Zurich University
of Applied Sciences

School of
Engineering

InIT Institute of Applied
Information Technology

Bachelor Thesis (Computer Science)

Using Reinforcement Learning to Play Bomberman

Authors Joram Liebeskind
Silvan Wehner

Supervisors Oliver Dürr
Thilo Stadelmann

Date June 7, 2018

Zusammenfassung

Deep Reinforcement Learning (DRL) ist ein Bereich der künstlichen Intelligenz (KI) an dem sehr aktiv geforscht wird. DRL hat zum Ziel, einem Agenten beizubringen, sich in einer Umgebung zurechtzufinden. Deep learning ist ein Teilbereich der KI, der in den letzten Jahren viele Probleme besser lösen konnte als klassische Methoden. Reinforcement Learning (RL) ist ein allgemeiner Ansatz für maschinelles Lernens, welcher vergleichbar mit menschlichem Lernen ist. Die Kombination der beiden Methoden ist DRL und dieser wird das Potenzial zugeschrieben, eine generelle KI zu erschaffen. Die Umgebung in RL befindet sich in einem Zustand, welcher der Agent beobachtet. Führt der Agent eine Handlung aus, so kommt die Umgebung in einen neuen Zustand. Der Agent erhält ein Feedback für seine Handlung, was als Reward (Belohnung) bezeichnet wird. Mit einem RL-Algorithmus lernt ein Agent für jeden Zustand die Handlung auszuführen, welche ihm den maximalen Reward erbringt. Dieser wiederholte Vorgang etwas zu sehen, eine Handlung auszuführen, Feedback zu erhalten, eine neue Beobachtung zu erhalten und daraus Verhaltenverbesserungen abzuleiten, ist der Ansatz, welcher allen DRL-Methoden zugrunde liegt.

Das Ziel dieser Arbeit ist es, einem Agenten beizubringen Pommerman, ein Klon des Spiels Bomberman, der für die Forschung mit DRL erstellt wurde, zu spielen. Pommerman wird von vier Agenten gleichzeitig gespielt. Es gibt verschiedene Modi. In dem "Free for All" Modus gewinnt der Agent, der als letztes überlebt und in den Teammodi, bei welchem die vier Agenten in zwei Teams aufgeteilt, das Team mit den letzten Überlebenden. Diese Arbeit untersucht zwei Ansätze. Als Erstes wurde der Multi-Agent Deep Deterministic Gradient (MADDPG) Algorithmus implementiert, welcher verspricht solche Probleme gut lösen zu können. Dieser Versuch ist erfolglos, was wahrscheinlich daran liegt, dass MADDPG für Umgebungen mit kontinuierlichen Handlugen entwickelt wurde. In Pommerman können jedoch ausschliesslich diskrete Handlugen ausgeführt werden. Es werden auch Ansätze basierend auf Deep Q-Network (DQN) getestet. Die Resultate zeigen, dass der double-DQN Algorithmus mit Verwendung eines Convolutional Neural Network (CNN) in der Pommerman Umgebung Potenzial hat, wenn nur ein Agent lernt und die Anderen eine fest programmierte Strategie verwenden.

Abstract

Deep Reinforcement Learning (DRL) is a very active field of research in Artificial Intelligence (AI) which aims to teach agents to navigate in environments. DRL is so popular because it combines deep learning, which solves many real-world problems better than classical methods, with Reinforcement Learning (RL), which is an approach to learning that closely resembles how humans learn. DRL is viewed as having the potential to be the foundation for a general AI. The environment in RL is always in a state. The agent can observe that state and take an action, which changes the state of the environment. With each action the agent takes, the environment returns a feedback, informing the agent about how well it is doing. This feedback is referred to as the reward. RL algorithms aim to find out, how to navigate the environment in a way that maximises the reward. This loop of receiving an observation, taking an action, receiving a reward and a new observation and then learning from that to take better actions in the future is the approach that all RL algorithms are based on. For this reason, DRL is seen as having the potential to build a more general AI. This bachelor thesis aims to build an agent that can play Pommerman, a Bomberman clone made for DRL research. Pommerman is played by four agents at once. There is a purely adversarial mode, where only the last player alive wins and a mixed cooperative-competitive mode, where the four players are split into two teams. We explore two approaches to solving this complex environment. Initially, the Multi-Agent Deep Deterministic Gradient (MADDPG) algorithm is used, but without success. This is most likely due to the algorithm being designed for continuous action spaces, while the actions in Pommerman are discrete. Deep Q-Network (DQN) and later double-DQN algorithms were also used in a simplified environment, where only one agent is learning and the other agents have hard-coded strategies. Using double-DQN with a Convolutional Neural Network (CNN) and tuned hyperparameters has shown promising results, where the training agent in some cases won against the hard-coded agents.

Preface

We had our experiences with the subject of AI and in particular with deep learning, during the fall semester project thesis and the AI course. The topic of our project thesis of the fall semester 2017 was the exploration of an optimisation method of the training of Generative Adversarial Networks. We have developed a great interest in deep learning. The AI course was all about analysing, understanding and solving hard problems using methods which, in some way, yield intelligent behaviour, which lead us to the subject of RL.

We tinkered with DRL by trying to create an ai for the game 2048[1]. The general approach of DRL algorithms is to learn by trial and error, similar to the human approach to learning. This being different to both supervised and unsupervised learning, caught our interest and motivated us to do our bachelors thesis on the subject. That DRL is still a relatively young and very active field of research motivated us further.

In a first step we studied the fundamentals of DRL and found an environment that fit our needs. As this environment allows for multiple learning agents, we were lead to the MADDPG algorithm, which supposedly can solve such problems. We spent a lot of time on implementing MADDPG, which ultimately did not yield great results. DRL algorithms are known to be fragile. Even small bugs in the code base have significantly impacted the results of experiments. While experiments on buggy code helped fixing those bugs, their results were not helpful towards answering the thesis' core question.

Due to time constraints, we were not able to create an agent for multi-agent environments trained using self-play. We did however manage to create an agent that learns in our setting and has potential to perform well, if it is tweaked further and improved upon.

This project was an intense and interesting experience, in which we learnt a lot about DRL, but still leaves us hungry for more. Last but not least, we learnt how to systematically approach complex problems. Taking many small steps and analysing how each step affected our progress, helped us find a track that could potentially lead us to a great solution.

We would like to thank our supervisors Dr. Thilo Stadelmann and Dr. Oliver Dürr for helping us to better understand DRL, pointing us in the right direction on multiple occasions and encouraging us to stay persistent, when we were uncertain that our approaches were correct. We also want to thank Gabriel Eyyi for sharing with us what he learnt from his masters thesis in DRL and without whose advice we would not have come as far as we did. Last but not least, we are greatful to Uri Liebeskind and Josua Wehner for proofreading this thesis.

[1] https://gabrielecirulli.github.io/2048/

Zurich University
of Applied Sciences

DECLARATION OF ORIGINALITY

Bachelor's Thesis at the School of Engineering

By submitting this Bachelor's thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party. (Group works: the performance of the other group members are not considered as third party).

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Bachelor thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

City, Date:	Signature:
…………………………………………………………………	……………………………………………………………………………
	……………………………………………………………………………
	……………………………………………………………………………

The original signed and dated document (no copies) must be included after the title sheet in the ZHAW version of all Bachelor thesis submitted.

Contents

1. Introduction **1**
1.1. Motivation . . . 1
1.2. Problem Formulation . . . 2

2. Fundamentals **3**
2.1. Mathematical Notation . . . 3
2.2. Reinforcement Learning . . . 4
2.2.1. Markov Decision Process . . . 4
2.2.2. Finding an Optimal Policy . . . 7
2.2.3. Passive Temporal-Difference Learning . . . 8
2.2.4. Exploration . . . 9
2.2.5. Q-Learning . . . 9
2.3. Deep Q-Learning . . . 10
2.3.1. Policy Gradient Methods . . . 11
2.4. Multi-Agent Deep Deterministic Policy Gradient . . . 12

3. Related Work **15**

4. Methods **17**
4.1. Environment: Pommerman . . . 17
4.1.1. Modes . . . 17
4.1.2. Competitions . . . 18
4.1.3. Using The Environment . . . 18
4.2. Experiment Setup . . . 20

5. Experiments **25**
5.1. Experiments using Deep Q-Learning . . . 25
5.1.1. DQN Experiments 1 and 2: Simple Environment and Self-Play . . . 25
5.1.2. DQN Experiments 3 and 4: Increasing γ to 0.999 . . . 26
5.1.3. DQN Experiment 5: Reward Shaping . . . 27
5.1.4. DQN Experiment 6: Single-Agent Learning . . . 29
5.1.5. DQN Experiment 7: Increased Complexity . . . 30
5.1.6. DQN Experiment 8: Competition Experiment . . . 31
5.2. MADDPG-Agent Experiments . . . 33
5.2.1. MADDPG Experiments 1 and 2: MADDPG-Agents in Pommerman . . . 33

6. Conclusion **35**

7. Future Work **37**

Bibliography **41**

List of Figures **43**

List of Tables **45**

Acronyms **47**

A. Network Architectures **I**
A.1. OpenAI fully connected and MADDPG fully connected . . . I

A.2. OpenAI CNN and OpenAI CNN tanh I

B. Experiment Hyperparameters **III**
B.1. OpenAI Baselines DQN-Agent Hyperparameters III
B.2. MADDPG-Agent Hyperparameters VIII

C. Preliminary Experiment **XI**

D. DQN Experiment 9: Changing the Activation Function **XIII**

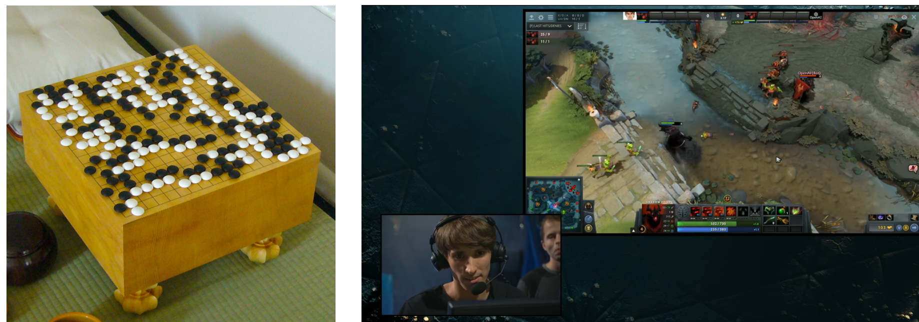
E. Experiment Hard- and Software **XV**

1. Introduction

1.1. Motivation

Deep Reinforcement Learning (DRL) is one of the most talked about topics in Artificial Intelligence (AI). This started with DeepMinds Nature publication [Mni+15], where they introduced an algorithm which can learn to play Atari games better than humans can. The algorithm called Deep Q-Network (DQN) did not require any prior knowledge. It learnt to play solely by receiving an input image of the game state and a reward and was the first algorithm to do so successfully on a wide range of tasks and without prior knowledge about the task.

The next big splash was once again by DeepMind, when they built an AI named AlphaGo for a game called Go. Go is a strategy board game for two players, where both try to surround more area of the game board than the opponent. A game of Go in progress is shown in figure 1.1a. Building an AI for Go was regarded as a particularly hard challenge, since there are approximately 250 valid moves per position and games last for around 150 moves. An AI based on exhaustive search would have to visit up to $250^{150} \approx 4.91 \cdot 10^{350}$ states, which is infeasible. AlphaGo was the first Go AI to beat professional players. By using Deep Neural Networks (DNNs) to guide search and evaluating board position values, they were able to achieve levels of play that were thought to take decades for an AI to reach [Sil+17b].



(a) A game of Go in progress [Gob07].

(b) Professional Dota 2 player Danil Ishutin competing against OpenAIs agent. Screenshot taken from [Ope17a].

Figure 1.1.: Depictions of the two mentioned games Go and Dota 2.

OpenAI also impressed the DRL community by creating an AI that was able to beat the best professional Dota 2 players in the world. Dota 2 is an online real time strategy game where two teams of five players each compete against each other. The details about the implementation of AI are still unknown however, as the agent was limited to a simplified version of the game and OpenAI first wants to create a full team of AIs, before they publish their methods. It is known however that they trained their agents purely through self-play [Ope17b], which makes this work particularly interesting.

DRL is about teaching agents to navigate an environment, for instance to play a game or drive a vehicle. The agent receives an observation and chooses an action based on that observation. The environment returns feedback, or reward, to the agent. The agent then learns to change its behaviour so that it can maximise the reward received by the environment. This is comparable to

how humans learn by trial and error. For that reason, DRL is often attributed the potential to be the foundation of a general AI.

This work focuses on environments in a multi-agent setting. The environment is constantly changing from a single agents point of view, as the other agents keep changing their behaviour through their own learning progress. Due to the interesting nature of this challenge, researches are very active in the field [Yu18]. Whilst some papers show very promising results, there are still no algorithms that can reliably solve a wide range of tasks in multi-agent settings, which motivates this thesis to train an agent to reliably solve such a task.

1.2. Problem Formulation

What DRL based approaches enable an agent to play Bomberman in a competitive multi-agent setting? This is the question we seek to answer in this bachelor thesis.

This requires to define an environment that allows for a Reinforcement Learning (RL) setup and is similar to Bomberman. Algorithms have to be evaluated on the environment in order to find one that shows the most potential play the game well. Once an algorithm is decided on, parameters are to be tweaked systematically in order to get the best possible performance from the algorithm.

The question is answered to satisfaction when a combination of algorithm, network architecture and parameters is found that learns to play in the environment well enough that visual observation shows strategic and (artificially) intelligent behaviour.

2. Fundamentals

This chapter serves as a summary of the theory that is required to understand this thesis thoroughly. *Recommended ressources* are referred to throughout this chapter and the reader is encouraged to study these before continuing with the rest of this thesis. This thesis presupposes knowledge of the inner workings of Artificial Neural Networks (ANNs), which are used as general nonlinear function approximators in the remaining chapters. To readers unfamiliar with the subject of deep learning, we recommend the following online and offline resources:

- The textbook “Deep Learning” by Ian Goodfellow, Yoshua Bengio and Aaron Courville. [GBC16] This book is also published online for free at `https://deeplearningbook.org`.
- Video Series “Neural networks” on YouTube by Grant Sanderson [San]: `https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi`
- Online book “Neural Networks and Deep Learning” by Michael Nielsen [Nie15].

2.1. Mathematical Notation

This section describes the mathematical notation used throughout this work. It is taken from the book “Deep Learning” [GBC16, Chapter “Notation”].

Numbers and Arrays

a	Scalar
$\boldsymbol{a}$	Vector
$\boldsymbol{A}$	Matrix
a	Scalar random variable
$\mathbf{a}$	Vector-valued random variable

Sets, Graphs and Tuples

$\mathbb{A}$	Set
$\{0,1\}$	Set containing 0 and 1
$[a,b]$	The real interval including a and b
$(a,b]$	The real interval excluding a but including b
$\mathcal{G}$	Graph
$\langle a,b,c \rangle$	A tuple containing scalars a,b and c

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$

Probability Theory

$P(a)$	A probability distribution over a variable with discrete values
$p(a)$	A probability distribution over a variable with continuous values, or over a variable whose type has not been specified.
$a \sim P$	Random variable a has probability distribution P
$\mathbb{E}_{\mathrm{x} \sim P}[f(x)]$	Expectation of $f(x)$ with respect to $P(\mathrm{x})$
$\mathbb{E} f(x)$	

Functions

$\sigma(x)$	Logistic sigmoid, $\frac{1}{1+\exp(-x)}$
$\lvert\lvert\boldsymbol{x}\rvert\rvert_p$	L^p norm of $\boldsymbol{x}$
$\lvert\lvert\boldsymbol{x}\rvert\rvert$	L^2 norm of $\boldsymbol{x}$

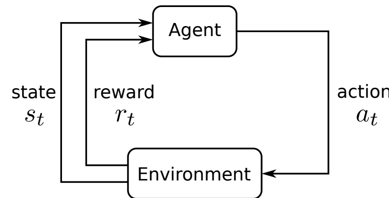


Figure 2.1.: The feedback loop that RL is based on. An agent receives a state observation and a reward. The agent sends actions to the environment based on the observation and learns to maximise its reward by adjusting its state-action association. Adapted from [SB18, p. 48].

2.2. Reinforcement Learning

An introductory story A chef is hired at a restaurant to bake cakes. He is ambitious and motivated to make the most delicious cakes ever. His ultimate goal is to make the customers happy. There is only one problem: he has never baked a cake before. Since the chef initially does not know where to start, he decides to mix random ingredients and figure things out on the go. At first, things go terribly wrong. He tries to put pure flour in a hot pan, and all he gets in return is a bad smell. Moreover, after he added some uncooked eggs to the now burnt flour, the customers' feedback was not good at all. The chef decided to reflect on his new experiences and tried to determine which of his actions lead to such an adverse outcome. He repeated this process of trying something and improving his skills based on his customers' feedback. Soon the restaurant was known for the best cakes in town, and the chef continued to improve steadily. He felt lucky that his boss was so patient with him.

In other words, RL is a class of algorithms that allow agents to learn how to navigate an environment in a way that allows them to collect the maximum reward. Agents can choose from a set of actions within an environment. The agents receive a reward as the environment changes its state. A reward for an action can be delayed, so actions cannot be directly associated with the reward received in the time-step. This feedback loop is visualised in figure 2.1. This general framework makes reinforcement learning interesting, as it allows agents to learn complex tasks in a wide range of environments. This chapter formalises reinforcement learning and its elements.

2.2.1. Markov Decision Process

Recommended Ressources

- Chapter 3: "*Finite Markov Decision Processes*" in [SB18]:
  - Online draft:
    `https://drive.google.com/file/d/1xeUDVGWGUUv1-ccUMAZHJLej2C7aAFWY/view`
- Lecture 2: "*Markov Decision Process*" in [Sil15, Lecture 2]:
  - Slides: `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MDP.pdf`
  - Video: `https://www.youtube.com/watch?v=lfHX2hHRMVQ`
- Chapter 17: "*Making Complex Decisions*" Section 17.1 in [RN16].

The Markov Decision Process (MDP) allows formalising environments, the states an environment can be in and what an agent experiences when traversing the environment. Formally, this means:

Definition 2.2.1 (Markov Decision Process)**.**
The Markov Decision Process [Sil15, Lecture 2], is a quintuple $\langle \mathsf{S}, \mathsf{A}, P, R, \gamma \rangle$, where:

S Is a finite set of states s which fully describe the environment at a time step.

A Is a finite set of actions a. Taking an action usually results in the environment changing to successor state $s' \in \mathsf{S}$.

P Is a state transition model $P(\mathrm{s}_{t+1} = s' \mid \mathrm{s}_t = s, \mathrm{a}_t = a)$ which denotes the probability of entering state s' when being in state s and taking action a. For reasons of readability, random variables like s_{t+k} will often be omitted.

R Is a reward function $R(s)$, which denotes the reward received for state s.

γ Is a discount factor $\gamma \in [0, 1]$.

Markov Property states that an environments state is fully described by its current state s: The probability of entering state s depends on its parents only and not on any history of states before that [Sil15, Lecture 2]:

$$P(\mathrm{s}_{t+1} | s_t, a_t) = P(\mathrm{s}_{t+1} | s_1, a_1, \ldots, s_t, a_t) \tag{2.1}$$

Solving an MDP [Sil15, Lecture 2] refers to finding a way to traverse the MDP that maximises the return G_t.

Definition 2.2.2 (Return)**.**
The return G_t is the total discounted reward from time-step t. [Sil15, Lecture 2]

$$G_t = R(s_t) + \gamma R(s_{t+1}) + \cdots = \sum_{k=0}^{\infty} \gamma^k R(s_{t+k}) \tag{2.2}$$

The discount factor γ is a way to define how important rewards in the far future are. A discount factor close to one means that rewards far in the future are of high importance, values close to zero mean that only next few steps are relevant. There are numerous reasons as to why this is done: In a circular MDP with positive rewards, the agent could, in theory, achieve an infinite return. This is mathematically inconvenient to compute. With $\gamma < 1$, a maximum return is always guaranteed. Using a discount factor also allows to model natural behaviours, where rewards in the near future are often preferred over rewards which are many time-steps away [Sil15, Lecture 2].

State-value function [Sil15, Lecture 2] If there was a way to measure the value of a state, it would be easier to solve the MDP. This idea introduces the state-value function $v(s)$, which is the expected discounted future reward when being in state s_t:

$$v(s_t) = \mathsf{E}[G_t \mid \mathrm{s}_t = s] \tag{2.3}$$

Given the definition of G_t in equation 2.2, the state-value function $v(s)$ can be reformulated.

$$\begin{aligned}
v(s) &= \mathsf{E}[\mathrm{G}_t \mid \mathrm{s}_t = s] && (2.4)\\
&= \mathsf{E}[R(\mathrm{s}_t) + \gamma R(\mathrm{s}_{t+1}) + \gamma^2 R(\mathrm{s}_{t+2}) + \cdots \mid \mathrm{s}_t = s] && (2.5)\\
&= \mathsf{E}[R(\mathrm{s}_t) + \gamma (R(\mathrm{s}_{t+1}) + \gamma R(\mathrm{s}_{t+2}) + \cdots) \mid \mathrm{s}_t = s] && (2.6)\\
&= \mathsf{E}[R(\mathrm{s}_t) + \gamma G(\mathrm{s}_{t+1}) \mid \mathrm{s}_t = s] && (2.7)\\
&= \mathsf{E}[R(\mathrm{s}_t) + \gamma v(\mathrm{s}_{t+1}) \mid \mathrm{s}_t = s] && (2.8)
\end{aligned}$$

This decomposition into two parts, the immediate reward $R(s_t)$ and the discounted value of the successor state $\gamma v(s')$, is a *Bellman equation*. A Bellman equation always consists of two parts, based on a series of choices. The first part is a value for an initial choice plus the second part, which is the value for the rest of the series of choices.

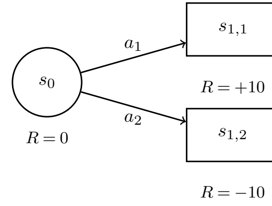


Figure 2.2.: A very simple MDP, three states, two terminal states $s_{1,1}$ and $s_{1,2}$ and discount factor $\gamma = 1$.

A Policy is what defines an agents behaviour. It is common to use the symbol π to refer to the policy. A policy is either stochastic $\pi(a|s)$, which is a distribution over states given actions or deterministic $a = \pi(s)$ which returns which action $a \in \mathsf{A}$ is to be chosen in state $s \in \mathsf{S}$ and therefore is defined as $\pi : \mathsf{S} \mapsto \mathsf{A}$ [RN16, p. 647].

State-value function with policy The state-value function $v(s)$ returns the long-term value of being in state s. But the value of a state changes when following a policy. Consider the example MDP in figure 2.2.

When not following a policy and the actions are chosen equally likely, the value of state s_0 is:

$$v(s_0) = R(s_0) + \gamma \frac{1}{2} v(s_{1,1}) + \gamma \frac{1}{2} v(s_{1,2}) = 0 + 1 \frac{1}{2} 10 + 1 \frac{1}{2}(-10) = 0 \tag{2.9}$$

When following a greedy policy π_g, which always visits the successor state s' with maximum value then $v_{\pi_g}(s_0)$ is:

$$v_{\pi_g} = R(s_0) + \gamma v(\pi_g(s_0)) = 0 + 1 \cdot 10 = 10 \tag{2.10}$$

And finally when following a bad policy π_b that for some arbitrary reason always takes action a_2 then $v_{\pi_b}(s_0) = -10$.

Equation 2.11 is a more general definition of the state-value function under a policy π: It gives the value of state s when following policy π.

$$v_\pi(s) = \mathsf{E}_\pi[R(\mathrm{s}_t) + \gamma v_\pi(s_{t+1}) \mid \mathrm{s}_t = s] \tag{2.11}$$

The action-value function [Sil15, Lecture 3] $q(s,a)$ extends the state-value function by taking the action taken at time step t into account: it returns of the value of taking action a in state s_t:

$$q_\pi(s,a) = \mathsf{E}_\pi[R(\mathrm{s}_t) + \gamma q_\pi(\mathrm{s}_{t+1}, \mathrm{a}_{t+1}) \mid \mathrm{s}_t = s, \mathrm{a}_t = a] \tag{2.12}$$

It expresses the value of taking action a in state s and then following policy π.

Optimal value functions [Sil15, Lecture 3] If it was possible to iteratively evaluate all policies, then it would be possible to find an optimal state-value function $v_*(s)$, which is the maximum state-value function over all policies:

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.13}$$

and an optimal action-value function $q_*(s,a)$, which is the maximum action-value function over all policies:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.14}$$

2.2.2. Finding an Optimal Policy

Recommended Ressources

- Chapter 4: "*Finite Markov Decision Processes*", Sections 4.1 - 4.4 in [SB18]:
  - Online draft:
    `https://drive.google.com/file/d/1xeUDVGWGUUv1-ccUMAZHJLej2C7aAFWY/view`.
- Lecture 3: "*Planning by Dynamic Programming*" in [Sil15]:
  - Slides: `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/DP.pdf`.
  - Video: `https://www.youtube.com/watch?v=Nd1-UUMVfz4`.
- Chapter 17: "*Making Complex Decisions*" Sections 17.2 - 17.3 in [RN16].

This section is shortened from [RN16, Section 17.1.2]. An optimal policy π_* is therefore any policy which can achieve both the optimal state-value function $v_{\pi_*}(s) = v_*(s)$ and the optimal action-value function $q_{\pi_*}(s,a) = q_*(s,a)$.

Value Iteration

The value iteration algorithm can find an optimal policy by calculating the value for each state. The policy then chooses the optimal action in each state based on these state values. For each state, the optimal value is defined by:

$$v_*(s) = R(s) + \gamma \max_{a \in \mathsf{A}} \sum_{s'} P(s'|s,a) v(s') \tag{2.15}$$

When executing the value iteration algorithm, the state-value functions $v_0, v_1, \ldots$ form a series of state-value functions v_k that converges to v_* for $k \to \infty$. To get from v_k to v_{k+1}, a **Bellman update** is applied:

$$v_{k+1}(s) = R(s) + \gamma \max_{a \in \mathsf{A}} \sum_{s'} P(s'|s,a) v_k(s') \tag{2.16}$$

This is done synchronously for each state, meaning that for the entire iteration, the right hand side of the computation uses $v_k(s')$ and not already newly computed values $v_{k+1}(s')$. This is repeated until the action-value function converges to a fixed point, that being v_*.

Policy Iteration

Policy iteration executes two steps in turns:

- Policy Evaluation
- Policy Improvement

Policy Evaluation is very similarly to value iteration: It finds the value function v_π by starting from a value function v_0 and synchronously applying a Bellman update to $v_k(s)$, but this time incorporating the policy π:

$$v_{k+1}(s) = R(s) + \gamma \sum_{s'} P(s'|s,\pi_k(s)) v_k(s') \tag{2.17}$$

Policy Improvement The policy is then updated greedily: For each action, it always chooses the action which leads to the state of maximum possible value as indicated by the state-value function v_{π_k}.

$$\pi_{k+1}(s) = \underset{a \in \mathbb{A}}{\operatorname{argmax}} \sum_{s'} P(s'|s,a) v_{\pi_k}(s') \tag{2.18}$$

Both steps *policy iteration* and *policy evaluation* are done synchronously, just like in the value-iteration algorithm. This procedure is repeated until the policy remains unchanged. If the policy remains unchanged it can no longer be improved and is therefore as good as the optimal policy π_*.

2.2.3. Passive Temporal-Difference Learning

Recommended Ressources

- Chapter 6: "*Temporal-Difference Learning*", Sections 6.1 - 6.3 in [SB18]:
  - Online draft:
    `https://drive.google.com/file/d/1xeUDVGWGUUv1-ccUMAZHJLej2C7aAFWY/view`.
- Chapter 21: "*Reinforcement Learning*" Section 21.2.3 in [RN16].

This section is a summary of [RN16, Section 21.2.3]. The problem with algorithms like value iteration or policy iteration is that they require knowledge of the state transitions of an MDP. $P(s'|s,a)$ provides that information and is fundamentally required in order to do the Bellman updates. In the real world, however such a state transition model is often not available and impractical to reconstruct manually. Methods are required that can converge to the optimal state-value function $v_*(s)$ or can find an optimal policy π_*.

The state transition probabilities give a weight to successor state values, based on how likely those successor states are to occur.

Successor states that are rarely visited don't contribute significantly to the expected future return of s. Temporal-Difference (TD) approximates this behaviour by averaging over many percepts in trials. A *trial* is a series of state and reward tuples, that an agent experiences while going from a starting state to a terminal state. Such a tuple of state and reward $\langle s, r \rangle$ is called a *percept* or *experience*. The term *Passive* indicates that an algorithm finds a value function $v_\pi(s)$ with a fixed policy π. The passive TD-learning algorithm updates a table of values for each state:

$$v_\pi(s) \leftarrow v_\pi(s) + \alpha(r + \gamma v_\pi(s') - v_\pi(s)) \tag{2.19}$$

Adding a learning rate α allows this equation to slowly converge to the true state-value function v_π. The full passive-TD algorithm, which is shown in [RN16, figure 21.4. p. 837]. The algorithm can be extended by adding a weight $N_s(s)$. N_s is a table which keeps track of how often a given state was visited. It is used to add higher importance to state-values of successor states that are visited more frequently:

$$v_\pi(s) \leftarrow v_\pi(s) + \alpha N_s(s)(r + \gamma v_\pi(s') - v_\pi(s)) \tag{2.20}$$

Methods that use temporally successive states to approximate a value function are temporal-difference methods.

2.2.4. Exploration

Recommended Ressources

- Chapter 5: "*Monte Carlo Methods*", Section 5.5 in [SB18]:
  - Online draft: `https://drive.google.com/file/d/1xeUDVGWGUUv1-ccUMAZHJLej2C7aAFWY/view`.
- Chapter 21: "*Reinforcement Learning*" Section 21.3.1 in [RN16].

This section is a summary of [RN16, Section 21.3.1]. So far it was not addressed how an agent should go about choosing actions in trials during training. It seems to make sense always to follow the policy. Since RL algorithms are finding optimal state-value functions, action-value functions or optimal policies, why not just follow what the policy recommends? At first, the policy does not know which action will lead to the maximum return. Once the value function finds rewards for an action in a state and those rewards turn out to be higher than the default values of non-visited, the policy will not select actions that lead to the non-visited states, as it expects the known paths with marginally higher value estimations to yield a higher return. Without deviating from what is known, much like in real life, the policy will only rarely find ways that maximise the return.

Greedily following the policy is called *exploitation*. In contrast to exploitation, *exploration* refers to a way of choosing an action different from what the policy recommends and by doing so finding new paths that yield a higher return. Choosing between exploration and exploitation is always a trade-off. One way to motivate exploration is to use a function $f(u,n)$, with u being the expected future reward for an action a $\sum_{s' \in \mathsf{S}} P(s'|s,a)v_\pi(s')$ and n the number of times the state-action pair s,a was visited. A simple function of f would then be:

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases} \tag{2.21}$$

where R^+ is the best possible reward obtainable by any state and N_e is the number of times the state-action pair s,a has been visited.

2.2.5. Q-Learning

Recommended Ressources

- Chapter 6: "*Temporal-Difference Learning*", Section 6.5 in [SB18]:
  - Online draft:
    `https://drive.google.com/file/d/1xeUDVGWGUUv1-ccUMAZHJLej2C7aAFWY/view`.
- Chapter 21: "*Reinforcement Learning*" Section 21.3.2 in [RN16].

This section is a summary of [RN16, Section 21.3.2]. Q-Learning aims to learn an action-value function as in equation 2.22.

$$Q(s,a) = R(s) + \gamma \sum_{s' \in \mathsf{S}} P(s'|s,a) \max_{a' \in \mathsf{A}} Q(s',a') \tag{2.22}$$

This equation once again requires a state transition model $P(s'|s,a)$. Using the TD-approach, equation 2.22 can be rewritten to equation 2.23. Similarly to equation 2.20, $N_{sa}(s,a)$ is a table keeping track of counts, but this time it records the number of occurrences of state-action pairs instead of just tracking states.

$$Q(s,a) \leftarrow Q(s,a) + \alpha N_{sa}(s,a)(R(s) + \gamma \max_{a' \in \mathsf{A}} Q(s',a') - Q(s,a)) \tag{2.23}$$

Algorithm 2.1 Q-learning Algorithm as shown in [RN16] on page 844, figure 21.8

```
1: function Q-LEARNING-UPDATE(percept) returns an action
2:     inputs: percept, a percept indicating the current state s'.and reward r'
3:     persistent: Q, a table of action values indexed by state and action, initially zero,
4:                 N_sa, a table of visitation counts for state-action pairs, initially zero
5:                 s, a, r, the previous state, action and reward, initially null
6:     if Terminal?(s') then
7:         Q(s', None) ← r'
8:     end if
9:     if s is not null then
10:        increment N_sa(s, a)
11:        Q(s,a) ← (Q(s,a) + α N_sa(s,a)(r + γ max_{s'∈A} Q(s',a') − Q(s,a)
12:    end if
13:    s, a, r ← s', argmax_{a∈A} f(Q(s',a'), N_sa(s',a')), r'
14:    return a
15: end function
```

Q-learning is an *off-policy* algorithm: for the target value $r + \gamma \max_{s' \in \mathbb{A}} Q(s', a')$ it uses the best value as suggested by the Q-function instead of using the action a' that was chosen by the policy in states$'$. This is in contrast to *on-policy* algorithms, which do train using the action a' which was actually chosen by π. One example for such an algorithm is SARSA, which uses tuples of $\langle s, a, r, s', a' \rangle$ directly retained from trials.

2.3. Deep Q-Learning

Recommended Ressources

- Nature paper: "*Human-level control through deep reinforcement learning*" [Mni+15].
- Chapter 16: "*Applications and Case Studies*", Section 16.5 in [SB18]:
  - Online draft: `https://drive.google.com/file/d/1xeUDVGWGUUv1-ccUMAZHJLej2C7aAFWY/view`.

DeepMind introduced in their landmark Nature publication [Mni+15] a new algorithm called *Deep Q Learning*. This algorithm was used for the experiments of this thesis. The algorithm uses a DNN to approximate the optimal action-value function:

$$Q^*(\boldsymbol{s}, a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \dots \mid \mathrm{s}_t = \boldsymbol{s}, \mathrm{a}_t = a, \pi] \tag{2.24}$$

which maximises the expected discounted future reward over all policies $\pi = P(a|s)$ starting at time step t, after making an observation $\boldsymbol{s}$ and taking an action a. The Q-network, with parameters θ, represents an approximation $Q_\theta(a, \boldsymbol{s}) \approx Q^*(a, \boldsymbol{s})$ of the optimal action-value function. To improve the stability of the training, the paper introduces two new elements:

Experience Replay The experience replay stores tuples $e_t = \langle \boldsymbol{s}_t, a_t, r_t, \boldsymbol{s}_{t+1} \rangle$ where:

- $\boldsymbol{s}_t$: state
- a_t: action chosen at time step t
- r_t: reward received after exiting state $\boldsymbol{s}_t$
- $\boldsymbol{s}_{t+1}$: successor state after choosing action a_t in state $\boldsymbol{s}_t$.

Experiences e_t are stored in an experience replay $\mathbb{D}_t = \{e_1, e_2, \dots, e_t\}$. The Q-network is updated using mini-batches of samples of experiences $\langle \boldsymbol{s}, a, r, \boldsymbol{s}' \rangle \sim U(\mathbb{D})$, drawn uniformly at random. Using an experience replay as described, creates a smoothing over changes in the data distribution and as such makes training more stable.

Target Network Using a target network $\hat{Q}$ with parameters θ^- to predict target values $y = r + \gamma \max_{a'} \hat{Q}_{\theta^-}(\boldsymbol{s}', a')$ reduces the correlation of target values y and predicted Q-values $Q_\theta(\boldsymbol{s}, a)$. The target network parameters are periodically updated to $\theta^- \leftarrow \theta$.

Loss The Q-network has to predict the target values $y = r + \gamma \max_{a'} \hat{Q}_{\theta_i^-}(\boldsymbol{s}', a')$, which is intuitively related to the value-iteration formulation. Thus, the loss to be minimised is:

$$L(\theta) = \mathbb{E}_{\langle \boldsymbol{s}, a, r, \boldsymbol{s}' \rangle \sim U(\mathbb{D})}[(y - Q_\theta(\boldsymbol{s}, a))^2 \mid \boldsymbol{s}, a] \tag{2.25}$$

Deep Q-Algorithm The deep Q-learning algorithm expects to receive some observation $\boldsymbol{o}$ and stores this in a sequence $\boldsymbol{s}$. For each step, the a new sequence $\boldsymbol{s}_{t+1} = \{\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{o}_{t+1}\}$ is created. This sequence serves as part of the input to the Q-networks. Since it is impractical to use inputs of arbitrary lengths for ANNs, a function $\phi(\boldsymbol{s})$ pre-processes sequences and maps them to a suitable input of fixed length.

Algorithm 2.2 Deep Q-learning with experience replay.

1: Initialise replay memory $\mathbb{D}$ to capacity N.
2: Initialise action-value function Q with random weights θ.
3: Initialise target action-value function $\hat{Q}$ with weights $\theta^- = \theta$.
4: **for** *episode* $= 1 \dots M$ **do**
5: Initialise sequence $\boldsymbol{s}_1 = \{\boldsymbol{o}_1\}$ and pre-process sequence $\boldsymbol{\phi}_1 = \phi(\boldsymbol{s}_1)$
6: **for** $t = 1 \dots T$ **do**
7: With probability ϵ select a random action a_t
8: otherwise select $a_t = \text{argmax}_a Q_\theta(\boldsymbol{\phi}(s_t), a)$
9: Execute action a_t in environment and observe reward r_t and image o_{t+1}
10: Set $\boldsymbol{s}_{t+1} = \boldsymbol{s}_t, a_t, \boldsymbol{o}_{t+1}$ and preprocess $\boldsymbol{\phi}_{t+1} = \boldsymbol{\phi}(s_{t+1})$
11: Store transition $\langle \boldsymbol{\phi}_t, a_t, r_t, \boldsymbol{\phi}_{t+1} \rangle$ in $\mathbb{D}$
12: Set $\begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}_{\theta^-}(\boldsymbol{\phi}_{j+1}, a') & \text{otherwise} \end{cases}$
13: Perform gradient descent step on $(y_j - Q_\theta(\boldsymbol{\phi}_j, a_j))^2$ with respect to the network parameters θ
14: Every C steps reset $\theta^- \leftarrow \theta$.
15: **end for**
16: **end for**

2.3.1. Policy Gradient Methods

Recommended Ressources

- Chapter 13: "*Policy Gradient Methods*", Sections 13.1 - 13.2 in [SB18]:
  - Online draft: `https://drive.google.com/file/d/1xeUDVGWGUUv1-ccUMAZHJLej2C7aAFWY/view`.
- Chapter 21: "*Reinforcement Learning*" Section 21.5 in [RN16].

This section is a summary of [RN16, Section 21.5]. Policy Gradient (PG) methods are discussed only briefly here. While they are essential, especially to continuous control problems, this thesis mainly uses methods based on Q-Learning.

Instead of using deterministic policies like $\pi(s) = \max_a Q(s,a)$ we now consider stochastic policies $\pi_\theta(s,a)$, which return the probability of choosing action a in state s. π_θ may use any function approximator, like an ANN, parametrised by weights θ. PG-methods aim to compute a gradient for the parameters θ which when followed, improves the performance of π_θ. To compute a gradient, π_θ has to be differentiable. A popular representation is the softmax function, with $Q_\theta(s,a)$ being the standard action-value function but using a function approximator with parameters θ:

$$\pi_\theta(s,a) = \frac{\exp(Q_\theta(s,a))}{\sum_{a' \in \mathsf{A}} \exp(Q_\theta(s,a'))} \tag{2.26}$$

Let $\rho(\theta)$ be the *policy-value*, which gives the expected return when executing policy π_θ. To improve the policy, we would then just need to compute and follow the gradient $\nabla_\theta \rho(\theta)$. This is is what is called the *policy gradient*. Since often a function like $\rho(\theta)$ is not available, the empirical gradient can be computed.

In the simple case of a nonsequential environment, where the reward $R(s)$ is obtained immediately after executing action a in state s_0, the policy value is just the expected value of the reward:

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_{a \in \mathsf{A}} \pi_\theta(s_0,a) R(a) = \sum_{a \in \mathsf{A}} (\nabla_\theta \pi_\theta(s_0,a)) R(a) \tag{2.27}$$

This can be approximated by samples generated from the policy π_θ. Let N be the number of trials and a_j the jth action taken:

$$\nabla_\theta \rho(\theta) = \sum_{a \in \mathsf{A}} \pi_\theta(s_0,a) \frac{\nabla_\theta \pi_\theta(s_0,a) R(a)}{\pi_\theta(s_0,a)} \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s_0,a_j)) R(a_j)}{\pi_\theta(s_0,a_j)} \tag{2.28}$$

The policy gradient is approximated by a sum of terms involving the gradient of action-selection probability in each trial. This can be generalised to equation 2.29 for the case of sequential environments:

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s,a_j)) R_j(s)}{\pi_\theta(s,a_j)} \tag{2.29}$$

where a_j is the action executed in state s on the jth trial and $R_j(s)$ gives the total reward received from state s onward in the jth trial.

2.4. Multi-Agent Deep Deterministic Policy Gradient

Recommended Ressources

- Publication: "*Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments*" [Low+17]

This section is a summary of [Low+17]. Multi-Agent Deep Deterministic Gradient (MADDPG) is based on Deep Deterministic Policy Gradient (DDPG) and is a PG method. Methods like this use two networks, the *actor* and the *critic*. The critic is equivalent to the Q-function introduced earlier. It delivers the expected return given a state and an action. The policy network is referred to as the actor. It is updated by computing and following the gradient of the actors' parameters with respect to the critics output. Figure 2.3

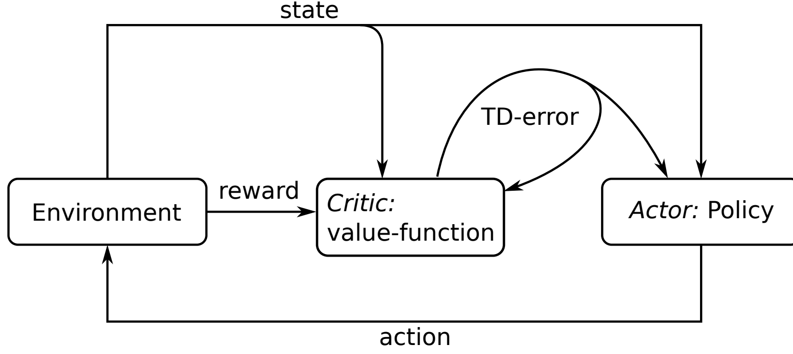


Figure 2.3.: The basic setup of actor-critic methods. The critic learns how good state-action pairs are and the critic maximises the reward predicted by the critic. Adapted from [Mar05].

.

For deterministic policies $\mu_\theta(a|\boldsymbol{s})$, the gradient for updating the actor parameters is written in equation 2.30, where D is a replay memory containing tuples $\langle \boldsymbol{s}, a, r, \boldsymbol{s}' \rangle$.

$$\nabla_\theta J(\theta) = \mathsf{E}_{\boldsymbol{s}\sim\mathsf{D}}[\nabla_\theta \mu_\theta(a|\boldsymbol{s}) \nabla_a Q^\mu(\boldsymbol{s}, a) \mid a = \mu_\theta(a)] \tag{2.30}$$

Q-functions have a hard time learning in an environment, where other agents policies are non-stationary. In multi-agent environments, the policy of each agent continually changes as it learns. To properly handle multiple learning agents, the critic is modified. Instead of using only the observation of the agent that the critic belongs to, the critic is fed with an observation that fully describes the state of the environment and the actions of all agents in that state. Figure 2.4 shows how the different components of MADDPG are connected.

Let $\boldsymbol{o}_i$ describe the observation of agent i, $\boldsymbol{x}$ state information, for example $\boldsymbol{x} = \{\boldsymbol{o}_1, \ldots, \boldsymbol{o}_N\}$, $\boldsymbol{\mu} = \{\mu_1, \ldots, \mu_N\}$ a set of all policies, $Q_i^{\boldsymbol{\mu}}$, a_i, $\boldsymbol{\mu}_i$ and θ_i the critic, action, policy and parameters of agent i. The replay memory D contains tuples $\langle \boldsymbol{x}, \boldsymbol{x}', a_1, \ldots, a_N, r_1, \ldots, r_N \rangle$ This leads to the following policy gradient:

$$\nabla_{\theta_i} J(\boldsymbol{\mu}_i) = \mathsf{E}_{\boldsymbol{x},a\sim\mathsf{D}}[\nabla_{\theta_i} \boldsymbol{\mu}_i(a_i|\boldsymbol{o}_i) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\boldsymbol{x}, a_1, \ldots, a_N) \mid a_i = \boldsymbol{\mu}_i(\boldsymbol{o}_i)] \tag{2.31}$$

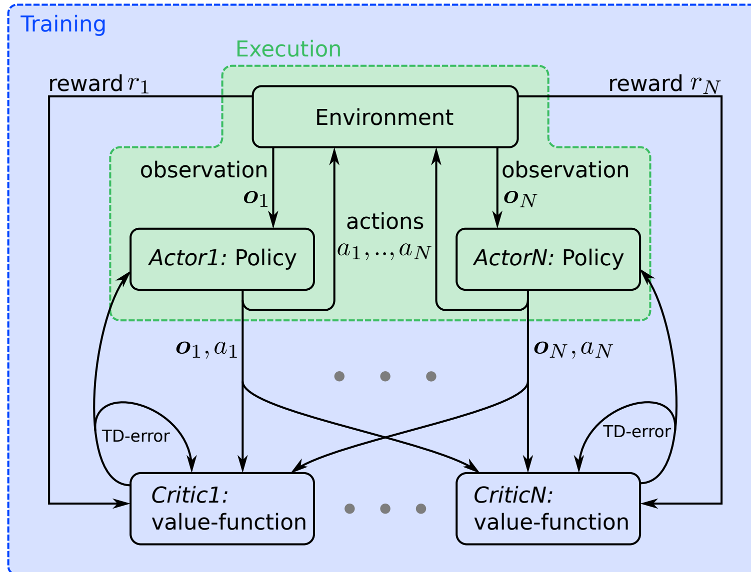


Figure 2.4.: The setup of MADDPG. The critics receive all inputs and observations of all actors and base value predictions on that information. This is only done during training. Only the critics are used in during execution. Adapted from [Low+17].

Algorithm 2.3 Multi-Agent Deep Deterministic Policy Gradients for N agents [Low+17, p. 13]

1: **for** episode = 1 to M **do**
2: Initialise a random process $\mathcal{N}$ for action exploration
3: Receive initial state $\boldsymbol{x}$
4: **for** $t = 1$ to max-episode-length **do**
5: for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(\boldsymbol{o}_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
6: Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\boldsymbol{x}'$
7: Store $\langle \boldsymbol{x}, a, r, \boldsymbol{x}' \rangle$ in replay memory $\mathcal{D}$
8: **for** agent $i = 1$ to N **do**
9: Sample a random minibatch of S samples $\langle \boldsymbol{x}^j, a^j, r^j, \boldsymbol{x}'^j \rangle$ from $\mathcal{D}$
10: $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\boldsymbol{x}'^j, a_1', \ldots, a_N')|_{a_k' = \boldsymbol{\mu}_k'(\boldsymbol{o}_k^j)}$
11: Update critic by minimizing the loss $L(\theta_i) = \frac{1}{S} \sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\boldsymbol{x}^j, a_1^j, \ldots, a_N^j) \right)^2$
12: Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S} \sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(\boldsymbol{o}_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\boldsymbol{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(\boldsymbol{o}_i^j)}$$

13: **end for**
14: Update target network parameters for each agent i:

$$\theta_i' \leftarrow \tau \theta_i + (1 - \tau) \theta_i'$$

15: **end for**
16: **end for**

With this method, the replay memory has to include the combined observations of all agents. The Q-function is updated by minimising its loss:

$$L(\theta_i) = \mathbb{E}\left[\left(Q_i^{\boldsymbol{\mu}}(\boldsymbol{x}, a_1, \ldots, a_N) - (r_i + \gamma Q_i^{\boldsymbol{\mu}'}(\boldsymbol{x}', a_1', \ldots, a_N')) \right)^2 \right] \Bigg|_{a_j' = \boldsymbol{\mu}_j'(\boldsymbol{o}_j)} \tag{2.32}$$

By giving each critic all observations and actions, essentially making all policies $\boldsymbol{\mu}$ part of each agents critic $Q_i^{\boldsymbol{\mu}}$, the authors of the paper have found that this model can successfully learn to act in mixed cooperative-competitive environments and outperform previously existing algorithms in this setting. The full algorithm is given in 2.3.

3. Related Work

[Mni+15] introduced an algorithm that can solve a wide range of RL problems without prior knowledge. Their agent was able to learn on the domain of classic Atari 2600 games [Bel+15] by looking at images from an emulator and receiving the game's score via a hard-coded channel. With this input, their agent was "*able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games*" [Mni+15]. This was achieved with the same algorithm and parameters for all games. However, none of the Atari 2600 games are multiplayer games. The Pommerman environment used in this thesis is played by four players.

AWESOME is an algorithm proposed by [CS07]. It stands for "*Adapt When Everybody is Stationary, Otherwise Move to Equilibrium*" [CS07]. The AWESOME algorithm works on finite games when playing against stationary opponents. Instead of learning through self-play like [Sil+17a], there is only one learning agent trying to match agents with a fixed strategy. [CS07] show an algorithm which is trying to "*adapt to enemies strategies when they appear stationary, but otherwise retreats to a precomputed equilibrium strategy*" [CS07]. With their publication, they provide the first RL algorithm which learns at a minimum to play optimally against stationary opponents in finite games by only using the other agents' actions as additional input.
The AWESOME algorithm, differs to this work by not solving environments where the agents' policies are non-stationary.

[Mar+14] focus on improving RL for multi-agent in dynamic environments. They propose a decentralised algorithm for multi-agent environments which is enhanced by a "*prediction mechanism that provides accurate information regarding up-coming changes*" [Mar+14]. Through pattern matching their agent detects changes in the environment and "*triggers a new model based subject to the latest observations and findings from a database*" [Mar+14]. [Mar+14] validated their algorithm in a realistic smart-grid scenario.
While Pommerman is a multi-agent environment, it does not change its behaviour. This algorithm might be more complex than necessary since Pommerman is not a dynamic environment.

[Foe+17] presents an algorithm called LOLA (Learning with Opponent-Learning Awareness) where the agents shape the anticipated learning rule of the other agents. This algorithm would be suitable for the Pommerman Team 2v2 environment, since they show that LOLA agents can cooperate effectively. Since we are focusing on the Free For All (FFA) mode, which is purely adversarial, LOLA does not fit our setting.

According to [Foe+16] the nonstationarity of Q-learning makes it incompatible with the experience replay memory which is used in deep Q-learning. To address this issue, [Foe+16] propose off-environment importance sampling to stabilise the experience replay. They also show an alternative approach which overcomes the weakness, that the agents view each other as part of the environment, meaning they ignore that the other agents policies change over time. [Foe+16] results on a "*challenging decentralised variant of StartCraft unit micromanagement*" [Foe+16] confirm the hypothesis of these methods. Our work focuses specifically on Pommerman, which this algorithm was not applied to. Doing so would potentially be interesting in future work.

While [Low+17] show a promising algorithm, which allows for multi-agent learning in environments including communication, mixed adversarial-cooperative elements and agents with different objectives. It extends the deep deterministic policy gradients algorithm by introducing a centralised critic for each agent, that receives observations and actions of all agents. We evaluate this algorithm in this thesis (see 2.4) without success and come to the conclusion that it is not fit for environments with discrete action spaces.

4. Methods

4.1. Environment: Pommerman

PlayGround is an effort by a group of DRL researchers where they seek to build standardised environments to explore multi-agent learning and make the research of it more accessible. One issue is, especially with research in cooperative environments, that results from different publications tend to be difficult to compare since many papers implement custom environments. Their goal to create standardised environments, which supports applying a wide range of algorithms, would make the comparison of DRL algorithms much easier [Res+]. We decided to use their environment called “Pommerman”, as it fits our needs. Pommerman is a reimplementation of the game “Bomberman”. Screenshots of both can be seen in figure 4.1.

4.1.1. Modes

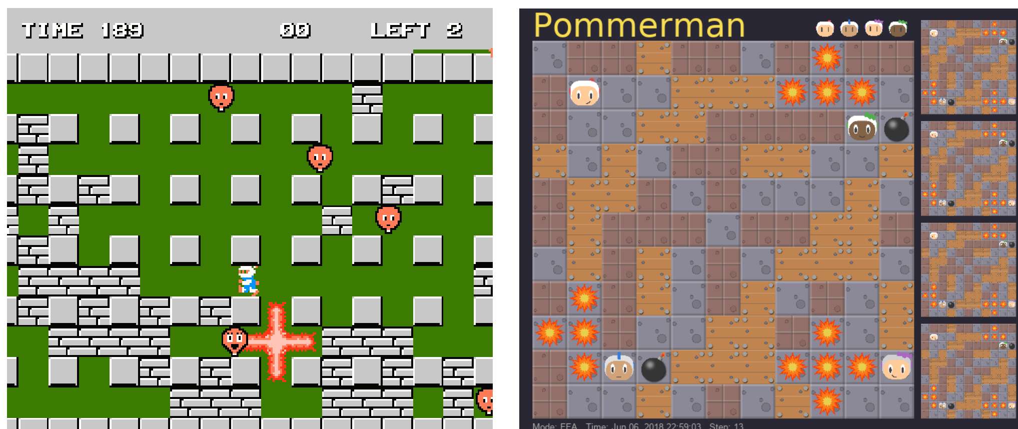
Pommerman is a game of survival, in which four agents play against each other and each agent tries to be the last one to survive. At the beginning of an episode, each agent is placed in one of the four corners in the 11×11 game world. They start off by blowing up “wooden” blocks to open up passageways to the enemy agents. The objective for each agent is to have the other agents stand in an explosion without dying themselves, and that way be the last agent alive.

There are four different modes in the environment:

FFA Each agent is on his own, competing against the other three enemy agents.

Team 2v2 The four agents are split into two teams of two.

Team Radio The four agents are again split into two teams of two. Additionally the agents in the team can communicate with each other. This is done by extending the agents action by two



(a) Screenshot of Bomberman 1983 [Wik16].

(b) Screenshot taken from a running game of Pommerman.

Figure 4.1.: Comparison of the original Bomberman and Pommerman, a reimplementation made for DRL research.

words. The words are limited to a dictionary of 8 elements, concluding with each action they can send one of $8^2 = 64$ messages.

4.1.2. Competitions

The creators of Pommerman have planned two competitions, one on June third that will be held using the FFA environment and one in November at the NIPS 2018 conference, which will be using the Team 2v2 environment. While it would be nice to participate in the FFA competition, it is not the focus of this work.

The game board is of size 11×11. There are 3 types of blocks on the board and examples of them can be seen in figure 4.1b:

passages Gray tiles designate the blocks that agents can stand on.

rigid Dark brown tiles cannot be passed through or destroyed.

wooden Wooden blocks appear as a light brown square and cannot be passed through either but become destroyed if they are close to an explosion.

Bombs can be placed by agents. When they explode, they spread flames horizontally and vertically. Any agent that steps into a flame dies. Bombs have two properties:

Blast strength indicates the range of the explosions.

Bomb life indicates in how many time-steps it will explode and starts at 10.

The agents each start in distinct corners. If the mode is either Team 2v2 or Team Radio, the allied agents start in the diagonally opposite (kitty) corner. Agents start with an ammo count of one. Ammo refers to the number of bombs an agent has. For each bomb placed, the ammo count decreases by one. If a placed bomb explodes, the ammo count is increased by one.

Power-ups change the agents' abilities in some way, usually making them stronger. Half of the wooden walls have hidden power-ups. If it is blown up, the power-up can be picked up by the agents by walking on the tile where the power-up is dropped. Pommerman implements four different power-ups:

Extra Bomb Increase the agents ammo count by one.

Increase Range: The blast strength is increased by one.

Can kick Once picked up, an agent can "kick" bombs away. It does this by moving into a bomb. The bomb then moves away from the agent at the speed of one unit per time-step. A bomb stops moving if it collides with a rigid or wooden wall, a bomb or an agent.

The end of the game is reached if only agents of one team remain. For FFA this means that only one agent remains alive. If through some (surprisingly common) circumstances all remaining agents blow themselves up in a choreographic manner, the game outcome is a draw.

4.1.3. Using The Environment

PlayGround environments closely follow the standards established by OpenAIs Gym environments. The general procedure is described in algorithm 4.1.

Algorithm 4.1 Reinforcement learning in Pommerman

1: $learning_agents \leftarrow$ initialize n learning agents
2: $other_agents \leftarrow$ initialize $4 - n$ non-learning agents
3: $env \leftarrow$ initialize environment
4: env.set_agents(concatenate($learning_agents, other_agents$))
5: **repeat**
6: $observations \leftarrow$s env.reset()
7: $terminal \leftarrow False$
8: **repeat**
9: env.render()
10: $actions \leftarrow env$.act($observations$)
11: $new_observations$, $rewards$, $terminal = env$.step($actions$)
12: $terminals \leftarrow terminals_i$ is true if $learning_agents_i$ perished in this time-step.
13: $ignores \leftarrow ignores_i$ is true if $learning_agents_i$ already was dead in the previous step.
14: **for** i from $1..n$ **do**
15: **if** $ignores_i$ is $False$ **then**
16: $learning_agents_i$.experience($observations_i, action_i, reward_i, new_observations_i$)
17: **end if**
18: **end for**
19: **for** i from $1..n$ **do**
20: $learning_agents_i$.train()
21: **end for**
22: $observations \leftarrow new_observations$
23: **until** $terminal$
24: **until** convergence

Comments on Algorithm 4.1

1-2: There can be two to four agents, not all agents have to be learners.

10: The act method queries all living agents for an action based on their observation. Observations is a list containing an observation for each agent.

10, 11: The act and step methods return lists, each with an entry per agent, except of terminal and info.

11: Terminal is a single boolean variable, which is true when the episode is over for all agents.

12-13: Once an agent received its terminal state, it should not continue to receive any further observations. Hence we introduce an ignores list, with an entry per agent, which is true if the agent has been dead for more than one time-step.

16: Each learning agent receives an experience, which it typically stores in its replay memory.

Observation

An observation for an agent is represented as a python dictionary with several entries. All values are discrete integers:

Actions

The agent can take one of 6 discrete actions per time-step. 5 movement actions and one which places a bomb. Table 4.2 lists all actions and their identifiers.

Table 4.2.: Actions and their IDs.

Action ID	0	1	2	3	4	5
Action Name	Stop	Up	Left	Down	Right	Bomb

Table 4.1.: The observation that the agents receive

Name	Type	Description
`alive`	ndarray	List of agent ids that are still alive.
`board`	11×11 ndarray	Each cell represents a block on the game board. The number indicates the block type.
`bomb_blast_strength`	11×11 ndarray	This is an empty version of the `board` field, whith integer numbers at the location of bomb. This number indicates the blast strength of the bomb at that location.
`bomb_life`	11×11 ndarray	This is similar to `bomb_blast_strength` but indicates the count-down to the bombs explosion.
`position`	tuple with two elements	The agents coordinates.
`blast_strength`	int	Blast strength of the bombs placed by the agent.
`can_kick`	boolean	Usually False, becomes True if the agent picks up the corresponding power-up.
`teammate`	AgentDummy/BaseAgent	in FFA, the value is AgentDummy with an integer value of 10. In Team 2v2 or Team Radio modes, this indicates which agent on the board is the teammate.
`ammo`	int	How many bombs the agent can put on the board at the same time.
`enemies`	list of Item.Agent$n$	The type Item.Agent$n$, where n is an identifier from 0 to 3, has a `value` property, which indicates that enemies value on the board.

Messages

Messages can only be sent in Team Radio mode. The message consists of 2 integer values, both in $[0,8]$. With each action, the agent additionally outputs the message which is in $[1,8]$. Messages are received by the teammate with its next observation and are set to zero if the allied agent has died, hence the restriction to $[1,8]$.

4.2. Experiment Setup

Observation Preprocessing

The observation given by the environment contains partially redundant information and data ranges are not optimal for use with DNNs [LeC+98, Section 4.3]. For this reason, the observations have to be preprocessed beforehand. Several pre-processors have been implemented:

ActorObservation Replaces all enemy player values on the `board` with 14 since the agent does not need to distinguish between them. Additionally the players own position on the board is replaced by 11. `position`, `enemies` and `teammate` are removed from the observation.

ScaleBoardObservation Scales all the values from the `board` ($[0,14]$) to $[-1,1]$.

ScaleOtherObservation Scales `ammo` ($[0,10]$) and `can_kick` ($[0,1]$) to $[-1,1]$.

ScaleBombLifeObservation Scales all the values from the `bomb_life` ($[0,25]$) to $[-1,1]$.

ScaleBombBlastStrengthObservation Scales all the values from the `bomb_blast_strength` ($[0,10]$) to $[-1,1]$.

CombineObservations Combines the observation dictionary to a flat array.

A preliminary experiment has shown, that ANNs can more easily learn to differentiate discrete values that were scaled to real values in a small range of non-integer numbers. Details about this experiment are in Appendix C.

Reward Shaping

Several reward shapers were created, which can be freely combined and added to the hyperparameters. The reward passed to the agent is determined by the sum of rewards returned by the active reward shapers.

DummyReward This is a placeholder reward and simply passes the rewards recieved by the environment through to the environment.

KillReward This reward shaper keeps track of bombs placed by the agent and adds a reward of +0.25 for each kill that the agent gets.

SurvivalReward As long as the agent doesn't die, it adds +1 reward per time-step.

VictoryReward This reward shaper adds +20 for a victory and removes all reward received in the episode when the agent dies.

VictoryRewardFix Similar to VictoryReward, it amplifies the reward for a victory, but this time only by +10. The main difference is that in the case of a loss, this reward shaper removes a fixed value of -10 in case of a loss or draw, instead of removing all rewards received in the episode.

Exploration

For experiments using the MADDPG-agent, instead of using the exploration as shown in algorithm 2.3, ϵ-greedy exploration is used. ϵ is calculated according to equation 4.2, where $\texttt{gs}$ is the current training iteration, which often called global step and $\epsilon\texttt{_factor}$ is a factor which controls how quickly epsilon should fall off. Calculating $\epsilon\texttt{_factor}$ is based on a target ϵ_t that should be reached at a target global step $\texttt{gs}_t$, which is done as shown in equation 4.1. An example of the function 4.2 is shown in figure 4.2.

$$\epsilon\texttt{_factor} = \left(\frac{1}{\epsilon_t} - 1\right)\frac{1}{\texttt{gs}_t} \tag{4.1}$$

$$\epsilon = \frac{1}{\texttt{gs} \cdot \epsilon\texttt{_factor} + 1} \tag{4.2}$$

Experiments based on the DQN-agent use a linear scheduler for calculating ϵ. This scheduler decreases with a constant rate from a starting value to a final value at a certain time-step and thereafter is fixed at the final value. An implementation by [SS17] is used for that.

Evaluation

To assess the agents learning progress, evaluation games are run at an interval with respect to the global step, for instance, every 100'000 training updates. In the self-play setting, four separate instances of the agents are loaded from checkpoints. One of the agents loads the most recent checkpoint, and the other agents load older checkpoints. In an ideal situation, if the agent is learning at a constant rate, the newest agent should win more often than any of the other agents. Exploration is completely removed, this way the agent plays to the best of its abilities. Experiences are not stored in the replay memory during evaluation. 100 evaluation games are run with each evaluation period. After the evaluation is over, the four agents are reset and load the latest checkpoint again, so that training can continue seamlessly.

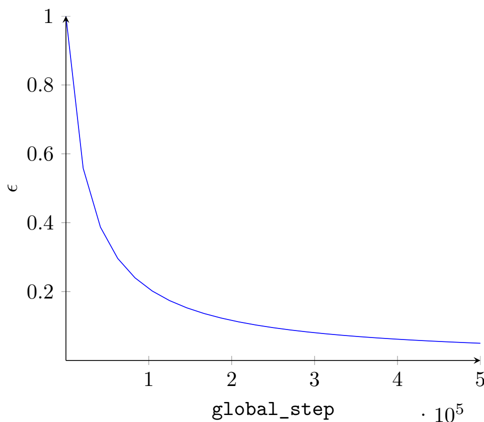


Figure 4.2.: ϵ-greedy exploration as used by the MADDPG-agent. In this example ϵ starts at 1 and is at 0.05 after $5 \cdot 10^5$ training iterations. Therefore $\texttt{gs}_t = 5 \cdot 10^5$, $\epsilon_t = 0.05$ and $\epsilon_\texttt{factor} = \left(\frac{1}{0.05} - 1\right) \cdot \frac{1}{5 \cdot 10^5} = 38 \cdot 10^{-6}$.

Agent Hyperparameters

The most important hyperparameters of the agents are explained here. For a comprehensive explanation of each hyperparameter, please refer to Appendix B.1 and Appendix B.2.

`learning_rate,actor_lr,critic_lr` Are the learning rates for the different kinds of networks. `learning_rate` is for the DQN-agent and the others for the critic and actor networks in the MADDPG-agent.

`tau` The MADDPG-agent gradually updates its target networks. This parameter defines how fast that is.

`evaluation_interval` Sets the interval with respect to training iterations at which evaluation games are to be run.

`minibatch_size` How many samples are drawn from the replay-memory when doing training-updates.

Environments

The Pommerman environment allows for some customisation: the board size, power-ups and block types can be adjusted. The order of the environment descriptions are in ascending order of increasing complexity.

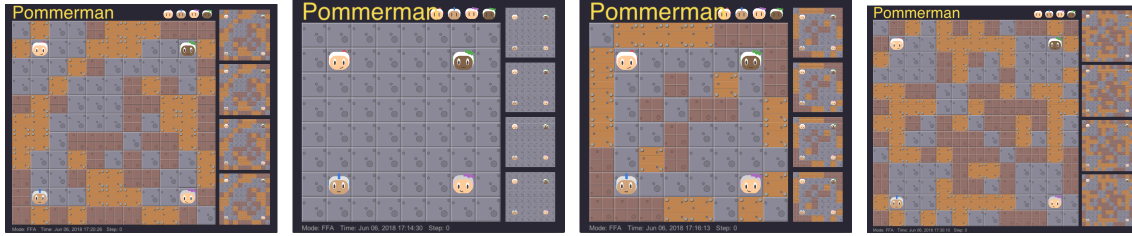
FFA, June competition This environment is used during the competition in the beginning of June. Its board is of size 11×11 and it contains all elements that the environment contains (figure 4.3a).

FFA, 8×8, no blocks, no power-ups This is a minimal environment: the board is reduced to the smallest size (8×8) allowed by the environment. All blocks are removed, which also means that no power-ups can be dropped (figure 4.3b).

FFA, 8×8, no power-ups Both wooden and rigid blocks are enabled, but no power-ups are dropped still (figure 4.3c).

FFA, 13×13 This 13×13 environment is the full FFA-mode game. It is like competition environment, but larger, resembling the original Pommerman FFA environment before it was removed (figure 4.3d).[1]

[1] While no experiments listed in this work use the 13×13 environment, quite a few were done on it. Since those experiments can be found in the code, this environment is still listed in descriptions where the information relevant, namely network architecture descriptions.



(a) 11×11 June competition environment. (b) 8×8 environment without blocks. (c) 8×8 environment with blocks and without power-ups. (d) 13×13 environment.

Figure 4.3.: Overview over all environments used in the experiments.

Network Architectures

Some of the ANN-archtectures are compatible with OpenAIs baseline DQN [SS17] implementation, while the others are built for MADDPG implementation. The model names are prefixed with "OpenAI" or "MADDPG" respectively. Four different ANN architectures used in the described experiments. Their detailed structure is described Appendix A and are referred to as *OpenAI fully connnected* (figure A.1, *OpenAI CNN* and *OpenAI CNN tanh* (figure A.3) and *MADDPG fully connected* (figure A.2).

The input to the ANNs is a flattened representation of the board and additional information. Table 4.3 shows how the varying board sizes result in different ANN input vector lengths.

Table 4.3.: Board sizes and resulting ANN input vector sizes.

Board Size	Input Vector Length
8×8	$3 \cdot 8^2 + 3 = 195$
11×11	$3 \cdot 11^2 + 3 = 366$
13×13	$3 \cdot 13^2 + 3 = 510$

For Convolutional Neural Networks (CNNs) the input is split into two parts: One containing the three boards and one with the additional information. The three boards are reshaped to dimensions of $b \times b \times 3$, where b is the length of a side of the board and are the input for the convolutional layer. The output of the convolutional layers is flattened and concatenated with the additional information vector. This new vector then is passed to the fully connected output layer.

5. Experiments

Only the most relevant plots are shown in this chapter. Please refer to the code repository or accompanying USB-stick as explained in E.

5.1. Experiments using Deep Q-Learning

All experiments in this section use the DQN-algorithm by [SS17].

5.1.1. DQN Experiments 1 and 2: Simple Environment and Self-Play

Environment FFA, 8×8, no blocks, no power-ups.

Agents Four DQN-agents train by playing against each other.

Reward Shaping Only the reward given by the environment is used.

Network Architectures DQN Experiment 1: *OpenAI fully connected*, DQN Experiment 2: *OpenAI CNN*.

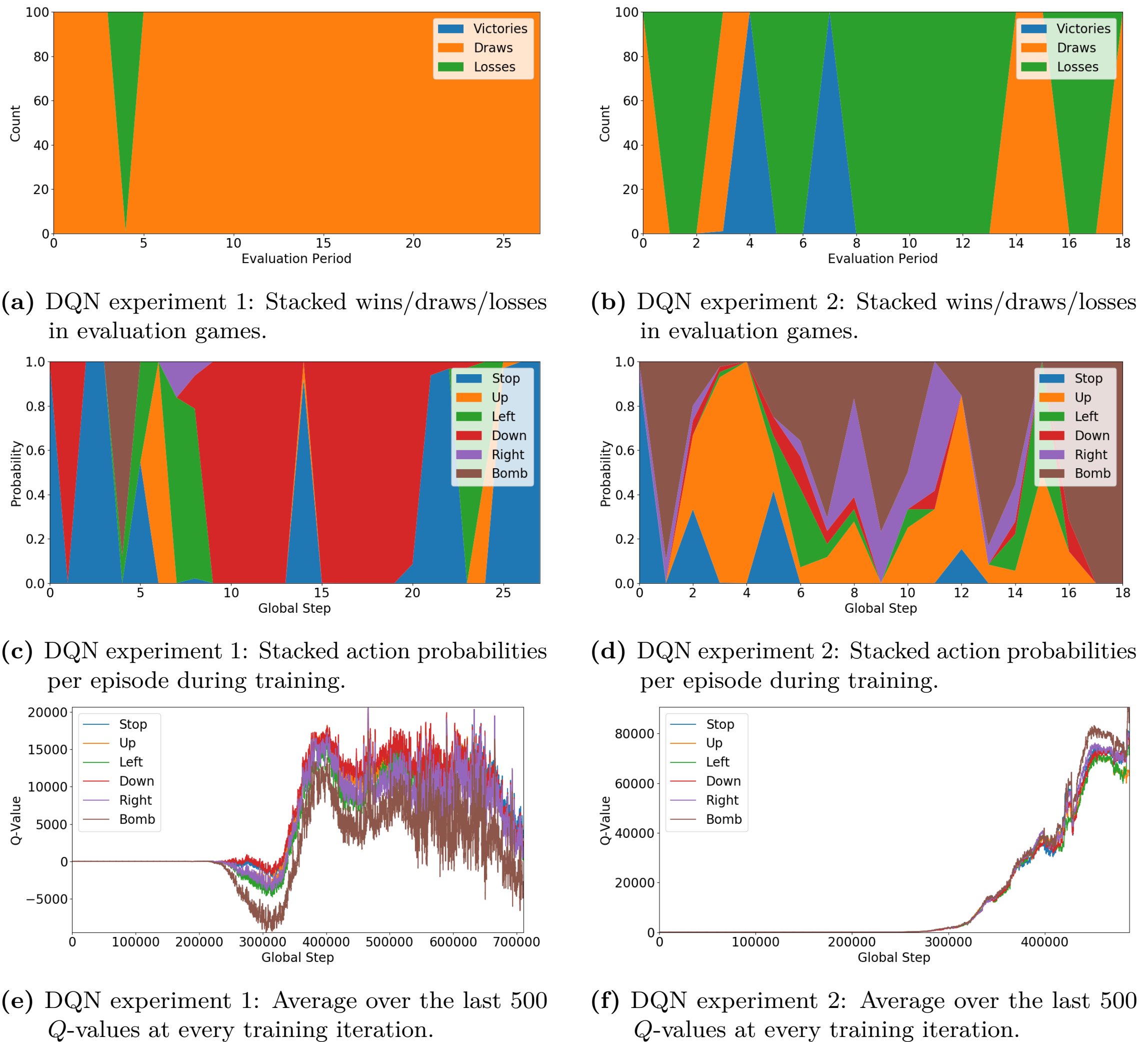
Hyperparameters $\gamma = 0.95$. learning rate=0.0005, minibatch size = 128. For a comprehensive list of all hyperparameters see table B.1 for DQN experiment 1 and table B.2 for DQN experiment 2.

Objective The objective of this experiment is to compare the two architectures *OpenAI fully connected* and *OpenAI CNN*. The hypothesis is that the CNN should perform better than the fully connected network. Convolutional layers have an easier time to detect features and process relative distances. While a fully connected layer can, in theory, do the same, the CNN should have an easier time solving the task than the fully connected network.

Results

DQN Experiment 1 Figure 5.1a shows that the agent usually ends games in a draw. In figure 5.1c it is visible that after half of the training is over, the agent mostly uses one unique action in a single episode. Figure 5.1e shows that the Q-values become large past half of the training period. The visual analysis shows that the agent has learnt to only walk upwards. During the evaluation, all agents walk upwards as far as possible and then stay stuck.

DQN Experiment 2 Figure 5.1b shows some outcome variety during evaluation games. Most of the games are lost, there are some draws and victories, however. There is a tendency to prefer one action over others during training episodes in figure 5.1d, but in most episodes, there is not one action that completely dominates. The Q-values become very large according to figure 5.1f. Visual inspection shows that the agents learnt to choose different actions in a single episode. The variety is not very large, and they get stuck in a fixed position after a few moves.



(a) DQN experiment 1: Stacked wins/draws/losses in evaluation games.

(b) DQN experiment 2: Stacked wins/draws/losses in evaluation games.

(c) DQN experiment 1: Stacked action probabilities per episode during training.

(d) DQN experiment 2: Stacked action probabilities per episode during training.

(e) DQN experiment 1: Average over the last 500 Q-values at every training iteration.

(f) DQN experiment 2: Average over the last 500 Q-values at every training iteration.

Figure 5.1.: Plots for DQN experiments 1 and 2.

Discussion

From comparing the evaluation game outcomes and action probabilities, one can deduce that the agent using *OpenAI CNN* can outperform *OpenAI fully connected* based agent. The visual analysis confirms that assumption. While the CNN based agent seems to also prefer one action heavily over all others at the end of the training, it is not nearly as pronounced compared to the other agent and seems to have resolved itself at the very end of the training.

Figures 5.1e and 5.1f show that the Q-function heavily overestimates the Q-values. Since the output layers use a linear activation, there is no limit in how high or low the predicted values can become. Constraining the Q-function output value range might improve the ANNs performance.

Details about this can be found in Appendix D. This experiment is not listed here since this approach had negative effects on the performance of the agent.

5.1.2. DQN Experiments 3 and 4: Increasing γ to 0.999

Environment To keep the experiments comparable, the environment stays the same as in the previous experiments.

Agents Four DQN-agents in self-play setting.

Network Architectures DQN Experiment 3: *OpenAI fully connected*, DQN Experiment 4: *OpenAI CNN*.

Hyperparameters Change $\gamma = 0.999$ (was 0.95 before), others stay the same. Hyperparameters in table B.3 for DQN experiment 3 and table B.4 for DQN experiment 4.

Objective After seeing the previous experiments perform poorly, one parameter that could have a significant impact on the performance is γ, as it defines how far or short-sighted the Q-function is. An episode on the 8×8 should not take longer than 500 time-steps. Assuming this worst-case scenario, a positive reward for winning after 500 time-steps contributes $0.95^{500} = 7.27 \cdot 10^{-12} \approx 0$ to the predicted Q-value and can as such not be taken into consideration. Increasing γ yields a new discount future reward of $0.999^{500} = 0.61$. With this much higher value, the agents should be able to better predict victories in the far future.

Results

DQN Experiment 3 The evaluation games in figure 5.2a show a few victories and more draws than in earlier results. The action counts in figure 5.2c seem to be more equally distributed. The Q-values are small in figure 5.2e. They are mostly negative at around -1. Visual inspection shows behaviour that is comparable to what was previously observed with the CNN based agent. The agents move in a fixed pattern and then get stuck.

DQN Experiment 4 The evaluation games in figure 5.2b show similar results compared to the previous experiment. Figure 5.2d highlights that the agent now usually prefers one action over all others. The Q-values are overestimated as can be seen in figure 5.2f. Visual inspection shows that often all four agents move down one tile in the first step and then stand still.

Discussion

The *OpenAI fully connected* based agent has improved significantly compared to previous experiments, notably the Q-value predictions are in a range close to what is expected. The predictions are mostly negative, but not below -1.

The performance of the CNN has decreased. Given as nothing besides the discount factor has changed, it is safe to assume that this is an irregularity as it can happen in DRL [Irp]. It is certainly not an argument against setting $\gamma = 0.999$ and as such this will not be reverted.

Looking at the fully connected network's predictions, it seems like it rarely predicts something that is not a loss, hence the negative values. This is plausible: there are much more experiences for losses in training than there are for victories. A draw is given the same reward of -1 as a loss and being victorious is the only way for an agent to receive a positive reward. As can be seen in the evaluation games, draws are much more likely than a single agent winning. This suggests that adding some artificial positive reward might help the agent to perform better.

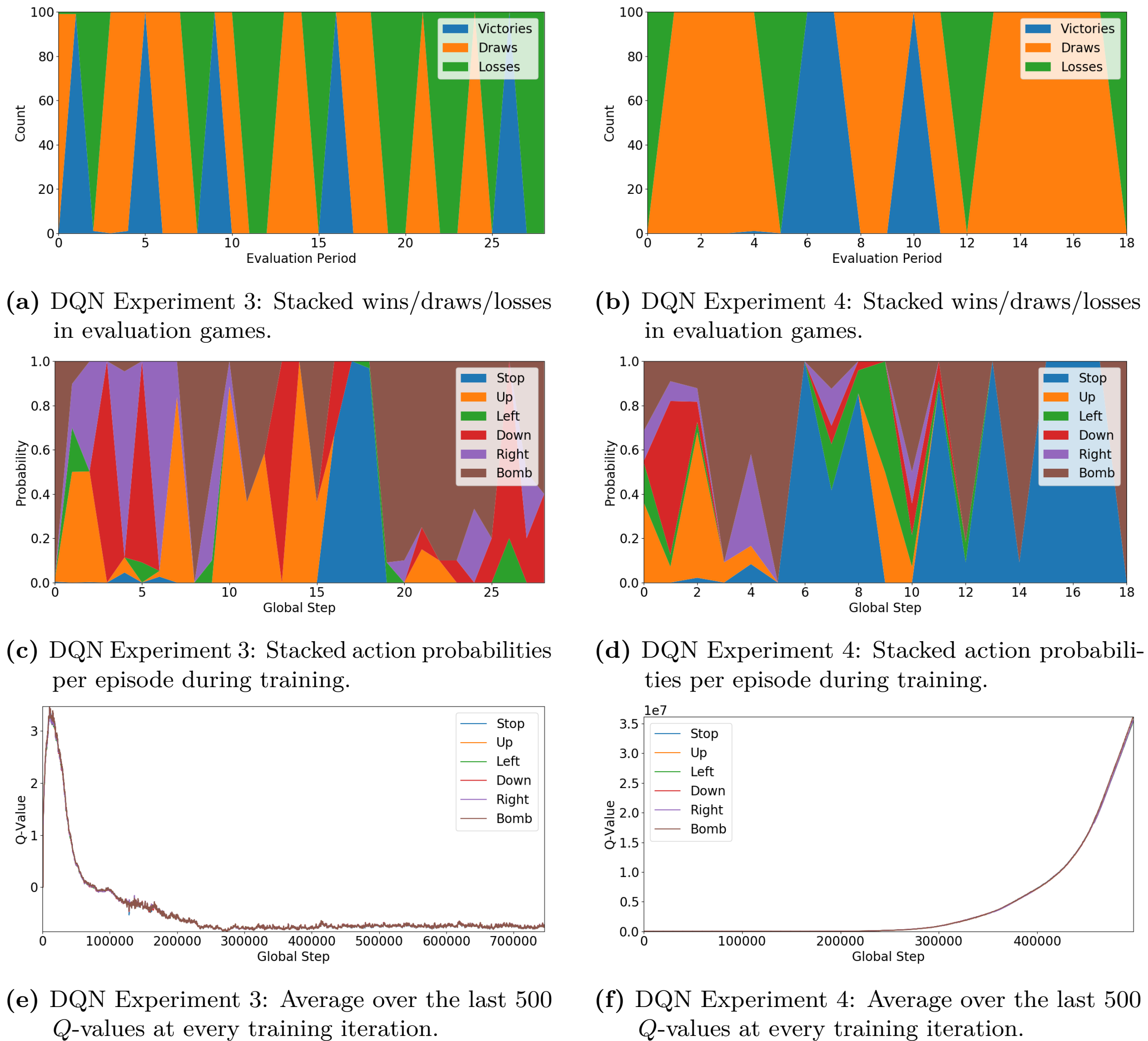
5.1.3. DQN Experiment 5: Reward Shaping

Environment 8×8, no blocks, no power-ups.

Agents Four DQN agents.

Network Architecture *OpenAI CNN*.

Hyperparameters This set of experiments is equal to the previous ones, with the only difference being an added reward for killing enemies being added to the reward shapers. See table B.5 for details.



(a) DQN Experiment 3: Stacked wins/draws/losses in evaluation games.

(b) DQN Experiment 4: Stacked wins/draws/losses in evaluation games.

(c) DQN Experiment 3: Stacked action probabilities per episode during training.

(d) DQN Experiment 4: Stacked action probabilities per episode during training.

(e) DQN Experiment 3: Average over the last 500 Q-values at every training iteration.

(f) DQN Experiment 4: Average over the last 500 Q-values at every training iteration.

Figure 5.2.: Plots of DQN experiments 3 and 4.

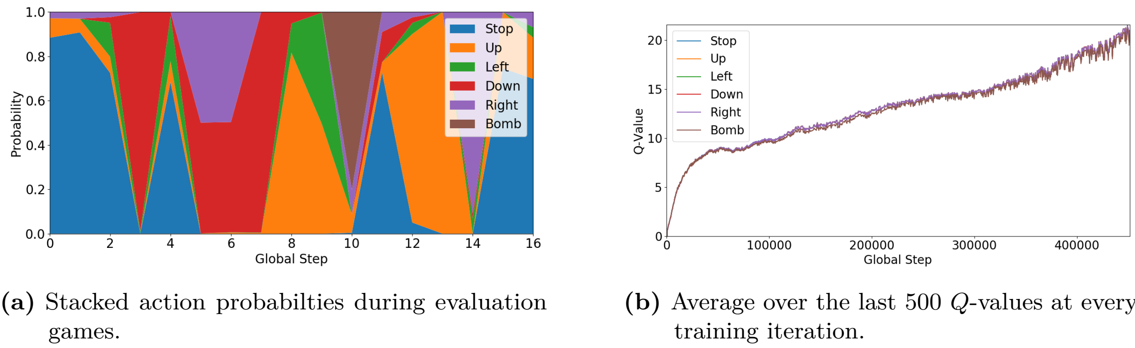
Objective The previous experiments have shown that it is challenging for the agents to learn. The rewards are extremely sparse, as the agent only gets one reward per episode. When using the kill reward, the agent can collect up to four positive rewards per game, which is still not a lot but should help the agent to more easily learn to associate state-action pairs with victories.

Results

The action counts during evaluation games in figure 5.3a show that the agent learns to not place bombs and some variety of actions chosen even if there is no exploration. Figure 5.3b shows that the Q-values are in a relatively small range, but continuously rising. A visual analysis shows that without exploration, all agent move down one step as soon as the episode starts and then stop there until the episode ends.

Discussion

Compared to DQN experiment 4, there seems to be a bit more variety of actions chosen during evaluation games. The Q-values are much better. There still seems to be a strong bias towards one action per round of evaluation games. It is possible that there is a more fundamental flaw in the setup. Looking at the agents playing without exploration indicates that the agent has not



(a) Stacked action probabilties during evaluation games.

(b) Average over the last 500 Q-values at every training iteration.

Figure 5.3.: Plots of DQN experiment 5.

learnt anything that would indicate meaningful behaviour, which is slightly contradicting to what figure 5.3a seems to tell. It seems plausible that adding the kill reward helped the network to predict Q-values more accurately.

Seeing as [Low+17] shows that DQN takes significantly longer to converge in environments where multiple agents are learning, this suggests that it would make sense to change the setting in the environment so that only one agent is learning while all others follow a stationary policy.

5.1.4. DQN Experiment 6: Single-Agent Learning

Environment 8×8, no blocks, no power-ups.

Agents One DQN-agent versus three SimpleAgents.

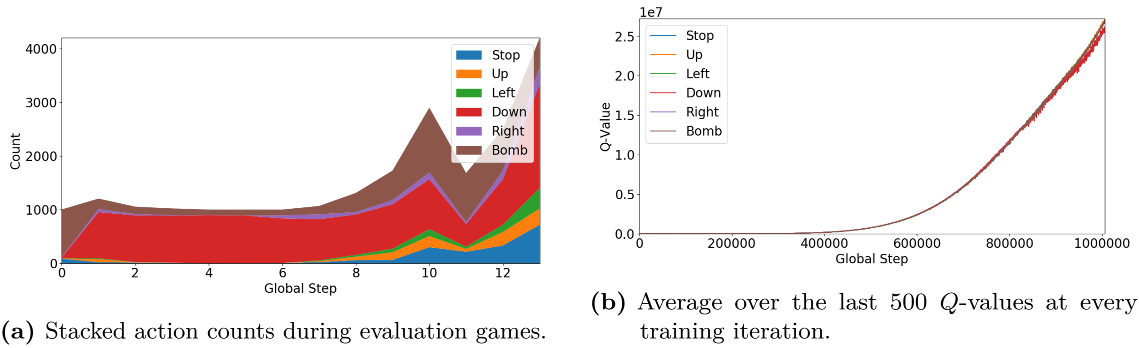
Network Architecture *OpenAI CNN.*

Hyperparameters Other than the single-agent setting, the hyperparameters stay the same. See table B.6.

Objective DQN is known to perform sub-optimally in settings where multiple agents are learning at the same time. From any single agents perspective, the other agents are part of the environment. When the enemy agents learn, this results in the environment changing, as the other agents improve. As a consequence, the environment is non-stationary, which is not a case that DQN is suited for [Low+17]. In this experiment, there is only one learning agent, which means that to the Q-function, the environment is stationary. In this setup, the agent should be able to learn better than in previous ones.

Results

Figure 5.4a shows that actions are chosen more equally towards the end of the training. During most evaluation periods choosing to go down is highly prioritised, which changes over time and actions are chosen more equally distributed. A visual analysis shows that the agent has learnt to walk into the bottom right corner if possible and does not seem to plant bombs. It is possible that it does place bombs rarely, visual inspection has not shown any signs of this, however. Looking at the total actions counts, it is visible that towards the end of the training, the agent survives longer. Figure 5.4b shows that the Q-value predictions become very large.



(a) Stacked action counts during evaluation games.

(b) Average over the last 500 Q-values at every training iteration.

Figure 5.4.: Plots of DQN experiment 6.

Discussion

Upon visual inspection, the agent now seems to have learnt promising behaviour, compared to the previous agent. This is a step in the right direction. It only moves toward the bottom right corner, which might be a local minimum in its behaviour, since the agent does survive longer that way, as figure 5.4a shows.

The Q-values have become huge. It is known that DQN-algorithms tend to overestimate Q-values [HGS16]. The parameters of the algorithm have not been changed compared to prior experiments, with the only difference being that self-play is no longer used. This does not explain the huge Q-values.

Because the statistics of chosen actions during evaluation games looks very promising, with almost all actions being represented in the later evaluation rounds, this strongly suggests that the single-agent learning setting works much better with DQN. In such a simple environment, there is almost no room for strategic behaviour. Adding some elements back into the environment might lead to more variety of behaviour.

5.1.5. DQN Experiment 7: Increased Complexity

Environment FFA, 8×8, no power-ups.

Agents One DQN-agent versus three SimpleAgents.

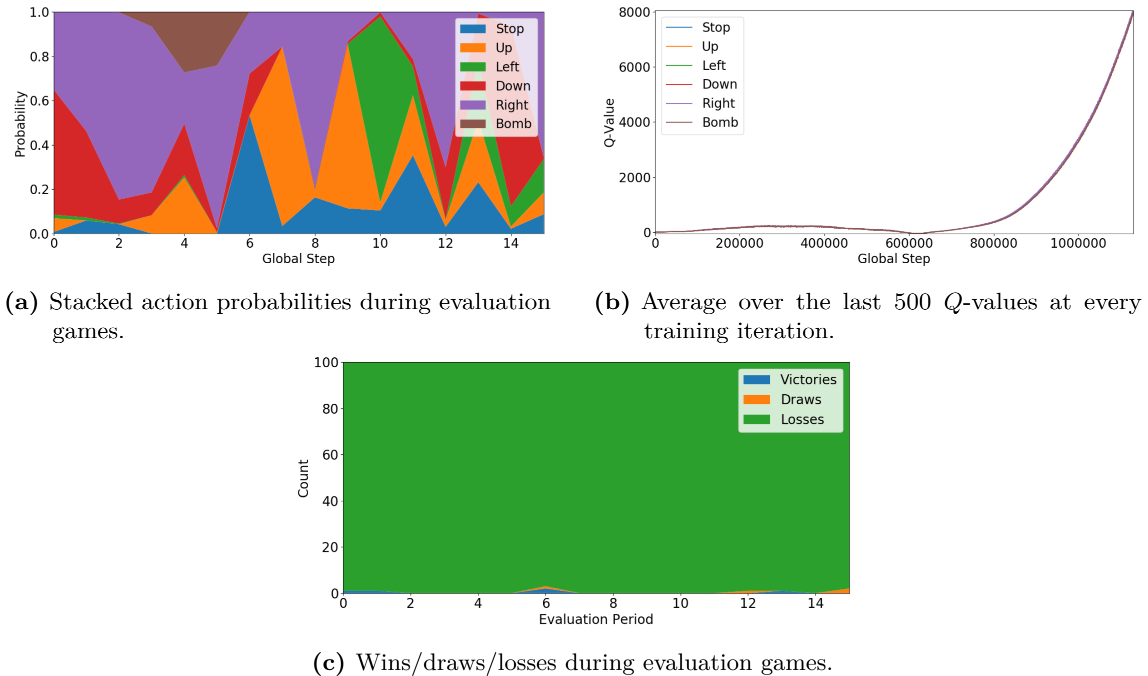
Network Architecture *OpenAI CNN.*

Hyperparameters This experiment is, besides the environment, equivalent to the previous experiment. See table B.7.

Objective The statistics seem to show that . As such it is of interest to see if the agent performs worse again when the complexity of the environment is increased again. Ideally it would be possible to observe how the agent learns to make use of rigid blocks as a simple strategic element.

Results

Figure 5.5a shows that actions are reasonably evenly distributed, especially towards the end. The Q-values are highly overestimated and seem to be exploding towards the end as is shown in figure 5.5b. Visual analysis suggests that the agent has developed a preference to walk to the right. It does show some variance in its behaviour, as it places bombs on rare occasions and moves to different parts of the board. It does not dodge bombs reliably. Figure 5.5c shows that the agent rarely ends the game in a draw or even wins.



(a) Stacked action probabilities during evaluation games.

(b) Average over the last 500 Q-values at every training iteration.

(c) Wins/draws/losses during evaluation games.

Figure 5.5.: Plots of DQN experiment 7.

Discussion

The evaluation games show that the agent can even win on infrequent occasions. Since the training agent is playing against simple agents, this is a interesting result. Together with the actions being selected more evenly, it seems that the choices made are more sensible. It still has developed a preference to move towards one side of the board, albeit less pronounced this time around.

The Q-values are much better but still very high. At the end of the training, the Q-values reach up to 8000 and would continue to rise, if the training would continue, as can be seen in figure 5.5b. Using double Q-learning might help with this issue, since [HGS16] have shown that using double Q-learning can help with Q-value overestimation just as it helps with the standard Q-learning algorithm.

5.1.6. DQN Experiment 8: Competition Experiment

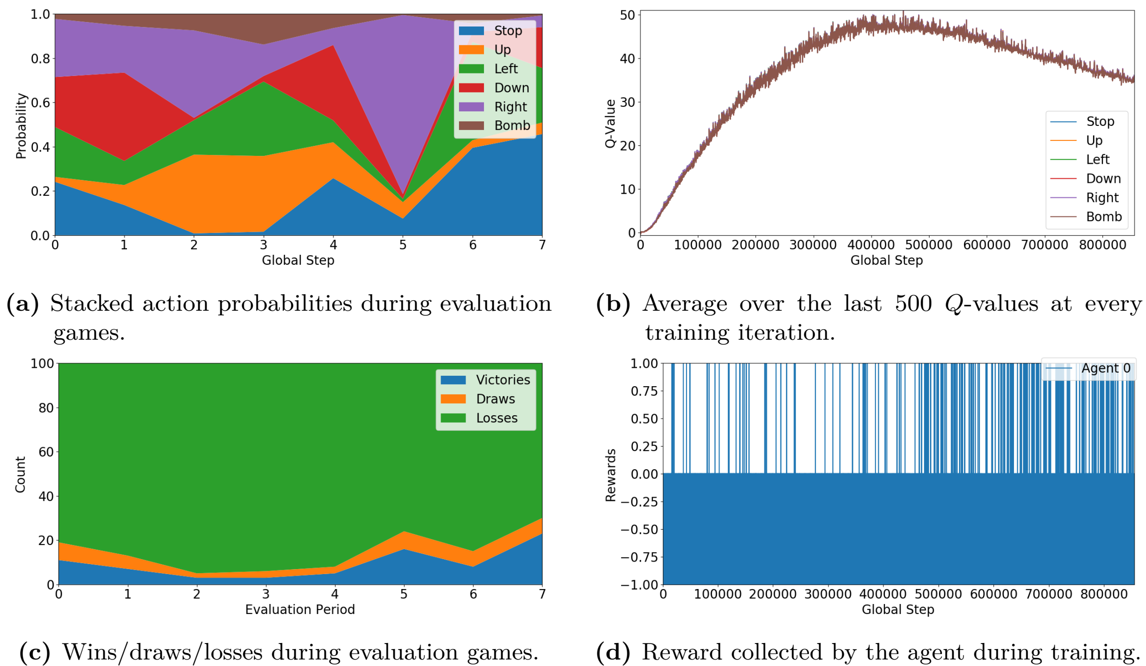
Agents DQN-agent with double Q-Network enabled.

Network Architecture *OpenAI CNN.*

Environment FFA, 11×11, June competition.

Hyperparameters Except of the environment used and use of the double-DQN algorithm by [SS17], the hyperparameters remain the same. See table B.8.

Objective This experiment is run shortly before the competition. To see whether the current approach would have any chance at all at the competition, this experiment is run on the full environment as it is used in the competition held at the beginning of June. Enabling double-Q-networks should help with the Q-value overestimation that was seen in previous experiments since that is one of the main primary of double-DQN [HGS16].



(a) Stacked action probabilities during evaluation games.

(b) Average over the last 500 Q-values at every training iteration.

(c) Wins/draws/losses during evaluation games.

(d) Reward collected by the agent during training.

Figure 5.6.: Plots of DQN experiment 8.

Results

The actions chosen during the evaluation are chosen even more evenly than before, as figure 5.6a. Figure 5.6b shows that the Q-values are much lower than in previous experiments, with the highest values staying below 50 and having a negative inclination towards the end of the training session. During the evaluation, the agent pulls out a draw and victories a few times every 100 games reliably, going up to 30 combined victories and draws in the last set of evaluation games (Figure 5.6c)

The agent collects positive rewards from time to time. In the second half of the training period, the agent collects more positive reward, which is shown in figure 5.6d

A visual inspection shows that the agent mostly chooses not to do anything. It chooses all actions from time to time but mostly sticks to not taking any action.

Discussion

Increasing the complexity of the environment seems to have helped the agent to win. This seems to come down to that the other agents do not only focus on the training agent but of course each other also. With the larger board, it takes longer until the learning agent is confronted with enemy agents. This might explain why the action chosen the most is to stop. On average, the enemy agents will then find each other faster than they will find the training agent, as the training agent does not work on creating a path towards his enemies.

The statistics look very promising. It seems like the agent has found a way to win from time to time, much more often than in any previous experiments.

Using double-DQN seems to have helped with more sane Q-values. The results suggest that this experiment is promising and should be run longer, to see if any other issues will arise. Due to time constraints, this is left for future work.

5.2. MADDPG-Agent Experiments

The experiments in this section use a custom implementation of MADDPG-algorithm as introduced by [Low+17]. Since the MADDPG-algorithm is made for use in multi-agent settings and is reduced to DDPG when not doing so, all experiments are learning in self-play setting.

Two experiments were shown, while more experiments were done using the MADDPG-algorithm, none of them has shown relevant results.

5.2.1. MADDPG Experiments 1 and 2: MADDPG-Agents in Pommerman

Environments 8×8, no blocks, no power-ups (MADDPG experiment 1) and 11×11 June competition (MADDPG experiment 2).

Network Architecture Both experiments use *MADDPG fully connected*.

Parameters Parameters are kept close to the double-DQN-Agents parameters. MADDPG-specific parameters were set as [Low+17] recommends. The most important hyperparameters are shared by the two experiments:

`actor_lr` 0.0001

`critic_lr` 0.0005

`tau` 0.01

`minibatch_size` 128

Hyperparameters of MADDPG experiment 1 are listed in table B.10 and of MADDPG experiment 2 are in table B.11.

Objective These were the last experiments run with MADDPG. They reflect the lessons learned from the DQN experiments, by using a high value for γ and both the KillReward and DummyReward. The only difference between the two experiments is the environment used. With the hyperparameters and environment being similar to DQN experiment 8, the agent should show some signs of intelligent behaviour.

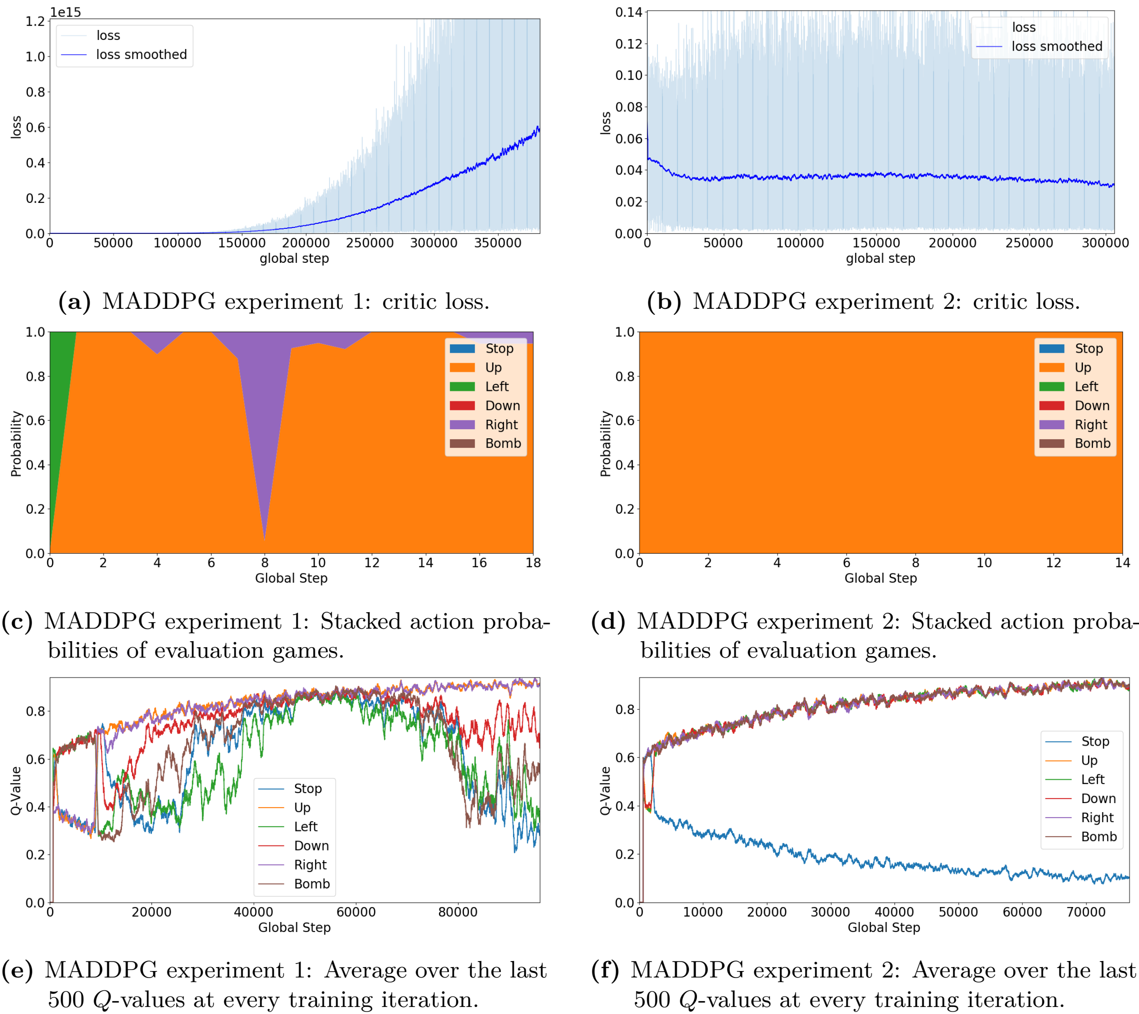
Results

MADDPG experiment 1 The loss in figure 5.7a grew to large values. The agent did vary his choices in actions during evaluation games, as figure 5.7c shows and the Q-values are in range in $[0, 1]$, as shown in figure 5.7e.

A visual analysis shows that the agents move to the top of the board and a few steps to the right. They then continue to move upwards, which is of course not possible, since they are already at the top border of the board.

MADDPG experiment 2 The loss is shrinking over time and is rather small over the entire course of the training, which can be seen in figure 5.7b. Figure 5.7d suggests that the agent did not learn to take any other action besides "Up" throughout the entire training. The Q-values in figure 5.7f are again in the range $[0, 1]$.

As is suggested by the actions logged during evaluation games, visual inspection shows that all agents move upwards as far as they can. They keep taking action "Up", which is, of course, ignored by the environment at that point.



(a) MADDPG experiment 1: critic loss.

(b) MADDPG experiment 2: critic loss.

(c) MADDPG experiment 1: Stacked action probabilities of evaluation games.

(d) MADDPG experiment 2: Stacked action probabilities of evaluation games.

(e) MADDPG experiment 1: Average over the last 500 Q-values at every training iteration.

(f) MADDPG experiment 2: Average over the last 500 Q-values at every training iteration.

Figure 5.7.: Plots of MADDPG experiments 1 and 2.

Discussion There are no signs pointing towards learning progress of any kind. Particularly interesting are the extremely large critic loss in figure 5.7a and how the critic loss is rather small in figure 5.7b, even though the only difference between the two experiments is the environment chosen. The loss should in theory not get this large, given that the Q-values are relatively small. This would hint an implementation bug. That would however pose the question, why this behaviour is completely different in MADDPG experiment 2.

These two experiments are representative of most other MADDPG experiments that are not shown in this thesis. After finding that deterministic policy gradients are for use in continuous action spaces, it was decided that this approach would not be further investigated and experiments should be focused on simpler environments with well tested algorithm implementations should be the next step

6. Conclusion

We were not able to fully answer the initial question of what approach can learn to play Bomberman in a multi-agent setting. Nevertheless, this work shows a few key challenges and brings forth a few concrete pointers which might bring good results in the Pommerman environment. Concluding we want to go over what approaches did not work, but more importantly, which methods might lead to success in future work.

Choice of Environment

We chose the Pommerman environment as a challenge to solve in this work. While the environment is great, in theory, and perfect for our criteria, it was and still is in early development. With the development of Pommerman, we also had to make adjustments to our code base repeatedly. Since some elements have been changed or removed early experiments are no longer reproducible with the latest version of Pommerman. Namely, the board size was reduced to 11×11, a power-up was removed, and the observation received by the environment was updated several times the last time only three weeks before finalising this work[1]. During the project, we have observed several bugs in the environment being fixed and encountered a few ourselves. This, of course, was never a huge hindrance and could mostly be dealt with efficiently. It does, however, pose an issue with reproducibility, as it cannot be guaranteed that the code accompanying this thesis will still work reasonably with Pommerman in a few months from the time of writing.

Choosing a Fitting Algorithm

While the MADDPG algorithm seems to be fit perfectly to solve the Pommermans environments, including communication to cooperate in Team Radio 2v2 mode, we did not succeed in getting it to work in Pommerman. The reason might be a bug in our implementation of the algorithm that we could not identify, or more likely, because it is based on deterministic policies, which is designed to work in continuous action spaces [Low+17].

The DQN algorithm is known to not work well in multi-agent settings, and our experiments have confirmed that the agent learns better when there is only one learning agent. Particularly the double-DQN algorithm has shown the potential for good results in such a simplified setting.

Starting Simple

Our experiments have shown that it makes sense first to start as simple as possible. Initially, we did not expect that having multiple agents learning would pose much of an issue with DQN. The outcome of our experiments has however shown that this is an issue. After simplifying the environment as much as possible by removing as many elements as possible, the experiments were much more successful.

From experience with this thesis, we have learnt that it is vital first to use a minimal environment to get the algorithms to work and then scale the environment up in complexity, step by step figuring out what it takes to get the agent to learn in the environment.

[1] The project was actively developed during the time period we were working on this bachelors thesis: `https://github.com/MultiAgentLearning/playground/commits/master`

Choosing Parameters

We chose parameters by using what has been proven to work in other environments already. This does not mean that those parameters will work just as well in Pommerman. For this reason, we decided to change γ to 0.999, as its previous value of 0.95 was too low for the agent to predict rewards for victories that on average were too far into the future.

Reward Shaping

Sparse rewards make it difficult for the agent to learn. It is challenging to do reward shaping correctly, as it can easily happen that the agent then learns to follow a heuristic instead of learning to solve the environment correctly. By adding a kill reward, which objectively is a good action in Pommerman, the agents seem to have been able to learn better behaviour.

Deep Reinforcement Learning in Pommerman

This work leaves many open questions. It does however illustrate some important findings. Double-DQN, with $\gamma = 0.999$ and a bit of reward shaping and only one learning agent has potential to solve the environment. Using CNNs to approximate the Q-function is performing better than using fully connected networks.

The MADDPG algorithm is not a well-suited algorithm for Pommerman, as none of the experiments has shown any behaviour that would indicate that the agent learnt anything sensible.

7. Future Work

Since this bachelor thesis was not able to solve Pommerman, there are still a lot of possible further approaches to achieving that goal.

Continue Work on Double-Q-learning

The possibilities with the Double-DQN agent are by no means exhausted. It would be interesting to continue working on the agent from DQN experiment 8 (Section 5.1.6), possibly using techniques like curriculum learning [Ben+09] to gradually improve the agent or applying replay memory stabilisation methods as shown in [Foe+16].

Network Architectures

The CNN architecture shows potential, for future work we recommend sticking to use of CNNs. The approach to follow would be to get an agent to win consistently against the SimpleAgent on the small 8×8 environment without power-ups or blocks. We are convinced that the latest experiment is, in theory, able to do so. Doing changes to the network architecture would show how these changes affect performance and a fitting network architecture could be found.

Using curriculum learning [Ben+09] the environment could then be increased in complexity. The network might have to be changed in this process as the environment becomes more complex.

Autoencoder

While autoencoders were not discussed in this thesis, one experiment was done using undercomplete autoencoders. This is inspired by [LR10], where they used autoencoders to compress features in input images. While Pommerman does not supply images as observations, autoencoders could still be helpful, as it would help the ANN to recognise patterns and useful information.

Search

Since the rules of the Pommerman environment are well defined, it would be feasible to apply adversarial search methods to the problem. In this framework of looking ahead, it might be possible to achieve better results. The solution, in this case, could be inspired by [Sil+17b], as they were using self-play and search as well.

Different Algorithm

Once there is a solid understanding of how complex the environment is, it would make sense to experiment with different algorithms that are better suited for multi-agent learning environments like Pommerman. The algorithm would ideally be based on DQN, as those methods are built for discrete action spaces [Mni+15], whereas methods based on DDPG are a better fit for continuous control problems [Lil+15].

Applying MADDPG to DQN would be one such algorithm we would suggest to explore. The MADDPG algorithm extends DDPG by using all agents observations and actions as input for the critic. This works because the additional information is available during training time. At evaluation time, only the actor-network is needed and only this one takes the agents own observation as input.

This could be implemented analogously with DQN, by using the same centralised critic. A “localised” critic then learns to approximate the centralised critic based on only the agent’s observation-action pair.

Bibliography

[Bel+15] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI’15. Buenos Aires, Argentina: AAAI Press, 2015, pp. 4148–4152. ISBN: 978-1-57735-738-4. URL: http://dl.acm.org/citation.cfm?id=2832747.2832830.

[Ben+09] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. “Curriculum learning”. en. In: ACM Press, 2009, pp. 1–8. ISBN: 978-1-60558-516-1. DOI: 10.1145/1553374.1553380. URL: http://portal.acm.org/citation.cfm?doid=1553374.1553380.

[CS07] Vincent Conitzer and Tuomas Sandholm. “AWESOME: A general multiagent learning algorithm that converges in self-play and learns a best response against stationary opponents”. In: *Machine Learning* 67.1 (2007-05), pp. 23–43. ISSN: 1573-0565. DOI: 10.1007/s10994-006-0143-1. URL: https://doi.org/10.1007/s10994-006-0143-1.

[Foe+16] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. “Learning to Communicate with Deep Multi-agent Reinforcement Learning”. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS’16. Barcelona, Spain: Curran Associates Inc., 2016, pp. 2145–2153. ISBN: 978-1-5108-3881-9. URL: http://dl.acm.org/citation.cfm?id=3157096.3157336.

[Foe+17] Jakob N. Foerster, Richard Y. Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. “Learning with Opponent-Learning Awareness”. In: *CoRR* abs/1709.04326 (2017). arXiv: 1709.04326. URL: http://arxiv.org/abs/1709.04326.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Gob07] Goban1. *My floor goban with a game in progress*. 2007-03. URL: https://commons.wikimedia.org/wiki/File:FloorGoban.JPG (visited on 2018-06-04).

[HGS16] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-Learning”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI’16. Phoenix, Arizona: AAAI Press, 2016, pp. 2094–2100. URL: http://dl.acm.org/citation.cfm?id=3016100.3016191.

[Irp] Alexander Irpan. *Deep Reinforcement Learning Doesn’t Work Yet*. URL: http://www.alexirpan.com/2018/02/14/rl-hard.html (visited on 2018-06-05).

[LeC+98] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. “Efficient BackProp”. en. In: *Neural Networks: Tricks of the Trade*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 1998, pp. 9–50. ISBN: 978-3-540-65311-0 978-3-540-49430-0. DOI: 10.1007/3-540-49430-8_2. URL: https://link.springer.com/chapter/10.1007/3-540-49430-8_2.

[Lil+15] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. “Continuous control with deep reinforcement learning”. In: *arXiv:1509.02971 [cs, stat]* (2015-09). arXiv: 1509.02971. URL: http://arxiv.org/abs/1509.02971.

[Low+17] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. “Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments”. In: *arXiv:1706.02275 [cs]* (2017-06). arXiv: 1706.02275. URL: http://arxiv.org/abs/1706.02275.

[LR10] Sascha Lange and Martin Riedmiller. "Deep auto-encoder neural networks in reinforcement learning". In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*. 2010-07, pp. 1–8. DOI: 10.1109/IJCNN.2010.5596468.

[Mar+14] Andrei Marinescu, Ivana Dusparic, Adam Taylor, Vinny Cahill, and Siobhán Clarke. "Decentralised Multi-Agent Reinforcement Learning for Dynamic and Uncertain Environments". In: *CoRR* abs/1409.4561 (2014). arXiv: 1409.4561. URL: http://arxiv.org/abs/1409.4561.

[Mar05] Mark Lee. *6.6 Actor-Critic Methods*. 2005-01. URL: https://cs.wmich.edu/~trenary/files/cs5300/RLBook/node66.html (visited on 2018-06-06).

[Mni+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. "Human-level control through deep reinforcement learning". en. In: *Nature* 518.7540 (2015-02), pp. 529–533. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14236. URL: http://www.nature.com/articles/nature14236.

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning*. en. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com (visited on 2018-05-27).

[Ope17a] OpenAI. *Dendi vs. OpenAI at The International 2017*. 2017-08. URL: https://www.youtube.com/watch?v=wiOopO9jTZw (visited on 2018-06-04).

[Ope17b] OpenAI. *More on Dota 2*. 2017-08. URL: https://blog.openai.com/more-on-dota-2/ (visited on 2018-06-04).

[Res+] Cinjon Resnick, Jakob Foerster, Denny Britz, and David Ha. *Pommerman*. URL: https://www.pommerman.com/ (visited on 2018-06-05).

[RN16] Stuart J. Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach, Global Edition*. English. 3rd ed. Pearson, 2016-05. ISBN: 978-1-292-15396-4.

[San] Grant Sanderson. *Neural networks*. URL: http://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi (visited on 2018-05-27).

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. en. Second Edition. Adaptive computation and machine learning. Cambridge, Massachusetts: MIT Press, 2018-05. ISBN: 978-0-262-19398-6. URL: https://drive.google.com/file/d/1xeUDVGWGUUv1-ccUMAZHJLej2C7aAFWY/view (visited on 2018-06-05).

[Sil+17a] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *arXiv:1712.01815 [cs]* (2017-12). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815.

[Sil+17b] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. "Mastering the game of Go without human knowledge". en. In: *Nature* 550.7676 (2017-10), pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270. URL: https://www.nature.com/articles/nature24270.

[Sil15] David Silver. *Advanced Topics in Machine Learning*. English. University College London, 2015. URL: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html (visited on 2018-05-09).

[SS17] Szymon Sidor and John Schulman. *OpenAI Baselines: DQN*. 2017-05. URL: https://blog.openai.com/openai-baselines-dqn/ (visited on 2018-05-27).

[Wik16] Wikipedia. *File:Bomberman (NES) gameplay.png*. en. Page Version ID: 731830136. 2016-07. URL: https://en.wikipedia.org/w/index.php?title=File:Bomberman_(NES)_gameplay.png&oldid=731830136 (visited on 2018-06-05).

[Yu18] Lantao Yu. *MARL-Papers: Paper list of multi-agent reinforcement learning (MARL)*. original-date: 2017-03-12T06:50:59Z. 2018-05. URL: https://github.com/LantaoYu/MARL-Papers (visited on 2018-05-07).

List of Figures

1.1. Depictions of the two mentioned games Go and Dota 2. 1

2.1. The feedback loop that RL is based on. An agent receives a state observation and a reward. The agent sends actions to the environment based on the observation and learns to maximise its reward by adjusting its state-action association. Adapted from [SB18, p. 48]. 4
2.2. A very simple MDP, three states, two terminal states $s_{1,1}$ and $s_{1,2}$ and discount factor $\gamma = 1$. 6
2.3. The basic setup of actor-critic methods. The critic learns how good state-action pairs are and the critic maximises the reward predicted by the critic. Adapted from [Mar05]. 13
2.4. The setup of MADDPG. The critics receive all inputs and observations of all actors and base value predictions on that information. This is only done during training. Only the critics are used in during execution. Adapted from [Low+17]. 13

4.1. Comparison of the original Bomberman and Pommerman, a reimplementation made for DRL research. 17
4.2. ϵ-greedy exploration as used by the MADDPG-agent. In this example ϵ starts at 1 and is at 0.05 after $5 \cdot 10^5$ training iterations. Therefore $\texttt{gs}_t = 5 \cdot 10^5$, $\epsilon_t = 0.05$ and $\epsilon\texttt{_factor} = \left(\frac{1}{0.05} - 1\right) \cdot \frac{1}{5 \cdot 10^5} = 38 \cdot 10^{-6}$. 22
4.3. Overview over all environments used in the experiments. 23

5.1. Plots for DQN experiments 1 and 2. 26
5.2. Plots of DQN experiments 3 and 4. 28
5.3. Plots of DQN experiment 5. 29
5.4. Plots of DQN experiment 6. 30
5.5. Plots of DQN experiment 7. 31
5.6. Plots of DQN experiment 8. 32
5.7. Plots of MADDPG experiments 1 and 2. 34

A.1. OpenAI fully connected. I
A.2. MADDPG fully connected. Actor and critic networks differ in input and output dimensions, but are otherwise identical. I
A.3. The image shows OpenAI CNN. OpenAI CNN tanh is identical, except of the activation in the output layer being tanh instead of linear. I

C.1. Network that learns to produce a one-hot encoding from a real input. XI
C.2. Loss on training set and test set over the 25 training episodes. XII

D.1. Plots for DQN experiment 9. XIII

List of Tables

4.2. Actions and their IDs. 19
4.1. The observation that the agents receive 20
4.3. Board sizes and resulting ANN input vector sizes. 23

B.1. Hyperparameters of DQN experiment 1 IV
B.2. Hyperparameters of DQN experiment 2 IV
B.3. Hyperparameters of DQN experiment 3 V
B.4. Hyperparameters of DQN experiment 4 V
B.5. Hyperparameters of DQN experiment 5 VI
B.6. Hyperparameters of DQN experiment 6 VI
B.7. Hyperparameters of DQN experiment 7 VII
B.8. Hyperparameters of double-DQN experiment 8 VII
B.9. Hyperparameters for experiment exp10 VIII
B.10. Hyperparameters of MADDPG experiment 1 IX
B.11. Hyperparameters of MADDPG experiment 2 X

C.1. Hyperparameters for preliminary experiment for real number to one-hot encoding. XI

E.1. USB / Repository contents XVI

Acronyms

AI Artificial Intelligence. , 1

ANN Artifical Neural Network. 3, 11, 21, 22, 23, 26, 37, 45, III, VIII, XI

CNN Convolutional Neural Network. , 23, 25, 27, 36, 37

DDPG Deep Deterministic Policy Gradient. 12, 32, 37

DNN Deep Neural Network. 1, 10, 20

DQN Deep Q-Network. , 1, 21, 22, 25, 26, 27, 29, 30, 31, 32, 33, 35, 36, 37, III

DRL Deep Reinforcement Learning. , 1, 2, 17, 27, 43

FFA Free For All. 15, 17, 18, 19, 22, XIII

MADDPG Multi-Agent Deep Deterministic Gradient. , 12, 21, 22, 33, 32, 33, 34, 33, 34, 35, 36, 37, 43, XV

MDP Markov Decision Process. 4, 5, 6, 8, 43

PG Policy Gradient. 11, 12

RL Reinforcement Learning. , 2, 4, 9, 15, 43

TD Temporal-Difference. 8, 9

A. Network Architectures

A.1. OpenAI fully connected and MADDPG fully connected

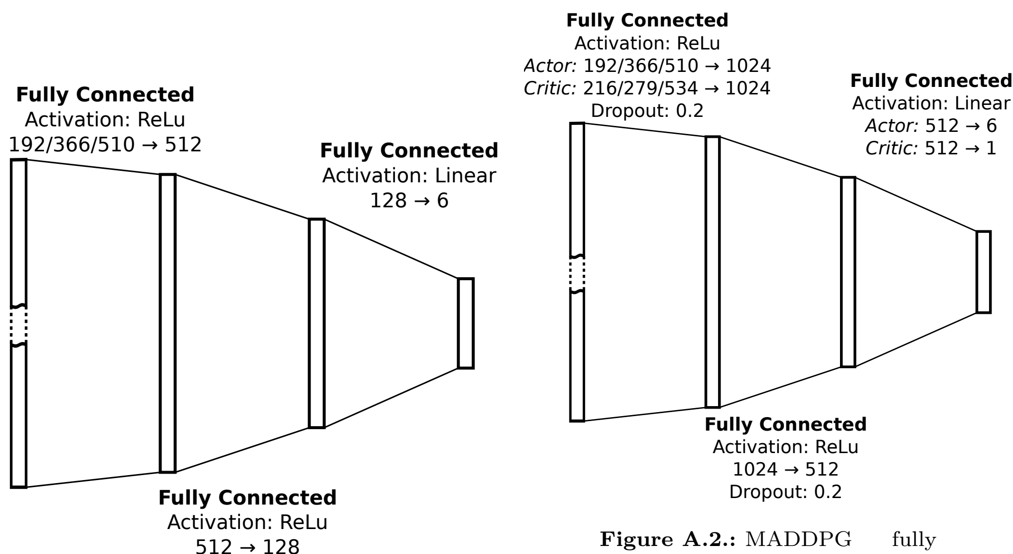


Figure A.1.: OpenAI fully connected.

Figure A.2.: MADDPG fully connected. Actor and critic networks differ in input and output dimensions, but are otherwise identical.

A.2. OpenAI CNN and OpenAI CNN tanh

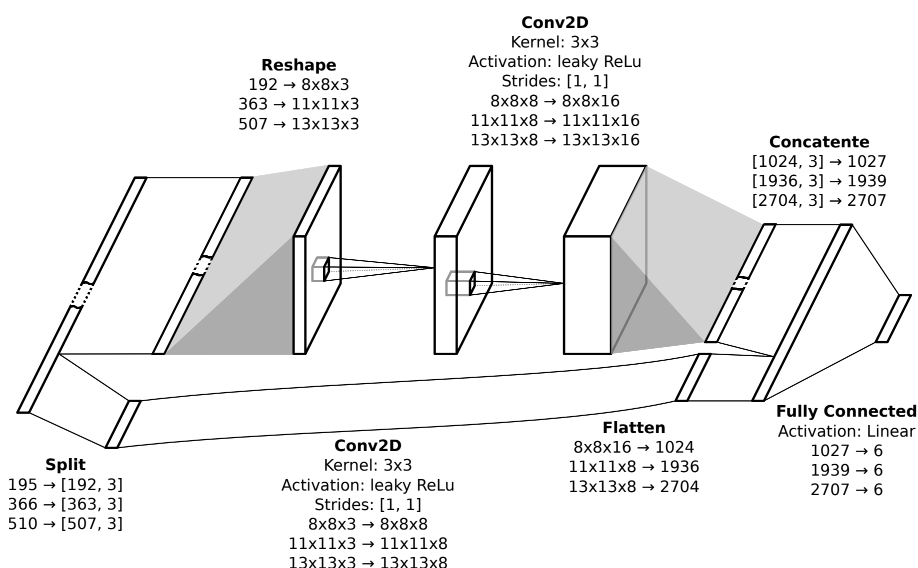


Figure A.3.: The image shows OpenAI CNN. OpenAI CNN tanh is identical, except of the activation in the output layer being tanh instead of linear.

B. Experiment Hyperparameters

This chapter is a list of tables, showing the hyperparameters of all relevant experiments.

B.1. OpenAI Baselines DQN-Agent Hyperparameters

`environment_config` A function that returns properties required by the Pommerman environment. This contains things such as board size, block types, available power ups and more.

`use_double_q` If set to true, double Q-learning is used.

`learning_rate` The learning rate used to update the ANN with.

`replay_memory_size` How many elements can be stored in the replay memory.

`min_training_replay_memory_size` The number of elements that need to be in the replay memory before training begins. Before that is reached, the network is not updated, only playing happens in order to gather enough experience tuples.

`minibatch_size` How many samples are drawn from the replay memory for an update.

`nr_episodes` How many games should be played before ending the training.

`exploration` OpenAI baselines has several different schedulers for computing ϵ, which is used for exploration. This parameter is an instance of any one of those schedulers. Only the LinearScheduler is used for DQN-agents.

`training_agents` A list of classes which of learning agents. These will be instantiated later on.

`playing_agents` A list of agents that need not be updated. These agents are added to the environment just like the `training_agents` are, but are ignored by the training algorithm.

`model` A python module containing a function that takes a tensorflow tensor object, the size of the action space and returns a tensorflow tensor object. When querying tensorflow for that tensor, the result is a vector with length of the action-space, where each element of the tensor represents the state-action value of an action.

`observation_preprocessor` An instance of `Observation`, to which a list of observation pre processors is passed.

`reward_shaper` A list of reward shapers.

`keep_max_n_checkpoints` The parameters are saved regularly. A short history is kept of past checkpoints and this parameter limits how many are kept.

`update_target_interval` Determines how often the target network is updated.

`evaluation_interval` Determines after how many training iterations evaluation games are run.

`evaluation_games` How many games are run in an evaluation period.

`evaluation_agents_checkpoints` A list of four elements, determining which checkpoints are to be used. A value of 0 means that the latest checkpoint is used, a value of -1 means that the previous checkpoint is used and so on. The `evaluation_interval` determines how often the models are saved.

`gamma` The discount factor as it is described in paragraph 2.2.1.

DQN Experiment 1

Table B.1.: Hyperparameters of DQN experiment 1

Hyperparameter	**Value**
environment_config	FFA, 8x8, no blocks, no power-ups
use_double_q	False
learning_rate	0.0005
replay_memory_size	500000
min_training_replay_memory_size	1024
minibatch_size	128
nr_episodes	10000000
exploration	Linear Scheduler: initial eps = 0, final eps = 0, num steps = 500000
training_agents	DeepQAgent, DeepQAgent, DeepQAgent, DeepQAgent
playing_agents	None
model	OpenAIDense3
observation_preprocessor	ActorObservation ScaleBoardObservation ScaleBombBlastStrengthObservation ScaleBombLifeObservation ScaleOtherObservation CombineObservations
reward_shaper	DummyReward
keep_max_n_checkpoints	10
update_target_interval	1000
evaluation_interval	100000
evaluation_games	100
evaluation_agents_checkpoints	0, -1, -1, -1
gamma	0.95

DQN Experiment 2

Table B.2.: Hyperparameters of DQN experiment 2

Hyperparameter	**Value**
environment_config	FFA, 8x8, no blocks, no power-ups
use_double_q	False
learning_rate	0.0005
replay_memory_size	500000
min_training_replay_memory_size	1024
minibatch_size	128
nr_episodes	10000000
exploration	Linear Scheduler: initial eps = 0, final eps = 0, num steps = 500000
training_agents	DeepQAgent, DeepQAgent, DeepQAgent, DeepQAgent
playing_agents	None
model	OpenAIConv2D2Dense1
observation_preprocessor	ActorObservation ScaleBoardObservation ScaleBombBlastStrengthObservation ScaleBombLifeObservation ScaleOtherObservation CombineObservations
reward_shaper	DummyReward
keep_max_n_checkpoints	10
update_target_interval	1000
evaluation_interval	100000
evaluation_games	100
evaluation_agents_checkpoints	0, -1, -1, -1
gamma	0.95

DQN Experiment 3

Table B.3.: Hyperparameters of DQN experiment 3

Hyperparameter	Value
environment_config	FFA, 8x8, no blocks, no power-ups
use_double_q	False
learning_rate	0.0005
replay_memory_size	500000
min_training_replay_memory_size	1024
minibatch_size	128
nr_episodes	10000000
exploration	Linear Scheduler: initial eps = 0, final eps = 0, num steps = 500000
training_agents	DeepQAgent, DeepQAgent, DeepQAgent, DeepQAgent
playing_agents	None
model	OpenAIDense3
observation_preprocessor	ActorObservation ScaleBoardObservation ScaleBombBlastStrengthObservation ScaleBombLifeObservation ScaleOtherObservation CombineObservations
reward_shaper	DummyReward
keep_max_n_checkpoints	10
update_target_interval	1000
evaluation_interval	100000
evaluation_games	100
evaluation_agents_checkpoints	0, -1, -1, -1
gamma	0.999

DQN Experiment 4

Table B.4.: Hyperparameters of DQN experiment 4

Hyperparameter	Value
environment_config	FFA, 8x8, no blocks, no power-ups
use_double_q	False
learning_rate	0.0005
replay_memory_size	500000
min_training_replay_memory_size	1024
minibatch_size	128
nr_episodes	10000000
exploration	Linear Scheduler: initial eps = 0, final eps = 0, num steps = 500000
training_agents	DeepQAgent, DeepQAgent, DeepQAgent, DeepQAgent
playing_agents	None
model	OpenAIConv2D2Dense1
observation_preprocessor	ActorObservation ScaleBoardObservation ScaleBombBlastStrengthObservation ScaleBombLifeObservation ScaleOtherObservation CombineObservations
reward_shaper	DummyReward
keep_max_n_checkpoints	10
update_target_interval	1000
evaluation_interval	100000
evaluation_games	100
evaluation_agents_checkpoints	0, -1, -1, -1
gamma	0.999

DQN Experiment 5

Table B.5.: Hyperparameters of DQN experiment 5

Hyperparameter	Value
environment_config	FFA, 8x8, no blocks, no power-ups
use_double_q	False
learning_rate	0.0005
replay_memory_size	500000
min_training_replay_memory_size	1024
minibatch_size	128
nr_episodes	10000000
exploration	Linear Scheduler: initial eps = 0, final eps = 0, num steps = 500000
training_agents	DeepQAgent, DeepQAgent, DeepQAgent, DeepQAgent
playing_agents	None
model	OpenAIConv2D2Dense1
observation_preprocessor	ActorObservation ScaleBoardObservation ScaleBombBlastStrengthObservation ScaleBombLifeObservation ScaleOtherObservation CombineObservations
reward_shaper	KillReward, DummyReward
keep_max_n_checkpoints	10
update_target_interval	1000
evaluation_interval	100000
evaluation_games	100
evaluation_agents_checkpoints	0, -1, -1, -1
gamma	0.999

DQN Experiment 6

Table B.6.: Hyperparameters of DQN experiment 6

Hyperparameter	Value
environment_config	FFA, 8x8, no blocks, no power-ups
use_double_q	False
learning_rate	0.0005
replay_memory_size	500000
min_training_replay_memory_size	1024
minibatch_size	128
nr_episodes	10000000
exploration	Linear Scheduler: initial eps = 0, final eps = 0, num steps = 500000
training_agents	DeepQAgent
playing_agents	SimpleAgent, SimpleAgent, SimpleAgent
model	OpenAIConv2D2Dense1
observation_preprocessor	ActorObservation ScaleBoardObservation ScaleBombBlastStrengthObservation ScaleBombLifeObservation ScaleOtherObservation CombineObservations
reward_shaper	KillReward, DummyReward
keep_max_n_checkpoints	10
update_target_interval	1000
evaluation_interval	100000
evaluation_games	100
evaluation_agents_checkpoints	0, -1, -1, -1
gamma	0.999

DQN Experiment 7

Table B.7.: Hyperparameters of DQN experiment 7

Hyperparameter	Value
environment_config	FFA, 8x8, no power-ups
use_double_q	False
learning_rate	0.0005
replay_memory_size	500000
min_training_replay_memory_size	1024
minibatch_size	128
nr_episodes	10000000
exploration	Linear Scheduler: initial eps = 0, final eps = 0, num steps = 500000
training_agents	DeepQAgent
playing_agents	SimpleAgent, SimpleAgent, SimpleAgent
model	OpenAIConv2D2Dense1
observation_preprocessor	ActorObservation ScaleBoardObservation ScaleBombBlastStrengthObservation ScaleBombLifeObservation ScaleOtherObservation CombineObservations
reward_shaper	KillReward, DummyReward
keep_max_n_checkpoints	10
update_target_interval	1000
evaluation_interval	100000
evaluation_games	100
evaluation_agents_checkpoints	0, -1, -1, -1
gamma	0.999

DQN Experiment 8

Table B.8.: Hyperparameters of double-DQN experiment 8

Hyperparameter	Value
environment_config	FFA, 11x11, June competition
use_double_q	True
learning_rate	0.0005
replay_memory_size	500000
min_training_replay_memory_size	1024
minibatch_size	128
nr_episodes	10000000
exploration	Linear Scheduler: initial eps = 1, final eps = 0, num steps = 500000
training_agents	DeepQAgent
playing_agents	SimpleAgent, SimpleAgent, SimpleAgent
model	OpenAIConv2D2Dense1
observation_preprocessor	ActorObservation ScaleBoardObservation ScaleBombBlastStrengthObservation ScaleBombLifeObservation ScaleOtherObservation CombineObservations
reward_shaper	KillReward, DummyReward
keep_max_n_checkpoints	10
update_target_interval	1000
evaluation_interval	100000
evaluation_games	100
evaluation_agents_checkpoints	0, -1, -1, -1
gamma	0.999

DQN Experiment 9

Table B.9.: Hyperparameters for experiment exp10

Hyperparameter	Value
environment_config	FFA, 8x8, no blocks, no power-ups
use_double_q	False
learning_rate	0.0005
replay_memory_size	500000
min_training_replay_memory_size	1024
minibatch_size	128
nr_episodes	10000000
exploration	Linear Scheduler: initial eps = 0, final eps = 0, num steps = 6000000
training_agents	DeepQAgent, DeepQAgent, DeepQAgent, DeepQAgent
playing_agents	None
model	OpenAIConv2D2Dense1tanh
observation_preprocessor	ActorObservation ScaleBoardObservation ScaleBombBlastStrengthObservation ScaleBombLifeObservation ScaleOtherObservation CombineObservations
reward_shaper	DummyReward
keep_max_n_checkpoints	10
update_target_interval	1000
evaluation_interval	200000.0
evaluation_games	100
evaluation_agents_checkpoints	0, -1, -1, -1
gamma	0.95

B.2. MADDPG-Agent Hyperparameters

`environment_config` A function that returns properties required by the Pommerman environment. This contains things such as board size, block types, available power ups and more.

`actor_lr` If set to true, double Q-learning is used.

`critic_rate` The learning rate used to update the ANN with.

`tau` At what rate the target network is updated.

`gamma` The discount factor as it is described in paragraph 2.2.1.

`gradient_clip_norm` Policy gradients can become large. This parameter defines at which absolute value the gradients are clipped.

`epsilon_global_step` Determins after how many training iterations an ϵ is reached, which is given in the next parameter.

`epsilon_at_epsilon_global_step` Determines the exact value of ϵ after a certain amount of training iterations.

`replay_memory_size` How many elements can be stored in the replay memory.

`min_training_replay_memory_size` The number of elements that need to be in the replay memory before training begins. Before that is reached, the network is not updated, only playing happens in order to gather enough experience tuples.

`minibatch_size` How many samples are drawn from the replay memory for an update.

`nr_episodes` How many games should be played before ending the training.

`training_agents` A list of classes which of learning agents. These will be instantiated later on.

playing_agents A list of agents that need not be updated. These agents are added to the environment just like the `training_agents` are, but are ignored by the training algorithm.

agent_model A python module containing a function that takes a tensorflow tensor object, the size of the action space and returns a tensorflow tensor object. When querying tensorflow for that tensor, the result is a vector with length of the action-space, where each element of the tensor represents the state-action value of an action.

observation_preprocessor An instance of `Observation`, to which a list of observation pre processors is passed.

reward_shaper A list of reward shapers.

keep_max_n_checkpoints The parameters are saved regularly. A short history is kept of past checkpoints and this parameter limits how many are kept.

evaluation_interval Determines after how many training iterations evaluation games are run.

evaluation_games How many games are run in an evaluation period.

evaluation_agents_checkpoints A list of four elements, determining which checkpoints are to be used. A value of 0 means that the latest checkpoint is used, a value of -1 means that the previous checkpoint is used and so on. The `evaluation_interval` determines how often the models are saved.

MADDPG Experiment 1

Table B.10.: Hyperparameters of MADDPG experiment 1

Hyperparameter	Value
environment_config	FFA, 8x8, no blocks, no power-ups
actor_lr	0.0001
critic_lr	0.0005
tau	0.001
gamma	0.999
gradient_norm_clip	5.0
epsilon_at_epsilon_global_step	0.01
epsilon_global_step	10000000.0
replay_memory_size	100000
min_training_replay_memory_size	1024
minibatch_size	128
nr_episodes	1000000000
training_agents	MADDPGAgent, MADDPGAgent, MADDPGAgent, MADDPGAgent
playing_agents	None
agent_model	Dense3Layers
observation_preprocessor	ActorObservation ScaleBoardObservation ScaleBombBlastStrengthObservation ScaleBombLifeObservation ScaleOtherObservation CombineObservations
reward_shaper	KillReward, DummyReward
keep_max_n_checkpoints	10
evaluation_interval	20000
evaluation_games	100
evaluation_agents_checkpoints	0, -1, -1, -1

MADDPG Experiment 2

Table B.11.: Hyperparameters of MADDPG experiment 2

Hyperparameter	Value
environment_config	FFA, 11x11, June competition
actor_lr	0.0001
critic_lr	0.0005
tau	0.001
gamma	0.999
gradient_norm_clip	5.0
epsilon_at_epsilon_global_step	0.01
epsilon_global_step	10000000.0
replay_memory_size	100000
min_training_replay_memory_size	1024
minibatch_size	128
nr_episodes	1000000000
training_agents	MADDPGAgent, MADDPGAgent, MADDPGAgent, MADDPGAgent
playing_agents	None
agent_model	Dense3Layers
observation_preprocessor	ActorObservation ScaleBoardObservation ScaleBombBlastStrengthObservation ScaleBombLifeObservation ScaleOtherObservation CombineObservations
reward_shaper	KillReward, DummyReward
keep_max_n_checkpoints	10
evaluation_interval	20000
evaluation_games	100
evaluation_agents_checkpoints	0, -1, -1, -1

C. Preliminary Experiment

In the reward shaping experiments, we scale the values to $[-1, 1]$, as is described in section 4.2. To check how difficult it is for a network to map a floating point number to a discrete value, an experiment was conducted to empirically show that this does work. The network is very simple and can be seen in figure C.1. All hyperparameters are listed in table C.1.

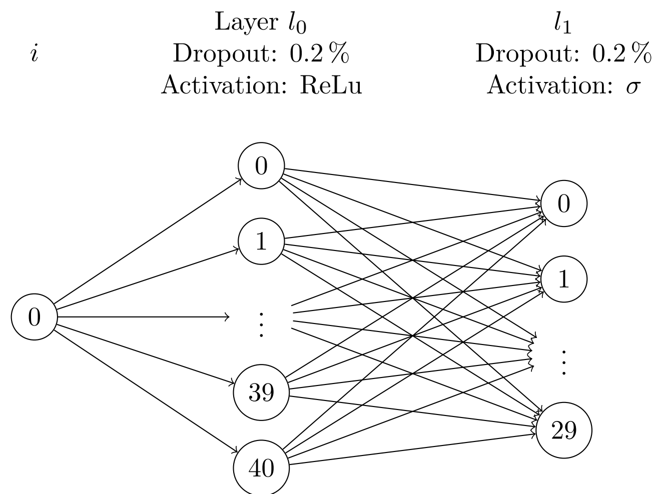


Figure C.1.: Network that learns to produce a one-hot encoding from a real input.

Table C.1.: Hyperparameters for preliminary experiment for real number to one-hot encoding.

Parameter	Value
Number of Samples	$2^{22} = 1048576$
Test-Set	0.2 %
Optimiser	Adam
Learning Rate	0.001
Minibatch Size	1024
Epochs	100

Data Generation The data required is very simple: In a first step, 2^{22} samples are generated, which serve as input data. In a second step, labels are generated, which are a one-hot representation of the previously generated random numbers. Lastly, the inputs are scaled from $[0, 29]$ to $[-1, 1]$.

Results The network can easily learn to map real numbers to a one-hot encoding. Without much tweaking, the ANN is able to learn to map all numbers from 0 to 29 mapped to $[-1, 1]$ to a one-hot encoding without any errors. The loss becomes very low quickly, which is shown in figure C.2. After

only 25 episodes of training, which is in total 81925 training iterations (3277 iterations per episode, based on number of sample minus the test set).

This experiment confirms that the way the preprocessing is done is not a source for error, as the networks should easily be able to differentiate 14 different values in the board. While the `bomb_life` values are in $[0, 25]$, these values are much less important to differentiate between, as it can be looked at as a continuous countdown. Even if this would not be the case, the network should still be able to learn this however.

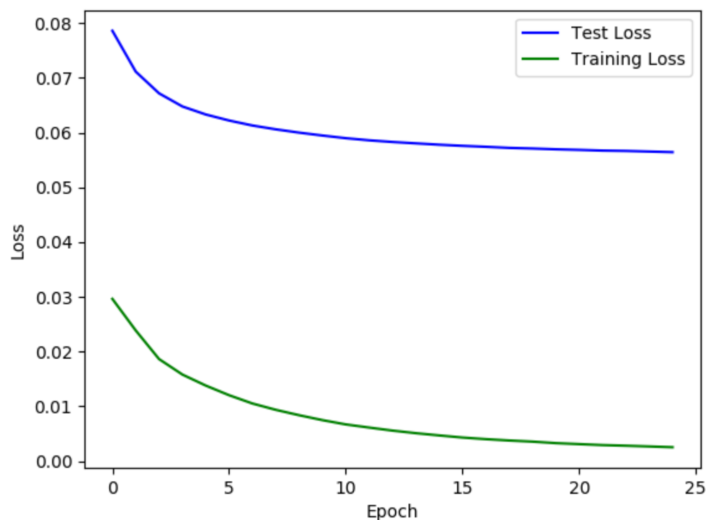


Figure C.2.: Loss on training set and test set over the 25 training episodes.

D. DQN Experiment 9: Changing the Activation Function

This experiment is the same as the previous ones, except of the following points:

Environment FFA, 8×8, no blocks, no power-ups.

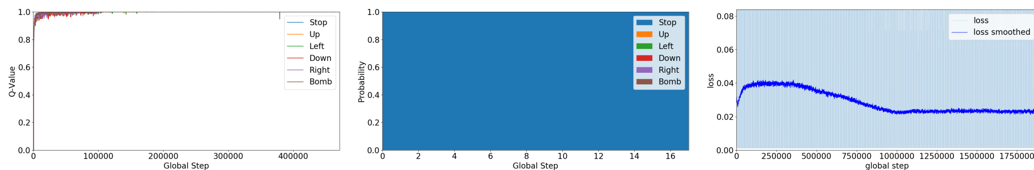
Agents Four DQN-agents.

Network Architecture *OpenAI CNN tanh*, see figure A.3 for detailed architecture.

Parameters learning rate $= 0.0005$, minibatch size $= 128$, detailed hyperparameters are listed in table B.9.

Objective Because the Q-values became huge in previous experiments, this experiment aims to limit the Q-function's output range by using tanh activation for the output layer. The hypothesis is that the agent might learn faster if the Q-values are in a better range.

Results



(a) Average Q-values over the last 500 values at each training iteration.

(b) Stacked action counts during evaluation games.

(c) Critic loss during training.

Figure D.1.: Plots for DQN experiment 9.

Figure D.1a shows how the Q-values quickly clip to 1. The stacked action probabilities plot in figure D.1b shows that the agent only chooses the action "Stop". The loss in figure D.1c is small.

Discussion

Since all Q-values are equal with a value of ≈ 1, $\text{argmax}_{a \in \mathbb{A}} Q(\boldsymbol{s}, a)$ always returns index 0, which is the action "Stop". It makes sense that the critic loss is relatively small, which also means that the agents will not learn much, however. It seems obvious at this point that the approach of adding a tanh activation function does not work.

E. Experiment Hard- and Software

Hardware

All experiments were run on the ZHAW internal GPU cluster using NVIDIA Titan Xp GPUs and Intel Xeon CPU E5-2650 v4 CPUs.

Software

Information on how to run our code is in the README.md on the usb stick. Following dependencies need to be installed first:

- Python 3.5 or higher
- high performance message passing library headers. (In ubuntu repositories: libopenmpi-dev)
- zlib compression library headers. (In ubuntu repositories: zlib1g-dev)

We recommend using the docker image provided in `src/docker` or on docker hub: `https://hub.docker.com/r/dujoram/pommerman-tensorflow-gpu/`

Python Packages

- pandas
- tk
- opengl
- pydot
- tensorforce
- keras
- Dependencies of Pommerman: `https://github.com/MultiAgentLearning/playground`
- Dependencies of OpenAI baselines: `https://github.com/openai/baselines`

USB / Repository contents

The USB[1] provided with this thesis contains the following folder structure:

[1] can also be found in the git repository `https://github.engineering.zhaw.ch/Sparclex/ba-stdm-3-2018`

Table E.1.: USB / Repository contents

Path	Description
Bachelor-Thesis-Stdm-3-2018.pdf	This document
README.md	A short description to reproduce the experiments
experiment_plots	All generated plots of the described experiments
src	Experiment code
src/ActorCriticModel	All models used for the experiments
src/agents	DeepQ and MADDPG agents
src/docker	The docker file, which was used to run the experiments
src/encoder_trainer.py	Encoder training runner
src/exp2tex.py	Converts the experiement hyperparameters to a tex table
src/experiment_data	models, plots and csv_logs
src/ExperimentLog.py	Experiment logger
src/experiments	Experiment classes which do the whole setup and handling of an experiment
src/hyperparameters.py	All the hyperparameters of the experiments
src/p	The pommerman repository
src/plot.py	Creates plots by the given experiment
src/preprocessors	Observation preprocessors
src/ReplayMemory.py	Implementation of a replay memory
src/reward_shapers	Reward shapers
src/runner.py	Runs the experiments by using the experiment class and the hyperparameters
src/tensorboard_runner.py	Handles all the tensorboard related stuff
src/util.py	Utility functions