CHALMERS
UNIVERSITY OF TECHNOLOGY

Comparison of Artificial Intelligence Algorithms for Pokémon Battles

Master's thesis in Complex Adaptive Systems

LINUS NORSTRÖM

Department of Space, Earth and Environment
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis 2019

Comparison of Artificial Intelligence Algorithms for Pokémon Battles

LINUS NORSTRÖM

Department of Space, Earth and Environment
Division of Physical Resource Theory
Chalmers University of Technology
Gothenburg, Sweden 2019

Comparison of Artificial Intelligence Algorithms for Pokémon Battles
LINUS NORSTRÖM

Supervisor: Kristian Lindgren, Department of Space, Earth and Environment
Examiner: Kristian Lindgren, Department of Space, Earth and Environment

Master's Thesis 2019
Department of Space, Earth and Environment
Division of Physical Resource Theory
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Comparison of Artificial Intelligence Algorithms for Pokémon Battles
LINUS NORSTRÖM
Department of Space, Earth and Environment
Chalmers University of Technology

Abstract

This thesis aims to compare various artificial intelligence algorithms in three scenarios of Pokémon battles increasing in complexity. The highest performing algorithm was used in a challenge against human players, with the goal of at least playing on human level.

The most successful algorithm was a *Monte Carlo search algorithm* which spend time, before a round, playing out the entire game from that round many times. Another successful algorithm was *linear combination of features*, which spend time before the battle playing many training games in order to adjust weights correlating to actions and the game state. In addition, the *random algorithm* which always picks a random action and the *hard coded algorithm* which picks a pre-defined action each round was used as benchmark algorithms for the different scenarios.

The first scenario is not even a Pokémon battle, but rather an abstraction of a Pokémon battle. It is simple enough to test the various algorithms in a game, but resembles a Pokémon battle at its core closely enough to be relevant. The optimal action sequence can be found analytically, and it is therefore used as a benchmark. The Monte Carlo search algorithm manages to use the optimal action sequence 83.8% of the time, while the linear combination of features always plays optimally.

The second scenario is based on the first battle the player encounters when playing the first generation Pokémon games on the Game Boy. The complexity is increased from the first game as stochasticity is introduced and the game is unbalanced. The highest performing action sequence is found through a parameter sweep, and the hard coded algorithm using it achieves the highest performance. Both the Monte Carlo search algorithm and the linear combination of features algorithm has a similar, but not quite as high, performance.

The third scenario is a full sized Pokémon battle, with both players using a pre-defined team of popular Pokémon used in competitive play. This scenario is seemingly too complex for the linear combination of features algorithm, which can not achieve high performance. The Monte Carlo search algorithm, on the other hand, achieves the highest performance of all algorithms, and was therefore chosen to challenge human players. Against human opponents, it managed to win half of all battles. Humans reporting higher previous experience with Pokémon battles seem to win more often.

A general conclusion is that the complexity of the game and the algorithm must match for optimal performance. Another very exciting conclusion is that the goal was reached, since the most successful algorithm won about half of all games when playing against human players.

Keywords: artificial, intelligence, reinforcement, learning, algorithm, Monte, Carlo, linear, combination, Pokémon.

Acknowledgements

I would like to express my deep gratitude to Professor Kristian Lindgren, for accepting and believing in my rather strange master thesis proposal, and for his guidance during the thesis work. I would also like to thank André Stén, for both his opposition at the presentation and moral support. The goal of this master thesis would not have been reached if not for all participants in the challenge: Lef Filippakis, Fredrik Hallhagen, Anders Hansson, Rikard Helgegren, Richard Hoorn, Carl Jendle, Yvonne Kay, Eric Lindgren, Gustav Lindwall, Anton Lord, Victor Nilsson, André Stén, Elona Wallengren, Love Westlund Gotby and Anton Älgmyr. Finally, I would like to thank Professor David Silver for his excellent course on machine learning and for making recordings of the lectures available online.

Linus Norström, Gothenburg, August 2019

Contents

List of Figures

List of Tables

1 Introduction

Research in artificial intelligence has been ongoing for the last 60 years [1].Great progress has been made in various sub fields of artificial narrow intelligence such as natural language processing [2], and research on artificial general intelligence is ongoing [3]. A great step in artificial narrow intelligence was recently taken when the AlphaGo program managed to beat the European Champion in Go in January 2016 [4]. One year later, AlphaGo won three games of Go in a row against the highest ranked player in the world, Ke Jie [5]. It achieved its performance using reinforcement learning and deep neural networks, playing against itself in order to refine its own play. The original iteration started by training on recorded human play before playing against itself, while the latest version AlphaZero [6] only used self-play to achieve its performance. This thesis aims to apply similar methods to another strategy game, Pokémon.

Pokémon is one of the worlds highest grossing franchises [7], but it all began with a pair of role-playing games for the hand held console Game Boy. Pocket Monsters Red and Green were released in Japan in 1996, and later in Europe as Pokémon Red and Blue in 1999. These games are classics today, having spawned seven generations of games for various hand held consoles so far, an anime which has been ongoing since 1997, and a lot of merchandise. One of the biggest parts of the Pokémon hand held games is Pokémon battles. The artificial intelligence that the player faces in these battles is very simple, ranging from pure random selectors to random selectors with a few additional decision rules. Winning a Pokémon battle against an opponent in the game is therefore not very challenging, even if the player and the opponent has similar teams. More complex algorithms couldn't be used either, because the console was restricted to a few kilobytes of internal memory [8] and the cartridges holding the games topped out at a few megabytes.

It isn't until playing against other players that the strategy of the games truly shines. Even the first games have support for player versus player battles, made possible by the Game Link Cable for the Game Boy, and this feature has been a part of the games since. With the internet, a community centered around playing Pokémon battles competitively grew, both using the official games, and using online simulators. The simulators allow players to create teams and battle with them against other players, entirely ignoring all other aspects of the games.

1.1 Purpose

The purpose of this thesis is to compare artificial intelligence algorithms for Pokémon battles. The goal is to make an artificial intelligence that can play on a human level or better.

1.2 Limitations

Each new version of the game brings new Pokémon, new abilities and possibly new features, further increasing the depth of the game. In order to keep the complexity reasonably low, the simulator is based on the first generation games.

The Pokémon teams used in the battles will be pre-determined, using the most common Pokémon in competitive play, eliminating all aspects of team building from the artificial intelligence.

2
Theory

All the general rules and mechanics of Pokémon battles are explained in Section 2.1. The information is datamined from the games by fans of the series and is available at various fansites such as Bulbapedia [9] and Smogon [10].

The thesis is split into three major parts, each represented by a Pokémon battle scenario. The specifics of each scenario is covered in Sections 2.2, 2.3 and 2.4.

All theory on AI found in 2.5 is based on a course given at UCL by David Silver [11] (lead researcher on the AlphaGo and AlphaZero projects) with slides and video recordings available online. The course in turn is based on "An Introduction to Reinforcement Learning" [12] and "Algorithms for Reinforcement Learning" [13].

2.1 Pokémon Battles

A first generation Pokémon battle is a turn-based strategy game, in which two players use Pokémon to battle each other. Each player has up to six Pokémon in their team and each Pokémon has up to four moves. There are two types of available actions: to switch the active Pokémon to another, or to use one of the moves of the active Pokémon. The game progresses one round at a time, during which both players simultaneously choose one action each. The actions are then carried out in an order decided by a priority system. One player wins when the other player has zero Pokémon with remaining hit points (HP) in their team, and at this point the game ends.

2.1.1 Pokémon

A Pokémon is a set of constants, variables and useable moves (with an image and a name added on top). The base strength of the Pokémon is decided by a set of five statistics: attack, defense, special, speed and maximum hit points (HP). At the start of the battle, the current HP of every Pokémon is equal to its maximum HP, and if the HP of a Pokémon reaches zero, it is unable to continue battling. Attack increases the ammount of HP that is removed from the opposing Pokémon when a physical damaging move is used, and defense decreases the HP removed when the opposing Pokémon uses a physical damaging move. When a special move is used instead of a physical move, special is used in calculations in place of attack and defense. Speed is used in the action priority system to determine the turn order, and it also decides the critical hit chance of any damage dealing move of the Pokémon.

The values of each of the statistics are decided by the base values, the individual values, the effort values and the level of the Pokémon. The base values are specific to each Pokémon species, and describe the general strength of a species, with values in [1, 255]. The individual values are generated when a Pokémon is first created in the game and are random values in [0, 15], except for the individual value of the maximum HP which is

$$\mathrm{HP}_i = 8 \cdot \frac{1 + (-1)^{A_i+1}}{2} + 4 \cdot \frac{1 + (-1)^{D_i+1}}{2} + 2 \cdot \frac{1 + (-1)^{Spe_i+1}}{2} + \frac{1 + (-1)^{Spc_i+1}}{2},$$

where A_i is the individual value of the attack, D_i is the individual value of the defense, Spe_i is the individual value of the speed and Spc_i is the individual value of the special. Both the effort values and the level of the Pokémon increase when opponents are defeated in the game, with the effort values in $[0,\ 65535] = [0,\ 256^2 - 1]$ and the level in [1, 100]. The formula for calculating the maximum HP of a Pokémon is

$$\mathrm{HP}_p = \left\lfloor \frac{\left((\mathrm{HP}_b + \mathrm{HP}_i) \cdot 2 + \left\lfloor \frac{\left\lceil \sqrt{\mathrm{HP}_e} \right\rceil}{4} \right\rfloor \right) \cdot L_p}{100} \right\rfloor + L_p + 10,$$

where HP_b is the base value, HP_i is the individual value and HP_e is the effort value of the maximum HP, and L_p is the level of the Pokémon. The value of the other four statistics are calculated by

$$S = \left\lfloor \frac{\left((B + I) \cdot 2 + \left\lfloor \frac{\left\lceil \sqrt{E} \right\rceil}{4} \right\rfloor \right) \cdot L_p}{100} \right\rfloor + 5,$$

where B is the base value, I is the individual value and E is the effort value of the statistic, and L_p is the level of the Pokémon. The square root of the effort value of any statistic isn't allowed to exceed 255 (due to the 8-bit limitation of the game), and will therefore be rounded down instead of up if the effort value exceeds $255^2 = 65025$.

Table 2.1: Type effectiveness chart for first generation Pokémon battles. The leftmost column represents the move type, while the top row represents the Pokémon type. Special types are written in italic, while physical types are written in regular.

	Nor	Fig	Fly	Poi	Gro	Roc	Bug	Gho	*Fir*	*Wat*	*Gra*	*Ele*	*Psy*	*Ice*	*Dra*
Normal	1	1	1	1	1	½	1	0	1	1	1	1	1	1	1
Fighting	2	1	½	½	1	2	½	0	1	1	1	1	½	2	1
Flying	1	2	1	1	1	½	2	1	1	1	2	½	1	1	1
Poison	1	1	1	½	½	½	2	½	1	1	2	1	1	1	1
Ground	1	1	0	2	1	2	½	1	2	1	½	2	1	1	1
Rock	1	½	2	1	½	1	2	1	2	1	1	1	1	2	1
Bug	1	½	½	2	1	1	1	½	½	1	2	1	2	1	1
Ghost	0	1	1	1	1	1	1	2	1	1	1	1	0	1	1
Fire	1	1	1	1	1	½	2	1	½	½	2	1	1	2	½
Water	1	1	1	1	2	2	1	1	2	½	½	1	1	1	½
Grass	1	1	½	½	2	2	½	1	½	2	½	1	1	1	½
Electric	1	1	2	1	0	1	1	1	1	2	½	½	1	1	½
Psychic	1	2	1	2	1	1	1	1	1	1	1	1	½	1	1
Ice	1	1	2	1	2	1	1	1	1	½	2	1	1	½	2
Dragon	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2

Each Pokémon has at least one and at most two types, and each move has one type. Damage dealing moves of certain types are more or less effective when used against Pokémon of other types, these relationships are shown in Table 2.1 as factors which are used in damage calculations. If a Pokémon has two types, the effectiveness is determined by multiplying the factors of both types. In addition, if the types of both the move and the Pokémon using the move are equal, the move will deal 50% more damage, this is often referred to as same type attack bonus (STAB).

One of the available types of actions is to switch the active Pokémon, as only one Pokémon can be used at a time. In general, all modifications of statistics will be reset when switching the active Pokémon, but status conditions will remain.

2.1.2 Moves

A move is an action that affects one or both players active Pokémon. Each move has a limited number of uses per battle called power points (PP), three constants (power, accuracy and maximum PP), and various secondary effects. The power of the move directly influences how much HP is removed from the opposing Pokémon, while the accuracy decides the probability of the move hitting or missing. Every time a move is used as an action, its PP is reduced by one. At the start of the battle, the PP is equal to the maximum PP, and if it reaches zero, the move can no longer be used. In the very rare case of no move having any PP remaining, the Pokémon can use a special move called Struggle, from which PP can't be removed.

All moves can hit or miss, with the exception of certain moves which don't have accuracy. A move will hit if

$$R_a < A_m,$$

where R_a is a random integer in [0, 255] and A_m is the accuracy of the move represented as an integer in [0, 255], where $A_m = 0$ represents 0%, $A_m = 255$ represents

100%. This results in a known bug where moves with 100% accuracy actually has a $\frac{1}{256} = 0.39\%$ chance to miss.

If the move has a power greater than zero, it is considered a damage dealing move and HP will be deducted from the opposing Pokémon when the move is used. The type of a damage dealing move decides if it is considered to be a physical or a special move, in accordance with Table 2.1. The HP to be removed when a physical move is used is calculated using integers according to the equation

$$\mathrm{HP} = \left\lfloor \frac{\left(\left\lfloor \frac{\left(\left\lfloor \frac{2 \cdot L_p}{5} \right\rfloor + 2 \right) \cdot P_m \cdot A_p}{50 \cdot D_p} \right\rfloor + 2 \right) \cdot R_{\mathrm{HP}}}{255} \right\rfloor, \tag{2.1}$$

where L_p is the level and A_p is the attack of the Pokémon using the move, P_m is the power of the move, D_p is the defense of the opposing Pokémon and R_{HP} is a random integer in [217, 255]. If the move is a special move as opposed to a physical move, the special of the Pokémon using the move will be used as A_p instead of the attack, and the special of the opposing Pokémon will be used as D_p instead of the defense. A critical hit is realized by doubling the level L_p during the calculation of (2.1), and occurs if

$$R_c < \left\lfloor \frac{Spe_b}{2} \right\rfloor,$$

where R_c is a random integer in [0, 255] and Spe_b is the base value of the speed of the Pokémon using the move. In addition, if a critical hit occurs, all statistic modifiers are ignored during the calculation of (2.1). The HP is then further modified by multiplying with the type factors shown in Table 2.1, and another factor 1.5 if STAB applies, rounding down after each multiplication with non-integers.

Table 2.2: The factors that are multiplied to a statistic which has been modified a certain number of stages. The product is always divided by 100 and rounded down, so stage 0 represents a non-modified statistic.

Stage	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
Factor	25	28	33	40	50	66	100	150	200	250	300	350	400

Non-damage dealing moves often have more unique or niche secondary effects, but two main groups of secondary effects exist: statistic modifying moves and status condition applying moves. Statistic modification is realized by increasing or decreasing stages corresponding to factors as shown in Table 2.2, which are multiplied with the statistic followed by dividing by 100 and rounding down whenever the statistic is used in a calculation. Status condition applying moves on the other hand, will apply a specific status condition.

2.1.3 Status Conditions

The Pokémon games contain many different status conditions that affect a battle. Only three status conditions will be included in this thesis: freeze, paralysis and sleep. The remaining status conditions aren't included because no move of any Pokémon in any team can apply them. Freeze and sleep renders the Pokémon unable to use any move, while paralysis stops the usage of moves 25% of the time. Freeze and paralysis persist indefinitely, while sleeping Pokémon wake up after between one and seven turns. Paralysis also decreases the speed of the Pokémon by 75%, which is separated from statistic modifications (meaning that both factors can apply). Only one status condition can be applied to a Pokémon at a time.

In order to keep these effects somewhat balanced, clauses are enforced, only enabling one Pokémon per team to be frozen and one to be sleeping at a time. Paralysis is not affected by clauses.

2.1.4 Action Priority

If both players choose to use a move, the Pokémon with the highest speed statistic carries out its move first. The speed of a Pokémon may be modified by a move, reduced by paralysis, or both at the same time. Both Pokémon may have the same speed, in which case the turn order is random. If a player chooses to switch their active Pokémon, the switch is always carried out before any move, and if both players choose to switch, the turn order is random.

2.2 The Simple Game

The simple game is an abstraction of a Pokémon battle in which both players have identical teams (a Squirtle with Tackle and Tail Whip). The players simultaneously choose one action each, which are then carried out in random order. Each player has two available actions: **gain** points, or **increase** the points gained. The base point gain is one, and increasing the points gained increases the points gained by one. The first player to reach n^2 points wins, where $n \in \mathbb{N}$, ending the game. Choosing a larger n simply increases the length of the game, here $n = 5$, making the goal $n^2 = 25$.

2.2.1 Optimal play

If a strategy should involve increasing, it is always optimal to do so at the start of the game. Thus, the optimal play can be found by examining the game length l as a function of the number of initial rounds spent increasing, i. Because the game ends when one player wins by reaching n^2 points, the winning player will decide the game length. The game length is the sum of i and g, where g is the number of rounds spent gaining. Therefore, this sum needs to be minimized. But g is also a function of i,

$$g_n(i) = \left\lceil \frac{n^2}{i+1} \right\rceil . \tag{2.2}$$

The game length $l = i + g$ can therefore be written as

$$l_n(i) = i + \left\lceil \frac{n^2}{i+1} \right\rceil , \tag{2.3}$$

substituting g using (2.2). As we're only considering integers, let's check the value of $l_n(i)$ for some values of i.

Table 2.3: The game length l when varying the number of initial rounds spent increasing i, calculated using (2.3). The number of rounds spent gaining g is also shown, calculated using (2.2).

i	0	1	2	3	**4**	5	6	7	...	24
g	25	13	9	7	**5**	5	4	4	...	1
l	25	14	11	10	**9**	10	10	11	...	25

Table 2.3 shows that $i = 4$ results in the shortest game length $l = 9$, and therefore the optimal play is to increase for four rounds, followed by gaining for five rounds. Any other strategy will finish the game in more than nine rounds. If both players follow the optimal strategy, the outcome is random, because the turn order is random. If one player follow the optimal strategy, but the other doesn't, the player following the optimal strategy will always win.

2.3 The Starter Scenario

The starter scenario is a replicated battle from the original hand held games, when the player chooses its first Pokémon. Three Pokémon are available to choose from: Bulbasaur, Charmander and Squirtle, being grass (and poison), fire and water types respectively. These three have a rock-paper-scissor like relationship according to Table 2.1, and in the games, the rival will always choose the Pokémon with a type advantage against the players choice. The first battle of the game then takes place with these Pokémon, which at this point can only use normal type moves, significantly simplifying the type table to what is shown in Table 2.4.

Table 2.4: Type effectiveness chart for the starter scenario. The leftmost column represents the move type, while the top row represents the Pokémon type. Special types are written in italic, while physical types are written in regular.

	Poison	*Fire*	*Water*	*Grass*
Normal	1	1	1	1

The three potential Pokémon choices lead to three different subscenarios, each with a different balance. The balances are decided by the statistics of the three Pokémon and their available moves rather than their types, as no move with a type advantage is available. Compared to the simple game, this scenario also increases the stochasticity, as moves can miss and critically hit.

2.3.1 The Pokémon

The three available Pokémon are shown in Table 2.5. The level of all Pokémon is five, and the effort values for all statistics are zero. The individual values are also set to zero rather than being random. As only physical moves are available, the special statistic is irrelevant for this scenario.

Table 2.5: The statistics of all three Pokémon available in the starter scenario.

Pokémon	Type(s)	HP	Attack	Defense	Special	Speed	Crit	Moves
Bulbasaur	Grass Poison	19	9	9	11	9	8%	Tackle Growl
Charmander	Fire	18	10	9	10	11	12%	Scratch Growl
Squirtle	Water	19	9	11	10	9	8%	Tackle Tail Whip

2.3.2 The Moves

Table 2.6 shows all the moves in the starter scenario. There's two damage dealing moves available, Scratch and Tackle, with Scratch having slightly higher power and accuracy. Two statistic modifying moves are also available: Growl and Tail Whip, lowering the attack and defense statistics respectively by one stage. All moves have sufficiently high PP for this scenario.

Table 2.6: The statistics of all the available moves in the starter scenario.

Move	Type	Power	Accuracy	PP	Secondary Effect
Scratch	Normal	40	100%	35	None.
Tackle	Normal	35	95%	35	None.
Growl	Normal	0	100%	40	Decreases the attack of the opponent by one stage.
Tail Whip	Normal	0	100%	30	Decreases the defense of the opponent by one stage.

2.3.3 Optimal play

By the same logic used in the simple game, it is always better to use a statistic modifying move at the start of the game, before using any damage dealing move. The highest performing number of initial statistic modifying moves i in this scenario aren't as easily found as in the simple game due to the added stochasticity, but can be found by simulations measuring the win ratio when varying i. The highest performing i is defined as the i with the highest average performance over all values of i for the opponent.

2.4 The Overused Scenario

The overused scenario is meant to represent a high level competitive first generation Pokémon battle. Overused is the name of the category of Pokémon which appears the most in competitive play. The rules also allow Pokémon from the less used categories to be in a team competing in the Overused category, banning only the Pokémon in the Uber category which are considered to be too powerful. In the first generation, only the Pokémon Mew and Mewtwo are restricted to being Uber, and fifteen Pokémon are in the Overused category.

While assembling a good team is a huge part of playing Pokémon (as important as the battle itself), the team is predetermined and equal for both players in this scenario. The team is made up of some of the most common Pokémon in the Overused category, with their most commonly found moves.

In contrast with the simple game and the starter scenario, the overused scenario allows for infinite games and ties. A limit to the number of rounds must be used in simulations in order to prevent infinite games, and a tie can occur if a damage dealing move with self-inflicted damage reduces the HP of both players final remaining Pokémon to zero. Because of the definition of winning a Pokémon battle, a game ended by reaching the round limit has no winners, and a tie is considered a win for both players.

2.4.1 The Pokémon

All six Pokémon in the team are shown in Table 2.7. The level of all Pokémon is 100, the effort values are above 65025 resulting in the square root of them being 255, and the individual values of all statistics are 15. In other words, they are as powerful as they can be.

Table 2.7: The statistics of all six Pokémon that make up the team in the overused scenario.

Pokémon	Type(s)	HP	Attack	Defense	Special	Speed	Crit	Moves
Exeggutor	Grass Psychic	393	288	268	348	208	10%	Stun Spore Sleep Powder Psychic Explosion
Rhydon	Ground Rock	413	358	338	188	178	7%	Body Slam Earthquake Rock Slide Substitute
Chansey	Normal	703	108	108	308	198	9%	Ice Beam Thunderbolt Thunder Wave Soft-Boiled
Tauros	Normal	353	298	288	238	318	21%	Body Slam Ice Beam Hyper Beam Earthquake
Snorlax	Normal	523	318	228	228	158	5%	Body Slam Hyper Beam Earthquake Self-Destruct

Zapdos	Electric Flying	383	278	268	348	298	19%	Drill Peck Thunderbolt Thunder Wave Agility

Not all types are represented in this team, simplifying the type table to Table 2.8. Half of the Pokémon in the team are of normal type, while the other three have some type advantages and disadvantages.

Table 2.8: Type effectiveness chart for the overused scenario. The leftmost column represents the move type, while the top row represents the Pokémon type. Special types are written in italic, while physical types are written in regular.

	Normal	Flying	Ground	Rock	*Grass*	*Electric*	*Psychic*
Normal	1	1	1	½	1	1	1
Flying	1	1	1	½	2	½	1
Ground	1	0	1	2	½	2	1
Rock	1	2	½	1	1	1	1
Grass	1	½	2	2	½	1	1
Electric	1	2	0	1	½	½	1
Psychic	1	1	1	1	1	1	½
Ice	1	2	2	1	2	1	1

2.4.2 The Moves

Table 2.9 shows all available moves in the overused scenario. Most of the secondary effects are too complicated to be fully explained in the table however.

Table 2.9: The statistic of all the available moves for all Pokémon in the team in the overused scenario.

Move	Type	Power	Accuracy	PP	Secondary Effect
Body Slam	Normal	85	100%	24	Has a 30% chance of paralyzing the opponent.
Ice Beam	Ice	95	100%	16	Has a 10% chance of freezing the opponent.
Hyper Beam	Normal	150	90%	8	Requires recharging on the following turn.
Drill Peck	Flying	80	100%	32	None.
Stun Spore	Grass	-	75%	48	Paralyzes the opponent.
Sleep Powder	Grass	-	75%	24	Puts the opponent to sleep.
Thunderbolt	Electric	95	100%	24	Has a 10% chance of paralyzing the opponent.
Thunder Wave	Electric	-	100%	32	Paralyzes the target.
Earthquake	Ground	100	100%	16	None.

Psychic	Psychic	90	100%	16	Has a 33.2% chance of decreasing the special of the opponent by one stage.
Agility	Psychic	-	-%	48	Increases the speed of the user by two stages.
Selfdestruct	Normal	130	100%	8	Causes the user to faint.
Softboiled	Normal	-	-%	16	Restores up to 50% of the users hit points.
Explosion	Normal	170	100%	8	Causes the user to faint.
Rock Slide	Rock	75	90%	16	None.
Substitute	Normal	-	-%	16	Removes 25% of the users hit points and creates a substitute.
Struggle	Normal	50	-%	10	Casues 50% recoil damage to the user.

Substitute can only be used by Rhydon in the overused scenario. When Rhydon uses Substitute, 103 HP is removed, and a substitute is created with 104 HP, the same type and the same stats as Rhydon. Substitute will fail if Rhydon has less than 103 HP, but if Rhydon has exactly 103 HP, it will faint and no substitute will be created. Once created, the substitute essentially absorbs all incoming damage until it breaks by running out of HP. Self-inflicted damage such as from Struggle will ignore any active substitute. While the substitute is active, Rhydons stat stages can not be lowered (by Psychic), it can not be frozen (by Ice Beam) and it can not be paralyzed by damage dealing moves (Body Slam or Thunderbolt), even if the damage from these moves break the substitute. Rhydon can however fall asleep or be paralyzed by non-damage dealing moves (Stun Spore and Sleep Powder, but not Thunder Wave due to Rhydon being immune to electric type moves) with an active substitute.

Body Slam and Thunderbolt can't paralyze Pokémon of the same type as the move, normal and electric respectively. Rhydon can't be paralyzed from Thunderbolt or Thunder Wave either, as Rhydon is immune to electric type moves. Stun Spore, Sleep Powder and Thunder Wave only inflict a status condition, and work even if the opponent has a substitute.

Hyper Beam doesn't always require recharging. If it misses, breaks a substitute, knocks out the opponent or if the user is put to sleep between using Hyper Beam and recharging, the user will not need to recharge.

Selfdestruct and Explosion does not cause the user to faint if it breaks the opponents substitute. Additionally, when calculating the damage to be inflicted, the defense of the opponent is halved, essentially giving these moves double power.

Struggle is a very special move, only usable if the Pokémon has no PP left for any move, and PP will never be deducted from Struggle. It inflicts damage to the opponent, half of which is also inflicted to the user, unless if Struggle breaks the opponents substitute.

2.5 Artificial Intelligence

The goal of the artificial intelligence (AI) is to choose the best possible action to take for each round in the game. This is quantified as the action value function $q(s, a)$, defined as the value of taking an action a when in a state s. The value of winning the game is set to 1 and the value of losing the game is set to -1. If the true action value function can be found, the best possible action is the one that maximizes $q(s, a)$. But as the complexity of the game increases, so does the difficulty of finding the true action value function. Some algorithm must therefore be used to find the best possible approximation $\hat{q}(s, a)$ of the action value function, and this algorithm is what separates the different AI from each other.

The AI must then choose an action based on $\hat{q}(s, a)$, by means of a policy. Because $\hat{q}(s, a)$ is an approximation, choosing the a that maximizes it for a specific s is naive. As an alternative policy, the softmax function

$$p(s, a) = \frac{e^{\frac{\hat{q}(s,a)}{\tau}}}{\sum_a e^{\frac{\hat{q}(s,a)}{\tau}}}, \tag{2.4}$$

where τ is a parameter called the temperature, is a way of converting $\hat{q}(s, a)$ to probabilities $p(s, a)$ of choosing an action a when in a state s, increasing the stochasticity of the choice while retaining some bias toward actions with higher $\hat{q}(s, a)$. As τ approaches zero, the softmax function approaches the normal maximum function.

2.5.1 Random AI

The completely random AI chooses a random action from all available actions each round. This can be likened to setting $\hat{q}(s, a)$ to any arbitrary value and letting τ approach infinity, or setting $\hat{q}(s, a)$ to the same value for every a and letting $\tau > 0$. Possibly the most simple AI imaginable, it has great secondary use in other algorithms. For example, one can use it as a search AI, or use it instead of another algorithm with a small probability ε in order to increase stochasticity.

It can be improved upon by enforcing simple rules or restrictions. In the case of Pokémon battles, such a rule can be to never switch Pokémon while the active Pokémon has remaining HP.

2.5.2 Hard coded AI

The hard coded AI has a predefined sequence of actions that are to be taken, one for each round. This can be likened to setting $\tau = 0$ and $\hat{q}(s, a_*) > \hat{q}(s, a)$ for the action a_* in the sequence that is to be chosen in the state s. The success of a hard coded algorithm depends entirely on the sequence of actions. As the stochasticity of the game increases, predefining a sequence of actions with high performance becomes increasingly difficult.

2.5.3 Linear Combination of Features AI

The linear combination of features AI approximates the action value function by

$$\hat{q}(s,a) = \mathbf{w}_a \cdot \mathbf{x}(s) + b_a, \tag{2.5}$$

where $\mathbf{x}(s)$ are the features of the state s, $\mathbf{w}_a$ are weights for an action a and b_a is a bias for a. The features $\mathbf{x}(s)$ are values in [0, 1] that are representative of the current game state.

The optimal values of the weights and biases are found by training: by having the AI play a large amount of games against itself and updating the values of the weights and biases depending on the results of the games. All weights and biases are initialized to zero and updated by stochastic gradient descent

$$\begin{cases} \Delta\mathbf{w}_a = \alpha \cdot (R - \hat{q}(s,a)) \cdot \nabla_{\mathbf{w}_a}\hat{q}(s,a) \\ \Delta b_a = \alpha \cdot (R - \hat{q}(s,a)) \cdot \frac{\mathrm{d}\hat{q}(s,a)}{\mathrm{d}b_a} \end{cases}, \tag{2.6}$$

where α is the learning rate and R is the reward from a played game. The reward of winning is 1 and the reward of losing is -1. The two derivatives of (2.5) are

$$\begin{cases} \nabla_{\mathbf{w}_a}\hat{q}(s,a) = \mathbf{x}(s) \\ \frac{\mathrm{d}\hat{q}(s,a)}{\mathrm{d}b_a} = 1 \end{cases}, \tag{2.7}$$

and inserting (2.7) into (2.6) gives the stochastic gradient descent as

$$\begin{cases} \Delta\mathbf{w}_a = \alpha \cdot (R - \hat{q}(s,a)) \cdot \mathbf{x}(s) \\ \Delta b_a = \alpha \cdot (R - \hat{q}(s,a)) \end{cases}.$$

Because R is decided by the result of the game, the update of the weights and biases are carried out after each game.

2.5.4 Pure Monte Carlo Search AI

The pure Monte Carlo search AI plays out many simulations of the game from the current game state s, using the completely random AI to choose all actions during the simulations. After the entire search is completed, the action value function is approximated as

$$\hat{q}(s,a) = \frac{w(s,a)}{c(s,a)}, \tag{2.8}$$

where $w(s,a)$ is the number of times taking action a when in state s led to winning simulations of the game, and $c(s,a)$ is the number of times action a was chosen in state s.

The first action a chosen in the simulations (the only one not chosen by the completely random AI) can be chosen in different ways. One way is to simply rotate between all available actions, ensuring the same amount of searches for each action. Another way is to use the partially completed $\hat{q}(s,a)$ from (2.8) to decide the first action a, introducing a bias toward more promising actions in the search.

2.5.5 Monte Carlo Tree Search AI

The Monte Carlo tree search AI expands the pure Monte Carlo search AI to use a search tree instead of the completely random AI to choose actions during the simulations. The search tree is stored as $w(s, \mathbf{a})$ and $c(s, \mathbf{a})$ values, where $\mathbf{a}$ is a sequence of actions starting from the state s. During the simulations

$$\hat{q}(s, \mathbf{a}) = \frac{w(s, \mathbf{a})}{c(s, \mathbf{a})} \tag{2.9}$$

is used as the action value function approximation for previously tried action sequences. If an action sequence $\mathbf{a}'$ has not been tried yet, $\hat{q}(s, \mathbf{a}')$ is set to some value larger than one. After every simulation of the game, the search tree is updated for all previously tried action sequences, and at most one new action sequence is added to the search tree. After the entire search is completed, the action value function is approximated by (2.8), in the same way as for the pure Monte Carlo search AI.

In addition, the depth of search tree is limited by the depth parameter d. If the number of simulated rounds exceed d, the random algorithm is used to finish the simulation. This also means that no action sequence longer than d is stored in the search tree. When $d = 1$, the Monte Carlo tree search AI is the same thing as the pure Monte Carlo search AI using the partially completed $\hat{q}(s, a)$ from (2.8) to decide the initial action.

3
Methods

Sections 3.1, 3.2 and 3.3 cover the implementations and usage of the AI algorithms for the three scenarios. The first scenario is an abstraction of a simple and symmetric Pokémon battle, and was mostly used to test the AI implementation and to make sure that everything worked. The second scenario is the first actual Pokémon battle found in the games, introducing unbalance and stochasticity to the game while keeping it fairly simple. The third and final scenario is a full scale competitive Pokémon battle, a scenario where the optimal actions aren't always obvious even to a human player. All code was written in Python 3, and all simulations were run on normal PCs.

3.1 The Simple Game

The simple game was used as a way of making sure that the implementation of the different algorithms worked as intentional. As the optimal action sequence was known from Section 2.2.1, the optimal action sequence usage was used as a measurement of performance.

3.1.1 Linear Combination of Features AI

The linear combination of features AI was implemented using softmax (2.4) with temperature τ during training and $\tau = 0$ (normal maximum) when measuring performance. The weights and biases were trained by self-battle for $n_t = 10^6$ games with $\alpha = 0.05$ and $\tau = 0.2$, and the average of 10^3 values spread evenly during training were taken as the final value for each weight and bias. The performance was then measured for 10^3 games using $\tau = 0$.

Two features were used to describe the state of the game, designed to emphasize the similarity to a Pokémon battle. The first describes the current number of points p_g as

$$f_g = \frac{n^2 - p_g}{n^2},$$

while the second describes the current number of points gained per round p_i as

$$f_i = \begin{cases} \frac{p_i - 1}{n^2 - 1}, & \text{if } p_i < n^2 \\ 1, & \text{else} \end{cases} .$$

They were designed to mirror the remaining HP and stages of statistic modification respectively.

3.1.2 Monte Carlo Tree Search AI

The Monte Carlo tree search AI was implemented using softmax (2.4) with temperature τ during the search simulations and $\tau = 0$ (normal maximum) when making the choice in the actual game. In addition, with a probability ε the random AI was used instead of $\hat{q}(s, \mathbf{a})$ from (2.9) to choose the next action during the search. The number of search simulations per round was limited by the parameter n_s.

In order to find optimal parameter choices for Monte Carlo tree search, parameter sweeps were carried out. The optimal action usage for 10^3 games was measured for $\varepsilon \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$ and $\tau \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$, with $d = 5$ and $n_s = 10^3$. Furthermore, both the optimal action usage and time spent searching for 10^3 games was measured for $d \in \{1, 5, 10, 15, 20, 25\}$ and $n_s \in \{200, 400, 600, 800, 1000\}$, with $\varepsilon = 0.2$ and $\tau = 0.2$.

The performance was measured for 10^3 games using $d = 15$, $\varepsilon = 0$, $n_s = 10^3$, $\tau = 0.2$ during the search simulations and $\tau = 0$ in the actual game.

3.1.3 Pure Monte Carlo Search AI

The Monte Carlo tree search AI implementation described in Section 3.1.2 with $d = 1$ could also be viewed as the pure Monte Carlo search AI using the partially completed $\hat{q}(s, a)$ from (2.8) to decide the initial action. The performance was measured for 10^3 games using $\varepsilon = 0$, $n_s = 10^3$, $\tau = 0.2$ during the search simulations and $\tau = 0$ in the actual game.

3.2 The Starter Scenario

The starter scenario consists of three separate but similar subscenarios. Their balance was therefore investigated first, and only the most balanced subscenario was used for measuring performance of the different AI. Performance was measured as the win ratio when the AI played against each other. The imbalance of the subscenario must therefore be taken into account when viewing the results.

3.2.1 Hard Coded AI

In order to determine the highest performing action sequence for the hard coded AI, simulations were carried out in which the hard coded AI battled itself $n_g = 10^6$ times with different number of initial statistic modifying moves, in accordance with Section 2.3.3. These simulations were carried out for all three subscenarios. The result of these measurements were used to decide which subscenario was the most suited for further study.

The hard coded AI was also used to measure the performance of both the linear combination of features AI and the Monte Carlo tree search AI. When used in this way, the hard coded AI always used the highest performing action sequence.

3.2.2 Linear Combination of Features AI

The linear combination of features AI was implemented using softmax (2.4) with temperature τ during training and $\tau = 0$ (normal maximum) when measuring performance. Because the game is not balanced, two separate set of weights and biases were used for the two players.

The learning rate α and temperature τ during training of the linear combination of features AI were optimized through parameter sweeps. The weights and biases corresponding to both players were trained by playing against each other, and the parameters were shared during the sweep. The performance after $n_t = 2{\cdot}10^4$ training games, from which the average of 10^3 values spread evenly from the second half of the training were taken as the final value for each weight and bias, was measured as the win ratio of 10^3 battles against the hard coded AI for $\alpha \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ and $\tau \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$. For each set of α and τ, the linear combination of features AI was trained and had its performance measured 100 times, enabling analysis of both the average performance and the maximum performance.

The linear combination of features AI was then trained using $\alpha = 0.1$ and $\tau = 0.2$ for $n_t = 10^6$ games, from which the average of 10^3 values spread evenly during training were taken as the final value for each weight and bias. Finally, the performance was measured using $\tau = 0$, against both the hard coded AI for 10^6 games and the Monte Carlo tree search AI for 10^3 games.

Features were constructed to give information about the remaining HP and stages of statistic modifications of all Pokémon, and the remaining PP of all moves, for both players. The remaining number of HP was represented as

$$f_{\text{HP}} = \frac{\text{HP}_{\text{remaining}}}{\text{HP}_{\text{max}}}, \tag{3.1}$$

where $\text{HP}_{\text{remaining}}$ is the remaining HP and HP_{max} is the maximum HP. In this scenario, both attack and defense decreasing moves exist, so these modifications were represented as

$$f_{\text{stat}} = \frac{-\text{stat}}{6}, \tag{3.2}$$

where stat is the current stage of status modification as described in Table 2.2. The feature representing the remaining PP

$$f_{\text{PP}} \begin{cases} 1 - \frac{\text{PP}_{\text{remaining}}}{4}, & \text{if PP} < 4 \\ 0, & \text{else} \end{cases}, \tag{3.3}$$

where $\text{PP}_{\text{remaining}}$ is the remaining PP, was constructed to only activate at very low PP, in order to enable discouraging (or encouraging) using a single move too much without discouraging (or encouraging) its normal usage.

3.2.3 Monte Carlo Tree Search AI

The Monte Carlo tree search AI was implemented using softmax (2.4) with temperature τ during the search simulations and $\tau = 0$ (normal maximum) after the search. In addition, the random algorithm was used instead of $\hat{q}(s, \mathbf{a})$ from (2.9) with a probability ε during the search. The number of search simulations per round was limited by the parameter n_s.

Parameter sweeps of ε and τ were carried out for the Monte Carlo tree search AI. The performance against the hard coded algorithm was measured as the win ratio of 10^3 battles against the hard coded AI for $\varepsilon \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$ and $\tau \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$, with $d = 5$ and $n_s = 200$.

The performance was measured for 10^3 games using $d = 10$, $\varepsilon = 0.1$, $n_s = 10^3$, $\tau = 0.2$ during the search simulations and $\tau = 0$ in the actual game, against both the hard coded AI and the linear combination of features AI.

3.3 The Overused Scenario

A few simple checks were implemented before entering the algorithms when an AI chooses an action. If a recharge is needed after a Hyper Beam, no other action can be taken, so the AI simply recharges. If there's only one available action to choose from (a highly unlikely scenario), the AI simply chooses this action. The AI started every round by constructing a list of all allowed actions given the entire game state. Forbidden actions could therefore never be chosen.

Because of the possibility of infinite games, a limit of 100 rounds was enforced in all games. Any game reaching the limit was aborted with no player declared winner. In addition, any game ending in a tie was considered a win for both players.

3.3.1 Completely Random AI

The completely random AI constructs a list of all allowed actions and then chooses one action randomly from the list.

3.3.2 Improved Random AI

The improved random AI is the combination of the completely random AI and a simple rule: only choose moves while the active Pokémon has hit points remaining. It constructs a list of all allowed moves if the active Pokémon has HP remaining, and a list of all allowed switches else. It then chooses one action randomly from the list.

3.3.3 Linear Combination of Features AI

The linear combination of features AI was implemented with learning rate $\alpha = 0.01$ using softmax (2.4) with temperature $\tau = 0.1$ during training and $\tau = 0$ (normal maximum) when measuring performance. Because each player has six Pokémon and any combination of active Pokémon will require different strategies, 6^2 sets of weights and biases for each of the four moves and 6 sets of weights and biases for the switch to each Pokémon were used in the algorithm.

The weights and biases were found through repeated training, performance measuring and reinitialization. Every fresh set of weights and biases was trained by playing against itself for 10^4 initial games. Its performance was then measured for

10^3 games against the improved random AI. If the measured performance was a new global record, the weights and biases were saved. Further training was then carried out by self-play for 10^3 games at a time, always followed by a performance measurement. If the performance at any point fell below 75% of the highest performance of the current initialization of the linear combination of features AI, the current AI was abandoned and a new one was initialized.

Features were constructed to give all relevant information about the game. After using Hyper Beam, a recharge turn is required. The recharging status of the opponent was given as

$$f_{\text{recharge}} = \begin{cases} 1, & \text{if opponent needs recharging} \\ 0, & \text{else} \end{cases}, \tag{3.4}$$

but as all other actions are forbidden during a recharge round, the AI doesn't need information about its own recharging status. All other features were mirrored for both players. Any active substitute was represented as

$$f_{\text{substitute}} = \frac{\text{remaining HP of substitute}}{\text{max HP of substitute}}. \tag{3.5}$$

Only special and speed can be modified in this scenario: special can be decreased and speed can be increased. Two separate types of features was constructed to represent this,

$$f_{\text{special}} = \frac{-\text{special}}{6}, \tag{3.6}$$

to represent special being decreased and

$$f_{\text{speed}} = \frac{\text{speed}}{6}, \tag{3.7}$$

to represent the increasing of speed, where special and speed is the current stage of modification as described in Table 2.2. All three available status conditions (freeze, paralysis and sleep) was represented as

$$f_{\text{status}} = \begin{cases} 1, & \text{if status is applied} \\ 0, & \text{else} \end{cases}, \tag{3.8}$$

and one final pair of features represented the remaining hit points as

$$f_{\text{HP}} = \frac{\text{remaining HP}}{\text{max HP}}. \tag{3.9}$$

When constructing $\mathbf{x}(s)$ for a move, the information about the active Pokémon of both players was used for the features, while $\mathbf{x}(s)$ for switches instead used information about the AIs Pokémon intended for switching to.

3.3.4 Pure Monte Carlo Search AI

Because the performance of search algorithms increases with search speed, the pure Monte Carlo search AI was implemented in the simplest possible way. The initial action is rotated between all available actions, and all other action choices are made using the completely random AI. The search length was defined using time rather than games, and was set to $t_s = 10\,\text{s}$ during the performance measurements.

3.3.5 Monte Carlo Tree Search AI

The Monte Carlo tree search AI was implemented using softmax (2.4) with temperature $\tau = 0.2$ as the search policy and $\tau = 0$ (normal maximum) after the search is completed. In addition, the random AI was used instead of $\hat{q}(s, \mathbf{a})$ from (2.9) with a probability $\varepsilon = 0.1$ to choose the next action during the search.

A parameter sweep of d was carried out, using $n_s = 250$. Performance in this parameter sweep was measured as the win ratio for 10^3 games against the improved random AI, for $d \in \{1, 5, 10, 15, 20, \}$. The time used per game was also measured.

3.3.6 Human Benchmark

As a final test of performance, participants were invited to battle against the AI with the highest performance in the internal tests. The participants were asked to sort themselves into three categories shown in Table 3.1 and given three weeks to complete as many games as possible using a simple command-line interface giving all information about the scenario.

Table 3.1: The categories of players.

Number	Description
1	Have never played any version of the hand held Pokémon games
2	Have played the hand held Pokémon games,
	but never competitively (Pokémon Showdown or similar)
3	Have played Pokémon competitively (Pokémon Showdown or similar)

The pure Monte Carlo search AI was chosen as the highest performing AI. When playing against human players, the search was conducted for at least $t_s = 60\,\mathrm{s}$ while the players made their choice, but if the human player used more than $60\,\mathrm{s}$ to decide, the algorithm would extend its search even further.

4

Results

The results is split into three parts in Sections 4.1, 4.2 and 4.3, representing the three Pokémon battle scenarios.

4.1 The Simple Game

Figure 4.1 shows the values of the weights and biases during the training of the linear combination of features AI. The weights and biases very quickly approach some values, and then oscillate around it. Repeating the initialization and training gives the same result. The result was reproduced with other values of τ, but increasing α too much resulted in divergence during training.

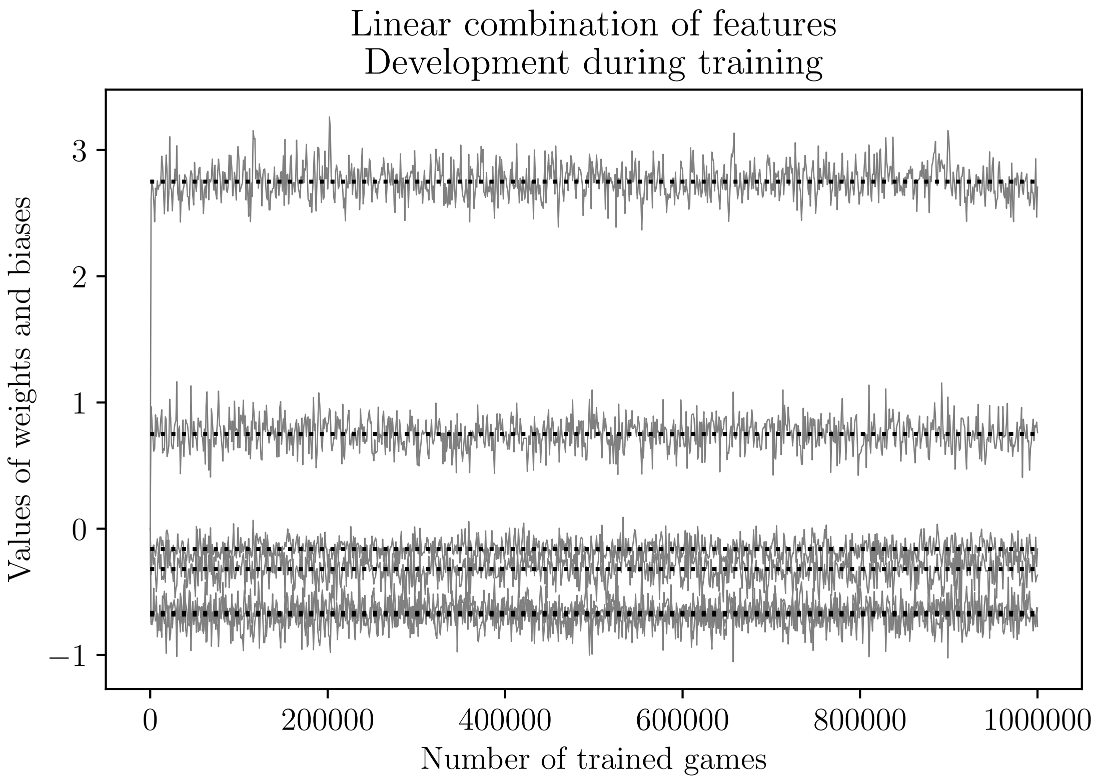

Figure 4.1: The development of the weights and biases during training of the linear combination of features AI. Each line is the value of a weight or bias. The dotted lines represent the average of each weight or bias.

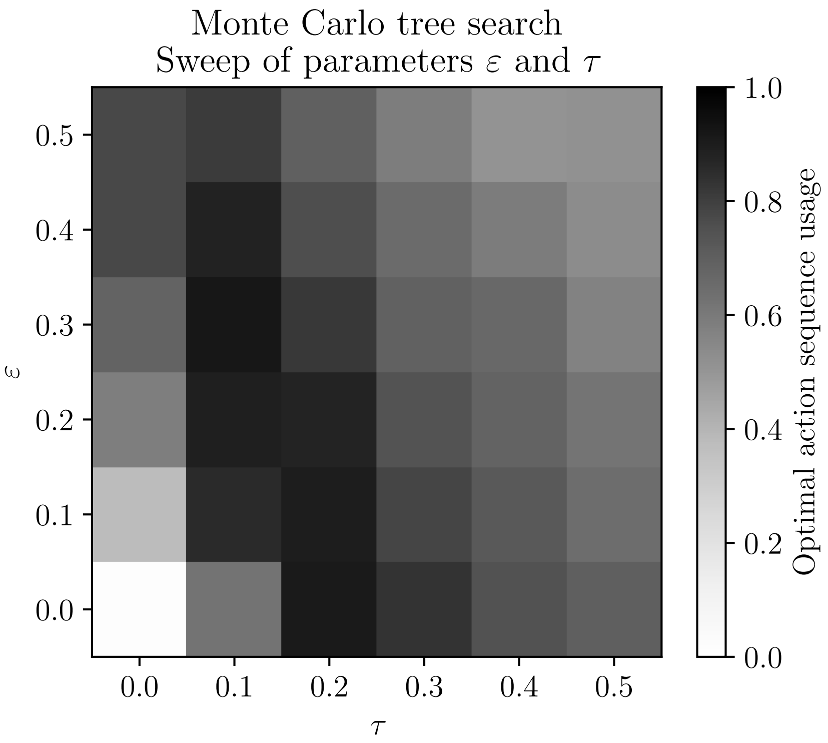

Figure 4.2: The performance of Monte Carlo tree search for different values of ε and τ.

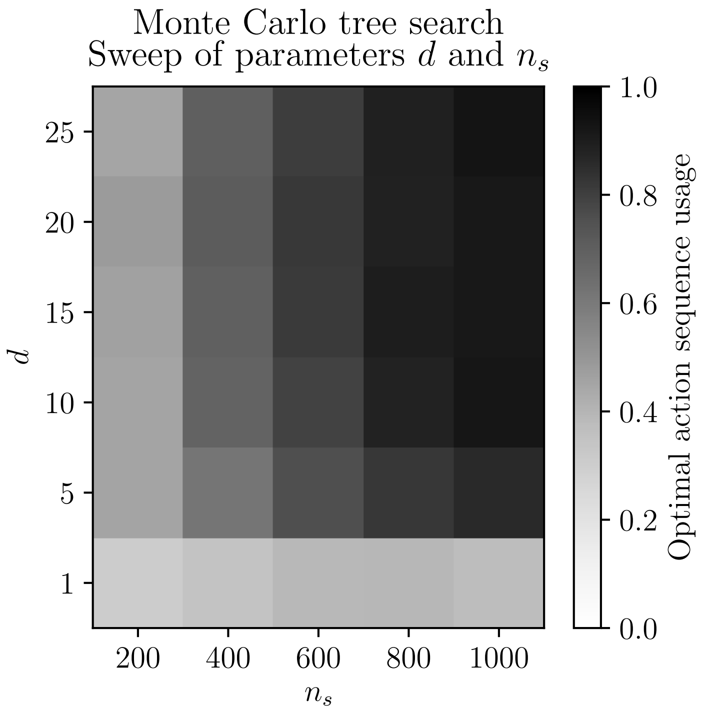

Figure 4.3: The performance of Monte Carlo tree search for different values of d and n_s.

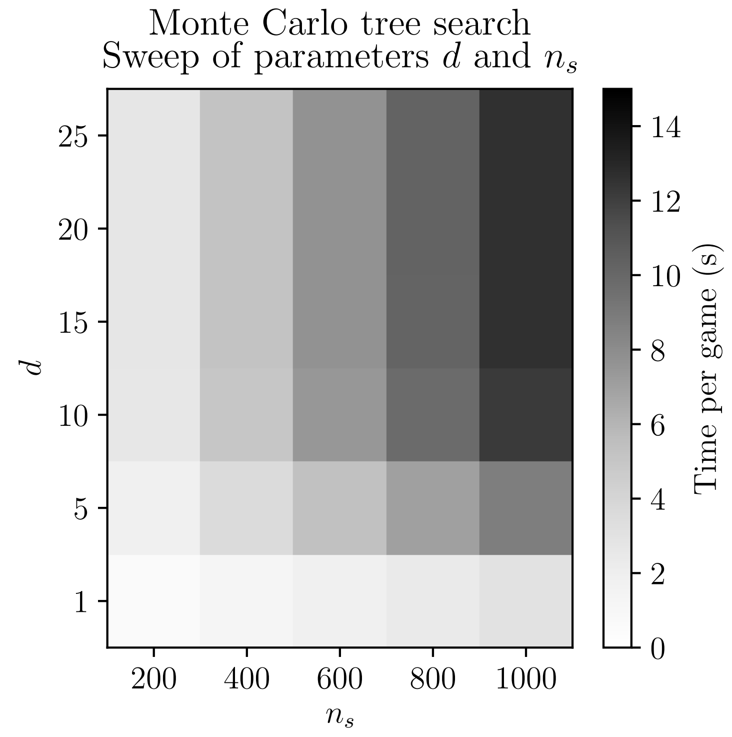

Figure 4.4: The time usage of Monte Carlo tree search for different values of d and n_s.

The result of the ε and τ parameter sweep for the Monte Carlo tree search AI is shown in Figure 4.2. Both ε and τ increase stochasticity during search, and increasing one requires decreasing the other in order to maintain performance. Setting $\varepsilon = 0$ and $\tau \neq 0$ has higher performance than setting $\varepsilon \neq 0$ and $\tau = 0$.

The result of the d and n_s parameter sweeps for the Monte Carlo tree search AI is shown in Figures 4.4 and 4.3. The performance generally increases with the time spent searching. Increasing d past 15 doesn't affect either the time or the performance at all. For the pure Monte Carlo search AI ($d = 1$), the performance doesn't increase with n_s at all.

Table 4.1: The performance of the A.I. algorithms in the simple game. Pure Monte Carlo search in this scenario is simply Monte Carlo tree search with $d = 1$.

A.I. algorithm	Optimal strategy usage
Pure Monte Carlo search	38.3%
Monte Carlo tree search	83.8%
Linear combination of features	100.0%

The performances measured in optimal action sequence usage for all AI are shown in Table 4.1. It is clear that the linear combination of features AI is very suited for this game as it has perfect performance. The pure Monte Carlo search AI has very low performance and the Monte Carlo tree search AI has higher, but not perfect, performance.

4.2 The Starter Scenario

The performance of the hard coded AI using all action sequences for all three subscenarios are shown in Figures 4.5, 4.6 and 4.7. The subscenario of Bulbasaur versus Charmander is very unbalanced, with Charmander winning most of the time.

The average performances of the hard coded AI using the highest performing action sequences against an opponent using all action sequences in the initial statistic modifying moves sweep equally are shown in Table 4.2, while the win ratio of the hard coded AI using the highest performing action sequences playing against each other are shown in Table 4.3.

The most balanced subscenario by a small margin is Charmander versus Squirtle, which was therefore used to measure the performance of all algorithms. From Figure 4.6, we can see that both Charmander and Squirtle can't improve their performance by changing action sequences given that the opponent uses their highest performing action sequence.

Figures 4.8, 4.9, 4.10 and 4.11, show the results of the parameter sweep for the training of the weights and biases for the linear combination of features AI. With a few exceptions, most parameter choices can result in high performance for both players, but the same can not be said for the average performance. Using $\alpha = 0.1$ and $\tau = 0.2$ yields high average and maximum performance for both players.

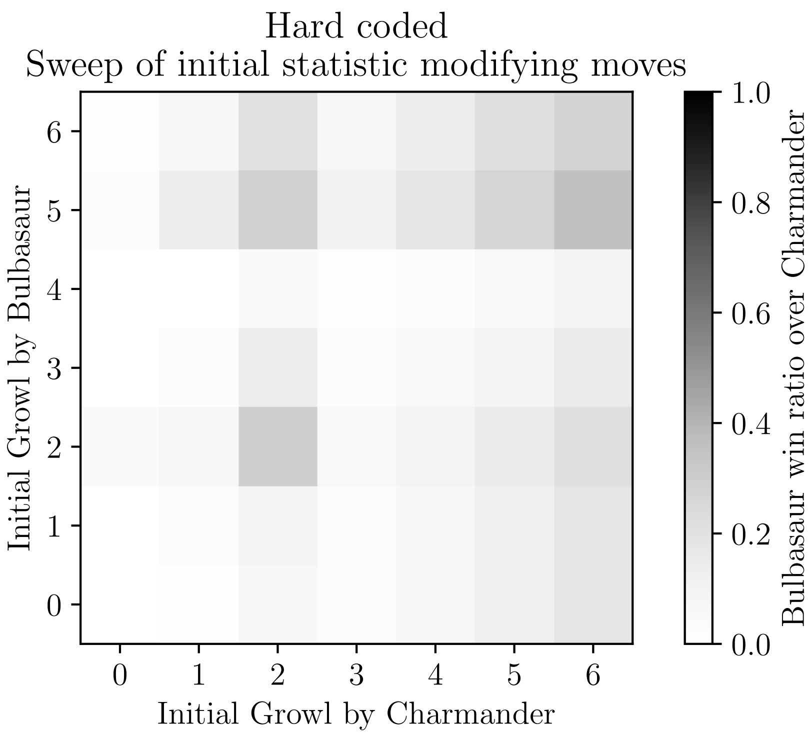

Figure 4.5: The win ratio of Bulbasaur against Charmander given that both follow an action sequence with a predetermined number of initial statistic modifying moves followed by only damage dealing moves.

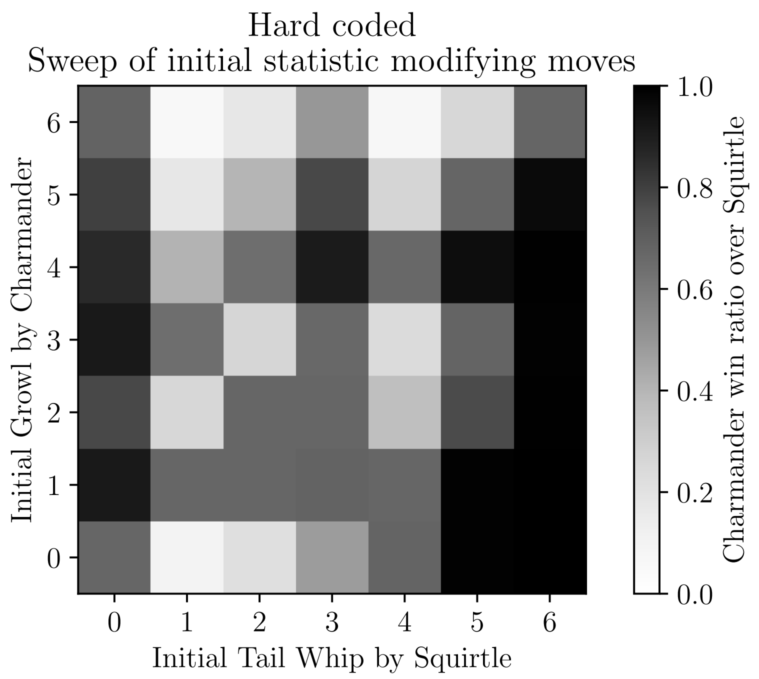

Figure 4.6: The win ratio of Charmander against Squirtle given that both follow an action sequence with a predetermined number of initial statistic modifying moves followed by only damage dealing moves.

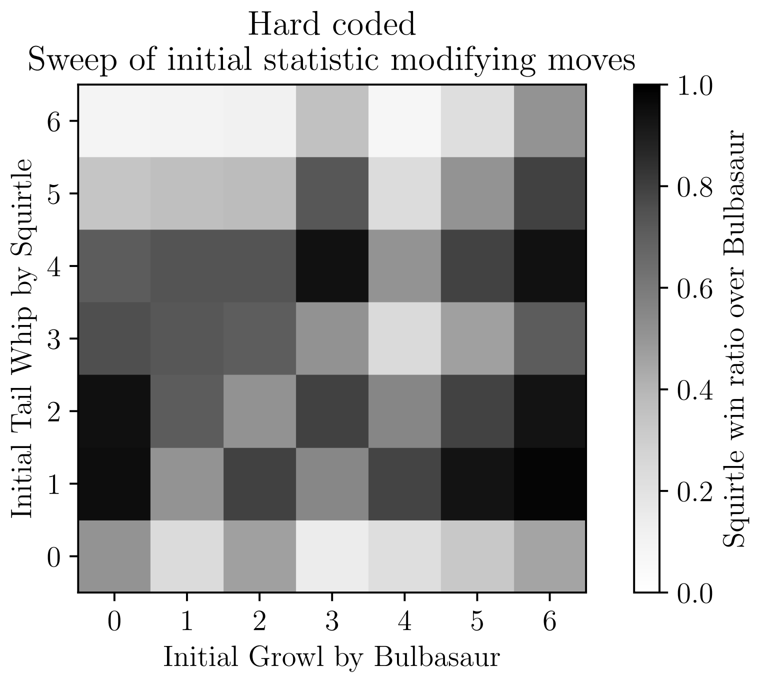

Figure 4.7: The win ratio of Squirtle against Bulbasaur given that both follow an action sequence with a predetermined number of initial statistic modifying moves followed by only damage dealing moves.

Table 4.2: The highest performing action sequences using a specific number of initial statistic modifying moves and their average performances in the initial statistic modifying moves sweep.

Player 1	Player 2	Player 1 initial statistic modifying moves	Player 1 win ratio
Bulbasaur	Charmander	5	19.4%
	Squirtle	4	62.4%
Charmander	Squirtle	1	79.9%
	Bulbasaur	0	98.9%
Squirtle	Bulbasaur	1	78.7%
	Charmander	1	67.2%

Table 4.3: The win ratios of all subscenarios given that both players use their highest performing action sequences.

Player 1	Player 2	Player 1 win ratio
Bulbasaur	Charmander	2.3%
	Squirtle	21.0%
Charmander	Squirtle	67.1%
	Bulbasaur	97.7%
Squirtle	Bulbasaur	79.0%
	Charmander	33.0%

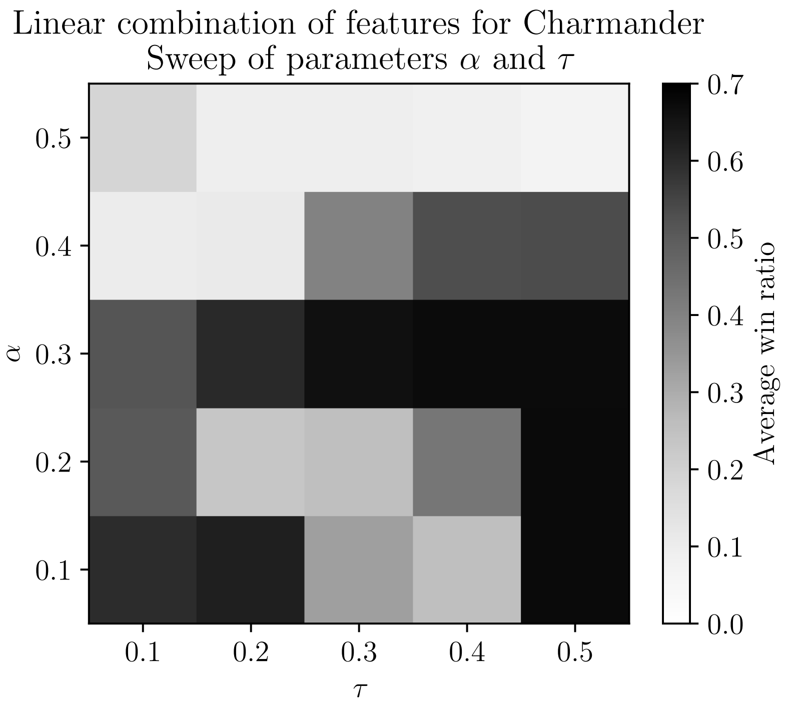

Figure 4.8: The average win ratio of Charmander when played using the linear combination of features AI, trained with different values of the parameters α and τ.

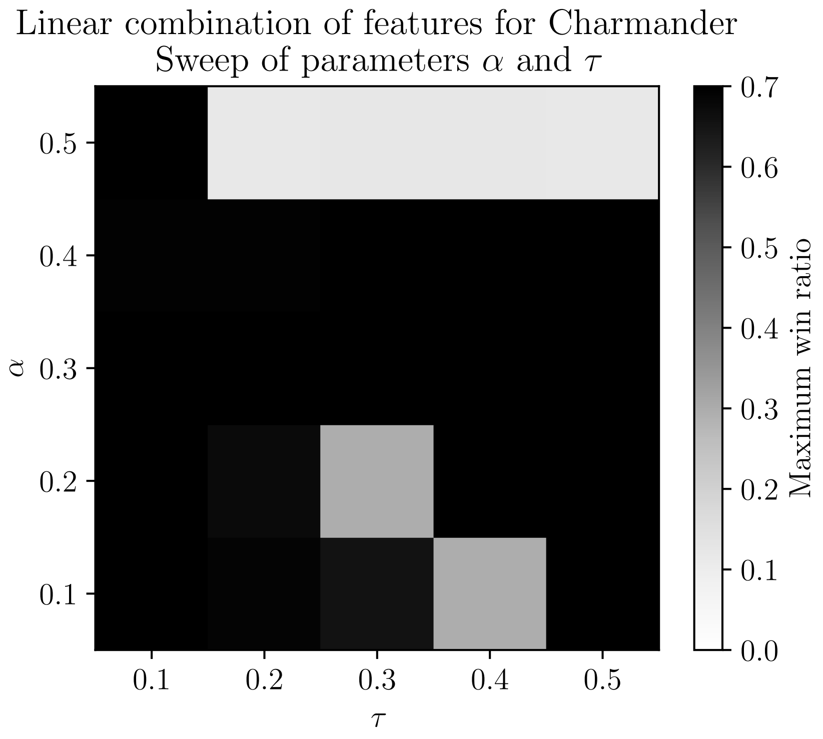

Figure 4.9: The maximum win ratio of Charmander when played using the linear combination of features AI, trained with different values of the parameters α and τ.

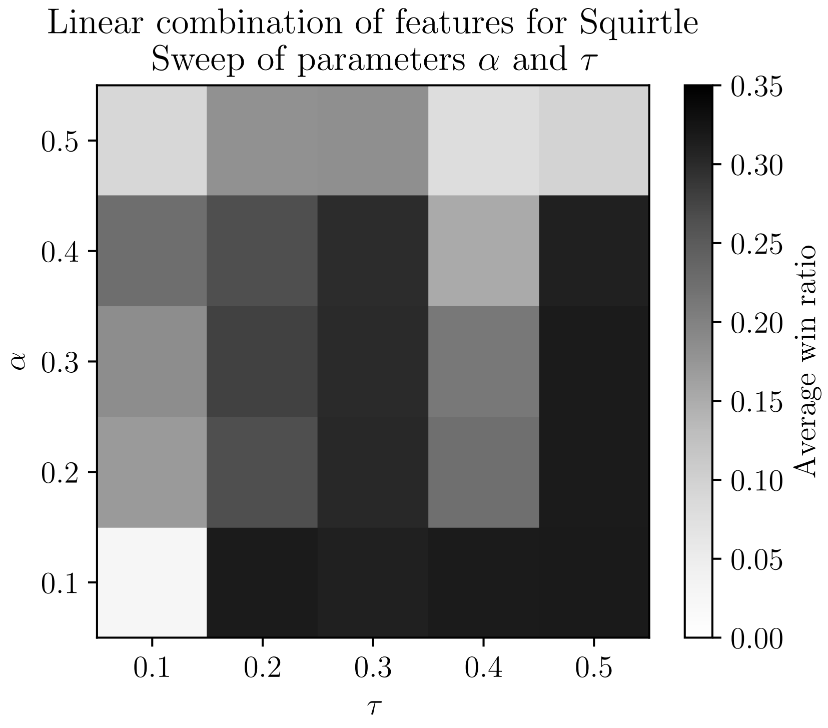

Figure 4.10: The average win ratio of Squirtle when played using the linear combination of features AI, trained with different values of the parameters α and τ.

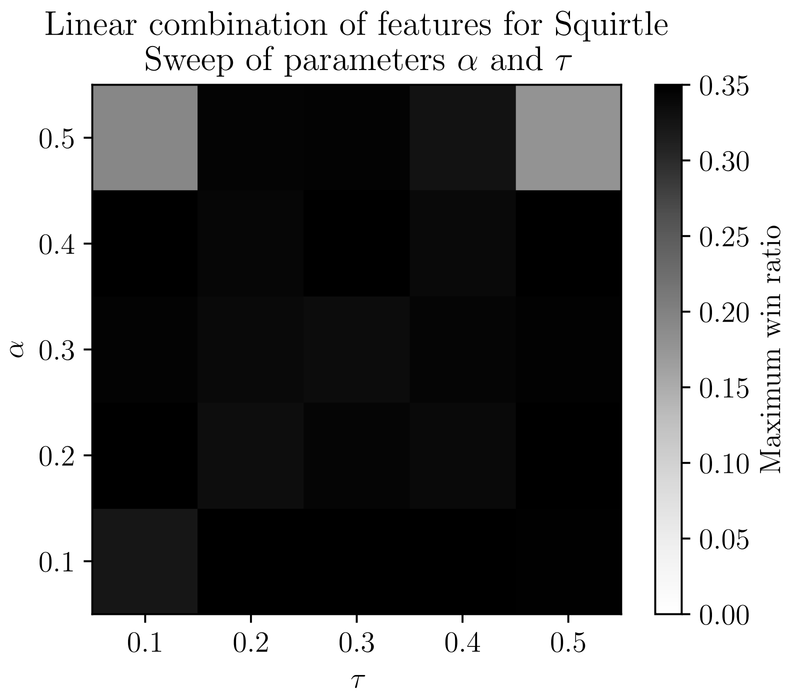

Figure 4.11: The maximum win ratio of Squirtle when played using the linear combination of features AI, trained with different values of the parameters α and τ.

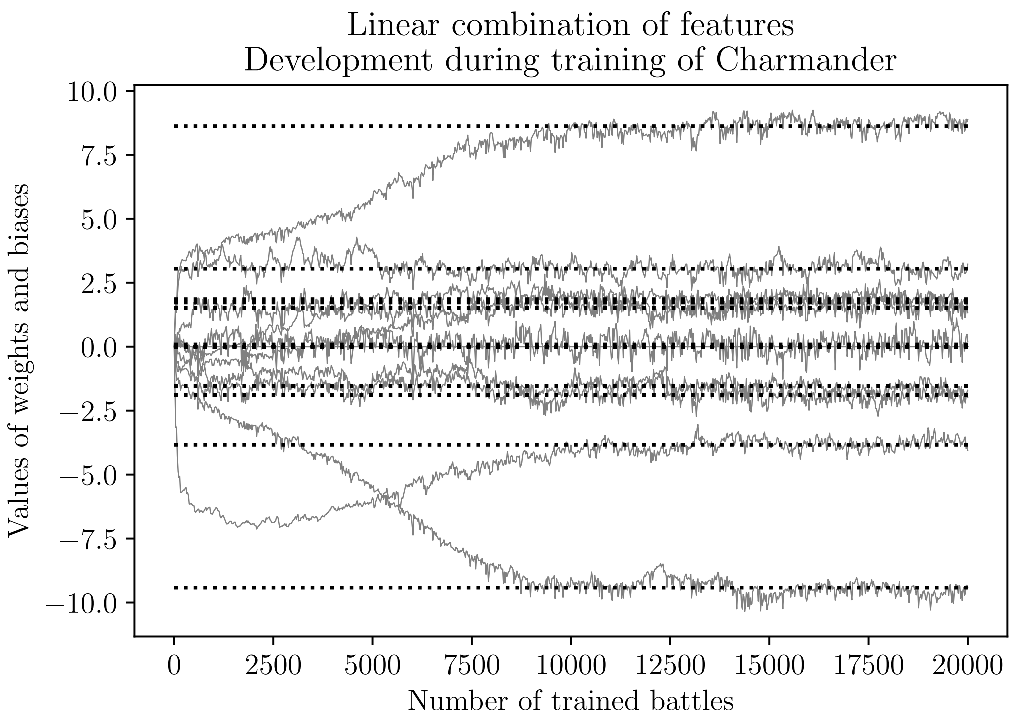

Figure 4.12: The development of weights and biases during training of the linear combination of features AI for Charmander, using $n_t = 2 \cdot 10^4$ as during the parameter sweep. Each line is the value of a weight or bias. The dotted lines represent the average of each weight or bias, disregarding the first 10^4 training battles.

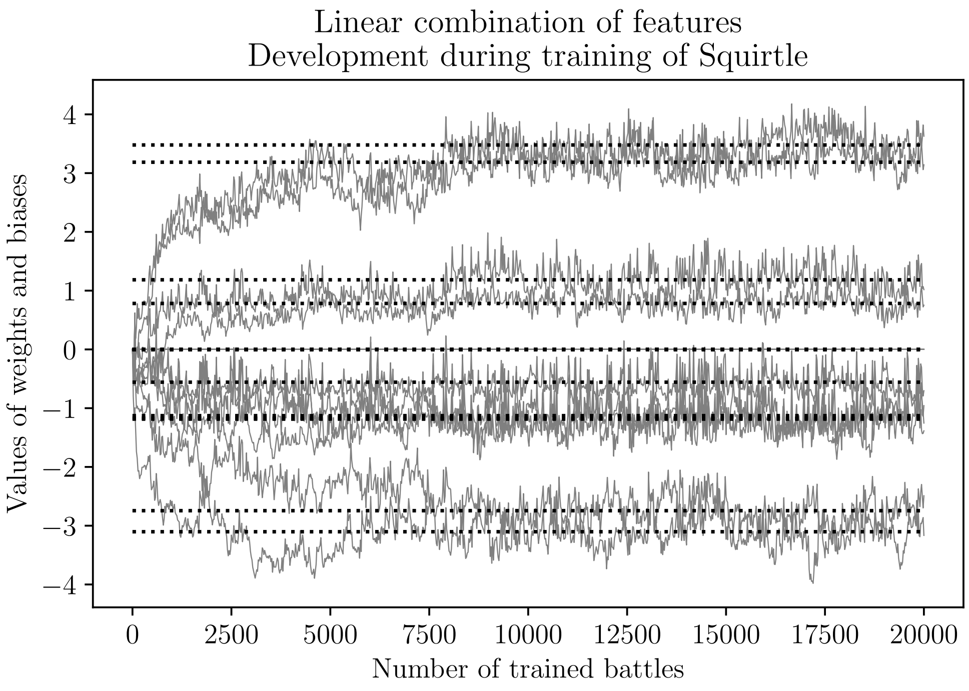

Figure 4.13: The development of weights and biases during training of the linear combination of features AI for Squirtle, using $n_t = 2 \cdot 10^4$ as during the parameter sweep. Each line is the value of a weight or bias. The dotted lines represent the average of each weight or bias, disregarding the first 10^4 training battles.

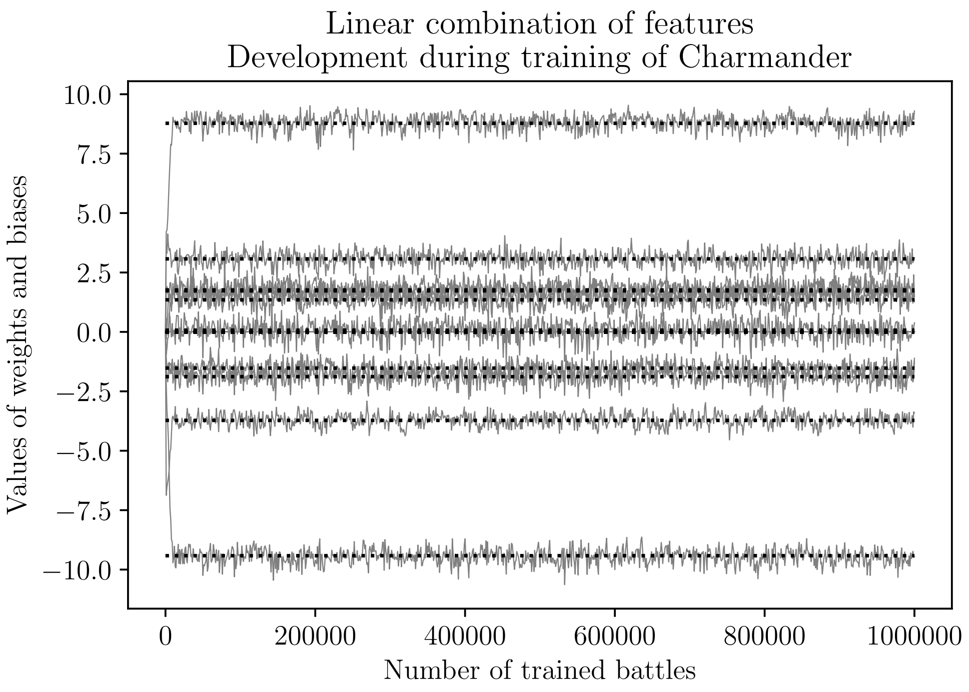

Figure 4.14: The development of weights and biases during training of the linear combination of features AI for Charmander, using $n_t = 10^6$. Each line is the value of a weight or bias. The dotted lines represent the average of each weight or bias.

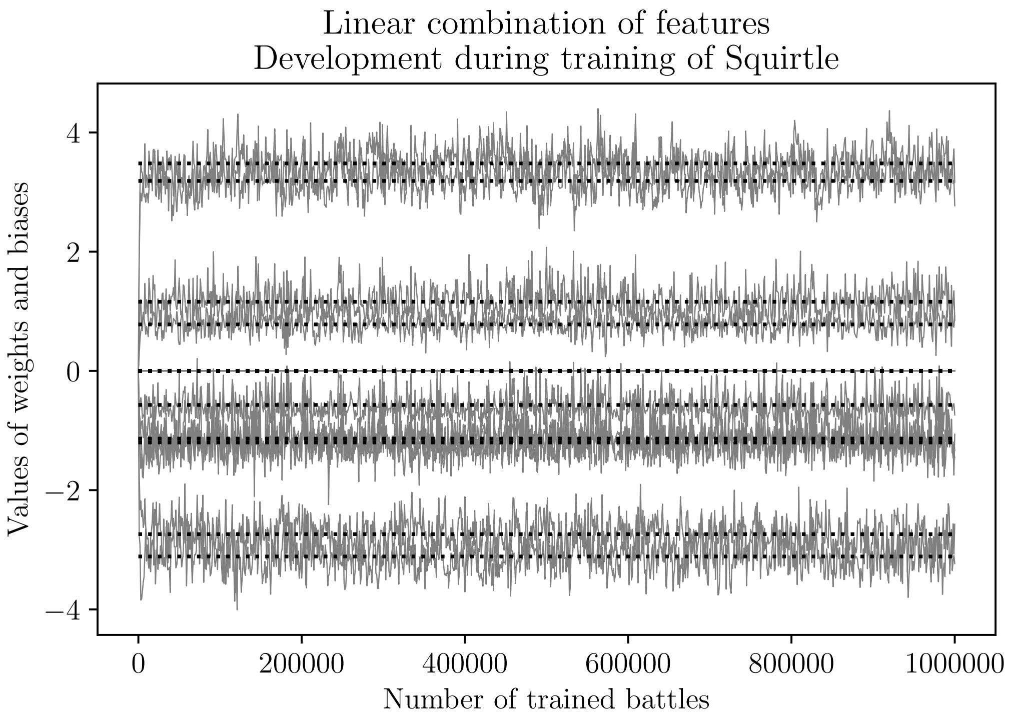

Figure 4.15: The development of weights and biases during training of the linear combination of features AI for Squirtle, using $n_t = 10^6$. Each line is the value of a weight or bias. The dotted lines represent the average of each weight or bias.

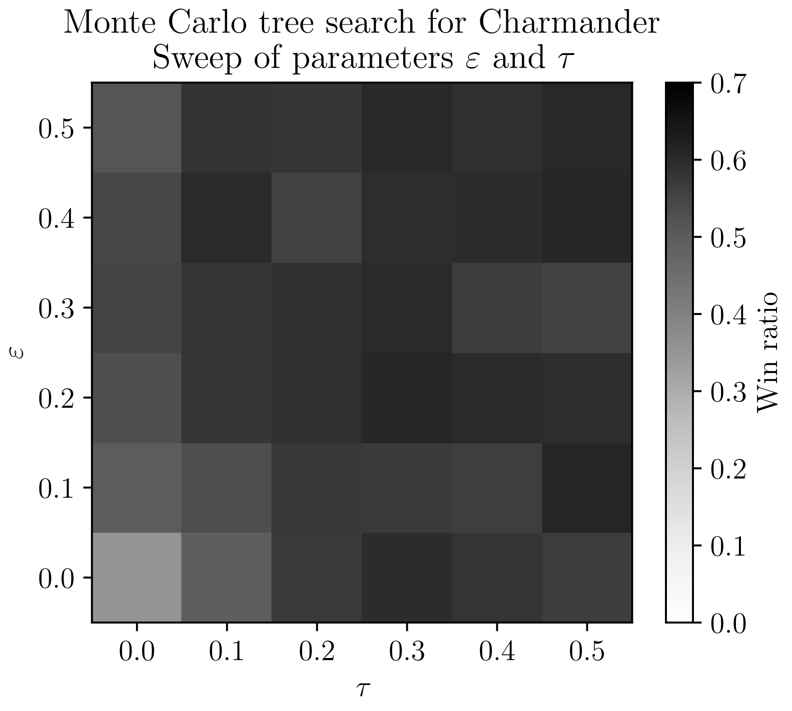

Figure 4.16: The performance of Charmander when using the Monte Carlo tree search AI for different values of ε and τ.

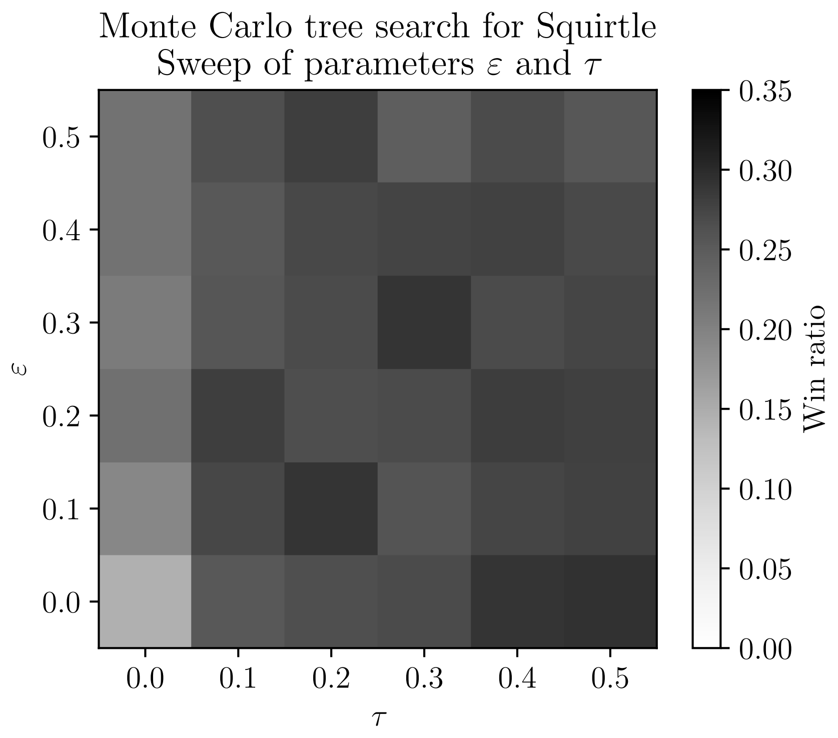

Figure 4.17: The performance of Squirtle when using the Monte Carlo tree search AI for different values of ε and τ.

Table 4.4: The performance measurement of all algorithms using their optimal parameter values. The hard coded A.I. playing against itself is included for reference, since the game isn't balanced.

Charmander	Squirtle	Charmander win ratio
H.C.	H.C.	67.1%
H.C.	L.C.F.	67.9%
H.C.	M.C.T.S.	69.6%
L.C.F.	H.C.	63.2%
L.C.F.	M.C.T.S.	60.1%
M.C.T.S.	H.C.	65.7%
M.C.T.S.	L.C.F.	72.9%

The development of the weights and biases during the parameter sweep training using $\alpha = 0.1$ and $\tau = 0.2$ can be seen in Figures 4.12 and 4.13. Both sets of weights and biases manage to reach their values in the first 10^4 training battles.

The development of the weights and biases when increasing the number of training battles to $n_t = 10^6$ can be seen in Figures 4.14 and 4.15. No further change in values occur after excessive training.

Figures 4.16 and 4.17 show the average performance during the parameter sweep of ε and τ for the Monte Carlo tree search AI. The local maxima are noise and do not reappear when replicating the parameter sweep. However, the minimum at $\varepsilon = 0$ and $\tau = 0$ does appear every time. In addition, slightly lower performance is measured for $\varepsilon \neq 0$ and $\tau = 0$ even when replicating the parameter sweep.

Table 4.4 shows the win ratio of Charmander when using the various AI to play both Charmander and Squirtle. The hard coded AI is the highest performing AI by a small marigin, while the Monte Carlo tree search AI performs higher against the linear combination of features AI than vice versa.

4.3 The Overused Scenario

Figure 4.18 shows the performance and time taken for different values of d in Monte Carlo tree search. It is clear that performance doesn't increase with d, but time taken increases significantly. The pure Monte Carlo search AI was therefore the only search based AI used in the overused scenario.

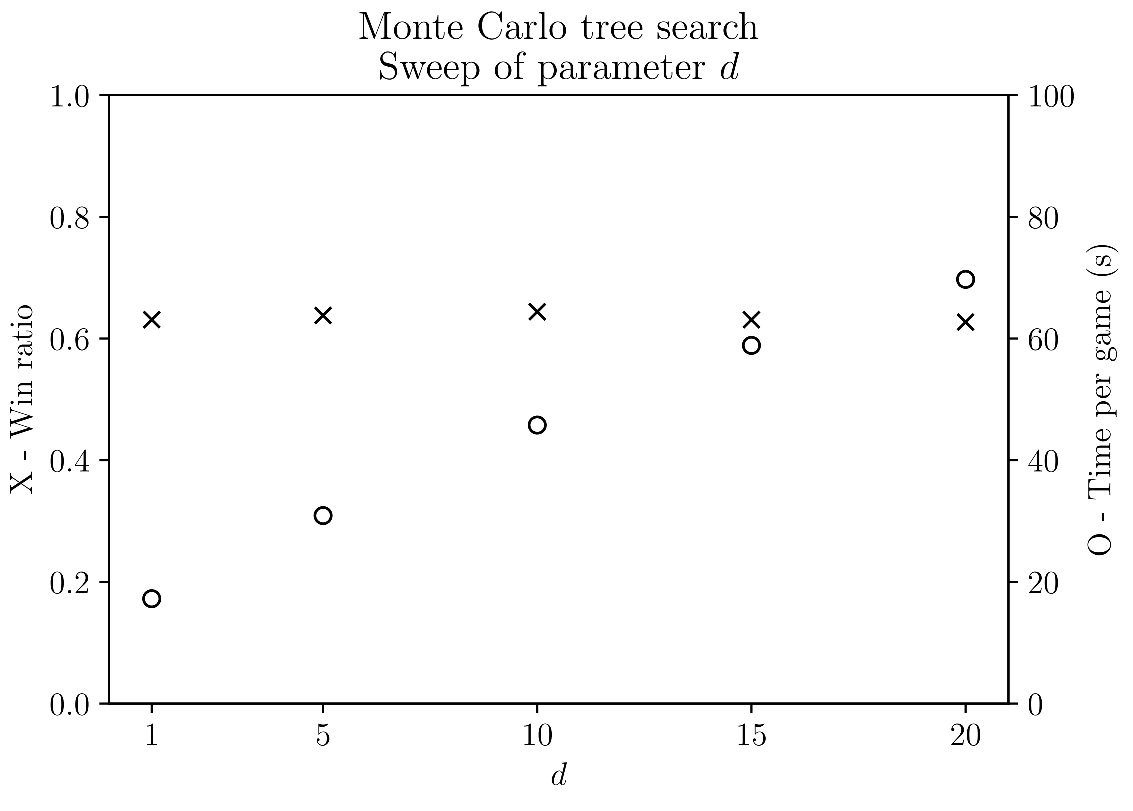

Figure 4.18: The win ratio against the improved random AI and time taken per game for varying values of the search depth d, when using the Monte Carlo tree search AI.

Table 4.5 shows the performance of all algorithms when paired against each other. The completely random AI is the lowest performing AI by far, losing most of the time to all other AI. While the improved random AI wins against the completely random AI most of the time, its performance against the more advanced AI is still low. Both the linear combination of features AI and the pure Monte Carlo search AI wins against the two random AI most of the time, but the pure Monte Carlo search AI wins slightly more and also wins more often against the linear combination of features AI.

Table 4.5: The performance of all AI measured against each other. The percentages don't add up to 100% because some games ended in a draw and some games were aborted.

	C.R.	I.R.	L.C.F.	P.M.C.S.
Completely random		5.6%	1.5%	0%
Improved random	90.7%		7.8%	6%
Linear combination of features	98.4%	91.8%		37%
Pure Monte Carlo search	100%	93%	62%	

Table 4.6: The performance of human players against the pure Monte Carlo search AI. There were no players in category 1 (have never played any version of the hand held Pokémon games).

Category	Participants	Human wins	AI wins	Ties
2	6	6	9	0
3	3	7	3	1

The result of the human benchmark is shown in Table 4.6. The AI won about half of all battles, winning more often against participants in category 2, who didn't have any previous experience with competitive Pokémon battles.

5
Discussion

The discussion is split into three parts in Sections 5.1, 5.2 and 5.3, representing the three Pokémon battle scenarios.

5.1 The Simple Game

Using the linear combination of features algorithm gives perfect performance, equal to using the hard coded algorithm with the optimal action sequence. Averaging was used to determine the values of the weights and biases because they oscillate around what is probably the optimal value due to the stochasticity of the game. The weights and biases approach their values for most parameter choices and do so very quickly. Other ways of defining the features were tried during development which also resulted in perfect performance, but this definition shows more clearly that it might be applicable on other Pokémon battle scenarios.

The performance of Monte Carlo tree search increased with the number of searches n_s. Increasing the depth $d > 15$ probably doesn't increase performance since the length of the game when played perfectly is only 9 rounds, so most searches never reach the depth limit anyway. Perfect performance might be possible for this algorithm as well with enough searches, but this also results in unfeasible simulation times. Setting $\varepsilon = 0$ and using only τ to control stochasticity results in higher performance than doing the opposite. This is probably becuase using τ retains bias toward better choices from $\hat{q}(s, a)$, while ε completely throws away $\hat{q}(s, a)$ to instead use the random AI.

Pure Monte Carlo search is probably underperforming because of the importance of maintaining a perfect action *sequence* throughout the game. The random algorithm probably rarely finishes the search in the best way, introducing a lot of noise to the search results.

5.2 The Starter Scenario

The highest performing action sequence can't be considered optimal play in every situation, due to the intrinsic stochasticity of the starter scenario. If a move misses, for example, another action sequence from that point on might perform higher. It is however the highest achievable performance for the hard coded AI, because it uses a fixed predetermined action sequence.

While using a high number of training battles n_t when training the linear combination of features AI is generally better, the run time for a parameter sweep with too high n_t quickly becomes infeasible. Some parameter choices can't reach high performance at all, which might be a sign of weights and biases diverging to infinity. The low average win ratio of other parameter choices when a high maximum win ratio can be reached might be a result of the weights and biases not having enough time to find their optimal value properly, or the oscillations being too large, requiring the average taken over more points. Figures 4.12, 4.13, 4.14 and 4.15 do show oscillation around some optimal values, indicating that performance can't improve further with this algorithm.

The problem with too long run times become even more severe when using the Monte Carlo tree search AI, as the algorithm require many games being played until their end, every round of the game. The parameter sweep of ε and τ, for example, took about 36 h to complete. While increasing the number of games used in the sweep by a factor 10 or even 100 would smooth out the noise, doing a five month long parameter sweep is out of the question. As the values of ε and τ didn't seem to matter as long as $\tau \neq 0$, $\tau = 0.2$ and $\varepsilon = 0.1$ was chosen to keep stochasticity relatively low. The search depth $d = 10$ was chosen to be about the length of a game, and $n_s = 1000$ was set as high as possible without making the run time infeasible.

5.3 The Overused Scenario

The nature of the overused scenario prevents using action sequences of any length larger than single actions successfully. This is partly because both players are able to switch their active Pokémon at any time, essentially being able to switch between 6^2 different subscenarios every round. The increased layers of stochasticity can also change the dynamics, as key moves such as Sleep Powder have a relatively high chance to miss. As a result of this, the hard coded algorithm can't be implemented because a good action sequence lasting the entire game can't be constructed.

The nature of the overused scenario is probably also the reason for the failure of the Monte Carlo tree search AI. But there's also the problem of rapidly growing search trees. The number of possible action sequences is

$$n_{\mathbf{a}} = \prod_{r=r_0}^{r_0+d} n_{a,r}$$

where r is the round starting from r_0, d is the search depth and $n_{a,r}$ is the number of allowed actions for round r (at most nine and very rarely less than four). This means that an infeasibly large n_s or t_s is required to search efficiently.

The parameter choices for the linear combination of features AI was chosen based on the results of the simple game and the starter scenario, but with both α and τ set lower to decrease fluctuations. The rather complicated knock-out training setup was used because the results of the training would differ greatly every time, and the weights and biases never seemed to approach any specific value. A very long training session with $5 \cdot 10^7$ initial training games and averaging over another $5 \cdot 10^5$ training games was carried out (over one weeks time of training), but the performance was lower than the best performing set of weights and biases from the knock-out training setup, at about 80% win ratio against the improved random AI.

This probably means that the linear combination of features algorithm is too simple for this scenario. Using a hidden layer in the algorithm could possibly increase its ability to handle the complexity of the game.

A lot of time was spent optimizing the run time of both the game itself and the pure Monte Carlo search AI. When using $t_s = 10\,\mathrm{s}$ on a normal computer, about $n_s = 7500$ search simulations are carried out for each round at the start of a game and about $n_s = 10^5$ close to the end of the game. This is because when the end of the game approaches, both the number of available actions and the remaining length of the game decreases. An interesting side effect of this is that the performance of the pure Monte Carlo search AI increases closer to the end of the game. Another side effect is that a human player can decrease the performance of the AI by simply using a bad computer to run the game.

The results of the performance measurement against human players should be taken with a grain of salt. An observed general trend is the improvement of human players when playing more than one game. The reason might be that the participants learned how the AI plays, or that they improved after playing the first game.

6
Conclusion

For a simple Pokémon battle scenario such as the starter scenario, it seems that most of the algorithms tried in this thesis are somewhat successful. But none of the algorithms could play on par with the hard coded AI when using the highest performing action sequence. It goes without saying that the hard coded AI achieves perfect performance in any game where the perfect action sequence is known. But when complexity increases the perfect actions sequence becomes hard to find, and if the game requires the player to react to a changing environment, such as switching Pokémon in the overused scenario, any predefined action sequence becomes useless.

The linear combination of features AI managed to achieve perfect play in the simple game, where the only stochasticity is the turn order of the players and there is only one fixed optimal action sequence. When the complexity of the Pokémon battle increases however, such as for the starter scenario or the overused scenario, it doesn't perform as well. For the overused scenario, its weights and biases don't even converge, indicating that it might be too simple of an algorithm for that scenario. The complexity of the algorithm could be increased by adding hidden layers, possibly enabling higher performance in more complex scenarios.

The various Monte Carlo search algorithms on the other hand, managed to play more evenly when the complexity of the game varied. It didn't perform too bad in the simple game (and probably would improve further with increased search time) and it was the best performing algorithm in this thesis for the overused scenario. Even against human players, it managed to win about half of all battles, arguably playing on human level. But in the end, the sample size isn't big enough to draw any real conclusion.

From all the different algorithms tested on all scenarios, the general conclusion is that the choice of algorithm should be made with consideration to the complexity of the game that is to be played. If the game is too complex for the algorithm then its performance will be subpar, and if a complex algorithm is used for a simple game then it might not be able to play consistently and it would use more time for training or searching than is needed.

6.1 Future research

For future research on this topic, I recommend to improve the algorithms by trying different combinations of them. The search algorithms could be used while training the linear combinations of features for example, or the linear combinations of features could be used while searching. In addition, linear combination of features should be extended with one or more hidden layers, which might help with the more complex game that is Pokémon.

A more general advice is to spend time optimizing the run time of the game. This applies to any of the two prominent algorithms in this thesis, and any game. A faster game will enable more searching and more training in shorter time, which is crucial for performance.

Bibliography

[1] J. McCarthy, M. L. Minsky, N. Rochester, C. E. Shannon, "A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, August 31, 1955", AI Magazine Vol. 27 No. 4, 2006-12-15. [Online] Available: https://doi.org/10.1609/aimag.v27i4.1904

[2] Y. Leviathan and Y. Matias "Google Duplex: An AI System for Accomplishing Real-World Tasks Over the Phone" ai.googleblog.com https://ai.googleblog.com/2018/05/duplex-ai-system-for-natural-conversation.html (Accessed: 2019-02-19)

[3] S. Baum, "A Survey of Artificial General Intelligence Projects for Ethics, Risk, and Policy", Global Catastrophic Risk Institute Working Paper 17-1, 2017-11-12. [Online] Available: https://ssrn.com/abstract=3070741

[4] D. Silver et al., "Mastering the game of Go with deep neural networks and tree search", Nature Vol. 529, pp. 484-489, 2016-01-28. [Online] Available: https://doi.org/10.1038/nature16961

[5] "The Future of Go Summit". events.google.com. https://events.google.com/alphago2017/ (Accessed: 2019-03-06)

[6] D. Silver et al., "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play", Science Vol. 362, Issue 6419, pp. 1140-1144, 2018-12-07. [Online] Available: http://doi.org/10.1126/science.aar6404

[7] "Business Summary". pokemon.co.jp. https://www.pokemon.co.jp/corporate/en/services/ (Accessed: 2019-01-05)

[8] "Technical data". nintendo.co.uk. https://www.nintendo.co.uk/Support/Game-Boy-Pocket-Color/Product-information/Technical-data/Technical-data-619585.html (Accessed: 2019-03-07)

[9] "Main Page". bulbapedia.bulbagarden.net. https://bulbapedia.bulbagarden.net/wiki/Main_Page (Accessed: 2019-03-25)

[10] "Smogon University". smogon.com. https://www.smogon.com/ (Accessed: 2019-03-25)

[11] "UCL Course on RL". cs.ucl.ac.uk http://www.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html

[12] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Cambridge, England: MIT Press, 1998.

[13] C. Szepesvári, *Algorithms for Reinforcement Learning*: Morgan and Claypool, 2010.