Royal Military Academy
Defence College

Academic year 2019-2020
Formation Candidate Superior Officer
45th session

Developing Strategies with Reinforcement Learning

by Cdt Koen BOECKX, ir

Thesis FCOS Short Version
under supervision of Mr Steven BEECKMAN, ir
Brussels, 2020

Contents

List of Figures

Acknowledgements

While only my name may appear as author of this thesis, it has been, as always, a team effort. First and foremost, I would like to thank my promotor, Mr. Steven BEECKMAN, ir. for taking the time to assist and guide me and for his invaluable suggestions to improve my work. Thank you, Steven.
Secondly, I would like to thank Prof. dr. ir. Xavier NEYT for his guidance and for his help with setting up the servers of the CISS department to improve my workflow. Special thanks also to Mr. Luc BONTEMPS, ir. for taking over (part of) my workload during the first semester when I was attending classes. Xavier and Luc, thank you.
Finally, I would like to thank my family for putting up with me.

1. Introduction

> We humans tend to overestimate AI advances and underestimate the complexity of our own intelligence.
>
> Melanie Mitchell

This thesis was realized as part of the *Intelligent Recognition Information System* (IRIS) project in which the CISS department of the RMA participates. This project, piloted by John Cockerill Defence, aims to develop an integrated software system to assist crews in armoured vehicles with the execution of their mission. To accomplish this, a functional pipeline has been designed:

1. By using the image and thermal sensors on board of the vehicle, automatically detect and recognize objects in the terrain. This results in a collection of detected and classified objects, along with their position in the terrain and the estimated distance from the observer.

2. Identify potential treats among the recognized objects and classify them according to their threat level and the danger they pose. This results in a *situational map* that represents what is known about the battlefield.

3. Based on this situational map, develop a strategy (a sequence of actions to take) to counteract the threat(s) and propose these actions to the crew.

The ultimate goal of the last step is, when confronted with a new situation - both new threats and/or an unknown terrain - to offer advice to the vehicle crew on how to proceed. This advice can take the form of discrete actions (`engage`, `move`, ...) or can be a sequence of actions to perform with a certain goal in mind. All this has to be done in real time, by either generalizing from situations that have been studied before or by recomputing on the spot a new or an adapted strategy. This thesis explores potential solutions to this problem.
The goal of this work is twofold. Firstly, an algorithmic model of a battlefield is developed. Both the actors on the battlefield like the own and enemy troops and the terrain itself will be modeled. The terrain will contain obstacles that can restrict movement and visibility. As all models, this one will be a large simplification of reality. The goal is to create something that is both useful and exploitable; useful because it captures vital elements of reality; exploitable because it is simple enough to feed it to some state-of-the-art algorithm in the domain of multi-agent learning and expect to obtain reasonable results in a reasonable amount of time.
A second goal of this thesis is to assess, using this model of the battlefield, whether we can develop strategies with multi-agent algorithms, based on recent developments in deep learning and reinforcement learning.
The thesis is structured around these two goals. In chapter 2 the different algorithms will be developed. This chapter gives a brief overview of reinforcement learning and deep learning before treating the algorithms that can be used in a multi-agent setting. Chapter 3 explains how the modelling of the battlefield is done and which simplifications are made. Combining algorithms and models is done in chapter 4, where the different results will be discussed. Chapter 5 discusses potential directions for further research and conclusions are drawn in chapter 6.

In the interest of readability, I've avoided to use any computer code in this thesis (except for the appendices). However, all results are the consequence of a significant coding effort in `Python 3` and `PyTorch`. This code can be found on the thesis' Github page: https://github.com/koenboeckx/VKHO. Some of the code was inspired by the excellent Starcraft Multi-Agent Challenge code repository [SRdW+19].

2. Algorithms

This chapter briefly explains the different algorithms used in this work. How they are used and what their purpose is for this thesis will be explained in chapter 4.
All the algorithms are part of a family of algorithms that can be viewed as *machine learning* (ML). Instead of telling a computer what to do to perform a specific task by using explicit instructions, the computer has to learn to correct his behavior based on examples and inference. ML is traditionally divided into 3 subcategories:

- *Supervised learning*, where we feed the algorithm with both the input and the correct output so that it can learn based on the predicted and expected output value,
- *Unsupervised learning*, where the algorithms learns to classify items based on their underlying distribution alone (and thus without being given the correct answer),
- *Reinforcement learning*, where there are no input/output examples available but the computer receives nevertheless a signal indicating how good or bad his decision was.

This chapter is constructed as follows: the first section introduces classical reinforcement learning concepts and algorithms. These algorithms can be classified into two main categories: Q-learning and policy gradients. Next, these algorithms will be extended with the use of neural networks, placing them squarely in the deep learning framework. Finally, we'll look at how these algorithms can be used in a multi-agent setting, where multiple agents interact with the environment and learn at the same time.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a form of Machine Learning where an agent learns what to do in order to maximize a numerical reward signal [SB18]. RL uses a *Markov Decision Process* (MDP) as the formal framework to solve a sequential decision making problem. In this framework, the learner and decision maker is the *agent* who interacts with the *environment* at discrete time steps $t = 0, 1, 2, \ldots$. At each time step t, the agent receives a representation of the environment's *state* S_t[1]. Based on this state information, the agent decides which action A_t to take and sends this action to the environment. The environment changes its internal state to a new state S_{t+1} and sends a reward signal $R_{t+1} \in \mathbb{R}$ back to the agent. This reward gives an indication how good the chosen action was. The Markov decision process is represented in figure 2.1.
The goal of RL is to choose a sequence of actions that maximizes the *cumulative discounted reward* G_t:

$$\begin{aligned} G_t &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.1}$$

[1] A note on notation: when we want to represent a generic state, action or reward, a lower case letter will be used (s, a, r). When we refer to a specific instance, e.g. obtained through sampling, we use capital letters (S_t, A_t, R_t) with an index that refers to the time instant when this sample was generated.

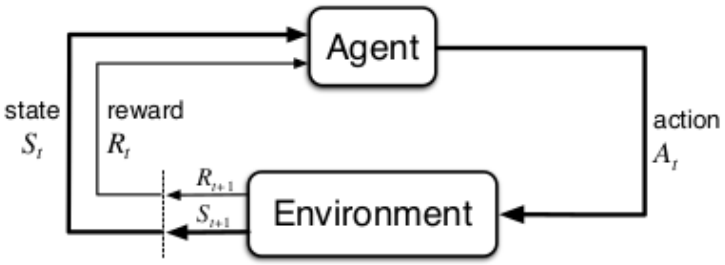


Figure 2.1: The agent-environment interface of an MDP (reproduced from [SB18])

where $0 \leqslant \gamma \leqslant 1$ is the *discount factor*.
This discount factor is a way to express that we are less certain about obtaining these future rewards. If $\gamma = 1$, all rewards, present and future, have the same impact. If $\gamma = 0$, we only consider the current reward. Typically, γ is chosen to be a bit less than 1. You could also interpret $1 - \gamma$ as the probability to end the episode at any stage, i.e. at every step we have a chance of $1 - \gamma$ to jump to a terminal state.
A function $\pi(s)$ is called a *policy* when it returns an action a to take in state s. More generally, a policy can also return a probability distribution over actions. In that case it is denoted as $\pi(a|s)$ (read: the policy over actions a given that we are in state s).
The *value* of a policy V_π is the expected value of G_t when starting in state $S_t = s$ and following policy π for the rest of the episode:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.2}$$

The expected value of taking action a in state s under policy π is denoted as $Q_\pi(s, a)$ and is called the *action-value function for* π:

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.3}$$

When a policy maximizes equation 2.1 it is called the optimal policy π_* and one can show that action-value function $Q_*(s, a)$ of this optimal policy obeys a recursive relationship called the *Bellman optimality equation*:

$$Q_*(s, a) = \sum_{s',r} p(s', r|s, a)\left[r + \gamma \max_{a'} Q_*(s', a')\right] \tag{2.4}$$

with $p(s', r|s, a)$ the probability of receiving reward r and moving to state s' if being in state s and taking action a. When $Q_*(s, a)$ is known, the optimal policy can be trivially computed with:

$$\pi_*(s) = \arg\max_a Q_*(s, a) \tag{2.5}$$

Several classical algorithms exist to solve this problem. However, they all assume that the state-transition probabilities $p(s', r|s, a)$ are known. Typically this is not the case and these probabilities have to be estimated. Methods that have no model of the transition probabilities are called *model-free*. These models learn by interacting with the environment by creating *trajectories* of successive states, actions and rewards:

$$\ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t, R_{t+1}, S_{t+1}, \ldots$$

The goal of RL is to learn the optimal policy $\pi_*(s)$. To accomplish this, two paradigms exist: *value approximation*, where we first approximate $Q(s, a)$ and then derive the policy according to 2.5, and *policy approximation*, where we derive the optimal policy $\pi_*(a|s)$ directly. Both paradigms are explained below.

2.1.1 Value approximation: Q-learning

The most well-known model-free method is *Q-learning* [Wat89]. This iterative algorithm builds on equation 2.4 to estimate $Q_*(s,a)$ by making a step in the direction of the *temporal difference*, the difference between the value estimate $Q(S_t, A_t)$ in the current state and the best value we can obtain from a next state:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big] \tag{2.6}$$

where α is the step size. The authors of [WD92] show that the empirical estimate $Q(S_t, A_t)$ converges to $Q_*(s,a)$.

The entire algorithm is shown in algorithm 1. Note the use of the current estimate Q to select the next action A to take.

Algorithm 1 Q-Learning

Input: step size α
Initialize $Q(s,a)$ for all $s \in \mathcal{S}$, all $a \in \mathcal{A}$
for *each episode* **do**
  for *each step of the episode* **do**
    Choose action A from state S using policy derived from Q (e.g. see eqn. 2.5)
    Take action A, observe reward R and next state S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  end
  until S is terminal
end

2.1.2 Policy approximation: Policy Gradients

The value approximation family of RL algorithms estimates the state-action value $Q(s,a)$ and then uses this estimate to select the best available action with equation 2.5. An other family of algorithms, the *policy gradient* (PG) methods [SMSM00], tries to find the optimal policy directly.

For these algorithms, a policy $\pi(a|s)$ is determined by a parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d$. The goal of the algorithm is to adjust $\boldsymbol{\theta}$ such that it maximizes a certain performance measure $J(\boldsymbol{\theta})$. In episodic MDP's, the performance measure will be the value of initial state: $J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(S_0)$. Maximizing $J(\boldsymbol{\theta})$ is typically done taking a step in the direction of the gradient of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$, a procedure called *gradient ascent*:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.7}$$

Computing the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is non-trivial in general, but under certain conditions we can use the *policy gradient theorem* which states that:

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}) \\ &\propto \mathbb{E}_\pi\Big[G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\Big] \end{aligned} \tag{2.8}$$

with $\mu(s)$ the distribution of states under policy π. The last expression is the one based on samples drawn from interacting with the environment. This equation is very useful because it links the gradient of the performance measure $J(\boldsymbol{\theta})$ to the gradient of the policy $\pi(A_t|S_t,\boldsymbol{\theta})$

which can easily be computed (especially with automatic differentiation frameworks like `PyTorch` - see section 2.2).
It follows that the policy parameter vector $\boldsymbol{\theta}$ is updated according to following rule:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\, G_t\, \frac{\nabla_{\boldsymbol{\theta}_t} \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)} \tag{2.9}$$

This expression has intuitive appeal: each increment of $\boldsymbol{\theta}$ is in the direction in parameter space that most increases the probability of taking action A_t in state S_t. This direction is multiplied by the expected reward G_t and divided by the action probability $\pi(A_t|S_t)$. The former causes the parameter to move most in directions that favor actions with higher returns. The latter makes sense because otherwise actions that are selected frequently have an unfair advantage.
Implementing equation 2.9 directly leads to the simplest PG algorithm, REINFORCE [Wil92]:

Algorithm 2 REINFORCE

Input: A differentiable policy $\pi(a|s, \boldsymbol{\theta})$
Initialize $\boldsymbol{\theta} \in \mathbb{R}^d$
while *not converged* **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(a|s, \boldsymbol{\theta})$
  for *each step of the episode* $t = 0, \ldots, T-1$ **do**
    $G_t \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\, G_t\, \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$
  end
end

Note that $\nabla_{\boldsymbol{\theta}} \ln \pi$ is shorthand for $\frac{\nabla_{\boldsymbol{\theta}} \pi}{\pi}$.

2.1.3 Actor-Critic Methods

While REINFORCE in theory converges to an optimal policy, it suffers from high variance: the gradient estimates are unbiased but very noisy. Thus convergence is slow and the algorithm is very sample-inefficient, i.e. many samples must be generated to achieve an acceptable result. A simple fix would be to add a *baseline* $b(S_t)$ that is then subtracted from the cumulative reward G_t. It is important that this baseline only depends on the state and not on the action. If that is the case, the expected value in 2.8 keeps its value, namely $\nabla J(\boldsymbol{\theta})$ and the estimator remains unbiased. The new update rule then becomes (from [SB18]):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left(G_t - b(S_t)\right) \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta}) \tag{2.10}$$

Actor-Critic methods [KT00] are methods that use a parametric policy $\pi(a|s, \boldsymbol{\theta})$ (the *actor*) to select actions and a function $\hat{V}(s|\boldsymbol{w})$ (the *critic*) to estimate the value of a state, just as in Q-learning methods. The baseline $b(S_t)$ then becomes the estimated state-value $V(S_t)$. Rewriting equation 2.10 leads to:

$$\begin{aligned} \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha \left(G_t - \hat{V}(S_t)\right) \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta}) \\ &= \boldsymbol{\theta}_t + \alpha \left(R_{t+1} + \gamma \hat{V}(S_{t+1}) - \hat{V}(S_t)\right) \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta}) \end{aligned} \tag{2.11}$$

where G_t is replaced by $R_{t+1} + \gamma \hat{V}(S_{t+1}$ according to equation 2.1. This is the update rule for the actor; in parallel, the critic is trained to accurately estimate the value of the state S_t.

Algorithm 3 Actor-Critic Algorithm (from [SB18])

Input: A differentiable policy $\pi(a|s, \boldsymbol{\theta})$ and value-function $V(S, \boldsymbol{w})$
Initialize $\boldsymbol{\theta} \in \mathbb{R}^d$
while *not converged* **do**
  Initialize S (first state of the episode)
  $I \leftarrow 1$
  while S *is not terminal* **do**
    Sample A $\pi(a|s, \boldsymbol{\theta})$
    Take action A, observe next state S' and reward R
    $\delta \leftarrow R + \gamma \hat{V}(S', \boldsymbol{w}) - V(S, \boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} I \delta \nabla_{\boldsymbol{w}} \hat{V}(S, \boldsymbol{w})$ ($\rightarrow$ Update rule for the *critic*)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$ ($\rightarrow$ Update rule for the *actor*)
    $I \leftarrow \gamma I$, $S \leftarrow S'$
  end
end

An alternative is to notice that according to equation 2.3, $Q(s, a)$ is an unbiased estimator for G_t. Replacing G_t by $\hat{Q}(S_t, A_t)$ and using baseline $\hat{V}(S_t)$ leads to

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left(\hat{Q}(S_t, A_t) - \hat{V}(S_t) \right) \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta}) \tag{2.12}$$

where the term $Q(s, a) - V(s)$ is also known as the *advantage* $\mathcal{A}(s, a)$ of taking action a in state s. $\mathcal{A}(s, a)$ is the improvement (positive or negative) that action a can bring in state s compared to the expected value $V(s)$ of that state. If we use the advantage function, the algorithm is known as *A2C* which stands for *Advantage Actor-Critic*.
Another advantage of the Actor-Critic algorithm in 3 is that the update of the both estimators can be done after each new sample, while in REINFORCE we must first generate an entire episode before any learning can be done. Notice also that the step sizes for critic and actor (resp. $\alpha^{\boldsymbol{w}}$ and $\alpha^{\boldsymbol{\theta}}$) are different.

2.2 Deep Reinforcement Learning

The RL algorithms described in the previous section use function approximators to represent the policy $\pi(a|s)$, the value function $V(s)$ or the state-action function $Q(s, a)$. These function approximators are most commonly implemented as neural networks. If that's the case, we speak of *Deep Reinforcement Learning* (DRL). In order to describe these DRL algorithms, we first explain what is *Deep Learning* (DL) and how it differs from other ML algorithms.

2.2.1 Deep Learning

This section briefly describes the essential aspects of deep learning needed to understand this thesis.

Neural Networks

Artificial intelligence in general and machine learning in particular made significant progress with the advent of *Deep Learning* [GBC16]. This is nothing more than a collection of methods for designing and training *neural networks* with several layers (hence *deep*). An artificial neural network is a function approximator that mimics the structure of the human brain, albeit in a very coarse way. It consists of layers of artificial neurons - see figure 2.2. These are mathematical objects that have several inputs $x_1, \ldots, x_m$, each of which are multiplied by a weight w_i and

then summed (together with a bias term b_i). This sum is then applied to an *activation function* $\phi(\cdot)$ to produce the output:

$$\begin{aligned} y &= \phi\Big(\sum_{i=1}^{m} w_i x_i + b\Big) \\ &= \phi(\boldsymbol{w}^T \boldsymbol{x} + b) \end{aligned} \tag{2.13}$$

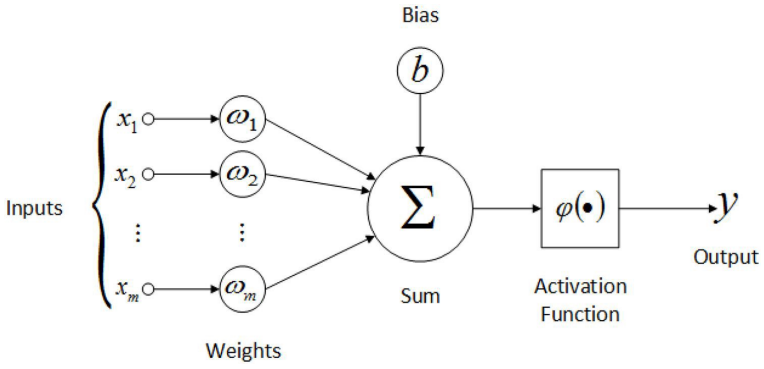


Figure 2.2: An artificial neuron

The activation function ϕ has to be a non-linear function like a sigmoid $\sigma(\cdot)$, a hyperbolic tangent function, a rectified linear unit (ReLU) or a leaky ReLU. These functions are represented in figure 2.3. It is important that these functions are non-linear; otherwise, the neural network would have no more representational power than a simple linear function. In recent algorithms, the non-linear function of choice will mostly be a ReLU.

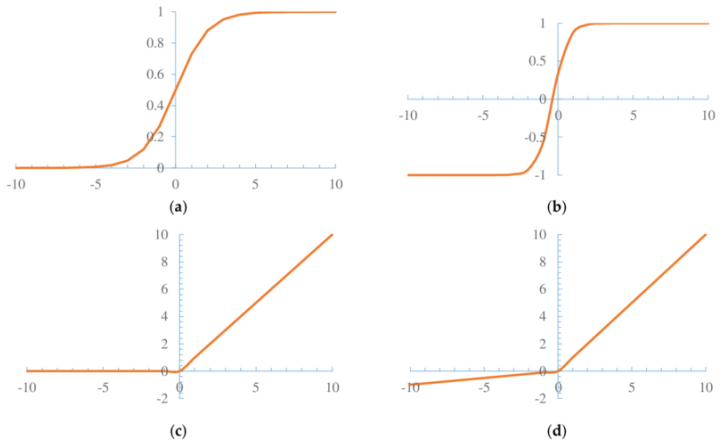


Figure 2.3: Activation functions: (a) sigmoid, (b): tanh, (c) ReLU, (d) Leaky ReLU

Neural networks [Nie15] are made of several layers of these artificial neurons (the white dots in figure 2.4). In this figure an input of 8 components is fed to the input layer and propagates through the different layers. At each step, the values are multiplied by the weights and passed through an activation function as in equation 2.13. Since the network has four output nodes, it can classify the input vector in four categories using one-hot encoding (i.e. for each category, only one output node will be activated while the others stay at zero).
The input to a neural network is a vector. Stacking multiple vectors alongside each other - for example when multiple samples are processed in parallel - gives an object which is referred to as a *tensor*. It can be thought of as the generalization of a matrix to multiple dimensions.

Stochastic Gradient Descent

Training these networks means adjusting the neuron weights to minimize some error criterion. This is done by feeding samples to the input layer and comparing the produced output with the desired output. The error between both is used to adjust the different weights w_i of the neurons with the help of an algorithm called *backpropagation*. For each training sample we have the corresponding correct output; hence, this procedure is called *supervised learning*.

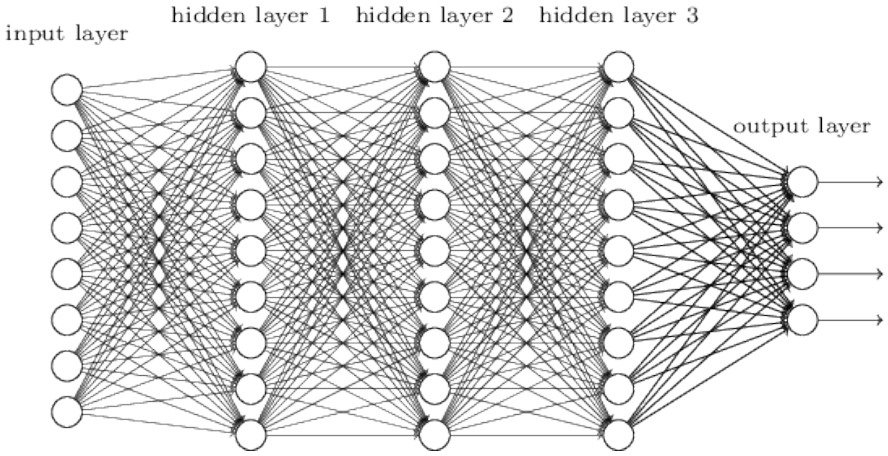


Figure 2.4: A multi-layer neural net (reproduced from [Nie15])

In order to adjust the network weights, the most commonly used procedure is *stochastic gradient descent*, which is the sample-based variant of expression 2.7. In stead of computing the gradient exactly, we estimate it based on one or on a few samples and their corresponding loss $J_i(\boldsymbol{\theta})$:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}) \tag{2.14}$$

with N the number of samples. The step size α is in this context often called the *learning rate* and in the future we will refer to it by this name.
While this technique only finds a local rather than a global minimum of the loss function, it has been shown that the resulting optimum works well in practice [Bot10]. However, if the local error surface is not well conditioned[2], convergence to the local minimum can be slow. A popular method to speed up convergence is SGD with *momentum* [Qia99], where the gradient that is used for the update is a moving average of the previous immediate gradients:

$$V_t = \beta V_{t-1} + (1-\beta) \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}) \qquad 0 \leq \beta < 1$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha V_t$$

A popular implementation of this paradigm is *Adam* [KB14], an algorithm that uses momentum with an adaptive learning rate. This is the optimization algorithm that will be used throughout this thesis.

Convolutional Neural Networks

A *Convolutional Neural Network* (CNN) [LBD+89] is a specific instance of an artificial neural network where several square windows 'glide' over an image and at each position the covered

[2] Technically, this means that there is a large difference in eigenvalues of the Hessian.

pixels are multiplied by the window weights and summed. This operation is called a *convolution*. These kind of networks are very well suited for processing images since they implicitly capture that the salient characteristics (forms, colors, texture, ...) of objects in images are invariant to translations.
The typical structure of a CNN used for classification is shown in figure 2.5. The input is an image of a robot. This image has a height and width (in number of pixels) and three color channels. Several convolutional windows process the image as described above and create a set of *feature maps*. One of these maps might be for example the vertical edges, another the horizontal .... The important thing is that the coefficients of these windows are learned by the system, and not defined by the user. Other elements however, like the size and number of these windows are part of the network architecture and must be defined by the programmer.

After the convolutions typically follows a subsampling phase, where for a window of pixels

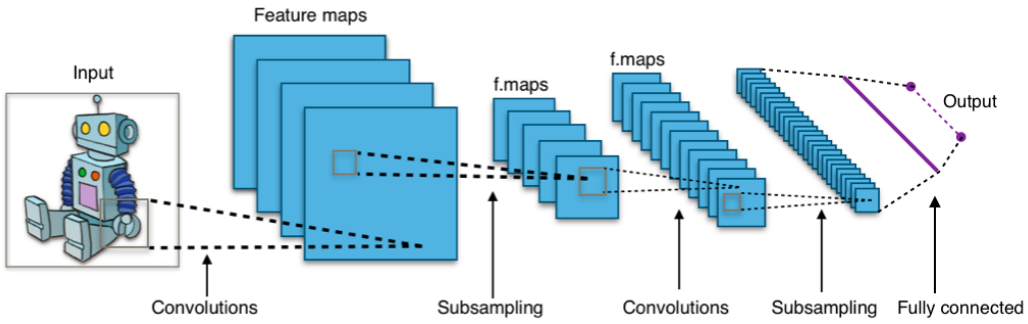


Figure 2.5: Structure of a typical CNN

only the one with the maximum value is retained (*max-pooling*) and which will reduce the computational overhead. Another sequence of convolutions and subsampling leads to a final set of feature maps. The idea is to learn more complex features as we move up in to hierarchy, going form edges to textures to parts of objects to complete objects. These feature maps are then reduced to a single vector which will be processed by a more traditional multi-layer network (a sequence of fully-connected layers).

Recurrent Neural Networks

Another special type of a neural network is a *Recurrent Neural Network* (RNN), which is mostly used to model sequential inputs. It is a normal neural net where the network output is at the next time step fed back to the input (after multiplication with weights V and activation), as shown in the left image of figure 2.6.
To analyze these kind of networks, they are "unfolded" so that each step is represented by a separate but identical network (the right part of figure 2.6). The internal state vector $\boldsymbol{h}$ (also called the *hidden state*) is then at each time step seen as a new "input" $\boldsymbol{h}_t$ to the network at time t, just as the true input $\boldsymbol{x}_t$.

The most prominent examples of RNN's are Long Short-Term Memory (LSTM) [HS97] networks and Gated Recurrent Unit (GRU) [CVMG+14] networks. These networks extend the basic structure of figure 2.6 to better model long-term dependencies in the input sequence. In what follows, GRU's will be explored in some detail since they will be the neural network of choice to model the policy- and value networks of the different agents.
GRU's use gating mechanisms to control and manage the flow of information between cells in the neural network. The goal is to capture dependencies from large sequences of data without discarding relevant information from earlier parts of the sequence. A GRU cell contains two gates: an *update gate* and a *reset gate* who are trained to selectively filter out any irrelevant information while keeping what is useful. In essence, these gates are vectors containing values

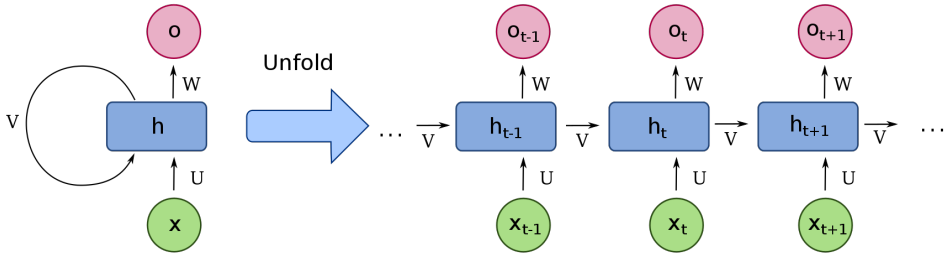


Figure 2.6: Structure of a typical RNN

between 0 and 1 which are multiplied with the input data or the hidden state. A 0 means this information is unimportant, while a 1 signifies the opposite. The reset gate is responsible for deciding which portions of the previous hidden state are to be combined with the current input to propose a new hidden state. The update gate is responsible for determining how much of the previous hidden state is to be retained and what portion of the proposed new hidden state is to be included in the final hidden state.
Figure 2.7 shows the details of a single GRU unit in a larger unfolded RNN. As before, the input of the cell is the input vector $\boldsymbol{x}_t$ and the hidden state $\boldsymbol{h}_{t-1}$ coming from the preceding cell. The reset vector $\boldsymbol{R}_t$ and update vector $\boldsymbol{Z}_t$ are generated by passing a linear combination of $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ through a sigmoid function σ, producing values between 0 and 1. Output $\boldsymbol{o}_t$ and new hidden state $\boldsymbol{h}_t$ are the result of applying a nonlinear function (here a tanh) to linear combinations of $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ and modulated by the value of the update and reset vector.

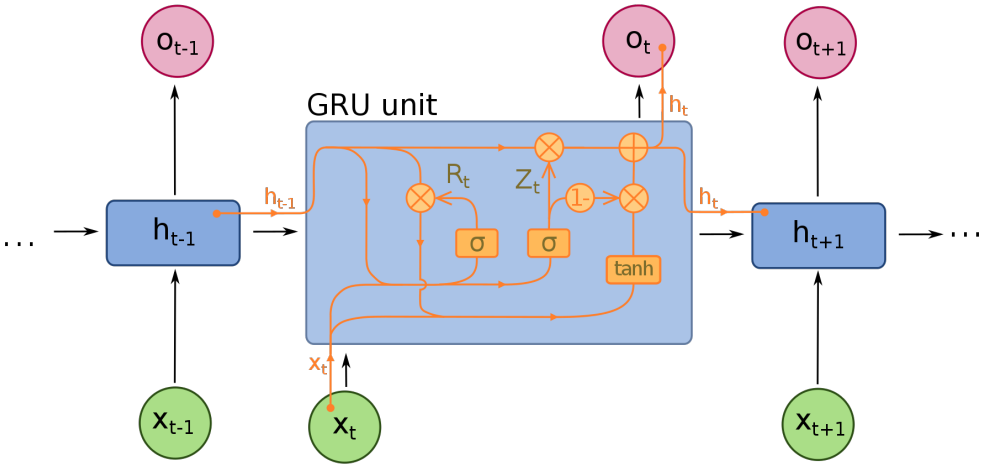


Figure 2.7: A GRU cell

The different parameters are thus computed as follows[3]:

$$
\begin{aligned}
R_t &= \sigma\Big[W_{ir}\boldsymbol{x}_t + W_{hr}\boldsymbol{h}_{t-1}\Big] \\
Z_t &= \sigma\Big[W_{iz}\boldsymbol{x}_t + W_{hz}\boldsymbol{h}_{t-1}\Big] \\
\boldsymbol{n}_t &= \tanh\Big[W_{in}\boldsymbol{x}_t + R_t \otimes W_{hz}\boldsymbol{h}_{t-1}\Big] \\
\boldsymbol{h}_t &= \big(1 - Z_t\big) \otimes \boldsymbol{n}_t + Z_t \otimes \boldsymbol{h}_{t-1}
\end{aligned}
$$

where $\otimes$ is an element-wise product. All the parameters W_{xx} of the linear combinations have to be learned by applying SGD and backpropagation to a loss function defined over the outputs. Implementing all this is a non-trivial task but is greatly facilitated by the existence of deep learning frameworks.
While the defining characteristics of neural networks have been known since the 1980s, it was the combination of larger datasets, more computing power and new engineering techniques that led to the deep learning boom. In addition, several deep learning frameworks have been created to speed up development. These frameworks allow easy (and often dynamic) construction of computational graphs (roughly the architecture of the neural network) and implement automatically the backpropagation algorithm for the specific network. The best known frameworks are `Tensorflow` (by Google), `PyTorch` (by Facebook) and `CNTK` (by Microsoft). In this work `PyTorch` will be used.

2.2.2 Deep Q-Networks

A deep Q-network (DQN) was one of the first implementations of RL with deep learning methods. Its first use was to discover policies to play Atari games [MKS+13] and is considered to be one of the breakthroughs of recent AI.
In DQN, a neural network is trained to estimate the Q-value of a particular state-action pair. This means that in algorithm 1 the $Q(S, A)$ table is replaced with a neural net $f_\theta(S)$ that has N_A outputs and thus produces the estimated Q-value for each action A. Applied to Atari games like Pong, the state of the environment, and thus the input of this neural network, is the image of the game projected on the computer screen. Consequently, the initial layers of the neural net are convolutional layers, while the final layers are fully connected layers to estimate the output values.
Training is done by generating samples through interaction with the environment. The error needed to adjust the network weights is the square of the temporal difference of the traditional Q-learning algorithm:

$$\mathcal{L} = \big[y_t - Q(S_t, A_t); \theta\big]^2 \tag{2.15}$$

where y_i is the target value, namely the best estimated Q-value reachable from the next state S_{i+1}, just as before:

$$y_t = R_t + \gamma \max_{A'} Q(S_{t+1}, A'; \boldsymbol{\theta}) \tag{2.16}$$

One of the problems with this setup is that machine learning techniques require that the samples that are used to learn from are independent and identically distributed (i.i.d.). Since the samples generated in the above way are part of the same trajectory, they are not independent. That's why the authors of [MKS+15] implement a *replay memory* D: all samples are pooled and minibatches are sampled from this pool to perform learning. This avoids instabilities due to non-i.i.d. samples.
Another trick to improve the stability of learning is the use of a *target network* $\hat{Q}$ [MKS+15], a

[3] Biases b removed for simplicity.

copy of the action-value network that is used to compute the Q-value in the temporal difference value of equation 2.16. This network is synchronized with the main network every C steps. In this way, we avoid computing the target Q-value with the same network that gets updated during every iteration step. The DQN-algorithm is shown in algorithm 4.
Note here the procedure for action selection: the best action according to equation 2.5 is not always selected; sometimes (more precisely, with probability ϵ) an action is chosen randomly from the action-space. This action selection improves the exploration of the state-space and is called *ϵ-greedy*. The value of ϵ is reduced during learning, to shift from an emphasis on exploration of the state-action space to exploitation of promising results. The tension between exploration and exploitation is a classic trade-off in reinforcement learning.

Algorithm 4 Deep Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights $\boldsymbol{\theta}$
Initialize target action-value function $\hat{Q}$ with random weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
for *episode* $= 1 \ldots M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ **for** t=*1...* **do**
    With probability ϵ, select a random action a_t
    Otherwise select $a_t = \arg\max_a Q(s_t, a; \boldsymbol{\theta})$
    Execute action a_t and observe reward r_t and next state s_{t+1}
    Store tuple (s_t, a_t, r_t, s_{t+1}) in D
    Sample a random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from D.
    If episode terminates at step $j+1$, set $y_j = r_j$
    Otherwise, set $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \boldsymbol{\theta}^-)$
    Perform gradient descent step on $\Big(y_j - Q(s_j, a_j; \boldsymbol{\theta})\Big)^2$
    Every C steps, synchronize $\hat{Q} = Q$
  end
end

2.2.3 Deep Policy Gradients

When applying deep learning methods to the policy gradient methods of section 2.1.2, the policy that is to be optimized is represented as a neural network (see figure 2.8). The input of the network is the system state (or observation) while the output is a distribution over the possible actions. The action to take is then sampled from this distribution. Adjusting the

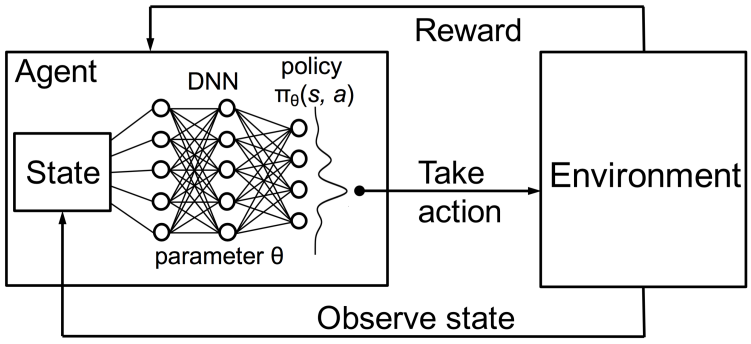


Figure 2.8: Deep Policy Gradient

network weights is done by using an update rule similar to equation 2.10. The cost function thus becomes the logarithm over the action distribution.
Similarly, the critic will also be represented as a neural network. Typically, both actor- and

critic-network share the lower layers in a common body, since they both have to process the same state and there is no reason to assume that the learned features in these layers should be different for actor or critic. This helps by letting actor and critic share the low-level features but combine them in a different way. The end result is a shared network which predicts action distributions and state values simultaneously. Learning is done more efficiently in this manner. Figure 2.9 represents this concept.

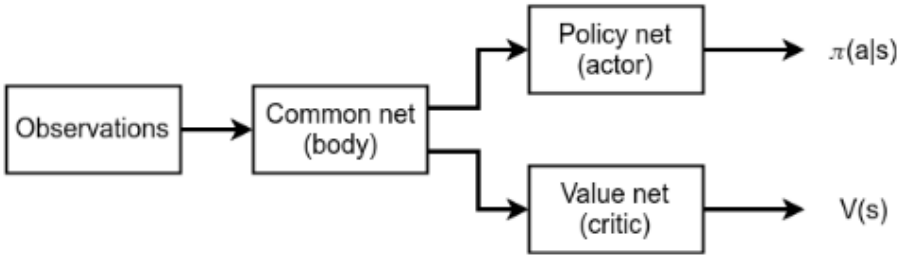


Figure 2.9: An actor-critic network with common body and separate heads

Just as in DQN, there is a high probability that an agent will converge to a policy that is locally optimal but is far from the global optimum. In other words, the policy space wasn't explored enough. In DQN, this can be solved by gradually decreasing the ϵ-parameter in ϵ-greedy action selection. While similar methods can be used for DPG, a better method is available. By adding the *entropy*[4] $H(\pi)$ to the loss-function, we can penalize policies where the agent is too sure of certain actions, effectively steering the agent to policies that improve exploration of the policy space.
In the computational graph of a network of type of figure 2.9, we define only a single loss function and a single optimizer that will use this loss function to update the different weights. The loss function thus becomes the sum of three parts:

- a term to simulate the step in the direction of an improved policy:

$$\mathcal{L}_{pol} = -\log(A_t|S_t)\mathcal{A}(S_t, A_t) \tag{2.17}$$

  Mind the minus sign because we want to increase the value of the policy as the loss is minimized.

- a term to make the critic converge to a correct estimate of the value of the states:

$$\mathcal{L}_{val} = \frac{1}{2}\left[R_t + \gamma \max_{A'} Q(S_{t+1}, A') - Q(S_t, A_t)\right]^2 \tag{2.18}$$

- a term that represents a fraction of the policy entropy, computed over all the states:

$$\mathcal{L}_{ent} = -\beta H(s) \tag{2.19}$$

The total loss is the combination of these three loss functions:

$$\mathcal{L} = \mathcal{L}_{pol} + \mathcal{L}_{val} + \mathcal{L}_{ent} \tag{2.20}$$

The policies in the preceding paragraphs were conditioned on the current state S_t. The expressive power of these policy networks increases when we could condition on the entire set of previously seen observations, namely the state trajectory τ. This is done implicitly by using recurrent networks like a GRU, where the hidden state encodes the information about the previously seen states. This is common practice in algorithms like COMA and QMix (see section 2.3.2).

[4]The entropy of a probability distribution equals $H(\pi) = -\sum_a \pi(a|s)\log\pi(a|s)$. It is maximum when π is uniform and minimal when all probability mass is placed on a single action.

2.3 Multi-Agent Reinforcement Learning

While reinforcement learning is a promising technique for training agents, the mathematical framework that justifies it is inappropriate for multi-agent environments. The theory of Markov Decision Processes assumes that the agent's environment is stationary and no other adaptive agents are present. This is no longer the case when we consider multiple learning agents.
Markov Games [Lit94] are an extension of the MDP framework that accounts for interacting agents. To describe a Markov game with k agents, we consider:

- a set of states $\mathcal{S}$,
- a collection of action sets $A_1, \ldots, A_k$, one set for each agent,

Transitions from one state to the next are controlled by a transition function $T : \mathcal{S} \times A_1 \times \ldots A_k \rightarrow \text{PD}(\mathcal{S})$ with $\text{PD}(\mathcal{S})$ the set of probability distribution over states. It is important to note that a transition is determined by the *joint action set* $A_1 \times \ldots A_k$ and not just a single action. Agents can thus influence each other in complex ways. A single instance of this joint action space is abbreviated $\boldsymbol{u} \in \mathcal{A} = A_1 \times \ldots A_k$.
Every agent i has an associated reward function $\mathcal{R}_i : \mathcal{S} \times A_1 \times \ldots A_k \rightarrow \mathbb{R}$ and tries to maximize its expected sum of discounted rewards $\mathbb{E}\Big[\sum_{j=0}^{\infty} \gamma^j R_{i,t+j}\Big]$. Thus, since the rewards and transition function depends on the joint action set, the value and action-value functions for agent i do as well:

$$V_i^{\pi} = \sum_{\boldsymbol{u}} \pi(s, \boldsymbol{u}) \sum_{s} \mathcal{T}(s, u_i, \boldsymbol{u}_{-i}, s') \big[\mathcal{R}_i((s, u_i, \boldsymbol{u}_{-i}) + \gamma V_i(s')\big] \tag{2.21}$$

and the optimal policy for agent i will also depend on the policies of the other agents. This is another way of saying that in the presence of other learning agents, the environment isn't stationary.
The following sections describe a couple algorithms that try to maximize this term in an efficient manner. These algorithms are typically classified under the header of *Multi-Agent Reinforcement Learning* (MARL). A particular issue with multi-agent learning is the challenge of multi-agent *credit assignment*: when two or more agents cooperate, which action of which agent led to the victory (or the loss)? Since the reward is global and the same for both players, this can be difficult to discern.

2.3.1 Independent Multi-Agent Learning

The simplest MARL algorithm, Independent Q-Learning (IQL) [Tan93] simply ignores the multi-agent setting, and lets each learning agent i train an state-action value function $Q_i(s, a_i)$ independently of the other agents. As such, this is just a simple extension of single-agent learning. Multiple agents always outperform a single agent because they have more resources and a better chance for receiving rewards. However, the true advantage from a multi-agent setting comes from cooperation between agents, something that is not addressed with this method. More implementation details on how the training of a set of agents with Independent Q-Learning is done can be found in appendix A.
Another algorithm along the same line is Independent Actor Critic (IAC), where the agents are trained independently with a DPG algorithm like REINFORCE or A2C. Both IQL and IAC have been implemented as part of this thesis. The goal is to provide a baseline against which other algorithms can be compared. An example implementation of IAC is given in appendix B.

2.3.2 QMix

QMix [RSDW+18] is recent MARL algorithm that relies on the DQN paradigm. Properly capturing the effects of the agents' actions requires a global action-value function $Q_{tot}(s, \boldsymbol{u})$.

However, this function is difficult to learn and even if learned, it is not obvious how to extract decentralized policies for individual agents. A simple solution would be to learn a centralised Q_{tot} that is the sum of individual value functions Q_i for the individual agents i:

$$Q_{tot}(s, \boldsymbol{u}) = \sum_{i=1}^{n} Q_i(s_i, a_i) \tag{2.22}$$

A decentralised policy for agent i would then simply be greedy action selection based on Q_i. This is how *Value Decomposition Networks* (VDN) [SLG+18] work. However, restraining Q_{tot} to the sum of individual Q-functions is too restrictive to approximate a joint action-value function. The idea behind QMix is that we only need to ensure that finding the best global joint action on Q_{tot} yields the same result as a set of best actions on individual Q_i's:

$$\boldsymbol{u}^* = \arg\max_{\boldsymbol{u}} Q_{tot}(s, \boldsymbol{u}) = \left[\arg\max_{a_1} Q_1(s, a_1), \ \ldots, \ \arg\max_{a_N} Q_1(s, a_N) \right] \tag{2.23}$$

A sufficient condition to realize this is to enforce a monotonicity constraint on the relationship between Q_{tot} and each Q_i:

$$\frac{\partial Q_{tot}}{\partial Q_i} \geq 0, \forall i \tag{2.24}$$

QMix does this by representing each Q_i by an *agent network*. All these Q_i's are combined into Q_{tot} with a *mixing network* in a non-linear way (contrary to VDN) but enforces conditions 2.24 by forcing the weights of the mixing network to be positive. The details of these networks and how they interact is shown in figure 2.10.

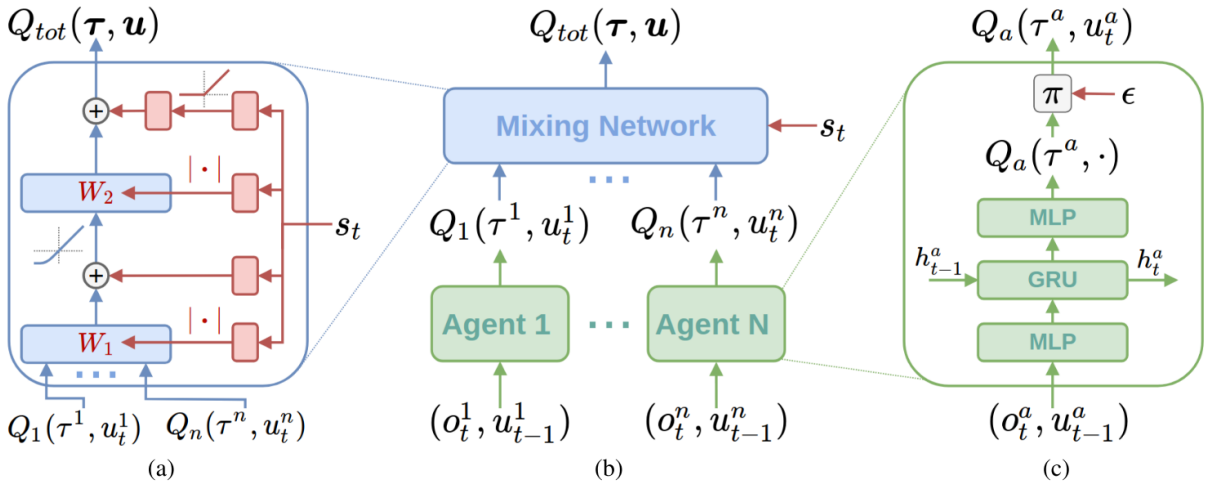


Figure 2.10: Structure of QMix (reproduced from [RSDW+18])

Figure (b) is the global structure, where the results of several agent networks are combined in a mixing network. Figure (c) shows an agent network. Once again, a recurrent net (a GRU) is used to track the agent's trajectory. The result is the "best" Q-value available in this state, determined by ϵ-greedy selection. Figure (a) shows the mixing network. Small hypernetworks (in red) take the state s and produce the weights W_k of the network. They consist of a single linear layer and an absolute activation function $|\cdot|$ which ensures that these weights are always positive.
This algorithm has two advantages compared with IQL and even VDN:

1. It combines the estimated Q-value of the different agents into a new Q_{tot} that is a non-linear combination of the individual Q-values. By doing this, the algorithm will prefer actions

that are good for the collective instead of just being good for individual agents. The idea is that this encourages the agents to cooperate. This could then result in emerging strategies that require interaction between agents.

2. During training, this combined Q_{tot} is computed by taking all state information into account. However, during execution, this state information is no longer available but it isn't needed, since the action selection is only based on the past and the current observation for each individual agent. This is an example of the centralised learning - decentralised execution paradigm [OSV08].

Appendix C shows how QMix is implemented.
Several other centralised learning - decentralised execution algorithms have been proposed in recent years. The most prominent example is *Counterfactual Multi-Agent Policy Gradients* (COMA) [FFA+18] which is an algorithm of the policy gradient family. While the initial goal was to also implement this algorithm, this has been abandoned due to time constraints.

3. Modelling of the Battlefield

As expressed in the introduction, this thesis wants to be innovative in 2 ways:

1. Create a mathematical model of the battlefield that can be used for simulation on a computer.

2. Assess whether state-of-the-art MARL techniques are able to develop interesting strategies in the framework of this model.

In this chapter, we will develop a simple and a more complex model of the battlefield. Chapter 4 will evaluate how the algorithms of the previous chapter interact with these models.
A mathematical - or algorithmic - model has two goals:

1. To make explicit which simplifications we'll make, both about the terrain itself as about the agents that interact on this terrain.

2. To allow a computer to manipulate the model, simulate different situations and predict their outcomes.

An important question is which level of detail we aspire to. As in all simulations, this must be a trade-off between making something as simple as possible to reduce computational complexity, and retain enough details to make it realistic and useful. To work towards this equilibrium, two versions of the model were developed with increasing level of complexity.
A model consists of two parts:

1. The *environment*, which contains the terrain and imposes the rules of the game. The environment keeps at all times a global state of the game which encapsulates all relevant knowledge about the current situation. The rules of the game determine which actions are allowed in which circumstances and what the next state s' will be when an action a is taken in a state s. The environment thus implements the transition probability $p(s'|s, a)$. It also provides a reward for each agent.

2. The *agents* that are present. Agents will be part of a *team*, in our case team "blue" and team "red". Only the agents of team blue are learning agents. The agents interact with the environment through their actions and receive observations.

This formalism is in line with the reinforcement learning paradigm as sketched in figure 2.1. At the same time, all forms of the game subscribe to the multi-agent formalism for Markov games, as defined in section 2.3.

3.1 Initial Model

Two opposing teams *blue* and *red* play against each other. Each team consists of two agents, which we'll consider to be tanks. The game is played on a flat terrain without obstacles. The size of the game board is small, typically 7 by 7.
This game offers perfectly symmetrical capabilities to both teams: every tank T_i is the same and has following individual state vector s_t^i:

1. a position (x, y) on the board,
2. a boolean value `alive` that determines whether the agent in question is still alive,
3. an integer `ammo` that specifies how many shots an agent has left,
4. an integer `aim` that specifies whether the agent is aiming at one of the opposing agents.

The global state $\boldsymbol{s_t}$ is then a tuple $(s_t^0, s_t^1, s_t^2, s_t^3) \in \boldsymbol{S}$ with $\boldsymbol{S}$ the set of global states.
Each agent can choose among 8 actions:

1. `do_nothing`
   As the name says, the agent does nothing and its state vector s^i doesn't change.
2. `aim0`
   Aims at the first tank of the opposing player and thus changes the `aim` variable of its state vector. The agent must be alive to be able to execute this action.
3. `aim1`
   Idem as above but aiming at the second tank of the opposing player.
4. `fire`
   Fire at the tank the agent is aiming at. This decreases the `ammo` counter. The agent must be alive, have sufficient ammo and must be aiming before executing this action.
   If the opposing tank is in range (determined by a parameter `max_range`), its `alive` indicator will be set to `False`. This means that every shot will result in a killed opponent if in range.
5. 4 `move` actions (`north`, `south`, `east`, `west`)
   The agent will move one step in the desired direction, unless he will drop of the board or tries to move to a tile occupied by an other agent.

The joint action space $\mathcal{A}$ thus consists of all tuples of actions $(a_t^0, a_t^1, a_t^2, a_t^3)$ whereby the first two actions are executed by agents belonging to player P_0 and the two others by agents of P_1. Actions are executed simultaneously. A player loses when both his tanks are dead or out of ammo. While the out-of-ammo criterion is a realistic assumption, it also avoids long games without end and thus speeds up learning. Additionally, a small penalty is introduced for every step an agent takes. This should induce an agent to prefer shorter episodes. Experience has shown that this addition does indeed shorten the average game time and speeds up learning compared to the situation without penalty. Additionally, at the reset of a game, all agents are placed in the same initial position.
All agents receive an observation from the environment that is based on the global state vector. An observation o_t^i for agent i consists of the following information:

- His own position, remaining ammo and, if applicable, the opposing agent he is aiming at.
- The relative position of his team mate and of the opposing agents, if they are still alive.

This implies that an agent does not know the ammo situation of other agents nor if any of the opposing agents is aiming and at whom. This is a design choice to mimic a realistic situation.
An advantage of this kind of representation is that it is defined from the view point of the agent itself thanks to e.g. the choice to communicate the position of others relative to the agents own position. This will allow us to share networks among agents from the same team which greatly increases sample efficiency and transferability of models, two advantages that will be discussed in chapter 4.
Figure 3.1 shows a standard visualization of the game, where tanks of the blue team are both dead (they are "greyed out").

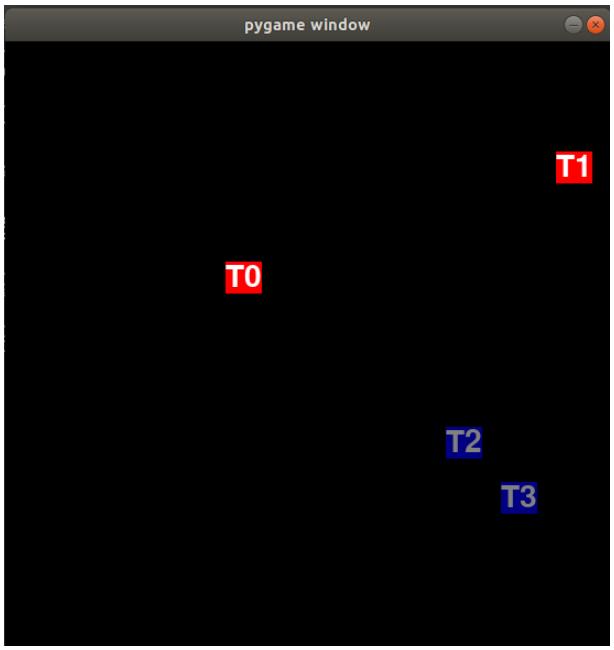


Figure 3.1: Visualization of the simple version of the game

3.2 Extended Model

The goal of extending the model is to make it more realistic while still keeping it manageable. Several extensions to the simple model have been considered. These proposed extensions include:

1. Using asymmetric agents with different capabilities (e.g. tanks vs. warriors with RPG).
2. More than 2 vs. 2 agents.
3. Complex terrain where both visibility and mobility are blocked by obstacles.
4. Larger state space, where e.g. fuel is limited.
5. More complex firing probability, e.g. an exponentially decreasing hit probability based on distance between shooter and target.

Since time for this thesis was limited, only the second and third option were retained; the others will be explored in future research.
This means that the second model is a model where the agents all have the same capabilities as before but the number of agents of both teams can be chosen freely and is not limited. Furthermore, we'll take into account the terrain which consists of free space and obstacles.
Since the number of opposing agents increases, the number of actions available to an agent also increases. A part from `do_nothing`, `fire` and the `move`-actions, the agent can now chose to aim at any opponent on the board. This is a minor change but it implies that the size of the action-space is no longer fixed which in turn has implications for the RL algorithms.
Obstacles in the terrain block both the movement and the line-of-sight of the agents. A blocked line-of-sight means that firing is not allowed, although aiming is still possible.
Just as before, each agent receives an observation from the environment which is derived from the global state. The agent receives information about:

- His own position, ammo and who he's aiming at.
- The relative position of the other agents.

This observation is extended with:

- Information about which tiles contain an obstacle and this for the entire board.
- Information about which enemy agent is visible and can thus be fired at.

The environment thus uses the terrain information to inform an agent whether the opposing agents are visible. The agent is aware of the entire terrain, even the parts that are not visible. This is a deviation from reality and this shortcoming will be addressed in future research.
This environment model, together with the models for agents, actions, observations and states, can be found in https://github.com/koenboeckx/VKHO/blob/master/env.py.

4. Implementation & Evaluation

4.1 Notes on implementation

This section describes a couple of technical points about the implementation of the algorithms of chapter 2.

- Because of the structure of the observation that the environment provides to an agent (see 3.1), all (learning) agents can use the same network to compute their policy or to estimate Q-values. This is known as *weight sharing* and improves the sample efficiency since only one common network must be trained instead of a network per agent.

- This common network takes as input a tensor constructed from the observation and puts it through a fully-connected layer. The output for this layer is than fed to a GRU. At every time step, the hidden state of the GRU is transformed via another fully-connected layer into the policy and/or the estimated V- or Q-values, depending on the type of algorithm. The entire network for an actor-critic agent, which does both value estimation as policy prediction, is schematically represented in figure 4.1.

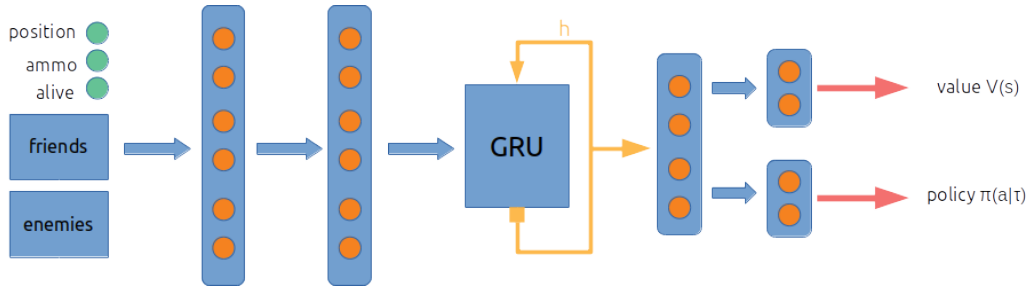


Figure 4.1: Actor-critic agent with a recurrent net

  A REINFORCE network only has the policy output. A Q-learning network outputs the Q-values for all potential actions.

- The output of a policy network isn't a probability distribution, but rather an array of numbers, one per possible action, computed by the final linear policy layer. These so-called *logits* l_i are transformed into probabilities $p(a_i|s)$ with a softmax function:

$$p(a_i|s) = \frac{e^{l_i}}{\sum_j e^{l_j}} \tag{4.1}$$

- In multi-agent settings, agents that have died no longer contribute to the game and the only action at their disposal is the `do_nothing` action. Care has been taken that for observations and actions that correspond to situations where the agent is not longer alive are not used for the update process. Experiments have shown that neglecting to do this leads to inferior results.

- In Q-learning based algorithms like IQL and QMix, the policy is created with the ϵ-greedy action selection mechanism as described in section 2.2.2. The ϵ-term is responsible for the trade-off between exploration and exploitation. To ensure enough exploration in the beginning, ϵ is typically reduced during training from its initial value of 1 to a low final value like 0.05. How this decrease of ϵ is done is determined by a scheduler which computes for each step in the learning process the corresponding value of ϵ. In this project, a linear scheduler was used.

- Algorithms of the Q-learning family, like IQL and QMix, have more hyperparameters than algorithms from the policy gradient family due to the presence of a replay buffer and a target network. These hyperparameters include the buffer and batch size, as well as the target synchronisation rate and the ϵ decay rate. This implies that searching an optimal combination of parameters becomes exponentially more difficult for these algorithms.

- In order to account for random effects, the average of the different measurements should be computed over multiple runs. However, this would mean that runs have to be repeated several times and development would slow down significantly. For this reason, rolling averages were computed by letting a window slide over a single run and averaging the measurements in this window. This can be justified by the fact that the learning rate is chosen small enough so that the weights of the network don't change significantly in this window[1].

- In all experiments, we used the ADAM variant of stochastic gradient descent. While the learning rate α is adapted during training by the algorithm as explained in section 2.2.1, the initial value that we have to set can have a big impact on the convergence of the overall algorithm.

4.2 Initial Model

4.2.1 Independent RL

This section discusses the application of IQL and IAC (section 2.3.1) to the simple battlefield model described in section 3.1. Two agents were trained with IQL and IAC on a 7-by-7 board against two opposing players who made random moves. The maximum range for all players was 5 tiles, so agents have to learn to approach other agents before firing. All training episodes were limited to maximum 100 steps.
A comment on random opponents: agents that make random moves are indeed simple to beat; the final goal however is to work in an iterative fashion:

1. Train two agents against two random agents.

2. Train two new agents against these two already trained agents until they can consistently beat them.

3. Continue in this way until no more progress is made.

An equivalent procedure to the one above and the results that were obtained will be discussed in section 4.4.
Before comparing the different algorithms, we will describe the results for independent REINFORCE on the problem defined above. The training of both agents was run for 50000 episodes, generated by sampling actions from the policy $\pi_i(a_t|s_t)$ and interacting with the environment.

[1] Technically, we assume that the signal is weakly stationary and the ergodic hypothesis thus holds.

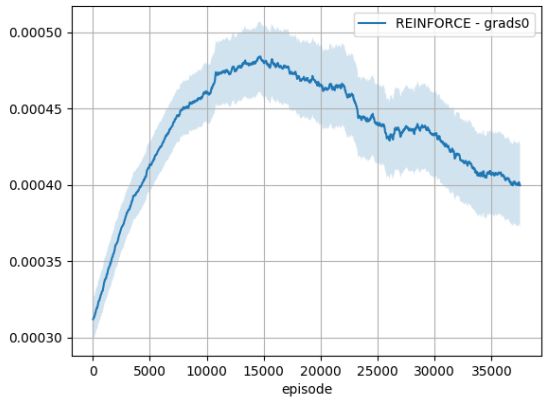


Figure 4.2: Gradient estimate for agent 0

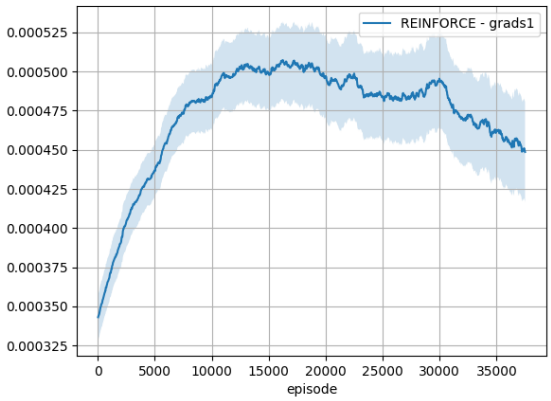


Figure 4.3: Gradient estimate for agent 1

First, let's compare the gradients of the loss function. As explained in section 2.1.2, these gradients will drive the weights of the network to their optimal values with the ADAM update procedure. Figures 4.2 and 4.3 show how the $L2$-norm[2] of each policy gradient $G_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$ evolves during training for both agents. This norm gives an indication how the gradient ascent algorithm changes the network parameters. Notice that while initially the gradient increases, it finally tops off and starts to decrease. Running the experiment longer will show both graphs converging to zero, meaning that the agents have reached their goal and are no longer learning. As explained previously, these curves are computed by taking the average over a moving window. The shaded area represents the confidence interval around the average value.
Figure 4.4 shows the average duration of an episode. Notice that initially, when the agent's policy network has random weights and the agent effectively behaves as a random agent, the episodes take a long time because none of the agents is taking any directed actions. Once the agent learns from experience, his behavior becomes more directive and the episode duration decreases rapidly. This behavior is encouraged by the fact that the agent receives a small negative reward for every step, incentivizing the agents to reduce the episode length. Figure 4.5 shows the average final reward per episode for the agents[3]. Initially, the agents lose as often as they win; the reward is negative, partially because agents are penalized for draws (e.g. because they run out of ammo). However, the win rate increases rapidly and finally the agents will win almost all the time. This curve is called the *learning curve* and will be the standard way to evaluate the performance of a MARL algorithm.

In this simple scenario, the agents have learned to (in this order):

1. Aim at the opposing agents (resp. $T0 \rightarrow T2$ and $T1 \rightarrow T3$).
2. Come closer until the opposing agent is in range.
3. Fire once the opposing agent is in range.

Figure 4.6 shows how this is done. A line between agents means one agent is aiming at the other. The actions taken by the four agents are shown in the top left corners.

Comparison

The next step consists of comparing the learning curves of the three types of independent learning algorithms: (i) Independent REINFORCE just as above, (ii) Independent Actor-Critic,

[2]The $L2$-norm is the square-root of the sum-of-squares of the gradients $\sqrt{\sum_i \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta})^2}$
[3]This is not the cumulative reward, and thus doesn't take into account any negative step rewards

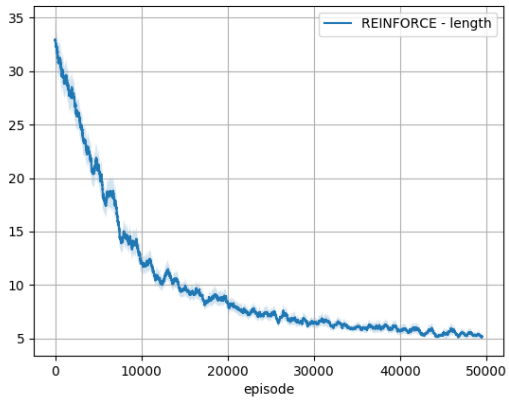


Figure 4.4: Mean episode length

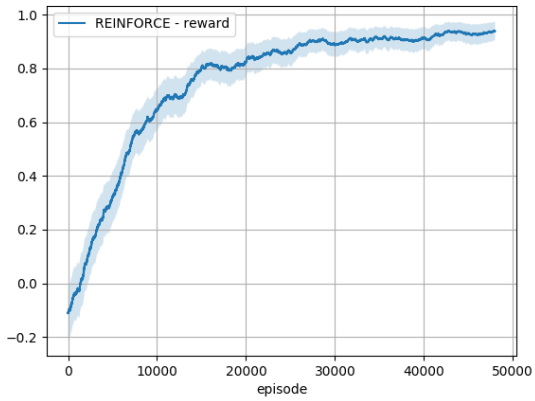


Figure 4.5: Mean reward for agents of team 1

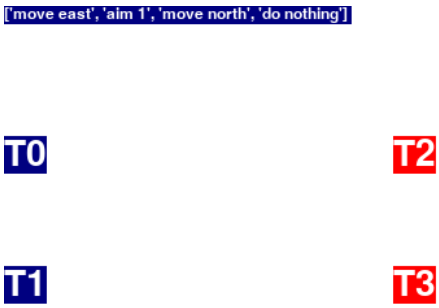
where REINFORCE is extended with a critic to estimate the state value and (iii) Independent Q-Learning (IQL), where the policy is derived from the Q-value for each action in a state. The resulting learning curves are represented in figure 4.7.
The following points are significant:

- Out of the 3 algorithms, REINFORCE works best since its learning curve is the steepest and it converges fastest to 1.
- IQL initially goes in the wrong direction (i.e. makes wrong moves) before starting to improve the reward. This is caused by the high initial ϵ in the ϵ-greedy action selection process which causes the behavior to be random instead of guided by the learned Q-values.
- To have a fair comparison, Q-learning algorithms will from now on be compared by generating episodes with an $\epsilon = 0$, thus always choosing the action with the highest estimated Q-value.
- Since IQL derives the policy from the learned Q-values, its behavior during learning undergoes significantly higher variation than REINFORCE. This is because gradual changes in $\hat{Q}(s, a)$ can lead to suddenly preferring one action over another.
- The learning curve of Independent Actor-Critic is slower than those for REINFORCE and IQL. This is probably because this algorithm has to learn both a policy and value-estimation function at the same time.
- Independent Actor-Critic is highly dependent on the inclusion of an entropy-term to avoid the policy distribution from collapsing to a distribution that puts all probability mass on a single action.

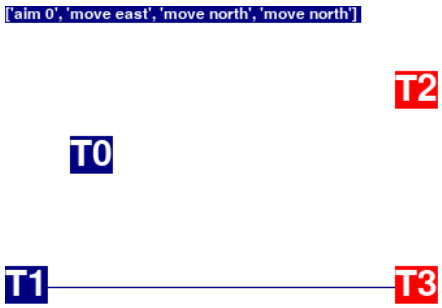
Independent REINFORCE definitely works best, and this has been confirmed by numerous experiments. IQL can be an alternative, but as mentioned above, many hyperparameters have to be tuned to achieve good performance. This point will also be addressed when discussing QMix. Independent Actor-Critic performs significantly worse than independent REINFORCE. Why this is the case shall be part of further investigations.
In the next section the performance of QMix will be compared against REINFORCE.

4.2.2 QMix

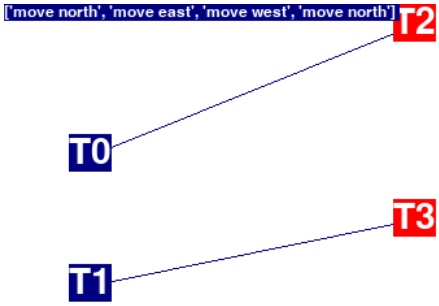
As explained in section 2.3.2, QMix is an algorithm of the Q-learning family that combines the Q-value estimate of the different agents into a Q_{tot} such that $\frac{\partial Q_{tot}}{\partial Q_i} \geq 0$ for all agents i. This



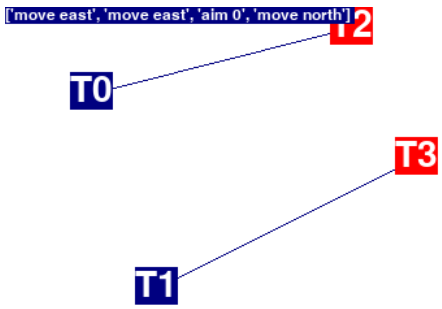
Initial positions.



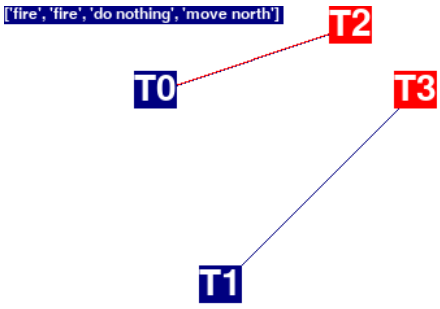
$T0$ moved east, while $T1$ aims at $T3$.



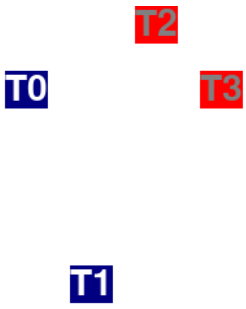
The opposite happens: $T0$ aims at $T2$ while $T1$ moved east. Both blue agents are now aiming but are too far away to fire.



Both blue agents keep moving east to get their opponents in fire range.



Once in range, they both fire.



Both red agents are dead.

Figure 4.6: A simple learned tactic

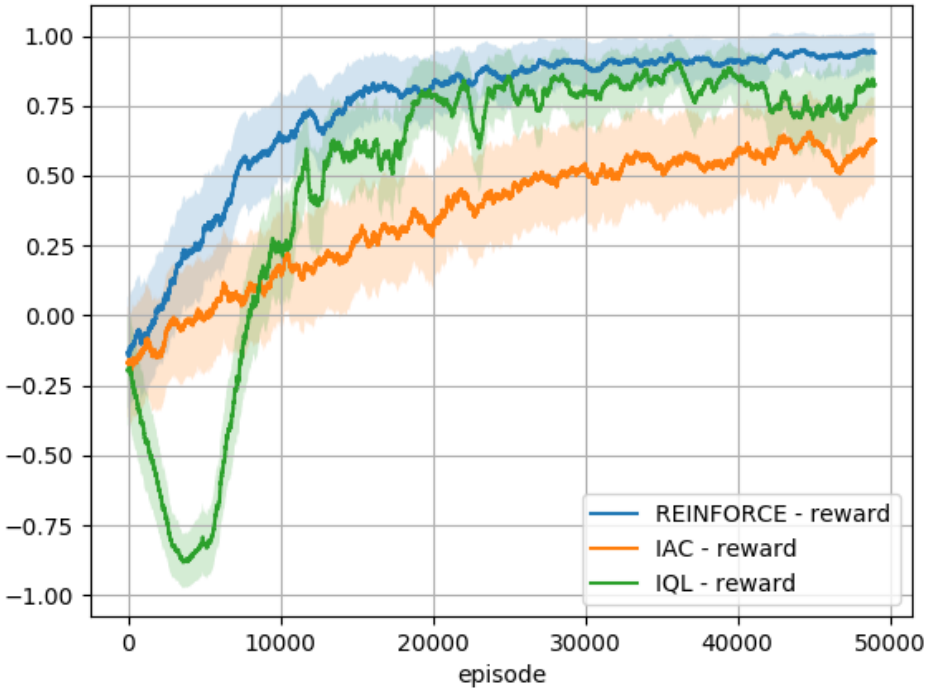


Figure 4.7: Learning rate for REINFORCE, Independent Actor-Critic and IQL for 2v2 games

guarantees that when each agent chooses his best action a_i, the combined action set $[a_1 \dots a_N]$ will also be the best possible joint action for the team.
QMix uses during training the state information s_t to set the weights of the mixing network in figure 2.10. To do this, the state information was transformed in a tensor of size `n_agents x 5`, where each agent is represented by 5 items: his position (x, y), whether he is still alive, his ammo level and at whom he is aiming at (if that is the case).
Figure 4.8 compares the learning curve of REINFORCE with the one for QMix. Both algorithms convergence to a situation where the blue team wins the large majority of its games. However, independent REINFORCE clearly works better, both in sample efficiency (since its curve is steeper) and in final reward which on average is much closer to 1. Indeed, investigation of the resulting strategies didn't show anything better than REINFORCE produced. However, QMix is an algorithm that encourages cooperation between agents and its advantages should become clearer when we look at the extended environment.
Experimentation has shown a few remarkable facts:

1. The discount factor γ is a critical parameter: policy gradient algorithms like REINFORCE require a high γ (around 0.99), while Q-learning based algorithms like IQL and QMix only converge when γ is lower (0.8 and below). A possible reason might be that PG algorithms base their update on a sequence of steps of a single episode, while Q-learning algorithms take random samples out a buffer of steps coming from multiple episodes. This means that the temporal effect of actions is more pronounced for PG and this can have an impact on the sensitivity on γ. However, this explanation has not been confirmed. The issue will be part of future research.

2. Convergence of QMix is much better when the step penalty is increased and the maximum episode length is kept low. REINFORCE in turn has no problem to converge with a lower (or even zero) step penalty.

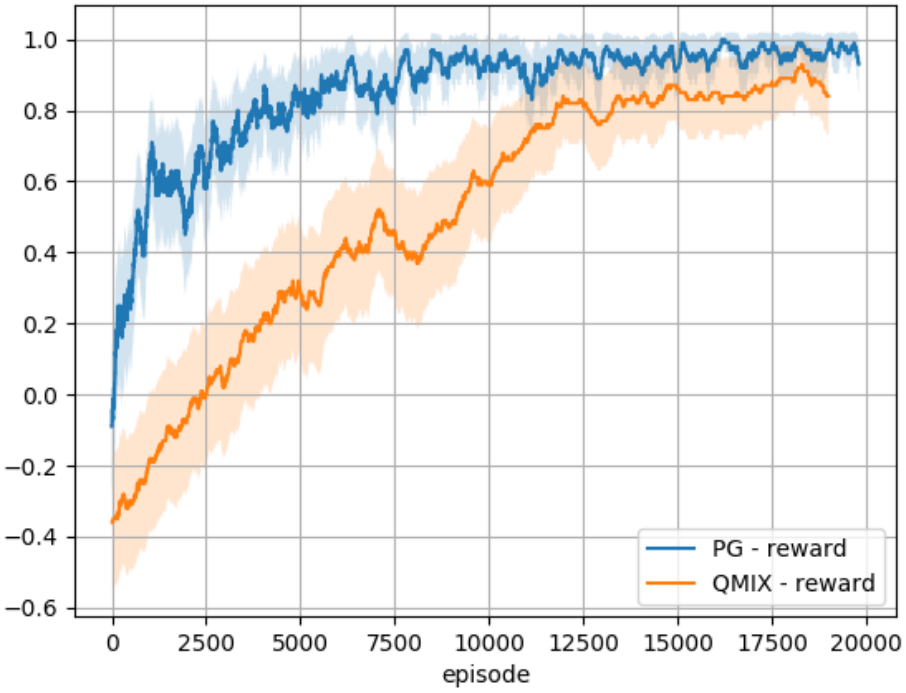


Figure 4.8: QMix vs independent REINFORCE in a simple environment

4.3 Extended Model

This section describes the obtained results when the different algorithms are applied to the second model of which the characteristics were sketched in section 3.2.
A new model also requires that a couple of technical points be addressed:

- Environments can host a number of agents that is not predefined. This implies that the action space must be dynamical since the number of `aim`-actions depends on the number of opposing agents. Hence the number of outputs of the policy- or Q-value network changes. This issue was already addressed in chapter 3.
- An additional requirement for being allowed to fire is that the opposing agent is visible, i.e. the line-of-sight is not blocked by an obstacle or other agent. This is communicated to the network by adding additional flags to the observation tensor, one flag per opponent that says whether he's visible or not.
- The terrain is added to the observation. Since the terrain doesn't change during play, this information is not stored when generating the episodes because that would waste resources. The information is appended to the observation matrix when it is offered to the network. For a board size n, this means an addition of n^2 elements to the input of the network, and thus also a severe increase in the number of network weights in the initial layer. This can be alleviated by using a convolutional network to process the board (containing both agents and terrain). This will be part of future research.

4.3.1 Independent RL

REINFORCE works reasonably well with the extended model. Figure 4.9 shows both win rate and average length for REINFORCE applied to the extended model with a large obstacle in

the middle (like in figure 4.11) on a 15-by-15 board, which is more than 4 times larger than the simple model. It takes about 20000 episodes to reach a 90% win rate; compare this with figure

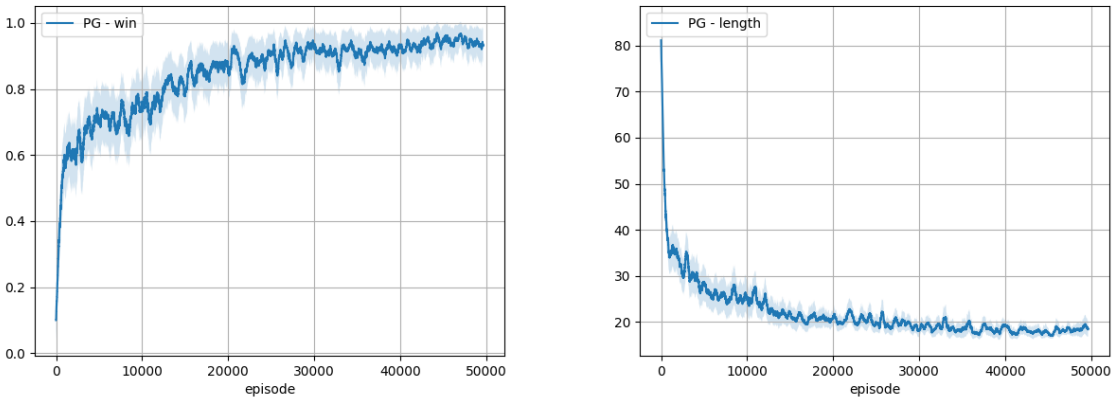


Figure 4.9: Win rate and average game length for REINFORCE on extended model 15-by-15 board

4.8 where REINFORCE requires about 6000 episodes to reach the same level of wins. However, increasing the board size with a factor of 4 entails an increase of the state-space size by a factor $\approx 4^4 = 256$ (since each of the 4 players can be placed on 4 times as many tiles). In this light the increase in required samples seems reasonable.
Another test is whether we can learn to win in a situation of 2 blue agents against 3 red ones, putting the blue team at a serious disadvantage. Figure 4.10 shows both the win rate and the episode length for this situation. These curves show that winning in that situation is definitely possible - although still against random agents - but notice that (compare with figure 4.8) the curve is a lot less steep, especially in the beginning. This would indicate that the algorithm needs a lot of episodes to figure out how to beat 3 opponents with 2 agents.

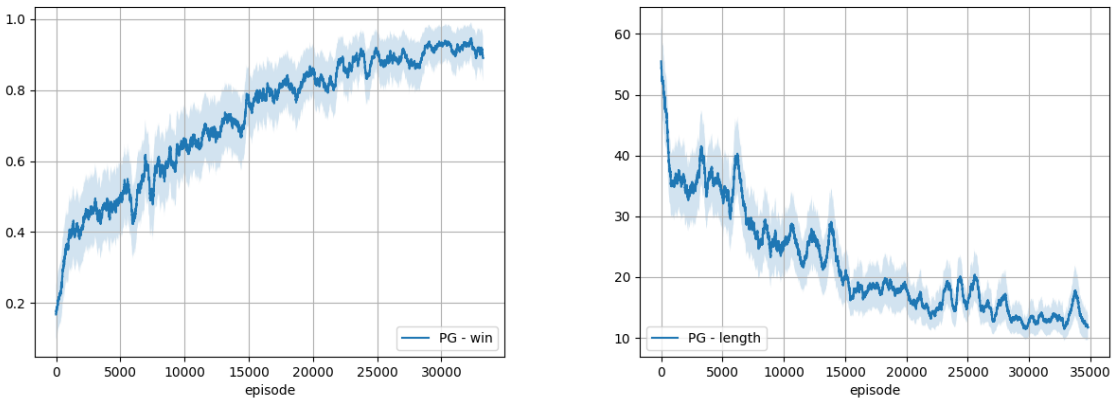
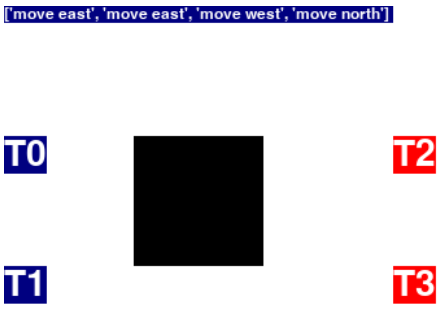
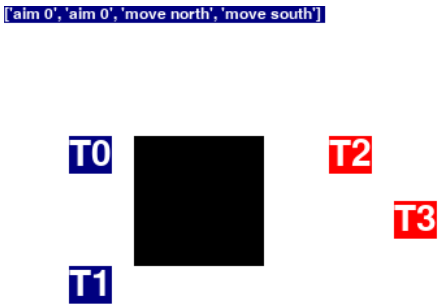


Figure 4.10: Win rate and average game length for REINFORCE on extended model 2v3 agents

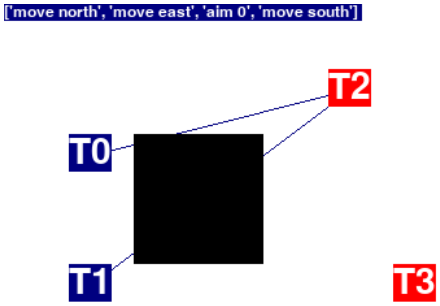
Figure 4.11 shows the play-out of a game on a 7-by-7 board with that same large obstacle in the middle. The blue and red lines between agents represent the aiming action of resp. blue and red agents. The action that the agents choose to take based on their current observation are shown in the top left of each figure. The captions below the figures explain how the agents behave.



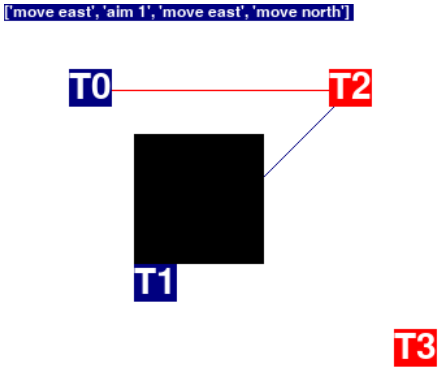
Initial positions with large obstacle in the middle of the board



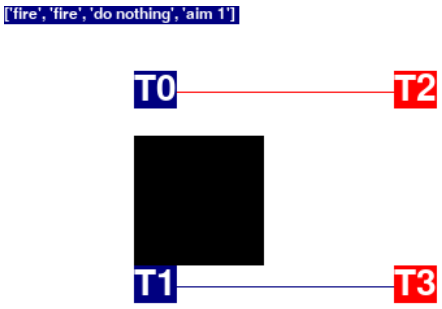
Both blue agents take action to aim at $T2$



Both blue agents are aiming at $T2$ but are not allowed to fire; agent $T0$ moves north to clear line-of-sight



$T0$ moves closer to $T2$ to get in fire range; agent $T1$ switches aim to agent $T3$



Both blue agents fire at opposing agents at the same time, killing both

Figure 4.11: A game with an obstacle

4.3.2 QMix

QMix works reasonably well on the extended model, as long as the board size is small. Once the board size becomes larger than 11-by-11, it becomes very hard to get QMix to converge. This is unfortunate, since intuitively, cooperation between agents becomes more important when the board size is large.
For smaller board sizes, QMix gives quite good results, as can been seen in figure 4.12 where the learning curve for REINFORCE and QMix is compared on a 9-by-9 board with obstacles in the terrain (a figure which is the counterpart of figure 4.8).

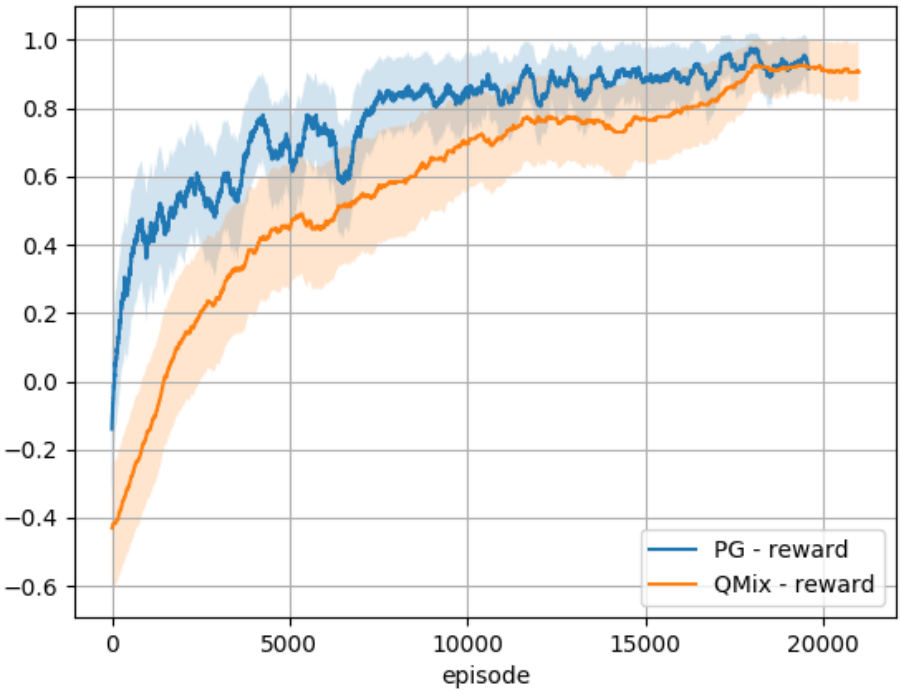


Figure 4.12: QMix vs independent REINFORCE on a 9-by-9 board

Notice that although being slower, QMix still reaches the 90% win rate in about 17000 episodes, which is reasonably compared to REINFORCE. Nevertheless, the conclusion remains that QMix lacks scalability, an issue that has not been resolved.

4.4 Model Transfer

Obtaining a model that performs well against an opponent that deploys a random strategy is one thing. More interesting things can happen when we train against more advanced agents. To do so, the neural network and its inputs on which an agent relies to choose his action has been constructed in such a way that:

- agents of the same team share the same network (weight sharing), and
- agents of the opposing team can reuse that trained network to obtain the same performance as the other team

In this way, we can swap the trained network from the blue team to the red one to get an opponent that doesn't behave randomly but uses a (slightly) smarter strategy. By doing this in an iterative fashion we continuously improve our strategy. More concretely:

1. Start training the blue team against a random opponent;
2. After a certain number of steps or once the blue team consistently beats the red one, transfer the network from the blue team to the red one;
3. Continue improving the blue team against the better opponent. Either start from a clean network or keep the already trained network. The former option might be useful to avoid getting stuck in a local minimum where the learned strategy is too brittle and only works in a limited number of cases. The latter option is obviously more sample efficient.
4. Go back to step 2.

Going forward, all figures were produced on a 7-by-7 board with teams of 2 agents and a terrain containing several obstacles.
Figure 4.13 shows the win rate for both blue and red teams where after each 5000 episodes the network was transferred to the red team and the agents from the blue team restarted training as random agents. The performance loss for the blue team - and the corresponding performance gain for the red team - is clearly visible every 5000-episode cycle. During some cycles (e.g. from episode 10000 to 15000) it is hard to overcome the opponent while during other cycles the blue team easily overcomes the red one (e.g. from episode 15000 to 20000).
Figure 4.14 is a rather striking example of the transfer of a QMix model from blue to red

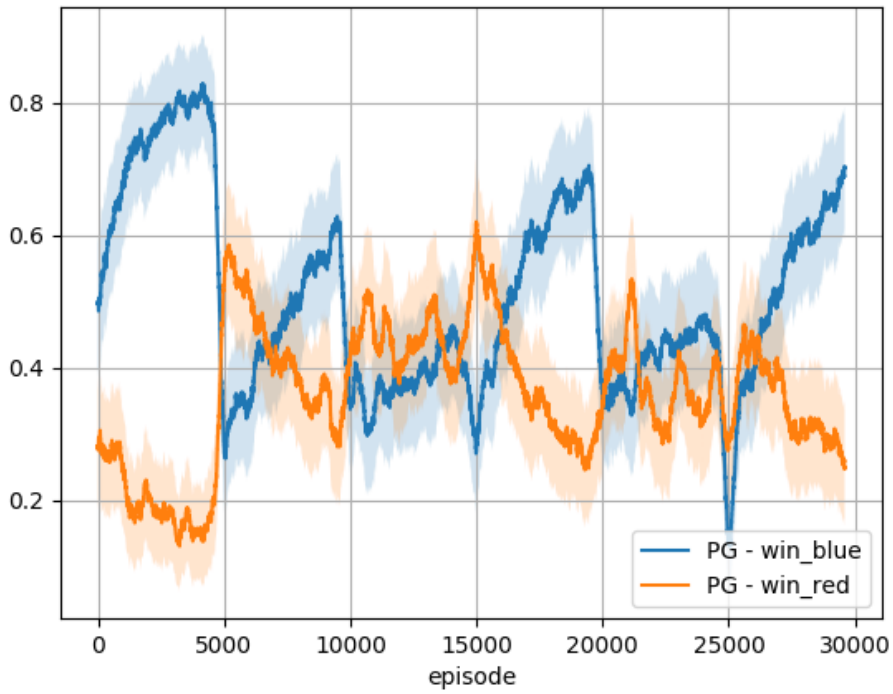


Figure 4.13: Sequence of model transfers from team blue to team red

whilst keeping the model for the blue team. A word of caution: this simulation ran for 60000 episodes, but a performance evaluation was only done every 10 episodes, thus the x-axis reports only 6000 episodes (actually a bit less because of the moving-window averaging with window size = 400). It is clear when the red team receives a new network since the performance briefly flares up. However, the blue performance remains relatively high during the entire sequence, a part from the occasional drop when the blue network is shared and the blue agents essentially have to compete against themselves.

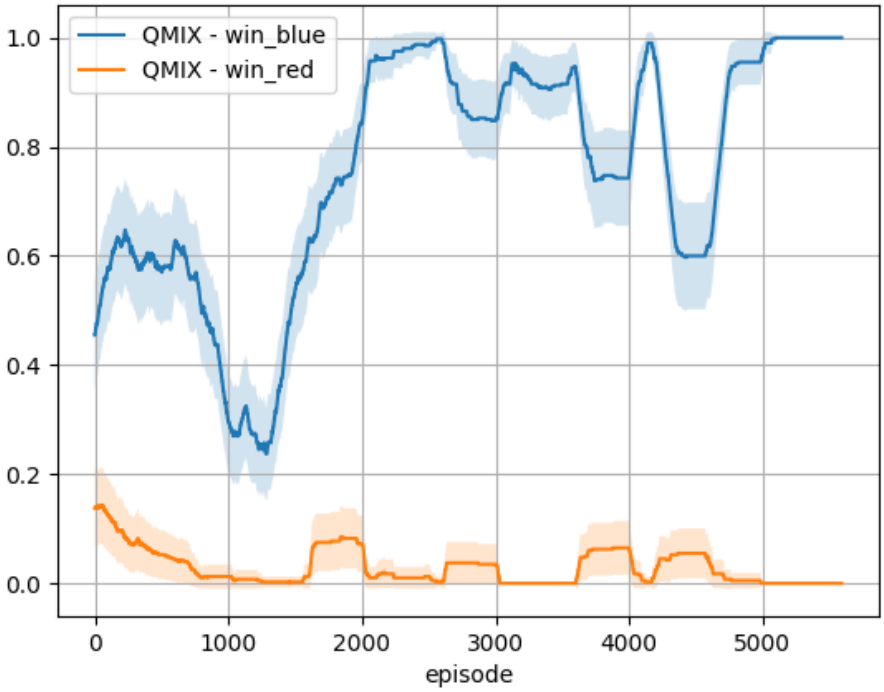


Figure 4.14: Sequence of model transfers for QMix - no network reset

5. Recommendations for Future Work

During the research for and the redaction of this thesis, a couple of points were raised that merit further research. These points include:

- The algorithms presented in this work are sometimes highly sensitive to a good choice of hyperparameters, like the learning rate, the shape and size of the neural network, the buffer size, ... More hyperparameters lead to a combinatorial explosion of the search space, making finding a good set of parameters very hard. A more complete exploration of this hyperparameter search space however might lead to better results.
  In particular, changing the size and shape of the neural networks has not been thoroughly explored and can have a significant impact on the performance.
- The impact of the discount factor γ on the convergence of algorithms of the two different algorithm families, as mentioned in section 4.2.2 should be examined.
- The significant worse performance of the actor-critic algorithm compared to the simpler REINFORCE algorithm while the theory suggests the opposite is something else that requires an explanation. This might be an implementation issue or be related to a bad choice of hyperparameters.
- The model should be made more realistic. This would include some of the suggestions offered in section 3.2 like asymmetric agents and more state variables like fuel. Another improvement might be to process the board with a convolutional neural network as suggested in section 4.3
- Other multi-agent algorithms, like the aforementioned COMA algorithm, should be implemented and tested

Aside from these points, I would like to propose some ideas for future research. Applying multi-agent RL to the problem as it has been set up is hard:

- The reward that an agent receives is sparse: only at the end of an episode does the environment give an indication of how good the sequence of moves was. This implies that the agent has no way to discern which particular action was good or which was bad. One way to address this would be to add some innate knowledge to the environment or the agent. This can be implemented in at least two different ways:

  1. Restrict certain actions in certain situations; e.g. we know that is of no use to fire at an enemy that is already dead. In the current model, these kind of actions are still allowed.
  2. Reward the agent for certain situations that seem beneficial; e.g. reward an individual agent for killing an opponent, even if this doesn't lead to the end of an episode. This is called *reward shaping* and can significantly reduce training time.

The danger of these methods is that certain beneficial strategies that were not anticipated will not be discovered.

- Similar to but distinct of the previous issue is the problem of temporal *credit assignment* and delayed rewards: the effect of good or bad actions might only be discovered after a while. From another point of view: when an episode is won it might not always be entirely clear which action or combination of actions led to the win. To improve selection of good actions, mechanisms must be put in place to select before an update those actions that contributed most to the final result. A simple example could be to purposefully decrease the weight of actions that didn't do anything, like taking a step down and then a step up to end up in the same spot.

- The assignment of credit among *agents* is also something that has not been addressed: actions from the first agent might lead to winning a game while the other agent does nothing of use. However, in the end they both receive the same reward.

Another problem is the problem of *generalization*:

- When the terrain or capabilities of the opposing agents changes, how well is the learned strategy able to cope with that change? Can we simply substitute one terrain model for another and expect the result to be as good? Should a little retraining happen (e.g. only retrain the final layer of the neural network that determines the policy while the other weights stay the same[1])? Or must the entire network be retrained?

- When the *strategy* of the opponent changes, how well does the learned strategy perform? This is a game-theoretic question: is there a strategy that performs best against all other strategies? The answer is most likely no. A strategy might work very well against a couple of other strategies but can be catastrophic against others, while another strategy might work reasonably well against a broad spectrum of opposing strategies.

Another avenue of research should be a more thorough exploration of the developed strategies, with more agents and in more challenging terrain. In a second phase, the strategies should be analyzed in cooperation with specialists from the Belgian Defence, e.g. the Defence College. This latter point was part of the initial thesis proposal; however, because the software and the results weren't mature enough, this hasn't been done yet.
A final suggestion for future research is more technical of nature. If these algorithms should be able to develop strategies in real time, a lot of work should be put in decreasing their run times. Running on my PC or even on the servers of the CISS department, a run time of a couple of hours is no exception. Analysis of the algorithmic run time has shown that the generation of episodes is the main bottleneck, followed by performing update steps on the neural network. The generation of episodes can be done in parallel before a network update is done since in that time frame the characteristics of the neural network don't change. Some existing algorithms, like *Asynchronous Advantage Actor-Critic* (A3C)[MBM+16], already do this and they might improve run time.

[1] This is known as *transfer learning*

6. Conclusion

As mentioned in the introduction, the goal of this thesis is twofold:

1. To develop an algorithmic model of a battlefield with opposing agents and a terrain that can both prohibit movement and visibility.

2. To assess the feasibility of using state-of-the-art RL algorithms on this model to develop strategies.

The first goal has been met: an algorithmic model has been designed and implemented that tries to take into account the most important elements of real-life combat situations. Furthermore, the implementation has been done in such a way that the model can be easily extended. These extensions may include things like additional state variables (e.g. fuel consumption), the addition of new restrictions on the behavior of agents, the addition of extra actions that an agent may take, creating obstacles that inhibit movement but not visibility or vice versa, different kinds of agents, ...
As for the second goal, a number of reinforcement algorithms have been implemented. These include algorithms from the policy gradient family that directly try to push the agents policy in the right direction, like REINFORCE or Actor-Critic algorithms, and algorithms from the Q-learning family where the goal is to make an accurate estimation of the Q-value and derive a policy from that. Examples of the latter family are Independent Q-learning and QMix.
The results of these algorithms is mixed. Purely based on performance, the simplest of these algorithms, REINFORCE, performs best. This might be because it's the best algorithm, because it is less sensitive to hyperparameters and the optimal set of hyperparameters for the other algorithms has not been found, or due to implementation issues with the other algorithms.
QMix is a true multi-agent algorithm, developed to stimulate coordination between agents and by doing so develop coordinated strategies. However, the limited experimentation performed with QMix on the environment model has not definitely shown this coordination. Furthermore, QMix has convergence problems when the board size becomes larger.
Care was taken during implementation to keep both the neural network and the observations for the different agents as generic as possible. This made it possible to easily transfer networks between agents, even of different teams. This in turn allowed the use of model transfer to learn to play against teams of increasing strength. This procedure works well and produces good results. However, more evaluation of the developed strategies is needed.
Personally, I would like to say that implementing these algorithms from scratch is not always an easy or straightforward feat. Some papers don't always make explicit which technical assumptions have been made, thus during implementation certain well considered choices had to be made in the hope that they align with the intention of the authors. Another difficulty is the large number of hyperparameters that each algorithm has. As mentioned previously, choosing a good combination of hyperparameters can be hard, and certain algorithms are very sensitive to a good choice of these parameters.
This work provided several contributions to the Belgian Defence:

- Participation in the IRIS project mentioned in the introduction.

- Development of a certain expertise in a domain that becomes more and more important, namely the deployment of AI in military operations.
- The start of the development of a tool that has the potential to contribute significantly to the decision making process.

Experience has shown that developing and maintaining AI-systems like the one proposed in this thesis requires a lot of resources. This not only includes computing power and data, but also personnel and time. The need for computing power is high because training these kind of systems is computationally expensive. Supervised and unsupervised learning systems require lots of training data before they become operational. Some aspects of AI have a steep learning curve, thus development needs engineers and programmers that have the required background. At least as important is the ability to interface with the end-users and the domain specialists to develop something that corresponds to their needs. Development of an AI-system is time-consuming because it will go through several iterations of back-and-forth between the development team and the domain specialists.
To conclude, I would like to say that using state-of-the-art RL algorithms to develop battlefield strategies seems to be possible; however it will take a lot of time and effort to develop something that is both useful & usable. An additional difficulty would be to deploy this in a vehicle where resources are limited and that can process all this information in real time.

Appendices

These appendices show how the major algorithms in this thesis (IQL, IAC, QMix) have been implemented. All imports, loggings and technical details that are less relevant have been removed for clarity. This means that as such, these algorithms are not functional.
To experiment with the algorithms, please consult the `GitHub` reference page:
https://github.com/koenboeckx/VKHO

A. Independent Q-Learning

```
def train(env, agents, lr=0.0001):

    # create and initialize model for agent
    input_shape = (1, env.board_size, env.board_size)
    for agent in agents:
        agent.set_model(input_shape, env.n_actions, lr)

    get_epsilon = create_temp_schedule(1.0, 0.1, 500000)

    reward_sum = 0 # keep track of average reward
    n_terminated = 0

    buffers = [ReplayBuffer(buffer_size) for _ in agents]
    state = env.get_init_game_state()

    for step_idx in range(int(n_steps)):

        eps = get_epsilon(step_idx)
        actions = [0, 0, 0, 0]
        for agent in env.agents:
            if agent in agents:
                # with prob epsilon, select random action a;
                # otherwise a = argmax Q(s, .)
                actions[agent.idx] = agent.get_action(state, epsilon=eps)
            else:
                # for other agents => pick random action
                actions[agent.idx] = agent.get_action(state)

        # Execute actions, get next state and rewards
        next_state = env.step(state, actions)
        reward = env.get_reward(next_state)

        done = False
        if env.terminal(next_state) != 0:
            n_terminated += 1
            reward_sum += reward[0]
            done = True
            next_state =  env.get_init_game_state()

        # Store transition (s, a, r, s') in replay buffer
        for idx, agent in enumerate(agents):
            exp = Experience(state=state, action=actions[agent.idx],
```

```
                    reward=reward[agent.idx],
                    next_state=next_state, done=done)
        buffers[idx].insert(exp)

    if len(buffers[0]) >= replay_start_size:
        for agent_idx, agent in enumerate(agents):
            agent.model.optim.zero_grad()
            # Sample minibatch and compute loss
            minibatch = buffers[agent_idx].sample(mini_batch_size)
            loss = calc_loss(agent, minibatch, gamma, device=device)

            # perform training step
            loss.backward()
            agent.model.optim.step()

            if step_idx > 0 and step_idx % sync_rate == 0:
                agent.sync_models()

    state = next_state
```

B. Independent Actor-Critic

```
class IACAgent(Agent):
    "An Actor-Critic Agent"
    def act(self, obs):
        "Select an action based on an observation"

        unavailable_actions = self.env.get_unavailable_actions()[self]
        # compute the logits based on network policy model
        _, logits = self.model([obs])

        # for unavailable actions, set logits to very low value
        for action in unavailable_actions:
            logits[action.id] = -np.infty

        # create a prob distribution based on logits and sample from it
        action_idx = Categorical(logits=logits).sample().item()
        return self.actions[action_idx]

    def update(self, batch):
        "Perform one update step according to the IAC algorithm"
        _, actions, rewards, _, dones, observations, hidden,\
            next_obs, unavail = zip(*batch)

        # only perform updates on actions performed while alive
        self_alive = [idx for idx, obs in enumerate(observations) if obs[self].alive]

        rewards = torch.tensor([reward[self.team] for reward in rewards])

        # use the actor-critic network to compute V(s) and logits
        values, logits, h = self.model(observations, hidden)
        next_vals, _, _   = self.target(next_obs, h)

        # pick the actions that were performed by the agent self
        actions = torch.tensor([action[self] for action in actions])[self_alive]

        # set logits for unavailable actions to very low value
        for idx in self_alive
            logits[idx,:] = -np.infty

        # compute the target value and the resulting advantage value
        target = rewards + gamma * next_vals * (1.0 - dones)
        advantage = target.detach() - values.squeeze()
        advantage = advantage[self_alive_idx]
```

```
        # compute the log probabilities for the chosen actions
        log_prob = F.log_softmax(logits, dim=-1)
        log_prob_act = log_prob[self_alive_idx, actions]
        log_prob_act_val = advantage.detach() * log_prob_act

        # compute the entropy over the resulting policy distribution
        probs = F.softmax(logits, dim=1)
        entropy = -(probs * log_prob).sum(dim=1)
        loss_entropy = entropy.mean()

        # compute policy and value losses and the global loss function
        loss_pol = -log_prob_act_val.mean()
        loss_val = advantage.pow(2).mean()
        loss = loss_pol + loss_val - beta * loss_entropy

        # feed loss backward through computational grad and take update step
        self.model.optimizer.zero_grad()
        loss.backward()
        self.model.optimizer.step()

def train():
    # create the agents and put them into teams
    team_blue = [IACAgent() for _ in range(n_friends)]
    team_red  = [Agent()    for _ in range(n_enemies)]
    training_agents = team_blue
    agents = team_blue + team_red

    # create the environment
    env = Environment(agents, args)

    # generate the neural network model and assign it to the training agents
    models = generate_model(input_shape=n_inputs, n_actions=n_actions)
    for agent in training_agents:
        agent.set_models(models)

    # start the train sequence
    for step_idx in range(int(n_steps):
        batch = []
        for _ in range(n_episodes_per_step):
            # generate an episode and add it to the batch
            episode = generate_episode(env)
            batch.extend(episode)

        for agent in training_agents:
            agent.update(batch)
            if step_idx % 50 == 0: # sync target network
                agent.sync_models()
```

C. QMix

```
class QMIXAgent(Agent):

    def act(self, obs, test_mode=False):
        "Select an action based on an observation"
        unavail_actions = self.env.get_unavailable_actions()[self]
        avail_actions = [action for action in self.actions
                         if action not in unavail_actions]

        qvals, self.hidden_state = self.model(obs, self.hidden_state)
        # remove unavailable actions
        for action in unavail_actions:
            qvals[0][action.id] = -np.infty
        action_idx = qvals.max(1)[1].item() # pick position of maximum

        if test_mode: # when in test_mode, always return 'best' action
            return self.actions[action_idx]

        # epsilon-greedy action selection
        eps = self.scheduler()
        if random.random() < eps:
            return random.choice(avail_actions)
        else:
            return self.actions[action_idx]

class MultiAgentController:
    """ Controls the different learning agents
        and their update process
    """
    def __init__(self, env, agents, models):
        # ...

        # create mixing network
        self.mixer        = QMixer()
        self.target_mixer = copy.deepcopy(self.mixer)
        self.sync_networks()
        self.parameters = list(self.model.parameters())
        self.parameters += list(self.mixer.parameters())
        self.optimizer = torch.optim.Adam(self.parameters, lr=lr)

    def update(self, batch):
        "Perform one update step according to the QMix algorithm"
        batch_size = len(batch)
```

```
        states, next_states, observations, next_obs, hidden, next_hidden, actions,\
            rewards, dones, unavail = self._build_inputs(batch)

        # reshape observations & hidden states to push them through network models
        observations = observations.reshape(batch_size * len(self.agents), -1)
        next_obs = next_obs.reshape(batch_size * len(self.agents), -1)
        hidden = hidden.reshape(batch_size * len(self.agents), -1)
        next_hidden = next_hidden.reshape(batch_size * len(self.agents), -1)

        # compute & reshape estimated Q-values
        current_q_vals,   _ = self.model(observations, hidden)
        predicted_q_vals, _ = self.target(next_obs, next_hidden)
        current_q_vals = current_q_vals.reshape(batch_size, len(self.agents), -1)
        predicted_q_vals = predicted_q_vals.reshape(batch_size, len(self.agents), -1)

        # gather q-vals corresponding to the actions taken
        current_q_vals_actions = current_q_vals[range(batch_size * len(self.agents))
                                                , actions.reshape(-1)]


        predicted_q_vals[unavail==1] = -1e10 # set unavailable actions to low value

        # pick maximum Q-value for Bellman update
        predicted_q_vals_max = predicted_q_vals.max(2)[0]

        # use mixer to mix all Q-vals
        current_q_tot   = self.mixer(current_q_vals_actions, states)
        predicted_q_tot = self.target_mixer(predicted_q_vals_max, next_states)

        # Bellman update
        target = rewards + gamma * (1. - dones) * predicted_q_tot
        td_error = current_q_tot - target
        loss = (td_error ** 2).mean()

        # Perform update
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        return loss.item()

    def sync_networks(self):
        self.target.load_state_dict(self.model.state_dict())
        self.target_mixer.load_state_dict(self.mixer.state_dict())

def train():
    team_blue = [QMIXAgent() for idx in range(n_friends)]
    team_red  = [Agent()     for idx in range(n_enemies)]
    training_agents = team_blue
    agents = team_blue + team_red
    env = Environment(agents, args)
```

```
models = generate_models(n_inputs, n_actions)
for agent in training_agents:
    agent.set_model(models)

buffer = ReplayBuffer(size=buffer_size)
mac = MultiAgentController(env, training_agents, models)
for step_idx in range(n_steps):
    episode = generate_episode(env)
    buffer.insert_list(episode)
    if len(buffer) < batch_size:
        continue
    batch = buffer.sample(batch_size)

    loss = mac.update(batch)

    # synchronize target networks
    if step_idx % args.sync_interval == 0:
        mac.sync_networks()
```

Bibliography

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[FFA+18] Jakob N Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KT00] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[LBD+89] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[Lit94] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994.

[MBM+16] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[Nie15] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA:, 2015.

[OSV08] Frans A Oliehoek, Matthijs TJ Spaan, and Nikos Vlassis. Optimal and approximate q-value functions for decentralized pomdps. *Journal of Artificial Intelligence Research*, 32:289–353, 2008.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[RSDW+18] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: monotonic value function factorisation for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1803.11485*, 2018.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SLG+18] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning based on team reward. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 2085–2087. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[SMSM00] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[SRdW+19] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft Multi-Agent Challenge. *CoRR*, abs/1902.04043, 2019.

[Tan93] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

[Wat89] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[Wil92] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.