Deep Reinforcement Learning for the cooperative card game Hanabi

Grooten, Bram J.

Award date:
2021

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science

Deep Reinforcement Learning for the cooperative card game Hanabi

Master Thesis

Bram Grooten

Supervisors:

Jim Portegies

Jelle Wemmenhove

Maurice Poot

1 September 2021

Abstract

In this research we pursue a better understanding of deep reinforcement learning, specifically actor-critic algorithms. We do this by providing mathematical descriptions of three algorithms and comparing their performance in a simplified environment of the cooperative card game Hanabi. Vanilla Policy Gradient outperforms Simple Policy Gradient and even Proximal Policy Optimization, scoring an average of 24.4 out of 25 points in two-player self-play mode with 70.5% perfect games. We seek beneficial algorithmic design options and hyperparameter settings, and find that adding an entropy term to the objective function and using specific reward shaping increased the learning pace in our experiments. Finally, in search for relevant game-theoretical properties of Hanabi we provide a proof for the maximum length of a perfect game (71 turns) and any game (89 turns).

The code belonging to this project can be found at:
gitlab.tue.nl/jim-portegies/student-projects/bram-grooten

Preface

I would like to thank my supervisors Jim Portegies, Jelle Wemmenhove, and Maurice Poot for their admirable support during this master research project. A wholehearted thank you to my family for all the warmth, love, and encouragement. I dedicate this report to my mother, who has shown me what real perseverance looks like.

Further gratitude goes to my friends at Serpentine, who got me excited about reinforcement learning and played many games of Hanabi with me, including that first perfect 25. Also, I thank the people behind the TU/e's High Performance Cluster for all their assistance. Last but not least, thank you to Nolan Bard for helping me set up his team's Hanabi Learning Environment.

Contents

Chapter 1

Introduction

In everyday life, we humans are often communicating with each other to reach a common goal. We have unconsciously been trained to recognize and interpret the beliefs of others, and compare them with our own. We act as a multi-agent system that can often find near-optimal behaviour without much effort.

In the future we probably have to collaborate not only with other humans, but with artificial intelligent (AI) agents as well. According to a recent survey, AI researchers believe there is a 50% chance of AI outperforming humans in all tasks in 45 years [16]. In order for this advance of AI to turn out well for us, we think it is important that AI agents learn how to effectively communicate with humans. Next to understanding our language, an AI should be able to sense our intentions and beliefs.

Hanabi is a card game where such an ability to sense the intentions and beliefs of others can greatly improve the outcome, making it a perfect testbed for the development of collaborative algorithms. The game is cooperative and partially observable, which means that all players must work together and communicate in a smart manner to get the highest possible score.

Figure 1.1: The card game Hanabi. Image from [1].

1.1 Rules of Hanabi

To get a good grip of the game we will be talking about, we begin with an explanation of the rules. In a game of Hanabi the goal is to get the highest possible score while playing together as a group. It can be played with two to five players. All 50 cards are shuffled and put in the deck. When there are two or three players, everyone gets 5 cards as an opening hand. In a four or five player game, each player starts with 4 cards. The twist of Hanabi is the fact that each player may not look at his or her own cards, but is able to see the cards of all other players.

Each card has a rank and a color. The possible ranks are 1, 2, 3, 4, 5 and the colors are red, blue, yellow, green, white. The objective of the game is to form five stacks of cards, one of every color. Each stack, or firework[1], is built up with ranks from 1 to 5. A stack can be started by playing a card with rank 1 on the table. Cards with higher ranks and the same color can subsequently be played on top, but only if the rank of the card to be played is one higher than the stack's current number.

During the game, players take turns in clockwise order. In each turn, the active player has to choose between three possible moves:

1. Give a hint.
2. Discard a card.
3. Play a card.

In order to obtain knowledge about the cards one is holding, players are allowed to give one hint per turn, as long as there are hint tokens left over. The game starts with 8 of these tokens. To give a hint, a player must choose one rank or one color, announce it, and point at all the cards with this property in another player's hand. A hint can only be given to one other player, and only if that player has cards with the chosen property[2]. After the hint is given, one hint token is taken out of the "communication budget".

Players can retrieve hint tokens by discarding a card. They do so by announcing it and placing one card on the discard pile. After this move, the player gets a new card from the deck and one hint token is put back into the budget. Discarding cards can only be done if the budget is not full already.

All players may observe the discarded cards. There are a certain number of cards for each color and rank, so players may be able to deduce properties of their own cards with information from the discard pile. For each color, there are three 1's, two 2's, two 3's, two 4's, and one 5. This gives a total of 50 cards.

When playing a card, the player must announce that he or she is doing so, and put one card on the table. If it fits correctly on one of the fireworks, the card has been played successfully. If it does not, the card is placed on the discard pile and the group loses one life token. In either case, the player gets a new card from the deck. Only when a 5 has been successfully played, the group retrieves one hint token as a bonus (but never going over the maximum of 8).

The game starts with 3 life tokens. If there are none left, the game ends immediately and the players score 0 points[3]. A game can also end by finishing all five fireworks and thus reaching 25 points, which is called a perfect game. If both of these options do not happen, the game comes to an end when the bottom card of the deck is drawn. All players, including the one who emptied the deck, get one more turn. The final score is the number of cards played successfully.

[1] *Hanabi* is Japanese for *fireworks*.
[2] In a variant players may hint: "You have zero red cards", but we will not consider this a legal move.
[3] In another variant the score still equals the total of the fireworks in this case, but we do not use it.

1.2 Simplified version

As a step on the way towards an algorithm that can play this collaborative, partially observable game well, we will be using a simplified version of Hanabi in this research project. On top of that, we will restrict ourselves to the two-player setting of the game. In the simplified version, players are allowed to cheat by looking at their own cards. The partial observability is hereby greatly reduced, as the only part of the game which remains hidden is the deck.

In the perfect information setting, where even the order of the cards in the deck is known to the players, it has been proven that the problem of finding a winning play sequence (to get the perfect score of 25 points) is NP-complete [7]. It should also be noted that not every initial configuration of Hanabi has a winning play sequence at all: imagine for example the situation where all the cards with rank 1 are on the bottom of the deck [42].

In Appendix B.1 we will come back to the full version of Hanabi, as described in Section 1.1, to talk about possible ways to extend our algorithms to this domain. We hope that future research finds some use in the directions provided.

1.3 Research questions

In this research our goal is to discover and describe the mathematics behind a couple of important deep reinforcement learning algorithms, while also implementing these algorithms for Hanabi to compare their performance. Three main algorithms will be analyzed: Simple Policy Gradient (SPG), Vanilla Policy Gradient (VPG), and Proximal Policy Optimization (PPO).

The questions we want to answer in this report are the following:

1. How can we develop a deep reinforcement learning algorithm that can play a simplified version of Hanabi well?
    a) Which algorithm (SPG, VPG, or PPO) reaches the highest average score?
    b) Which algorithmic design options and hyperparameter settings are beneficial to the learning pace?
2. What are the mathematical descriptions of the deep reinforcement learning algorithms SPG, VPG, and PPO?
3. What are game-theoretical properties of Hanabi relevant to the design of an algorithm?

In Chapter 2 we provide a brief overview of related work and the current state of the art. Chapter 3 presents a mathematical background for Hanabi and deep reinforcement learning. We go more in depth into the algorithms SPG, VPG, and PPO in Chapter 4 to answer Question 2. In Chapter 5 we briefly describe the design process of our implementation. We present the experiments with their setup and results in Chapter 6 to compare the different algorithms and hyperparameters, answering Question 1. The last question is addressed in Chapter 7, where we prove certain properties of Hanabi. In Chapter 8 we discuss the most important lessons learned and point to directions for future research. We conclude the report in Chapter 9.

Chapter 2

Related work

The challenge of designing a good Hanabi playing agent has been coined by a team of researchers at DeepMind just last year, in their paper "The Hanabi challenge: A new frontier for AI research" [8]. The paper served as a starting point for this research project. Our implementations build on the Hanabi Learning Environment which they have set up [11].

Bard et al. [8] defined two separate domains in their Hanabi challenge paper, called *self-play* and *ad-hoc*. In self-play an agent only plays with copies of itself, while in ad-hoc agents must be able to play with a wide range of other agents or even human players. Most of the current literature focuses on self-play, with a couple of exceptions. Our research also stays in the self-play domain, but we are very curious to see what future research can bring for the ad-hoc setting.

Another important distinction is the approach used to program an agent for Hanabi. We separate them into the categories: with or without machine learning. We call the agents that do not use any learning method *rule-based*, and it turns out that they are still outperforming the learning agents in many cases. In the following two subsections we will explore both approaches.

2.1 Rule-based agents

Within the rule-based regime there again exist two categories: bots that are based on human Hanabi conventions such as [13], and bots that use *hat-guessing* strategies [10]. Both approaches can achieve quite decent scores in self-play, but not in ad-hoc play.

The hat-guessing method is based on a mathematical game where players have to guess the color of their own hat. In Hanabi players do not know the color (and rank) of their own cards, so this called for similar strategies. By using modular arithmetic, a lot of information can be given with a single hint, provided that all players follow the same algorithm.

Some of the best bots that use human conventions include SmartBot [30] and FireFlower [44]. The state-of-the-art in self-play is held (for 3+ players) by a bot that uses the hat-guessing strategy, called WTFWThat [45]. Its scores have been improved later on by the use of search methods [26].

Canaan et al. focus on the ad-hoc gameplay in their paper about creating agents that vary in behaviour [9]. They define two measures, *risk aversion* and *communicativeness*, with values between 0 and 1. By dividing both of these dimensions into 20 bins they defined 400 categories in which a certain agent could be placed. The agents with high risk aversion (about 0.8) and moderate communicativeness (0.5) performed best.

There has also been some research into the area of ad-hoc play between computers and humans. In 2017 a paper by Eger et al. provides a few rule-based agents "designed to play better with a human cooperator" [12], based on Grice's communication theory [17] from the field of psychology. Only the two-player version of Hanabi was used in this study, where one person would play with

an agent. The 224 participants scored an average of 15.0 points with the best agent, although the scores were widespread (standard deviation: 4.2).

2.2 Learning agents

For their overview paper, Bard et al. [8] applied two existing approaches of deep reinforcement learning to the Hanabi Learning Environment which they provide. The Rainbow agent [19], which combines improvements on Deep Q-Networks, scores an average of about 18.2 in self-play, taken over all player modes (2, 3, 4, and 5). The Actor-Critic-Hanabi-Agent or ACHA, which Bard et al. have built based on [27], performed better by getting an average score of 20.3. However, in the ad-hoc case both agents had scores close to zero.

A team of researchers at Facebook has worked on self-play agents which they call Action Decoders. In September 2019 the Bayesian Action Decoder (BAD) [14] set a new record for 2-player games of Hanabi. Not much later, in December 2019, the team improved their bot with the Simplified Action Decoder (SAD) [21], which drastically increased the scores among learned policies in self-play for any number of players. The state-of-the-art for 3 to 5 players is still held by the rule-based bot WTFWThat [45], but reinforcement learning is ahead in the 2-player domain, as shown in Table 2.1.

The SAD agent provided a simple, yet elegant solution to the problem of updating beliefs during the exploration phase. In this phase many random actions are taken, which can give misleading information about the state of the game to other agents. Thus, only during training, the agents were allowed to communicate their preferred action, while performing a different random action. This simplified the Bayesian reasoning process.

The scores of SAD were further improved by the same research team through search methods [26]. The agents start off with a blueprint policy, which can be any strategy, also a learned one. In every step of the game, the agents perform a search for the best action using many Monte Carlo rollouts. This action can deviate from the blueprint policy. To make sure that the other agents do not misinterpret the action taken, all agents redo the search of every other agent themselves, using the same random seed (which is shared before the game starts). Agents now know whether an action came from the blueprint policy or from search. This improved the state-of-the-art in self-play for every number of players, see Table 2.2.

Table 2.1: An overview of agents in the self-play domain from both approaches: rule-based at the top, learning methods at the bottom. Each entry shows the mean score along with the standard error of the mean and the percentage of perfect games. The best scoring agent in each regime is indicated in bold. Data is taken from [8], except for the last row which comes from [21].

Agent	2P	3P	4P	5P
SmartBot [30]	**22.99 ± 0.00**	23.12 ± 0.00	22.19 ± 0.00	20.25 ± 0.00
	29.6%	13.8%	2.1%	0.004%
WTFThat [45]	19.45 ± 0.03	**24.20 ± 0.01**	**24.83 ± 0.01**	**24.89 ± 0.00**
	0.28%	**49.1%**	**87.2%**	**91.5%**
Rainbow [19]	20.64 ± 0.11	18.71 ± 0.10	18.00 ± 0.09	15.26 ± 0.09
	2.5%	0.2%	0%	0%
ACHA [8]	22.73 ± 0.12	20.24 ± 0.15	21.57 ± 0.12	16.80 ± 0.13
	15.1%	1.1%	2.4%	0%
SAD [21]	**24.08 ± 0.01**	**23.99 ± 0.01**	**23.81 ± 0.01**	**23.01 ± 0.01**
	56.09%	**50.37%**	**41.45%**	**13.93%**

Table 2.2: The state-of-the-art agent in self-play for each number of players. The (+s) indicates that all these agents use the search methods [26], which increased their original scores. Data is taken from [26].

	2P	3P	4P	5P
Agent	SAD(+s)	WTFWThat(+s)	WTFWThat(+s)	WTFWThat(+s)
Score	24.61 ± 0.01 75.5%	24.83 ± 0.006 85.9%	24.96 ± 0.003 96.4%	24.94 ± 0.004 95.5%

Chapter 3

Background

In this chapter we provide a detailed formulation of the game in Section 3.1 and a brief mathematical background of deep reinforcement learning (DRL) in Section 3.2. Together they form the recommended prerequisite knowledge for the rest of the report.

3.1 Mathematical formulation of Hanabi

In this research we will approach the game from a mathematical perspective. We will thus give some definitions of a Hanabi game, taken from [42] and changed slightly. They looked at the game in a general sense, with arbitrary numbers of colors, players, etc.

Definition 1. An *initial configuration* of Hanabi is defined as a 7-tuple of the form $H = (n, k, p, h, D_0, m_0, l_0)$. Denoting $\text{Cards}(H) = \{1, \dots, n\} \times \{1, \dots, k\}$, we interpret the parameters in the following way:

1. $n \in \mathbb{N}$ is the number of available card *ranks*,
2. $k \in \mathbb{N}$ is the number of available card *colors*,
3. $p \in \mathbb{N}$ is the number of *players*,
4. $h \in \mathbb{N}$ is the *hand size* of every player,
5. $D_0 = (c_i)_{i=1}^{N}$ is an ordered finite sequence of elements $c_i \in \text{Cards}(H)$ with $N \in \mathbb{N}$ such that $p \cdot h \leq N$ forming the *initial deck*,
6. $m_0 \in \mathbb{N}_0$ is the number of hint tokens initially available, and
7. $l_0 \in \mathbb{N}$ is the number of life tokens initially available.

The classic game of Hanabi, as described in the rules above, would be $H = (5, 5, p, h, D_0, 8, 3)$ with $p \in \{2, \dots, 5\}$,

$$h = \begin{cases} 5 & \text{if } p \in \{2, 3\} \\ 4 & \text{if } p \in \{4, 5\} \end{cases}$$

and the initial deck $D_0 = (c_i)_{i=1}^{50}$ with the cards $c_i = (x_i, y_i)$ where x_i denotes the rank and y_i denotes the color, shuffled uniformly random and being such that the following multisets have these specific number of elements:

$$\forall\ \mathsf{color} \in \{1, \dots, 5\}: \quad \Big|\{c_i = (1, y_i) \mid c_i \in D_0, \quad y_i = \mathsf{color}\}\Big| = 3$$

$$\forall\ \mathsf{rank} \in \{2, 3, 4\},\ \mathsf{color} \in \{1, \dots, 5\}: \quad \Big|\{c_i = (x_i, y_i) \mid c_i \in D_0,\ x_i = \mathsf{rank},\ y_i = \mathsf{color}\}\Big| = 2$$

$$\forall\ \mathsf{color} \in \{1,\ldots,5\}: \quad \Big|\{c_i = (5, y_i) \ \mid c_i \in D_0, \qquad\qquad y_i = \mathsf{color}\}\Big| = 1.$$

Next to an initial configuration, we need more definitions of elements within a Hanabi game, to be able to define the complete state of a game later. From [42] we use:

Definition 2. Given an initial configuration of Hanabi $H = (n, k, p, h, D_0, m_0, l_0)$ we define the following.

1. The remaining *deck* is a sequence $D = (c_i)_{i=1}^{L}$ with $c_i \in \text{Cards}(H)$ for some L with $0 \leq L \leq N$. When $L = 0$ we have have $D = \varnothing$, possible near the end of a game.
2. The *discard pile* is a sequence $Z = (c_i)_{i=1}^{M}$ with $c_i \in \text{Cards}(H)$ such that $0 \leq M \leq N$. At the start of a game we have $M = 0$, giving us $Z = \varnothing$.
3. The *hand* of player i is a sequence $h_i = (c_j)_{j=1}^{h^{(t)}}$ with $c_j \in \text{Cards}(H)$. The length of the sequence, $h^{(t)}$, is usually equal to the initial hand size h, except maybe near the end of a game. When the deck is empty it is possible that some players may have $h^{(t)} = h - 1$. We denote all hands by $\boldsymbol{h} = (h_1, \ldots, h_p)$.
4. The *fireworks* $f_y = ((j, y))_{j=1}^{\nu_y}$ are strictly increasing sequences with respect to the order of ranks denoted here by j. For every color $y \in \{1, \ldots, k\}$ there is one sequence which consists of ν_y elements $(j, y) \in \text{Cards}(H)$ for some $0 \leq \nu_y \leq n$. Initially we have $\nu_y = 0$ for all y, meaning $f_y = \varnothing$. We denote all fireworks by $\boldsymbol{f} = (f_1, \ldots, f_k)$. An example of a firework is $((1, 1), (2, 1), (3, 1))$.

In the rest of this report, a single game of Hanabi will often be called an *episode*.

3.2 Deep Reinforcement Learning

By applying the power of neural networks to reinforcement learning, researchers have created tools that are able to learn well performing policies for many different games [6]. We will first define our setup of the reinforcement learning problem.

3.2.1 Reinforcement Learning

The reinforcement learning (RL) problem generally consists of an *agent* that needs to choose *actions* inside a particular *environment* in order to maximize the expected total *rewards* that it will receive. This interactive process is visualized in Figure 3.1. The environment is often modeled as a Markov decision process (MDP) consisting of a state space, action space, transition function, reward function, and sometimes an episode horizon and initial state distribution as well.

In the full version of Hanabi, we are dealing with a multi-agent reinforcement learning problem with imperfect information, which needs an extension of the MDP. The mathematical framework

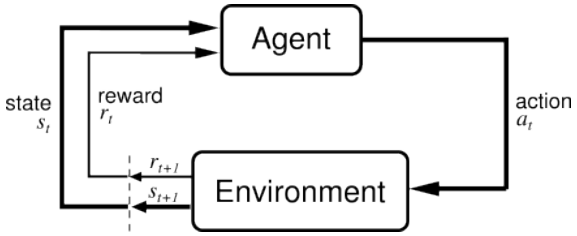


Figure 3.1: The interactive process of reinforcement learning [40].

Table 3.1: Different mathematical frameworks (with examples of games in parentheses).

	single-agent	multi-agent identical rewards	multi-agent different rewards
fully observable	MDP (PacMan, Tetris)	Dec-MDP or MMDP[1] (Pandemic)	SG[2] (Chess, Go)
partially observable	POMDP (Minesweeper)	Dec-POMDP (Hanabi)	POSG[3] (Poker)

of a Dec-POMDP (decentralised partially observable Markov decision process) is regularly used for this [21]. Hanabi is perfectly fit for this setting, see Table 3.1. In Section B.1 of the Appendix we go deeper into the Dec-POMDP for Hanabi, but we will not use it in the main body of this report.

Since we will be working with the simplified version of Hanabi and stay in the self-play domain, we can model our reinforcement learning problem as an MDP. We will now elaborate on the two arguments why we can reasonably go from a Dec-POMDP for full Hanabi to an MDP in our case.

(i) In simplified Hanabi there is no partial observability anymore, because the private observations of all players are equal to the public information. This turns the Dec-POMDP into a Dec-MDP, according to Table 3.1. To be complete: the order of cards in the deck is still missing information, but we get around this by modelling the deck as part of the stochasticity of the transition function.

(ii) To go from a Dec-MDP to an MDP, we must turn the multi-agent problem into a single-agent one. We simply let one agent control all the players in the game, which is possible since all players receive the same rewards and we stick to self-play.

Our MDP representing the simplified Hanabi environment becomes $E = \langle \mathcal{S}, \mathcal{A}, T, R, \eta, \rho_0 \rangle$. We describe the elements of the 6-tuple below.

1. $\mathcal{S}$ is the finite set of states in which the environment can be.
2. $\mathcal{A}$ is the finite set of actions the agent can choose from. For a particular state $s \in \mathcal{S}$, some actions $a \in \mathcal{A}$ may be illegal. With $\mathcal{A}_{\text{legal}}(s) \subseteq \mathcal{A}$ we denote the set of legal actions corresponding to state s.
3. $T : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$ is the transition function, which is stochastic in our model of Hanabi. The probabilities depend on the cards that are still in the deck. Since the deck is shuffled, we assume a uniform distribution over the cards that are left.
4. $R : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$ is the immediate reward function, which is deterministic. $\mathcal{R}$ is the set of possible immediate rewards. As Hanabi is a collaborative game, the rewards are identical for all players. They can be seen as a single reward signal going to our single controlling agent. We initially choose the rewards to be the change in score of the Hanabi game. This gives $\mathcal{R} = \{+1, 0, -1, -2, \ldots, -24\}$, since an action that results in the loss of the third life token makes the cumulative score go down to zero.
5. η is the horizon of the problem. Since the length of a Hanabi episode is not fixed, we define our horizon to be infinite. We extend the state space $\mathcal{S}$ with one more state s_{END}, which is

[1] Multiagent Markov decision process.
[2] Stochastic game. Despite the name, the environment does not necessarily have to be stochastic.
[3] Partially observable stochastic game.

an absorbing state with reward 0 in every time step, where all episodes reside once the game is finished.

6. $\rho_0 \in \mathcal{P}(\mathcal{S})$ is the initial state distribution at time step $t = 0$. Thus $\rho_0(s)$ gives the probability of starting in state s, which is only non-zero for states corresponding to initial configurations of Hanabi.

During an episode of Hanabi a stochastic sequence such as the following will appear:

$$(s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots).$$

The state $s_t \in \mathcal{S}$ captures the configuration of the game at time step t. It contains the fireworks, the hand of every player, the discard pile, the number of hint and life tokens left over, the remaining deck size[4] and the player who's turn it is: $s_t = (\boldsymbol{f}, \boldsymbol{h}, Z, m, l, d, i)$. Each element may also have a subscript t to clarify the time step if shown separately from s_t.

At every time step t an action a_t is taken by the agent, in which it must play or discard a card from the hand of player i_t or give a hint about the cards of other players $j \neq i_t$. In simplified Hanabi hinting for information is unnecessary, but it may still be a useful action in order to pass and give the turn to another player.

Objective

The agent chooses actions by sampling from its policy π. This policy is allowed to be stochastic, so we define π to be a function from states to probability distributions over the action space:

$$\pi : \mathcal{S} \longrightarrow \mathcal{P}(\mathcal{A})$$

with

$$\pi(s) = \boldsymbol{a}$$

where $\boldsymbol{a}$ can be seen as a vector of action selection probabilities with length $|\mathcal{A}|$. Next to this, we will also use another notation throughout the report:

$$\pi(a|s) = p$$

where $p \in [0, 1]$ denotes the probability of selecting action a in state s with policy π.[5]

In reinforcement learning the goal is to get as close as possible to the optimal policy π^*, which maximizes the expected total return. The total return is given by g_0 (or G_0 when referring to the random variable) where the return from time step t onward is given by

$$g_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3.1}$$

where $\gamma \in (0, 1]$ is an optional discount factor. The infinite sum even converges when $\gamma = 1$ in our case, as it is practically a finite sum. Every Hanabi game must terminate after a maximum of 89 turns[6], after which the final absorbing state of our MDP produces a reward of 0 in every time step.

Our objective function which we want to maximize is the expected total return:

$$J(\pi) = \mathbb{E}_\pi\big[G_0\big] \tag{3.2}$$

[4] Just the total number of cards left in the deck. This is public information in Hanabi.
[5] So technically, in that case we have that the function π is defined on a different domain: $\pi : \mathcal{S} \times \mathcal{A} \longrightarrow [0, 1]$.
[6] See Lemma 4.

where the expectation is over the whole trajectory of possible states, actions, and rewards in an episode. Their respective random variables for time step t are given by S_t, A_t, R_t.

The objective function can also be written as

$$J(\pi) = \sum_{s_0 \in \mathcal{S}} \rho_0(s_0)\, v^\pi(s_0) \tag{3.3}$$

where v^π is the true value function for π defined as

$$v^\pi(s) = \mathbb{E}_\pi\big[G_t \,|\, S_t = s\big]. \tag{3.4}$$

The optimal policy maximizes the objective function: $\pi^* = \max_\pi J(\pi)$.

3.2.2 Value-based

The objective function value $J(\pi)$ for a particular policy π is generally unknown. It can be estimated with prediction methods that often use a value function. Algorithms that use such an estimated value function are called value-based.

There are two main types of value functions. The *state* value function $v^\pi : \mathcal{S} \to \mathbb{R}$ is defined as the expected total return from state s onward, given that the agent will follow policy π. Its definition is given in (3.4).

Another value function that agents could estimate is the *state-action* value function, often called $q^\pi(s, a)$ for the *quality* of a particular action. This maps from state-action pairs to real numbers, $q^\pi : \mathcal{S} \times A \to \mathbb{R}$. It provides the expected total return from state s, given that the agent first takes action a and follows policy π in all subsequent steps:

$$q^\pi(s, a) = \mathbb{E}_\pi\big[G_t \,|\, S_t = s, A_t = a\big]. \tag{3.5}$$

Reinforcement learning algorithms that are purely value-based often determine and improve their policy by acting greedily. This means their policy will select the action which has the highest Q-value, where Q is the algorithm's estimate of the true state-action value function.

To make sure that agents explore enough of the state space, an ε-greedy policy is also possible. The agent then makes a random move with probability ε, and otherwise chooses the action that maximizes expected return. The probability of selecting action a in state s becomes:

$$\pi(a|s) = \begin{cases} \frac{\varepsilon}{|\mathcal{A}|} + 1 - \varepsilon & \text{if } a = \arg\max_{a'} Q(s, a') \\ \frac{\varepsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases}.$$

The state spaces of certain problems can be so large, that it is unpractical to store a value for each state, let alone each state-action pair. In Hanabi the number of states is at least the number of initial deck configurations, which is $50!/((3!)^5(2!)^{15}) \approx 10^{56}$, intractable for tabular methods. We would have to use a function approximator to represent our value function, so that we only need to store the corresponding parameters or weights $\boldsymbol{w}$. Our estimated value function becomes $V^\pi_{\boldsymbol{w}}$ or $Q^\pi_{\boldsymbol{w}}$, where $\boldsymbol{w}$ comes from the parameter space W.

When estimating a value function with $V^\pi_{\boldsymbol{w}}$, the objective is to come as close as possible to the true value function v^π. We can naturally define a cost function to minimize:

$$C(\boldsymbol{w}) = \mathbb{E}_\pi\big[(v^\pi(S) - V^\pi_{\boldsymbol{w}}(S))^2\big].$$

If we knew v^π we could compute the gradient of the cost function and use it for gradient descent with step $\Delta\boldsymbol{w}$. The step size is determined (partially) by the learning rate α.

$$\Delta\boldsymbol{w} = \alpha \nabla_{\boldsymbol{w}} C(\boldsymbol{w})$$

$$= -2\alpha\, \mathbb{E}_\pi\big[(v^\pi(S) - V^\pi_{\boldsymbol{w}}(S))\, \nabla_{\boldsymbol{w}} V^\pi_{\boldsymbol{w}}(S)\big].$$

In stochastic gradient descent (SGD) a sample or batch of samples is used:

$$\Delta \boldsymbol{w} = -2\alpha\, (v^\pi(s) - V^\pi_{\boldsymbol{w}}(s))\, \nabla_{\boldsymbol{w}} V^\pi_{\boldsymbol{w}}(s). \tag{3.6}$$

However, the true value function $v^\pi(s)$ is unknown and not given to us by any supervisor. The agent must learn from experience, and use this as a substitute target. There exist multiple algorithms for this, of which we will describe a few briefly, based on [37].

In Monte Carlo (MC) learning the agent runs a full episode to gather experience, after which it improves its value function (and then its policy with an ε-greedy update). For each visited state s_t, the algorithm computes the corresponding return g_t, giving it a set of "training data":

$$\langle s_1, g_1 \rangle, \langle s_2, g_2 \rangle, \ldots, \langle s_T, g_T \rangle.$$

The return g_t from each "data point" $\langle s_t, g_t \rangle$ is an unbiased estimator for $v^\pi(s_t)$. It can be used to train the function approximator with SGD:

$$\Delta \boldsymbol{w} = \alpha\, (g_t - V^\pi_{\boldsymbol{w}}(s_t))\, \nabla_{\boldsymbol{w}} V^\pi_{\boldsymbol{w}}(s_t).$$

Temporal Difference (TD) learning speeds up the process by updating the value function after every step, instead of waiting for the end of the episode. In TD(0) a one-step look-ahead is used to form an estimator of v^π. We execute one action in the environment, store the received reward and add our discounted value function of the next state: $r_{t+1} + \gamma V^\pi_{\boldsymbol{w}}(s_{t+1})$. Together this sum is called the TD target, which we substitute for v^π in (3.6):

$$\Delta \boldsymbol{w} = \alpha \left(r_{t+1} + \gamma V^\pi_{\boldsymbol{w}}(s_{t+1}) - V^\pi_{\boldsymbol{w}}(s_t)\right) \nabla_{\boldsymbol{w}} V^\pi_{\boldsymbol{w}}(s_t).$$

TD(λ) provides a mix between MC and TD(0) by including information from all steps ahead. By the use of eligibility traces this can still be done online, so there is no need to wait until the end of an episode. The combined target named g_t^λ is substituted for v^π:

$$\Delta \boldsymbol{w} = \alpha \left(g_t^\lambda - V^\pi_{\boldsymbol{w}}(s_t)\right) \nabla_{\boldsymbol{w}} V^\pi_{\boldsymbol{w}}(s_t).$$

3.2.3 Policy-based

Another class of algorithms focuses on improving the policy π directly, without requiring a value function to choose actions. This makes it easy to define a stochastic policy, which can be useful or even optimal in partially observable environments [38].

Similar to the value-based approach above, we also parameterize our policy to cope with the large state space. The function approximator that represents our policy is denoted by $\pi_{\boldsymbol{\theta}}$, where the parameters come from a separate parameter space: $\boldsymbol{\theta} \in \Theta$.

We use the objective function $J(\pi_{\boldsymbol{\theta}})$ as defined in (3.3). We want to find the optimal parameters $\boldsymbol{\theta}$ for the policy, such that $J(\pi_{\boldsymbol{\theta}})$ is maximized. A useful optimization algorithm to approach this problem with is gradient ascent. The gradient of $J(\pi_{\boldsymbol{\theta}})$ which is needed for this algorithm can be difficult to compute, but fortunately there exists a useful theorem:

Theorem 1 (Policy Gradient Theorem)**.** *Let $J(\pi_{\boldsymbol{\theta}})$ be the expected total return of a policy $\pi_{\boldsymbol{\theta}}$, $\mu^{\pi_{\boldsymbol{\theta}}}(x)$ be the discounted state distribution under policy $\pi_{\boldsymbol{\theta}}$, and $q^{\pi_{\boldsymbol{\theta}}}(s, a)$ be the true action-value function under $\pi_{\boldsymbol{\theta}}$. These functions are defined by:*

$$J(\pi_{\boldsymbol{\theta}}) = \sum_{s \in \mathcal{S}} \rho_0(s)\, v^{\pi_{\boldsymbol{\theta}}}(s),$$

$$\mu^{\pi_{\boldsymbol{\theta}}}(x) = \sum_{k=0}^{\infty} \gamma^k \, \mathbb{P}(S_k = x \,|\, \pi_{\boldsymbol{\theta}}, \rho_0),$$
$$q^{\pi_{\boldsymbol{\theta}}}(s,a) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\big[G_t \,|\, S_t = s, A_t = a\big].$$

Assume that $\gamma \in (0,1)$, the size of the action space $|\mathcal{A}|$ is finite, the action-value function is bounded: $|q^{\pi_{\boldsymbol{\theta}}}| \leq L_q$, and the gradients are bounded: $\|\nabla_{\boldsymbol{\theta}} v^{\pi_{\boldsymbol{\theta}}}\| \leq L_v$ and $\|\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}\| \leq L_\pi$ for some $L_q, L_v, L_\pi \in \mathbb{R}$. Then we have:

$$\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \sum_{x \in \mathcal{S}} \mu^{\pi_{\boldsymbol{\theta}}}(x) \sum_{a \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|x) \cdot q^{\pi_{\boldsymbol{\theta}}}(x,a).$$

Sutton and Barto prove this important theorem in their book on reinforcement learning [41, sec. 13.2]. Since their proof is quite compact, we attempt to write a more detailed version here. We added more assumptions to the theorem, compared to [41], which we use in our proof. Some of these assumptions might not be necessary when providing a different proof.

Before we prove the policy gradient theorem, we introduce two useful lemmas. We are going to use shorthand notation in regards to the random variables S_t and A_t, representing the state and action at time step t respectively. With $\pi(a|s)$ we mean $\pi(A_t = a \,|\, S_t = s) = \mathbb{P}(A_t = a \,|\, \pi, S_t = s)$ and $\mathbb{P}(s' \,|\, s, a)$ is equal to $\mathbb{P}(S_{t+1} = s' \,|\, S_t = s, A_t = a)$, where the right time step index t should be clear from context.

Lemma 2 (Reaching state s_k)**.** *We let $\mathbb{P}(S_k = s_k \,|\, \pi, \rho_0)$ denote the probability of reaching state s_k under policy π in exactly k steps from the start of an episode. We have $\mathbb{P}(S_0 = s_0 \,|\, \pi, \rho_0) = \rho_0(s_0)$ and for $k \geq 1$:*

$$\mathbb{P}(S_k = s_k \,|\, \pi, \rho_0) = \sum_{\substack{s_0, s_1, \ldots, s_{k-1} \in \mathcal{S} \\ a_0, a_1, \ldots, a_{k-1} \in \mathcal{A}}} \rho_0(s_0) \prod_{t=0}^{k-1} \Big[\pi(a_t|s_t) \; \mathbb{P}(s_{t+1} \,|\, s_t, a_t)\Big] \tag{3.7}$$

or in terms of the previous time step:

$$\mathbb{P}(S_k = s_k \,|\, \pi, \rho_0) = \sum_{\substack{s_{k-1} \in \mathcal{S} \\ a_{k-1} \in \mathcal{A}}} \mathbb{P}(S_{k-1} = s_{k-1} \,|\, \pi, \rho_0) \; \pi(a_{k-1}|s_{k-1}) \; \mathbb{P}(s_k \,|\, s_{k-1}, a_{k-1}). \tag{3.8}$$

Proof of Lemma 2. We will use proof by induction for both equations. The notation $\mathbb{P}(S_k = s_k \,|\, \pi, \rho_0)$ will be shortened to $\mathbb{P}(S_k = s_k)$, but one should keep in mind that in every $\mathbb{P}(\cdot)$ in this proof, π and ρ_0 are given.

For $k = 0$: $\mathbb{P}(S_0 = s_0) = \rho_0(s_0)$, as no actions or state transitions could have happened yet. The probability of reaching state x in zero steps is equal to the value of the initial distribution $\rho_0(\cdot)$ at the state s_0.

For $k = 1$, which is the base case of our induction proof on both (3.7) and (3.8), we have:

$$\begin{aligned}
\mathbb{P}(S_1 = s_1) &= \sum_{s_0 \in \mathcal{S}} \mathbb{P}(S_1 = s_1 \,|\, S_0 = s_0) \; \mathbb{P}(S_0 = s_0) \\
&= \sum_{s_0 \in \mathcal{S}} \mathbb{P}(S_1 = s_1 \,|\, S_0 = s_0) \; \rho_0(s_0) \\
&= \sum_{s_0 \in \mathcal{S}} \sum_{a_0 \in \mathcal{A}} \mathbb{P}(S_1 = s_1 \,|\, S_0 = s, A_0 = a_0) \; \mathbb{P}(A_0 = a_0 \,|\, S_0 = s_0) \; \rho_0(s_0)
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{s_0 \in \mathcal{S}} \sum_{a_0 \in \mathcal{A}} \mathbb{P}(s_1 \,|\, s_0, a_0) \pi(a_0|s_0)\ \rho_0(s_0) \\
&= \sum_{\substack{s_0 \in \mathcal{S} \\ a_0 \in \mathcal{A}}} \rho_0(s_0)\ \pi(a_0|s_0)\ \mathbb{P}(s_1 \,|\, s_0, a_0) \\
&= \sum_{\substack{s_0 \in \mathcal{S} \\ a_0 \in \mathcal{A}}} \rho_0(s_0) \prod_{t=0}^{0} \Big[\pi(a_t|s_t)\ \mathbb{P}(s_{t+1} \,|\, s_t, a_t)\Big] && \text{(for (3.7))} \\
&= \sum_{\substack{s_0 \in \mathcal{S} \\ a_0 \in \mathcal{A}}} \mathbb{P}(S_0 = s_0)\ \pi(a_0|s_0)\ \mathbb{P}(s_1 \,|\, s_0, a_0). && \text{(for (3.8))}
\end{aligned}
$$

Now let us assume that (3.7) holds for $k = n$, which we call our Induction Hypothesis (IH). We will prove in this case that the same equation holds for $k = n + 1$.

$$
\begin{aligned}
\mathbb{P}(S_{n+1} = s_{n+1}) &= \sum_{s_n \in \mathcal{S}} \mathbb{P}(S_{n+1} = s_{n+1} \,|\, S_n = s_n)\ \mathbb{P}(S_n = s_n) \\
&= \sum_{s_n \in \mathcal{S}} \sum_{a_n \in \mathcal{A}} \mathbb{P}(S_{n+1} = s_{n+1} \,|\, S_n = s_n, A_n = a_n)\ \mathbb{P}(A_n = a_n \,|\, S_n = s_n)\ \mathbb{P}(S_n = s_n) \\
&= \sum_{s_n \in \mathcal{S}} \sum_{a_n \in \mathcal{A}} \mathbb{P}(s_{n+1} \,|\, s_n, a_n)\ \pi(a_n|s_n)\ \mathbb{P}(S_n = s_n) && \text{(for (3.8))} \\
&\overset{\text{IH}}{=} \sum_{s_n \in \mathcal{S}} \sum_{a_n \in \mathcal{A}} \mathbb{P}(s_{n+1}|s_n, a_n)\, \pi(a_n|s_n) \sum_{\substack{s_0,..,s_{n-1} \in \mathcal{S} \\ a_0,..,a_{n-1} \in \mathcal{A}}} \rho_0(s_0) \prod_{t=0}^{n-1} \Big[\pi(a_t|s_t)\, \mathbb{P}(s_{t+1}|s_t, a_t)\Big] \\
&= \sum_{\substack{s_0,..,s_{n-1},s_n \in \mathcal{S} \\ a_0,..,a_{n-1},a_n \in \mathcal{A}}} \rho_0(s_0) \prod_{t=0}^{n} \Big[\pi(a_t|s_t)\ \mathbb{P}(s_{t+1} \,|\, s_t, a_t)\Big]. && \text{(for (3.7))}
\end{aligned}
$$

This concludes the proof by induction. ⏻

The next lemma gives an expression for the gradient of the state-value function: $\nabla_{\boldsymbol{\theta}} v^{\pi_{\boldsymbol{\theta}}}$.

Lemma 3 (Unpacking ∇v)**.** *Let $v^{\pi_{\boldsymbol{\theta}}}$ be the true value function for a policy $\pi_{\boldsymbol{\theta}}$. Then we have:*

$$
\nabla_{\boldsymbol{\theta}} v^{\pi_{\boldsymbol{\theta}}}(s) = \sum_{a \in \mathcal{A}} \Big[\nabla \pi_{\boldsymbol{\theta}}(a|s)\, q^{\pi_{\boldsymbol{\theta}}}(s, a) + \pi_{\boldsymbol{\theta}}(a|s) \sum_{s' \in \mathcal{S}} \mathbb{P}(s' \,|\, s, a) \gamma\, \nabla_{\boldsymbol{\theta}} v^{\pi_{\boldsymbol{\theta}}}(s')\Big].
$$

Proof of Lemma 3. To keep notation simple, just like in [41] we leave it implicit in all cases that π is a function of $\boldsymbol{\theta}$ and all gradients are with respect to $\boldsymbol{\theta}$. We use the linearity of the gradient a couple of times throughout the proof.

$$
\begin{aligned}
\nabla v^{\pi}(s) &= \nabla \sum_{a \in \mathcal{A}} \pi(a|s)\, q^{\pi}(s, a) \\
&= \sum_{a \in \mathcal{A}} \nabla \big[\pi(a|s)\, q^{\pi}(s, a)\big] \\
&= \sum_{a \in \mathcal{A}} \big[\nabla \pi(a|s)\, q^{\pi}(s, a) + \pi(a|s)\, \nabla q^{\pi}(s, a)\big] \\
&= \sum_{a \in \mathcal{A}} \Big[\nabla \pi(a|s)\, q^{\pi}(s, a) + \pi(a|s)\, \nabla \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} \mathbb{P}(s', r \,|\, s, a)\big(r + \gamma v^{\pi}(s')\big)\Big]
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{a\in\mathcal{A}} \left[\nabla\pi(a|s)\, q^\pi(s,a) + \pi(a|s) \nabla \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} \big(\mathbb{P}(s', r \mid s, a) r + \mathbb{P}(s', r \mid s, a) \gamma v^\pi(s') \big) \right] \\
&= \sum_{a\in\mathcal{A}} \left[\nabla\pi(a|s)\, q^\pi(s,a) + \pi(a|s) \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} \Big(\nabla \big[\mathbb{P}(s', r \mid s, a) r \big] + \nabla \big[\mathbb{P}(s', r \mid s, a) \gamma v^\pi(s') \big] \Big) \right] \\
&= \sum_{a\in\mathcal{A}} \left[\nabla\pi(a|s)\, q^\pi(s,a) + \pi(a|s) \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} \nabla \big(\mathbb{P}(s', r \mid s, a) \gamma v^\pi(s') \big) \right] \\
&= \sum_{a\in\mathcal{A}} \left[\nabla\pi(a|s)\, q^\pi(s,a) + \pi(a|s) \sum_{s'\in\mathcal{S}} \nabla \sum_{r\in\mathcal{R}} \mathbb{P}(s', r \mid s, a) \gamma v^\pi(s') \right] \\
&= \sum_{a\in\mathcal{A}} \left[\nabla\pi(a|s)\, q^\pi(s,a) + \pi(a|s) \sum_{s'\in\mathcal{S}} \nabla \big(\mathbb{P}(s' \mid s, a) \gamma v^\pi(s') \big) \right] \\
&= \sum_{a\in\mathcal{A}} \left[\nabla\pi(a|s)\, q^\pi(s,a) + \pi(a|s) \sum_{s'\in\mathcal{S}} \mathbb{P}(s' \mid s, a) \gamma \nabla v^\pi(s') \right].
\end{aligned}
$$

Notice that both $\mathbb{P}(s', r \mid s, a)$ and r do not depend on the policy parameters $\boldsymbol{\theta}$, which makes $\nabla_{\boldsymbol{\theta}} \mathbb{P}(s', r \mid s, a) r = 0$. Also, $\sum_{r\in\mathcal{R}} \mathbb{P}(s', r \mid s, a) = \mathbb{P}(s' \mid s, a)$. ⏻

The proof of Lemma 3 shows how the gradient of the value function can be unpacked, such that we look at a state one step further into the future (s' follows from s). We will apply this lemma multiple times in the proof of the policy gradient theorem.

Proof of the Policy Gradient Theorem (Theorem 1). We use the same notation as in the proof of Lemma 3. We will apply Lemma 3 a few times, after which we use Lemma 2 to further simplify the expression.

$$
\begin{aligned}
\nabla J(\pi) &= \nabla \sum_{s\in\mathcal{S}} \rho_0(s)\, v^\pi(s) \\
&= \sum_{s\in\mathcal{S}} \nabla \big[\rho_0(s)\, v^\pi(s) \big] && \text{(by linearity of } \nabla) \\
&= \sum_{s\in\mathcal{S}} \rho_0(s)\, \nabla v^\pi(s) && (\rho_0 \text{ does not depend on } \boldsymbol{\theta}) \\
&= \sum_{s\in\mathcal{S}} \rho_0(s) \sum_{a\in\mathcal{A}} \left[\nabla\pi(a|s)\, q^\pi(s,a) + \pi(a|s) \sum_{s'\in\mathcal{S}} \mathbb{P}(s' \mid s, a) \gamma \nabla v^\pi(s') \right] && \text{(Lemma 3)} \\
&= \sum_{s\in\mathcal{S}} \rho_0(s) \sum_{a\in\mathcal{A}} \nabla\pi(a|s)\, q^\pi(s,a) \\
&\quad + \sum_{s\in\mathcal{S}} \rho_0(s) \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathbb{P}(s' \mid s, a) \gamma \nabla v^\pi(s') \\
&= \sum_{s\in\mathcal{S}} \rho_0(s) \sum_{a\in\mathcal{A}} \nabla\pi(a|s)\, q^\pi(s,a) \\
&\quad + \sum_{s\in\mathcal{S}} \rho_0(s) \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathbb{P}(s' \mid s, a) \gamma \\
&\quad \cdot \sum_{a'\in\mathcal{A}} \left[\nabla\pi(a'|s')\, q^\pi(s',a') + \pi(a'|s') \sum_{s''\in\mathcal{S}} \mathbb{P}(s'' \mid s', a') \gamma \nabla v^\pi(s'') \right] && \text{(Lemma 3)}
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{s\in\mathcal{S}} \rho_0(s) \sum_{a\in\mathcal{A}} \nabla\pi(a|s)\, q^\pi(s,a) \\
&\quad + \sum_{s\in\mathcal{S}} \rho_0(s) \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathbb{P}(s' \mid s,a)\gamma \sum_{a'\in\mathcal{A}} \nabla\pi(a'|s')\, q^\pi(s',a') \\
&\quad + \sum_{s\in\mathcal{S}} \rho_0(s) \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathbb{P}(s' \mid s,a)\gamma \sum_{a'\in\mathcal{A}} \pi(a'|s') \sum_{s''\in\mathcal{S}} \mathbb{P}(s'' \mid s',a')\gamma\nabla v^\pi(s'')
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{s\in\mathcal{S}} \rho_0(s) \sum_{a\in\mathcal{A}} \nabla\pi(a|s)\, q^\pi(s,a) \\
&\quad + \sum_{s\in\mathcal{S}} \sum_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \rho_0(s)\ \pi(a|s)\ \mathbb{P}(s' \mid s,a)\,\gamma \sum_{a'\in\mathcal{A}} \nabla\pi(a'|s')\, q^\pi(s',a') \\
&\quad + \sum_{s\in\mathcal{S}} \sum_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \sum_{a'\in\mathcal{A}} \sum_{s''\in\mathcal{S}} \rho_0(s)\ \pi(a|s)\ \mathbb{P}(s' \mid s,a)\ \pi(a'|s')\ \mathbb{P}(s'' \mid s',a')\,\gamma^2\,\nabla v^\pi(s'')
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{s} \rho_0(s) \sum_{a\in\mathcal{A}} \nabla\pi(a|s)\, q^\pi(s,a) && \text{(Lemma 3)} \\
&\quad + \sum_{a,s,s'} \rho_0(s)\ \pi(a|s)\ \mathbb{P}(s' \mid s,a)\,\gamma \sum_{a'\in\mathcal{A}} \nabla\pi(a'|s')\, q^\pi(s',a') \\
&\quad + \sum_{a,a',s,s',s''} \rho_0(s)\,\pi(a|s)\,\mathbb{P}(s' \mid s,a)\,\pi(a'|s')\ \mathbb{P}(s'' \mid s',a')\,\gamma^2 \sum_{a''\in\mathcal{A}} \nabla\pi(a''|s'')\, q^\pi(s'',a'') \\
&\quad + \sum_{a,a',a'',s,s',s'',s'''} \rho_0(s)\pi(a|s)\mathbb{P}(s'|s,a)\pi(a'|s')\mathbb{P}(s''|s',a')\pi(a''|s'')\mathbb{P}(s'''|s'',a'')\gamma^3\nabla v^\pi(s''')
\end{aligned}
$$

Notice that the string of action selection probabilities $\pi(a|s)$ and state transition probabilities $\mathbb{P}(s' \mid s,a)$ together form one probability. As shown in Lemma 2, this is the probability of reaching a particular state x under policy π in exactly k time steps from the start of an episode, which we denote by $\mathbb{P}(S_k = x \mid \pi, \rho_0)$ or $\mathbb{P}(S_k = x)$ in short.

$$
\begin{aligned}
&= \sum_{s} \mathbb{P}(S_0 = s) \sum_{a\in\mathcal{A}} \nabla\pi(a|s)\, q^\pi(s,a) \\
&\quad + \sum_{s'} \mathbb{P}(S_1 = s')\,\gamma \sum_{a'\in\mathcal{A}} \nabla\pi(a'|s')\, q^\pi(s',a') \\
&\quad + \sum_{s''} \mathbb{P}(S_2 = s'')\,\gamma^2 \sum_{a''\in\mathcal{A}} \nabla\pi(a''|s'')\, q^\pi(s'',a'') \\
&\quad + \sum_{s'''} \mathbb{P}(S_3 = s''')\,\gamma^3\,\nabla v^\pi(s''')
\end{aligned}
$$

$$
= \sum_{k=0}^{2} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x) q^\pi(x,a) + \sum_{s\in\mathcal{S}} \mathbb{P}(S_3 = s)\gamma^3 \nabla v^\pi(s).
$$

Now we have unpacked ∇v^π three times, but we can continue this arbitrarily often. Suppose we have unpacked K times:

$$
\nabla J(\pi) = \sum_{k=0}^{K-1} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x) q^\pi(x,a) + \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K \nabla v^\pi(s).
$$

Then we can also unpack $K+1$ times:

$$\nabla J(\pi) = \sum_{k=0}^{K-1} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x)\, q^\pi(x,a) + \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K \nabla v^\pi(s)$$

$$= \sum_{k=0}^{K-1} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x)\, q^\pi(x,a) \qquad \text{(Lemma 3)}$$
$$+ \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K \sum_{a\in\mathcal{A}} \left[\nabla\pi(a|s)\, q^\pi(s,a) + \pi(a|s) \sum_{s'\in\mathcal{S}} \mathbb{P}(s' \mid s,a)\gamma \nabla v^\pi(s') \right]$$

$$= \sum_{k=0}^{K-1} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x)\, q^\pi(x,a)$$
$$+ \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K \sum_{a\in\mathcal{A}} \nabla\pi(a|s)\, q^\pi(s,a)$$
$$+ \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathbb{P}(s' \mid s,a)\gamma \nabla v^\pi(s')$$

$$= \sum_{k=0}^{K} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x)\, q^\pi(x,a)$$
$$+ \sum_{s\in\mathcal{S}} \sum_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \mathbb{P}(S_K = s)\ \pi(a|s)\ \mathbb{P}(s' \mid s,a)\, \gamma^{K+1} \nabla v^\pi(s')$$

$$= \sum_{k=0}^{K} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x)\, q^\pi(x,a) + \sum_{s'\in\mathcal{S}} \mathbb{P}(S_{K+1} = s')\gamma^{K+1} \nabla v^\pi(s').$$

(by Equation (3.8))

To continue, we will take the limit as we want to unpack all ∇v^π.

$$\nabla J(\pi) = \lim_{K\to\infty} \left[\underbrace{\sum_{k=0}^{K-1} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x)\, q^\pi(x,a)}_{\text{First term}} + \underbrace{\sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K \nabla v^\pi(s)}_{\text{Second term}} \right]$$

We are allowed to split this limit if the limit of both terms exists, which we will check below.

$$= \lim_{K\to\infty} \sum_{k=0}^{K-1} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x)\, q^\pi(x,a) + \lim_{K\to\infty} \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K \nabla v^\pi(s).$$

When looking at the second limit, we notice that

$$\begin{aligned} \left\| \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K \nabla v^\pi(s) \right\| &\le \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K \|\nabla v^\pi(s)\| \\ &\le \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K L_v \\ &= \gamma^K L_v \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s) \end{aligned}$$

$$= \gamma^K L_v.$$

Thus we know that

$$0 \leq \lim_{K\to\infty} \left\| \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K \nabla v^\pi \right\| \leq \lim_{K\to\infty} \gamma^K L_v = 0.$$

So we must have

$$\lim_{K\to\infty} \sum_{s\in\mathcal{S}} \mathbb{P}(S_K = s)\gamma^K \nabla v^\pi(s) = \mathbf{0}.$$

Therefore,

$$\begin{aligned}\nabla J(\pi) &= \lim_{K\to\infty} \sum_{k=0}^{K-1} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x)\, q^\pi(x,a) \\ &= \sum_{k=0}^{\infty} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x)\, q^\pi(x,a). \end{aligned} \tag{3.9}$$

We may change the order of summation if the infinite sum of (3.9) converges absolutely. We will prove this by showing that the absolute sum is bounded from above. Combined with the fact that the absolute sum is an increasing sequence, this means that the sum must converge absolutely.

$$\begin{aligned}\sum_{k=0}^{\infty} \left\| \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x) q^\pi(x,a) \right\| &\leq \sum_{k=0}^{\infty} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \|\nabla\pi(a|x)\|\ |q^\pi(x,a)| \\ &\leq \sum_{k=0}^{\infty} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} L_\pi L_q \\ &= \sum_{k=0}^{\infty} \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x)\gamma^k\, |\mathcal{A}|\, L_\pi L_q \\ &= \sum_{k=0}^{\infty} \gamma^k\, |\mathcal{A}|\, L_\pi L_q \sum_{x\in\mathcal{S}} \mathbb{P}(S_k = x) \\ &= \sum_{k=0}^{\infty} \gamma^k\, |\mathcal{A}|\, L_\pi L_q \\ &= |\mathcal{A}|\, L_\pi L_q \sum_{k=0}^{\infty} \gamma^k \\ &= |\mathcal{A}|\, L_\pi L_q \frac{1}{1-\gamma}. \end{aligned}$$

Therefore, we may change the order of summation and arrive at:

$$\begin{aligned}\nabla J(\pi) &= \sum_{x\in\mathcal{S}} \sum_{k=0}^{\infty} \mathbb{P}(S_k = x)\gamma^k \sum_{a\in\mathcal{A}} \nabla\pi(a|x)\, q^\pi(x,a) \\ &= \sum_{x\in\mathcal{S}} \mu^\pi(x) \sum_{a\in\mathcal{A}} \nabla\pi(a|x)\, q^\pi(x,a). \end{aligned}$$

This concludes the proof of the policy gradient theorem.

The policy gradient as stated above has the $\mu^\pi(x)$ term, which can be difficult to use in practice. We can also compute $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$ in a slightly different way, such that this term does not appear anymore. Let us pick up the derivation of the policy gradient from Equation (3.9) and go in a different direction. We have:

$$\begin{aligned}
\nabla J(\pi) &= \sum_{k=0}^{\infty} \sum_{x \in \mathcal{S}} \mathbb{P}(S_k = x) \gamma^k \sum_{a \in \mathcal{A}} \nabla \pi(a|x)\, q^\pi(x,a) \\
&= \sum_{k=0}^{\infty} \gamma^k \sum_{x \in \mathcal{S}} \mathbb{P}(S_k = x) \sum_{a \in \mathcal{A}} \nabla \pi(a|x)\, q^\pi(x,a) \\
&= \sum_{k=0}^{\infty} \gamma^k \mathbb{E}_{X_k \sim p_k} \left[\sum_{a \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \,|\, X_k) \cdot q^{\pi_{\boldsymbol{\theta}}}(X_k, a) \right]
\end{aligned}$$

where p_k is the PMF representing the distribution over the reachability of possible states in k steps, taking into account the environment's transition probabilities and the policy's action selection probabilities. We proceed by multiplying and dividing by $\pi_{\boldsymbol{\theta}}$, in order to create another (conditional) expectation inside the existing one:

$$\begin{aligned}
\nabla J(\pi) &= \sum_{k=0}^{\infty} \gamma^k \mathbb{E}_{X_k \sim p_k} \left[\sum_{a \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a \,|\, X_k) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \,|\, X_k)}{\pi_{\boldsymbol{\theta}}(a \,|\, X_k)} \cdot q^{\pi_{\boldsymbol{\theta}}}(X_k, a) \right] \\
&= \sum_{k=0}^{\infty} \gamma^k \mathbb{E}_{X_k \sim p_k} \left[\sum_{a \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a \,|\, X_k) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \,|\, X_k) \cdot q^{\pi_{\boldsymbol{\theta}}}(X_k, a) \right] \\
&= \sum_{k=0}^{\infty} \gamma^k \mathbb{E}_{X_k \sim p_k} \left[\mathbb{E}_{A_k \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_k \,|\, X_k) \cdot q^{\pi_{\boldsymbol{\theta}}}(X_k, A_k) \Big| X_k \right] \right]. \qquad (3.10)
\end{aligned}$$

This expression with two expectations turns out to be equivalent to one expectation over state-action pairs (X_k, A_k). For ease of notation, we write $g(X, A) = \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A \,|\, X) \cdot q^{\pi_{\boldsymbol{\theta}}}(X, A)$ and inspect the expectations of (3.10). It seems like the law of total expectation is applicable here, but unfortunately we have a function $g(X, A)$ which does *not* depend on A only. Let us expand the expression:

$$\begin{aligned}
\mathbb{E}_X \Big[\mathbb{E}_A \Big[g(X,A) \Big| X \Big] \Big] &= \sum_{x \in \mathcal{S}} \mathbb{P}(X = x) \mathbb{E}_A \Big[g(X,A) \,|\, X = x \Big] \\
&= \sum_{x \in \mathcal{S}} \mathbb{P}(X = x) \sum_{a \in \mathcal{A}} \mathbb{P}(A = a \,|\, X = x)\; g(x,a) \\
&= \sum_{x \in \mathcal{S}} \sum_{a \in \mathcal{A}} \mathbb{P}(X = x)\; \mathbb{P}(A = a \,|\, X = x)\; g(x,a) \\
&= \sum_{x \in \mathcal{S}} \sum_{a \in \mathcal{A}} \mathbb{P}(A = a, X = x)\; g(x,a) \\
&= \sum_{(x,a) \in \mathcal{S} \times \mathcal{A}} \mathbb{P}(A = a, X = x)\; g(x,a) \\
&= \mathbb{E}_{(X,A)} \Big[g(X,A) \Big]
\end{aligned}$$

We end up with:

$$\nabla J(\pi) = \sum_{k=0}^{\infty} \gamma^k\, \mathbb{E}_{(X_k, A_k) \sim P_k} \Big[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_k \,|\, X_k) \cdot q^{\pi_{\boldsymbol{\theta}}}(X_k, A_k) \Big]$$

where P_k is the joint PMF stemming from p_k and $\pi_{\boldsymbol{\theta}}$. We take the sum inside the expectation to arrive at

$$\nabla J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \cdot \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \cdot q^{\pi_{\boldsymbol{\theta}}}(S_t, A_t) \right] \tag{3.11}$$

where the expectation is over the whole trajectory of possible state-action pairs (S_t, A_t).

Note that multiple papers on reinforcement learning use the similar expression:

$$\mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \cdot q^{\pi_{\boldsymbol{\theta}}}(S_t, A_t) \right]$$

as an expression for the policy gradient $\nabla J(\pi)$. This is shown by Nota and Thomas [29] to be incorrect[7] if $\gamma \neq 1$, as it leaves out the discount factor γ^t. Despite this error, this expression has been used effectively in practice.

Now that we have found a neat expression for the gradient of our objective function, we want to use it in order to improve our policy. The goal here is to adjust the current value of the parameters $\boldsymbol{\theta}_k$ such that the new values $\boldsymbol{\theta}_{k+1}$ produce a policy which performs better, meaning:

$$J(\pi_{\boldsymbol{\theta}_{k+1}}) \geq J(\pi_{\boldsymbol{\theta}_k}).$$

Updating the policy parameters is often done with the gradient ascent method:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \widehat{\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}_k})} \tag{3.12}$$

or more complex optimizers which generally build on the same principle.

Notice that the policy gradient has a hat in (3.12). That is because we do not know the true state-action value function $q^{\pi_{\boldsymbol{\theta}}}$, so we can only use an estimate of the policy gradient in practice. Section 4.1 on the Simple Policy Gradient algorithm will explain how we approximate the q-value.

3.2.4 Actor-Critic

Actor-Critic is a class of deep reinforcement learning algorithms which combine the strength of policy-based methods and value-based methods. The policy is called the actor, since it acts in the environment. The critic is often a value function, giving feedback to the actor on how good or bad a particular action was.

By adding a critic, we can adjust the expression of the policy gradient to stabilize the learning algorithm, without actually changing the value of the policy gradient. Let us first look into the reason behind the change of expression, and then check that the value of the policy gradient remains the same.

What is often used as a critic, is a second neural network which represents an approximate state value function $V^\pi_{\boldsymbol{w}}$ with parameters $\boldsymbol{w}$. The objective of this second network is to come as close as possible to the true value function v^π. If we just append v^π to the expression for $\nabla J(\pi)$ given in (3.11), we get

$$\nabla J(\pi) \stackrel{?}{=} \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \cdot \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \cdot \Big(q^{\pi_{\boldsymbol{\theta}}}(S_t, A_t) - v^{\pi_{\boldsymbol{\theta}}}(S_t) \Big) \right]. \tag{3.13}$$

[7] In fact, they show it cannot be a gradient of any function at all. This makes it unclear which objective is being optimized.

The question mark is added, because we do not know yet whether the equation still holds. We will show this later. We now shorten the expression to:

$$\nabla J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \cdot \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \cdot A^{\pi_{\boldsymbol{\theta}}}(S_t, A_t) \right] \tag{3.14}$$

where $A^{\pi_{\boldsymbol{\theta}}}$, the *advantage function*, is defined as

$$A^{\pi_{\boldsymbol{\theta}}}(s, a) = q^{\pi_{\boldsymbol{\theta}}}(s, a) - v^{\pi_{\boldsymbol{\theta}}}(s).$$

It indicates the advantage in value of taking action a from state s as compared to the average return one would expect in state s. If $A^\pi(s, a)$ is positive, it means that choosing action a is better than what our policy normally does in state s. So we want to adjust our policy parameters $\boldsymbol{\theta}$ such that the new $\pi_{\boldsymbol{\theta}}$ has a higher probability of selecting action a in state s. Vice versa for $A^\pi(s, a) < 0$.

For each state-action pair (s, a) the policy gradient in (3.14) consists of two parts:

(i) $\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s)$ determines the steepest direction of the gradient such that the probability of selecting action a in state s increases[8],

(ii) $A^\pi(s, a)$ determines the length of the gradient vector *and* whether it should point in the increase or decrease direction!

If $A^\pi(s, a) < 0$, the policy gradient now adjusts the parameters such that the probability of selecting action a is decreased. Without our addition of v^π in $\nabla J(\pi)$ this would not necessarily be the case. For example, think of a state s where any action a results in a positive value for $q^\pi(s, a)$. Then for all actions a the probability of selecting a would be increased, even though we would rather only increase the probability of selecting the optimal action.

Nevertheless, we still need to check whether we can remove the question mark from the expression in (3.13). This is indeed the case, not just for the value function $v^\pi(s)$, but for any so-called *baseline* function $b(s)$ which does not depend on an action a. A brief proof is given in [41, sec. 13.4], of which we will provide a more detailed version for the specific expression of $\nabla J(\pi)$ given in (3.11).

We have that

$$\begin{aligned}
\nabla J(\pi) &= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \cdot q^{\pi_{\boldsymbol{\theta}}}(S_t, A_t) \right] \\
&= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \cdot (q^{\pi_{\boldsymbol{\theta}}}(S_t, A_t) - b(S_t)) \right] \\
&= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \cdot q^{\pi_{\boldsymbol{\theta}}}(S_t, A_t) \right] - \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \cdot b(S_t) \right]
\end{aligned} \tag{3.15}$$

if it is true that

$$\mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \cdot b(S_t) \right] = 0. \tag{3.16}$$

[8] Actually, we should say: "such that the *log* of the probability [...] increases". However, since the log is a monotonically increasing function, this is equivalent.

We assume here that we are allowed to split the infinite sum, or in other words that both sums of the expression in (3.15) converge. Let us confirm that (3.16) holds:

$$
\begin{aligned}
&\mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \, \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \, b(S_t) \right] \\
&\quad = \sum_{t=0}^{\infty} \gamma^t \, \mathbb{E}_{(S_t, A_t) \sim P_t} \Big[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \, b(S_t) \Big] \\
&\quad = \sum_{t=0}^{\infty} \gamma^t \sum_{s_t \in \mathcal{S}, a_t \in \mathcal{A}} \mathbb{P}(S_t = s_t, A_t = a_t) \, \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \, b(s_t) \\
&\quad = \sum_{t=0}^{\infty} \gamma^t \sum_{s_t \in \mathcal{S}} \sum_{a_t \in \mathcal{A}} \mathbb{P}(S_t = s_t) \; \mathbb{P}(A_t = a_t \mid S_t = s_t) \; \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \; b(s_t) \\
&\quad = \sum_{t=0}^{\infty} \gamma^t \sum_{s_t \in \mathcal{S}} \mathbb{P}(S_t = s_t) \sum_{a_t \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a_t|s_t) \; \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \, b(s_t) \\
&\quad = \sum_{t=0}^{\infty} \gamma^t \sum_{s_t \in \mathcal{S}} \mathbb{P}(S_t = s_t) \, b(s_t) \sum_{a_t \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a_t|s_t) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}}(a_t|s_t)} \\
&\quad = \sum_{t=0}^{\infty} \gamma^t \sum_{s_t \in \mathcal{S}} \mathbb{P}(S_t = s_t) \, b(s_t) \sum_{a_t \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t|s_t) \\
&\quad = \sum_{t=0}^{\infty} \gamma^t \sum_{s_t \in \mathcal{S}} \mathbb{P}(S_t = s_t) \, b(s_t) \, \nabla_{\boldsymbol{\theta}} \sum_{a_t \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a_t|s_t) \\
&\quad = \sum_{t=0}^{\infty} \gamma^t \sum_{s_t \in \mathcal{S}} \mathbb{P}(S_t = s_t) \, b(s_t) \, \nabla_{\boldsymbol{\theta}} 1 \\
&\quad = 0.
\end{aligned}
$$

Thus, we can indeed include a baseline function $b(s)$ in the expression for the policy gradient.

One important question remains: if the value of the policy gradient does not change, why would it help to include a baseline? We are still using the policy gradient to update the parameters, as shown in (3.12).

The important detail in (3.12) is the fact that in practice we can only use an *estimate* of the policy gradient. This estimate can vary quite a lot, being much higher or lower than the actual policy gradient. By including $V^\pi_{\boldsymbol{w}}$ as a baseline this variance is reduced, which results in faster and more stable policy learning [2].

The algorithms Vanilla Policy Gradient (VPG) and Proximal Policy Optimization (PPO) as described in Sections 4.2 and 4.3 are both Actor-Critic algorithms that use a second neural network to estimate v^π.

Chapter 4

Algorithms

In this chapter we describe three deep reinforcement learning algorithms. We will go into the mathematics of each algorithm, and show how we have implemented it for Hanabi. We chose these algorithms because we were curious to see how the actor-critic approach would perform in this domain.

4.1 Simple Policy Gradient

The Simple Policy Gradient (SPG) algorithm is a method which directly adjusts the policy parameters to try to increase the value of the objective function. It is quite similar to the REINFORCE algorithm by Williams [43], as they are both Monte Carlo, Policy Gradient Control methods. However, our implementation of SPG which is based on [3] performs *one* update to the parameters $\boldsymbol{\theta}$ after a certain batch of actions has been carried out, while REINFORCE updates the parameters many times (an update for every time-step after it has played a full episode).

Whenever we update the parameters, we want to change their values in the direction of the steepest ascent of the objective function, which is given by the policy gradient $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$. This gives the basic gradient ascent update step:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}_k}).$$

In our implementations we use the slightly more complex update step provided by the Adam optimizer [23], which stands for adaptive moment estimation. It uses estimates of the first and second raw moment of the policy gradient.

To compute $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}_k})$ we use the result of the policy gradient theorem shown in (3.11), repeated here:

$$\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}_k}) = \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t \cdot \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}_k}(A_t \mid S_t) \cdot q^{\pi_{\boldsymbol{\theta}_k}}(S_t, A_t) \right].$$

As this is an expectation over the states and actions within a trajectory, we can estimate its value with a sample of such a trajectory. We run an episode of Hanabi, and collect the experience: $\langle (s_0, a_0, r_1), (s_1, a_1, r_2), \ldots, (s_{T-1}, a_{T-1}, r_T) \rangle$ where s_T is the final absorbing state, after which no action is necessary and all rewards are 0.

With this data, we compute g_t as defined in (3.1) for every time step t as an unbiased estimate for $q^{\pi_{\boldsymbol{\theta}_k}}(s_t, a_t)$. Together with the gradient of the log probability of the chosen action, the estimate of the policy gradient becomes:

$$\widehat{\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}_k})} = \sum_{t=0}^{T-1} \gamma^t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}_k}(a_t|s_t)\, g_t. \tag{4.1}$$

Depending on the batch size, we usually run multiple episodes and average over the outcomes of (4.1) to get a lower variance estimate of the policy gradient. In our implementation, the batch size indicates the minimum number of time steps of experience we want to collect, before doing a parameter update step. After we reach this number, we let the current episode finish to complete the batch of experience. For example, a batch size of 1000 contains about 17 episodes when the agent plays well, as an episode then has an average length of about 60 time steps. In the beginning, when the agent is still bad at playing the game, episodes are much shorter. Note that in the special case of batch size 1, we always have one full episode of experience (not just 1 time step).

Let us define a batch $\mathcal{B}$ as the set $\{1, 2, \ldots, j\}$ of j episodes which have run in this epoch. Each epoch collects one batch of experience after which it makes a parameter update to the policy network. The total number of time steps in the batch is equal to $T_\mathcal{B} = T_1 + T_2 + \ldots + T_j$. When the batch size is set to 1000, we often have $T_\mathcal{B} \approx 1020$ as we let the last episode finish.

The estimate of the gradient that we use in our implementation now becomes:

$$\widetilde{\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta_k}})} = \frac{1}{T_\mathcal{B}} \sum_{i \in \mathcal{B}} \sum_{t=0}^{T_i - 1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta_k}}(a_t|s_t)\, g_t.$$

Notice that we also excluded the γ^t term, even though we stated at the end of Section 3.2.3 that this is incorrect. We discovered this relatively late in the project. The algorithm was still effective, but we are curious what the effect of adding γ^t would have been.

Furthermore, we are taking the mean over all time steps, not just over the number of episodes, as is done in [3, line 52] as well. This gives a scaled estimate for the gradient vector, but the direction of the vector remains the same.

This gives the SPG algorithm shown in Algorithm 1.

Algorithm 1: Simple Policy Gradient

input: a differentiable policy parameterization $\pi_{\boldsymbol{\theta_0}}$, learning rate α, discount factor γ
output: a (hopefully) improved policy $\pi_{\boldsymbol{\theta_k}}$

for *epoch* $k = 1, 2, \ldots$ **do**
  Collect a batch of experience $\mathcal{B}_k$ by running the current policy $\pi_{\boldsymbol{\theta}_k}$ in the environment.
  Compute the discounted return g_t for every collected time step t.
  Estimate the policy gradient as

$$\widetilde{\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta_k}})} = \frac{1}{T_{\mathcal{B}_k}} \sum_{i \in \mathcal{B}_k} \sum_{t=0}^{T_i - 1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta_k}}(a_t|s_t)\, g_t$$

  Perform the policy update with gradient ascent algorithm Adam, learning rate α.
end

A small optional addition to SPG is to renormalize the discounted returns g_t before computing the estimate of the policy gradient. This means we subtract the mean ($\overline{g}$) and divide by the standard deviation (σ) of all the discounted returns in the current batch. For each time step t the new value $\widetilde{g}_t$ is given by

$$\widetilde{g}_t = \frac{g_t - \overline{g}}{\sigma}.$$

In the implementation we add a tiny term (10^{-8}) to the denominator to avoid dividing by zero.

4.2 Vanilla Policy Gradient

The Vanilla Policy Gradient (VPG) algorithm is an actor-critic extension of SPG, since it includes a second neural network for the critic. The critic is represented here by the approximate state value function $V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}$, which is trained alongside the policy network $\pi_{\boldsymbol{\theta}}$. The idea behind adding a critic, as explained in Section 3.2.4, is to get a better evaluation of the actions taken, and use it in the expression for the policy gradient.

In order to get a good estimate of the true value function $v^{\pi_{\boldsymbol{\theta}}}$, we want to minimize the distance $|V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}(s) - v^{\pi_{\boldsymbol{\theta}}}(s)|$ over all states $s \in \mathcal{S}$. We define a loss function with the mean squared error as the objective to minimize for $V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}$:

$$L\big(V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}\big) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \big(V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}(s) - v^{\pi_{\boldsymbol{\theta}}}(s)\big)^2. \tag{4.2}$$

The gradient of this loss function is:

$$\nabla_{\boldsymbol{w}} L\big(V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}\big) = \frac{2}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \big(V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}(s) - v^{\pi_{\boldsymbol{\theta}}}(s)\big) \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}(s).$$

Since the number of states $|\mathcal{S}|$ is enormous[1] and we do not know the true value function $v^{\pi_{\boldsymbol{\theta}}}$, we estimate this gradient with the data we collect by running a few episodes:

$$\widehat{\nabla_{\boldsymbol{w}} L\big(V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}\big)} = \frac{1}{T_{\mathcal{B}}} \sum_{i \in \mathcal{B}} \sum_{t=0}^{T_i - 1} \big(V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}(s_t) - g_t\big) \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}(s_t). \tag{4.3}$$

We use the discounted return g_t as our target (estimate for $v^{\pi_{\boldsymbol{\theta}}}(s_t)$) when updating the parameters $\boldsymbol{w}$ of our value function network.

Similar to [4], we update the value network multiple times per epoch, as opposed to the policy network which we only update once per epoch.[2] Our idea behind this is that the value function needs to correspond to the current policy, which is changing every epoch. Therefore the value network might need a few extra update iterations to keep track of the adjustments. The number of value network updates per epoch is a hyperparameter which we usually set to 5.

Now that we have a good way to train our value function, we can use $V_{\boldsymbol{w}}^{\pi_{\boldsymbol{\theta}}}$ in our expression of the policy gradient as shown in Section 3.2.4 to improve the training of our policy. We compute an estimate of the advantage function for every collected time step t in our batch as follows:

$$\hat{A}_t = g_t - V_{\boldsymbol{w}_k}(s_t) \tag{4.4}$$

where the discounted return g_t is our estimate of $q^{\pi}(s_t, a_t)$, as it was in SPG as well.

In the pseudocode of VPG shown in Algorithm 2 below we will drop π from the notation of $V_{\boldsymbol{w}_k}^{\pi_{\boldsymbol{\theta}_k}}$ for readability, but one should keep in mind that our value function network always tries to approximate the value function corresponding to the current policy $\pi_{\boldsymbol{\theta}_k}$.

[1] At least 10^{56}, see Section 3.2.2.

[2] We have also tried updating the policy network multiple times per epoch, but this did not significantly improve the algorithm.

Algorithm 2: Vanilla Policy Gradient, adapted from [4]

input: a differentiable policy parameterization $\pi_{\boldsymbol{\theta}_0}$, a differentiable value function parameterization $V_{\boldsymbol{w}_0}$, learning rates α_π, α_V, discount factor γ, number of value function updates per epoch i_V

output: a hopefully improved policy $\pi_{\boldsymbol{\theta}_k}$

1 **for** *epoch* $k = 1, 2, \ldots$ **do**

2 Collect a batch of experience $\mathcal{B}_k$ by running the current policy $\pi_{\boldsymbol{\theta}_k}$ in the environment.

3 Compute the discounted return g_t for every collected time step t.

4 Compute the advantage estimates $\hat{A}_t$ based on the current value function $V_{\boldsymbol{w}_k}$ for every collected time step t:

$$\hat{A}_t = g_t - V_{\boldsymbol{w}_k}(s_t)$$

5 Estimate the policy gradient as

$$\widetilde{\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}_k})} = \frac{1}{T_{\mathcal{B}_k}} \sum_{i \in \mathcal{B}_k} \sum_{t=0}^{T_i - 1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}_k}(a_t|s_t)\, \hat{A}_t$$

6 Perform the policy update with gradient ascent algorithm Adam, learning rate α_π.

7 **for** *iteration* $i = 1, 2, \ldots, i_V$ **do**

8 Estimate the gradient of the value function loss as

$$\widehat{\nabla_{\boldsymbol{w}} L(V_{\boldsymbol{w}_{k_i}})} = \frac{1}{T_{\mathcal{B}_k}} \sum_{i \in \mathcal{B}_k} \sum_{t=0}^{T_i - 1} \left(V_{\boldsymbol{w}_{k_i}}(s_t) - g_t \right) \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}_{k_i}}(s_t)$$

9 Perform the value function update with gradient descent algorithm Adam, learning rate α_V.

10 **end**

11 **end**

An optional adjustment to the basic version of VPG presented in Algorithm 2 is to use Generalized Advantage Estimation (GAE). The GAE technique of Schulman et al. [35] estimates the advantage of each action in more detail than the basic method in (4.4). They devised the following:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{4.5}$$

where λ is a new hyperparameter in $[0, 1]$ and δ_t^V is defined as

$$\delta_t^V = r_t + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t)$$

or in other words, the TD residual of $V_{\boldsymbol{w}}$ with discount γ [41]. To summarize briefly, GAE does not only take the difference between all rewards coming after an action (g_t) and the value function, but instead uses all the TD residuals of the episode coming after action a_t. The weight factor λ is used to establish how the importance of all these TD residuals is distributed. A value close to 1, such as $\lambda = 0.95$, is recommended by [35].

The adjustments necessary to the pseudocode of Algorithm 2 are:

- In line 3 the δ_t^V needs to be computed for every collected time step t, instead of g_t.
- $\hat{A}_t$ needs to be replaced by $\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)}$ in line 4 and 5, with the correct formula (4.5) in line 4.

- In line 8 the target for the value network, which is now g_t, gets replaced by: $\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} + V_{\boldsymbol{w}_k}(s_t)$.

Notice that the new target, just like the old one, stays fixed throughout the i_V number of value function updates.

4.3 Proximal Policy Optimization

The actor-critic algorithm named Proximal Policy Optimization (PPO) [36] comes with a different approach. In our previous algorithms, the small learning rate α_π makes sure that the parameters $\boldsymbol{\theta}$ of our policy network cannot change much in one update. This means the new policy is close to the old one in parameter space, but not necessarily in the space of action selection probabilities. A little tweak to $\boldsymbol{\theta}$ could change the action selection probabilities a lot.

PPO avoids this by ensuring that the new policy does not stray too far away from the old one in terms of action selection probabilities. With a new hyperparameter ε, the incentive to increase (or decrease) a certain action selection probability above $1+\varepsilon$ (or below $1-\varepsilon$) times the old one is taken away. This is achieved by the *clip* function, defined for $a < b$ as:

$$\mathrm{clip}(x, a, b) = \begin{cases} b & \text{if } x > b \\ x & \text{if } a \leq x \leq b \\ a & \text{if } x < a \end{cases}.$$

This function is used to define a totally new objective for the policy. The classical reinforcement learning objective of maximizing $J(\pi)$ is replaced by:

$$\underset{\boldsymbol{\theta}}{\text{maximize}}\ L^{\mathrm{CLIP}}(\boldsymbol{\theta})$$

with

$$L^{\mathrm{CLIP}}(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[\min \left(r(\boldsymbol{\theta}) \hat{A}^{\pi_{\boldsymbol{\theta}_k}},\ \mathrm{clip}\left(r(\boldsymbol{\theta}), 1-\varepsilon, 1+\varepsilon\right) \hat{A}^{\pi_{\boldsymbol{\theta}_k}} \right) \right] \tag{4.6}$$

where the expectation is over state-action pairs (S, A) to be encountered, taking into account the environment's transition probabilities and the policy's action selection distribution. Furthermore, $\hat{A}^{\pi_{\boldsymbol{\theta}_k}} = \hat{A}^{\pi_{\boldsymbol{\theta}_k}}(S, A)$ is any advantage function estimate, and $r(\boldsymbol{\theta})$ is the ratio between the new policy and the old one:

$$r(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(A \mid S)}{\pi_{\boldsymbol{\theta}_k}(A \mid S)}. \tag{4.7}$$

In contrast to the previous two algorithms, instead of updating the policy parameters only once per epoch, PPO performs multiple update iterations to the policy network per epoch. The old policy, represented by $\pi_{\boldsymbol{\theta}_k}$ in the denominator of (4.7), is the policy used to collect the batch of experience. It remains *fixed* throughout the update iterations of the current epoch. Only in the first iteration, the new and old policy are the same.

Intuitively, in the ratio $r(\boldsymbol{\theta})$ we want that $r > 1$ for "good" actions, meaning where $A > 0$, such that in our policy the probability of selecting these good actions increases. Similarly, for bad actions (where $A < 0$) we would like to have $r < 1$. Suppose we have sampled one state-action pair (s, a) from the environment, such that the objective becomes

$$L_1^{\mathrm{CLIP}}(\boldsymbol{\theta}, s, a) = \min \left(r(\boldsymbol{\theta}) \hat{A}^{\pi_{\boldsymbol{\theta}_k}},\ \mathrm{clip}\left(r(\boldsymbol{\theta}), 1-\varepsilon, 1+\varepsilon\right) \hat{A}^{\pi_{\boldsymbol{\theta}_k}} \right).$$

This single sample objective is shown in Figure 4.1, split into the cases of positive and negative advantage.[3] When $A > 0$ the algorithm is stimulated to increase L^{CLIP} up to the point where

[3] To be complete: when $A = 0$ we do not want to change the policy, which is indeed what this objective function achieves in that case ($\forall\ \boldsymbol{\theta}:\ L^{\mathrm{CLIP}}(\boldsymbol{\theta}) = 0$).

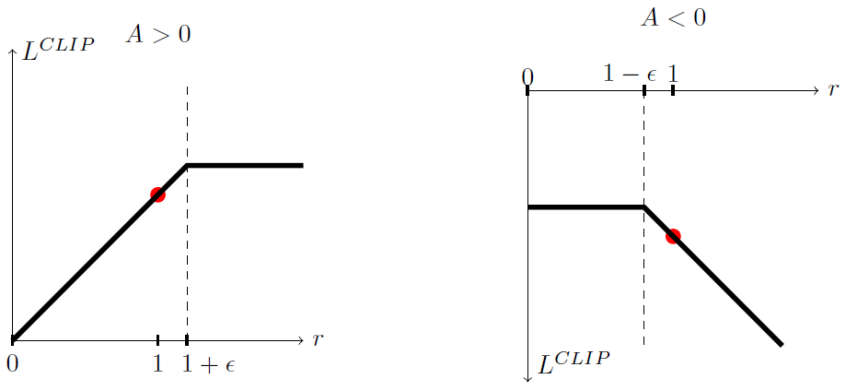


Figure 4.1: Single sample L^{CLIP} objective, taken from [36]. Keep in mind that we are trying to maximize L^{CLIP}. The red dot indicates where the update iterations start (at $r = 1$, since the new and old policy are then the same). The incentive for r to grow or shrink more than ε is taken away by the *clip* function.

$r = 1 + \varepsilon$. After that the objective function is flat, giving a gradient of zero. For $A < 0$ we still want to increase our objective L^{CLIP}, but now up to the point where $r = 1 - \varepsilon$. Notice that in both cases, if the policy updates go into the wrong direction of r for this particular state-action pair, the incentive to go back remains no matter how far off the new policy is. This is achieved by putting the *min* function in the objective.

In order to compute the gradient, let us consider possible values of the advantage estimate $\hat{A}^{\pi_{\boldsymbol{\theta}_k}}$, which we will continue to denote by A. Note that this estimate does not depend on $\boldsymbol{\theta}$ but on the old parameter values $\boldsymbol{\theta}_k$, and thus remains fixed throughout the update iterations. We separate two cases:

- If $A \geq 0$, we have

$$\begin{aligned} L_1^{\text{CLIP}}(\boldsymbol{\theta}, s, a) &= \min\big(r(\boldsymbol{\theta})A,\ \text{clip}\left(r(\boldsymbol{\theta}), 1-\varepsilon, 1+\varepsilon\right)A\big) \\ &= \min\big(r(\boldsymbol{\theta}),\ \text{clip}\left(r(\boldsymbol{\theta}), 1-\varepsilon, 1+\varepsilon\right)\big)A \\ &= \min\big(r(\boldsymbol{\theta}), 1+\varepsilon\big)A \\ &= \begin{cases} \frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)}A & \text{if } r(\boldsymbol{\theta}) < 1+\varepsilon \\ (1+\varepsilon)A & \text{if } r(\boldsymbol{\theta}) \geq 1+\varepsilon \end{cases}. \end{aligned}$$

  This gives

$$\nabla_{\boldsymbol{\theta}} L_1^{\text{CLIP}}(\boldsymbol{\theta}, s, a) = \begin{cases} \frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)}A & \text{if } r(\boldsymbol{\theta}) < 1+\varepsilon \\ 0 & \text{if } r(\boldsymbol{\theta}) \geq 1+\varepsilon \end{cases}.$$

  Actually, the gradient at the point in $\boldsymbol{\theta}$-space where $r(\boldsymbol{\theta}) = 1 + \varepsilon$ is undefined. For easier implementation we define it to be 0.

- If $A < 0$, we have

$$\begin{aligned} L_1^{\text{CLIP}}(\boldsymbol{\theta}, s, a) &= \min\big(r(\boldsymbol{\theta})A,\ \text{clip}\left(r(\boldsymbol{\theta}), 1-\varepsilon, 1+\varepsilon\right)A\big) \\ &= \max\big(r(\boldsymbol{\theta}),\ \text{clip}\left(r(\boldsymbol{\theta}), 1-\varepsilon, 1+\varepsilon\right)\big)A \\ &= \max\big(r(\boldsymbol{\theta}), 1-\varepsilon\big)A \end{aligned}$$

$$= \begin{cases} \frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)} A & \text{if } r(\boldsymbol{\theta}) > 1-\varepsilon \\ (1-\varepsilon)A & \text{if } r(\boldsymbol{\theta}) \leq 1-\varepsilon \end{cases}.$$

This gives

$$\nabla_{\boldsymbol{\theta}} L_1^{\text{CLIP}}(\boldsymbol{\theta}, s, a) = \begin{cases} \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)} A & \text{if } r(\boldsymbol{\theta}) > 1-\varepsilon \\ 0 & \text{if } r(\boldsymbol{\theta}) \leq 1-\varepsilon \end{cases}.$$

Putting it all together, we have this expression for the gradient of the PPO objective, for one sampled state-action pair:

$$\nabla_{\boldsymbol{\theta}} L_1^{\text{CLIP}}(\boldsymbol{\theta}, s, a) = \begin{cases} \dfrac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)} A & \text{if } \begin{array}{l} (A > 0 \text{ and } r(\boldsymbol{\theta}) < 1+\varepsilon) \text{ or} \\ (A < 0 \text{ and } r(\boldsymbol{\theta}) > 1-\varepsilon) \end{array} \\ 0 & \text{otherwise} \end{cases}. \tag{4.8}$$

Note that it is essential for PPO to perform multiple update iterations on one batch of experience, otherwise the *clip* function will never be used. In the first update the number of clipped state-action pairs will be zero, since the new policy is then still equal to the old policy. This does not mean that the gradient will be zero at the current parameters $\boldsymbol{\theta}_k$, so the policy does change for the second iteration. That is the first moment where clipping can happen.

During each policy update we average over all the single sample gradients in the batch to get our estimate for $\nabla_{\boldsymbol{\theta}} L^{\text{CLIP}}(\boldsymbol{\theta})$:

$$\nabla_{\boldsymbol{\theta}} \widehat{L^{\text{CLIP}}}(\boldsymbol{\theta}) = \frac{1}{T_{\mathcal{B}_k}} \sum_{i \in \mathcal{B}_k} \sum_{t=0}^{T_i - 1} \nabla_{\boldsymbol{\theta}} L_1^{\text{CLIP}}(\boldsymbol{\theta}, s_t, a_t).$$

PPO also has a value function network, which we use to compute advantage estimates with the same two methods as in VPG (choosing between the basic version or GAE). The symbol $\mathcal{T}_t^V$ shown in Algorithm 3 indicates the target of the value function network, which depends on the chosen method. In the experiments of Chapter 6 it will always be indicated which one we applied.

Algorithm 3: Proximal Policy Optimization, adapted from [5]

input: a differentiable policy parameterization $\pi_{\boldsymbol{\theta}_0}$, a differentiable value function parameterization $V_{\boldsymbol{w}_0}$, learning rates α_π, α_V, discount factor γ, number of policy and value function updates per epoch i_π, i_V, clipping parameter ε

output: a hopefully improved policy $\pi_{\boldsymbol{\theta}_k}$

for *epoch* $k = 1, 2, \ldots$ **do**
  Collect a batch of experience $\mathcal{B}_k$ by running the current policy $\pi_{\boldsymbol{\theta}_k}$ in the environment.
  Compute the advantage estimates $\hat{A}_t$ based on the current value function $V_{\boldsymbol{w}_k}$.
  for *iteration* $i = 1, 2, \ldots, i_\pi$ **do**
    Estimate the L^{CLIP} gradient as

$$\widehat{\nabla_{\boldsymbol{\theta}} L^{\text{CLIP}}}(\boldsymbol{\theta}_{ki}) = \frac{1}{T_{\mathcal{B}_k}} \sum_{i \in \mathcal{B}_k} \sum_{t=0}^{T_i - 1} \nabla_{\boldsymbol{\theta}} L_1^{\text{CLIP}}(\boldsymbol{\theta}_{ki}, s_t, a_t)$$

    Perform the policy update with gradient ascent algorithm Adam, learning rate α_π.
  end
  for *iteration* $i = 1, 2, \ldots, i_V$ **do**
    Estimate the gradient of the value function loss as

$$\widehat{\nabla_{\boldsymbol{w}} L(V_{\boldsymbol{w}_{k\,i}})} = \frac{1}{T_{\mathcal{B}_k}} \sum_{i \in \mathcal{B}_k} \sum_{t=0}^{T_i - 1} \left(V_{\boldsymbol{w}_{k\,i}}(s_t) - \mathcal{T}_t^V \right) \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}_{k\,i}}(s_t)$$

    Perform the value function update with gradient descent algorithm Adam, learning rate α_V.
  end
end

Chapter 5

Implementation

We have seen the three main algorithms in Chapter 4, but there is still a lot of designing and tweaking to do before the algorithm learns to play well. Section 5.1 will talk about the early design process and the choices we have made. For the larger design options and selection of important hyperparameter settings we devised experiments, which will be presented in Chapter 6. The metrics that we record during the experiments are explained in Section 5.2.

5.1 Design process

One of the first steps one has to take when implementing a DRL algorithm is defining the input and output of the neural network(s). In the first algorithm, SPG, we use one neural network representing the policy, which has a certain state representation as input and selection probabilities of actions as output. There are many ways in which the state (or observation, in the case of partial observability[1]) can be represented as a vector of numbers. Similarly, the action set can be represented in different ways as well.

We should note that in our implementation, the algorithm (or agent) controls both players in Hanabi. Recall that we stick with the two-player setting for this report, as stated in Section 1.2. We may sometimes use *the agent* as a synonym to *the current player*, but strictly speaking they are not the same.

Initial input

When we tried to implement our first DRL algorithm, we included just two components of the state information in the input vector: the firework stacks and the current player's cards. The rest of the state information was left out, to keep the input vector simple. We will often call the input vector an *observation vector* or *state representation*.

Let us explain how we converted the two state components into numbers. The firework stacks were represented by an integer in $[0, 5]$ for each color. For each of the current player's cards we include a pair of integers in the order: (color, rank). We mapped each color to an integer, by the mapping $\{(R \mapsto 0), (Y \mapsto 1), (G \mapsto 2), (W \mapsto 3), (B \mapsto 4)\}$. We represented the ranks of the cards by the values $0, 1, 2, 3, 4$ instead of their actual values $(1, 2, 3, 4, 5)$. In the example of Figure 5.1 the observation vector would look as follows:

$$[\ \underbrace{0, 0, 2, 1, 3}_{\text{fireworks}},\ \underbrace{0, 4,\ \ 4, 3,\ \ 0, 1,\ \ 3, 1,\ \ 2, 0}_{\text{cards in hand}}\ ].$$

[1] And, as presented in Appendix B.1, it is probably a good idea to include more information than just the current observation, such as some memory of previous observations, beliefs over your cards and (nested) beliefs of other players.

Figure 5.1: Example of a state. Player 1's blue and white card are directly playable. In simplified Hanabi players are allowed to view their own cards.

Notice that each firework stack just gets its current value in the observation vector, and that for instance the R5 card is represented by the pair $(0, 4)$.

Initial output

As output vector we provided the policy network with 10 nodes, representing the probabilities of selecting a certain action. For each card in the player's hand, it can choose to discard or play that card, giving 10 actions in total. At first we thought including the hinting actions would be unnecessary, since all information is available in the simplified version of Hanabi. This turned out to be a mistake.

Showing playability

The initial setup described above did not work for SPG, so we tried an even simpler approach. We acted as an oracle that tells the agent which cards are directly playable, and which are not. In the example of Figure 5.1, we would produce the observation vector: $[0, 1, 0, 1, 0]$ since the B4 and W2 cards can be played immediately. A few experiments that we have run with this input vector, after adding some critical adjustments described below, are shown in Appendix A.1. As soon as this setup worked, we stepped away from showing playability in the observation vector, since we want our agent to recognize the playability of a card by itself.

Improved output

The algorithm was trying to perform illegal actions quite often. We realized that it was not allowed to discard anything, because the hint tokens budget was always full at 8/8. The agent must give a hint before it can discard.[2] Thus, we included an 11^{th} action node in the output vector, which would result in giving a random legal hint. Our agent ignores the information of the hint, but it was now able to use this action to lower the hint token budget and pass on the turn to the next player.

Improved input

With the current input vector[3] our agent is not able to observe whether discarding is legal or not. For this we needed to include the hint token budget in the observation vector. We added

[2] See Section 1.1 for the rules of Hanabi.
[3] This holds for both the initial input and the input showing playability.

the life tokens as well, such that the agent could directly observe some effect of playing a bad card. Both quantities are converted to real numbers in $[0, 1]$ to make sure they cover the same range. Without normalization these ranges would differ ($[0, 8]$ and $[0, 3]$) which is not beneficial for gradient descent [20].

Illegal actions

The adjustment that finally made our SPG algorithm work was giving a negative reward to illegal actions, such that it would learn not to choose them. If our agent chooses an illegal action, we save the action in the experience batch along with the negative reward and the current observation. However, we do not pass this action to the environment as it would give an error. This means we leave the state of the environment unchanged: the agent will observe the exact same state in the next time step. The hope is that due to the stochasticity of the policy, the agent will now select a different (legal) action. Of course, it might happen that the agent chooses an illegal action many times consecutively. If a certain maximum number of illegal actions have been selected by our agent, we go straight to the policy update step without finishing the episode.[4] In this way, the algorithm usually quickly learns not to make illegal moves.

5.2 Recorded metrics

In this section we will give an overview of the metrics which we collected during our experiments. Many of these metrics will occur in the graphs in Chapter 6. In most of these graphs the values are presented by taking an average every 100 epochs, to make the graphs more readable.

The **return** is the undiscounted sum of rewards per episode: $\sum_{t=1}^{T} r_t$, where T is the final step of the episode. This is g_0 from (3.1), with $\gamma = 1$. Notice that we use a discount factor of $\gamma < 1$ in our objective during the experiments. This metric is a separate value, detached from the g_t used in (4.1) for example, just so we can keep track of the undiscounted sum of rewards.

The **score** is the actual outcome of the Hanabi game per episode. It is a value in $\{0, 1, \ldots, 25\}$. The **fireworks** metric is the sum of the firework stacks at the end of the game. This means it is often equal to the score, except when all life tokens have been lost. Then the score will be 0, but the fireworks can be anything in $\{0, 1, \ldots, 24\}$[5]. When the fireworks go up during training, it is a useful metric to see that the algorithm is actually learning something even though the score stays at 0.

The **lives** metric shows how many life tokens were left at the end of an episode. At the start of training this is often 0, until the algorithm discovers how to retain some life tokens.

With **illegal actions** we keep track of how many times the agent tried to perform an illegal move during the epoch. Even for one episode there could be more illegal actions than the maximum length of a Hanabi game[6] since we keep the state of the game in place when the agent attempts an illegal action (as explained in Section 5.1).

In the **entropy** metric we follow the entropy of our policy's distribution. For a particular state s it is computed as $\sum_{a \in \mathcal{A}} -\pi(a|s) \log \pi(a|s)$. In the graphs the average entropy over all states in a batch will be shown. These values are averaged again over every 100 epochs.

To plot the **action probabilities** we average over all policy distributions in a batch B, as in $\frac{1}{|B|} \sum_{s \in B} \pi(s)$, where $\pi(s)$ represents the vector of action selection probabilities for state s. We

[4] We have set this maximum number of illegal actions equal to $\max(100, \text{batch size})$. This makes sure that there are at least 100 illegal actions in a batch when an episode is cut off because many illegal actions occurred, even when the batch size is small.

[5] Not 25 in this case, as the game is finished when the 25^{th} point is scored, leaving no opportunity to lose the third life token.

[6] Which is 89, as shown in Section 7.1.

only extract this once every 100 epochs (instead of taking an average again) to save computing time.

In order to inspect how the **value network** is evolving, we keep track of the average output of the network, along with the average target we train it towards. Just like the action probabilities, we compute these metrics once every 100 epochs. The average value output of that particular epoch is $\frac{1}{|B|}\sum_{s\in B} V(s)$. The average target is also computed over all states in a batch, and depends on the type of target chosen, as explained in Section 4.2.

For PPO we have an extra metric called the **clip fraction**. This records how many of the state-action pairs in a batch resulted in a clipped ratio, for the policy objective shown in (4.6). The measured clip fraction is the total number of clipped state-action pairs, divided by the current batch size. Per batch PPO makes multiple policy network update iterations, so we should add that we measure the clip fraction at the last iteration. This often (but not always) has the highest clip fraction of all iterations, since after more updates the new policy is likely differing even more from the old policy (the one used to collect the experience of this batch).

A new metric we defined ourselves is called **positional bias**. We use this to track how large the difference is in the policy's preference for a particular card position relative to the others. The algorithms often showed a bias towards certain card position. For example, it would play a card from index 1 much more often than from index 3. To quantify this bias, we define

$$b_g = \frac{\max_{i,j\in\mathcal{A}_g}\left(\left|p_i - p_j\right|\right)}{\sum_{k\in\mathcal{A}_g} p_k}$$

where g can refer to any subset of actions and p_i is the average probability of selecting action i given the visited states of the current batch: $p_i = \frac{1}{|B|}\sum_{s\in B}\pi(i|s)$. We track the positional bias of two subsets: the five play actions and the five discard actions.

In words, the positional bias is the greatest distance between two action selection probabilities within the same subset of actions. Also, we rescale this distance to a probability distribution on this specific subset of actions only, to be able to fairly compare the play bias with the discard bias, even if for example the agent discards much more than it plays.

Chapter 6

Experiments

In this chapter we will describe the experiments that we have run in order to answer the research questions: *Which algorithmic design options and hyperparameter settings are beneficial to the learning pace?* and *Which algorithm reaches the highest average score?* The different settings that are being compared will be explained for each experiment. We directly show the results within each section to conclude which options worked best.

Sections 6.1 and 6.2 contain experiments with only the SPG algorithm, while VPG and PPO are added in Sections 6.3 and 6.4. In Section 6.5 we describe a final comparison between all three algorithms.

6.1 Observation vectors

We devised a variety of possible observation vectors, of which we will compare the four most important ones. To distinguish these different state representations, we refer to the length of the observation vector. An overview is shown in Table 6.1.

Table 6.1: Overview of the various observation vectors we designed.

	components of state information						
length	fireworks	own cards	other's cards	lives	hints	discards	type
37	✓	✓		✓	✓		mix
62	✓	✓		✓	✓	✓	mix
136	✓	✓		✓	✓	✓	binary
186	✓	✓	✓	✓	✓	✓	binary

- Vector 37: the basics.

  The information that the agent needs to determine which of her cards are playable, can be fully observed from the state in simplified Hanabi. The current firework stacks and the current player's cards are sufficient. Thus, this observation vector contains these two pieces of information. It also still holds the number of life and hint tokens left over.

  The fireworks are represented by normalizing the current value of the stack for each color. In the example of Figure 5.1 this becomes: $[0, 0, 0.4, 0.2, 0.6]$. For each of the player's own cards we include a one-hot encoding for the color, and a normalized value for the rank of the card. For example, a Y4 card will be represented as $[0, 1, 0, 0, 0,\ 0.6]$.

  Notice that we subtract one from the rank of a card before normalizing it. The rank representing input value is thus computed as $\frac{\text{rank}-1}{\text{max rank}}$, where the maximum rank is 5. This means that this single input node of the neural network can only take on the values $\{0, 0.2, 0.4, 0.6, 0.8\}$, and not 1. Recognizing when a card is playable is easier now: a

firework stack and a corresponding playable card should have the same value representing their rank.

To complete the observation vector we concatenate all pieces together, while ensuring that the order of the player's cards is correctly maintained. This gives a length of $5+5\cdot6+1+1 = 37$. Recall that a player has 5 cards in her hand and that we include a normalized value for the life and hint tokens.

- Vector 62: adding the discard pile.

  This observation vector contains the same information as vector 37, with the addition of the discarded cards. During a game of Hanabi, players are allowed to view the contents of the discard pile. This can be helpful in deducing the possible cards in your own hand, when playing the original version of Hanabi. In the simplified version it can still be useful to determine which cards should not be discarded anymore (when there is only one duplicate of a card left in the game).

  We translate the discard pile into numbers in the following way. For each of the 25 unique cards we include a normalized value representing the number of discarded copies of that card. We group them per color, so for example the vector piece $[0.667, 0, 0.5, 0, 0]$ indicates that two rank 1 cards and one rank 3 card of a certain color are in the discard pile.[1]

- Vector 136: convert to binary.

  As shown in Table 6.1 this observation vector contains the same components of the state information as vector 62. However, this vector represents all information with binary numbers instead of real (normalized) values.

  The firework stacks are now represented in thermometer style, with five binary numbers for each color. For example, $[1, 1, 1, 0, 0]$ means that the firework of a certain color is at rank 3. For each of the player's own cards we include a one-hot encoding for the color as well as the rank. The Y4 card is now represented by the piece $[0, 1, 0, 0, 0,\ 0, 0, 0, 1, 0]$. The discard pile is included with 10 binary values per color. For instance, $[1, 1, 0,\ 0, 0,\ 1, 0,\ 0, 0,\ 0]$ means that two rank 1 cards and one rank 3 card of a certain color have been discarded. Lastly, the vector pieces $[1, 1, 0]$ and $[1, 1, 1, 1, 1, 0, 0, 0]$ indicate that there are 2 life tokens and 5 hint tokens left. The total length of this observation becomes $5 \cdot 5 + 5 \cdot 10 + 5 \cdot 10 + 3 + 8 = 136$.

  Since this is a larger input vector, the neural network has more connections to the first hidden layer of which the weights can be tweaked. This could make it easier to train towards desired behavior. More connections unfortunately also means a longer training time.

  A different drawback can be the fact that the policy network might be less able to generalize between states. For example, cards with rank 4 and 5 are closer related in gameplay than cards with rank 1 and 5. A B5 card can be played directly after a B4 card. If our agent has never played a B5 card before, we would still like it to generalize and output an almost similar policy for the B5 card as for a B4 card. (Namely: do not play them at the beginning of a game, but do so towards the end if possible.)

  The information that B4 and B5 are similar cards was indirectly included in the normalized rank values of vectors 37 and 62, because 0.6 and 0.8 are much closer than 0 and 0.8. In vector 136 this is lost, since a different rank just means a different bit is turned on.

  On the other hand, B4 and B5 are similar but definitely not the same card. They cannot both be playable at the same time. It might be advantageous to try just the opposite of vectors 37 and 62, and make the differences between cards clearer. That is the main idea behind vector 136.

[1] There are three duplicates of a rank 1 card, but only two duplicates of a rank 3 card. The other ranks (of this color) have not been discarded yet, in this example.

Table 6.2: Settings of the experiments to compare the observation vectors.

	37v62	62v136	136v186
Algorithm	SPG	SPG	SPG
Representations			
State	**37 / 62**	**62 / 136**	**136 / 186**
Action	11	11	11
Rewards			
Successful play	+1	+1	+3
Lost all lives	−score	−score	−score
Illegal move	−10	−0.5	−1
Lost one life	0	0 / −0.1	−0.1
Discard	0	0	0
Play	0	0 / +0.02	+0.02
Hint	0	0 / −0.02	−0.02
Discard playable	0	0 / −0.1	−0.1
Discard useless	0	0 / +0.1	+0.1
Discard unique	0	0 / −0.1	−0.1
Network architecture			
Hidden layers	[64]	[64, 64]	[128, 128, 64]
Activation function	Tanh	Tanh	Tanh
Hyperparameters			
Batch size	500	1000	1000
Entropy coefficient	0	0.01	0.01
Renormalize	no	yes	yes
Discount factor	0.99	0.99	0.99
Learning rate	10^{-3}	10^{-3}	$3 \cdot 10^{-4}$

- Vector 186: adding the other player's cards.

  This observation vector is constructed in the same way as vector 136, with the addition of the other player's cards. In simplified Hanabi, observing the other player's cards is much less relevant than the knowledge of your own cards, but it could still be useful. For example, if you have many playable cards and your companion has none, you might want to play a card that makes one of her cards playable (e.g. you have a playable R2 and she has a R3).

  We add the other player's cards in the same manner as the current player's cards in vector 136. This means the vector is extend by $5 \cdot 10$ binary values, becoming length 186.

The only component of state information that is not mentioned in Table 6.1 is the current deck size. We excluded this small piece of information to keep the observation vectors simple. However, it could still be a useful addition for future research as Hanabi players sometimes need to be careful to make sure the deck is not emptied too quickly.

The *mix* and *binary* categorisations in the last column of Table 6.1 refer to the type of values which are in the observation vector. Vectors 136 and 186 consist of only 0's and 1's, while the others contain a mix of integer values and real values, but always inside the interval $[0, 1]$.

We created these observation vectors at different times throughout the project. Unfortunately, we have not run a single experiment where all four of the described observation vectors are used. We will compare the different vectors in a pairwise manner. The settings of these runs can be seen in Table 6.2. Note that the comparison of the observation vectors with length 62 and 136 is not completely fair, as the reward settings of their respective runs differ.

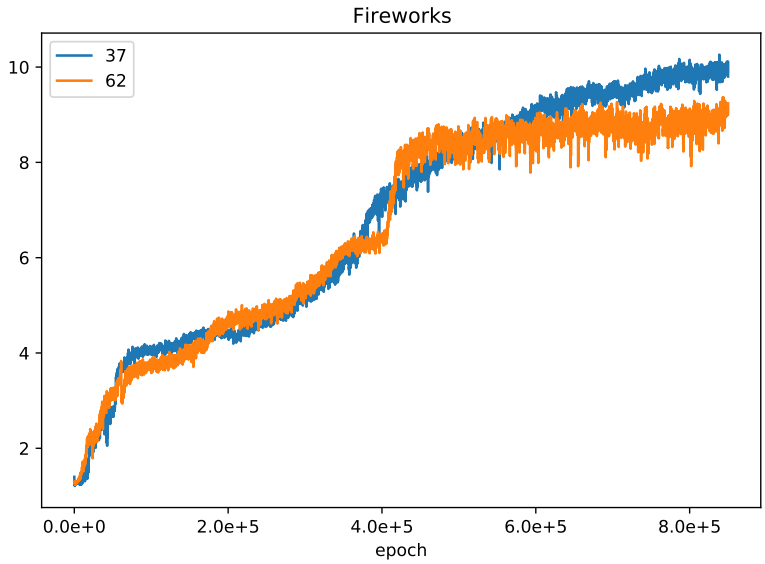


Figure 6.1: Observation vectors with length 37 and 62. Performance is similar throughout training, with vector 37 having a slight advantage near the end. See column 37v62 in Table 6.2 for the settings.

Results of *Observation vectors*

In general, all graphs presented in Figures 6.1, 6.2, 6.3 show a version of the SPG algorithm that is beginning to learn how to play Hanabi. The differences in performance and learning speed are not very large.

For the comparison of observation vectors 37 and 62 in Figure 6.1 the fireworks metric is used, as we did not keep track of the scores yet at that point. These two metrics are often very close, as explained in Section 5.2. Vector 37 seems to perform slightly better after 800,000 epochs, but we continued our research with vector 62. This was because it includes information on the discard pile, which we think is quite essential (especially near the end of a game).

Between vectors 62 and 136, shown in Figure 6.2, there is a larger difference in performance, with 62 having the upper hand. However, we continued with the observation vector of length 136 for many future experiments. We assessed it as more promising for VPG and PPO, since it presents the differences between distinct states clearer. Unfortunately, we have not tested vector 62 against 136 for VPG and PPO, which would have been an interesting experiment.

Observation vector 186 shows promising growth as well, as can be seen in Figure 6.3. When we invented this vector near the end of the project we also tried it on VPG and PPO. These experiments showed similar behaviour to Figure 6.3, with vectors 136 and 186 reaching about equal scores. We decided to stick with vector 136 for faster training times.

Overall, it is difficult to say which observation vector performs best. It seems like the differences are not very significant, and depend a lot on the algorithm and other settings used.

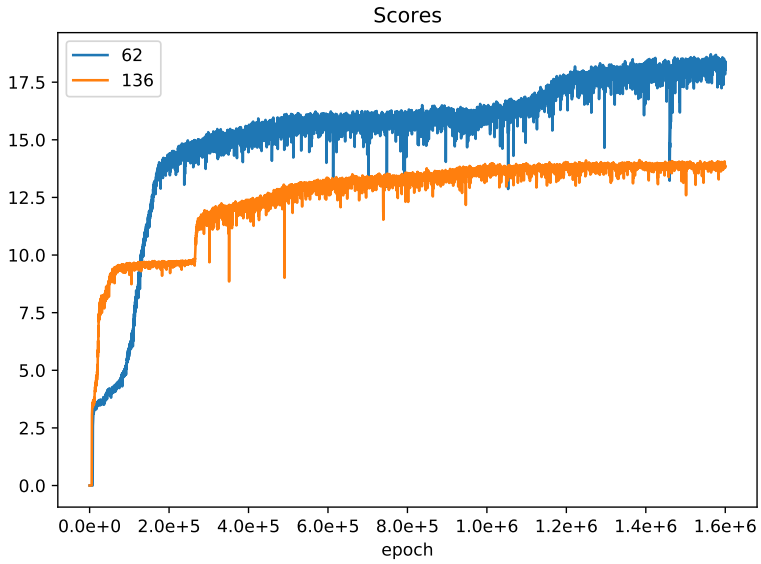


Figure 6.2: Observation vectors with length 62 and 136. Vector 136 learns more quickly, but seems to hit a plateau in this particular setting. See column 62v136 in Table 6.2 for the settings.

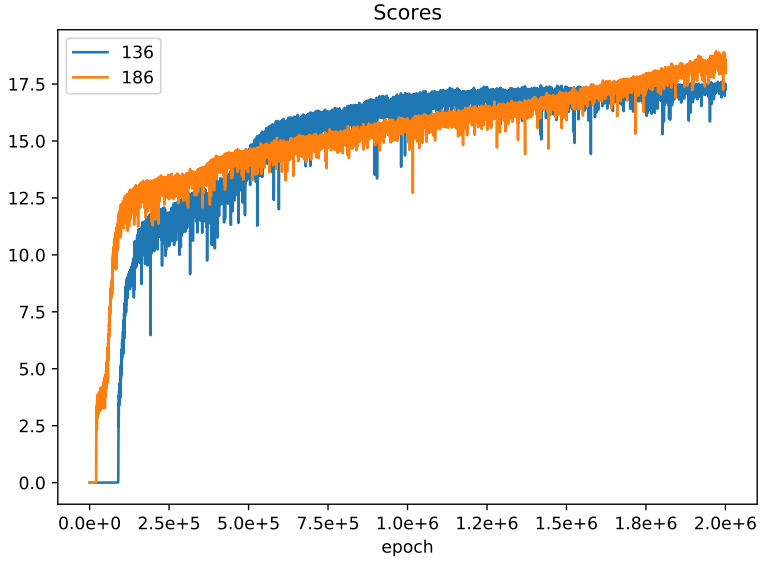


Figure 6.3: Observation vectors with length 136 and 186. Both vectors work well, with vector 186 showing promising growth. See column 136v186 in Table 6.2 for the settings.

6.2 Stimulate exploration

Exploration is one of the key aspects of deep reinforcement learning algorithms that we struggled with during our project. The stochastic policy of our algorithm often turned almost deterministic rather quickly. A deterministic-like policy is acceptable if the agent is converging towards desired performance. However, our agents were converging into low-entropy policies too quickly, while the actions chosen were definitely not optimal yet.

Out of the five card positions (indices $0, 1, 2, 3, 4$) our agent can choose to play or discard from, it often converged to playing from only one position (for example, always choosing index 0). We wanted to stimulate exploration, such that the agent would try out different card positions for a much longer time during training.

Multiple ideas were tried, but we will present the details of the three main concepts. These are: including an exploration term in the policy, adding an entropy term to the objective, and shuffling the cards in the player's hand. Other ideas are briefly described in Appendix B.2.

Exploration term

In this approach we add an exploration term[2], $\epsilon \in [0, 1]$, to the action selection procedure of the policy. The idea is drawn from the value-based ϵ-greedy algorithm. We wanted to try out if this setup would work for a policy-based algorithm as well.

In our regular policy $\pi_{\boldsymbol{\theta}_k}$ the next action is sampled from the output of the policy network given a particular state: $a_{t+1} \sim \pi_{\boldsymbol{\theta}_k}(s_t)$. With the added exploration term ϵ, this policy distribution becomes:

$$\pi^{\epsilon}_{\boldsymbol{\theta}_k}(s_t) = U_{\text{legal}}(s_t) \cdot \epsilon + \pi_{\boldsymbol{\theta}_k}(s_t) \cdot (1 - \epsilon)$$

where $U_{\text{legal}}(s_t)$ is a (discrete) uniform distribution over all legal actions given state s_t. For a specific action a (which is legal given the current state s_t) we have that the probability of selecting a is:

$$\pi^{\epsilon}_{\boldsymbol{\theta}_k}(a|s_t) = \frac{1}{|\mathcal{A}_{\text{legal}}(s_t)|} \cdot \epsilon + \pi_{\boldsymbol{\theta}_k}(a|s_t) \cdot (1 - \epsilon)$$

For an illegal action i the probability is: $\pi^{\epsilon}_{\boldsymbol{\theta}_k}(i|s_t) = \pi_{\boldsymbol{\theta}_k}(i|s_t) \cdot (1 - \epsilon)$ as it will not be selected by the random exploration part of $\pi^{\epsilon}_{\boldsymbol{\theta}_k}$.

The effect of ϵ on $\pi_{\boldsymbol{\theta}_k}(s_t)$ is visualized in the diagrams of Figure 6.4. In this example we assume to have a state where all 11 actions are legal[3]. All action selection probabilities that are under $\frac{1}{11}$ are increased, while any probabilities above that threshold are decreased.

Entropy term

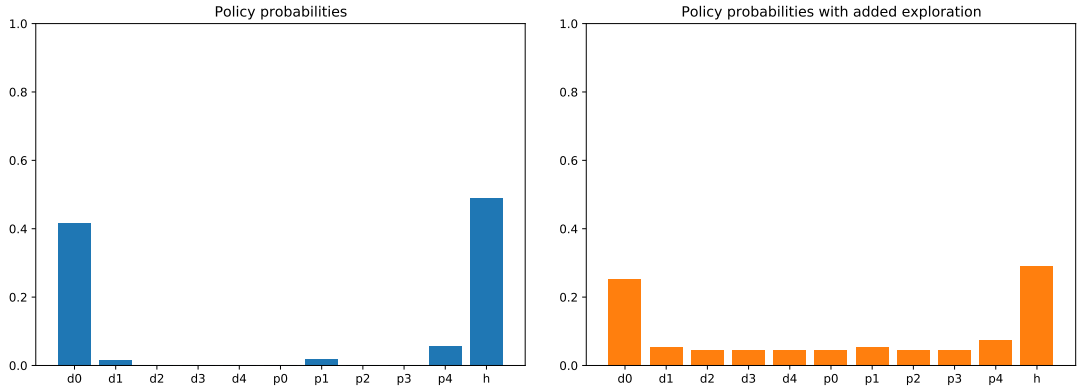
The fact that our policy was becoming almost deterministic during training, means that the entropy of our policy's distribution is very low. In order to stimulate more exploratory behaviour, we included an entropy term in the objective. We define the entropy H of the policy π for a certain state s as

$$H_\pi(s) = \sum_{a \in \mathcal{A}} -\pi(a|s) \log \pi(a|s). \tag{6.1}$$

This means that the new objective becomes

$$J_{\text{new}}(\pi) = J_{\text{old}}(\pi) + \beta \cdot \mathbb{E}_\pi[H_\pi(S)] \tag{6.2}$$

[2]Note that this is a different ϵ than the ε used as the clipping parameter of PPO.
[3]Which is quite often the case: when the hint token budget is not equal to 0 or 8.



(a) $\pi_{\boldsymbol{\theta}_k}(s_t)$ (b) $\pi^{\epsilon}_{\boldsymbol{\theta}_k}(s_t)$ with $\epsilon = 0.5$

Figure 6.4: Shifting probabilities with ϵ. In the new policy, action selection probabilities are more spread out, to stimulate exploration. The labels are: d for discard, p for play, and h for giving a hint. The numbers next to d or p indicate from which index (position in the player's hand) a card is chosen for that action.

where $J_{\text{old}}(\pi)$ can be the objective function of SPG, VPG, or PPO, and the expectation in the second term is over all the states one may encounter using policy π, taking into account the environment's transition probabilities and the policy's action selection probabilities. The *entropy coefficient* β is a new hyperparameter to determine the importance of the entropy term.

Shuffling cards

We noticed that our agents often had a large positional bias (defined in Section 5.2) in choosing the index for a play or discard action. The Hanabi Learning Environment (HLE) is made in such a way that whenever a player is dealt a new card from the deck, the new card is added at the end of the player's hand. The other cards move up one position until the empty card slot is filled. We think this way of dealing cards might introduce a bias in the way our agent chooses its actions. We see it sometimes only playing and discarding from position index 0 or 1, while waiting for playable cards to reach that position.

To test if it has any influence, we tried to reprogram the HLE such that new cards are put into the empty slot directly. It turned out that this was more difficult than initially thought, so we came up with a different approach. Each turn of the game the cards of the active player are now shuffled. This means that the newly dealt card could be in any position in the player's hand, which hopefully decreases positional bias and increases performance.

Experiments

The settings of the experiments we ran to compare these three approaches are shown in Table 6.3. Notice that for each approach we show a couple of runs: one which does not use the exploration stimulus, and a few with different values for the relevant setting.

Table 6.3: Settings of the experiments to compare the exploration stimuli.

	Epsilon	Entropy	Shuffle
Algorithm	SPG	SPG	SPG
Representations			
State	62	62	62
Action	11	11	11
Rewards			
Successful play	+1	+1	+1
Lost all lives	−score	−score	−score
Illegal move	−10	−2	−0.5
All others	0	0	0
Network architecture			
Hidden layers	[64]	[64]	[64, 64]
Activation function	Tanh	Tanh	Tanh
Hyperparameters			
Batch size	500	500	1000
Renormalize	no	no	yes
Discount factor	0.99	0.99	0.99
Learning rate	10^{-3}	10^{-3}	10^{-3}
Exploration			
Epsilon term[4]	**0**	0	0
	0.5 (fast)		
	0.5 (slow)		
	0.05		
Entropy coefficient	0	**0**	0.01
		0.0001	
		0.01	
		0.1	
Shuffle cards	no	no	**no**
			yes

Results of *Stimulate exploration*

The first experiment, in which we try to stimulate exploration by adding an ϵ term to the policy, was not successful. Figure 6.5 shows that for any value of ϵ and the different decay strategies (exponential, linear, no decay) that we tried, the fireworks did not increase, or only barely. We decided to drop this approach.

One possible reason for the fact that this setup did not work could be that this RL algorithm is now off-policy, while the backbone of the algorithm (SPG) expects an on-policy approach. This approach is off-policy because the experience is collected using policy $\pi^{\epsilon}_{\boldsymbol{\theta}_k}$, while the updates are to the network $\pi_{\boldsymbol{\theta}_k}$.

Adding an entropy coefficient does seem to work, as shown in Figure 6.6. The scores and fireworks for $\beta = 0.01$ are reaching the same level as the run without entropy stimulation ($\beta = 0$). From Figure 6.7 is it clear that the mean entropy of the policy is decreasing less quickly when we use a $\beta > 0$, which means that our agent has a greater chance of exploring. Also, the positional bias stayed much lower (especially for discard actions) when using a not-too-small entropy coefficient,

[4] About the speed of decay: ϵ decays in an exponential manner: $\epsilon_t = \epsilon_0 \cdot f^t$, where f is a set decay factor and t is the epoch number in this case. In the first run with $\epsilon > 0$ (fast decay) it is equal to $(0.001/0.5)^{(1/10^4)} \approx 0.9994$ such that ϵ is low (defined as 0.001) at epoch 10^4. In the next run (slow decay) we have $f = (0.001/0.5)^{(1/10^6)} \approx 0.999994$. For the last run $f = (0.001/0.05)^{(1/10^6)} \approx 0.999996$.

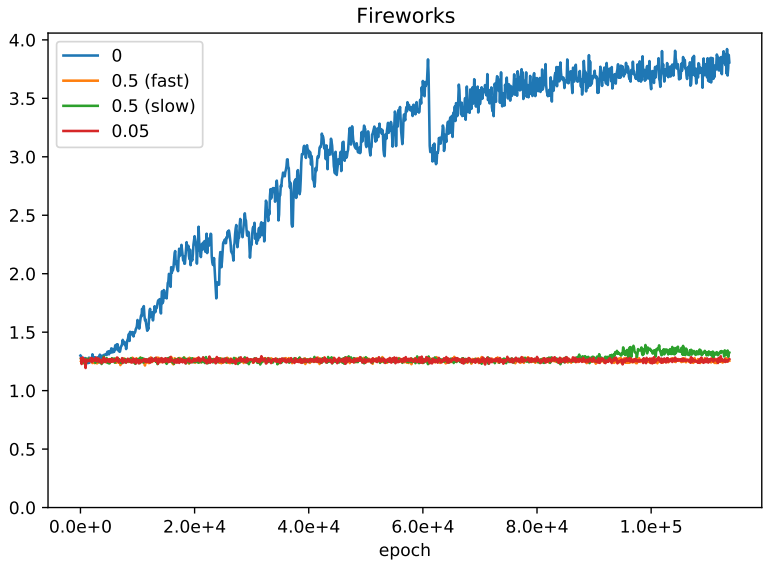


Figure 6.5: Non-zero values for the exploration term ϵ show similar behaviour: not or barely increasing in performance. See column *Epsilon* in Table 6.3 for the settings.

as presented in Figure A.2. This is another indication that our policy is not becoming deterministic too quickly, which is what we were looking for. We decided to continue with $\beta = 0.01$ in our future experiments.

To conclude the entropy experiment, Figure 6.9 displays histograms presenting the average action selection probabilities of our agent's current policy. It is clearly visible that the exploration is hindered without an entropy stimulus, as that agent only plays from index 1 and 4. The play probabilities of the agent with $\beta = 0.01$ are low, but at least they are all larger than 0.

Shuffling the cards did not help to improve the exploration of our agents. In fact, it did not work at all, as shown in Figure 6.10. Whenever we shuffled the cards the score stayed at zero and the fireworks did not increase either. We are not sure about the reason for this. There might be a bug in the code or it could be the case that shuffling the cards is a bad idea in general. Either way, we did not continue with this setup.

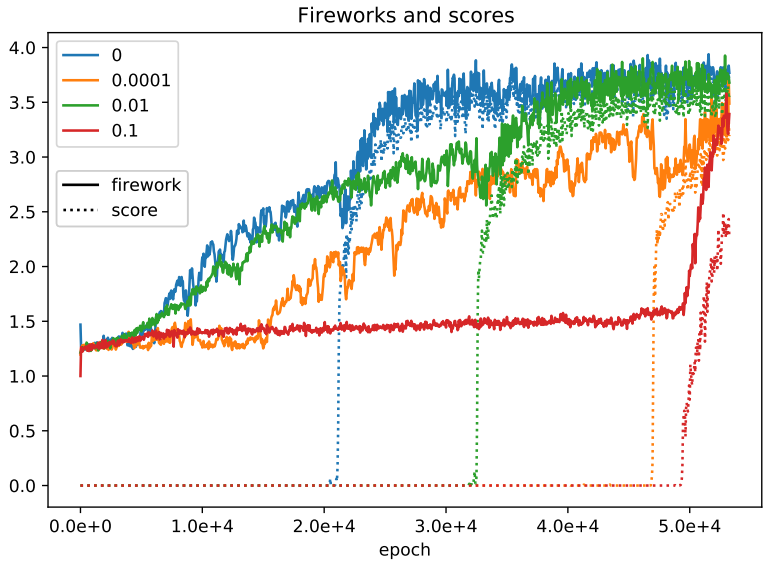


Figure 6.6: Different values for the entropy coefficient β. No entropy term ($\beta = 0$) still seems to work best, although $\beta = 0.01$ is very close. See column *Entropy* in Table 6.3 for the settings.

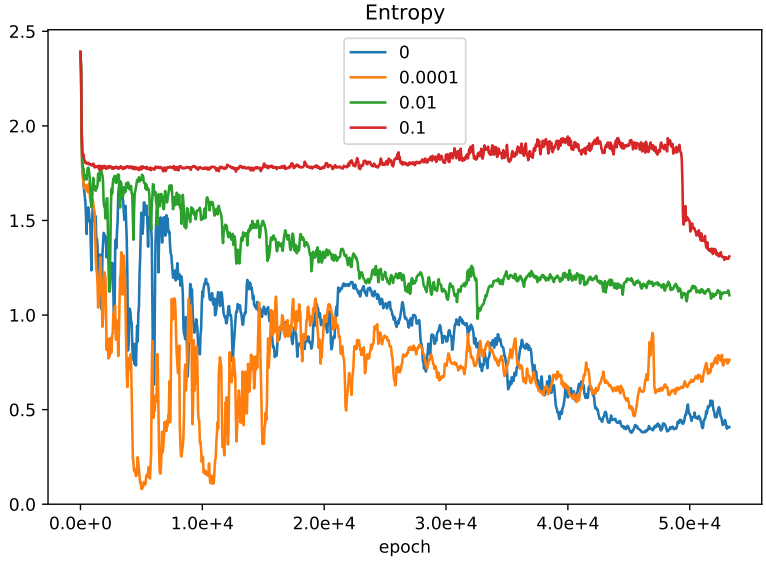
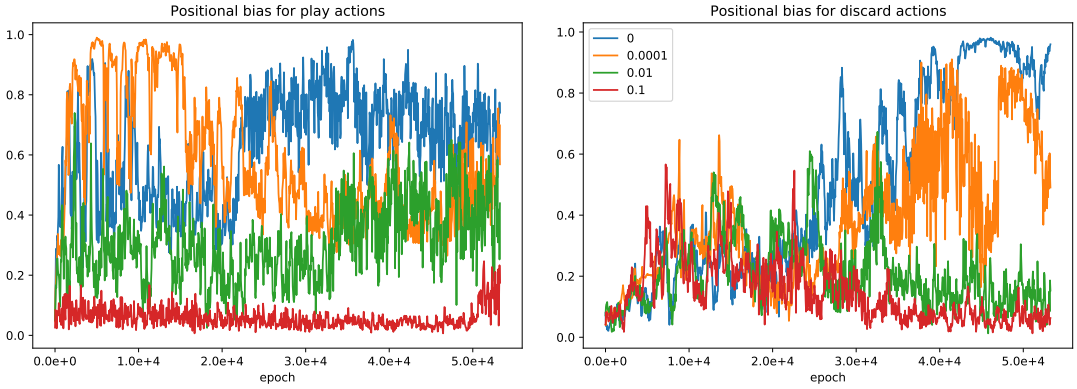


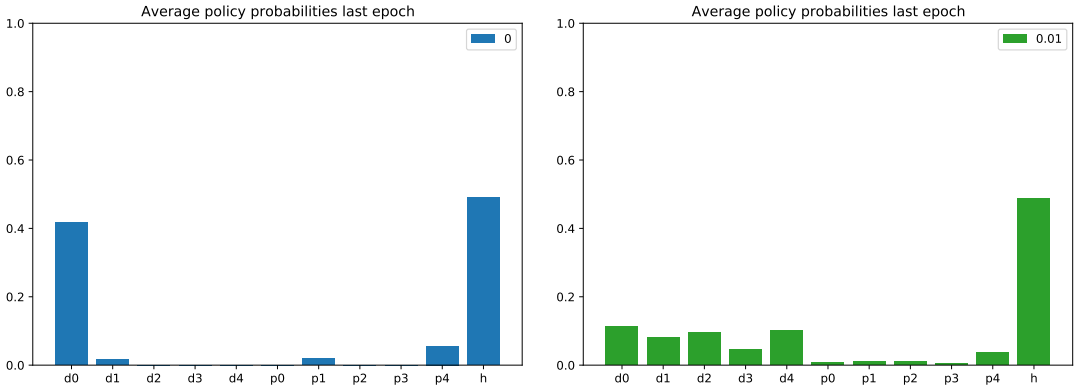
Figure 6.7: Entropy of the policy, with different values for the entropy coefficient β. As expected: the larger the entropy coefficient, the longer the entropy stays high. See column *Entropy* in Table 6.3 for the settings.



(a) Play bias

(b) Discard bias

Figure 6.8: Positional bias for different values of the entropy coefficient β. In general we see that a larger entropy coefficient gives a lower bias. See column *Entropy* in Table 6.3 for the settings.



(a) $\beta = 0$

(b) $\beta = 0.01$

Figure 6.9: The action selection probabilities of our policy during epoch 53,250. The probabilities are averaged over all ∼500 states encountered during the epoch. The labels are: d for discard, p for play, and h for giving a hint. The numbers next to d or p indicate from which index (position in the player's hand) a card is chosen for that action. New cards always enter the hand at index 4, other cards slide to the left (one index lower) if necessary. Notice that our agent often plays with the newest card, while discarding with the oldest. Including an entropy term helps to spread out this positional preference. See column *Entropy* in Table 6.3 for the settings.

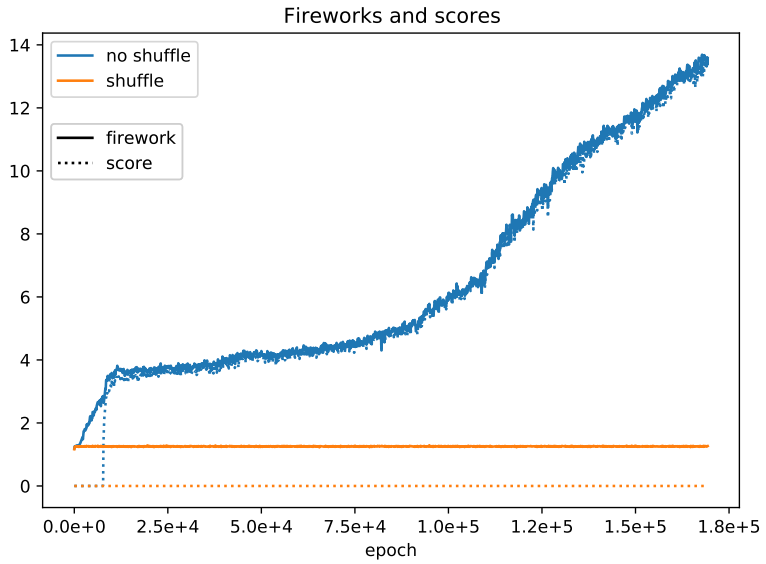


Figure 6.10: Shuffling cards in a player's hand did not work, the agent cannot learn how to increase its performance anymore. See column *Shuffle* in Table 6.3 for the settings.

6.3 Reward shaping

The game of Hanabi has a natural reward system in itself, as we can follow the score to measure the performance of our agent. This gives a +1 reward for every successfully played card, and a reward of $-score$ when the last life token is lost (as the score goes down to zero in that case).

Aside from these standard rewards, we can tweak the reward function to encourage the agent to take (what we think are) beneficial actions, and discourage bad actions. We must be careful not to send the agent's policy in a suboptimal direction, as our estimates of the helpfulness of a particular action could be inaccurate. It can sometimes be better to let the agent figure out the best behaviour by itself without extra reward shaping, as it might discover unknown beneficial moves. On the other hand, reward shaping can assist the agent in getting to a good performance more quickly [25].

In Table 6.4 we have provided an overview of the actions (that lead to certain results) of which we have adjusted the rewards in some experiments. We will proceed with an explanation of the non-standard rewards shown in the table. The illegal actions are dealt with in a special manner, as explained in Section 5.1. Losing one life token is generally bad, but should not be punished too much, as this could scare the agent to take a risky chance once in a while. Discarding, playing, and hinting are neutral actions normally, but we noticed that our agent was hinting and discarding a lot while playing only occasionally. We tried to motivate the agent to play more by giving this action a small positive reward. Furthermore, we shaped the rewards corresponding to different outcomes of a discard action more specifically. Discarding a card that was actually directly playable is almost always a bad move, while getting rid of a useless card[5] is often good. Lastly, discarding a unique card[6] is never advantageous when trying to reach the perfect score.

[5]For example, a card with a rank lower than or equal to the current firework stack of its color. Another possibility: a card with a high rank which is not reachable anymore, since all duplicates of a certain lower rank in the same color have been discarded.

[6]A card of which there is only one copy left in the game. All other duplicates are in the discard pile.

Table 6.4: Different rewards used throughout the design and experimentation process.

Action	Assessment	Default reward	Other rewards tried
Standard Hanabi rewards			
Successfully played a card	good	+1	$+2, +3, +5, +10, +100$
Lost all life tokens	bad	−score	−50
Non-standard rewards			
Illegal action	bad	0	-0.5, -1, -2, -10
Lost one life token	bad	0	−0.1
Discard	neutral	0	−0.02
Play	neutral	0	+0.02
Hint	neutral	0	−0.02
Discard playable	bad	0	−0.1
Discard useless	good	0	+0.1
Discard unique	bad	0	−0.1

We will present a few experiments that use different reward functions. The settings of these experiments are shown in Table 6.5. With the two runs in column A we try to answer the question: Is it beneficial to apply extra[7] reward shaping? The $3 \cdot 4$ runs in the columns under B attempt to discover the best setting for the (arguably) most important reward: when a card is successfully played. We do this separately for each of the three algorithms. Notice that the settings in column B describe the first experiment where we use the algorithms VPG and PPO as well, instead of only SPG.

Results of *Reward shaping*

In Figure 6.11 one can see that reward shaping is useful in our environment. Even though the scores take a little longer to get off 0, the fireworks start developing early on and are not finished yet. We kept using this extra reward shaping in our future experiments.

Figures 6.12, 6.13, 6.14 show the differences of the successful play rewards for SPG, VPG, PPO separately. The results vary quite a lot per algorithm. In SPG and PPO the +5 reward works best, while in VPG it is the worst. The reward of +100 seems to train well in VPG, but becomes unstable near the end. In SPG and PPO this reward also had the most variation.

Some of the runs had so much variation in score, that it made the graphs unreadable.[8] This is why we made the graphs in Figures 6.12, 6.13, 6.14 smoother by averaging the results again over every 5000 epochs (instead of only 100 epochs). Keep in mind that the runs, especially with reward +100, have large variations in performance. It is one of the reasons why we decided to continue with +10 as our main reward for a successfully played card in the experiments to come.

[7] Extra, since we always still have the standard rewards and the necessary punishment for an illegal action.
[8] See Figure A.7 in the Appendix for a comparison.

Table 6.5: Settings of the experiments to compare the reward functions.

	A	B		
Algorithm	SPG	SPG	VPG	PPO
Representations				
State	136	136	136	136
Action	11	11	11	11
Rewards				
Successful play	+1	**+2**	**+2**	**+2**
		+5	**+5**	**+5**
		+10	**+10**	**+10**
		+100	**+100**	**+100**
Lost all lives	−score	−score	−score	−score
Illegal move	−0.5	−1	−1	−1
Lost one life	**0 / −0.1**	−0.1	−0.1	−0.1
Hint	**0 / −0.02**	−0.02	−0.02	−0.02
Play	**0 / +0.02**	+0.02	+0.02	+0.02
Discard	0	0	0	0
Discard playable	**0 / −0.1**	−0.1	−0.1	−0.1
Discard useless	**0 / +0.1**	+0.1	+0.1	+0.1
Discard unique	**0 / −0.1**	−0.1	−0.1	−0.1
Network architecture				
Hidden layers π	$[128, 64]$	$[64, 64]$	$[64, 64]$	$[64, 64]$
Activation function π	Tanh	Tanh	Tanh	Tanh
Hidden layers V	-	-	$[64, 64]$	$[64, 64]$
Activation function V	-	-	Tanh	Tanh
Hyperparameters				
Batch size	1000	1000	1000	1000
Entropy coefficient	0.01	0.01	0.01	0.01
Renormalize	yes	yes	yes	yes
Discount factor	0.99	0.99	0.99	0.99
Learning rate π	10^{-3}	10^{-3}	10^{-3}	10^{-3}
Learning rate V	-	-	10^{-3}	10^{-3}
Update iterations π	1	1	1	5
Update iterations V	-	-	5	5
Advantage type	-	-	GAE	GAE
Lambda	-	-	0.95	0.95
Clipping parameter	-	-	-	0.2

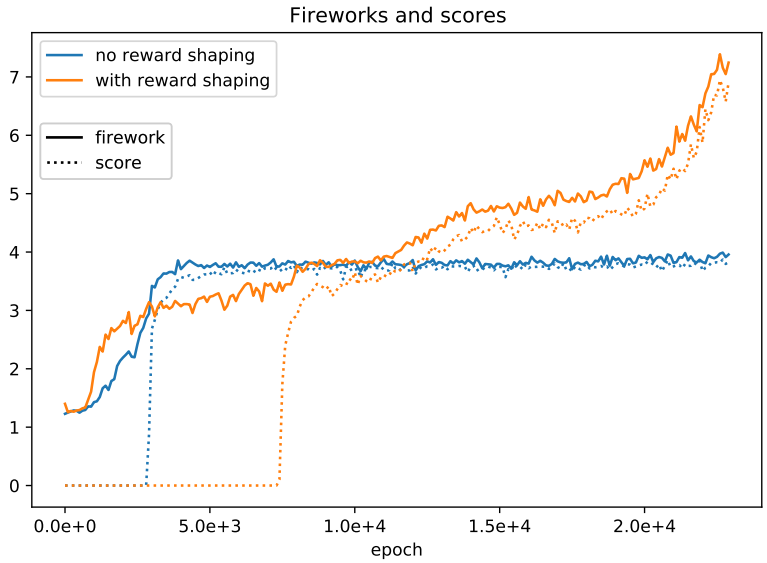


Figure 6.11: Reward shaping seems to be beneficial, as its performance surpasses the plateau of the opposing setting. See column A in Table 6.5 for the settings.

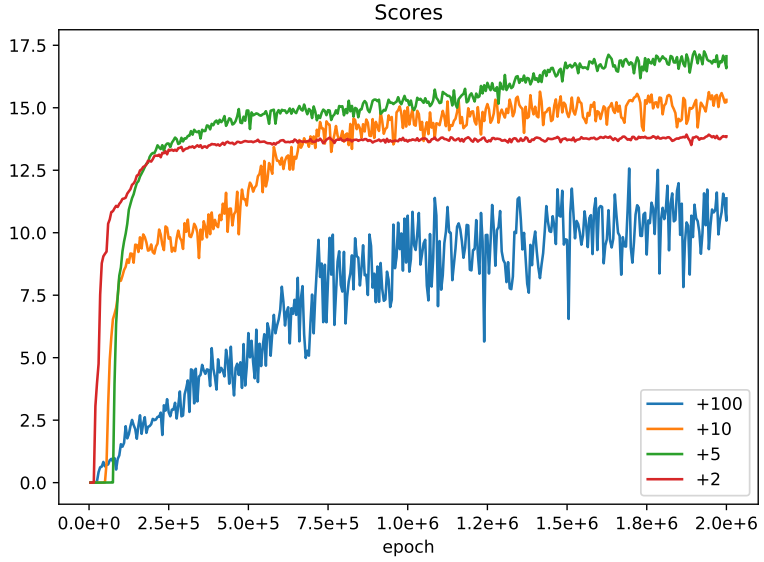


Figure 6.12: Different rewards for a successful play, with SPG. Giving a +5 reward seems to work best, with +10 not far behind. See column SPG under B in Table 6.5 for the settings.

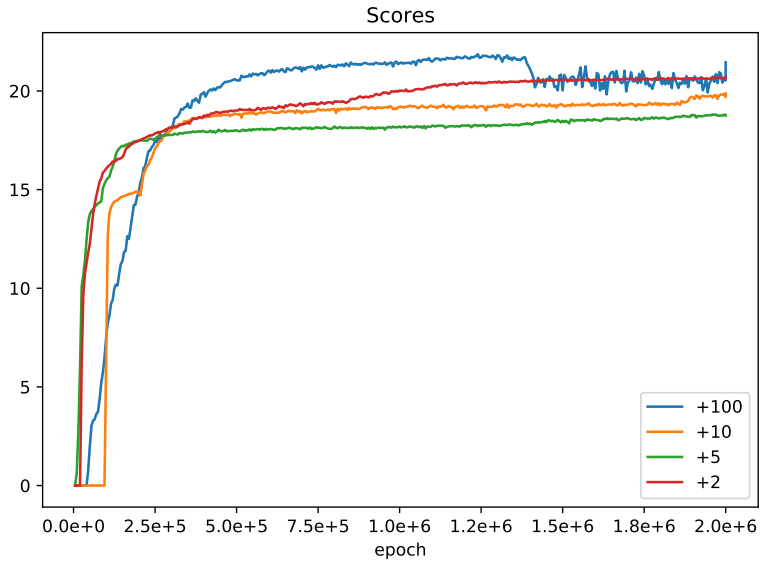


Figure 6.13: Different rewards for a successful play, with VPG. The +100 reward scores best, but shows unstable performance near the end. See column VPG under B in Table 6.5 for the settings.

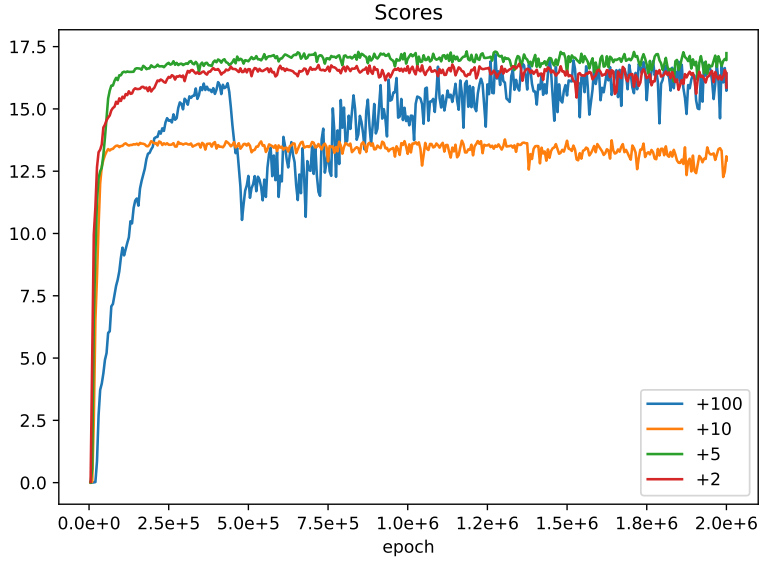


Figure 6.14: Different rewards for a successful play, with PPO. The rewards +2, +5, and +10 learn quickly, but all of them hit a certain plateau. See column PPO under B in Table 6.5 for the settings.

6.4 Deep networks

We started many experiment with relatively shallow networks, containing just a few hidden layers. We were interested to see if deepening the neural networks would be beneficial for the performance and learning pace. For VPG and PPO we extend both the policy network and the value network up to 6 hidden layers. SPG only has a policy network, which we adjust similarly.

The type of the layers remains the same throughout the experiments, being fully connected linear layers. The activation functions in between are all hyperbolic tangents (Tanh). The output layer is the only exception: in the policy network it has a softmax activation to convert the output values to action selection probabilities, in the value network it has the identity function. This makes sure that the only node in the output layer of the value network can produce any number in $\mathbb{R}$.

The experiment settings are shown in Table 6.6. Each column represents 3 runs where one algorithm is tried with different network sizes.

Table 6.6: Settings of the experiments to compare networks of different sizes.

Algorithm	SPG	VPG	PPO
Representations			
State	136	136	136
Action	11	11	11
Rewards			
Successful play	+10	+10	+10
Lost all lives	−score	−score	−score
Illegal move	−1	−1	−1
Lost one life	−0.1	−0.1	−0.1
Hint	−0.02	−0.02	−0.02
Play	+0.02	+0.02	+0.02
Discard	0	0	0
Discard playable	−0.1	−0.1	−0.1
Discard useless	+0.1	+0.1	+0.1
Discard unique	−0.1	−0.1	−0.1
Network architecture			
Hidden layers π	4: $[\mathbf{128(\times 3), 64}]$ 5: $[\mathbf{128(\times 4), 64}]$ 6: $[\mathbf{128(\times 5), 64}]$	4: $[\mathbf{128(\times 3), 64}]$ 5: $[\mathbf{128(\times 4), 64}]$ 6: $[\mathbf{128(\times 5), 64}]$	4: $[\mathbf{128(\times 3), 64}]$ 5: $[\mathbf{128(\times 4), 64}]$ 6: $[\mathbf{128(\times 5), 64}]$
Hidden layers V	-	same as π	same as π
Hyperparameters			
Batch size	1000	1000	1000
Entropy coefficient	0.01	0.01	0.01
Renormalize	yes	yes	yes
Discount factor	0.99	0.99	0.99
Learning rate π	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$
Learning rate V	-	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$
Update iterations π	1	1	5
Update iterations V	-	5	5
Advantage type	-	GAE	GAE
Lambda	-	0.95	0.95
Clipping parameter	-	-	0.2

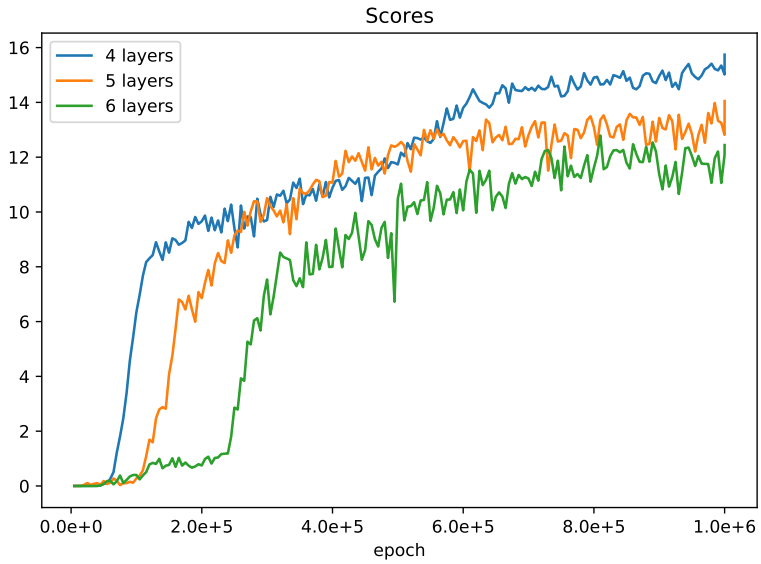


Figure 6.15: Different number of hidden layers, with SPG. The deeper the network, the later the performance starts to increase. See column SPG in Table 6.6 for the settings.

Results of *Deep networks*

In Figures 6.15, 6.16, 6.17 we can see that in general less layers give a better performance. However, it could of course be the case that the deeper networks have the advantage in the long run but need much more training to get there. It interesting to see that especially for PPO smaller networks seem to work better.

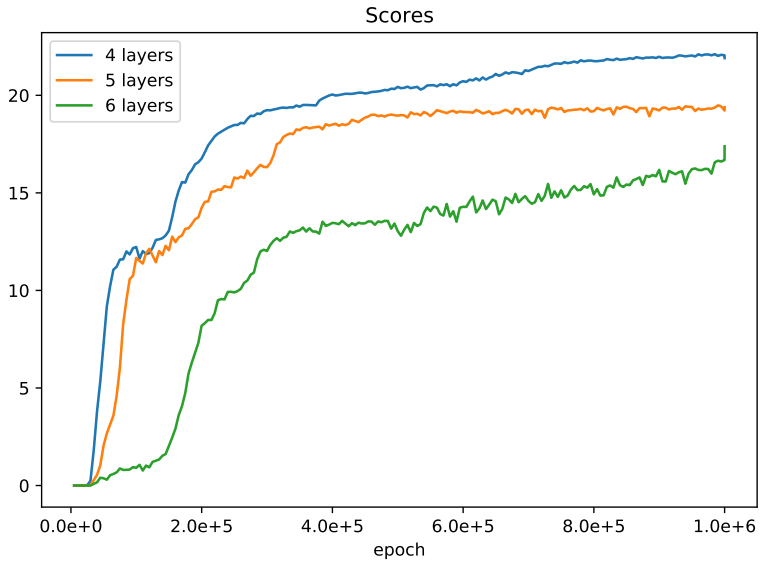


Figure 6.16: Different number of hidden layers, with VPG. Going from 5 to 6 hidden layers seems to be a larger difference than from 4 to 5. See column VPG in Table 6.6 for the settings.

Figure 6.17: Different number of hidden layers, with PPO. All network sizes learn quickly, but 4 hidden layers works best. See column PPO in Table 6.6 for the settings.

6.5 Comparing algorithms

In this final experiment we compare the best runs of each of the three algorithms: Simple Policy Gradient (SPG), Vanilla Policy Gradient (VPG), and Proximal Policy Optimization (PPO). The optimal design choices and hyperparameter settings have been set by the previous experiments. The settings of the runs that we will compare are presented in Table 6.7.

Results of *Comparing algorithms*

The conclusion from the final comparison between the three algorithms shown in Figure 6.18 is quite clear. VPG is the best working algorithm in our environment of simplified Hanabi, scoring an average of 24.4 points. Although we might expect PPO to work better, as it is a newer and arguably more sophisticated algorithm, the agent had hit a plateau around 20. The learning speed of PPO was superior in the beginning, as you can see in Figure 6.19, but after around 10^5 epochs VPG surpassed it. The learning pace of SPG was the slowest, but it is interesting to see that it still has not stopped learning, and could perhaps even outperform PPO if we had continued the experiment for another week.

The outcomes of the 1000 test trials that we ran with the best version of each algorithm are shown in Table 6.8. Histograms of the scores of our best agents are presented in Figures 6.20, 6.21, 6.22. The VPG agent achieves a lot of perfect games, while SPG and PPO struggle to reach 25 points.

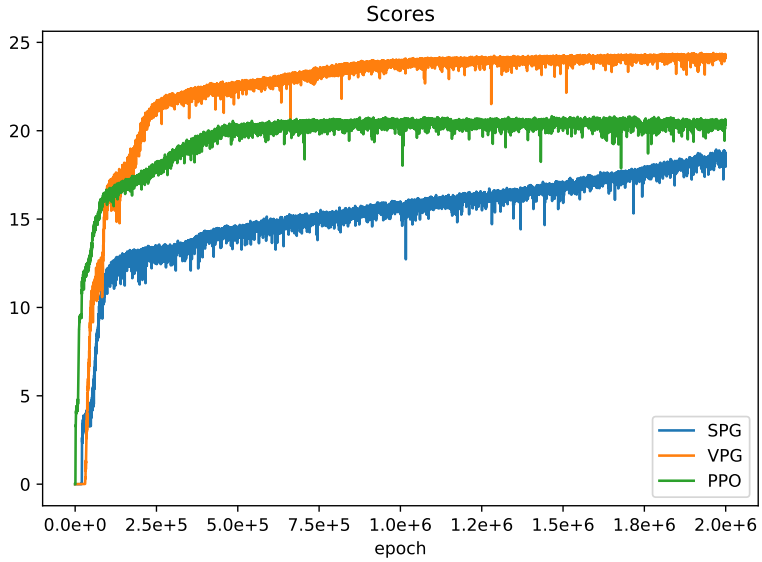


Figure 6.18: Comparing the three algorithms. VPG performs best, as PPO is unable to surpass its plateau. SPG is still slowly but steadily increasing. See Table 6.7 for the settings.

Table 6.7: Settings of the experiments to compare the three algorithms.

Algorithm	**SPG**	**VPG**	**PPO**
Representations			
State	136	136	136
Action	11	11	11
Rewards			
Successful play	+3	+10	+1
Lost all lives	−score	−score	−score
Illegal move	−1	−1	−1
Lost one life	−0.1	−0.1	−0.1
Hint	−0.02	−0.02	−0.02
Play	+0.02	+0.02	+0.02
Discard	0	0	0
Discard playable	−0.1	−0.1	−0.1
Discard useless	+0.1	+0.1	+0.1
Discard unique	−0.1	−0.1	−0.1
Network architecture			
Hidden layers π	$[128, 128, 64]$	$[128, 128, 64]$	$[128, 128, 64]$
Hidden layers V	-	$[128, 64, 32]$	$[128, 64, 32]$
Hyperparameters			
Batch size	1000	1000	1000
Entropy coefficient	0.01	0.01	0.01
Renormalize	yes	yes	yes
Discount factor	0.99	0.99	0.99
Learning rate π	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$
Learning rate V	-	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$
Update iterations π	1	1	5
Update iterations V	-	5	5
Advantage type	-	GAE	GAE
Lambda	-	0.95	0.95
Clipping parameter	-	-	0.2

Table 6.8: Metrics of 1000 test games after two million epochs of training. The corresponding settings are shown in Table 6.7.

	SPG	VPG	PPO
Mean score	18.37	24.38	20.50
Median score	18	25	21
Standard deviation	2.33	1.20	2.16
Perfect games	0.0%	70.5%	1.0%

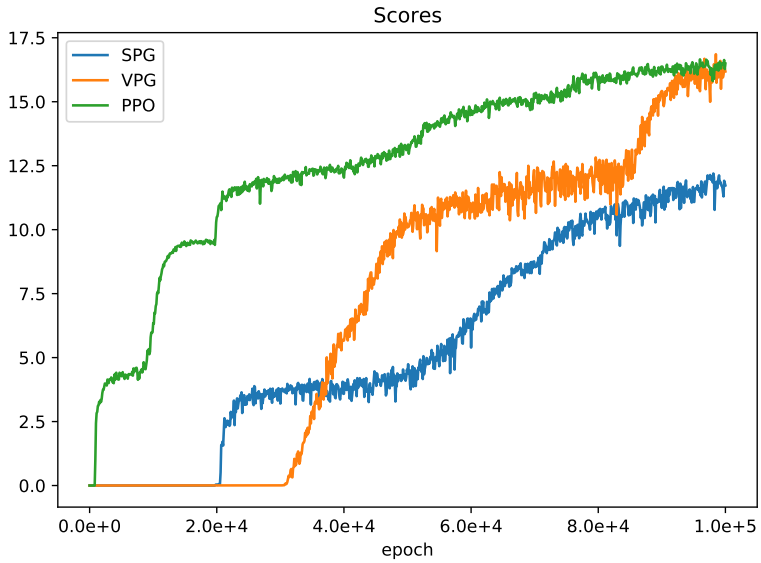


Figure 6.19: Comparing the three algorithms after just 10^5 epochs. PPO learns very quickly, but is about to be outperformed by VPG. See Table 6.7 for the settings.

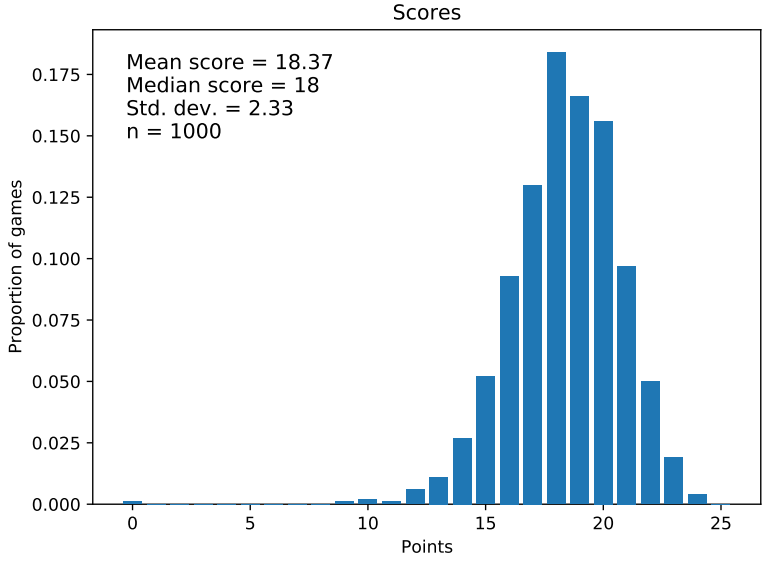


Figure 6.20: Scores of the SPG agent in 1000 test games after two million epochs of training. Although some runs came close, no perfect games have been played. See column SPG in Table 6.7 for the settings.

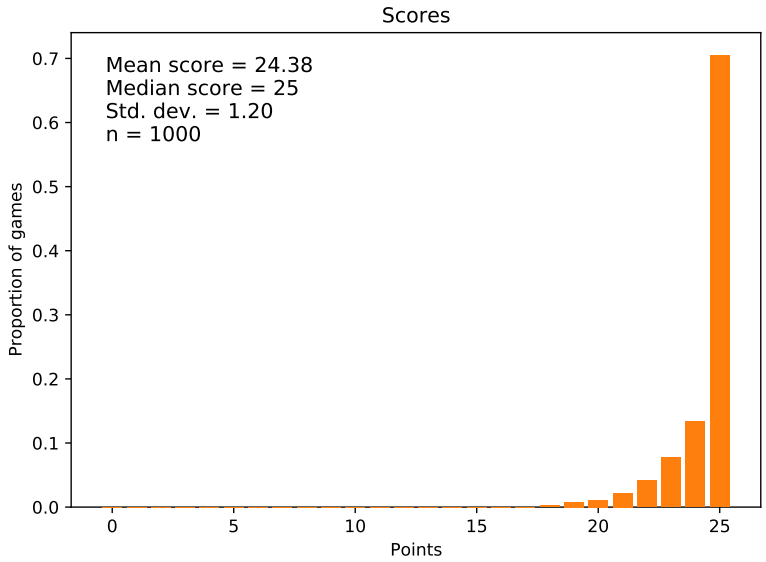


Figure 6.21: Scores of the VPG agent in 1000 test games after two million epochs of training. The agent reached the perfect score in about 70% of the games. See column VPG in Table 6.7 for the settings.

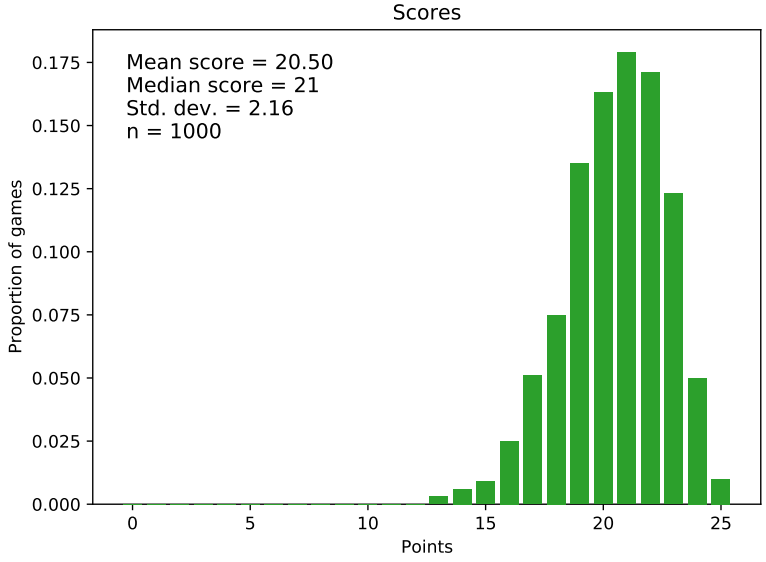


Figure 6.22: Scores of the PPO agent in 1000 test games after two million epochs of training. In a little more than half the games the agent scored higher than 20. See column PPO in Table 6.7 for the settings.

Chapter 7

Game properties

In this chapter we will analyze some game-theoretic properties of Hanabi. When programming an algorithm for a game such as Hanabi, it is useful to know how many turns a game can take, so we include a proof of the maximum length of a game in Section 7.1. In Section 7.2 we adjust the proof to the maximum length of a *perfect* game, for which we have a different answer than found in literature. The number of possible hands of cards is computed in Section 7.3. This might be useful information for algorithms that tackle the full version of Hanabi, where a belief over the cards in the current player's hand could be tracked. Finally, in Section 7.4 we show that many Hanabi states can only be reached at one specific time step, which provides some insight in the type of Markov decision process we are working with.

7.1 Maximum length of a game

In this section we will investigate the maximum length of a Hanabi game. With "a Hanabi game" we mean a game of Hanabi according to the rules described in Section 1.1.

Proposition 4. *The maximum length of a Hanabi game is 89 turns.*

Proof. This proof consists of two parts. First we will show that there exists a Hanabi game of length 89. After that we will prove that no Hanabi game can have a higher number of turns than 89.

Part 1. Take a Hanabi game of 2 players. At the start, each player has 5 cards so there are 40 cards left in the deck. Suppose the players start the game by giving hints until all information tokens are gone. This takes 8 turns. Then they start a pattern by alternating one discard action and one hint action, continuing until the deck is empty. After the last discard action (which empties the deck) there have been 40 discard actions, with 39 hints in between. Each player gets one more turn, in which they could discard another card. This gives a total of $8 + 40 + 39 + 2 = 89$ turns.

Part 2. In this part we will define a value Σ_t for a Hanabi game. We will show that it is impossible for this value to increase during the game, from which the maximum number of turns follows. The value Σ_t can be interpreted as the maximum possible number of total turns that is still reachable, after time step t.

In a Hanabi game, we denote the current total number of turns taken by t, and the current deck size after turn t by d_t. The number of hint tokens left over after turn t, with the restriction that these tokens can still be used before the deck is empty, is denoted by c_t:

$$c_t = \begin{cases} m_t & \text{if } d_t > 0, \\ 0 & \text{if } d_t = 0. \end{cases} \tag{7.1}$$

Recall from Section 3.1 that m_t is the number of hint tokens available, without any restrictions. We add the restriction to c_t here to differentiate between the situations before and after the deck has been emptied. This is because once the deck is empty, hint actions cannot be used to stall the game anymore. At that point, when $d_t = 0$, there is a fixed maximum number of turns left, which we denote by p_t (initially equal to the number of players p).

We define one more value: u_t, which we call the *undisclosed hints*. This value keeps track of how many cards can still retrieve a hint token which could be used before the deck is empty. We have:

$$u_t = \begin{cases} d_t - 1 & \text{if } d_t > 0, \\ 0 & \text{if } d_t = 0. \end{cases} \tag{7.2}$$

Every card that is played or discarded can retrieve a hint token. This can be done d_0 times in total and then the deck is empty. However, if the last card that empties the deck retrieves a hint token, this token is only usable after the deck is empty. Thus, the value of u_t is always one less than the current deck size d_t (except when the deck is already empty).

We will now look into the effect of different actions on the values of t, c_t, d_t, u_t, and p_t. A player can choose three actions in each turn: play, discard, or hint. The effect of each action on the different values is summarized in Table 7.1. Notice that with every *play* or *discard* action the number of undisclosed hints decreases by one: $u_t = u_{t-1} - 1$, as there is one less card available to retrieve a hint token from. We denote this change as $\Delta u_t = u_t - u_{t-1}$. The special cases are:

- When a card with rank 5 is played successfully the number of hint tokens increases by one, if the number of hint tokens is not already at its maximum.
- When the deck is emptied (by a *play* or *discard* action at $d_{t-1} = 1$), the value c_t drops down to zero. Any left over hint tokens cannot be used before the deck is empty, anymore. The number of undisclosed hints is already at zero when $d_{t-1} = 1$, so it does not change.
- When the deck is already empty, any action only decreases p_t by one (and increases t). It has no influence on c_t, d_t or u_t, which are all at zero already.

The value Σ_t is defined as the sum over all values included:

$$\Sigma_t = t + c_t + d_t + u_t + p_t. \tag{7.3}$$

Recall that Σ_t represents the maximum possible number of total turns that is still reachable, after time step t.

Table 7.1: Effect of actions on the different values. The three exceptions at the bottom have priority over the three standard actions at the top.

action	Δt	Δc_t	Δd_t	Δu_t	Δp_t	$\Delta \Sigma_t$
play	$+1$	0	-1	-1	0	-1
discard	$+1$	$+1$	-1	-1	0	0
hint	$+1$	-1	0	0	0	0
play a rank 5 card successfully[1]	$+1$	$+1$	-1	-1	0	0
action that empties the deck	$+1$	$-c_{t-1}$	-1	0	0	≤ 0
action when the deck is empty	$+1$	0	0	0	-1	0

[1] Only if $d_{t-1} > 1$ (otherwise it counts as an action that empties the deck or happens when the deck is empty) and $m_{t-1} < 8$ (otherwise it counts as a normal play action, since we do not gain a hint token with a rank 5 card if the hint budget is already full).

Thus, the value of Σ_t can never increase during a game. Furthermore, the values t, d_t, p_t, and m_t must always stay non-negative according to the rules of Hanabi. This also implies that the values of c_t and u_t must always be non-negative, since m_t and d_t in (7.1) and (7.2) are non-negative and integer. With this information, and from (7.3), we can conclude that we must always have $t \leq \Sigma_t$.

Thus, the maximum value that t could possibly reach is equal to the value of Σ_0 (before any action has been taken). These starting values of Σ_t are shown for every possible number of players p in Table 7.2. We always have $c_0 = 8$, and with the number of players $p = 2, 3, 4, 5$ the initial deck size after everybody has been given their opening hands is $d_0 = 40, 35, 34, 30$ respectively.

Table 7.2: Starting values of Σ_t.

p	2	3	4	5
Σ_0	89	80	79	72

As shown in Part 1, there is a particular sequence of actions in a Hanabi game, that gives the outcome of Table 7.3.

Table 7.3: Possible values of t, c_t, d_t, u_t, p_t for a two-player game.

	t	c_t	d_t	u_t	p_t	Σ_t
start	0	8	40	39	2	89
end	89	0	0	0	0	89

Therefore, the maximum length of a Hanabi game is 89 turns.

7.2 Length of a perfect game

The maximum length of a *perfect* Hanabi game, where 25 points are scored, is 71 turns (not 65, as stated briefly in [14]). We prove this in the following proposition.

Proposition 5. *The maximum length of a perfect Hanabi game is 71 turns.*

Proof. This proof consists of two parts. First we will show that there exists a perfect Hanabi game of length 71. After that we will prove that no perfect Hanabi game can have a higher number of turns than 71.

Part 1. Assume a two player Hanabi game. The initial deck size is 40. The players start out by spending their 8 hints. Then they play 22 cards successfully, finishing four fireworks. This gives them 4 extra hints, which they use immediately. The players now start a pattern of first discarding one card, and then giving one hint. This can be done 17 times. Then 1 card is played successfully that empties the deck. Both players have one more turn, in which they successfully play the rank 4 and 5 cards of the remaining firework. The number of turns is $8 + 22 + 4 + 17 \cdot 2 + 1 + 2 = 71$.

Part 2. In this part we will use the same values as defined in the proof of Proposition 4. We use the fact that it is impossible for Σ_t to increase during the game, and show that it must decrease to at most 71 for a game to finish in a perfect score.

To reach the perfect score, 25 cards must be played successfully. This means we need at least 25 play actions. In Table 7.1 it is shown that every play action decreases Σ_t by 1, aside from a few exceptions. These exceptions are:

(1) Play a rank 5 card successfully when the number of hint tokens is less than 8.

(2) Any action that empties the deck when the number of hint tokens is 0.

(3) Any action when the deck is empty.

These exceptions can all be play actions that do not decrease the value of Σ_t. Let's try to keep Σ_t as high as possible (as it represents the maximum number of turns we can reach) while still scoring 25 points. For this we need to make sure as many play actions as possible are classified as one of the three exceptions.

A perfect game can end before the deck is empty, on the deck-emptying move, or when it is already empty. Let us investigate the maximum number of exception play moves in all cases.

If the game ends

- before the deck is empty: we can use exception (1) five times,
- on the deck-emptying move: we can use (1) four times, and must use (2) once,
- when the deck is empty: we can use (1) four times (not five, then the game would be over already), we must use (2) once, and we can use (3) p number of times. Recall that p stands for the number of players.

From all these cases, we see that the maximum possible number of exception play moves is $5 + p$. In a two player game, this would mean that 7 play moves do not decrease Σ_t, while the other $25 - 7 = 18$ do. The maximum number of turns in that case is $89 - 18 = 71$. Recall that 89 is the starting value of Σ_t in the two player case, see Table 7.2.

Table 7.4: Maximum potential number of turns for each number of players.

number of players	maximum value of Σ_t at end of perfect game
2	$89 - (25 - (5 + 2)) = 71$
3	$80 - (25 - (5 + 3)) = 63$
4	$79 - (25 - (5 + 4)) = 63$
5	$72 - (25 - (5 + 5)) = 57$

An overview of the maximum potential number of turns for different values of p is shown in Table 7.4. We see that in the two player case this value is the highest, meaning that no perfect Hanabi game can possibly be longer than 71 turns. In Part 1 we have shown that a perfect game of this length is indeed possible. Therefore, the maximum length of a perfect Hanabi game is 71 turns.

7.3 Number of possible hands

The number of possible distinct hands for one player in Hanabi is equal to $9,095,150$ when the hand size h is 5 cards. Notice that this is a bit less than the naive estimation of $25^5 = 9,765,625$ which only uses the fact that there are 25 unique cards. However, it is not possible to have more than three duplicates of a card.

The order of the cards in a player's hand matters, as players need to know specifically which card to play. For instance, the hand $(R1, R1, G3, G4, Y2)$ is counted as a different hand than $(G3, R1, R1, G4, Y2)$ even though the same cards are included.

The calculation is shown in Table 7.5, where $\binom{5}{2,2,1} = 30$ according to the multinomial expression. The numbers in the grouping column represent the number of duplicate cards. For example, 1-1-1-1-1 means all cards are different, while 3-1-1 means there are three cards with the same color and rank, and two other distinct cards.

Notice that there are 25 unique cards to choose from. When a card needs to be chosen twice, then there are 20 unique possibilities: all ranks except the 5's. And finally, there are only 5 unique possibilities if a card is included in a hand three times: only the rank 1 cards.

Table 7.5: Number of possible hands in Hanabi when $h = 5$.

grouping	hands
1-1-1-1-1	$25 \cdot 24 \cdot 23 \cdot 22 \cdot 21$
2-1-1-1	$\binom{5}{2} \cdot 20 \cdot 1 \cdot 24 \cdot 23 \cdot 22$
2-2-1	$\binom{5}{2,2,1} \cdot 20 \cdot 1 \cdot 19 \cdot 1 \cdot 23$
3-1-1	$\binom{5}{3} \cdot 5 \cdot 1 \cdot 1 \cdot 24 \cdot 23$
3-2	$\binom{5}{3} \cdot 5 \cdot 1 \cdot 1 \cdot 19 \cdot 1$
total	9,095,150

In the case when the hand size is 4, then the total number of possible hands is only 372,600 as shown in Table 7.6. This is also less than the number one would get with the naive estimation: $25^4 = 390,625$.

Table 7.6: Number of possible hands in Hanabi when $h = 4$.

grouping	hands
1-1-1-1	$25 \cdot 24 \cdot 23 \cdot 22$
2-1-1	$\binom{4}{2} \cdot 20 \cdot 1 \cdot 24 \cdot 23$
2-2	$\binom{4}{2} \cdot 20 \cdot 1 \cdot 19 \cdot 1 \cdot 23$
3-1	$\binom{5}{3} \cdot 5 \cdot 1 \cdot 1 \cdot 24 \cdot 23$
total	372,600

7.4 The time step of a state

Many states in the game of Hanabi can only be reached at a single fixed time step. Only the states where the fireworks have been completed for some colors have a few extra time step possibilities.

Proposition 6. *Any Hanabi state s which has no completed fireworks yet can only be reached at precisely one specific time step t.*

Proof. We prove this by taking an arbitrary Hanabi state without completed fireworks s, and then computing its corresponding time step t, arguing that this is the only possible t for s.

Recall that in Section 3.2.1 we defined a Hanabi state to be of the form: $s = (\boldsymbol{f}, \boldsymbol{h}, Z, m, l, d, i)$. The elements represent the current fireworks, hand of every player, discard pile, hint tokens, life tokens, deck size, and current player.

From the fireworks $\boldsymbol{f}$ we can see how many cards have been played successfully. Let us call this number f. From the life tokens l we can compute how many cards have been played unsuccessfully. Define: $u = l_{\max} - l = 3 - l$. This means the total number of play actions was: $a_p = f + u$.

With u we can compute how many discard actions have taken place, as we subtract it from the size of the discard pile: $a_d = |Z| - u$.

We then compute the total number of hint actions taken by looking at the number of hint tokens m and noting the fact that hint tokens were added with each discard action. No fireworks have been completed yet, so this means no extra hint tokens have been added as a result of successfully playing a rank 5 card. The total number of hint actions becomes: $a_h = a_d + m_{\max} - m = a_d + 8 - m$. Note that discarding is illegal if $m = m_{\max}$, so indeed every discard action produces a hint token.

Any actions in Hanabi can only be a play, discard, or hint action. This means that the total number of turns taken to reach state s must have been: $t = a_p + a_d + a_h$. It is impossible to reach s in a different number of time steps $\hat{t}$, as this would require a different number of play, discard or hint actions, which would result in a different state $\hat{s}$.

If state s has some completed fireworks, then this argument does not hold. This is because a hint token is added for every action that successfully plays a rank 5 card, *except* when the hint token budget is already full at that moment. Whether or not this hint token was added in the past cannot be read from the current state. Thus, a state with 2 finished fireworks for example, could have had 0, 1, or 2 extra hint actions in the past.

Chapter 8

Discussion

In this chapter we will analyze unexpected results and describe what we could have done better. We discuss these lessons in Section 8.1, while providing potential directions for future research in Section 8.2.

8.1 Lessons learned

An unexpected result was the fact that VPG outperformed PPO in our experiments, despite PPO being the newest algorithm of the two. Perhaps VPG really does work better for simplified Hanabi, or possibly PPO did not reach its full potential. One of the reasons for the latter could be the following. Early in our research project many hyperparameter were tweaked, and then left on their best performing setting. Many of these parameters, such as the learning rate, discount factor, and batch size, were set by trials of SPG or VPG. PPO was the last algorithm we implemented, for which we have not repeated most of these hyperparameter searches. It could be the case that a batch size of 1000 works great for VPG, but is actually not so effective for PPO.

One of the methods we should have used earlier in the project was to analyze the results of different runs in the same graph, as they are shown in Chapter 6. This gives a clearer comparison of the performance, instead of looking at many separate graphs.

During the project, there were many design options and hyperparameter settings that we tried, often quickly jumping from the previous idea to the next. It would be a good strategy to go through these trials in a more structured manner, and document their performance in an overview table as we did near the end. This table is included in Appendix A.4.

It would have been better if we repeated a some more settings that we already tried before, as they might have worked well in combination with new design options or new algorithms. For example, we could have given observation vector 62 another chance, or tried out a different learning rate for PPO.

During our experiments we learned that the batch size should not be too small. Setting it to 1000 environment steps means that usually about 10 to 20 episodes are played, which is apparently what we needed. Increasing the batch size beyond that did not improve learning much, while it did slow down the rate of subsequent policy updates of course, as it takes more time to collect experience.

We noticed that Generalized Advantage Estimation (GAE) seems to work better in combination with multiple policy update iterations per epoch. When using just one policy update iteration per epoch, the basic advantage estimation worked at least as well for VPG. In PPO it is essential to use multiple iterations, so we have not tried performing only one. This would have been an interesting experiment to be able to check the theory.

An important lesson from my supervisors is: do not be afraid to change some settings drastically. You might discover a well working setting, even though you did not expect it to work. The reward settings are an example of this. Make sure you are able to measure the performance of the algorithm separately from the returns if you change the rewards a lot. It is difficult to compare different reward settings if you do not have a separate performance measure. In our case we looked at the Hanabi score or the firework stacks.[1]

8.2 Future research

We would love to see future students or other researchers continuing this project. To help them forward, we provide a few potential directions.

First and foremost, it would be interesting to extend our algorithms to the full version of Hanabi. Deep reinforcement learning algorithms have already been applied to Hanabi, as presented in Section 2.2. However, the state-of-the-art is held by a purely value-based approach, so it would be interesting to see if this performance can be reached with actor-critic algorithms as well. The greatest challenge for this extension is the partial observability that comes into play. We provide some advice on how to deal with this in Appendix B.1.

We believe that our work has set up a good starting point to continue with actor-critic algorithms in the full version of Hanabi. Algorithms based on VPG or PPO could perhaps be contenders for the state-of-the-art, if the necessary adjustments and additions are made. For example, our observation vector needs to change such that the cards of the current player are not visible anymore. The output vector of our policy network must increase in length to include probabilities for distinct hint actions. Additionally, we advise to compute a belief of the current player in every time step and include this in the observation vector. This belief can contain for example: the player's own cards, the next action of other players, and the belief of other players on their cards. Keeping a memory of positive and negative hints received on a card may be helpful to compute the beliefs.

In the Deep Q-Network of Hu and Foerster [21] a clever trick is used to update these beliefs during training. It can be difficult to form a belief when other agents are occasionally just making random moves, due to their ε-greedy approach. So whenever an agent takes a random action, the other agents now also observe its preferred action. They update their beliefs based on the latter. A similar approach is possible for policy-based algorithms, where the preferred action can be defined as the action with the highest selection probability (instead of the highest Q-value).

A separate option is of course to disregard the actor-critic approach, and focus on improving the deep Q-learning algorithms of [19, 21]. In self-play these algorithms might be hard to beat, but especially in the ad-hoc play domain there is quite some room for improvement. This domain is much more difficult to grasp, as agents should be able to play well with a large variety of teammates. Defining a clear and concrete way to measure the success of an agent in ad-hoc play would also be a valuable contribution.

A less complicated, but still interesting extension would be to redesign our algorithms so that they are applicable to the 3, 4, and 5 player setting of Hanabi as well, instead of only 2. In simplified Hanabi this should not make much of a difference. However, in the full version this adds quite some complexity in deciding which player the agent wants to give a hint to.

The rules of Hanabi are pretty strict when the 3 life tokens are lost: the score goes back to zero. Perhaps it is better for a learning agent to relax this restriction, and leave the score at the current fireworks. Some other papers do this as well, as explained in [8]. If you want to adhere to the rules, a similar possibility would be to only remove the negative reward corresponding to a lost third life token.

[1] We used the fireworks especially when all runs of an experiment lost 3 lives most of the time.

There are still some design choices and hyperparameter settings that we have not adjusted a lot. For example, we used linear, fully connected layers in all our experiments. It might be interesting to try other types of layers or add sparsity to potentially increase the performance. Similarly, we have not experimented with many activation functions. The *Tanh* function, often used in DRL, worked better than $ReLU$ for us. Other activation functions could be tried out as well.

Including the current deck size in the observation vector might help increase the performance near the end of a game. We thought it was not important enough, but it might be the case that an agent pays attention to this information when the deck is almost empty. It could be sufficient to add one normalized value representing the current deck size divided by the maximum deck size.

Perhaps it would be beneficial to deal with illegal actions in a different way than we did. Giving them a negative reward does not rule them out completely. Purely value-based algorithms such as Deep Q-Networks can easily solve this by giving an illegal action a Q-value of negative infinity. The analogous approach for policy-based algorithms would be: just make $\pi(a|s) = 0$ for all illegal actions and renormalize the rest. However, this gave us an occasional error when the probabilities of all the legal actions were zero. We are definitely curious for better ways to manage illegal actions in policy-based algorithms.

Chapter 9

Conclusions

The goal of this research was to gain a better understanding of deep reinforcement learning (DRL), specifically actor-critic algorithms. We did this by mathematically describing the three algorithms Simple Policy Gradient (SPG), Vanilla Policy Gradient (VPG), Proximal Policy Optimization (PPO), and comparing their performance in a simplified version of Hanabi. Overall, our best agent is able to play 70% perfect games in the two-player self-play setting.

In our research questions we asked:

1. How can we develop a DRL algorithm that can play a simplified version of Hanabi well?
2. What are the mathematical descriptions of the DRL algorithms SPG, VPG, and PPO?
3. What are game-theoretical properties of Hanabi relevant to the design of an algorithm?

For the properties of Hanabi, we have proven that the maximum length of any game is 89 turns, while the maximum length of a perfect game is 71 turns. Furthermore, we showed that many states in the Markov decision process representing Hanabi can only be reached at one specific time step.

We provided a mathematical description of SPG, VPG, and PPO. In particular, we proved the Policy Gradient Theorem in more detail than often found in popular texts, formalizing the repeated unpacking of ∇v. The most important difference between VPG and PPO lays in the objective function. VPG (and SPG) maximize the expected total return, while PPO uses a *clip* function to restrict the policy from deviating too much. It is essential for PPO to update to the parameters more than once per batch of experience, otherwise this *clip* function is never used.

Finally, we designed a DRL algorithm that can play simplified Hanabi well. We implemented SPG, VPG, and PPO, after which we experimented with various algorithmic design options and hyperparameter settings. The options that were beneficial include: stimulating exploration with an entropy term, shaping the reward function to focus on playing cards successfully, and dealing with illegal actions through negative rewards.

We noticed that including a value function estimator, which VPG and PPO have, works well as they both outperform SPG. Our implementation of VPG achieves the highest average score of 24.4 out of 25, while PPO and SPG reached 20.5 and 18.4 respectively. Interestingly, the learning pace of PPO was much better than that of VPG in the beginning, but VPG surpassed PPO in performance after about 100 million environment steps.

We hope future research can extend our work to Hanabi matches with more than two players and tackle the full version of the game, perhaps even in the complex ad-hoc setting.

References

[1] Hanabi, Rules of Play. https://rulesofplay.co.uk/products/hanabi-en. Accessed: 2021-04-14. 1

[2] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018. Baselines in Policy Gradients, https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html#baselines-in-policy-gradients. Accessed: 2021-08-06. 22

[3] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018. Implementing Reward-to-Go Policy Gradient, https://github.com/openai/spinningup/blob/master/spinup/examples/pytorch/pg_math/2_rtg_pg.py. Accessed: 2021-08-06. 23, 24

[4] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018. Vanilla Policy Gradient Pseudocode, https://spinningup.openai.com/en/latest/algorithms/vpg.html#pseudocode. Accessed: 2021-08-06. 25, 26

[5] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018. Proximal Policy Optimization Pseudocode, https://spinningup.openai.com/en/latest/algorithms/ppo.html#pseudocode. Accessed: 2021-08-06. 30

[6] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the Atari Human Benchmark. In *Int. Conf. on Machine Learning*, pages 507–517. PMLR, 2020. 8

[7] Jean-François Baffier, Man-Kwun Chiu, Yago Diez, Matias Korman, Valia Mitsou, André van Renssen, Marcel Roeloffzen, and Yushi Uno. Hanabi is NP-complete, Even for Cheaters who Look at Their Cards. *CoRR*, abs/1603.01911, 2016. 3

[8] Nolan Bard, Jakob Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H. Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, Iain Dunning, Shibl Mourad, Hugo Larochelle, Marc Bellemare, and Michael Bowling. The Hanabi challenge: A new frontier for AI research. *Artificial Intelligence*, 280:103216, 2020. 4, 5, 65

[9] Rodrigo Canaan, Julian Togelius, Andy Nealen, and Stefan Menzel. Diverse Agents for Ad-Hoc Cooperation in Hanabi. *CoRR*, abs/1907.03840, 2019. 4

[10] Christopher Cox, Jessica de Silva, Philip Deorsey, Franklin Kenter, Troy Retter, and Josh Tobin. How to Make the Perfect Fireworks Display: Two Strategies for Hanabi. *Mathematics Magazine*, 88(5):323–336, 2015. 4

[11] DeepMind. The Hanabi Learning Environment. https://github.com/deepmind/hanabi-learning-environment. Accessed: 2020-09-01. 4

[12] Markus Eger, Chris Martens, and Marcela Alfaro Córdoba. An Intentional AI for Hanabi. In *2017 IEEE Conf. on Computational Intelligence and Games (CIG)*, pages 68–75. IEEE, 2017. 4

[13] James Nesta et al. Hanabi Conventions for The Hyphen-ated Group. https://github.com/Zamiell/hanabi-conventions. Accessed: 2020-09-16. 4

[14] Jakob Foerster, H. Francis Song, Edward Hughes, Neil Burch, Iain Dunning, Shimon Whiteson, Matthew Botvinick, and Michael Bowling. Bayesian Action Decoder for Deep Multi-Agent Reinforcement Learning. *CoRR*, abs/1811.01458, 2018. 5, 60

[15] Andrew Fuchs, Michael Walton, Theresa Chadwick, and Doug Lange. Theory of Mind for Deep Reinforcement Learning in Hanabi. *arXiv preprint arXiv:2101.09328*, 2021. 85

[16] Katja Grace, John Salvatier, Allan Dafoe, Baobao Zhang, and Owain Evans. When will AI exceed human performance? Evidence from AI experts. *Journal of Artificial Intelligence Research*, 62:729–754, 2018. 1

[17] Herbert Grice. Logic and conversation. In *Speech acts*, pages 41–58. Brill, 1975. 4

[18] Eric Hansen, Daniel Bernstein, and Shlomo Zilberstein. Dynamic programming for partially observable stochastic games. In *AAAI*, volume 4, pages 709–715, 2004. 84

[19] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. *CoRR*, abs/1710.02298, 2017. 5, 65

[20] Geoffrey Hinton. Lecture 6.2 — A bag of tricks for mini batch gradient descent, 2017. https://youtu.be/iNucJB-0vYs?t=330. Accessed: 2021-08-31. 33

[21] Hengyuan Hu and Jakob Foerster. Simplified Action Decoder for Deep Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:1912.02288*, 2019. 5, 9, 65, 85

[22] Leslie Kaelbling, Michael Littman, and Anthony Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99 – 134, 1998. 83

[23] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014. 23

[24] Bo Klaasse. Condition-based maintenance policies using hidden Markov models, 2020. Master thesis, Eindhoven University of Technology. https://pure.tue.nl/ws/portalfiles/portal/172235497/Klaasse_B..pdf. 83

[25] Adam Laud. *Theory and application of reward shaping in reinforcement learning*. University of Illinois at Urbana-Champaign, 2004. PhD dissertation. https://core.ac.uk/download/pdf/4820036.pdf. Accessed: 2021-08-30. 46

[26] Adam Lerer, Hengyuan Hu, Jakob Foerster, and Noam Brown. Improving Policies via Search in Cooperative Partially Observable Games. *Proc. of the AAAI Conf. on Artificial Intelligence*, 34(05):7187–7194, April 2020. 4, 5, 6

[27] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *CoRR*, abs/1602.01783, 2016. 5

[28] Ranjit Nair, Milind Tambe, Makoto Yokoo, David Pynadath, and Stacy Marsella. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *IJCAI*, volume 3, pages 705–711, 2003. 84

[29] Chris Nota and Philip Thomas. Is the Policy Gradient a Gradient? *arXiv preprint arXiv:1906.07073*, 2019. 20

[30] Arthur O'Dwyer. Framework for writing bots that play Hanabi, 2018. https://github.com/Quuxplusone/Hanabi. Accessed: 2020-09-16. 4, 5

[31] Frans Oliehoek. *Decentralized POMDPs*, pages 471–503. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 85, 86

[32] Frans Oliehoek, Shimon Whiteson, and Matthijs Spaan. Lossless clustering of histories in decentralized POMDPs. In *Proc. of the* 8^{th} *Int. Conf. on Autonomous Agents and Multiagent Systems*, pages 577–584, 2009. 84

[33] Frans Oliehoek, Stefan Witwicki, and Leslie Kaelbling. A Sufficient Statistic for Influence in Structured Multiagent Environments. *arXiv preprint arXiv:1907.09278*, 2019. 83, 85

[34] Stéphane Ross and Drew Bagnell. Efficient Reductions for Imitation Learning. In *Proc. of the* 13^{th} *Int. Conf. on Artificial Intelligence and Statistics*, pages 661–668. JMLR Workshop and Conf. Proc., 2010. 73

[35] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv preprint arXiv:1506.02438*, 2015. 26

[36] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 27, 28

[37] David Silver. Lectures on Reinforcement Learning, 2015. https://www.davidsilver.uk/teaching/. Accessed: 2021-08-13. 12

[38] David Silver. Lectures on Reinforcement Learning, 2015. Lecture 7: Policy Gradient Methods. https://www.davidsilver.uk/wp-content/uploads/2020/03/pg.pdf. Accessed: 2021-08-13. 12

[39] Matthijs Spaan. Partially observable Markov decision processes. In *Reinforcement Learning*, pages 387–414. Springer, 2012. 82

[40] Richard Sutton. The Agent-Environment Interface, 2005. http://www.incompleteideas.net/book/ebook/node28.html. Accessed: 2021-08-12. 8

[41] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts London, England, 2018. 13, 14, 21, 26

[42] Mark van den Bergh. Hanabi, a cooperative game of fireworks, 2015. Bachelor thesis, Leiden University. http://www.math.leidenuniv.nl/scripties/BSC-vandenBergh.pdf. 3, 7, 8

[43] Ronald Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. 23

[44] David Wu. A rewrite of Hanabi-bot in Scala, 2018. https://github.com/lightvector/fireflower. Accessed: 2020-09-16. 4

[45] Jeff Wu. State of the art Hanabi bots + simulation framework in rust, 2018. https://github.com/WuTheFWasThat/hanabi.rs. Accessed: 2020-09-16. 4, 5

[46] Luke Zettlemoyer, Brian Milch, and Leslie Kaelbling. Multi-agent filtering with infinitely nested beliefs. *Advances in Neural Information Processing Systems*, 21:1905–1912, 2008. 84

Appendix A

Extra experiments

Next to the experiments described in Chapter 6, we have run multiple other experiments. We leave these out of the body of the paper since they do not answer the research questions posed in Section 1.3. Nevertheless, we would still like to present a few interesting ones here.

A.1 Showing playability

In Section 5.1 we described a very simple observation vector which showed directly to the agent which cards are playable. After also including the life and hint tokens in the vector this was our first agent to finally score some points. The state of Figure 5.1 would now be represented as:

$$[0, 1, 0, 1, 0,\ 0.625, 0.667].$$

The different settings that we tried in the experiment with this playability vector can be seen in Table A.1. All possible combinations of settings were tried, giving $4 \cdot 2 \cdot 2 = 16$ runs in total. In this experiment we were mostly interested in a discovering a well-working network architecture, activation function, and batch size. The reward for an illegal move, the discount factor, and the learning rate had been tweaked before, and were left on their best performing setting so far.

An example of a set of well working settings is shown in Figure A.1. The illegal moves quickly go down in the first few epochs. After a while, about 25000 epochs in this case, the algorithm

Table A.1: Settings of the experiments with the playability observation vector.

Setting	Chosen values
Algorithm	SPG
Representations	
State	7 (playability vector)
Action	11
Rewards	
Successfully played a card	+1
Lost all life tokens	−score
Illegal move	−10
All others	0
Network architecture	
Hidden layers	**[16]** / **[32]** / **[16, 16]** / **[32, 32]**
Activation function	**Tanh** / **ReLU**
Batch size	**1** / **500**
Discount factor	0.99
Learning rate	10^{-3}

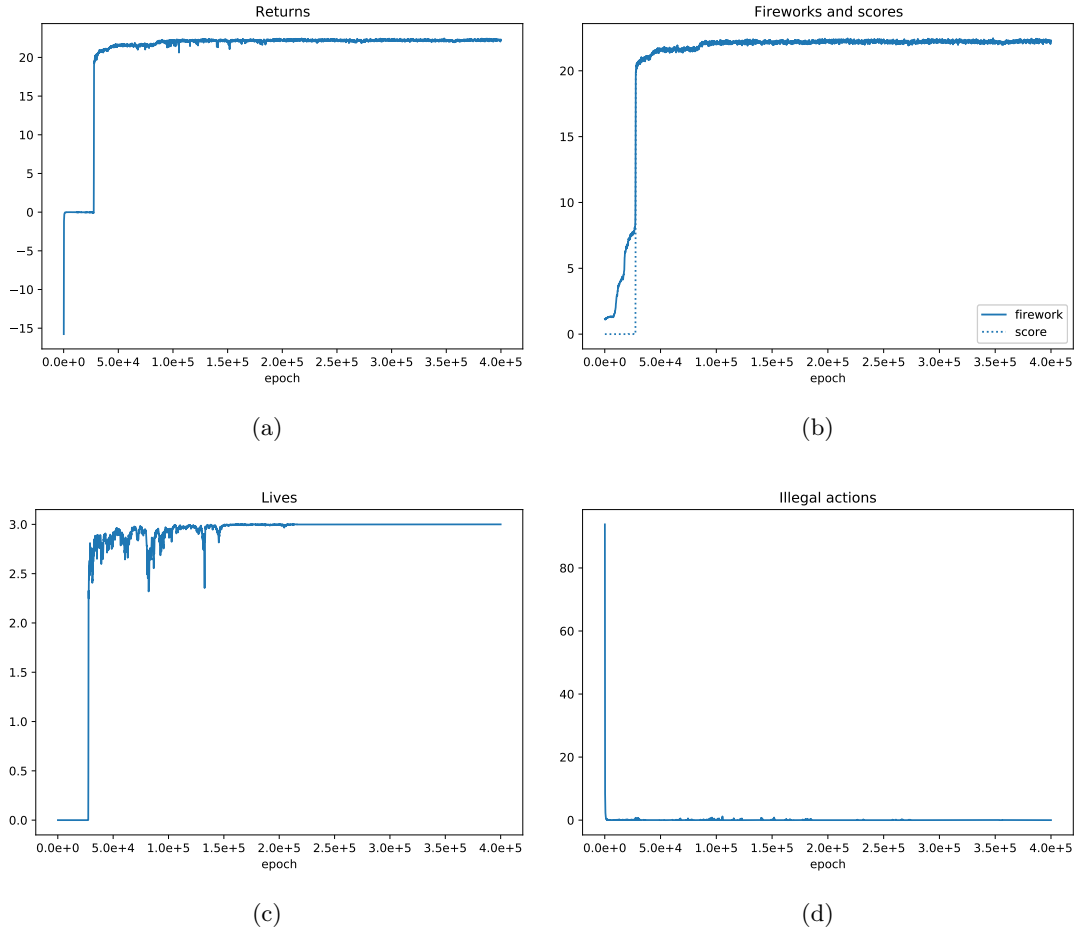
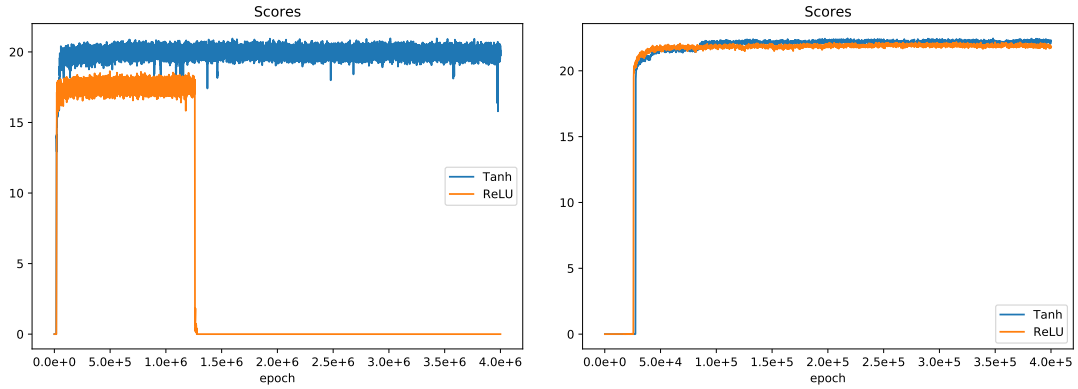


Figure A.1: The playability vector. After the agent learns to retain some life tokens, the scores quickly rise up towards about 22. Settings of the experiment:
Hidden layers: [32]
Activation: Tanh
Batch size: 500

learns to retain life tokens until the end of an episode, such that the score ends up above 0. That moment is quite a tipping point in the learning curve. After that, the score only improves very gradually towards the potential maximum of 25. Notice that before the tipping point the algorithm is already learning to play cards correctly, which we can see by the increasing fireworks.

When we decrease the batch size to 1 the SPG algorithm learns a bit less stable. The ReLU activation function can sometimes be nearly as good as Tanh, see Figure A.2b. However, when it is combined with a small batch size the algorithm becomes unstable, as shown in Figure A.2a. When the batch size is small, the variance in the collected experience is higher, which might lead to this behaviour.

The results show a preference for a bigger batch size and the hyperbolic tangent (Tanh) as the activation function. The best number of layers and width of the layers remained an open question, which we have looked into in the experiments of Section 6.4.



(a) Batch size: 1

(b) Batch size: 500

Figure A.2: Comparison of activation functions in a policy network with one hidden layer of size 32. ReLU and Tanh perform similarly on a larger batch size, but ReLU is unstable when only one episode is played per epoch.

A.2 Supervised learning on a rule-based policy

In this experiment we took a completely different approach. Instead of using a reinforcement learning algorithm, we applied supervised learning (SL) to the actions of our agent. This agent's goal is to imitate the examples provided by a rule-based (RB) teacher agent which we designed. Using SL to learn desired behaviour in this way is also called imitation learning [34].

The parameters of the network of the SL-agent are randomly initialized, so for any state s the outputted action selection probabilities are at first all pretty close to $\frac{1}{11}$, for example:

$$[0.085, 0.086, 0.080, 0.082, 0.105, 0.064, 0.103, 0.090, 0.088, 0.085, 0.132].$$

The RB-agent always outputs one action:

$$[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0].$$

We act as if the rule-based agent always makes the optimal move. Thus, the SL algorithm tries to tweak the network's parameters such that its output is closer to the output of the RB-agent. We measure this *closeness* by the mean squared error (MSE) loss function:

$$L(\boldsymbol{\theta}, s) = \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} \left(\pi_{\boldsymbol{\theta}}(a|s) - \widetilde{\pi}(a|s)\right)^2$$

where $\pi_{\boldsymbol{\theta}}$ is the SL-agent and $\widetilde{\pi}$ is the RB-agent. The number of actions to choose from, given by $|\mathcal{A}|$, is 11 in our case.

The objective of our SL algorithm is to minimize this loss function for all states:

$$L(\boldsymbol{\theta}) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} L(\boldsymbol{\theta}, s)$$

which we would like to do by gradient descent. However, since computing the gradient of $L(\boldsymbol{\theta})$ is intractable, it is often estimated by sampling a few states. In our case we just sample one state and use that to estimate the gradient:

$$\widehat{\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})} = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, s) = \frac{2}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} (\pi_{\boldsymbol{\theta}}(a|s) - \widetilde{\pi}(a|s)) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s).$$

Algorithm 4: Supervised Learning for Hanabi

input: a differentiable policy parameterization $\pi_{\boldsymbol{\theta_0}}$, a teacher policy $\widetilde{\pi}$, learning rate α
output: policy $\pi_{\boldsymbol{\theta_k}}$ which hopefully imitates $\widetilde{\pi}$

Initialize the environment for a new episode.
for *step* $k = 1, 2, \ldots$ **do**
  Select teacher action $a_k = \widetilde{\pi}(s_k)$.
  Compute student policy vector $\pi_{\boldsymbol{\theta_k}}(s_k)$.
  Estimate the gradient with the MSE loss:

$$\widehat{\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_k)} = \frac{2}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} (\pi_{\boldsymbol{\theta_k}}(a|s_k) - \widetilde{\pi}(a|s_k)) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta_k}}(a|s_k)$$

  Update parameters $\boldsymbol{\theta_k}$ via the gradient descent algorithm Adam with learning rate α.
  Act in the environment with the teacher action a_k.
  if *episode is finished* **then**
    Initialize the environment for a new episode.
  end
end

Table A.2: Settings of the supervised learning experiment.

Setting	Chosen values
Algorithm	Supervised Learning
Representations	
State	**37 / 62**
Action	11
Network architecture	
Hidden layers	**[32] / [64] / [32, 32] / [64, 64]**
Activation function	**Tanh / ReLU**
Learning rate	$\mathbf{10^{-2}/10^{-3}/10^{-4}/10^{-5}/10^{-6}}$

The SL algorithm we implemented is shown in Algorithm 4.

The settings that we tried out in this experiment are shown in Table A.2. We ran all possible combinations of settings, giving $2 \cdot 4 \cdot 2 \cdot 5 = 80$ runs in total. We will only present a few interesting results.

The SL algorithm could imitate the rule-based bot quite well and score around 21 points after just 15 hours of training. The loss often decreased nicely, especially for the well working learning rates 10^{-3} and 10^{-4}, as shown in Figure A.3.

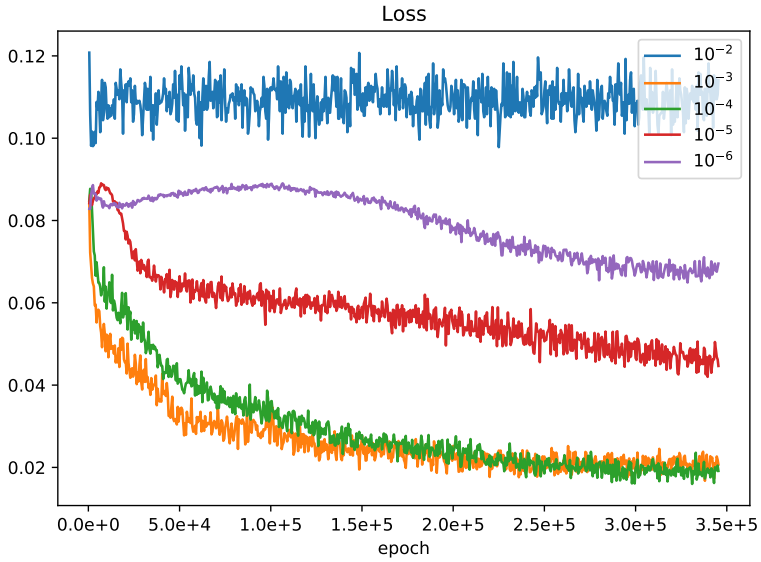


Figure A.3: Comparison of different learning rates. The loss decreases most quickly for 10^{-3} and 10^{-4}. Other settings at:
State representation: 62
Hidden layers: $[64, 64]$
Activation: Tanh

The RB-agent scores an average of about 24.1 points, see Figure A.4. This shows that it is probably not the optimal teacher agent, meaning that we cannot expect the SL-agent to perform optimal either.

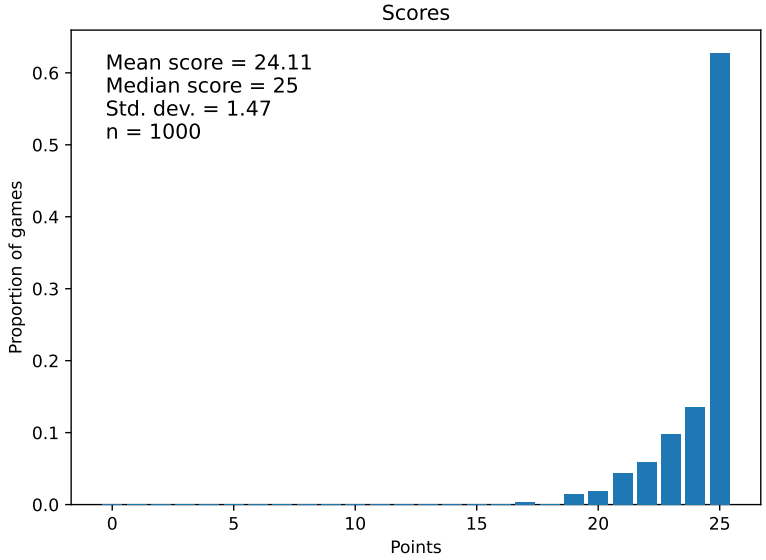


Figure A.4: Scores of the rule-based agent in 1000 test games. 62.8% of the games were perfect.

Continue with reinforcement learning

We wanted to see if the parameters of the policy network learned by the supervised learning algorithm could be improved by continuing with reinforcement learning (RL). We used the SPG algorithm with the same settings as the SL algorithm in Table A.2, but with additional RL settings shown in Table A.3. We only continued with learning rates 10^{-3} and 10^{-4}.

Table A.3: Additional settings of the SL experiment continued with SPG.

Setting	Chosen values
Algorithm	SPG
Rewards	
Successfully played a card	+1
Lost all life tokens	−score
Illegal move	−10
All others	0
Batch size	500
Discount factor	0.99

The results were decent. Policy networks that did not score so high yet climbed in performance, and the others continued to score high. Nevertheless, we notice some unstable performance for learning rate 10^{-3} in Figure A.5. The smaller learning rate 10^{-4} is perhaps more suitable to continue with.

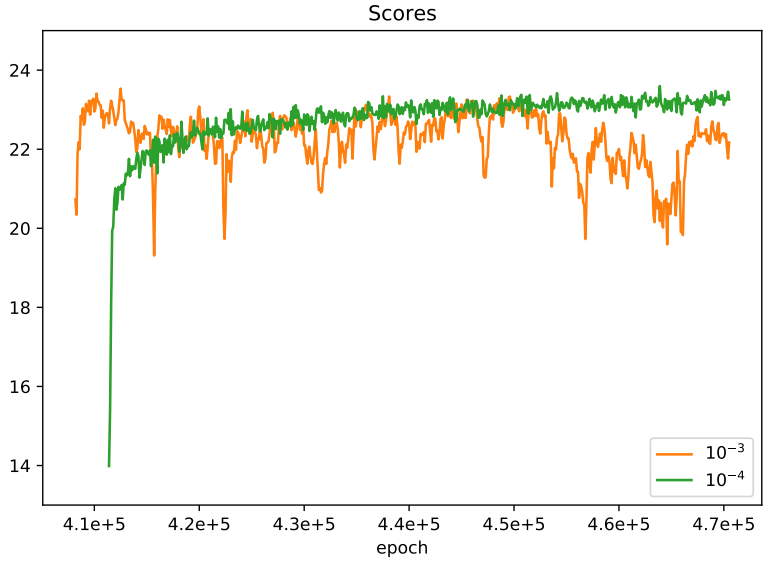


Figure A.5: Comparison of different learning rates with continued RL training. Even though the performance of 10^{-4} was much lower coming out of the SL experiment, the SPG algorithm quickly increases its score and keeps the performance stable. With a 10^{-3} learning rate SPG shakes up and down much more. Other settings are:
State representation: 62
Hidden layers: $[64, 64]$
Activation: Tanh

We tested the best agent of this experiment with 1000 games. See the results in Figure A.6.

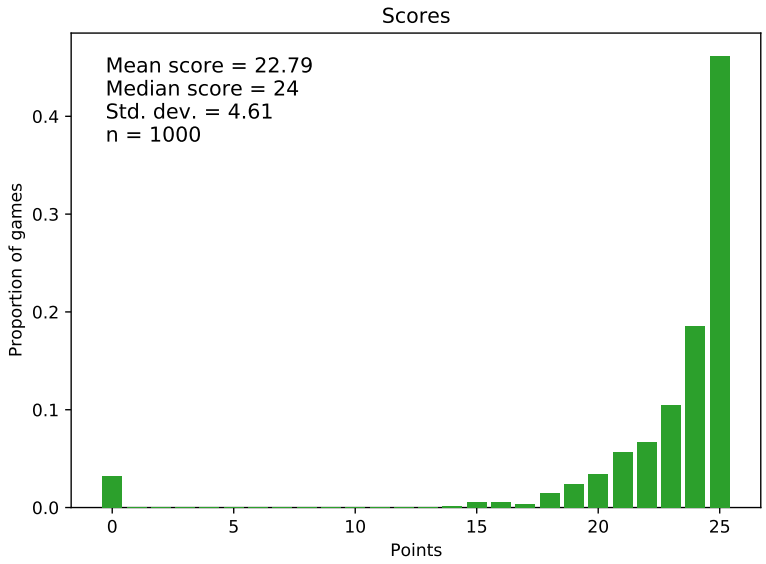


Figure A.6: Scores of the best agent in 1000 test games after continued training with RL. Almost half of the games were perfect, but some games ended with zero points. Settings:
State representation: 62
Hidden layers: $[64, 64]$
Activation: Tanh
Learning rate: 10^{-4}

A.3 More graphs

We present one graph in this section that belongs to the experiments of Chapter 6. It was not quite readable due to large fluctuations in performance.

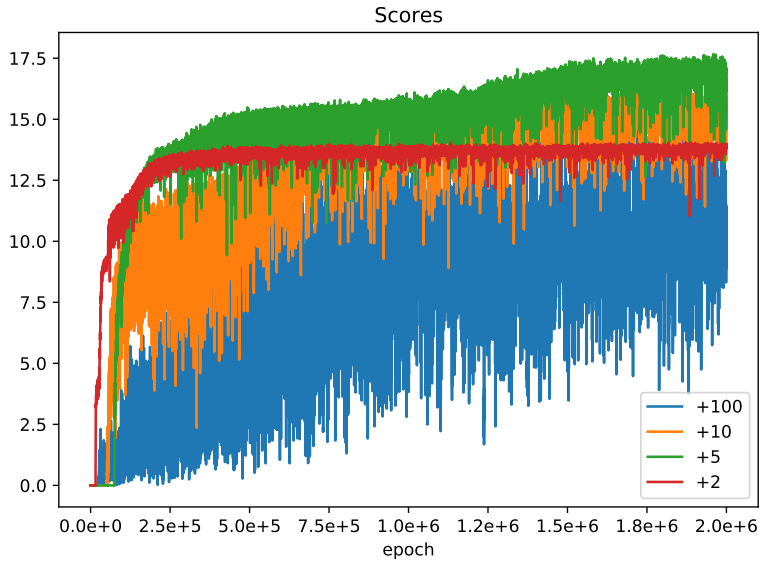


Figure A.7: Different rewards for a successful play, with SPG. Average score per 100 epochs. See Figure 6.12 in Section 6.3 for a more readable version of this graph.

A.4 More results

In this section we provide a large table with settings and average scores of experiments that we ran for a longer time (a few weeks). In the **epochs** column we show how many epochs the particular run has finished.

These experiments have some settings in common, which we leave out of the large table. These settings are shown in Table A.4. Also, none of these experiments use the epsilon approach for exploration (described in Section 6.2), as we discovered early on that this did not work. None of them shuffle the cards in the current player's hand either, even though one experiment is called *shuf_spg*. That run was one of the few control runs that did not shuffle the cards. The *shuf_spg* experiment is the only exception to Table A.4. It used the default setting for all reward shaping options.

The column i_π in the large table stands for the number of update iterations per epoch for the policy network π. The highest average score for each algorithm (SPG, VPG, PPO) is shown in bold. Note that these runs have not trained for an equal number of epochs. The names of the experiments correspond to the names used in our code repository, available at:
gitlab.tue.nl/jim-portegies/student-projects/bram-grooten

Table A.4: Settings which all experiments (except *shuf_spg*) of the large table below have in common.

Setting	Value	Value for *shuf_spg*
Representations		
Action	11	11
Rewards		
Lost all lives	−score	−score
Lost one life	−0.1	0
Hint	−0.02	0
Play	+0.02	0
Discard	0	0
Discard playable	−0.1	0
Discard useless	+0.1	0
Discard unique	−0.1	0
Network architecture		
Activation function π	Tanh	Tanh
Activation function V[1]	Tanh	-
Hyperparameters		
Batch size	1000	1000
Entropy coefficient	0.01	0.01
Renormalize	yes	yes
Update iterations V[2]	5	-
Discount factor	0.99	0.99
Clipping parameter[3]	0.2	-

[1]Only for VPG and PPO. SPG does not have a value network.
[2]Only for VPG and PPO.
[3]Only for PPO.

name	**id**	**epochs**	**score**	**suc. play**	**ill. act.**	**l.r. π**	**hid. layers π**	**i_π**	**l.r. V**	**hid. layers V**	**adv. type**	**λ**	**obs. vec.**
shuf_spg	5	3212900	**19,10**	+1	−0.5	0.001	[64, 64]	1	-	-	-	-	62
vpg_fast	0	2935850	18,32	+1	−0.5	0.001	[64, 64]	1	0.001	[64, 64]	basic	-	136
vpg_gae	0	2673400	20,15	+1	−0.5	0.001	[64, 64]	1	0.001	[64, 64]	basic	-	136
vpg_gae	1	2649350	17,90	+1	−0.5	0.001	[64, 64]	1	0.001	[64, 64]	gae	0.95	136
vpg_gae	2	2598800	16,97	+1	−0.5	0.001	[64, 64]	1	0.001	[64, 64]	gae	1.0	136
ppo_gae	0	2704900	9,35	+1	−0.5	0.001	[64, 64]	5	0.001	[64, 64]	no	-	136
ppo_gae	1	2644200	12,11	+1	−0.5	0.001	[64, 64]	5	0.001	[64, 64]	basic	-	136
ppo_gae	2	2605200	12,71	+1	−0.5	0.001	[64, 64]	5	0.001	[64, 64]	gae	0.95	136
vpg_iters	0	2513500	10,73	+1	−0.5	0.001	[64, 64]	2	0.001	[64, 64]	basic	-	136
vpg_iters	1	2455800	16,75	+1	−0.5	0.001	[64, 64]	2	0.001	[64, 64]	gae	0.95	136
vpg_iters	2	2536700	17,32	+1	−0.5	0.0001	[64, 64]	2	0.001	[64, 64]	basic	-	136
vpg_iters	3	2481000	15,33	+1	−0.5	0.0001	[64, 64]	2	0.001	[64, 64]	gae	0.95	136
vpg_iters	4	2509400	20,60	+1	−0.5	0.0001	[64, 64]	5	0.001	[64, 64]	basic	-	136
vpg_iters	5	2508600	20,40	+1	−0.5	0.0001	[64, 64]	5	0.001	[64, 64]	gae	0.95	136
rew_spg	0	2760800	13,79	+2	−1	0.001	[64, 64]	1	-	-	-	-	136
rew_spg	1	2656500	17,83	+5	−1	0.001	[64, 64]	1	-	-	-	-	136
rew_spg	2	2651800	15,43	+10	−1	0.001	[64, 64]	1	-	-	-	-	136
rew_spg	3	2741600	11,75	+100	−1	0.001	[64, 64]	1	-	-	-	-	136
rew_vpg	0	2296300	20,58	+2	−1	0.001	[64, 64]	1	0.001	[64, 64]	gae	0.95	136
rew_vpg	1	2267400	18,87	+5	−1	0.001	[64, 64]	1	0.001	[64, 64]	gae	0.95	136
rew_vpg	2	2284200	19,75	+10	−1	0.001	[64, 64]	1	0.001	[64, 64]	gae	0.95	136
rew_vpg	3	2308100	20,94	+100	−1	0.001	[64, 64]	1	0.001	[64, 64]	gae	0.95	136
rew_ppo	0	2398100	15,85	+2	−1	0.001	[64, 64]	5	0.001	[64, 64]	gae	0.95	136
rew_ppo	1	2405900	16,94	+5	−1	0.001	[64, 64]	5	0.001	[64, 64]	gae	0.95	136
rew_ppo	2	2426600	9,62	+10	−1	0.001	[64, 64]	5	0.001	[64, 64]	gae	0.95	136
rew_ppo	3	2621600	17,22	+100	−1	0.001	[64, 64]	5	0.001	[64, 64]	gae	0.95	136
combi_spg	0	2122900	15,53	+1	−1	0.0003	[128, 128, 64]	1	-	-	-	-	136
combi_spg	1	2109300	9,91	+1	−1	0.0003	[128, 128, 64]	1	-	-	-	-	186
combi_spg	2	2083500	17,46	+3	−1	0.0003	[128, 128, 64]	1	-	-	-	-	136
combi_spg	3	2069900	18,37	+3	−1	0.0003	[128, 128, 64]	1	-	-	-	-	186
combi_spg	4	2105700	17,47	+10	−1	0.0003	[128, 128, 64]	1	-	-	-	-	136
combi_spg	5	2095600	17,78	+10	−1	0.0003	[128, 128, 64]	1	-	-	-	-	186
combi_spg	6	2209600	17,34	+100	−1	0.0003	[128, 128, 64]	1	-	-	-	-	136
combi_spg	7	2196500	16,14	+100	−1	0.0003	[128, 128, 64]	1	-	-	-	-	186

name	**id**	**epochs**	**score**	**suc. play**	**ill. act.**	**l.r. π**	**hid. layers π**	**i_π**	**l.r. V**	**hid. layers V**	**adv. type**	**λ**	**obs. vec.**
combi_vpg	0	2092900	21,54	+1	−1	0.0003	[128, 128, 64]	1	0.0003	[128, 64, 32]	gae	0.95	136
combi_vpg	1	2038100	22,39	+1	−1	0.0003	[128, 128, 64]	1	0.0003	[128, 64, 32]	gae	0.95	186
combi_vpg	2	2110500	24,12	+3	−1	0.0003	[128, 128, 64]	1	0.0003	[128, 64, 32]	gae	0.95	136
combi_vpg	3	2052400	23,28	+3	−1	0.0003	[128, 128, 64]	1	0.0003	[128, 64, 32]	gae	0.95	186
combi_vpg	4	2113300	**24,22**	+10	−1	0.0003	[128, 128, 64]	1	0.0003	[128, 64, 32]	gae	0.95	136
combi_vpg	5	2059700	23,12	+10	−1	0.0003	[128, 128, 64]	1	0.0003	[128, 64, 32]	gae	0.95	186
combi_vpg	6	3880800	15,38	+100	−1	0.0003	[128, 128, 64]	1	0.0003	[128, 64, 32]	gae	0.95	136
combi_vpg	7	3838000	14,80	+100	−1	0.0003	[128, 128, 64]	1	0.0003	[128, 64, 32]	gae	0.95	186
combi_ppo	0	1990800	18,03	+1	−1	0.0003	[128, 128, 64]	5	0.0003	[128, 64, 32]	gae	0.95	136
combi_ppo	1	2024800	**20,50**	+1	−1	0.0003	[128, 128, 64]	5	0.0003	[128, 64, 32]	gae	0.95	186
combi_ppo	2	2038300	18,95	+3	−1	0.0003	[128, 128, 64]	5	0.0003	[128, 64, 32]	gae	0.95	136
combi_ppo	3	2040200	20,06	+3	−1	0.0003	[128, 128, 64]	5	0.0003	[128, 64, 32]	gae	0.95	186
combi_ppo	4	2040200	18,78	+10	−1	0.0003	[128, 128, 64]	5	0.0003	[128, 64, 32]	gae	0.95	136
combi_ppo	5	2041800	19,26	+10	−1	0.0003	[128, 128, 64]	5	0.0003	[128, 64, 32]	gae	0.95	186
combi_ppo	6	3219600	12,61	+100	−1	0.0003	[128, 128, 64]	5	0.0003	[128, 64, 32]	gae	0.95	136
combi_ppo	7	2993500	5,86	+100	−1	0.0003	[128, 128, 64]	5	0.0003	[128, 64, 32]	gae	0.95	186
big_spg	0	989900	15,12	+10	−1	0.0003	[3 * 128, 64]	1	-	-	-	-	136
big_spg	1	1008900	14,49	+10	−1	0.0003	[3 * 128, 64]	1	-	-	-	-	186
big_spg	2	957300	14,09	+10	−1	0.0003	[4 * 128, 64]	1	-	-	-	-	136
big_spg	3	959100	12,26	+10	−1	0.0003	[4 * 128, 64]	1	-	-	-	-	186
big_spg	4	906700	12,04	+10	−1	0.0003	[5 * 128, 64]	1	-	-	-	-	136
big_spg	5	916600	11,87	+10	−1	0.0003	[5 * 128, 64]	1	-	-	-	-	186
big_vpg	0	1028200	21,95	+10	−1	0.0003	[3 * 128, 64]	1	0.0003	[128, 128, 64, 32]	gae	0.95	136
big_vpg	1	940700	22,50	+10	−1	0.0003	[3 * 128, 64]	1	0.0003	[128, 128, 64, 32]	gae	0.95	186
big_vpg	2	900400	18,87	+10	−1	0.0003	[4 * 128, 64]	1	0.0003	[3 * 128, 64, 32]	gae	0.95	136
big_vpg	3	885800	19,37	+10	−1	0.0003	[4 * 128, 64]	1	0.0003	[3 * 128, 64, 32]	gae	0.95	186
big_vpg	4	853500	15,74	+10	−1	0.0003	[5 * 128, 64]	1	0.0003	[4 * 128, 64, 32]	gae	0.95	136
big_vpg	5	843200	16,61	+10	−1	0.0003	[5 * 128, 64]	1	0.0003	[4 * 128, 64, 32]	gae	0.95	186
big_ppo	0	931400	19,43	+10	−1	0.0003	[3 * 128, 64]	5	0.0003	[128, 128, 64, 32]	gae	0.95	136
big_ppo	1	928300	16,11	+10	−1	0.0003	[3 * 128, 64]	5	0.0003	[128, 128, 64, 32]	gae	0.95	186
big_ppo	2	898200	17,18	+10	−1	0.0003	[4 * 128, 64]	5	0.0003	[3 * 128, 64, 32]	gae	0.95	136
big_ppo	3	882100	15,60	+10	−1	0.0003	[4 * 128, 64]	5	0.0003	[3 * 128, 64, 32]	gae	0.95	186
big_ppo	4	843800	16,26	+10	−1	0.0003	[5 * 128, 64]	5	0.0003	[4 * 128, 64, 32]	gae	0.95	136
big_ppo	5	836400	13,36	+10	−1	0.0003	[5 * 128, 64]	5	0.0003	[4 * 128, 64, 32]	gae	0.95	186

Appendix B

Extra ideas

Throughout our research we have come up with many different ideas. We would like to share a few promising ones that did not make it into the main body of the report here, to give future researchers a head start. Section B.1 extends the views of Section 3.2 providing possible directions to attack the challenges of imperfect information. In Section B.2 we list a few extra ideas that we came up with to stimulate exploration.

B.1 Dealing with partial observability

In the methods described in Chapters 3 and 4, the current state s_t is used as input for the policy and the value function. This is only possible when the agent is able to fully observe the state of the environment at every time step. In the full version of Hanabi, players can only partially observe the current state, so we need a different approach. Let us start by exploring single-agent partially observable environments, after which we will extend to multi-agent problems, like Hanabi.

B.1.1 Single-agent partial observability

Partially observable reinforcement learning problems with a single agent are usually modelled as a POMDP, which stands for *partially observable Markov decision process*. In this model the underlying environment is still Markovian, meaning that the state transitions and rewards only depend on the previous state and action. But the observations that the agent receives are not a Markovian signal anymore, meaning that the probability of a certain next observation given only the current observation, is not equal to the probability of that next observation given all previous observations. A direct mapping from observations to actions is not sufficient for optimal behaviour [39]. Memorizing previous observations and actions, in some concise manner, is therefore necessary.

If the problem is small, meaning there are only a couple of possible observations and actions in each time step, then agents might be able to store their full action-observation history (AOH). The AOH up to the current time step is given by: $\tau_t = (o_0, a_0, o_1, a_1, \ldots, o_{t-1}, a_{t-1}, o_t)$, where o_t denotes the agent's observation at time step t. The AOH τ_t comes from the set of all possible AOHs at time t, denoted by $\mathcal{T}_t$.

However, even in a small problem the size of $\mathcal{T}_t$ grows exponentially as the length of an episode progresses: $|\mathcal{T}_t| = (|A| \times |O|)^t$. Summarizing the AOH into something more compact would be helpful. A solution that is often used for this, is defining a *belief* on the current state.

A belief $\boldsymbol{b}$ in a POMDP is a probability distribution over the state space $\mathcal{S}$ given the current AOH:

$$\boldsymbol{b} : \mathcal{T}_t \to \mathcal{P}(\mathcal{S}) \qquad \text{with} \qquad \boldsymbol{b}(\tau_t) = \begin{bmatrix} \mathbb{P}(S_t = s^{(1)}|\tau_t) \\ \mathbb{P}(S_t = s^{(2)}|\tau_t) \\ \vdots \\ \mathbb{P}(S_t = s^{(|\mathcal{S}|)}|\tau_t) \end{bmatrix}.$$

The notation: $b(s_t) = \mathbb{P}(S_t = s_t|\tau_t)$ is used to denote the belief in a specific state s_t. A policy can now be based on the beliefs $\pi : \mathcal{P}(\mathcal{S}) \to A$. When we include the possible stochasticity of the policy we get:

$$\pi : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(A).$$

When we examine the beliefs closely, it does not look like we have made any progress. The beliefs still depend on the full action-observation history τ_t, meaning an agent must remember all observations and actions. Fortunately, there is a useful lemma that eliminates this problem.

In a POMDP, the current belief b_t only depends on the previous belief b_{t-1} and the new action-observation pair (a_{t-1}, o_t), as shown in Lemma 7. This means that keeping track of the belief over the hidden environment state is a sufficient statistic for optimal decision making [22]. Oliehoek et al. [33] explain it clearly: this belief is a lossless summary of the action-observation history, as "it allows the agent to reach the same performance as an agent that would act optimally based on the AOH". If the observation function Ω and transition function T of the POMDP are known, then this belief update can be computed.

Lemma 7 (Belief in a POMDP, cf. [22, sec. 3.3])**.** *Let $b_t(s) = \mathbb{P}(S_t = s|\tau_t)$ be the probability of being in state s given the full action-observation history τ_t at time step t. Then we have:*

$$b_t(s_t) \;\propto\; \Omega(o_t \,|\, a_{t-1}, s_t) \sum_{s_{t-1} \in \mathcal{S}} T(s_t \,|\, s_{t-1}, a_{t-1})\; b_{t-1}(s_{t-1}).$$

Proof. We follow the proof shown in [24] in a similar manner.

$$\begin{aligned}
b_t(s_t) &= \mathbb{P}(S_t = s_t \,|\, \tau_t) \\
&= \mathbb{P}(S_t = s_t \,|\, \tau_{t-1}, A_{t-1} = a_{t-1}, O_t = o_t) \\
&= \frac{\mathbb{P}(O_t = o_t \,|\, \tau_{t-1}, A_{t-1} = a_{t-1}, S_t = s_t)\; \mathbb{P}(S_t = s_t \,|\, \tau_{t-1}, A_{t-1} = a_{t-1})}{\mathbb{P}(O_t = o_t \,|\, \tau_{t-1}, A_{t-1} = a_{t-1})} \quad \text{(Bayes' theorem)} \\
&= \frac{\mathbb{P}(O_t = o_t \,|\, A_{t-1} = a_{t-1}, S_t = s_t) \sum_{s_{t-1} \in \mathcal{S}} \Big(\mathbb{P}(S_t = s_t \,|\, A_{t-1} = a_{t-1}, S_{t-1} = s_{t-1}) \cdot \mathbb{P}(S_{t-1} = s_{t-1} \,|\, \tau_{t-1}) \Big)}{\mathbb{P}(O_t = o_t \,|\, \tau_{t-1}, A_{t-1} = a_{t-1})} \\
&= \frac{\Omega(o_t \,|\, a_{t-1}, s_t) \sum_{s_{t-1} \in \mathcal{S}} T(s_t \,|\, s_{t-1}, a_{t-1})\; b_{t-1}(s_{t-1})}{\mathbb{P}(O_t = o_t \,|\, \tau_{t-1}, A_{t-1} = a_{t-1})} \\
&\propto\; \Omega(o_t \,|\, a_{t-1}, s_t) \sum_{s_{t-1} \in \mathcal{S}} T(s_t \,|\, s_{t-1}, a_{t-1})\; b_{t-1}(s_{t-1}).
\end{aligned}$$

□

Notice that in the last step of the proof the denominator was left out. Since we know that $\sum_{s \in \mathcal{S}} b(s) = 1$ (the individual beliefs over every state must add up to one), we do not need this

normalizing factor to compute the new beliefs. Thus, we can calculate it using only the previous belief and the new action-observation pair.

To show how one could calculate this, we first estimate the new beliefs for all states s_t using the expression from Lemma 7:

$$\forall\, s_t \in S \;:\; \widetilde{b}_t(s_t) = \Omega(o_t \,|\, a_{t-1}, s_t) \sum_{s_{t-1} \in \mathcal{S}} T(s_t \,|\, s_{t-1}, a_{t-1})\; b_{t-1}(s_{t-1})$$

after which we normalize these values to end up with a proper probability distribution:

$$\forall\, s_t \in S \;:\; b_t(s_t) = \frac{\widetilde{b}_t(s_t)}{\sum_{s_t \in \mathcal{S}} \widetilde{b}_t(s_t)}.$$

In a large state space $\mathcal{S}$ this is intractable. However, for a single-agent version of Hanabi[1] it might be doable since most of the states can be ruled out based on the observation. The agent's own hand is the only part of the state which remains hidden, giving at most $9,095,150$ possible states as shown in Section 7.3. When playing with 4 or 5 players, the hand size shrinks to four, leaving only $372,600$ states. This is still quite a large number, but it can decrease further if hints are given to the agent.

B.1.2 Multi-agent partial observability

Unfortunately, this useful property of the beliefs cannot easily be carried over to the multi-agent extension of the POMDP, which is the Dec-POMDP. A belief only on the state of the environment is not a sufficient statistic for optimal behaviour anymore [32], as an agent must also predict what actions the other agents are going to take. In this setting it is not obvious what the beliefs should be over, so we will discuss some options.

Since there are multiple agents, we now have a separate action-observation history τ_t^i for each agent $i \in \mathcal{I}$. Some researchers have used all these AOHs directly. Nair et al. [28] built dynamic programming algorithms based on beliefs over the entire observation history of all other agents. Oliehoek et al. [32] try to improve the performance by clustering pairs of histories together if the resulting joint beliefs are identical.[2] As the size of the histories grows quickly, these solutions are only applicable to very small problems. Even then, they can only run for a couple of time steps before running out of memory. For example, in the Dec-Tiger problem they reached a horizon of up to 7 time steps.

Zettlemoyer et al. [46] work with (infinitely) nested beliefs. These are beliefs based on different levels of inference. The zeroth level belief for agent i at time step t, written $b_t^{i,0}$, is the same as the single-agent belief shown in Section B.1.1. The first level belief captures a probability on the current state (just like the zeroth level) but also keeps track of the zeroth level beliefs of other agents: $b_t^{i,1} = \mathbb{P}(s_t, b_t^{-i,0} \,|\, \tau_t^i)$, where the minus notation is used to denote all agents $j \neq i$. This continues on higher levels, where the n^{th} level belief depends on the $(n-1)^{th}$ level beliefs of other agents: $b_t^{i,n} = \mathbb{P}(s_t, b_t^{-i,n-1} \,|\, \tau_t^i)$. When extended infinitely, written as $b_t^{i,*}$ it is a sufficient statistic to predict the next infinitely nested belief $b_{t+1}^{i,*}$. However, it is not necessarily a sufficient statistic for optimal behaviour in the sense that it can replace the full AOHs of all agents.

There have been found sufficient statistics for acting optimally, such as the "multi-agent belief" by Hansen et al. [18]. They have shown that a belief specified over states and future policies of other agents is in fact a sufficient statistic in Dec-POMDPs. However, keeping track of the possible future policies of all other agents is computationally infeasible in the case of Hanabi, as

[1] This can perhaps be set up by considering the other agents as part of the environment.

[2] A joint belief on a particular state s is the probability of being in that state, given the AOHs of all agents: $\mathbb{P}(S_t = s \,|\, \tau_t^1, \ldots, \tau_t^{|\mathcal{I}|})$.

the number of joint policies in a Dec-POMDP [31] is

$$O\left(|A^*|^{\frac{|\mathcal{I}|(|\mathcal{O}^*|^\eta - 1)}{|\mathcal{O}^*| - 1}}\right)$$

where $\mathcal{I}$ is the set of agents, $\mathcal{A}^*$ and $\mathcal{O}^*$ are the largest individual action and observation sets, and η is the planning horizon.

Oliehoek et al. [33] look at the *influence* of certain factors of the state, if a state is able to be factorized. They proceed by working only with the relevant factors of a state. In the spirit of this idea, we propose a "partial belief" as an approximation, with factors that we think are the most relevant for an agent in Hanabi. Recall that the current state of a Hanabi game is defined as $s_t = (\boldsymbol{f}, \boldsymbol{h}, Z, m, l, d, i)$ in Section 3.2.1. Of this state, an agent can observe everything except its own hand $h_j \in \boldsymbol{h}$.

The partial belief of agent j includes:

1. the agent's own hand of cards: h_j,
2. the belief of other agents on their own hand,
3. the next action of other agents.

The number of possible hands is quite large, as shown in Section 7.3. Thus, it might be beneficial to factorize this part of the state even further, into the possible colors and ranks of the separate cards in an agent's hand. For example, the specific belief about the color of the first card is represented as follows:

$$b(\text{color}(c_1)) = \begin{bmatrix} \mathbb{P}(\text{color}(c_1) = R \mid \tau_t^i) \\ \mathbb{P}(\text{color}(c_1) = G \mid \tau_t^i) \\ \mathbb{P}(\text{color}(c_1) = B \mid \tau_t^i) \\ \mathbb{P}(\text{color}(c_1) = Y \mid \tau_t^i) \\ \mathbb{P}(\text{color}(c_1) = W \mid \tau_t^i) \end{bmatrix}$$

which could be represented as Figure B.1 for example. Whenever a (color related) hint is given to the agent, this distribution shifts. Working with beliefs over the beliefs of other agents is often called Theory of Mind. It is used by the ToM-Hanabi agents developed by Fuchs et al. [15].

Dec-POMDP

We will describe a Dec-POMDP framework [31] which can be used to research the full version of Hanabi. Hu and Foerster also assume this setup for Hanabi in their paper on the Simplified Action Decoder [21]. Our Dec-POMDP is a 9-tuple representing the environment of Hanabi: $E = \langle \mathcal{I}, \mathcal{S}, \boldsymbol{\mathcal{A}}, T, R, \boldsymbol{\mathcal{O}}, \Omega, \eta, \rho_0 \rangle$. We describe the elements below.

1. $\mathcal{I} = \{1, \ldots, p\}$ is the set of p agents.
2. $\mathcal{S}$ is the finite set of states in which the environment can be. The state $s_t \in \mathcal{S}$ captures the full configuration of the game at time step t and is the same as the state defined for the MDP in Section 3.2.1.
3. $\boldsymbol{\mathcal{A}} = \times_{i \in \mathcal{I}} \mathcal{A}^i$ is the finite set of joint actions. At every time step a joint action $\boldsymbol{a} = (a^1, \ldots, a^N)$ is taken. In our case, this contains "pass" actions for all agents, except for the agent who's turn it is: only a^{i_t} is a regular Hanabi action.

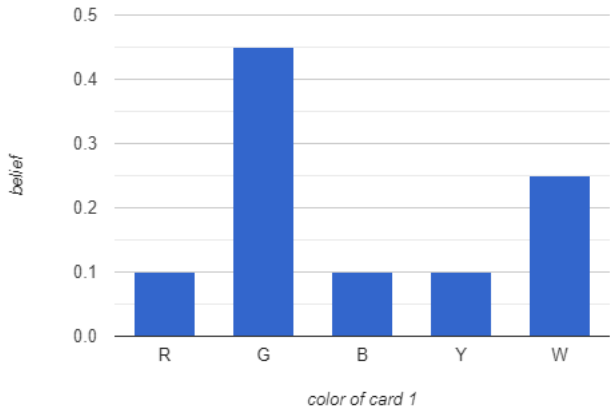


Figure B.1: An example of a belief about the color of one specific card.

4. $T : \mathcal{S} \times \boldsymbol{\mathcal{A}} \to \mathcal{P}(\mathcal{S})$ is the transition function, which is stochastic since we model the deck as part of this function. The probabilities depend on the cards that are still in the deck. Because the deck is shuffled, we assume a uniform distribution over the cards that are left.

5. $R : \mathcal{S} \times \boldsymbol{\mathcal{A}} \to \mathcal{R}$ is the immediate reward function, where $\mathcal{R}$ is the set of possible immediate rewards. The rewards are identical for all agents.[3] We choose the rewards to be the change in score of the Hanabi game. This gives $\mathcal{R} = \{+1, 0, -1, -2, \ldots, -24\}$, since an action that results in the loss of the third life token makes the cumulative score go down to zero.

6. $\boldsymbol{\mathcal{O}} = \times_{i \in \mathcal{I}} \mathcal{O}^i$ is the finite set of joint observations. Any agent i cannot see the full state s_t, but only receives the private observation $o_t^i \in \mathcal{O}^i$. All observations made during one time step form the joint observation: $\boldsymbol{o}_t = (o_t^1, \ldots, o_t^N) \in \boldsymbol{\mathcal{O}}$.

7. $\Omega : \mathcal{S} \times \boldsymbol{\mathcal{A}} \to \boldsymbol{\mathcal{O}}$ is the deterministic observation function, which generates the observations. In Hanabi, agents are able to observe the received rewards and the actions of all other agents.

8. η is the horizon of the problem. Since the length of a Hanabi episode is not fixed, we define our horizon to be infinite. We extend the state space $\mathcal{S}$ with one more state s_{END}, which is an absorbing state with reward 0 in every time step, where all episodes reside once the game is finished.

9. $\rho_0 \in \mathcal{P}(\mathcal{S})$ is the initial state distribution at time step $t = 0$. Thus $\rho_0(s)$ gives the probability of starting in state s (which is only non-zero for states corresponding to initial configurations of Hanabi). We assume the cards are shuffled and the players receive the correct number of cards, as described in the rules of Hanabi in Section 1.1.

[3] Otherwise we would be in the setting of a POSG (partially observable stochastic game), which is a generalization of Dec-POMDPs [31].

B.2 Stimulating exploration

In Section 6.2 we described three approaches to stimulate exploration during training. These main ideas were: changing the policy with an exploration term (ϵ), adding a entropy term to the objective, and shuffling the cards in the current player's hand. Next to these, we devised a couple more ideas, which we have not researched extensively. They are briefly described below.

More life tokens

In order to give the agent more opportunity to learn about playing cards, we changed the environment in this experiment to have many more life tokens. In this way the agent can keep trying to play cards, even though it fails many times. One could even let the number of life tokens decay after a while to converge towards the 3 life tokens used in the rules of Hanabi.

The results in our experiments with this were slightly promising. One reason we stopped using this approach however, was the fact that it is difficult to justly compare the performance of runs with different numbers of life tokens.

Different action representation

The purpose of this experiment was to try out a different representation of the action space. Instead of the 11 possible actions used before (discard index 0-4, play index 0-4, give a random hint[4]), we now have 51 possible actions. They are given by the 25 unique cards that exist. Each unique card has a corresponding discard and play action node, and there is still one extra node for hinting.

The settings in this experiment did not perform well at all. A reason for this could be the fact that there are many illegal actions in this case (all the unique cards that the current player does not have in her hand).

When switching to the full version of Hanabi one would need to devise a different action representation than ours anyway, since it will be necessary to separate the different hint possibilities.

Regularization

In this experiment we added two regularization techniques in an effort to slow down the speed at which our policies seemed to converge towards deterministic behaviour. The methods we applied were dropout and weight decay. We used dropout rates of 0.5, 0.2, 0 and a weight decay parameter of $0.01, 0.001, 0$. Unfortunately they did not improve the performance. After plotting the actual values of the weights in our networks it turned out they were not really growing unreasonably large in the first place.

[4] In simplified Hanabi it does not really matter which hint you give. One could perhaps communicate about which card should be played next, but we did not use this.