcement Learning," no. July, p. 112, 2013.

[14] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pp. 1–15, 2015.

[15] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-Dimensional Continuous Control Using Generalized Advantage Estimation," pp. 1–14, 2015.

[16] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[17] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pp. 1724–1734, 2014.

[18] J. N. Foerster, G. Farquhar, T. Afouras, N. Nardelli, S. Whiteson, and U. Kingdom, "Counterfactual Multi-Agent Policy Gradients," 2007.

[19] J. N. Foerster, Y. M. Assael, N. De Freitas, and S. Whiteson, "Learning to communicate with deep multi-agent reinforcement learning," *Advances in Neural Information Processing Systems*, pp. 2145–2153, 2016.

[20] A. Graves, "Generating Sequences With Recurrent Neural Networks," pp. 1–43, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850

[21] P. Henderson, J. Romoff, and J. Pineau, "Where Did My Optimum Go?: An Empirical Analysis of Gradient Descent Optimization in Policy Gradient Methods," vol. 14, no. October, 2018.

Appendix A
Action Space Comparison

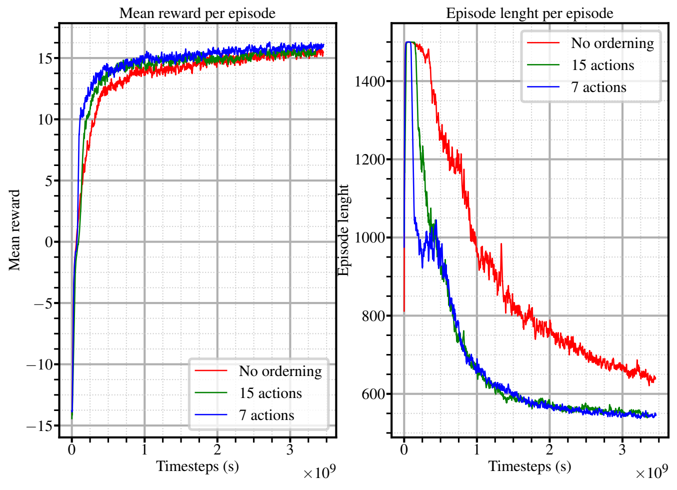


Fig. 9. Training results that highlight the first and second phase of the training. Top left shows mean reward, top right shows episode lenght. Bottom left shows amount of losses of separation and bottom right shows number of landed aircraft.

An empirical comparison is made between different action space definitions; The impact of using vanilla action space discretization versus the implementation of the *ordinal distribution network architecture* [12]. Concurrently the influence of the amount of discretization bins is investigated. Two different amount of discretization bins are compared; 7 and 15 different actions. The maximum and minimum effectiveness of the actions stay the same; $[-15, 15]$ degrees relative heading change. This resulted in two action space representations:

$$A_7 = \big[-15, -10, -5, 0, 5, 10, 15\big]$$

$$A_{15} = \big[-15, -12, -10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10, 12, 15\big]$$

In total, three different experiments where run;

- 7 actions and no ordinal architecture
- 7 actions with ordinal architecture
- 15 actions with ordinal architecture

Observation space representation and learning method are the same throughout all experiments. The methods used are the same methods used throughout this work. Figure 9 shows the training result of three different settings. The left-hand side of the figure shows mean reward, while the right hand side shows episode length. It can be seen that the mean reward roughly converges to the same value, with the 7 discrete action setting performing slightly better at maximum training time. In the initial phases of training the 15 actions setting is outperformed by the 7 actions settings, while the unordered setting gets significantly outperformed by the ordered methods. This is more evidently seen in the right hand side of figure 9. Episode length converges for both ordered methods relatively at the same pace, except for differences in the initial stages of the training. However, it can clearly be seen that the unordered method struggles with convergence compared to the ordered methods. Reason for quicker initial convergence with regard to the 15 and 7 action setting can be explained due to the fact that the 7 action is an easier method to solve; the to be solved for action space is simply smaller. However, no extra mean reward is attained due to this increase in resolution when moving from 7 to 15 discrete actions. This implies that the action resolution of 7 discrete actions does not limit the solution finding capabilities of the model in its current form.

Appendix B
Experiment Setup and Hyperparameters

This section contains parameters used during training, and the explanation of these parameters. It is well known that reinforcement learning problems can be sensitive to the correctly tuning of these parameters. Table III gives an overview of the hyper parameters, and their settings.

TABLE III
Overview of used parameters during training

Parameter	Description	Setting
n_ac	Number of aircraft generated at start	25
n_neighbours	Number of s_n states	5
LoS	Loss of separation distance	5
spawn_sep	Min separation of aircraft when generated	15
lr	Learning rate	0.0001
γ	Discount factor	0.99
λ	Lambda used for GAE	0.95
ϵ	Clip parameter used for PPO	0.3
entropy_coef	Coefficient for entropy regularization	0.01
train_batch_size	Experience collected each optimize step	200,000
num_sgd_iter	Optimizer passes each training batch	10
minibatch_size	size of optimizer minibatch	100,000

The value for the discount factor γ is determined from other work, as well as reasoning. 0.99 is an commonly used number for the discount factor. Also logically the air traffic control problem should look ahead as much as possible, so an low discount factor is unwanted. The GAE parameter λ is set based on other work, and was not empirically justified.

The entropy coefficient is set at 0.01, which was determined by an short comparison between an value of 0.01 and 0.001. As seen in figure 10 it can be seen that there is not much difference between gained reward based on this number. The faded lines are standard deviation plots based on the moving average of both datasets. It can be seen that there is a slight reduction of variance due to the higher entropy setting. The goal of entropy regularization is to improve exploration. Due to the minimal differences an value of 0.01 is used. Unfortunately due to time constrains, fully trained examples where unfeasible.

There are two commonly used optimization methods that are used to update the deep learning network weights; ADAM [14] and RMSprop [20]. The origin of these optimizers have their roots in deep learning tasks. However, most deep learning tasks do not optimize for a moving underyling distribution. RMSprop and ADAM workings on reinforcement learning is poorly understood, however empirically showed that performance is highly dependent on the problem [21]. Figure B shows an comparison between ADAM and RMSprop as optimizer, and the difference in performance of learning rate.

Hendreson et al [21] showed that most often an learning rate between 1×10^{-3} and 1×10^{-5} showed the best performance. This is also the case for many other reinforcement learning problems, where the learning rates are within this bandwidth. When the learning rate is to high, the problem cannot converge due to large steps resulting in unstable gradient updates. An

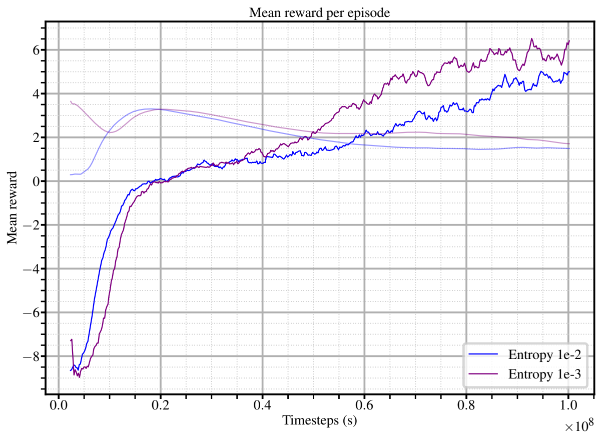


Fig. 10. Effect of entropy on learning performance. The fully coloured lines represent the mean reward per timestep, and the faded lines is the standard deviation of the data.

learning rate to low can lead to being stuck in local optima, which is also unwanted. First off, the performance of the 1×10^{-2} learning rate is weak, and there is no convergence. This option is quickly discarded. When looking at the performance of an learning rate of 1×10^{-3} and 1×10^{-4} it can be seen that an learning rate of 1×10^{-3} results in significantly more reward than an learning rate of 1×10^{-4}.

When comparing ADAM and RMSprop, it can be seen that RMSprop consistently performs weaker than ADAM. Especially when comparing them with the same learning rate, the difference is quite significant.

The choice was eventually made to keep the learning rate at 1×10^{-4}, and use ADAM as optimizer. Even though that an higher learning rate results in quicker convergence, there is also a risk that the model settles for sub-optimal behaviour due to quickly adjusting parameters to this found sub-optimal solution, with the risk of being stuck there. An example from the OpenAI team when working on the OpenAI Five

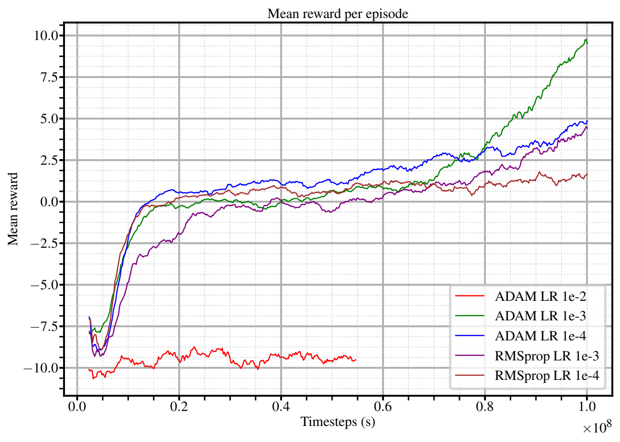


Fig. 11. Comparison between performance of two different optimizers; ADAM and RMSprop as well as the effect of the learning rate.

project, was that their philosophy was that an high enough batch size with a small learning rate would result in the best outcome. This luxury of infinite computing power is of course unfeasible, so an compromise was made.

Part III

Preliminary Thesis

Chapter 1

Introduction

In recent years reinforcement learning (RL) gained notable attention due to the defeat of Lee Sedol by the RL agent AlphaGo [1] in the game Go, and the performance of a full team on team match of the OpenAI Five agent [2] against OG, a professional E-sports team. The results of these matches gained widespread media attention, mainly due to the fact that these "AI's" showed some form of insight into the game. Inspired by these performances, the idea came to life of applying reinforcement learning to another form of game; the routing of aircraft in an airspace, i.e. Air Traffic Control.

Before diving into the exact workings of reinforcement learning, first an introduction is given regarding air traffic control, as well as setting up an example to grasp the general concept. This chapter also introduces the underlying principles of reinforcement learning, namely Markov Decision Processes.

In chapter two the basics of reinforcement learning are discussed, as well as some general information regarding reinforcement learning is given. Then in chapter three, the research questions are presented. These research questions are used to subsequently structure the chapters after that, using the research questions as guidelines for information. Chapter four will contain the bulk of the literature survey, as this concerns itself with finding a solution to the multi agent nature of the problem. Chapter five presents arguments for the practical side of the experiment. Chapter six will shortly discuss safety and efficiency, and how to measure these to serve as experiment variables. Chapter seven will concern itself with the practical implementation and setup of the experiment. Chapter eight will wrap-up all the aforementioned information, to serve as a stepping stone for the planning. The planning will be presented in chapter nine.

1-1 Air Traffic Control and Reinforcement Learning

Air traffic control is a ground service that provides guidance for airborne craft, to prevent collisions, organize the flow of air traffic and provide information and other support systems. The application of reinforcement learning on air traffic control will limit itself to the organization of air traffic, which includes routing, preventing collisions and optimizing the air flow. To test the performance of a reinforcement learning model, a platform is required that simulates air traffic control. Luckily this is available in the form of the BlueSky, an air traffic control simulation platform[3].

For now avoiding the details of reinforcement learning, a few basic requirements have to be set. Like most other machine learning techniques, reinforcement learning is based on a natural occurring process; learning from interaction with the environment. The process of an agent in a certain environmentally determined state, subsequently executing an action which in turn changes your state in the environment while receiving some form of reward. This interaction between agent, environment and reward can be seen in figure 1-1, using an air traffic control scenario as an example.

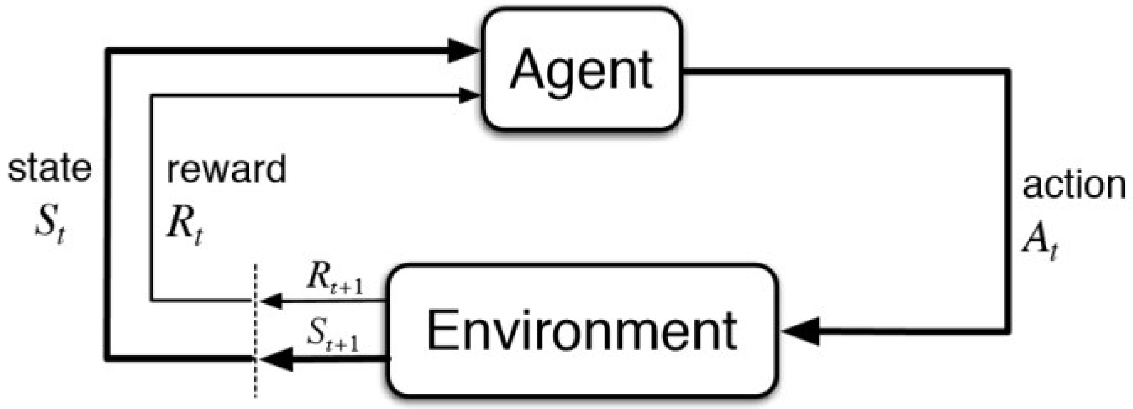


Figure 1-1: Depiction of interaction between environment and Agent

As seen in the figure 1-1, the fundamentals of reinforcement learning are: *Environment*, *Agent* and *Reward*. Figure 1-2 shows an overview of the BlueSky ATC simulator. In this figure the environment can be described as the airspace simulated in BlueSky, in this case the airspace of the Netherlands. The environment provides a certain depiction of the physical world. It includes locations, waypoints, airports and other physical properties of the environment. The agent in this environment is a single aircraft, depicted on the screen by the green triangle shape. The agent has a certain state determined by the environment. As an example, the agent's state is defined as a tuple containing its longitude, latitude and current heading. This could be expanded to current speed, current height, total available fuel etc. However for this example the three previously defined states are used.

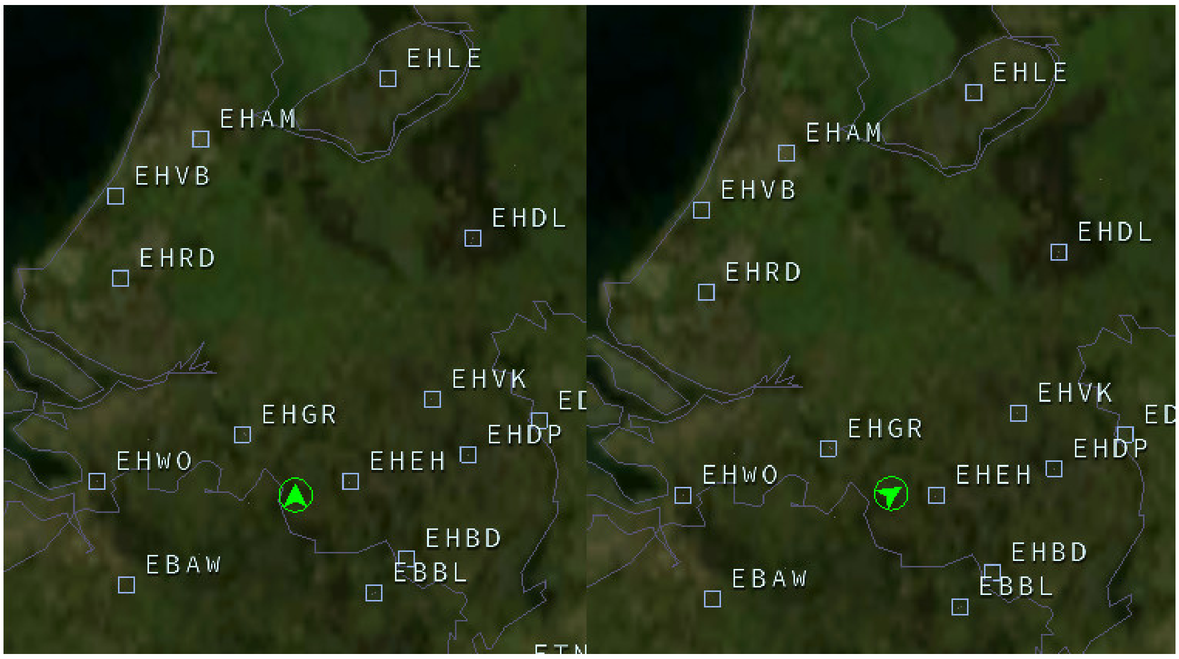


Figure 1-2: Overview of the environment provided by BlueSky

The left side of figure 1-2 is the initial situation starting at $t = 0$. The agent can now execute an action. The tuple that contains all the available actions can vary in size. One could define all available actions as small as a single rudder deflection. However, to keep things simple, actions are defined as selecting a new heading ranging from 0 to 359 degrees. The agent executes a heading change of 60 degrees. On the next timestep the aircraft would start to turn, impacting its own state. This can be seen in figure 1-2. The ability of an agent to impact its own state is an important aspect of the reinforcement learning process.

Reinforcement learning is based around a numerical reward signal. The recurrent interaction of actor, environment and received reward is the basis for reinforcement learning. In this case, the heading change affects the perceived reward. The reward can again be defined in numerous ways. However, the goal of the reward is to facilitate an indication of "good" and "bad" behaviour of the agent. In the example the reward signal is defined in proportion to the distance of the agent's location and its destination, EHAM. As can be seen in the right side of figure 1-2, the heading change turns the aircraft away from its destination, which intuitively results in a lower reward compared to flying straight to EHAM.

The learning part of reinforcement learning is the mapping of set of states to actions that maximize the reward signal. This maximization can be a maximization of the reward each timestep, but in general the total cumulative reward is maximized across an episode, i.e. each step contributes to a larger goal. If correctly defined, to achieve this larger goal, the cumulative reward will always be maximized when achieving this larger goal. Figure 1-2 assumes a timestep large enough to accommodate for an action, i.e. the heading change, to be executed in full.

Table 1-1 summarizes this single agent-environment step as seen in figure 1-2, displaying the change of state. Note the change in all three states through the executed action of a heading change to 60 deg.

Table 1-1: Change in state values due to an action, a heading change from 0 to 60 degrees.

	t_0	t_1
Latitude	51.41	51.27
Longitude	5.15	5.11
Heading	0	60

As can be seen, the state values are defined in a continuous fashion. To create a clearer example, discretization of the action space and state space is desired. This discretization will make the subsequent explanation of key concepts within reinforcement learning easier to visualize. Figure 1-3 shows this discretization. Each grid can be seen as a state the aircraft can be in. Heading of the plane is ignored, and the assumption is made that the aircraft can hop between those grids. As for the action space, the amount of actions available is equal to the amount of connected grids to the grid the plane currently is in. This results in an action space of 8 different movements.

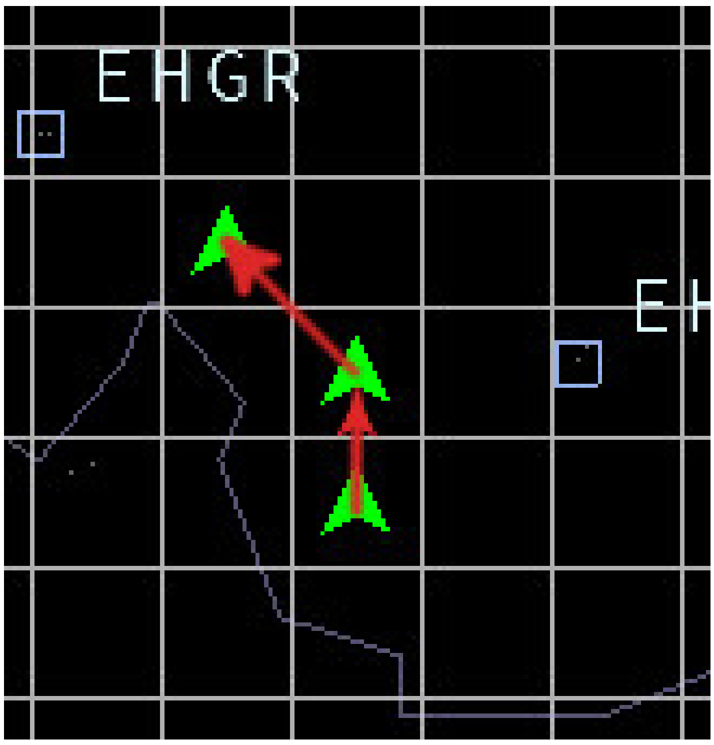


Figure 1-3: Discretization of the action space

To properly formalize the process of reinforcement learning *Markov Decision processes* are used. Markov Decision Processes provide the mathematical framework for reinforcement learning.

1-2 Markov Decision Process

A Markov Decision Process (MDP) is the classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent states, and through those affect future rewards. Each time step, the process is in a certain state, $s \in S$. The agent (the decision maker) chooses an action, $a \in A$, dependent on the state, which returns a reward, $R(s, s')$. This action then subsequently moves the agent into the next state, $s' \in S$. Additionally, for each action there is a probability that certain action leads to a certain state, formalized as the state transition probability, $P_a(s, s')$.
A general remark concerning the content of this section; most is based on the extensive reinforcement learning bible *Reinforcement Learning: An Introduction* by R. Sutton, and A. Barto [4].

An important fundamental assumption regarding MDPs is the *Markov Property.* A state is said to be a Markov state, i.e. has the Markov Property, when the current state contains all information needed to determine the state transition, and is not depended on previous states. In short, "The future is independent of the past given the present".
In the current running example, this is true. The state currently consists of longitude, latitude and heading. These properties are absolute. When executing an action the subsequent latitude, longitude and heading changes can all be determined based on the current state information. To summarize, an MDP is defined as a size 4-tuple (S, A, P_a, R), where S represents the finite set of different states, A the different available set of actions, P_a the state transition matrix and R the reward function.

As stated before, the agent's goal is to maximize the cumulative reward it receives over the full trajectory. To formalize this, the *return* is defined as the sequence of reward received over the trajectory, as seen in equation 1-1.

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{1-1}$$

G_t is the return. γ is the discount factor, which balances the importance of future reward versus immediate reward. T is the timestep at the end of the trajectory. The return encompasses the total future discounted reward from the current timestep forward. More specifically, forward means that it is an accumulation of reward looking in the future, not look backwards at already received reward.

Another widely used group of functions for all reinforcement learning algorithms are called the *value functions.* These value functions are needed to estimate whether or whether not the current state is a "good" or "bad" state to be in. This "good or "bad" is determined by the *expected return*, i.e. the <u>expected</u> cumulative reward. However, rewards depend on the actions the agent takes in each state. This behaviour of the agent is called a *policy.* A policy is a mapping between the state and the probability of selecting each possible action, and is defined as $\pi(a|s)$. To be precise, the value function under a certain policy is defined in equation 1-2.

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s] \tag{1-2}$$

$v_\pi(s)$ is the value function under policy π, G_t is the return and s the current state. Formally, $v_\pi(s)$ is called the *state-value function* for policy π. Concurrent with the state-value function, the *action-value function* is defined. The difference between the value functions is that the action-value relies on both the state and action, as can be seen in equation 1-3.

$$q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a] \tag{1-3}$$

The action-value function $q_\pi(s, a)$ is the expected return starting from state s, taking action a and then subsequently following policy $\pi(a|s)$. Take careful note of the *then subsequently following policy* $\pi(a|s)$ part.
Figure 1-4 gives a visual interpretation of how a fully determined state-value function would look like under a certain policy would look like. As can be seen, each grid contains a colour. Red means a low value, while green indicates a high value for that state. What is also shown in the figure are my extraordinary garbage figure creation skills.

More complex reward signals are difficult to determine visually, but eventually the value function has to converge to a shape that indicates being in the vicinity of EHAM is preferred to being further away. How the actual numerical values of the value- and action-value function are determined is shown in the next section.

To end this section, an important recursive relation is highlighted. The return, which is an summation of the expected reward, can also be defined as a recursive equation with itself. This can be seen in equation 1-4.

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...) \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{1-4}$$

This relation is used in the following section to find a recursive relation for the value-functions.

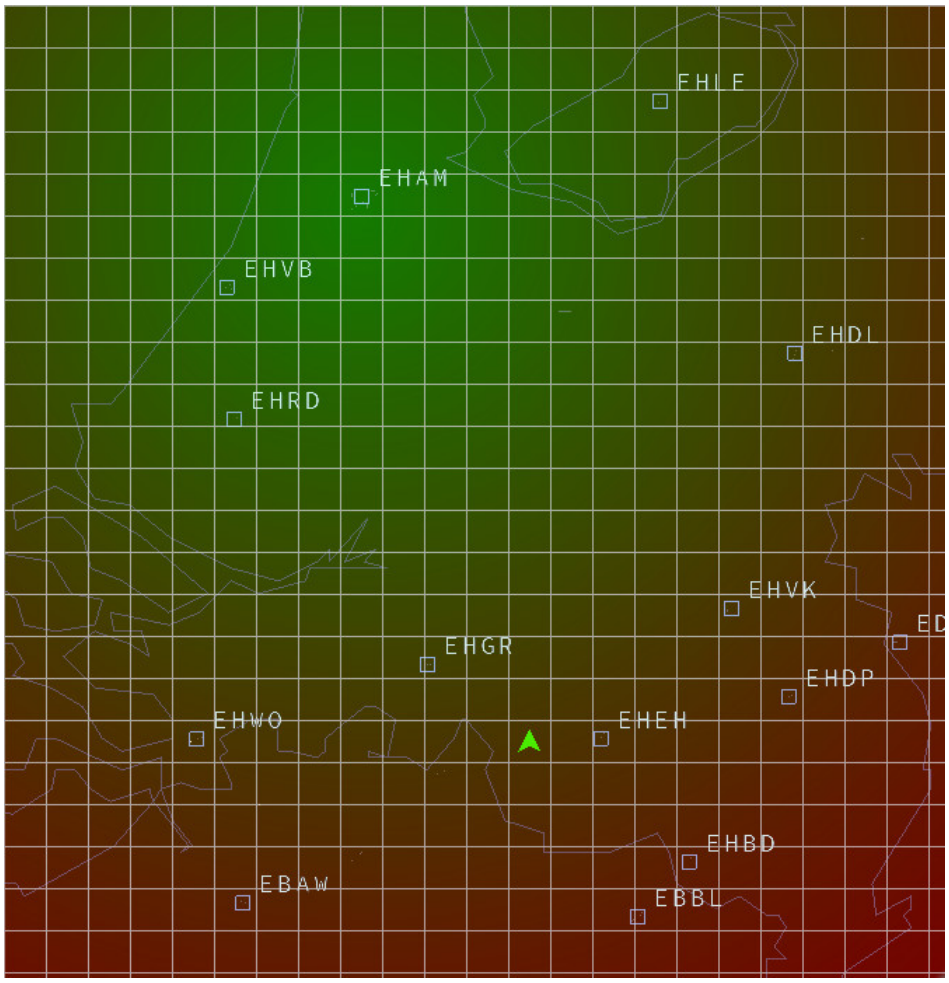


Figure 1-4: Graphical depiction of a state-value function. Green indicates a higher value, while red indicates a lower value.

1-3 Finding a solution to the MDP

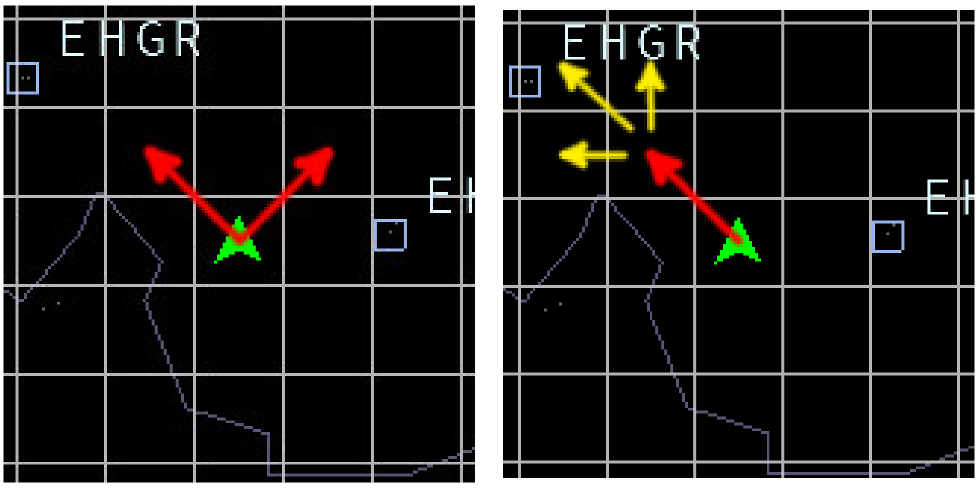
The previous introduction of value-functions and policies expand the arsenal with which information is given about how the agent is doing, as well as what the agent's behaviour is. The question remains how to determine the behaviour which results in the maximization of the cumulative reward. In short, a policy $\pi(a|s)$ has to be found that maximizes the cumulative reward, $v_\pi(s)$. However, the previous definitions of the value functions are not fit for any numerical solution method, so to be able to find this optimal policy, a recursive relation is needed for the state-value and action-value functions. The following recursive relations use the property of the recursiveness of the return as baseline, as seen in equation 1-4.

Consider figure 1-5a; this figure shows two of the eight available actions to take in this particular state. To determine the current value of this state under policy $\pi(a|s)$, the value function $v_\pi(s)$ is utilized as seen in equation 1-5. The total value of that state is the summation of all the action-value functions weighted by the probability of selecting the corresponding action.

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s,a) \tag{1-5}$$

However, just as $v_\pi(s)$ is dependent on $q_\pi(s,a)$, $q_\pi(s,a)$ is dependent on $v_\pi(s)$ When a certain action is taken, the action-value function gives a value to this action. However, there is a certain probability on the outcome of the action. As an unrealistic example, wind might blow the plane in another direction than intended. The chance of taking a certain action and ending up in a certain state is encapsulated in the state transition probability, $P^a_{ss'}$. Figure 1-5b shows an example. The red arrow is the action chosen, while the yellow arrows indicate the other locations the aircraft can end up in due to wind. Taking this action and transitioning to this state also results in a certain reward, specified by the environment. This results in the following equation for the state-action value function

$$q_\pi(s,a) = R^a_s + \gamma \sum_{s' \in S} P^a_{ss'} v_\pi(s') \tag{1-6}$$



(a) Depiction of actions (red arrows) in a discretized environment

(b) Effect of the transition probability on final end state.

Figure 1-5: Left side: Discretized actions. Right side: Effect of transitional probability

Equation 1-5 and 1-6 can be combined to create a recursive relation between initial state and the transition state. This can be done for both $v_\pi(s)$ and $q_\pi(s,a)$. These equations are shown in equation 1-7 and 1-8 respectively.

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \Big(R^a_s + \gamma \sum_{s' \in S} P^a_{ss'} v_\pi(s') \Big) \tag{1-7}$$

$$q_\pi(s,a) = R^a_s + \gamma \sum_{s' \in S} P^a_{ss'} \sum_{a' \in A} \pi(a'|s') q_\pi(s',a') \tag{1-8}$$

These equations are called the *Bellman Equations*. They express the relationship between the value of the current state and its successor state. As a final form of visualization, figure 1-6 shows a backup diagram of the above defined process for the value function. A backup diagram can be viewed as a form of search tree, in which each leg is a certain action, and every hollow circle is a new subsequent state.

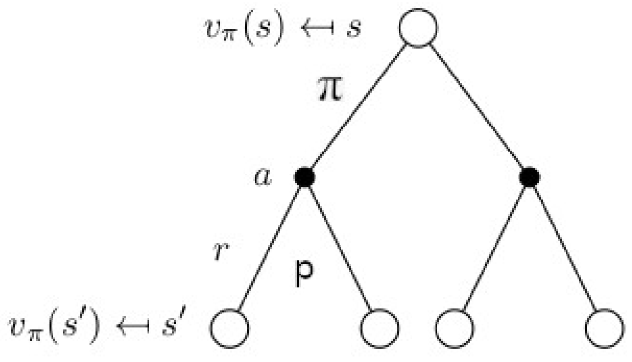


Figure 1-6: Backup diagram of the value function [4]

Now that the recursive relation is established it is time to search for a method to find the optimal policy; a policy that maximizes the accumulated reward over the long run. The policy defines the behaviour of the agent, and is of significant influence on the value of both value functions. A policy π is better than or equal to a policy $\pi^{'}$ if the expected return is greater than or equal to that of $\pi^{'}$ in all states. Luckily, the value function facilitates this. In short;

$$\pi \geq \pi^{'} \text{ if and only if } v_\pi(s) \geq v_{\pi^{'}}(s) \text{ for all } s \in S$$

There is always a policy that adheres to this statement, which by definition is an optimal policy. Note the is *an* optimal policy. Multiple optimal policies can exist, however they all share the same value function. From this, the optimal state-value function and state action value function can be defined as $v_*(s) = \max_\pi v_\pi(s)$ and $q_*(s,a) = \max_\pi q_\pi$. Finding these, especially q_*, facilitates the optimal policy to be found. To summarize:

- There exists an optimal policy π_* that is better than or equal to all other policies, $\pi_* \geq \pi, \forall \pi$
- All optimal policies achieve the optimal value function, $v_{\pi_*}(s) = v_*(s)$
- All optimal policies achieve the optimal action-value function, $q_{\pi_*}(s,a) = q_*(s,a)$

Combining the above with the previously defined Bellman equations seen in equation 1-7 and equation 1-8, the Bellman equations can be rewritten for the case of an optimal policy. These result in the *Bellman Optimality Equations*, seen in equation 1-9 and equation 1-10. Note the absence of $\pi(a|s)$. This is due to the fact that under an optimal policy, the value functions depict the maximum expected return for that state. This means that simply taking the action that results in the maximum expected return *is* an optimal policy. Always selecting an action that results in the maximization of the value is called a *greedy* policy. Once either $v_*(s)$ or $q_*(s,a)$ are found, the solution to the MDP is then relatively simple; always acting greedily with respect to the value-functions is the optimal policy.

$$v_\star(s) = \max_a \Big(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\star(s') \Big) \tag{1-9}$$

$$q_\star(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_\star(s',a') \tag{1-10}$$

A fundamentally important method of computing the optimal policy is a collection of algorithms called *Dynamic Programming (DP)*. DP algorithms are however of limited practicality due to the requirement of knowing the environment's perfect model as well as their computational expense. Nevertheless, all other methods that attempt to find the optimal policy can be seen as an approximation attempt at DP, but with less computational effort and without perfect knowledge of the environmental model. The idea behind DP is that the value functions are used to structure the search for optimal policies.

A first step in DP is to be able to compute the state-value function v_π for a given policy $\pi(a|s)$. Recall equation 1-7, which gives the recursive relation for state-value function calculation. Determining the state-value function for a given policy is called *policy evaluation.* Converting equation 1-7 into a iterative update rule results in equation 1-11, given that the policy is stationary.

$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s) \Big(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \Big) \tag{1-11}$$

Because this is an iterative process, initial conditions have to be set. Depending on the problem description these conditions can vary, but often are $v_0(s) = 0$ for all $s \in S$. In general, it can be shown that $v_k \to v_\pi$ for $k \to \infty$.

Policy evaluation enables a method of evaluating the "quality" of a certain policy through iterative solving of the state-value function. This however also set the stage for finding better policies. Suppose the state-value function v_π is fully determined under a arbitrary policy π. For each state s an assessment can be made to see if there is an action a that is $a \neq \pi(s)$ which results in better behaviour, i.e. a higher value. Luckily, the action-value function, $q_\pi(s,a)$, does just that. So if there is an action that can be selected which results in more value and thereafter following policy π, it can be said that selecting this action in this state will always result in more value for the whole trajectory. In short, $q_\pi(s, \pi^{'}(s)) \geq v_\pi(s)$, where $\pi^{'}(s)$ is the new policy that results in better behaviour in the current state. This subsequently also implies that $\pi^{'}(s) \geq \pi(s)$, hence a better policy is found. This is called *policy improvement.* So for all states, selecting the action that maximizes the value through $q_\pi(s,a)$ can be written as seen in equation 1-12. Using this form of maximization for each action is called a greedy optimization policy.

$$\pi^{'}(s) = \arg\max_a \; q_\pi(s,a) \tag{1-12}$$

Now that both a method for policy evaluation and policy improvement are found, these can be used in an alternating sequence to first evaluate the policy, then improving it, then evaluate and so on. This is called *policy iteration*, as seen in equation 1-13. The E stands for evaluation step and I for the improvement step.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \ldots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_{\pi_*} \tag{1-13}$$

However, generally it is not necessary for policy evaluation to go to full convergence in each step. To expand, when evaluating a certain policy the value function often takes a multiple iterations for convergence to the true value function. For most application the absolute converged value is not necessary, only the ordering and general tendency of these values. Reducing the amount of iterations will decrease computational requirements. A special case of reducing the policy evaluation iterations is called *value iteration*. Value iteration only does a single policy evaluation iteration. Then, this can subsequently be combined with the policy improvement step and reduce the multiple policy evaluations and subsequent policy improvement to a single update rule. The equation for value iteration is shown in equation 1-14.

$$v_{k+1}(s) = \max_a \sum_{s'} P^a_{s,s'}[R_a(s,s') + \gamma v_k(s')] \tag{1-14}$$

Note the relation of value iteration to the Bellman Optimality equation. Value iteration takes the greedy policy improvement step and merges this with the update of the value function under the new policy.

The previously mentioned iterative scheme for interaction between evaluation and improvement is called *generalized policy iteration (GPI)*. The idea of GPI is seen often in reinforcement learning methods; the interaction of improving the policy depending on the current value function, and then updating the value function under this new policy. Figure 1-7 summarizes this interaction. When both the updates to the policy and the value function stabilize, i.e. the updates each step get below a certain threshold, the Bellman optimally equations are correctly approximated and the optimal policies and value function are found. As a final note on DP, DP actually provides a consistent way of finding the optimal policies and value functions. However, bigger state representations require a lot of complete sweeps resulting in strain on computational requirements. Also, and even more mentionable, they require the full model of the environment. This is where model free methods are introduced. Model free methods rely on probing the environment through exploration, and try to estimate the value functions and policy based on feedback from the environment alone. Model free methods that rely on experience are called *Reinforcement Learning* methods.

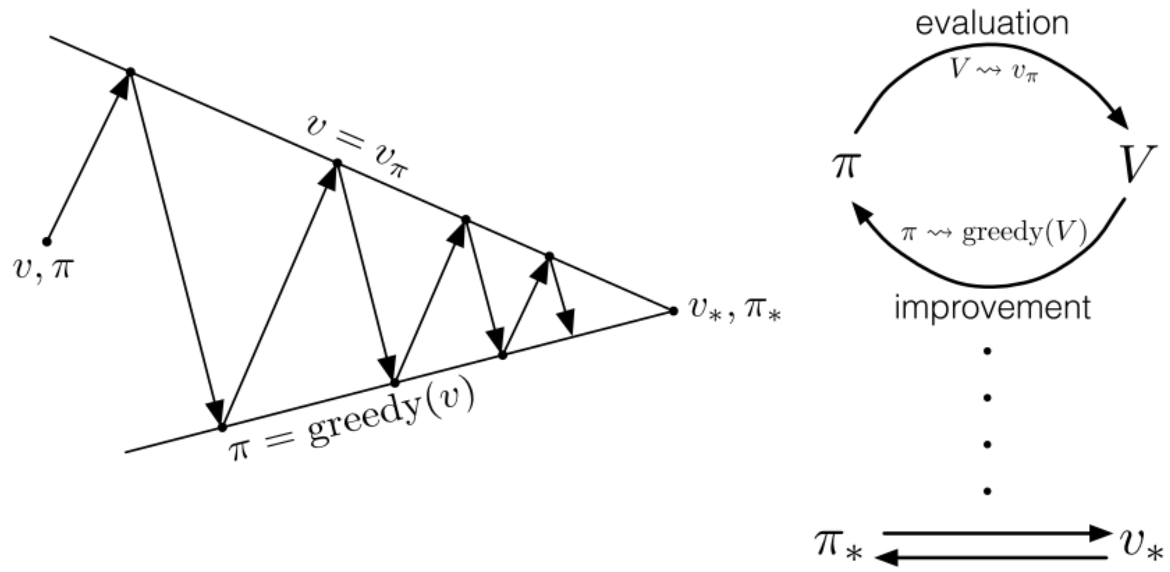


Figure 1-7: GPI trajectory for a state-value function

Chapter 2

Model Free Methods

Model free reinforcement learning methods lack the knowledge about the transition probability of the environment, and learn solely from interaction with the environment. A distinction can be made between value function based solutions and policy gradient methods. Value function based solutions are a model free extension to the dynamic programming methods discussed in the previous section. Policy gradient methods are a more recent development, where the policy space is parameterized and estimated directly. First, value based model free methods are discussed. After that policy gradient methods are expanded upon and a few widely used concepts are introduced.

2-1 Value Based Model Free methods

First a few traditional and widely used methods are explained, which originate from dynamic programming to tackle model free problems. Then, at the end of the section deep learning is combined with value based methods, providing a foundation for many state-of-the-art reinforcement learning methods.

2-1-1 Monte Carlo methods

Monte Carlo (MC) methods are a school of learning methods that depend on the sampled return, i.e. gained experience of a complete episode. This sets the requirement that each problem needs to have a finite horizon. In contrast to DP methods where the value functions are directly calculated for all states, MC methods learn the value-functions from sample returns gained on a trajectory through the environment. In this section, the same structure is used as seen in the DP section. First Monte Carlo Prediction, also often named Monte Carlo value estimation, is shortly explained. Then a step is made towards Monte Carlo for action-values functions and finally Monte Carlo control is explained.

Like with DP, a method of evaluating a certain policy has to be put in place. The main difference between DP and MC is the fact that not all information is available, and consequently the experience, or rewards, have to directly fill up the value function estimation. So how to estimate $V(s) \approx v_\pi(s)$, under an certain policy π and an arbitrarily initialized value function $V(s)$. This is done by keeping track of all the gained reward-state pairs during a single episode, i.e. trajectory. Then for each visited state s during this trajectory, calculate the return $G(s)$. When the returns are calculated the value function $V(s)$ is simply estimated by taking the average of the returns for each state visited over all episodes; $V(s) = average(G(s))$. An important note is that for every trajectory done, a single state can be visited multiple times. The choice can be than made of determining the value function depending on *every-visit* of the state or only taking into account the *first-visit* of each state. In general, first-visit is preferred. The previously stated is known as *First-visit MC Prediction*, where prediction is a term for policy evaluation which provides a more natural and describing term to the method. As calculating the return for a state requires the estimated return of the states before it, updating the value function can only happen when the full trajectory is complete. When complete, the returns are then estimated by working backwards from the end of the trajectory towards the beginning.

However, estimating the state-value function is of lesser interests compared to the estimation of the action-value function, $Q(s, a) \approx q_\pi(s, a)$. The reason is, is that for any control problem it is more convenient to know the value a certain action contributes to the return. The downside of estimating the action-value function is that the amount of to-be-estimated parameters increases significantly depending on the amount of available actions. Another hurdle to overcome is the fact that when the policy that is being evaluated is deterministic, i.e. for each state-action pair a certain action is always chosen, for each state only one action-value is being estimated. This ties in closely with the *exploration* problem, which will be discussed later. So to apply First-visit MC prediction naively to action-value functions, a policy has to be used that ensures a non-zero probability for all actions in each state.

Now, again on par with the sequence of steps seen in the DP section, a method of improving the policy is implemented. Recall general policy iteration (GPI) in which the evaluation and iteration steps are executed in a alternating fashion. The GPI for the action-value function is shown in equation 2-1

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \ldots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_{\pi_*} \tag{2-1}$$

The evaluation step is discussed in the previous, however the iteration step differs from that of DP. This is due to the fact that in DP, state-value functions where used, which require the known model. However, when using action-value functions the policy can easily be reduced; for every state the action is taken that will maximize the reward, i.e. a greedy policy. However, instead of at each evaluation step waiting for full convergence, the amount of iterations can be truncated. An earlier version of this was seen in DP, which was value iteration. Value iteration only took a single evaluation step and then did policy improvement. The GPI for MC would follow this general behaviour; After each episode (trajectory) the returns are used for policy evaluation for each visited state. Then a policy improvement is done for each visited state.

In the above the problem of exploration is not managed. Exploration on its own is a recurring topic for reinforcement learning techniques, some more light is shed on exploration in section 2-3. Control methods face a dilemma; they seek to learn action values conditional on subsequent optimal behaviour, but to determine what optimal behaviour is, sub-optimal behaviour is required to explore all actions. In general, a distinction can be made between two methods of evaluation, *on-policy* and *off-policy*. On-policy methods evaluate or improve the policy that is also used to make decisions. This in contrast to off-policy, where the policy that is used to evaluate or improve is different from the policy that is used to generate data (trajectory). An on-policy method for improved exploration is called *ϵ-greedy*. In contrast to a normal greedy policy, ϵ-greedy behaves greedy with a probability of $1 - \epsilon$, while choosing an action at random with a probability of ϵ. This method forces actions that are considered sub-optimal which in turn result in exploratory behaviour.

Consider the idea of having two separate policies. One policy which converges to the optimal policy, while another policy is made that ensures exploratory behaviour. This is where off-policy methods come in. The policy that is being learned is called the *target policy*, while the exploratory policy is called the *behaviour policy*. This division is the underlying principle for off-policy methods. The data that is learned from is off the target policy. Off-policy is a powerful tool, however some additional care is needed. The reason for this extra care is the fact that the target policy is being estimated, while the data gathered is not actually from this policy, inducing greater variance and slower convergence.

The target policy is denoted by π, while the behaviour policy is b. Note that for off-policy methods, both the target policy and behaviour policy have to ensure that both policies result in visitation of the same set of state-action values. This is called *coverage*. Before moving towards off-policy MC control, first MC evaluation is considered. Due to the difference in the target and behaviour policy compensations have to be made to correct this difference when learning. This is done by *importance sampling*. Importance sampling tries to determine the expected values of a distribution by using samples of another distribution; in this case the target policy and behaviour policy. Importance sampling is used to weight the returns according to relative probability of their trajectories occurring under the target and behaviour policies. This weighting is called the importance-sampling ratio, and can be seen in equation 2-2. $p_{t:T-1}$ is the importance-sampling ratio from time-step t to the end of the current trajectory $T - 1$.

$$p_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k}{b(A_k|S_k)} \tag{2-2}$$

An extension to importance sampling is by using *weighted importance sampling*. In contrast to ordinary importance sampling this uses an weighted average instead of an absolute average. Using the weighted importance sampling ratio, to estimate $V(s)$, the returns are scaled by the weighted importance sampling ratio. This can be seen in equation 2-3. Note that the sum denotes every first-visited occurrence over all episodes.

$$V(s) = \frac{\sum p_{t:T-1} G_t}{\sum p_{t:T-1}} \tag{2-3}$$

Weighed importance sampling is in practice almost exclusively used in contrast to ordinary importance sampling.

However, as most methods an incremental update rule for the weighed importance sampling ratio, which is shown in equation 2-4. W_k is the importance sampling ration and G_k the return.

$$V_n = \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1}} \tag{2-4}$$

For each update the value of the previously applied weights has to be known, W_k. This is done by keeping track of previously applied weights in a new variable, C_n. The update rule for V_{n+1} and C_{n+1} are shows in equation 2-5 and equation 2-6 respectively.

$$V_{n+1} = V_n + \frac{W_n}{C_n}[G_n - V_n] \tag{2-5}$$

$$C_{n+1} = C_n + W_{n+1} \tag{2-6}$$

Now that all elements are in place, *Off-policy Monte Carlo Control* is introduced, combining all of the aforementioned. As a clear overview of the workings of off-policy MC control, the pseudo code is given. This can be seen in algorithm 1.

Algorithm 1: Off-policy MC control

```
Initialize, for all $s \in S, a \in A(s)$:
    $Q(s,a) \leftarrow$ arbitrary
    $C(s,a) \leftarrow 0$
    $\pi(s) \leftarrow \arg\max_a Q(S_t, a)$
Repeat forever:
    $b \leftarrow$ any policy with coverage of $\pi$
    Generate an episode using $b$:
        $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T, S_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    For $t = T-1, T-2, ...,$ down to 0:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
        $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$
        If $A_t \neq \pi(S_t)$ then exit For loop
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$
```

2-1-2 Temporal Difference methods

Temporal Difference (TD) methods are another school of methods that are highly correlated with DP and MC methods. Compared to MC, it also uses experience and does not need to know the system dynamics. However TD does not wait for a full episode, but rather learns based on other learned estimates. First, as done before, a method for policy evaluation is given. Subsequently, a method for control is given, based on the GPI.

The policy evaluation step in TD is similar to that of MC, however with a slight but impactful difference. MC waits till the end of the episode to exactly calculate the return, and updates its value functions accordingly. TD does not wait until the end of the episode, but updates its estimate of the value function after each time step. Equation 2-7 first shows the MC Prediction step, while the bottom equation shows the TD prediction step. This is done for ease of comparison between the two methods. Note that this is a slightly different formulated MC prediction step as seen before, but still under the same mechanics. α is a constant step-size.

$$\begin{aligned} V(S_t) &\leftarrow V(S_t) + \alpha[G_t - V(S_t)] \\ V(S_t) &\leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \end{aligned} \tag{2-7}$$

As seen in these equations, the main difference is that for MC the estimated return is used as an update target, while for TD the reward is used in conjunction with the discounted estimated value of the next state as a target. Updates based on existing estimates are called a *bootstrapping* method. The aforementioned TD equation is called *one-step TD*, because it only bootstraps one step ahead, $V(S_{t+1})$. Note the definition of $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$. This is basically the error between the old value, $V(S_t)$ and the new value estimate $R_{t+1} + \gamma V(S_{t+1})$. This is called the *TD error* δ and is a widely used term throughout reinforcement learning.

On-policy TD control follows as DP and MC did the pattern of GPI, only now using the TD method of evaluation. The same as with MC methods, instead of evaluating the state-value function the action-value function is used. The update rule is shown in equation 2-8.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1}\gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2-8}$$

For the update to be done a few elements are required, and achieved in the following manner; $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$. This sequence gives name to this on-policy TD method, namely $SARSA$.

Naturally, an off policy method also exists for TD Control. This method is one of the most widely used value based learning algorithms, namely *Q-learning*. Q-learning, as SARSA does, bootstraps. However, its off policy behaviour is dictated by the use of the $\max_a$ operator. Equation 2-9 shows the Q-learning update rule.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2-9}$$

The nature of off-policy for Q-learning is how action A_t is selected. So the behavioural policy b for Q-learning is often $\epsilon -$ greedy for selecting A_t, while the target policy is greedy, i.e. the $\max_a$ operator. Up until now, it is assumed that the Q-values are stored in a tabular fashion. Q-learning is at the basis of most modern value-based *deep learning* extensions. The deep learning extensions come in the form of Q-value approximation using deep learning as function approximation. One of the most noteworthy combination of Q-learning with deep learning techniques, is *Playing Atari with Deep Reinforcement Learning* by Mnih et al.[5]. This paper will now be discussed to provide an introduction to value-based deep reinforcement Learning. An more in-depth explanation of how various deep Learning methods work can be found in section 2-4.

2-1-3 Deep Q-learning

Playing Atari with Deep Reinforcement Learning gained widespread attention due to the generalization performance of the algorithm, being able to play Atari games solely based on visual input on par with human performance. However, naively applying deep learning methods for Q-value estimation results in poor performance and a few modifications have to be put in place to overcome these. Firstly, reinforcement learning depends on a reward signal for learning. This reward signal can often be sparse, noisy and delayed. The delay between certain actions and subsequent reward is problematic especially when most deep Learning methods rely on directly gained feedback on their estimation performance, for example in many supervised learning applications. Another hurdle is the fact that the data feedback from reinforcement learning is often highly correlated. For example, the state of an aircraft after a certain action is closely related to the previous state. Lastly, due to the change in policy the data distribution changes. This non-stationarity of the data distribution is unwanted for deep learning techniques, as they assume a fixed underlying data distribution.
The update rule is working on the same principles as equation 2-9, however with slightly different notation.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a))] \tag{2-10}$$

To use deep learning as a function approximator the action-value function, $Q(s,a)$, has to be parameterized. This results in $Q(s,a;\theta) \approx Q(s,a)$ where $Q(s,a;\theta)$ is parameterized with weights θ. To update the weights a differentiable loss function is required on which gradient decent or other optimization method can be applied. The loss function is defined in equation 2-11.

$$L_i(\theta_i) = (y_i - Q(s,a;\theta_i))^2 \tag{2-11}$$

L_i is the loss function for iteration i and y_i is the target. In this case, the target is the left side of the TD error, $y_i = r + \gamma \max_{a'} Q(s',a';\theta_{i-1})$. Note that the target is being kept fixed with respect to its older parameters. This is done to stabilize the network updates, while alleviating the non-stationarity and corrolation issues. Finally, the loss function has to be differentiated with respect to the network weights θ_i. This differentiation is a underlying requirement for any deep learning optimization technique. This results in equation 2-12.

$$\nabla_{\theta_i} L_i(\theta_i) = \big(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\big)\nabla_{\theta_i} Q(s,a;\theta_i) \tag{2-12}$$

Now that the mechanics are in place, there is still no "hard" solution for the aforementioned problems when applying deep learning to Q-learning methods. Mnih et al. utilized a technique called *experience replay*. Experience replay stores for each iteration the gained experience, action and transition states; $e_t = (s_t, a_t, r_t, s_{t+1}$. This information is subsequently saved in a database called the *replay memory.* Instead of learning directly from each executed action at runtime, the replay memory is sampled and transitions from the memory are used for learning. This method has multiple advantages. First, the data efficiency is increased greatly, because transitions are used multiple times for weight updates. Secondly, randomly sampling from the replay memory breaks the correlation normally found when learning from consecutive samples, which aids in the non-stationarity and correlated data problems discussed in the beginning of this section.

Mnih et al. used a *Convolutional Neural Network (CNN)* as a function approximator. As input to the CNN the raw pixel values where used that normally would be shown on the screen, and as output the CNN would return the estimated Q-values for each state-action pair. Using a CNN allowed for generalization across multiple games, due to the fact that instead of hand crafted state vectors, the screen is given as an input and interpreted by the CNN much like a human would do. The *Deep Q-network (DQN)* returned state-of-the-art results for 6 out of the 7 games the DQN was trained over. Figure 2-1 shows an ensemble of the games the DQN learned.

Figure 2-1: An example of the games played by the DQN [5]

2-2 Policy gradient methods

Up until now all methods discussed resolve around the estimation of a value function, that in term determine the policy by taking an action in the direction of the highest return. In this section the policy is directly optimized. In short, $\pi(a|s, \theta) = \Pr\{A_t = a|S_t = s, \theta_t = \theta\}$, where θ are the policy weights. To allow for the policy to be optimized, equation 2-13 is given. α is the step-size and $J(\theta_t)$ is an arbitrary performance measure. A requirement for gradient optimization methods is that the underlying function approximation is differentiable.

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{2-13}$$

This arbitrary performance measure often takes shape in the form of a value function. The use of a value function in conjunction with an policy estimation method results in a school of methods called *actor-critic* methods. The value function is considered the critic, which serves an opinion about how "good" or "bad" the state is to be in. The actor is the policy, which determines what action to take.

Parameterization of the policy brings some advantages compared to the greedy policies commonly used with value based methods. First off, the policy space can be approximated using stochastic function approximation. Depending on the problem, stochastic function approximation enables for the policy to converge to a stochastic optimal policy. An example of a situation that would require an stochastic policy would be that of the game; rock, paper scissors. The optimal policy would be an equal chance of selecting each action. Another advantage is that estimating the policy itself in some cases is easier than finding the estimating the complete value function for the complete problem. As previously shown with value iteration (section 2-1), full convergence of the value function is not required to determine the optimal policy. A final advantage is that when using policy optimization, the action probabilities change smoothly during updates. When using a value based approach, when a certain action-value would surpass that of another action-value, under a greedy policy this other action would be chosen instantly. This can result in erratic behaviour

To allow weight updates for the function approximation, the performance measure $J(\theta)$ has to be defined. An example is defining the performance measure as the value of the starting state s_0, where $v_{\pi_\theta}(s_0)$ is the true value function for policy π_θ. This results in the following definition:

$$J(\theta) \doteq v_{\pi_\theta}(s_0) \tag{2-14}$$

The performance measure varies between different problems, and can be constructed as seen fit. The effect of the policy on this performance measure is needed, and more specifically the gradient of the performance measure with respect to the parameters. Based on the *Policy Gradient Theorem*, which establishes a proportional relationship between the gradient of the performance measure and the gradient of the policy, an definition can be given. This is shown in equation 2-15. The derivation of this relationship is quite extensive, and can be found in [4].

$$\nabla J(\theta) = \mathbb{E}_\pi \left[G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}\right] \tag{2-15}$$

S_t and A_t are the sampled actions and visited states and G_t is the trajectory return. Combining equation 2-13 and equation 2-15 results in a classic policy gradient method, called *REINFORCE*, as shown in equation 2-16.

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \\ &= \theta_t + \alpha G_t \nabla_\theta \ln \pi(A_t|S_t,\theta) \end{aligned} \tag{2-16}$$

Equation 2-16 has an intuitive appeal. The parameters are moved in the direction of space that increases or decreases its probability of repeating the action, weighted by the received return. This implies that high returns for certain actions will be updated more heavily. Meanwhile, the division by the actual probability of selecting the action will result in balancing out the updates depending on the probability of selection. Otherwise action that are already visited frequently would be updated unfairly due to the higher chance of taking that action.

REINFORCE uses the full return at each time step, putting it in the same school as Monte Carlo methods. Updates are only done at the end of each episode when the full trajectory and return is known.

An extension can be made to the REINFORCE algorithm that introduces a *baseline* with which the action value can be compared to. This sets the stage for the introduction of *actor-critic* methods. When using action values for the parameter updates the interest is not the absolute value of the action values but more the difference between them; which action is relatively better compared to the other available actions. When using the absolute values the variance between states can be high, and subsequent update steps. The baseline is used to normalize the updates, reducing the variance in each update step. The modified algorithm, *REINFORCE with baseline* is shown in equation 2-17, where $b(S_t)$ is a arbitrary baseline value depending on the current state.

$$\theta_{t+1} = \theta_t + \alpha(G_t - b(S_t))\frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \tag{2-17}$$

The baseline value can be any arbitrary value. However, baseline should have as property that when in a state where the actions have high value, the resulting baseline should be high in value as well. When all actions in a certain state are of low value, the baseline should be low of low value. An solution for this would be the use of an estimated state value, $\hat{v}(S_t, w)$. w are the weights for the state value estimation. Due to the REINFORCE algorithm being a Monte Carlo method, the value function is estimated using a Monte Carlo method. Using an approximated state value function as baseline is closely related to actor-critic methods, which will be discussed next.

2-2-1 Actor-Critic

The REINFORCE with baseline is not considered an actor-critic method, however still gives the general intent of the critic in actor-critic methods. The reason that it is not considered an critic is that the value function is not used as an estimate, but as a baseline with respect to the return. Also, an episodic method of estimation such as MC converge slowly and are not suitable for online estimation. A natural step is to replace the return with an value function estimate. The value function is updated using a bootstrapping methods, as seen previously done with TD methods. Recall that $\mathbb{E}(G_t) = V(S_t) = \hat{v}(S_t, w)$. This results in a one step actor-critic method showed in equation 2-18, where δ_t is the TD error.

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha\Big(R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)\Big)\frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \\ &= \theta_t + \alpha\delta_t \ln \pi(A_t|S_t,\theta)\end{aligned} \tag{2-18}$$

This results in a online process, that can be updated every step. For clarity, the critic in this case is the estimation of the value function, $R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)$. The actor is the policy estimation part, $\frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$. This double estimation is the basis for all actor-critic methods.

Often the critic and actor are both estimated by using deep learning as function approximation techniques. The combination of deep learning and actor-critic methods will be expanded on in section 4-1-2. Now that the basics of policy gradient methods are presented, there are a few more challenges to overcome. When using deep learning in combination with policy gradient methods, additional to the policy optimizations steps have to be introduced to provide stability for the gradient updates. This will be covered in the next section, as well as expand on the application of deep learning with policy gradient methods.

2-2-2 Policy optimization

In this section two different policy optimization methods are expanded upon, namely *Trust Region Policy Optimization (TRPO)* [6] and *Proximal Policy Optimization (PPO)* [7] by Schulman et al. The aforementioned papers are discussed as they are used often in subsequent literature, as well required to extend policy gradient methods to the deep learning domain. First TRPO is expanded upon and lastly PPO.

The goal of TRPO is to provide a method that guarantees policy improvement for nonlinear policy approximation methods that use thousands of parameters. Schulman et al. first provides a theoretical basis on which the algorithm TRPO is founded. The goal of TRPO is to find a method of regulating the step-size, which is required for complex nonlinear policies. Simply using a constant step-size will mostly result in non-convergence or local optima, especially for nonlinear policies. However using an step size that is to small most likely never converge.

Additions are made to the previously defined variables. First, the *advantage function* is defined. The advantage function is an often used substitution for either the state-value or action-value function. The advantage function normalizes the value function's value, by subtracting the state-value function from the action-values, as shown in equation 2-19.

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{2-19}$$

A_π is the advantage function, while $Q_\pi(s,a)$ is the action-value function and $V_\pi(s)$ the state-value function. Note that the value functions are discounted. Another term that needs introduction is the *policy performance* η, which is the discounted expected reward as defined in equation 2-20.

$$\eta(\pi) = \mathbb{E}_{s_0,a_0,\ldots}\Big[\sum_{t=0}^{\infty} \gamma r(s_t)\Big] \tag{2-20}$$

Schulman et al. utilized an useful identity that expresses the policy performance of one policy in terms of another policy and the advantage function, which was proven by Kakade and Langford [8]. Equation 2-21 shows this identity, where π and $\tilde{\pi}$ are both policies, and A_π is the advantage function determined by policy π.

$$\eta(\tilde{\pi}) = \eta(\pi) + \mathbb{E}_{s_0,a_0,\ldots\ \tilde{\pi}}\Big[\sum_{t=0}^{\infty} \gamma^t A_\pi(s_t,a_t)\Big] \tag{2-21}$$

This identity is expanded towards a form that resolves around states instead of timesteps, which adheres more to the "sample" based nature of reinforcement learning. This results in equation 2-22, where $\rho_{\tilde{\pi}}(s)$ is the discounted visitation frequency, a measure of state visitation.

$$\eta(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(a|s) A_\pi(s,a) \tag{2-22}$$

Equation 2-22 states in short that when the expected advantage at every state s is nonnegative, that the policy performance of $\tilde{\pi}$ is atleast equal or higher than the policy performance of policy π. This equation carries a lot of similarities when compared to the policy iteration theorem presented in section 2-2.

Note that in equation 2-22 there is a dependency on the state visitation frequency of the new policy. This is difficult to determine when using information from the old policy, so an approximation has to be made to the state visitation frequency. This is done by using the state visitation frequency of the old policy $\rho(\pi)$ instead of $\rho(\tilde{\pi})$ as an approximation. Proof is given that for a sufficiently small policy improvement step the derived identity still holds, now the question remains on how to determine the size of this step. Equation 2-23 gives the final *local* approximate identity depending on mentioned constrains.

$$L_\pi(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \tilde{\pi}(a|s) A_\pi(s,a) \tag{2-23}$$

As stated before, the question remains on how to determine a step size that is sufficiently small to not break the policy improvement guarentee, but still assures policy improvement. Kakade and Langford derived a theoretical lower bound for this stepsize, guaranteeing an improvement in policy. Schulman et al. expanded this to a theorem by utilizing the KL divergence between the old and new policy, resulting in equation 2-24. KL divergence is a measure of difference between two probability distributions. Using the KL divergence allowed the theorem to be of practical use for most policies.

$$\eta(\tilde{\pi}) \geq L_\pi(\tilde{\pi}) - C\dot{D}_{KL}^{max}(\pi, \tilde{\pi}) \tag{2-24}$$

$D_{KL}^{max}(\pi, \tilde{\pi})$ is the KL divergence between the two policies, $C = \frac{4\epsilon\gamma}{(1-\gamma)^2}$ and $\epsilon = max_{s,a}|A_\pi(s,a)|$. The combination of the KL divergence and the variable C define the lower bound approximation of the performance function. Before giving an explanation for the use of this equation, another term is introduced. Assume $M_i(\pi) = L_{\pi_i}(\pi) - C\dot{D}_{KL}^{max}(\pi_i, \pi)$, combining this with equation 2-24 results in equation 2-25.

$$\eta(\pi_{i+1}) - \eta(\pi_i) \geq M_i(\pi_{i+1}) - M(\pi_i) \tag{2-25}$$

Equation 2-25 shows that the actual policy performance is always higher or equal to that of M_i. This means that maximizing M_i will always result in a policy performance improvement. This algorithm of optimization is called a *minorization-maximization (MM)* algorithm. An overview of this process can be seen in figure 2-2.

Due to the fact that parameterized policies are learned, all the previous notations concerning a policy π will be subsituted for the policy parameters, θ. The resulting objective function is then the maximization of $M_i(\theta)$, parameterized by the policy parameters θ;

$$\underset{\theta}{\text{maximize}}\big[L_{\theta_{old}}(\theta) - C\dot{D}_{KL}^{max}(\theta_{old}, \theta)\big] \tag{2-26}$$

Schulman et al. noted that the when using C to penalize the function, the updates would be to small to result in convergence. Also determining a constant C that would give good

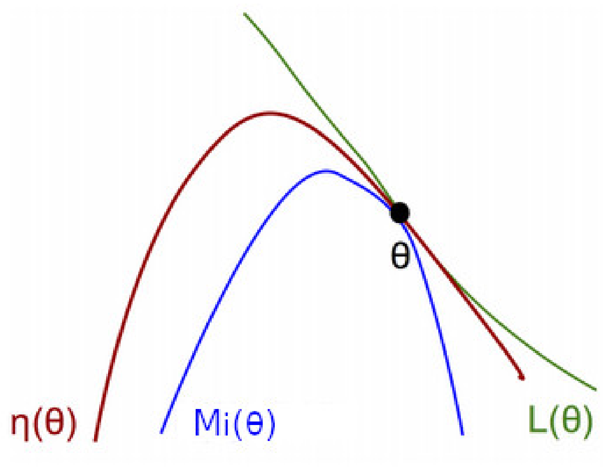


Figure 2-2: An graphical overview of the different used approximations of the performance measure (edited from source: [9])

performance posed difficult due to the dependency on the KL divergence. This is why instead of penalizing the KL divergence, a constraint is set on the KL divergence; $D_{KL}^{avg}(\theta_{old}, \theta) \leq \delta$. Note that the determination of the max KL divergence is deemed unstable, so the average KL divergence is used. Also, KL divergence has no upper bound, posing the threat of instability. This results in the final set of constrains that encapsulate Trust Region Policy Optimization, as shown in equation 2-27.

$$\begin{aligned}&\underset{\theta}{\text{maximize}}\ L_{\theta_{old}}(\theta)\\&\text{subject to}\ \bar{D}_{KL}(\theta_{old}, \theta) \leq \delta\end{aligned} \tag{2-27}$$

Now that the theoretical limits are in place, lets define a practical objective function to be optimized subjected to the theorem provided by equation 2-27. Recall equation 2-23. Rewrite the objective function in more convenient and widely used form. Equation 2-28 shows the subsequent surrogate objective function.

$$\begin{aligned}\Delta\eta &= \sum_s \rho_{\pi_{\theta_{old}}}(s) \sum_a \pi_\theta(a|s) A_{\pi_{\theta_{old}}}(s,a)\\&= \sum_s \rho_{\pi_{\theta_{old}}}(s) \sum_a \pi_{\theta_{old}}(a|s) \Big[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\pi_{\theta_{old}}}(s,a)\Big]\end{aligned} \tag{2-28}$$

The first part of the equation, $\sum_s \rho_{\pi_{\theta_{old}}}(s) \sum_a \pi_{\theta_{old}}(a|s)$ is determined by the trajectory returns of the old policy. The part that has to be optimized is contained in the brackets, $[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\pi_{\theta_{old}}}(s,a)]$. Combining equation 2-27 and equation 2-28 and converting it to a sample based approach results in equation 2-29.

$$\begin{aligned}&\underset{\theta}{\text{maximize}}\ \hat{\mathbb{E}}_t\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_{\pi_{\theta_{old}}}(s_t,a_t)\Big]\\&\text{subject to}\ \hat{\mathbb{E}}_t[\bar{D}_{KL}(\theta_{old}, \theta)] \leq \delta\end{aligned} \tag{2-29}$$

The TRPO method of optimizing this objective function results in the use of Monte Carlo roll outs to determine the state visitations, Q-values and KL divergence estimate. Then to optimize the policy parameters a conjugate gradient algorithm is used, which requires a local approximation of both the objective function and the constraint on the KL divergence. TRPO showed promising results on continuous state and action problems, such as a locomotion problem. Also it showed good convergence properties on a wide range of Atari games.

While TRPO has a good theoretical basis, the actual computational requirements are heavyg, and incompatible with some methods used in the field of deep learning. However, TRPO has set the stage for the use of Trust Regions and proven its worth. Trust Regions, atleast as an idea, are extensively used in most RL algorithms. From this point PPO is introduced, which is an computational less expensive method and easy to implement while building on the foundation of TRPO.

PPO is an simplification to the TRPO method, which allows the use of standard gradient decent methods in contrast to more extensive methods such as conjugate gradients required for TRPO. There are two general ideas behind PPO. TRPO provides a constraint on the update step by calculating the KL divergence between two policies. PPO suggests instead of applying a constraint, the update step should be limited depending on the actual probability ratios between the old and new policy. Recall equation 2-29 and define the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. Now instead of using the constraint, a new loss function is made that limits the update step due to constraints invoked on $r_t(\theta)$, as seen in equation 2-30.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\big] \tag{2-30}$$

Where ϵ is a tuneable hyper parameter. The min operator considers either the clipped or unclipped probability ratio. Lets take a look at figure 2-3. Figure 2-3 shows the effect of the modified loss function. In short, when under the new policy a certain action has a high chance of being executed with respect to the old policy, and the associated advantage function is positive, this can result in a large update step when unbounded. However, with this update rule the size of the step is limited by the clip boundaries, in this case ranging between $[1-\epsilon, 1+\epsilon]$. This ensures more conservative update steps. The same happens for a negative advantage function and a probability ratio that is well below 1.

The results presented in the PPO paper showed significant performance gains over other policy gradient methods. This in combination with ease of implementation and the ability to use more commonly used gradient methods makes PPO an state-of-the-art policy optimization method used in many different RL problems.

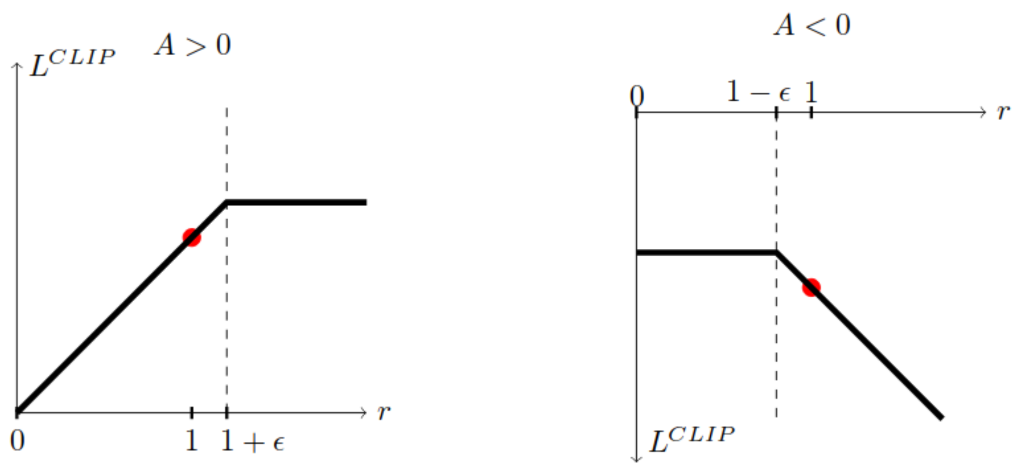


Figure 2-3: Depiction of interaction between environment and Agent

2-3 Hurdles of Reinforcement Learning

Reinforcement learning is notorious for instability and difficulty on converging towards an (optimal) solution. In any multi-agent extension of reinforcement learning these problems are amplified. In this section a few of the 'hurdles' are put in the spotlight, to keep them in mind when implementing RL techniques. Note that the single agent variants are discussed in this section, while the effects of multi-agent are discussed in section 4-1.

First off, the *Curse of dimensionality* is discussed. This "curse" depicts the growth of the state and action space when problem complexity increases. Simple discrete gridworlds, such as the example used in chapter 1-1, are on the lower end of the spectrum. They require relatively little iterations for convergence. When using a method such as Q-learning, each state has to be repeatedly visited in combination with every available action to build up the required value estimation. Assuming a tabular approach for this case, increasing the size of the gridworld and selection of available actions exponentially increases the amount of samples required to build up the value estimation. Then, shifting towards continuous state and action domains this increases even further. Generalization by function approximation, such as a neural network, will allow for generalization across the state space to counter act this. However, one can still imagine that more complex problems quickly strain computational requirements, which in turn demand better and more efficient function approximation. This comes most of the time at a cost, such as increased variance or other stability issues.

This exponential path of the increase in "search space" ties in closely to another hurdle, that of the *explore or exploit* dilemma. As stated before, to build up a representation of the environment through the use of values, states have to be visited. Initially, the agent begins its trajectory without any knowledge about the states. When visiting these states, it assigns value through the feedback of a reward signal. As an example for Q-learning, an ϵ-greedy policy is taken. This policy will take an action randomly with probability ϵ, while moving in the direction of maximum return otherwise. In the first few iterations of learning, only a few different trajectories are taken and consequently assigned a value. It will only deviate of its path of known value when taking a random action. When the search space increases, chances are that certain areas of this search space are never visited, implying that potential better solutions are also not found. In short, the solution presented is that of a local optima. This is where the explore vs exploit dillema originates from. When to use currently available knowledge and exploit this in search of the optimal solution, or when to deviate from this path to explore the search space in hopes of a finding a better solution.

Another hurdle is that of the *credit assignment* problem. The credit assignment problem originates from the difficulty in determining what actions have led to a certain reward. A certain trajectory of actions can lead to a reward, however determining what actions contributed positively towards the received reward is a problem. This can be partly overcome by decreasing the sparseness of the reward signal, however this ties into the problem of giving the agent to much information. This constrains the finding of exotic solutions by the RL agent, one of the core additions of value when using RL in the first place.

2-4 Function approximation by deep learning techniques

In this section, a quick overview is given of the various different *deep learning* techniques. They are explained to give a brief insight in how they work and most importantly their use. At the end a short section is given that will expand on the use of regularization and its importance.

The term deep learning originates from the use of multiple layers of mathematical *neurons*, that when put in a sequence will form a deep layered network. These neurons are the bread and butter of all *neural network* based techniques. The neuron in itself is strikingly simple in its form. Figure 2-4 shows a depiction of a single neuron, with the output of multiple neurons as input. Each neuron gives a certain value as output x_i, which is multiplied by a

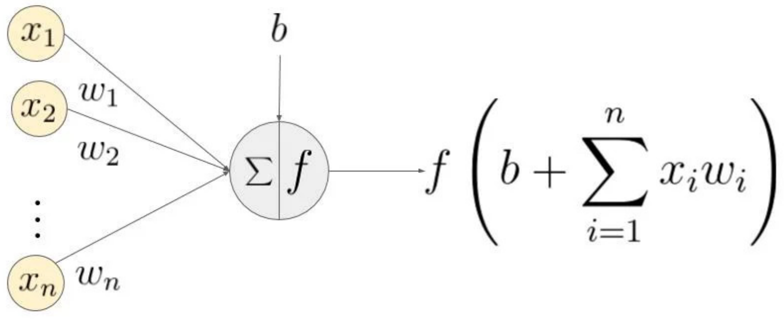


Figure 2-4: A graphical depiction of a neuron, at the core of Neural Network based techniques[10]

weight w_i. These values are then summed and shifted by the neuron with a bias b. This resulting value is then transformed by an activation function, which results in a the neurons output. The part that "learns" are the weights and biases, which can be modified by a wide range of numerical optimization methods, such as stochastic gradient descent. Also, a range of different activation functions are available to tailor the behaviour of the network to the problem. The concept underlying the activation function is that of enabling the neuron to "fire" when enough input signal is given. Most activation functions have step like behaviour, it will output zero until a certain threshold is met. After this threshold is met the activation function ”allows” information to be propagated forward. Note that an important aspect of the activation function is that it should be differentiable. This is needed to calculate the shift of weights and biases depending on the loss function.

Using these individual neurons as building blocks for layered networks is the essence of deep learning. The first and most classical use of these neurons in deep learning is that of the *Multi-layered Perceptron (MLP)*. The MLP are multiple layers of neurons stacked sequentially, creating a network. This is the most common method of function approximation.

When dealing with higher dimensional data, such as images, extensions are needed to cope with this extra dimension as well as to capture the spatial relation between values. In simple image examples, one could flatten the image and feed this into a MLP. However, when spatial features are important this relation has to be captured. This is where *Convolutional Neural Networks (CNN)* come into play. A CNN uses trainable *kernels* that convolute over an image, giving an output of the product of kernel and image. This kernel can be seen as a trainable ”lens” which is used to view the image. Using multiple kernels and layers of convolutions and stacking these in sequence establishes an network architecture that is able to detect a variety of various spatial features. CNN’s are commonly used for image classification or other computer vision based problems. The information is compressed through the architecture which in the end is then fed to an MLP for classification. Figure 2-5 shows an overview of such an architecture.

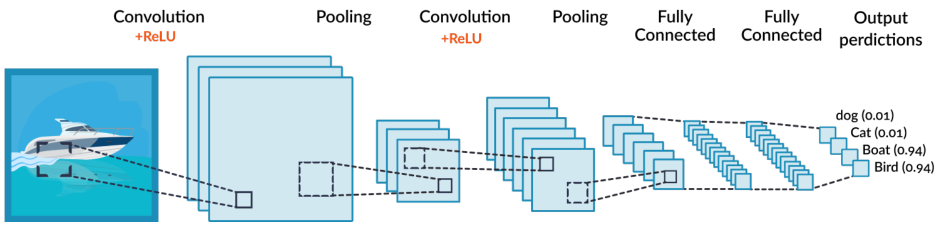


Figure 2-5: An overview of an CNN architecture, with an MLP at the end for classification [11]

For spatial information not originating from an 3D field such as an image, but for a sequence in time, *Recurrent Neural Networks* are used. Recurrent Neural Networks gain their recurrency from the fact that the past output of the network is used as an input, in conjunction with the new input. This is visualized in figure 2-6.

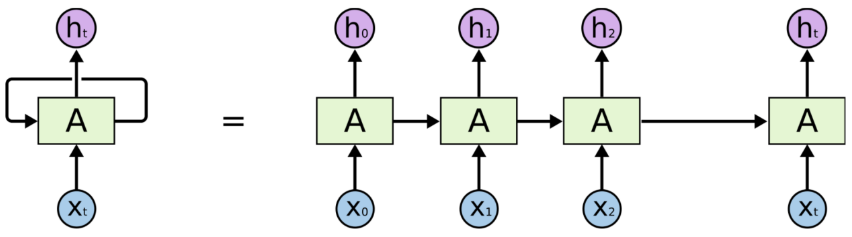


Figure 2-6: Structure of a Recurrent Neural Network [12]

The idea behind these types of networks is to have a form of "memory" that enables learning over sequences. For example, they are widely used in speech recognition or text classification. The RNN is a basis for more modern approaches to this memory property in neural networks, which resulted in *Gated Recurrent Unit (GRU)* and *Long Short-term Memory (LSTM)*. These are more advanced successors to the RNN, however their intent is the same.

Regularization and overfitting, which are basically the same but from a different point of view, is a big gripe within the neural network domain. An example of what happens when the network architecture is wrongly defined; If the network has to many layers, instead of "understanding" the underlying behaviour of the process, it could simply "remember" all the given inputs and outputs. When validating this network then on a new set of data, its success rate will be low, as this is new information not seen before and only knows what to do with the information already stored. The idea behind using any machine learning technique is to find the underlying structure of the information, not learning this information by heart. This understanding of the underlying structure is called generalization, which is achieved through regularization. Overfitting ties into the remembering of the answers. This is a delicate process that can be balanced with various techniques, different architectures and correctly presenting the data to the network.

2-5 Extensions of the Markov Decision Process

The *Markov decision Process (MDP)* is widely used as a mathematical framework for decision making problems. The MDP was introduced in section 1-2. In this section, variations of the MDP are expanded upon. These variants expand on the basis of the normal MDP, extending them to be utilized for various different schools of problems. The structure of this section is odd at first, but clearifies later on. In the first sections, the Markov Decision Process, *Partial Observable Markov Decision Process (POMDP)* and *Decentralized Partial Observable*

Markov Decision Process (Dec-POMDP) are explained in sequence. These sections will mostly contain mathematical definitions. In the final section, the order is reversed and more practical explanation is given. The reason for this is that in order to grasp the concept, extending from MDP to Dec-POMDP is a natural order. However, it is then shown that the MDP and POMDP are special cases of the Dec-POMD, which constitutes the reversal of order. Figure 2-7 shows the relation between the different MDP forms.

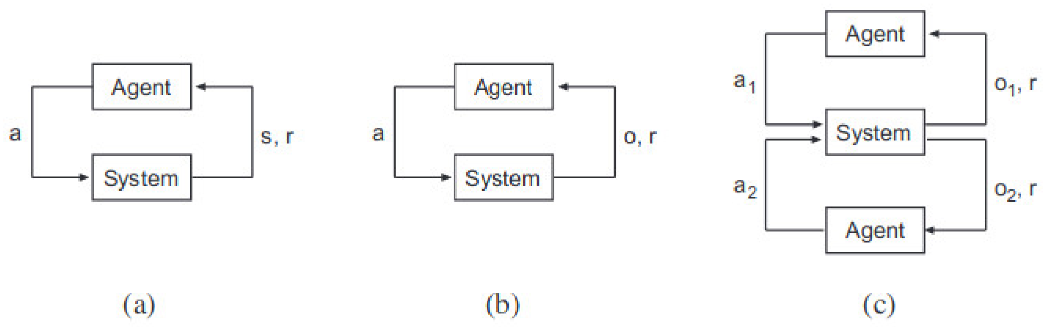


Figure 2-7: (a) Markov decision process. (b) Partially observable Markov decision process. (c) Decentralized partially observable Markov decision process with two agents.

The reason for this section is to provide insight in the various ways of defining decision making problems. This includes the fact that depending on the definition of the Markov process, the difficulty in solving the problem can scale from solvable to exceptionally hard. To define the solving difficulty, in most work complexity classes are used to categorize the difficulty of finding solutions. In this document however, these are omitted for the sake of simplicity. However, a relative difficulty is given for each framework compared to a standard MDP. Another reason is to generalize the multiple variants of the MDP to be used as baseline in following sections. Most Reinforcement Learning solutions use MDPs or variants thereof as a basis. This section is based on *Complexity analysis and optimal algorithms for decentralized decision making* by Bernstein [13, 14] and *Autonomous Agents and Multi-Agent Systems* by Shani et al.[15].

2-5-1 Mathematical framework for MDPs

The MDP models an agents actions in an environment, with the goal of maximizing long-term reward. An MDP is described by the tuple (S, A, P, R). S is a finite set of possible states, where $s \in S$ describes each state in this set. A is the finite set of possible actions, where $a \in A$ describes each action available in this set. P are the transition probabilities, which describe the chance that when taking an action a in state s, that the agent ends up in state s'. In short, $P(s'|s, a)$. R is the reward function, quantifying the amount of received reward when taking action a in state s, described by $R(s, a)$.

Partially Observable Markov Decision Process

In comparison to an MDP, when an agent is unable to directly observe the state of the environment, it has partial observability. Instead it receives noisy observations of the state in comparison to direct state observations when looking at an MDP. A Partially Observable Markov Decision Process (POMDP) can be described by the following tuple, (S, A, P, R, ω, O) where S, A, P, R are defined exactly the same as for an MDP. ω is the observation the agent receives after taking action a, and O is the conditional probability of the observation the Agent receives; $O(o|a, s')$. This implies that the Agent has no direct observability of the environment states, s, but makes a observation ω that are under probability O of reflecting the true state.

Decentralized Partially Observable Markov Decision Process

Now the POMDP is extended to the case when the Markov Process is controlled by a team of distributed agents. This means that actions are taken based on each agents own local observations, not by a centralized party that has knowledge of the full system. Decentralized Partially Observable Markov Decision Proceses (Dec-POMDP) is a multi-agent generalization of a POMDP. Assume m are the number of Agents, a Dec-POMDP is described by the following tuple; $(m, S, A_m, P, R, \omega_m, O)$. S is the finite set of states. A_m is the set of actions taken by each agent, i.e. the set of joint actions $(a_1, ..., a_m)$. P is the transition probability, $P(s'|s, a_1, ..., a_m)$. R the reward function, $R(s, a_1, ..., a_m)$. ω_m the joint set of observations by each agent, i.e. $(o_1, ..., o_m)$. O contains the observational probabilities, $O(o_1, ..., o_m|a_1, ..., a_m, s')$, the probability of Agents seeing observation o, given that a was taken resulting in state s'.

2-5-2 the Dec-POMDP Family

All techniques explained in the previous section can all be seen as special cases of a Dec-POMDP, as seen in figure 2-8.

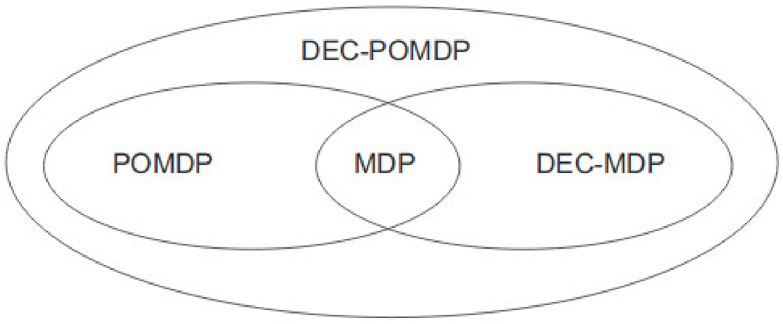


Figure 2-8: Relationship among the different MDP models

First, lets expand the running example to fit the description of a Dec-POMDP. Note, that the state definition is anything from well defined for this problem, but is used as an intuitive example. The state definitions are defined this way to exaggerate the partial observability of the example. Looking at figure 2-9, the problem can now be defined into a distributed system.

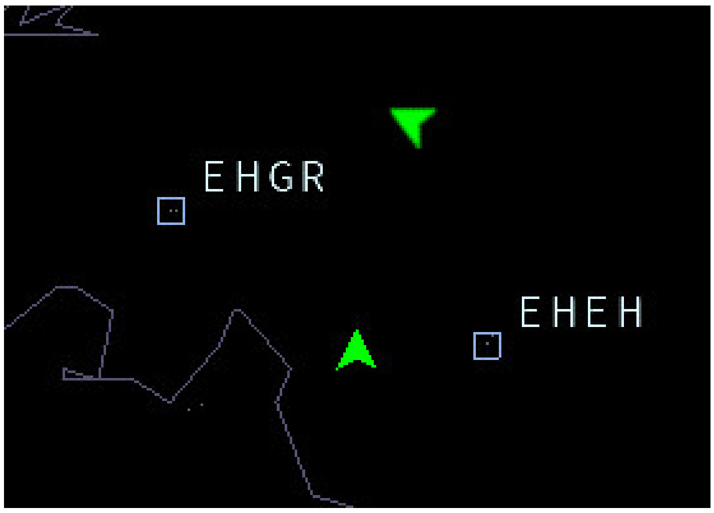


Figure 2-9: Example of a multi-agent system, used to describe the various MDP variants

Each aircraft is considered an agent, where each agents observes its own state; longitude, latitude, current heading and distance and relative bearing to the destination. Also, each agent observes the distance to the other agent. The current definition of distance to another agent makes agent's location partially observable, due to the fact that heading of the other agent is not included.
Now each agent makes decisions based on its local observations. The subsequent state transition depends on the collection of joint actions of each individual agent. More so, the collection of joint actions result in a collective reward, where each agent contributed to this reward. Note that there is now a strong decoupling between each agents individual action and contribution to the reward. A Dec-POMDP is proven to be significantly more difficult to solve in a worst-case scenario, compared to a standard MDP.

A special case of the Dec-POMDP is the Dec-MDP. The difference between the two is that when the observations of all agents in the environment collectively encapsulate the whole state, it is considered an Dec-MDP, even though no single agent observes the whole state. This opens up some flexibility in solving the problem, however in the worst case scenario it is still proven to be significantly more difficult to solve compared to a standard MDP. The example problem would be considered a Dec-MDP when the heading of the other agent would also be observable.

Moving from a Decentralized approach, Dec-POMDP, to a centralized approach, POMDP, reduces the number of agents to one. Instead of each aircraft making its own decisions, a centralized agent is used to determine the action of each aircraft from a global point of view. As an example relative distances between aircraft would become obsolete, due to having access to full state, i.e. each location of the aircraft etc. As an example, the observations of this agent would include all location and heading information of the aircraft simultaneously. The action space would consists of two heading changes that are given to the two aircraft in the sector, all determined by the centralized agent. In contrast to the decentralized approach, where each agent would only determine a single action for itself. In the running example, partial observability would be difficult to express, due to the completeness of information available. However, partial observability could be seen as instead of receiving absolute cartesian coordinates of each aircraft, a noisy estimates of these coordinates are given. POMDP are considered relatively solvable compared to their decentralized variants, however still more difficult than a standard MDP. Decentralization add lots of decision and state space complexity. When considering this example without the noise on the coordinates, the problem is subsequently reduced to a standard MDP. A MDP is considered the least difficult framework to use and solve.

The different presented frameworks offer different ways of approaching the problem. Logically, a standard MDP framework is desired due to the fact that these are significantly easier to solve compared to the other frameworks. However, the size of the MDP would grow rapidly when adding more agents, exploding state space and action space representation. More insight in this is provided throughout the document. Lots of research dive into the use of different MDP representations, and multiple representations can be combined to reduce complexity locally and creating a more computationally efficient and solvable problem, such as centralized learning with decentralized execution.

Chapter 3

Problem Description and Research Question

As stated in the introduction of this document, there is less of a problem, but more of a curiosity present. Reinforcement Learning has seen promising results over the years, and applying this to air traffic control could lead to new emergent behaviour or other air traffic management solutions. The "ultimate" goal would consists of a reinforcement learning method that could fully take over the task of air traffic control while ensuring safety and maximizing efficiency. However, finding "a" solution will also be considered satisfactory. To summarize this vision, a research question is established.

How to apply reinforcement learning to the air traffic control task of merging and spacing while maximizing safety and efficiency?

As can be seen, this research question can be split up in three distinct parts, which can be answered through sub-questions.

How to apply reinforcement learning:

- How to deal with the multi-agent nature of the problem?
- How should the state space be defined?
- How should the action space be defined?
- How should the reward structure be defined?

Air traffic control:

- How to simulate the Air Traffic Control setting?
- How to setup the experiment?

- How to maximize data throughput?

Maximizing safety and efficiency

- How to define efficiency?
- How to ensure safety?

The questions will be used as chapters, and the subsequent sub-questions as sections. By using these questions as guide, a complete overview of the required knowledge to solve the questions is provided. Note that, while not clearly stated before, it is assumed that from this point onwards a Multi-Agent Reinforcement Learning (MARL) approach is used as baseline. The reason why MARL is taken as baseline is that the problem most likely will be categorized as a MARL problem. When this is not the case, MARL gives insight in how to deal with certain aspects of the ATC problem that persist even when taking a single agent based approach.

Chapter 4

How to apply Reinforcement Learning

In this chapter, the question *How to apply Reinforcement Learning* is being investigated. The sections are divided along the predefined sub-question structure. First, the multi-agent problem is expanded upon. Prior understanding of Reinforcement Learning explained previously in this document is well advised, as the rate in which more advanced techniques will be expanded upon will increase rapidly. Then, defining state, action and reward structures will be discussed in this order.

4-1 The Multi-agent problem

Air traffic control concerns itself with a region containing multiple aircraft that all need to be guided towards their destination. Each aircraft has its own destination and current position in the airspace, which can result in conflicts with other aircraft while navigating. For reinforcement learning to be effective while guiding these aircraft, solutions need to be made with regard to all aircraft, due to the fact that changes made to one aircraft, influence the options available of other aircraft. This is the first main challenge that arises from using multi-agent systems, the non-stationarity of the environment. All agents observe the impact of actions on the environment of other agents, instead of only the result of their own action. This makes learning difficult, due to the fact that an action a single agent took, can possibly not be accredited solely to that agent himself, but also due to the action of other agents. On top of this, each agent evolves with each time step by updating its policy, increasing unpredictability from a single agent's point of view. This non-stationarity of the environment violates the Markov Property, which implies no guarantee of convergence to the optimal policy.

Another issue is how conflicts are resolved in a multi-agent setting. In a multi-agent setting, the distinction can be made between either cooperation or competition between agents. This means that either the total reward is prevalent, or the agents individual reward is more important with disregard for the other agents performance. In the air traffic control scenario, solely cooperative tasks are considered. The reason for this is that in air traffic control the global optimal solution is required, compared to a single aircraft having the least amount of travel time, while the other agents behave sub-optimal.

Another important factor is the availability of information regarding other agents for each agent. For cooperative behaviour to be fully cooperative, knowledge of each agent action is required. An good example of this would be conflict resolution within the airspace. When two aircraft are heading towards an LoS, a solution is needed. When there would be no cooperation or communication about each agents action intent, when both aircraft would turn west, the conflict would persist. However, communicating that one of the two agents will only take action, or both turning in opposite direction would resolve the conflict.

Lastly, the system will likely be partially observable. This is due to the fact that it is not realistic for each agent to have full observability of the system. The state space of each agent grows exponentially with each added agent, when full observability is wanted, exploding the state space. So partial observability is needed, for the agent to only consider its immediate surroundings. This is a realistic setting in an ATC environment. Single aircraft are flying towards their far-off goal, while locally solving conflicts. [16][17]

Multi-agent Reinforcement Learning (MARL) is a field on its own, due to the differences in environment and communication compared to single agent reinforcement learning. In the next section multiple different approaches are presented that work with the non-stationarity and communication problems of MARL. A distinction is made between value based methods, such as extensions of DQN techniques and policy gradient based methods, such as COMA.

4-1-1 Value based reinforcement learning

This section will contain multiple *value based* reinforcement learning techniques that are designed to work with MARL problems. First off, an fundamental first step towards MARL is presented, namely independent Q-learning. After that, an solution is presented to the principle of Q-value overestimation and its non-stationarity. Then an section is given about how to implement specific communication into RL algorithms. Finally the concept of leniency is introduced, in conjunction with double Q-learning techniques.

Independent Q-learning

As introduced in section 2-1, Q-learning is a fundamental technique for off-policy reinforcement learning. Q-learning was extended by approximating the Q-values using a CNN, dubbed Deep Q-networks (DQN). For MARL, an extension is made to Q-learning called *independent Q-learning* [18], which basically means that every agent learns its own Q-function that only depends on its own state and actions. IQL considers other agents stationary parts of the environment. In practice, this can work for more simple MARL problems. However, for more complex problems, IQL is extended by using deep learning. This is done by applying the working principles of *Deep Q-networks (DQN)* on IQL

As discussed in subsection 2-1, DQN is stabilized by experience replay. However, when applied to multi-agent problems, experience replay does not hold, due to non-stationarity. The used experience that is learnt from does not reflect the environment correctly, due to the ever changing policies of all the agents. To make DQN work, a few modifications have to be presented to experience replay mechanic. First off all, the state space of each agent could include the whole environment including policies of all other agent. This makes the environment non-stationary, but at great costs of exploding state-space. To work around this, Foerster et al. [19] proposes two techniques, *Multi-Agent Importance Sampling* and *Multi-Agent Fingerprints.*

Multi-Agent Importance Sampling is an extension of importance sampling. Importance Sampling is used in off-policy methods that optimize over expected values. This is due to the fact, that inherently to off-policy methods, the agent gathers experience with a different policy than the to be learned optimal policy. This distinction in policies are called the behavioural policy and target policy, already touched upon in section 2-1. The agent is trying to find the optimal target policy, while learning from behaviour done under another policy. The target en behavioural policies can be seen as two different random variables under high correlation. [4]

Foerster et al. state that the non-stationarity of the environment can be reduced by using importance sampling not for correcting off-policy gained data, but on *off-environment* gained data. What this means is that in both naive IQL and DQN each agent considers the other agents as part of the environment. However, with time each agent changes its behaviour, indirectly changing the environment (because the agents are considered part of the environment). The experience replay consequently stores, instead of only off-policy data, also off-environment data. This can be corrected for by using multi-agent importance Sampling. Equation 4-1 shows the augmented loss function. Note that in the explanation of all variables, joint means a collective. For example, $\boldsymbol{u_{-a}}$ means the collective actions of all agent other than a

$$\mathcal{L}(\theta) = \sum_{i=1}^{b} \frac{\pi_{-a}^{t_r}(\mathbf{u}_{-a}|s)}{\pi_{-a}^{t_i}(\mathbf{u}_{-a}|s)} [(y_i^{DQN} - Q(s,u;\sigma))^2] \tag{4-1}$$

- θ : The parameters of the value-function approximation.
- b : The amount of sampling batches from Experience Replay memory.
- $\boldsymbol{\pi}$: Denotes the joint policy.
- $-a$: Denotes joint quantities over all agents other than a.
- $\boldsymbol{u}$: Joint actions.
- s : State.
- t : Time, of either the time of the replay t_r or time of the collected sample t_i.
- y_i^{DQN} : Output of the DQN.

Equation 4-1 instead of using importance sampling to correct for a older policy with respect to the newer policy, it corrects the change of all the other agents policies with the current policies of all other agents. Which in turn depicts the change in environment.

Another method for improving experience replay, is the use of multi-agent fingerprints. As stated before, but repeated for clarity, IQL treats other agents as part of its environment and ignores the fact that the policies of other agents are changing, making the Q-function non-stationary. This means that if the Q-function could take into account the change in behaviour, i.e. the policy of other agents, the Q-function could be made stationary. This could be done by directly incorporating the policy of all other DQN agents into the observation space. However, due to the amount of parameters θ each DQN has, the observation space would explode. However, Foerster et al. noted that not all possible network configurations where required, only those that appear in the replay memory. The sequence of policy data can be seen as a single trajectory of improvement through the high-dimensional policy space. For the DQN to be able to keep track of this trajectory, i.e. link the changes of the environment with that of the policy changes, a fingerprint is needed. Foerster et al. created a simple fingerprint, consisting of ϵ and e. ϵ is the training iteration number, and e is the rate of exploration. This fingerprint was added to the agents state-space.

It was shown that both methods where able to improve experience replay in a MARL setting. Both techniques are made around solving the non-stationarity problem, resulting in comparable results between multi-agent importance sampling and multi-agent fingerprints.

Q-value overestimation and Deep Repeated Update Q-networks

Castaneda [20] proposes a method for stabilizing non-stationary environments, *Deep Repeated Update Q-network (DRUQN).* DRUQN is based on an inherent deficiency when using function approximation with Q-learning, the tendency to overestimate Q-values. This overestimation is amplified by a non-stationary environment. First, the origin of Q-value overestimation is discussed, as this is a recurrent topic in MARL. After that DRUQN is expanded upon.

Thrun and Schwartz [21] quantified the overestimation problem with Q-learning using function approximation. Equation 4-2 shows the Q-learning update rule, where s is the state, taking action a, resulting in state s' and receiving a reward r_s^a. γ is the discount factor.

$$Q(s,a) \leftarrow r_s^a + \gamma \max_{\hat{a}} Q(s', \hat{a}) \tag{4-2}$$

When using function approximation for Q, it introduces noise, depicted in equation 4-3, where Q^{approx} is the Q-value approximation, Q^{target} the true value and $Y_{s'}^{\hat{a}}$ the introduced random variable that depicts noise due to function approximation. $Y_{s'}^{\hat{a}}$ is assumed to have a zero mean.

$$Q^{approx} = Q^{target} + Y_{s'}^{\hat{a}} \tag{4-3}$$

Thrun and Schwartz then created a simple comparison to show the effect of this noise component on Q-learning, shown in equation 4-4, where Z_s is the newly defined random variable.

$$\begin{aligned} Z_s &= r_s^a + \gamma \max_{\hat{a}} Q^{approx}(s', \hat{a}) - \big(r_s^a + \gamma \max_{\hat{a}} Q^{target}(s', \hat{a})\big) \\ &= \gamma\big(\max_{\hat{a}} Q^{approx}(s', \hat{a}) - \max_{\hat{a}} Q^{target}(s', \hat{a})\big) \end{aligned} \tag{4-4}$$

When looking at equation 4-4, when taking into account the added noise component $Y_{s'}^{\hat{a}}$ to Q^{approx} in combination with taking the max of available actions, will often result in a positive value. This means that even if $\mathbb{E}[Y_{s'}^{\hat{a}}] = 0$, often $\mathbb{E}[Z_s] > 0$. This is the result of using the max operator in Q-learning in combination with stochastic function approximation. This results in a upwardly bias of Q-learning, frequently reinforcing certain actions that have a high value. In non-stationary environments this effect is amplified even further, due to changing policies. Previously optimal actions would still be selected, even when sub-optimal[20].
Note that, even considering this upward bias and non-stationary of the environment, with enough iterations Q-learning can still converge to the optimal values. However, the probability of this happening diminishes rapidly in higher order state space environments.

Abdallah and Kaisers [22] addressed this issue by introducing *Repeated Update Q-learning (RUQL)*. They introduced a term called *policy-bias*. This refers to the dependency between the *rate of update* for an action value and the *probability of selecting* the corresponding action. This phenomena is more severe when considering the trend of Q-value overestimating, especially in combination with non-stationary environments, as explained in the previous. The main idea of RUQL is to repeat the update of an action inversely proportional to the probability of choosing that action, i.e. $\frac{1}{\pi(s|a)}$ times. So when less frequently visited actions are taken, this action value is then reinforced multiple times in comparison to high probability actions. However, directly using $\frac{1}{\pi(s|a)}$ poses issues, due to the fact that the times of updates explodes when $\pi(s|a)$ approaches zero, as well as fact that only a integer number of iterations can be done, and $\frac{1}{\pi(s|a)}$ results often in a non-integer. To solve for this, the final RUQL Equation 4-5 shows the augmented Q-learning update rule.

$$\begin{aligned} Q^{t+1}(s,a) &= [1-\alpha]^{\frac{1}{\pi(s,a)}} Q^t(s,a) + \big[1 - (1-\alpha)^{\frac{1}{\pi(s,a)}}\big][r + \gamma \max_{a'} Q^t(s', a')] \\ &\text{using } z_{\pi(s,a)} = 1 - (1-\alpha)^{\frac{1}{\pi(s,a)}} \\ &= Q(s,a) + z_{\pi(s,a)}[r + \gamma \max_{a} Q(s', a') - Q(s,a)] \end{aligned} \tag{4-5}$$

Castaneda extended RUQL to the Deep Learning domain, dubbed DRUQN. DRUQN requires only slight modification to the loss function differentiation. Combining equation 4-5 with the deep learning extension mentioned above results in equation 4-6.

$$\alpha \nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s',z_{\pi(s,a)})\sim U(D)}[\mathbf{z}_i(y_i - Q(s,a:\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)] \tag{4-6}$$

α is included in equation 4-6 for clarification in the differences compared to ordinary Q-learning. $\mathbf{z}_i$ now contains the modified learning rate, which is the addition of the RUQL, i.e. the factor that balances the frequency of Q-value updates.It was shown that DRUQN improved multi-agent behaviour on almost all environments compared to naive DQN methods.

Communication with Deep Loosely Coupled Q-networks

Castaneda also introduced the concept of *Deep Loosely Coupled Q-network(DLCQL)*, which is an deep learning extension of *Loosely Coupled Q-Learning (LCQL)*[23]. DLCQN works around the non-stationarity problem by issuing a mode of communication between agents when required. Communication is only done when other agents significantly influence each others state, while the influence is minimal the agent only takes its own local observations into consideration. First, LCQL fundamentals are considered, then the deep learning approach by Castaneda is given.

Yu et al. used the difference in various MDP frameworks to split up the problem into grades of independence of each individual agent. Described in section 2-5 are the various MDP frameworks. Yu et al. expanded on the definition of a Dec-MDP by stating that the transition function P and reward function R could be split up in a function that is solely build up from each agents individual reward and transitions, which is called transition/reward-independent Dec-MDP. For the transition probabilities it was stated that the next local state of a single agent, is independent of the local states of other agents, only depending on its own previous local state and taken action. For the reward function it was stated that this could be build up as a function of each agents individually gained reward during each step; $R(s,a) = f(R_1(s_1,a_1),...,R_n(s_n,a_n)$. This special case of a Dec-MDP results in the ability to split up the Dec-MDP in n independent MDPs which can be solved for separately. This however can only be achieved when the agents are indeed not influencing each other. In most problems, this is not the case, and agents do influence each others local states. The problem can then be split up in parts where agents are either independent or dependent. To quantify this an *independence Degree* $\mathcal{E}$ is introduced, as well as two Q-functions that define the local and dependent part of the problem. Equation 4-7 shows the composition of the Q-function, where $Q^k(\hat{s_k},a_k)$ is the local value function, $Q^I(s,a)$ is the dependent value function and $\mathcal{E}$ is the degree of independence. Note, the local value function depends on $\hat{s_k}$ and a_k, which are the local states and actions of each agent individually, where the dependent function relies on the full state and action representation.

$$Q_k(s,a) = \mathcal{E}^k Q^k(\hat{s_k},a_k) + (1-\mathcal{E}^k)Q^I(s,a) \tag{4-7}$$

To implement this independence degree, a few definitions are required. First, it is assumed that when an agent receives negative reward, i.e. a collision occurs, that coordination is required in the corresponding state. However, not only the state the agent is in now is fully responsible for the collision. Previously visited states that led to this situation contributed to this state. First the similarity between state is calculated. Yu et al. defined this as absolute numerical difference between states, where $\zeta(s,s^*)$ is the similarity between state s and the target state s^*. $\zeta(s,s^*)$ is then used to create a diffusion function, that depending on the similarity, creates an *factor of influence* to be used as a reflection of contribution towards the negatively gained reward. Equation 4-8 is the diffusion equation, where r is the gained reward, s^* the target state that contained the reward and s the state considered to have had influence on this reward.

$$f^r_{s^*}(s) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\zeta(s,s^*)^2} \tag{4-8}$$

However, not only similarity is required to determine independence due to the fact that multiple states can share similarity but not having had influence on the current trajectory towards the negative reward. To solve for this, an eligibility trace ε is introduced to keep track of the previously visited states. An eligibility trace is a numerical value that keeps track of what states are visited, decaying in intensity each time step. Combining these result in a definition of the independence degree, as seen in equation 4-9. Yu et al. showed in an experimental setting that the proposed approach resulted in near-optimal performance on multiple environments.

$$\mathcal{E}_i^k(t+1) = \mathcal{E}_i^k(t) + \varepsilon_i^k(t) f_{s^*}^r\left(s_i^k\right) \tag{4-9}$$

Castaneda extended this principle to the deep learning domain. The first problem encountered with determining the independence degree is the fact that when using function approximation, it is unfeasible to do the required similarity comparison between each state. Castaneda proposed to adjust the independence degree based on if another agent is observed when receiving negative reward. The similarity degree is then defined based on the similarity between states of both agents, $\zeta(s_j, s_k)$. The eligibility trace is now used to keep track of when other agents are observed, instead of what states are visited.For DLCQL two value functions have to be parameterized. The local agents Q-value function, Q_k and the cooperative variant, Q_c. Q-value parameterization and parameter loss function can be found in section 2-1-3.
In conclusion Castaneda showed results on multiple environments, where DLCQL would out perform standard DQN. However, depending on the specific environment the performance changed drastically.

Solving Q-value overestimation with Weighted Double DQN's

Another method of working around the Q-value overestimation problem, that inherently helps against the non-stationarity problem of MARL, is *Weighted Double Deep Multi-agent Reinforcement Learning (WDDQN)* [24] by Zheng et al. This method is an extension to the *Double Deep Q-networks (Double DQN's)* [25] introduced by Hasselt et al, with the addition of a *Lenient Reward Network (LRN)* [26]. However, before WDDQN is explained, the origins of double Q-networks are expanded upon. After that, leniency and its use are considered. Finally, the workings of WDDQN are presented.

Hasselt et al. tackled the Q-value overestimation problem by learning two separate Q-value functions. For completeness, a small recap is given, however approached from an different angle. The overestimation arises from the fact that the Q-value is estimated on the future expected reward. The future expected reward is calculated by using the Q-value in transitioned state, and picking the action that maximizes this value. There is now a significant chance that this action that maximizes the Q-value is based on an overestimated Q-value. Subsequently, the Q-function is updated by this overestimated Q-value which results in an positive upwards bias. Hasselt et al. wanted to decouple this process, by learning two separate Q-functions. Both Q-functions are estimated using Q-learning, but without any overlap in samples so that both functions are estimated on their own distinct subset of the complete sample space. This reduces the chance of overestimating the same Q-value twice. The two functions are used in cohesion. One is used to determine the action to maximize the future expected reward

Q-value, while the other one is used to determine the actual Q-value while using the determined best action. This can be seen in equation 4-10. For clarity, this equations shows the tabular implementation of Double Q-networks. Q^A and Q^B are the two Q-functions, while the rest follows the standard notation already introduced. Note that during each step, either the action selection comes from Q^A and Q^B is the target, or the other way around.

$$\begin{aligned} &\text{define}\, a^* = arg\, max_a Q^A(s,a) \\ &Q^A(s,a) \leftarrow Q^A(s,a) + \alpha\big(r + \gamma Q^B(s',a^*) - Q^A(s,a)\big) \\ &\text{define}\, b^* = arg\, max_a Q^B(s,a) \\ &Q^B(s,a) \leftarrow Q^B(s,a) + \alpha\big(r + \gamma Q^A(s',b^*) - Q^B(s,a)\big) \end{aligned} \tag{4-10}$$

It was proved that Double-QN significantly reduced overestimation of the Q-value, however it introduced an underestimation of the Q-value. It is proven empirically that this underestimation is preferable to overestimation, but still unwanted. To solve either the over- or underestimation of the Q-value, *Weighted Double Q-learning (WDQ)* by Zhang et al. [27] was introduced. WDQ balances normal Q-learning, which leads to overestimation, with Double-QN, which leads to underestimation. This is done by introducing a dynamic heuristic value β depending on a set constant c that determines this balancing. This heuristic value is based around the Kullback-Leibler divergence, which gives a notion of difference between two random distributions. In this case, the two sets of data used for the updating of the Double-QN. As seen in equation 4-10, depending on the Q-value used to determine the best action, a^*, the other Q-value is used to update the Q-value. When the same Q-value that is used to determine the best action is used to update its Q-value, the equation reduces to ordinary Q-learning. This is what WDQ balances, and can be seen in equation 4-11.

$$\begin{aligned} &\text{define}\, a^* = arg\, max_a Q^A(s,a) \\ &Q^{A,WDQ}(s,a) = \beta Q^A(s,a^*) + (1-\beta)Q^B(s,a^*) \\ &\text{define}\, b^* = arg\, max_a Q^B(s,a) \\ &Q^{B,WDQ}(s,a) = \beta Q^B(s,b^*) + (1-\beta)Q^A(s,b^*) \end{aligned} \tag{4-11}$$

As can be seen in equation 4-11, the equation reduces to ordinary Q-learning when β is 1. So the first term is the term that induces the overestimation. When β is 0, the equation used ordinary Double Q-learning, which tends to underestimate.

However, all of the above is still considering the case of tabular Q-values, and a leap to the deep learning domain has to be made. This is done in the same fashion as with DQN's, which can be seen in section 2-1-3. Zheng et al. extended WDQ to the deep learning domain, as well as introducing auxiliary mechanisms to stabilize the multi agent setting for the MARL problem. One of these auxiliary mechanism is a *Lenient Reward Network*, based on *Lenient Multi-Agent Deep Reinforcement Learning* [26].

Rewards tend to have a stochastic nature due to the non-stationarity of the environment. Zheng et al. used a neural network to approximate the reward function, $R(s,a)$. This function approximation averages the reward for each state-action pair, and is trained by using the stored transitions in the replay memory. However, due to the mentioned above multi-agent interference on the reward additional enhancements are needed to the reward signal, which resulted in the aforementioned Lenient Reward Network. The update rule for this network is as follows:

$$R_{t+1}(s_t, a_t) = \begin{cases} R_t(s_t, a_t) + \alpha\delta & \text{if } \delta > 0 \text{ or } x > l(s_t, a_t). \\ R_t(s_t, a_t) & \text{otherwise} \end{cases}$$

$R_t(s_t, a_t)$ is the reward approximation of the state-action pair, $\delta = \overline{r}_t^{s,a} - R_t(s_t, a_t)$ is the error between the approximation and the target reward, $\overline{r}_t^{s,a}$. The target reward is the average of all immediate rewards of that state-action pair stored in the transition memory. $l(s_t, a_t)$ is the *leniency* that is associated with the state-action pair. Leniency ensures that in initial stages of the training, the agent remains optimistic, ignoring negative reward updates. Palmer et al. [26] introduced the general concept of leniency, however applied to Q-value updates in contrast to the lenient reward network, but the driving principles are the same. In short, lenient agents map Q-values to a decaying temperature value that determines the leniency towards negative Q-value updates. This results in optimistic Q-value update behaviour, favouring positive over negative updates. Empirically it is proven that this stimulates cooperation between agents in a cooperative MARL setting. Another problem in MARL is called *relative overgeneralization*. This occurs when agents gravitate towards working, but sub-optimal joint policies. This is due to the fact that the mutual influence of multiple agents on the environment creates noise for the policy updates. Leniency has shown that it increases likelihood of convergence towards a global optimal solution in contrast to sub-optimal solutions, due to the inherent property of leniency promoting exploration of unvisited and positive effect states. The leniency towards more frequently visited state-action pairs decreases over time, while less visited state-action pairs retain their optimism, encouraging exploration. To put the concept of temperature and leniency into context, equation 4-12 is given, where $T_t(s_t, a_t)$ is the current temperature, K the leniency moderation factor determining the rate of decay and $l(s_t, a_t)$ is the leniency value. Note that each state-action pairs is initialized with a maximum temperature, and is decayed each time this state-action pair is visited.

$$l(s_t, a_t) = 1 - e^{-K * T_t(s_t, a_t)} \tag{4-12}$$

After an update, the temperature is adjusted by a simple rule using a discount factor β; $T_{t+1}(s_t, a_t) = \beta T_t(s_t, a_t)$.

Leniency is then subsequently applied to the Q-value update as follows:

$$Q(s_t, a_t) = \begin{cases} Q(s_t, a_t) + \alpha\delta & \text{if } \delta > 0 \text{ or } x > l(s_t, a_t). \\ Q(s_t, a_t) & \text{if } \delta < 0 \text{ and } x < l(s_t, a_t). \end{cases}$$

where δ is the TD-error, $\delta = Y_t - Q(s_t, a_t; \theta_t)$ and $x \sim U(0, 1)$ is a random variable used to ensure that an negative update is done with a probability of $1 - l(s_t, a_t)$.
The above considers Q-values that are stored in a tabular fashion. Like all the other methods, this is expanded towards the deep learning domain. However, when the concept is used for a continuous state space, a function approximation for the temperature value is also needed to generalize across similar states. Palmer et al. solved this by training an auto encoder to

cluster states, which in turn allows for applying a temperature value. An auto-encoder tries to replicate its input, while creating a "bottleneck" in the network structure. This basically implies that the state representation is compressed, resulting in a so called "hash-key". This hash-key is then a compressed representation of the state.

Applying leniency however is not straight forward, and a few algorithmic auxiliary methods have to be introduced. A problem with the temperature decay, is that when a similar initial state are used, states in close proximity of the initial state decay rapidly due to often visitation of these states. This is where Palmer et al. introduced *Average Temperature Folding (ATF)*. The principle behind ATF is that the temperature of the current state is adjusted by balancing the change in temperature using the specific state-action decay and the average temperature of the next state. This balancing is done by another hyper-parameter v, as seen in equation 4-13. $\overline{T}_t(s_{t+1})$ is the average temperature of the state transition.

$$T_{t+1} = \beta(1-v)T_t(s_t, a_t) + v\overline{T}_t(s_{t+1}) \tag{4-13}$$

It is shown however, that in certain cases where rewards are sparse, that ATF is not sufficient in solving the rapid decay problem. So on top of ATF, a *Retroactive Temperature Decay Schedule* is introduced. In short, this is a in-episode schedule that decreases decay in earlier phases of the episode in contrast to later phases where decay is higher. This balances decay even more for more visited initial states, balancing decay across the whole episode. This is done by making a schedule for the temperature discount faction β. The scheduled β is determined by using an exponential function that can be adjusted using a certain decay rate,d^t, and exponent p, as seen in equation 4-14. n is the step limit, where $\max n$ is the step limit, i.e. terminal state.

$$\beta_n = e^{pd^t} \tag{4-14}$$

Finally, an modification is made to the standard ϵ-greedy exploration strategy. Intuitively, the temperature that is used throughout the Lenient-DQN is also an indication of visitation of certain state-action pairs, which in turn when temperature is low gives a certain degree of certainty to the state-action values. So, in contrast to the standard ϵ-greedy, $\overline{T}(s_t)$-Greedy is used. $\overline{T}(s_t)$ is the average temperature of all state-action values. ξ is used to control the pace in which agents transition from explorers to exploiters. So, in short, the agent selects action $a = argmax_a Q(s,a)$ with probability $1-\overline{T_t}(s_t)^\xi$ and a random action with probability $\overline{T_t}(s_t)^\xi$.

An overview of the Lenient-DQN architecture can be seen in figure 4-1

Palmer et al. showed that the Lenient-DQN outperformed other similar technique networks on a complex fully cooperative task. They also showed that the auto encoder worked for generalization of the state-space.

After a rather long detour, all elements are finally in place to finalize the concept of Weighted Double Deep Q-networks. WDDQN combines the use of leniency with that of double Q-learning and normal Q-learning, extended into the deep learning domain. WDDQN solves the under- and overestimation problem of Q-learning, while using leniency to solve certain MARL induced problems. Leniency also comes with the added benefit of increasing the

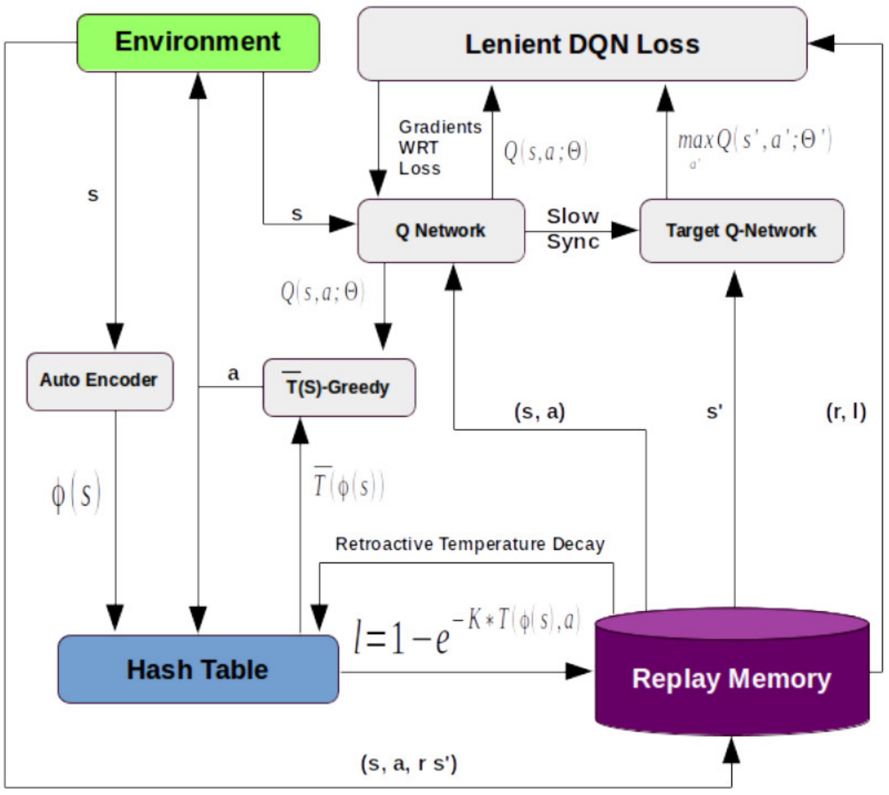


Figure 4-1: An overview of the full Lenient-DQN architecture including all auxiliary mechanics.[26]

exploratory behaviour, because the temperature can be seen as an indication for the frequency of visitation. Zheng et al. showed that WDDQN outperformed all other "lesser" variants that are mentioned above, on a standard test environment.

4-1-2 Policy gradient methods and other approaches

In this section a mix of various different approaches to solving the MARL problem are discussed. The previous section focused mainly on how to use value functions in a MARL environment, while this section will take a look into direct parameterization of the policy space. However, next to the use of policy gradients, there are many different ways to work around the MARL issues. For example, *centralized learning and decentralized execution*, which learns a global policy for all agents, but executes action based on local observation. Or methods that handle the MARL issues by using different forms of machine learning techniques, such as *Recurrent Neural Networks* or *Communication Neural Networks*. The reason for the coarse division between on on hand value based and the other policy gradients and other techniques is that value based approaches are applicable to other methods that are policy based, through the use of actor-critic methods. This section will give general insight in various potential useful techniques that approach the MARL problem from multiple angles.

Coarse comparison of various policy based approaches to MARL

Gupta et al. [28] compared three different RL training schemes for a MARL domain. First, a centralized learning approach was considered. A centralized approach means that the observations and actions are all collected in a single joint observation and joint action respectively. The downside to this approach is that the observation and action spaces grow exponentially with the increase of the amount of agents. Gupta et al. addressed this by factoring the joint action space to collection of individual actions of each agent. This allows the mapping from observations to action be executed by a number of sub-policies that each map the joint observation space to a single agents action. The disadvantage is that the observation space is still growing exponentially.

Another training scheme is that of concurrent learning. In concurrent learning each agent learns its own individual policy based on its own local observations. This is advantageous when there are different types of agents for different tasks. However, the downside is is that the sample efficiency is drastically reduced because experience is not shared among agents. Also, the change in policy for each agent causes non-stationarity. Concurrent training could be augmented with a form of communication, which works around this non-stationarity.

Finally parameter sharing is considered. When all agents are homogeneous the policy can be shared between those agents. The policy gets trained by the collective experience of all agents. Each agent can still exhibit different behaviour, due to the fact that the local observations for each agent are used, as well as their index, during control.

These training schemes where applied by using three different RL algorithms, namely *Deep Deterministic Policy Gradient (DDPG)*, *Deep Q-Networks (DQN)* and *Policy Sharing Trust Region Policy Optimization (PS-TRPO)*. These where then applied to a few scenarios, with both discrete and continious state and action space representations. Also on the side of function approximation, three different neural network architectures where compared. The standard feed-forward MLP, a recurrent neural network using GRU units and a convolutional neural network (CNN).

The first experiment was one in a discrete setting that had multiple agents catch a ”thief”. This experiment requires coordination between agents due to the fact that the ”thief” has to boxed in. The optimization method used was PS-TRPO, while a comparison was made between the different network architectures and training schemes. Gupta et al. showed that the decentralized and concurrent approach outperformed the centralized approach, while using a MLP as an function approximator. Combining the GRU network with decentralized approach resulted in the highest return. The reason why a recurrent net such as GRU outperforms a feed-forward network such as a MLP is that the observations are correlated in time. Having an function approximator that can capture these correlations improves the performance. This notion was found among all other experiments where this correlation is present. An comparison was also made between PS-DQN/DDPG an PS-TRPO, where PS-TRPO outperformed the other methods on all experiments. The next experiment was in the continious state domain. Again the centralized method performed significantly worse than the decentralized and concurrent approaches. MLP performed better than the GRU network, which was credited to the fact that a GRU network is difficult to train. Still PS-TRPO outperformed the other methods. The final experiment on another domain showed similar results. Another results was that for all experiments, local rewards showed better performance than global rewards. Finally it was shown that when the number of agents increased, the performance degraded significantly. However, Gupta et al. used a curriculum for training which showed significant performance improvement. The curriculum was setup in a way that the environment that was trained on could include any number of agents. First, a distribution was sampled that determined the amount of agents. Then a environment was generated containing this amount of agents and trained on for enough iterations to reach a certain return threshold. This was done continuously till a certain treshold was reached for all environments.

Counterfactual Multi-Agent Policy Gradients

Counterfactual Multi-Agent Policy Gradients by Foerster et al. [29] is an approach to multi agent learning that addresses several MARL problems. These problems are based on an actor-critic implementation of independent Q-learning, called *Independent actor-critic (IAC)*. To recap, independent Q-learning is a multi agent implementation of Q-learning in which each agent learns its own value function and policy. Extend this to independent actor-critic the difference is also learning an individual parameterized policy. Every agent learns on its own local observations, which results in the first problem with IAC. There is no centralized learning, which makes cooperation difficult. Secondly, there is the problem of multi-agent credit assignment discussed in section 4-4.

IAC fails to exploit the fact that learning is centralised. COMA overcomes this limitation. Firstly, COMA instead conditions its critic on the global state s opposed to that of only local agent observations, u^a. Assume the case that based on this critic the policy gradient is calculated by calculating the TD error of this critic, as seen in equation 4-15.

$$g = \nabla_{\theta_{pi}} \log \pi(u|\tau_t^a)(r + \gamma V(s_{t+1}) - V(s_t))g = \nabla_{\theta_{pi}} \log \pi(u|\tau_t^a)A(s_t, a_t) \tag{4-15}$$

However, this approach does not address the multi-agent credit assignment problem. To solve this, COMA introduces a *counterfactual baseline* which is inspired by *difference rewards* [30]. As seen in section 2-2, using a baseline for policy gradient methods reduces its variance. A widely used method for a baseline is that of the advantage function. In difference rewards the agents learns from a shaped reward given by equation 4-16, where $\mathbf{u}$ is the joint action and c_a a default action.

$$D^a = r(s, \mathbf{u}) - r(s, (\mathbf{u}^{-a}, c_a)) \tag{4-16}$$

The idea behind difference rewards is to figure out an agents individual contribution towards a certain global reward. The difference is taken between the global reward, and the reward that would be received when that particular agent would have no influence on the reward. This is what $r(s, (\mathbf{u}^{-a}, c_a)$ depictates, the action of an agent is replaced by a default action to be able to measure this difference. However, to determine the difference reward for each agent a full simulation cycle has to be executed per agent to determine the impact of each individual agents default action. Also deciding on what a default action is can be cumbersome.

Foerster et al. solved this by using approximation techniques that allow the determination of the difference reward in a single pass for each agent. This is done by cleverly utilizing the underlying neural network for the critic.
For each agent an advantage function is computed, $A^a(s, \mathbf{u})$, which calculates the advantage value that compares the Q-value for the current action u_a to a counterfactual baseline that marginalises u^a out. This can be seen in equation 4-17.

$$A^a(s, \mathbf{u}) = Q(s, \mathbf{u}) - \sum_{u'^a} \pi^a(u'^a|\tau^a) Q(s, (\mathbf{u}^{-a}, u'^a)) \tag{4-17}$$

Note that τ^a are the local observations of the agent a. This is not an exact copy of the process described earlier, where the exact reward contribution is determined by using a default action. In this case the default action is the average action value of all possible actions by agent a, while keeping the other agents actions, $\mathbf{u}^{-a}$, fixed. In this case, the last part of equation 4-17 is considered the baseline. Figure 4-2 shows an overview of the architectural setup of COMA.

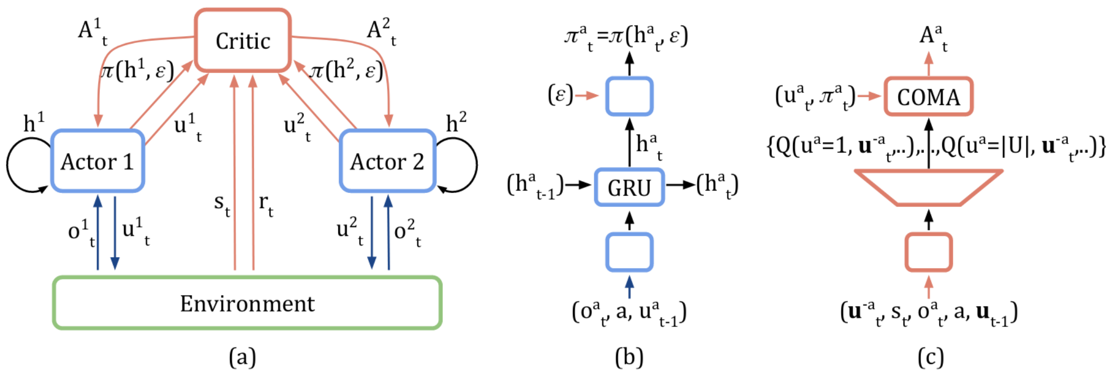


Figure 4-2: In (a), information flow between the decentralised actors, the environment and the centralised critic in COMA. In (b) and (c), architectures of the actor and critic. [19]

As can be seen in figure 4-2 the critic network architecture is set up in a way that as input, it receives all actions of the other agents and as output the Q-value for each available action for a single agent. This allows for calculation of the counterfactual baseline in a single pass, reducing computational overhead. Note that only during learning this exchange of information is done between the critic and actors, when executing the policies no communication is done. Foerster et al. showed that COMA improved learning and maximum reward compared to simpler but widely used methods, such as IQL, on a difficult test bed. COMA is particularly interesting in that it shows a potential way of dealing with the credit assignment problem.

Multi-Agent Actor-Critic for Mixed Cooperative-competitive Environments

Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments by Lowe et al. [31] is a commonly mentioned paper that extends the method of DDPG, *Deep Determinisitic Policy Gradients*, to that of the multi-agent domain. In contrast to the COMA method, MADDPG learns a separate critic for each agent, however learning is centralized. MADDPG build on the single agent algorithm DDPG, which is a policy gradient method that uses deterministic policies in contrast to stochastic policies. The centralized critic is defined as follows;

$Q_i^\pi(\mathbf{x}, a_1, ..., a_n)$, where $\mathbf{x}$ is the global state information and a_n the actions of each other agent. Note, that by incorporating the actions of each agent into the critic, the environment is made (sort of) stationary from the point of view of the critic for each agent. DDGP uses an experience replay buffer which is used to sample trajectories. Note that for updating both the critic and actor, a target network is used. This is generally an older state of the neural network, to ease convergence. Figure 4-3 gives an overview of MADDPG.

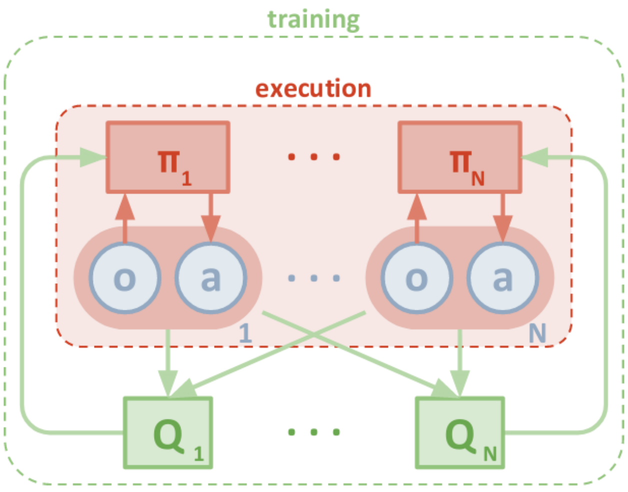


Figure 4-3: Overview of MADDPG, decentralized actor centralized critic approach. [31]

A danger, especially with competitive environments, is that of over fitting with respect to another agents policy. The policy of an agent that is overfitting on the behaviour of another agent can suddenly collapse when the other agent changes policy drastically. Lowe et al. tackled this by learning an ensemble of k multiple sub-policies, where each policy is randomly selected for each agent to execute, and subsequently updated. This increases the computational overhead significantly, however spreads out learning more gradually by using these sub policies. Each policy has its own replay buffer.

Lowe et al. demonstrated that MADDPG showed significant performance increase compared to single agent methods applied to various MARL problems. However, no comparison was made between different MARL approaches.

4-2 How to define the state-space

In this section the representation of the state space is discussed. This can be split up roughly in two parts. A description is needed of what state information is required to define the Markov state. Secondly, a more technical description of how to present state information to the deep learning algorithm. Note that this section will be used to try to paint a picture of what thought is required to tackle this problem, in contrast to giving a full solution. The solution to the problem is highly dependant on the final choice of algorithm and general approach.

4-2-1 The Markov state

As a baseline for the required state information a look is given to current conflict detection and resolution methods, such as *modified voltage potential* [32] or *4DTBO* [33]. While both methods differ in approach, they use the same set of state information or a close variation. Note however that depending on the reinforcement learning algorithm applied, the representation of the state information also varies. When using an centralized approach, i.e. an all seeing eye, the state information is different than that from a decentralized local perspective. Nonetheless enough information has to be provided to paint a clear picture for all variants in algorithm.

First off, the velocity vector is required. This consists of the heading and ground speed. However, due to the fact that action space will consist of heading changes only, and velocity will stay the same across all agents, velocity can be omitted. Secondly the location of each agent has to be known. There are two options; absolute cartesian coordinates and heading, or relative distance and heading. A cartesian system makes more sense in a centralized approach, as this will provide all information including relative distances inherently. The latter makes more sense in a decentralized approach. Knowing the distance and heading information of neighbouring aircraft is a more concise representation compared to a set of grid coordinates and heading. This is also seen in the conflict detection and resolution method of modified voltage potential and 4DTBO. Basically all information that is required are velocity and heading information, as well as time. Time is used to make predictions about the aircraft's future position to determine whether or whether not the plane is on a course that results in a conflict. The addition of this closest point of approach could be added to the state information of each agent. Time is somewhat incorporated in the algorithm by the step size of the simulator, and having an absolute time will not have any influence due to the calculation being in a relative time frame. However, depending on the goal formulation, absolute time can be a needed feature. Such as being done in 4DTBO where a time of arrival is added to the goal in combination with the location of the to be landed airport. Another important piece of information is the distance to the goal, i.e. airport. This can be described in the same fashion as stated above, depending on the scenario. Note that in the case of relative distances and heading the agent does not need any information about its own location, as this is determined relative to all other instances, such as other agents and goal.

4-2-2 How to present the state information

All RL algorithms use neural networks as the technique for function approximation. There is a lot variability in how the neural networks are setup, varying from a multilayered perceptron to recurrent neural networks. To start there are a few subjects to take into account when presenting the state information. First, there is a continuity in time. Current states are highly correlated with past states. This information of correlation is important to provide to the algorithm for stabilization and generalization when training the neural networks, as well as insight in the influence of the control action on the dynamics of the system. [28, 29] However, this information can also lead to stability issues when there is to much correlation, as seen with for example Deep Q-networks [5]. There are multiple ways of incorporating this time continuity. Providing the state information as a set of trajectories gives insight in the continuity of states, while an MLP would be able to capture this spatial relation.

When representing the state at each time point individually, the network is not trained on this continuity. This can be resolved by using a form recurrent neural networks. These networks ”remember” past states for a set amount of time allowing the network to pick up on these continuities. A commonly used method in reinforcement learning is that when a display output is available, that instead of creating feature vectors of state information a convolutional neural network is used directly on the display output. Using convolutional neural networks to interpret the state information the network can adapt itself to extract as much information as possible, in contrast to directly providing the state information as a feature vector on which information is extracted.

Secondly, a common problem with deep learning is that of regularization, as discussed in section 2-4. There are various methods that focus on the network itself to provide regularization, but regularization is also increased by how information is fed to the network. An method to improve regularization would be to present the algorithm with a dynamic scenario, varying for example the starting position and the goal state. As seen in section 4-1-2, varying the number of agents in a form of an curriculum also assists in the convergence and regularization. Varying starting states also ties into the exploration dilemma found often in reinforcement learning problems.

The choice of neural network and way of presenting the state information is highly dependant on the final selection of approach and algorithm. Gupta et al. [28] did a short survey between the use of RNN and MLP's. They concluded that in general the RNN showed better result, due to the aforementioned ability to capture spatial relations.

4-3 How to define the action space

In this section the action space definition is given, both from a representational and implementation perspective. First the representational part will be discussed, which will cover how the action space should be represented. In the implementation part the perspective will be more from a reinforcement learning point of view.

4-3-1 Practical definition of the action space

As discussed in section 4-2 the actions of each agent are limited to heading changes only, in comparison to also using velocity and height changes. However, on the matter of how an heading change should be performed there a few options are available. First and the most intuitive method is allowing each agent to directly select its new absolute heading as an action. The action space subset would then consists of $a \in A$ where $A = [0, 359]$. However, the problem with this definition is the presence of an continuity at 0 and 360 degrees, which could be difficult for a RL algorithm to be trained on. Another way of describing the action space would be to use a relative heading change. Instead of absolute changes, the actions would act more as a ”steering wheel” with relative control commands to its own heading. The action space is then defined as; $a \in A$ where $A = [-90, 90]$. The strength of relative action commands is discussed in *Learning Dexterous In-Hand Manipulation* [34], where relative action command where used for the continuous control task of a robot arm. Relative control reduces the action space while also denying the circular behaviour in the action space. There are more exotic methods of action space implementation, such as using changable waypoint locations as actions or presenting a waypoint location inside a certain precalculated solution space as an action. However, these options would be considered when the more "free" choices do not work satisfactory.

4-3-2 How to implement the action space

The action space, or better said the policy, can take on several forms. First off, the decision between a continuous or discrete action space representation has to be chosen. The difference between them is that a discrete representation has a fixed number of discrete actions in contrast to a continuous action space, which samples from a distribution. A continuous action space representation is done by learning two defining distribution variables, μ and σ, the mean and variance respectively. The action is then sampled from a normal distribution constructed with these two parameters. This also ties in to either a stochastic or deterministic policy. A stochastic policy returns a chance of selecting a certain action, such as what is done by a normal distribution. A deterministic policy maps each state to a specific action, without any probabilistic element. A stochastic policy would turn into a deterministic policy when its variance, σ, is reduced to 0.

Depending on the definition of the Markov problem, such as a centralized or decentralized approach, the action space is represented differently. For example, when using a centralized approach a single policy will represent all actions for each agent in a single pass. There will be the same amount of end nodes for each agent, returning an action for each. However, when a more decentralized approach is done a single output will be sufficient, however multiple passes over the policy are required to determine actions for each agent individually.

Difficulties with how to represent the action space lies in what is needed from the environment as well as what is computationally feasible. When discretizing the current ATC environment, a bin size has to be selected. This can be done based on the required accuracy from the environment. Another factor to take into account is to avoid the *curse of dimensionality.* An increase in amount of different action to take when discretizing increases the amount of information needed about each action, especially when in multi-agent environments.

Tang et al. [35] investigated the use of discretization of an continuous action space. An advantage of this discretization is that a discrete distribution can be more expressive. The reason is that probabilities of action can be divided over the whole action space, in contrast to an Gaussian distribution where there is a single "hill" that covers a certain part of the action space. This distinction between distribution types is called *multi-modal* and *unimodal* respectively. The downside of discretization is that when having a to coarse discretization the policy does not have enough capacity to achieve good performance. Another issue with discretization of an continuous action space is the loss of information that was given by the ordering of the action space. However, based on the *stick-breaking parameterization [36]*, Tang et al. proposed to incorporate this ordering by modifying the output of the policy. The idea behind this principle is to slightly distribute the probability of a high probability action to neighboring discrete action bins. Tang et al. showed that discretization of the action space while keeping the ordering, improved learning on most multi-action continuous control problems. For the ATC problem however there is a relatively simple continuous action space available. Nevertheless, it provides an argument for discretization of the action space and a solution for the unimodal nature of Gaussian policies.

Another method of creating a more expressive policy space while remaining in the continuous policy space is the use of *normalizing flow (NF)*, also done by Tang et al. [37]. They applied the principle of normalizing flows as a policy to TRPO to enable multi-modal policies as well as better exploratory behaviour. First off, the concept of normalizing flows is introduced. The idea behind normalizing flows is to create a sequence of invertible non-linear transformations,g_θ, that transform a source noise, ϵ, to a certain output distribution, x. Equation 4-18 shows this process.

$$x = g_{\theta_K} \circ g_{\theta_{K-1}} \circ ... \circ g_{\theta_2} \circ g_{\theta_1}(\epsilon) \tag{4-18}$$

As the technique behind this is quite extensive, the general concept is presented. The sequenced transformations transform a source to any probability distribution as output. This is done by the fact that these transformation layers are trainable, such as a neural network. This allows for highly expressive probability density functions, which would function as a policy representation. An overview for this process can be seen in figure 4-4

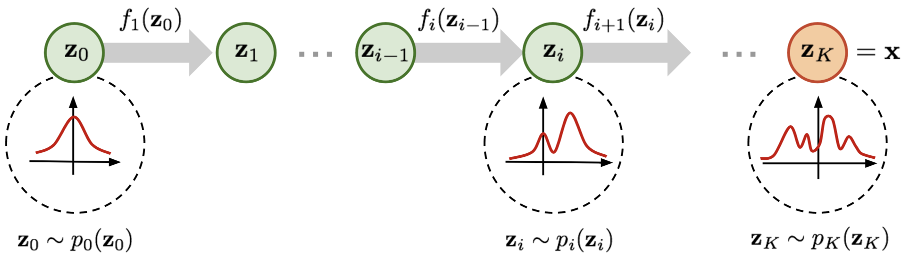

$f_1(\mathbf{z}_0)$
$\mathbf{z}_0$
$\mathbf{z}_1$
$\mathbf{z}_{i-1}$
$f_i(\mathbf{z}_{i-1})$
$\mathbf{z}_i$
$f_{i+1}(\mathbf{z}_i)$
$\mathbf{z}_K = \mathbf{x}$
$\mathbf{z}_0 \sim p_0(\mathbf{z}_0)$
$\mathbf{z}_i \sim p_i(\mathbf{z}_i)$
$\mathbf{z}_K \sim p_K(\mathbf{z}_K)$

Figure 4-4: The process of Normalizing Flows [38]

Tang et al. investigated the performance of normalizing flow when combining it with TRPO. As TRPO updates restrict the update step for the policy, having a more expressive policy space the convergence rate is improved. Tang et al. examined the difference between the expressiveness of a normal Gaussian policy versus a NF policy. They showed that Gaussian policy distributions tend to get stuck on local optima, due to the step size restriction of TRPO in combination with the fact that Gaussians are less expressive. NF increased expressiveness, which in turn allowed for better exploratory behaviour due to the fact that samples with a much higher variance could be achieved. They showed that on nearly all continuous control benchmarks performance was improved, especially on problems that are ambiguous on its control actions. This means action spaces where certain actions are not implicitly tied to a certain state, but for example both left and right are actions of equal probability. A game such as rock, paper and scissors would be a good example.

4-4 How to define the reward structure

Differentiating various reward contributions to an agent is one of the *hurdles of reinforcement learning*. In one case, the reward can be sparse and delayed, increasing the difficulty for the

RL agent to determine what actions let to this reward, if getting any reward at all. However, using sparse reward does put emphasis on "exotic" behaviour as the agent is completely left to his own devices. An example of a sparse reward would be a reward of 0 at all times, while giving a reward of 1 at the goal state.

The reward should be tailored towards the envisioned goal. For example, when it is paramount for the system to achieve globally optimal solution, each agent should receive the globally achieved amount of reward. In contrast to this, each agent could also receive locally obtained rewards. However, this tends to gravitate towards selfish behaviour. The global reward could be an accumulation of all the locally achieved rewards. This would also assist in determining what agent contributed what reward, due to the fact that the global reward is build up of these individual contributions. The use of creating global rewards solely from local observations has to be investigated empirically.

Another issue with rewards, especially in the multi agent case, is that of credit assignment. Credit assignment embodies the difficulty to differentiate what agent contributed to the global reward. On this subject was touched in section 4-1-2, by COMA, which used *difference rewards*. Difference rewards are a method of determining how much a single agent contributed towards the total receive reward each step, and uses that to correctly reinforce the agents behaviour. Equation 4-19 gives the definition of difference rewards.

$$D_i(z) = G(z) - G(z_{-i}) \tag{4-19}$$

$D_i(z)$ is the difference reward of agent i depending on state z, which can be either the state or state-action pair. $G(z)$ depicts the global reward. $D_i(z)$ can either be directly calculated or estimated using various methods. For example, when using cumulative local rewards as a global reward.

Another commonly method that is used to guide the agent to optimal behaviour is the use of *reward shaping*. The idea behind reward shaping is to provide a supplementary reward signal that simplifies learning. This idea is incorporated in *Potential-based reward shaping (PBRS)*, which is defined in equation 4-20.

$$PBRS = r + \gamma\Phi(s') - \Phi(s) \tag{4-20}$$

r is the original reward received, γ the same discount factor while Φ is a potential function that is defined over a source s and a destination state s'.

Tumer et al. [30] investigated the use of both reward shaping and difference rewards on multi-agent systems. Reward shaping is a delicate matter, and cannot naively be applied. Tumer et al. proved that when correctly using PBRS the underlying optimal solution to the MDP does not change. They united both techniques into a single framework, as shown in equation 4-21.

$$\begin{aligned} r_{shaped} &= r(s,a,s') + F(s,s') \\ r(s,a,s') &= G(s,a,s') \\ F(s,s') &= \gamma\Phi(s') - \Phi(s) \end{aligned} \tag{4-21}$$

Where $G(s, a, s')$ is the global received reward, and $F(s, s')$ the reward shaping function. The used values in equation 4-21 are typically used, however this can be anything. For example, the received rewards $r(s, a, s')$ could also be a locally received reward, $L_i(s, a, s')$. Note that s and s' are in terms of state transitions, meaning that $F(s, s')$ is the discounted difference in shaped reward due to the transition.

Based on this framework *Difference Rewards incorporating Potential-Based Reward Shaping (DRiP)* is introduced, as seen in equation 4-22.

$$\begin{aligned} r_{shaped} &= r(s, a, s') + F(s, s') \\ r(s, a, s') &= G(s, a, s') \\ F(s, s') &= -G(s'_{-i}) + \gamma\Phi(s') - \Phi(s) \end{aligned} \tag{4-22}$$

Where $-G(s'_{-i})$ is the difference reward, and $\Phi(s)$ a potential function that incorporates knowledge. The potential function can be used to incorporate environmental knowledge. Tumer et al. showed that this drastically increased the learning in a discrete MARL environment. The extension to a continuous domain would be possible, as well as using the framework to include various other reward signals. The biggest take-away of this theory is that if reward shaping is desired, it should be a relative value, instead of an continuous absolute addition.

As seen in section 4-1-2, multiple methods where shown that also manipulate the reward structure to its benefits.

Chapter 5

Simulated Air Traffic Control setting

In this section, all three sub-questions are answered jointly; *How to simulate the Air Traffic Control setting?*, *How to setup the experiment?* and *How to maximize data throughput?*. Albeit each sub-question having its own significance and therefore reason for mention, they are closely related to each other. Choices made for one sub-question, directly weight on the choices for the other.

As discussed in the example at the beginning of the document, the actual experiment setup will be quite similar. First off, BlueSky is used as an environment for simulation. BlueSky offers easy interaction with each agent, as well as all the necessary data streams available for the reinforcement learning. BlueSky also offers an interface, to visualize progress that is made during training[3]. Reinforcement Learning is very sensitive to learning parameters as well as problem definitions. Being able to visualize what the agent is doing, is a good basis for a trail-and error approach in solving the problem.
There are multiple different machine learning libraries available that are performance optimized and ease the strain of implementing the various different algorithms. For reinforcement learning various platforms are available, such as OpenAI baselines, RLLIB and keras-RL. For this experiment, RLLIb is used. RLLib offers good performance as well as one of the few libraries that is build around multi-agent reinforcement learning problems. Deep learning made serious leaps through the past few years, both in facilitating ease of implementation as well as computational efficiency. While reinforcement learning hinges on these improvements, there is a general lack in systems that specifically target reinforcement learning. RLLib fills this gap by creating a from the ground up reinforcement learning implementation that enables centralized program control and parallelism encapsulation[39]. RLlib achieves this through the use of Ray, general-purpose cluster-computing framework that enables simulation, training, and serving for RL applications[40]. Ray offers the ability to parallelise various calculations, as well as asynchronous updates invoking both the CPU and GPU.

The importance of computing performance cannot be understated. Success stories about reinforcement learning applications such as Alpha Zero and OpenAI five required lots of computational power. It took Alpha Zero, a single agent discrete state game, 13 days of training for the game Go while utilizing 5000 first-generation TPU's and 16 second generation TPU's [1]. OpenAI five, a multi-agent reinforcement learning problem with continuous state and action space, required 128000 CPU cores and 256 P100 GPU's for training, generating around 180 years of experience each day[2]. In contrast to the well known Moore's law, which states that the transistor density of electronic hardware has a 18 month doubling period, a 3.5 month doubling time in required computing power for machine learning is observed [41]. This can be seen in figure 5-1. Luckily, this research problem has a considerably lower requirement when looking at computing requirements. However it is still relevant due to the fact that huge amounts of computing power are not available.

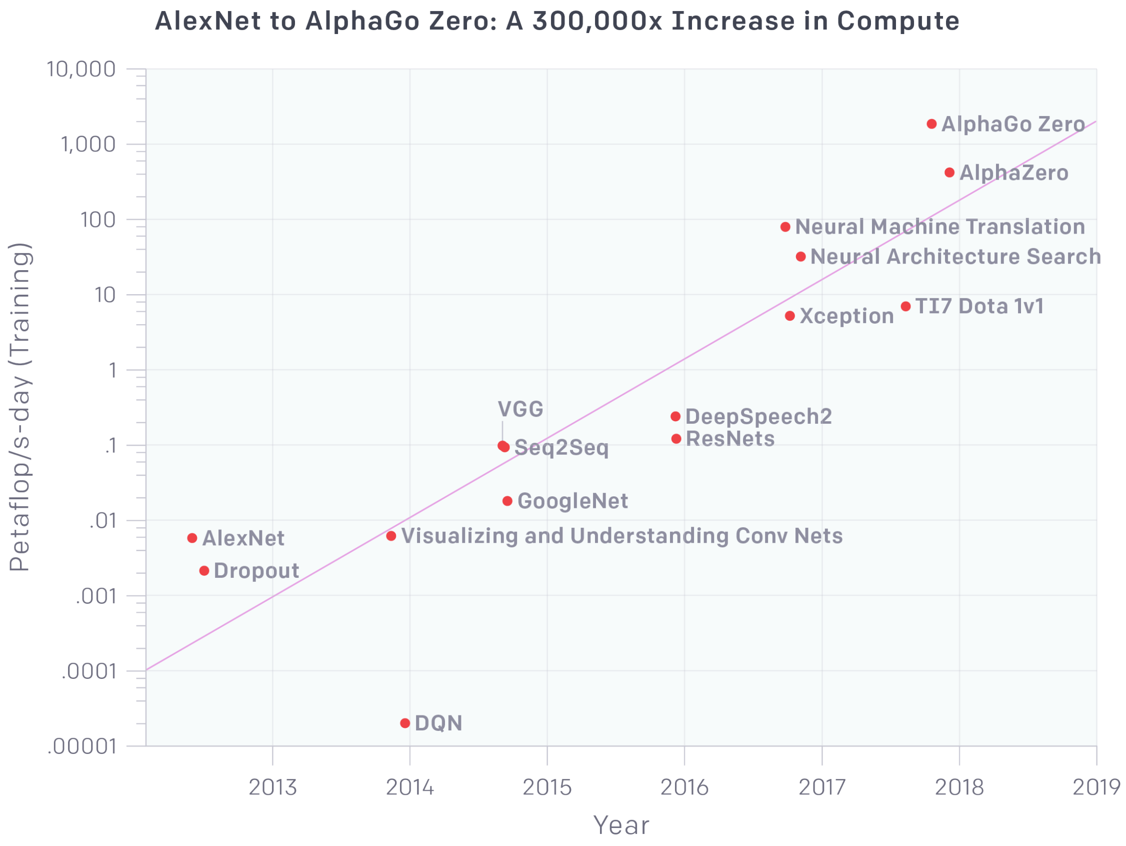


Figure 5-1: Trend seen in the increase of computational requirements for state of the art machine learning [41]

Chapter 6

Maximizing safety and efficiency

In this relatively short chapter a few key points are discussed regarding the maximization of safety and efficiency. First, an method of measuring the overall practical performance of the algorithm is established. This is done by comparison of the algorithm versus a baseline solution. Secondly the performance of the algorithm can be investigated further with more reinforcement learning specific method. Finally, a short explanation is given about safety.

The goal, as described in the introduction, consists of creating an algorithm that is able to guide aircraft efficiently to their destinations, while providing safe merging and spacing. A well performing conflict resolution method currently available is called *Modified Voltage Potential (MVP)*[42]. The governing principles of MVP are the repulsive properties of similarly charged particles. Simulating aircraft as these similar charge particles results in a "in-between" push between aircraft, due to repelling forces. This repelling force is centered around the *closest points of approach*, ensuring that a loss of separation is avoided. This push then translates to certain required displacement, which is used as a control input for both aircraft that are in conflict. Figure 6-1 gives an overview of this process.

The MVP can be used as a baseline solution in conflict resolution, measuring efficiency as the amount of losses of separation and total travel time during conflict resolution. A few other measures of efficiency can also be established. The amount of heading changes for the aircraft trajectory from begin point to destination are going to be compared. Also total trajectory time is a measure of efficient flight.

Collecting these dependent variables and comparing the MVP and autopilot performance as baseline versus the algorithm's performance can be used as validation and efficiency comparison.

An obvious reinforcement learning specific performance measure would be the total accumulated reward during a single episode. This information can be collected throughout learning, to give an indication if the algorithm is converging. Most reinforcement learning papers also use this as their foremost benchmarking tool. Another method of determining performance would be to check the weight distibution of the underlying neural network. This could aid in determining if there is any overfitting present.

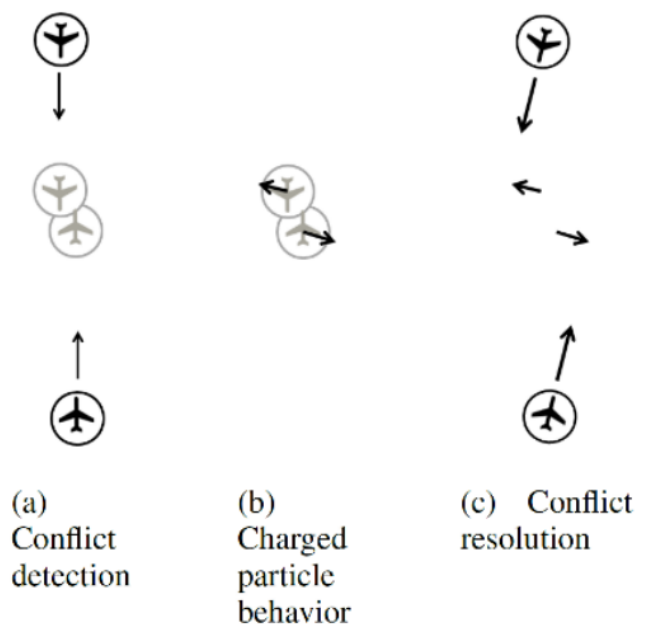


Figure 6-1: Workings of the Modified Voltage Potential. Source:[42]

As for safety, this can only be steered by using the proper reinforcement learning tools. Ensuring safety is mostly done through the use of a proper reward structure. One that penalizes conflicts, but awards efficiency. If safety is lacking, either the rewards have to be adjusted or there is not enough state information present to provide solutions.

Chapter 7

Experimental setup

In this chapter the experimental setup is presented. The experimental setup will go into more detail of the actual implementation in Python, as well as expand on the requirements of the setup.

This section will expand on the practical implementation, which can be split up in roughly three parts; the BlueSky side, RLlib side and Tune. An overview can be found in figure 7-1. In this section both sides of the experiment will be expanded upon.

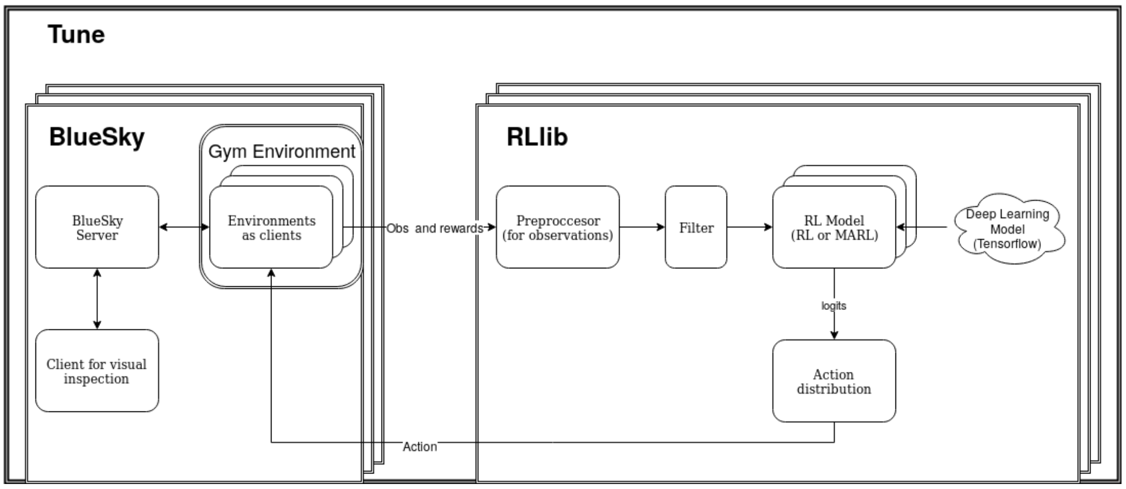


Figure 7-1: Overview of data flows for the experimental setup

First, the RLlib side. A few requirements are set which are; computational efficiency and extensibility. First off, computationally efficiency is achieved by RLlib in twofold; the highly integrated use of optimized deep learning libraries, such as Tensorflow, and the high throughput communication channels with BlueSky. RLlib allows for policy or value function evaluations to be done by *Tensorflow*, which is a GPU optimized library as well as allowing for ease of implementation. RLlib offers a "preprocessor" in its pipeline. This can be used to transform observations and rewards to their required format. The "filter" section is implemented to stabilize the information stream in such way that deep learning methods converge with more ease. This mostly contains normalization filters to prevent exploding gradients withing any neural network based function approximation.

Extensibility is given in the form of the section "RL model", the selected "deep learning method" and "action distribution". These are all external modifiable section, which allow for customization of various aspects of the required RL algorithms. For the RL model, this varies to all the previously mentioned methods. Also either a MARL or normal single agent solution can be implemented. With MARL, depending on the definitions, multiple policies can be trained at once. The deep learning methods can be varied to all different available techniques. The action distribution is a transformation of the model output to actions useable by the environment, as well as an intermediate step for deep learning methods calculate their policy losses.

The general idea behind the RLlib implementation is to provide a basis that is easy customizable with various different RL techniques, allowing for ease of experimentation.

On the side of BlueSky, a few different aspects can be differentiated. BlueSky offers the ability to run a server, in which clients can be connected to. For RLlib to gather trajectories and returns, a *Gym* environment is used as a client. Gym was introduced by OpenAI to provide a standardized format for RL problem implementations, to allow for ease of use and extendability to other RL libraries. Gym requires the environment to be able to provide as output the observations and rewards, while accepting an action as input. The BlueSky client functionality is converted to be compatible with the gym format, while remaining compatible with the server/client architecture of BlueSky itself. This allows for proper communication with on one side RLlib, and the other the BlueSky server. By using this server/client interface, visual inspection and manual commands are also allowed by using BlueSky's own observer client system. On top of this, the use of an server/client architecture ties in nicely with the parallelization abilities of RLlib, allowing multiple workers to gather trajectory returns.

As a final addition to the experimental setup, *Tune* [43] can be used for hyperparameter determination as well as properly conducting experiments over various configurations. Depending on the RL technique used, various hyperparameters have to be set before training. Hyperparameters are subjective to the actual problem, and can only be determined based on the problem definition, trail and error and example papers. As an example for hyperparameters, some commonly used are; γ, the discount rate and α, the learning rate. Tune assists in determining the hyperparameters by doing efficient rollouts of the algorithm when varying these hyperparameters. Another usefull property of Tune is to allow multiple configurations to be executed sequentially, while collecting pre-set dependent variables. An example of these measurements are reward per episode, neural network convergence rate, loss at each update step, weight distribution of the used neural networks etc. This on top of the variables discussed in section 6 These measures are usefull in determining the stability and performance of the to be tested RL algorithms.

A final addition is that of *Tensorboard*. Tensorboard allows the user to monitor the performance of the algorithm while training. This is usefull to monitor progress, as it can take long periods of time before an RL algorithm obtains convergence.

Chapter 8

Wrap-up and Planning

This chapter contains a wrap-up with regards to all the gathered information. This section will focus mainly on combining all the aforementioned information, and present it with my opinion on how to proceed. This section also functions as an foundation for the planning, which can be found at the end.

Throughout the document the problems with reinforcement learning and more specific, multi-agent reinforcement learning are viewed from different perspectives. While many sources tackle certain problems from different point of view, there is no unison in what "the" solution would be. This section tries to alleviate that issue and present a top down overview of the different aspects. However, note that not all possibilities to move forward are considered, but more of a general tendency is presented with which to move forward. Still some coarse decisions are made to narrow the area of investigation.

Note figure 8-1, which present a break down of different aspects to the problem. Concatenated arrows represent choices, while spaced out arrows represent a further breakdown. The larger grey blocks are used as a reference for the text. For colours, red indicates a disregarded option. Orange is an disregarded option in a certain layer, but the information is still useful later on. Purple indicates some extra consideration for that respective block.

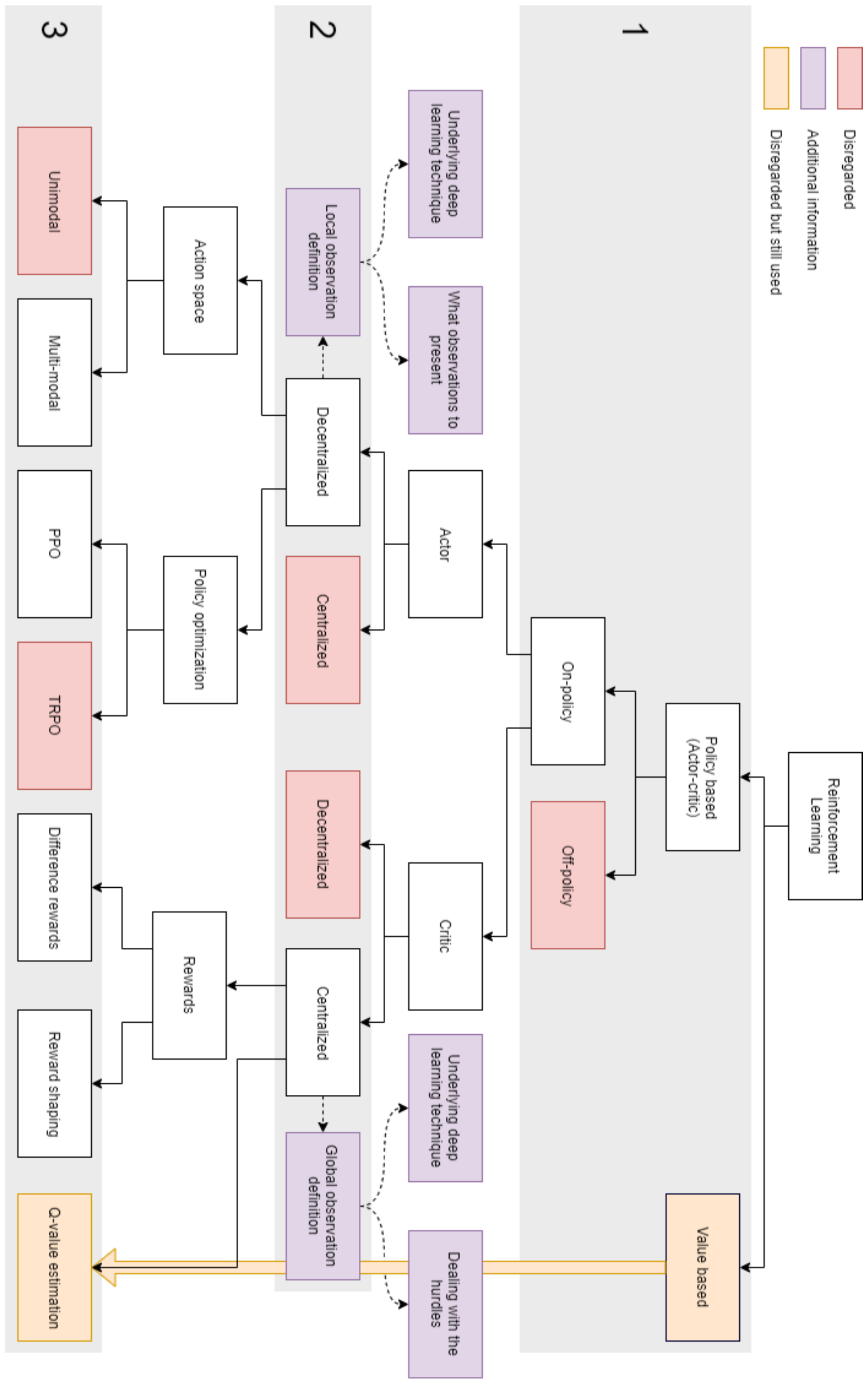


Figure 8-1: Property profile of the diverse library compared to the compound pool.

1:
Policy gradient methods and value function based methods where extensively discussed. This coarse division between two categories of methods was based around the fact that all value methods base their policy directly on the estimated value function, often resulting in a $\epsilon -$ greedy policy. This in contrast to policy gradient methods, which directly parameterize the policy space and use gradient optimization methods to update this policy. These gradients are often calculated with respect to a value function, hence the term actor-critic. Both methods are often extended to the deep learning domain, using some form of gradient optimization. Value based methods however struggle with complex scenarios where the underlying environment is highly non-stationary. Value based methods also do not leverage the additional benefits of policy parameterization, as described in section 2-2. Even though value based functions are more robust when able to determine the optimal value function, in the case of MARL problems the finding of the optimal value function is unlikely. That is why actor-critic methods are the preferred route. Actor-critic allows for keeping certain options open, while also be able to leverage the stability techniques made for value based approaches as seen in sections 4-1-1.

On-policy and off-policy methods both have their advantages and disadvantages. The main advantages of off-policy is the better sample efficiency, because the trajectories are stored and used multiple times to update the algorithm. However, to be able to use off-policy methods properly extra mechanics have to be in place to correct between the difference in policies. This difference increases even more with MARL, due to the non-stationarity of the environment. This renders older trajectories obsolete at a higher pace. This is why an on-policy method is preferred, it eases implementation as well as there are many hurdles to overcome while using off-policy methods. However, if computational strain get to high, off-policy may be required to increase sample efficiency.

2:
Many of the different approaches to the problem considered either a centralized or decentralized approach. Centralized and decentralized imply the use of a critic or actor that has full observability, or is restricted to only local observations. A widely used scenario within MARL is the use of *centralized learning with decentralized execution*. This allows the critic to have full state observability in training, while actions are sampled from local observations only. An example of this is COMA, found in section 4-1-2. An example of decentralized learning and execution would be that of independent Q-learning, found in section 4-1-1. On top of these principles, the method of function approximation can differ as well. For example, as seen in section 4-1-2, a single policy can be trained conditioned on each agents local observations. MADDPG, as seen in section 4-1-2, does this the other way around. It learns a policy for each agent individually, but shares the parameters of the critic, which turns it towards the centralized learning with decentralized execution approach.

All these different shapes of the macro level approach to the problem concern themselves around two points; computational efficiency and application. For the latter, some of the problems that are being solved with reinforcement learning assume that each agent has limited observability. As an example, robots with a set of sensors that have a limited field of view. Luckily this is not the case for the ATC problem, however limiting state information may still be required to lower the computational load.

Regarding computational efficiency, it is generally the case that having full state access for a MARL scenario will pose high computational requirements, as learning a joint action based on a joint observations takes into account all available information. While this is by far the most complete approach, this is generally unfeasible due to the high computational requirements. On top of this not all information is needed to solve the problem from an outside perspective. In the case of the ATC problem it makes sense to have a single policy that is conditioned on local observations by each agent, due to the uniformity of all agent. As an aircraft only has to solve its conflicts locally, where conflicts far away have little on local states. This would require a mix of the principles of policy sharing with decentralized execution.

The local observation definition has a practical limitation, as most deep learning techniques require a set input length. This means that having a dynamic input as local observations is cumbersome. Dynamic inputs would be the case when for example all aircraft in a certain radius will be included in the local observation of each agent. The structure behind recurrent neural networks would allow for dynamic input. The efficiency of this however has to be tested, however for theoretical purposes this is the ideal situation. Static inputs can be made by limiting the amount of state information of other agents as local observation by a set number of maximum agents.

The idea of a centralized critic ties in nicely with the concept of ATC, as well as looking promising due to various other groups applying this with success. Unfortunately the problem regarding computational requirements still persist when given complete state information.

3:
Regarding the action space definition, a multimodal representation offers many benefits over a unimodal representations. Increased tendency for exploration and a more expressive action space are both preferential elements to have. Due to the interaction between multiple agents, the chances of a non converging policy are increased. Expressiveness of the action space assists with the finding of better solutions, as more ground can be covered when selecting actions. A disadvantage is the feasibility of implementation, however the difficulty of this has to be determined later during implementation.

Another aspect of the critic is that it should be able to converge to an solution within this non-stationary environment, as well as deal with the problems described in the "hurdles of reinforcement learning" section 2-3. The application of for example a variant of double Q-learning, as described in section 4-1-1, could aid with the non-stationarity.

In terms of the reward, the credit assignment problem could be alleviated by using difference rewards, as described in section 4-4. Difference rewards allow the critic to be trained by reducing the noise of the received rewards by multiple agents. As for reward shaping, this ties in to the problem of exploration. Reward shaping assists by presenting a constant reward signal, while not altering the underlying optimal solution.

As for the policy optimization method, there are two widely used techniques available. TRPO and PPO, explained in section 2-2-2, are both good candidates. However, as PPO is the easier to implement while also being less stringent on computational requirements makes this the preferred method.

Finally, the hurdle of exploration could be lightened by various techniques. As said, difference rewards will help, as well as a more expressive action space. The concept of leniency, as seen in section 4-1-1, shows promise in mitigating the exploration promise. However, leniency has not been applied extensively towards continuous space domain problems. The use of random starting locations and goals for the agents is an easy but effective way of increasing the rate of exploration, as well as assist in the regularization of the deep learning method.

8-1 Planning

In this section the planning is given, using the wrap-up as foundation. Figure 8-2 gives an overview of the required steps remaining for the experiment.

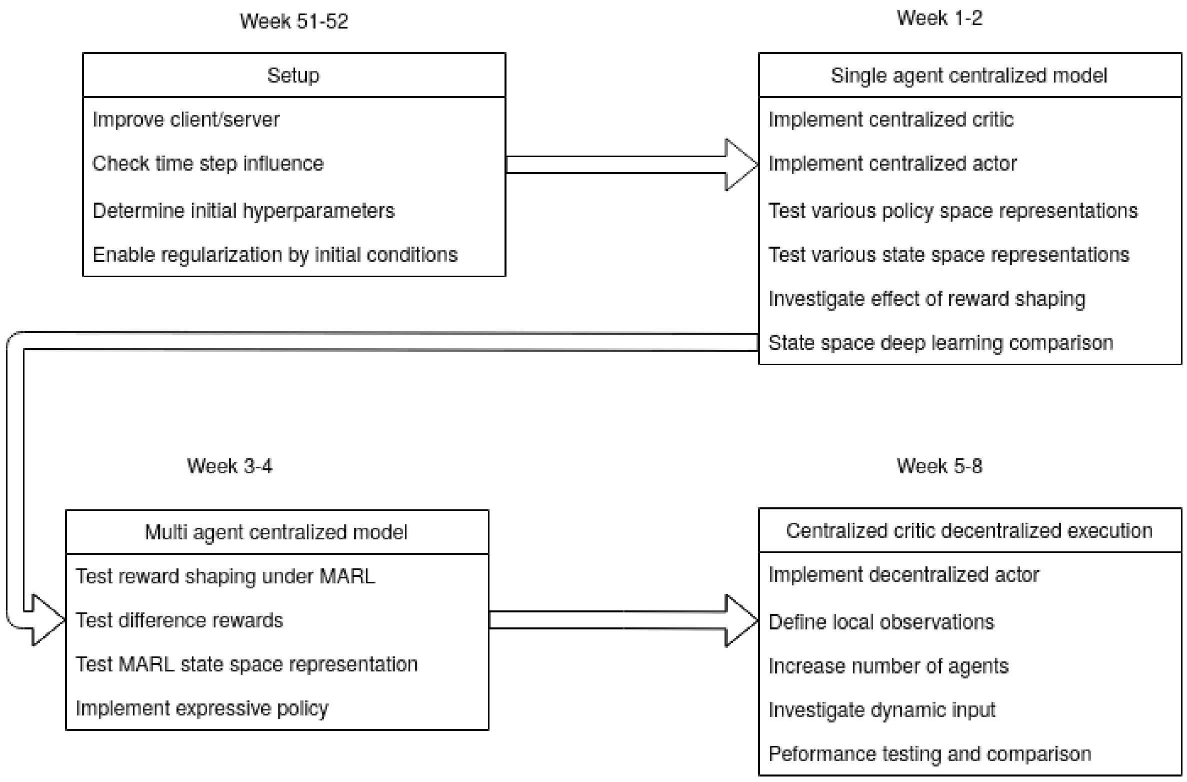

Week 51-52
Setup
Improve client/server
Check time step influence
Determine initial hyperparameters
Enable regularization by initial conditions
Week 1-2
Single agent centralized model
Implement centralized critic
Implement centralized actor
Test various policy space representations
Test various state space representations
Investigate effect of reward shaping
State space deep learning comparison
Week 3-4
Multi agent centralized model
Test reward shaping under MARL
Test difference rewards
Test MARL state space representation
Implement expressive policy
Week 5-8
Centralized critic decentralized execution
Implement decentralized actor
Define local observations
Increase number of agents
Investigate dynamic input
Peformance testing and comparison

Figure 8-2: Planning for the experiment

The first box ”setup” concerns itself mostly with the improvement of the server and client functionality described in chapter 6. A basic implementation is already present, but to leverage the full capability of the architecture a few improvements are still needed. Another important aspect to investigate before beginning with actual testing is the influence of the simulator timestep on certain aspects of the simulation. The timestep will increase the resolution of each simulation step, however this can have negative consequences to the convergence rate of the RL techniques. Also, the timing between the server actions and the output of the RL algorithm has to be in sync. Hyperparameter determination for the various RL algorithms has to be investigated to be sure these have no averse effects on the subsequent step. Finally a system has to be made that allows for dynamic scenarios, to improve regularization. The second step of the planning, a single agent centralized model is used to determine the effects of various different approaches and implementations. The underlying idea of first using a simpler single agent model is that most of these results will be comparable to the results under an MARL environment, however without the extra stability issues MARL invokes on the environment. The results of this section will determine with what techniques to move forward. Then, a multi agent scenario is made and compared to the single agent case. If needed, various extra stability techniques can be implemented. Most notably, an more expressive policy is likely required. Finally, depending on computational constrains and the previous results, a centralized critic and decentralized configuration is implemented. As these effects are difficult to predict, hopefully it will increase capacity for an richer MARL environment, as well as allow for better convergence.

Note that for each stage of the planning, data and results will be stored and the trajectory itself will be kept track of. As the process itself towards an working solution is a result in itself. Also, the configurations of each stage are stored. This allows for experiment validation later on, when doing performance comparison in contrast to MVP.

References

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

[2] OpenAI. Openai five. `https://blog.openai.com/openai-five/`, 2018.

[3] Jacco M Hoekstra and Joost Ellerbroek. BlueSky ATC Simulator Project: an Open Data and Open Source Approach. *7th International Conference on Research in Air Transportation*, pages 1–8, 2016.

[4] Richard S Sutton and Andrew G Barto. Reinforcement learning Complete Draft. 2017.

[5] Martin Riedmiller Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra. Playing Atari with Deep Reinforcement Learning. *IJCAI International Joint Conference on Artificial Intelligence*, 2016-Janua:2315–2321, 2016.

[6] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. *32nd International Conference on Machine Learning, ICML 2015*, 3:1889–1897, 2015.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. pages 1–12, 2017.

[8] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, pages 267–274, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[9] J. Schulman. Advanced policy gradient methods: Natural gradient, trpo, and more. Slides, 26 august, 2017.

[10] Aditya Sharma. Understanding activation functions in deep learning, 2017.

[11] Missing Link AI. Convolutional neural network tutorial: From basic to advanced, 2017.

[12] Prajjwal. Everything you need to know about recurrent neural networks, 2018.

[13] DS Bernstein. Complexity analysis and optimal algorithms for decentralized decision making. *Complete Thesis*, (September), 2005.

[14] Daniel S Bernstein, Shlomo Zilberstein, and Neil Immerman. The Complexity of Decentralized Control of Markov Decision Processes. *Complexity*, 1999.

[15] Guy Shani, Joelle Pineau, and Robert Kaplow. A survey of point-based POMDP solvers. *Autonomous Agents and Multi-Agent Systems*, 27(1):1–51, 2013.

[16] Dipti Srinivasan and Lakhmi C. Jain. *Innovations in Multi-Agent Systems and Applications*. 2001.

[17] Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. Deep Reinforcement Learning for Multi-Agent Systems: A Review of Challenges, Solutions and Applications. (1992):1–27, 2018.

[18] Ming Tan. Game theory and Multi-Agent RL. *Machine Learning Proceedings 1993*, pages 330–337, 1993.

[19] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip H. S. Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning. 2017.

[20] Alvaro Ovalle Castañeda. Thesis: Deep Reinforcement Learning Variants of Multi-Agent Learning Algorithms. 2016.

[21] Sebastian Thrun and Anton Schwartz. Issues in Using Function Approximation for Reinforcement Learning. *Proceedings of the 4th Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, pages 1–9, 1993.

[22] Sherief Abdallah, Shario@ieee Org, Michael Kaisers, and Jan Peters. Addressing Environment Non-Stationarity by Repeating Q-learning Updates *. *Journal of Machine Learning Research*, 17:1–31, 2016.

[23] Chao Yu, Minjie Zhang, Fenghui Ren, and Guozhen Tan. Multiagent learning of coordination in loosely coupled multiagent systems. *IEEE Transactions on Cybernetics*, 45(12):2853–2867, 2015.

[24] Yan Zheng, Zhaopeng Meng, Jianye Hao, and Zongzhang Zhang. Weighted double deep multiagent reinforcement learning in stochastic cooperative environments. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11013 LNAI:421–429, 2018.

[25] Hado Van Hasselt, Arthur Guez, and David Silver. Double DQN.pdf. *Aaai*, pages 2094–2100, 20016.

[26] Gregory Palmer, Karl Tuyls, Daan Bloembergen, and Rahul Savani. Lenient Multi-Agent Deep Reinforcement Learning. (July), 2017.

[27] Zongzhang Zhang, Zhiyuan Pan, and Mykel J. Kochenderfer. Weighted double Q-learning. *IJCAI International Joint Conference on Artificial Intelligence*, (2):3455–3461, 2017.

[28] Jayesh K. Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative Multi-agent Control Using Deep Reinforcement Learning. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10642 LNAI:66–83, 2017.

[29] Jakob N Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, Shimon Whiteson, and United Kingdom. Counterfactual Multi-Agent Policy Gradients. 2007.

[30] Kagan Tumer. Modeling Difference Rewards for Multiagent Learning (Extended Abstract). pages 1397–1398.

[31] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in Neural Information Processing Systems*, 2017-December:6380–6391, 2017.

[32] J. M. Hoekstra, R. N.H.W. Van Gent, and R. C.J. Ruigrok. Designing for safety: The 'free flight' air traffic management concept. *Reliability Engineering and System Safety*, 75(2):215–232, 2002.

[33] Siqi HAO, Shaowu CHENG, and Yaping ZHANG. A multi-aircraft conflict detection and resolution method for 4-dimensional trajectory-based operation. *Chinese Journal of Aeronautics*, 31(7):1579–1593, 2018.

[34] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning Dexterous In-Hand Manipulation. 2018.

[35] Yunhao Tang and Shipra Agrawal. Discretizing Continuous Action Space for On-Policy Optimization. 2019.

[36] Mohammad Emtiyaz Khan, Shakir Mohamed, Benjamin M. Marlin, and Kevin P. Murphy. A stick-breaking likelihood for categorical data analysis with latent Gaussian models. *Journal of Machine Learning Research*, 22:610–618, 2012.

[37] Yunhao Tang and Shipra Agrawal. Boosting Trust Region Policy Optimization by Normalizing Flows Policy. (1), 2018.

[38] Lilian Weng. Flow-based deep generative models, 2018.

[39] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for Distributed Reinforcement Learning. 2017.

[40] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. 2017.

[41] Open Ai. Ai and compute. `https://openai.com/blog/ai-and-compute/`, 2018.

[42] J Maas, E Sunil, J Ellerbroek, and J Hoekstra. The Effect of Swarming on a Voltage Potential-Based Conflict Resolution Algorithm. *Proceedings of the 7th International Conference on Research in Air Transportation*, (June), 2016.

[43] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.