POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Study on reinforcement-learning-based decision-making and planning in the context of non-deterministic scenarios

Supervisors	Candidate
Giovanni SQUILLERO	Alessandro FRANCO
Alberto TONDA	

April 2023

Summary

Reinforcement learning (RL) is a useful tool to allow algorithms to perform unsupervised training in order to achieve a goal [1]. In the context of games, the final goal is to successfully complete the selected game or level. Games can be employed as a simple way to test algorithms thanks to their clear objectives, rules and sandbox, allowing players to elaborate strategies and tackle new challenges in many different forms: competitive or cooperative, cards or tabletop, the choice is very wide and many different games may be chosen for different reasons. Along with the times and technological advances, new challenges became available under the form of video games, providing new environments for learning algorithms and, from Atari games [2] to Doom, many different titles have been the test bench for reinforcement learning [3].

In order to provide a significant challenge, games should require a certain level of skill to successfully play; however, depending on the genre, the set of skills may vary: for example a strategy game needs players to be able to plan in the long term to succeed, while action-based games are a test of reflexes and so on. Additionally, the player may be bound to perform actions within a limited time or under uncertain conditions, such as in simultaneous games (for example Rochambeau) where both of these aspects are included: each player needs to select a move without knowing the opponent's, meaning that both players have the chance to pick a very favorable move at the expense of the other, however none will know until the move is performed.

The key points for a game to have for this research are, apart from the turn-based mechanism, required to have a clear separation of different states, some degree of randomness intrinsic within the game's sandbox, meaning that the same sequence of actions may not bring to the same end result. This detail is needed to deploy an algorithm that must adapt to new scenarios and, in this way, the code may be put in a condition similar to a real-life decision process, where rarely things have ideal reactions to an agent's decisions. Backgammon for example has been part of thorough studies in the field of reinforcement learning and results have shown that competent agents are very well achievable with a suitable state recognition and training.

The combination of the previous points should be able to produce an environment

that is not quite the same as a real-time scenario but it should provide a close approximation.

The goal of this project is to develop an algorithm capable of playing Pokémon [4][5], one of the most famous and grossing games in the industry, which covers many of the characteristics described above:

- A single game of Pokémon revolves around defeating the opponent's team in a turn-based environment, players must devise a strategy using the selected Pokémons, since Pokémon is a simultaneous game. This system requires agents to develop strategies taking into account the interactions with other players which will affect the way states are recognized and play a significant role in improving performance.

- Pokémon species are described by their name and each species has a unique statistics distribution that determine how effective they will be during the game (health, attack, defense, speed etc.). Base values are publicly known, however according to players' needs and preferences, the final value can be tweaked up to some extent, resulting in the exact value for those statistics to be technically unknown. However, since the total amount of custom values cannot exceed a game-defined rule, players can still guess the final values.

- Each Pokémon can use up to four unique moves with different levels of power and side effects, unknown to the opponent until used during the game, which obviously reveals them: players start therefore with imperfect information on the opponent's team.

- Moreover, moves are associated to a percentage-based accuracy value that determines how likely is a move to successfully land on an opponent under standard conditions, which results in a non-deterministic result to each move that has not maximum accuracy.

- Players can also elect to swap the active Pokémon to another one from the team; however, damage suffered and some of the penalties that can be inflicted are not restored unless special conditions are met: the previous actions therefore affect the following turns in different ways to be accounted for when planning future moves.

These rules suit well the context of a Markovian Decision Process (MDP), to which the algorithm must find a policy able to take into account the different aspects of the game and elaborate a strategy. A reinforcement learning algorithm should be set to gather information between turns based on the effects of both players' moves and, with enough training, random variables would be taken into account when planning a strategy. Moreover, two instances of the algorithm playing

competitively [6] may result in a performance improvement with the increasing number of games, given that moves are selected with the intention to win on both sides to provide realistic reward values for states.

The results should indicate the impact of randomness and other uncontrollable variables to the overall training and how effective the algorithm is when evaluating moves and strategies.

To verify the effectiveness of the code, multiple tests were set up to mimic the different skill levels of opponents online that any player may encounter, from newbies to experienced players. The simulations will be performed with modified versions of the project's algorithm [7] in order to adapt to the desired behavior, by selecting random moves, greedy moves and by resetting the initial experience produced by training games respectively to represent new players and inexperienced opponents. Moreover, the test involving the untrained version acts as an indicator to determine how the code adapts to a player reacting optimally to certain situations: given that the training games do not provide an encyclopedic knowledge of the game, the trained version will play the role of an experienced player by performing close to optimal moves and developing a coherent strategy along the games, which serves as a close approximation to what competent players do compared to the inexperienced ones.

Eventually, the code was able to elaborate strategies according to the opponent it was facing with variable win rates depending on the other player's habits: for example a trained code struggled more against a random opponent than a greedy code, where win rates are noticeably different with 56.6% and 60% respectively out of a batch of a thousand test games. However, the results proved that additional performance upgrades can be applied to the algorithm [8], since it suffers from some of the well known problems of this kind of approach, namely disk space requirements and issues when recognizing and handling similarities.

Overall the project proved that this approach is a reasonable solution to the problem at hand and it may prove useful even in different contexts, nevertheless additional improvements are available for better performance and to provide a more accurate reactions in similar scenarios.

Table of Contents

Chapter 1

Background and Problem Definition

1.1 Machine Learning

Learning is essential to any living being: from animal cubs to children, any youngling is presented with new tasks to perform or a new skill to master in order to grow and be successful. Many techniques have been developed to undergo this process where the student must understand the issue at hand, elaborate a course of action and apply the strategy to evaluate its effectiveness: children may be given puzzles while young tigers may fake ambushes on their parents but the results are the same, learning.

Nature has many way to convey new information across generations but the next step for some computer engineers is teaching an algorithm how to perform a task. While some may think this as a modern problem, machine learning research dates back to the '50s with Turing's "learning machine" and the first algorithms to play checkers, up to the more recent Go algorithms (Alpha Go) and neural networks applied to a wide variety of different fields.

In order to replicate a similar process to the one used by biological entities, a computer must be presented with some key elements to undergo the learning process:

- Context, or environment: machines must be able to probe and interact with the environment they are deployed in, namely by measuring its variables and the effect of actions over them, allowing the code to understand and analyse their performance. Moreover, the code must understand its available actions and how to perform them during the process.

- Model: the code may be given an initial model to follow when learning the

ropes of a new task; one can imagine the model as a map to have a general idea of the position and direction in an environment. A model however is not mandatory and model-free algorithms have the advantage of unbiased learning, in particular when approaching a task for the first times.

- Representation: the available information gathered from the environment and previous experience must be represented in an effective way to support the code when learning and producing better results. Data representation is a very important aspect when dealing with machine learning, since it determines how detailed the available data will be and consequently the amount of disk space required to store this information.

- Feedback: a fundamental aspect of learning, since it determines which actions are suited to the context and which ones are not. Algorithms will use this data to adapt accordingly in many different forms, from rewards to fitness, resulting in a more accurate response to known scenarios.

The developer's goal is to provide software with the right tools to learn how to perform the desired task and, possibly, to exploit the gathered experience for other similar tasks.

1.2 Reinforcement Learning

Reinforcement learning is a field of machine learning that aims at training an agent to make intelligent choices by means of rewards; the code is required to build a strategy in order to obtain the maximum overall reward[1], rather than settling for the best local decision at any given state (greedy approach).

This approach is similar to training applied to pets: the subject is given a command and it is given a treat if it responds correctly, or a punishment when the goal is not achieved. Eventually, subjects start to behave according to the desired pattern without the need to request it beforehand [1].

This solution is usually employed when an algorithm is set to produce a solution for a problem with little to no supervision, relying solely on the results gathered during run-time from the environment; therefore no previous information is required for the code to properly work.

Since supervision is not strictly required, the agent is not given instructions on the available moves and how they affect the states it may find itself in: this is a key aspect of the algorithm, allowing the agent to perform unbiased decisions when

[1] usually is associated to the correct result of the proposed task, in our case it means that the agent won a game

elaborating a strategy and, possibly, to evaluate more moves than a human agent would normally do. Combined with a faster computing speed than the average human, the code is allowed to perform a large amount of tries to determine the optimal policy for a given task, without the need for detailed explanations about the requirements and the available tools at disposal, effectively starting at a great advantage compared to human players.

In the learning process, the algorithm has two mutually exclusive paths it may choose from: exploration, the action that provides the code with new states and new move values for future reference, and exploitation, which conversely tests the previously gathered knowledge and updates its values with data from following states. Since initially the code has virtually zero knowledge of the environment, the algorithm should be able to balance these two paths in order to develop a consistent and useful policy. Due to how tasks usually vary, both in terms of states and available moves, algorithms must have a reasonable way to balance these two paths, the most common of which is the ε-greedy approach, that determines the chance of the code following the exploitation path with probability $1 - \varepsilon$, very simple but quite effective solution to provide a reliable performance in many different environments. Generally, these aspects can be customized via global parameters[2] that determine how the code chooses between the two, as well as the extent to which new values affect previously saved policies.

The reason behind the selection of this kind of algorithms can be found in how they approach unknown moves and data from the environment, allowing for a faster learning process, especially under conditions of non deterministic moves and interactions.

1.2.1 Episode

A task usually has a starting condition, a goal and a set of actions required to be completed: these elements combined are considered an episode in reinforcement learning. An algorithm may undergo a large number of episodes to learn the required skill to complete the given task, each time gathering new data on how the environment reacts to each action, thus effectively reproducing the process of learning used in many other contexts: for example, when studying for a math class, students are given homework consisting in some exercises to be solved; each of them is an episode they are required to complete and the final result indicates the correctness of the actions performed during the episode. In this way, both the student and a reinforcement learning algorithm can understand how they arrived to a solution and determine if the episode was successful or not, providing an

[2] also called hyper parameters, ε for example

evaluation of the actions performed along the way.

Tasks that do not follow this model are called continuous tasks and usually differ from episodic environments due to the lack of ending state, effectively requiring a different approach when handling rewards and policy evaluation, mainly due to the timings at which rewards are assigned.

1.2.2 States

During an episode, the code will need to perform many different actions to complete the task; a further division of the problem can therefore be achieved via the use of states. States usually describe the situation of an episode at a given point in time, usually before the agent's action, so that the resulting action sequence can be evaluated accurately. Following the example of the student described above, states can be identified in the various steps when solving an equation, the most common point in which the student is required to apply theorems in order to simplify the equation. Each action will move the episode to a new state and, depending on the result, some action may appear more effective than others to achieve the end goal.

1.2.3 Rewards

Rewards are a way to evaluate how effective moves are, following an equation that takes into account many different factors, ranging from the effects on the state to the effectiveness of following moves, resulting in a simple form of evaluation for both the move and the pattern. Due to the fact that many moves do not change substantially the state, there is a significant chance that the reward for a given move is zero: however, with the increasing number of episodes (and thus experience of the states) the agent will update values based on moves and values that lead to successful conclusions for the task, effectively settling the values to more accurate predictions. Reprising the example of the student seen in the previous points, the final reward is the grade of the student when handing out their homework to the teacher, which will affect the outcome of the final exam due to the fact that the successful actions when completing the assignment will be the best option when taking the test.

1.2.4 Markovian Process

A Markovian Decision Process (MDP), extension of the Markovian chain described by the homonym mathematician Andrey Markov, is a model that describes discrete-time stochastic control processes, which is the most effective way to identify the kind of problem many turn-based games pose. The key aspect to keep in mind is that actions lead to different states following a determined probability function: in

the context of games, the process rarely loops around the same nodes but it can happen.

MDPs can be represented in directed graphs where nodes are the different states of the process whereas edges are the available actions associated with the probabilities to follow the arc to the pointed state. When considering the entire process, the code will need to evaluate the route to reach the end state, keeping into account the probabilities of the various actions as they will impact the final effective route and time of traversal. A possible example of this scenario can be found in how people in a big city plan a route to reach their destination: some inconveniences may happen, such as road blocks or heavy traffic in some areas of the city, more frequently depending on the time, affecting the final path choices.

Considering a game environment, the probabilities are a good approximation of random events during the game, such as dice rolls or card draws that, since they cannot be predicted without risk, may have an impact on the overall strategy, forcing updates and adjustments to complete the process successfully. When dealing with non deterministic actions, many approaches can be viable depending on the overall state of the game: when in a losing position, it may seem easier to resort to risky moves compared to being in a more favorable state. Overall, reaching balance between risk and reward is often the end goal for players when planning a general strategy going into a new game but it is usually a guideline taking into account various different opportunities along the way. For shorter games, it is possible to have a more accurate plan to deal with random occurrences in a methodical way, taking educated guesses and controlled risks in order to limit the impact of other random events: for example, holding on high value cards in a car game or keeping count of played cards help players maintaining a clear view of the game state.

Markovian processes are widely used as test bench for various forms of machine learning and dynamic programming algorithms, due to the clear definition of states and their stochastic property. Moreover, being able to clearly define states is invaluable when dealing with algorithms, since it may result in a much more accurate response once experience starts playing a significant role when learning.

Performing several test simulations or deploying the algorithm in a real-life environment provides the code with a larger amount of data, the "experience", that will be recorded and stored for future lookup, eventually leading to a more accurate analysis of each game state and more elaborate strategies for complex situations. However, with enough training and accurate design, software may be able to learn effective strategies by itself, exploiting the processing speed of computers to perform a larger amount of games in preparation of matches against human players [6].

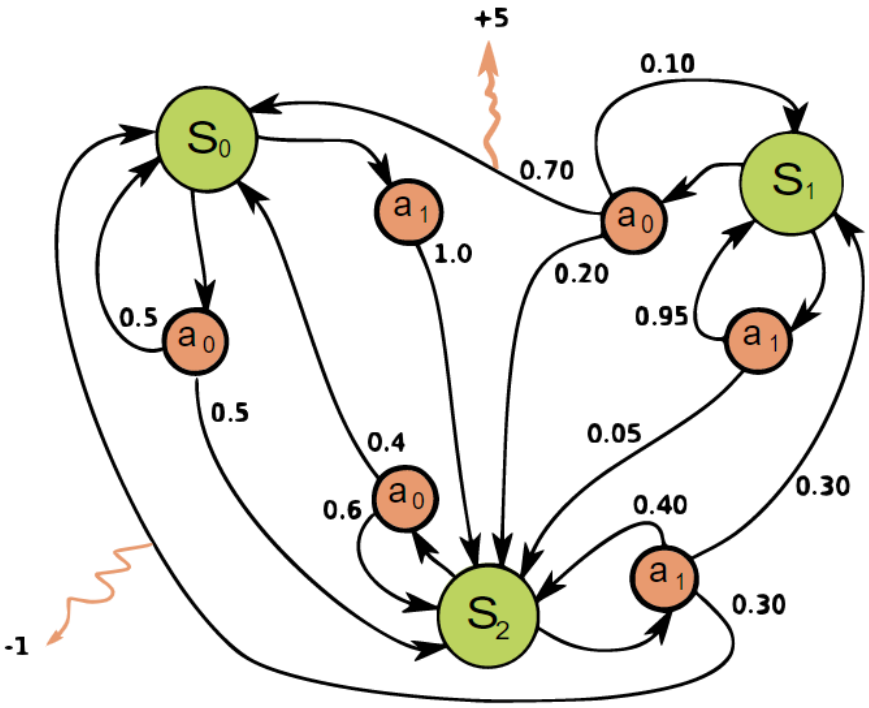

Figure 1.1: Example graph of a MDP. [3]

1.3 Algorithm overview

When designing the code to tackle this project, many different approaches were analysed to evaluate the most efficient way to learn how to play the game and elaborate effective strategies, basing the decisions on the learning process.

The most important features for the final selection are the ability to perform unsupervised learning, in particular in environments where actions have unknown side effects[4], as well as good performance in terms of memory usage for storage and execution time required to elaborate a strategy and select a move.

Overall, many algorithms were up to the task; however, the final decision was based on the performance of the code across different games, keeping in mind that

[3]File:Markov Decision Process.svg by waldoalvarez is licensed under CC BY-SA 4.0.

[4]in games, rules usually explain side effects, but writing code to actually be able to understand written language is a much more sophisticated project

the final review was conducted by analysing the data structures that each algorithm produced as "experience", as well as by evaluating the games themselves compared to matches played by human players.

1.3.1 Rule-based algorithms

Rule-based algorithms are a very simple way to approach this problem: given the state of the game and the available moves, the code is able to roughly evaluate the effectiveness of each action from parameters. This approach is often paired to learning algorithms to identify and add new potentially useful rules to the system. In the context of this project, a slight customization was added to prevent predictability, due to repetition of moves in similar states, a common issue in this kind of algorithms: moves were randomly selected among the available ones using as weights the estimated values of actions; in this way, the number of options the code would select at any given turn was increased to a pool of three or four. However, since the initial version of the algorithm did not include any rule update policies, due to the very large amount of parameters to keep track of and the different random effects of moves, this solution proved to be much less effective than anticipated.

1.3.2 Temporal Difference Learning

Temporal difference (TD) learning algorithms are a common choice when dealing with model-free reinforcement learning. They use existing action data to predict rewards for current and future actions, allowing for a totally unsupervised self-learning mechanism, since moves in the game have additional side effects that have to be discovered by software[5]; this approach proved to be particularly successful to better understand the real worth of the available actions at any given state.

Algorithm 1 Idea behind reinforcement learning

```
table ← values[m][n]  ▷ m (number of states) by n (number of available moves)
state ← 0
while state is not terminal do
    Select one of the available moves
    Observe consequences of actions
    values[state][action] ← new_value                ▷ Value gets updated
    state ← new_state
end while
```

[5]the alternative would be to read the description of the move

The key for this approach lies in previous experiences, as multiple solutions are attempted to complete the task, more data about which actions are effective or not is gathered and the model becomes increasingly more accurate.

The reason behind this improvement is to find in the way rewards are assigned: the equation that gives a numeric value to actions must take into account the changes in the perceived state, as well as previous known values and how accurate they were, both for the current state and the following.

$$V(S_t) = V(S_t) + \alpha[G_t - V(S_t)]$$

The general idea is similar to other methods of reinforcement learning, such as Monte Carlo search [9], that require the code to go through many different states to gather information about the actions and then proceed to evaluate the values to correct the initial observations with the value $G_t - V(S_t)$, that represents the observed error. The error is then used to adjust the move reward and, with time and simulations, the values for different moves will move towards their real worth, allowing for an accurate estimate of each move when deciding the next actions [10].

However the Monte Carlo search requires the algorithm to complete the episode before evaluating its performance, forcing full executions before realizing that a policy is detrimental to the final goal.

Instead, temporal difference algorithms use information gathered during state explorations to update data about moves during simulations, while taking into account the following states, therefore not completely sacrificing any long term benefits of a strategy. In particular, the special case of temporal difference learning TD(0) updates values on the go, considering the following state when updating values, progressively improving the strategy as it executes.

$$V(S_t) = V(S_t) + \alpha[R_t +_1 +\gamma V(S_t+_1) - V(S_t)]$$

Following this equation, rewards are updated each time the same move is selected, in particular, some factors are significant: first, the old value is not completely replaced but rather updated with the new information acquired, namely the new reward R, together with the estimate for the next state $\gamma V(S_t+_1)$, is a way to evaluate the move and its effects towards the completion of the task, to which the previous value $V(S_t)$ is subtracted, resulting in a error evaluation that helps correct the initial value.

Finally, γ and α are two parameters that can be customized to improve performance along the number of completed simulations: they represent respectively the weight of new values over the old ones and how much of the error will affect the final value. They can either be updated automatically with the number of episodes or manually, depending on the performance of the agent or to test different configurations in a given scenario.

1.3.3 Q-learning

When discussing reinforcement learning and temporal difference algorithms, probably the most known among them is the Q-learning: it relies on the same premises described above but it adds to the reward the value of the best move for the following state. This approach was particularly useful due to how effective Q-learning is for Markovian processes. With this algorithm and a refined reward value assignment the code reached a satisfying knowledge of the game and it was able to perform significantly better than the previous versions.

In particular, Q-learning exploits previous experience of future states and actions to test different policies to reach a successful solution by trying to follow actions that will provide the maximum cumulative reward.

Algorithm 2 Example of Q-learning

```
table ← values[m][n]  ▷ m (number of states) by n (number of available moves)
state ← 0
while state is not terminal do
    N ← random value      ▷ usually between 0 and 1 with uniform distribution
    if N ≤ ε then
        Select randomly one of the available moves
    else
        Select the best of the available moves (max value in table[state])
    end if
    values[state][action] ← new_value                    ▷ Value gets updated
    state ← new_state
end while
```

The main difference between the previous algorithm and Q-learning is that it can directly evaluate the strategy elaborated by the agent across simulations, by selecting the best rewarding actions at any given state expecting an effective move.

In normal conditions, the code would be able to replicate the results from previous simulations and achieve the best possible outcome with the available data; however, in the context of an MDP, the stochastic nature of the process may provide additional information on the moves, forcing the algorithm to update its data and potentially lowering the original values for some of them. This possibility may occur either when moves have side effects that disrupt the original plan or when the environment reacts differently due to some random occurrences, described by the MPD but unknown by the code until it stumbles upon said random reactions.

As an additional tool, the algorithm may be forced to take sub-optimal decisions while exploring the solution space in order to avoid resorting to only a single well-performing strategy, possibly getting stuck using a single pattern of moves and

not other more effective solutions. To this end, the mentioned balance between exploration and exploitation must be included in the code (here in the form of "if clause"), to ensure that moves which do not follow the optimal pattern are selected to ensure that they are not a potentially optimal solution. This constraint however can be progressively lifted by adjusting the ε parameter to reduce the chance of selecting random moves at any given state [11].

1.4 Game environment

Pokémon is probably the most known video game franchise in the world. Players collect, train and battle each other with the titular Pokémons, imaginary monsters with unique characteristics and skills. Various games have been published since 1996, but competitions involving video games started only in 2009 with rules set by the official organizations “Play! Pokémon” and “The Pokémon Company International”, that oversee every aspect of the many categories of tournaments.

Video games tournaments are usually organized as a way to promote the latest game released, hence teams and tactics may vary across different years; however, the rules were fine tuned with the experience of more international events and the growing number of participants across the world up to the current version. The key aspects of the rules employed in official competitions are how battles are performed and the limitations regarding team compositions: the former requires players to play in a 2v2 scenario, as opposed to the more common 1v1 greatly used along the game's story, to allow more complex strategies involving many different elements, while the latter is an easy way to avoid single elements to completely overshadow the rest of the teams and strategies. Moreover, since 2017, IEEE organizes competitions on a clone of Pokémon games with a similar rule set from the official competitions, where researchers are given a team, or are required to build their own, and submit a code that will enter the competition and play in their stead [4].

Games are divided in turns, in which players select their moves and cannot affect the results of their decision until the following turn, which means once both players have selected the actions to perform, they must wait until the next turn to choose a new move, similarly to rounds of Rochambeau; some exceptions are possible but generally games are played out with this structure.

Pokémons in a team are identified by their species and their effectiveness is evaluated by their statistics:

- *Attack*, which affect the damage dealt by physical moves.
- Defense, which affect the damage reduction for physical moves.
- Special Attack, that will determine damage dealt for special moves.

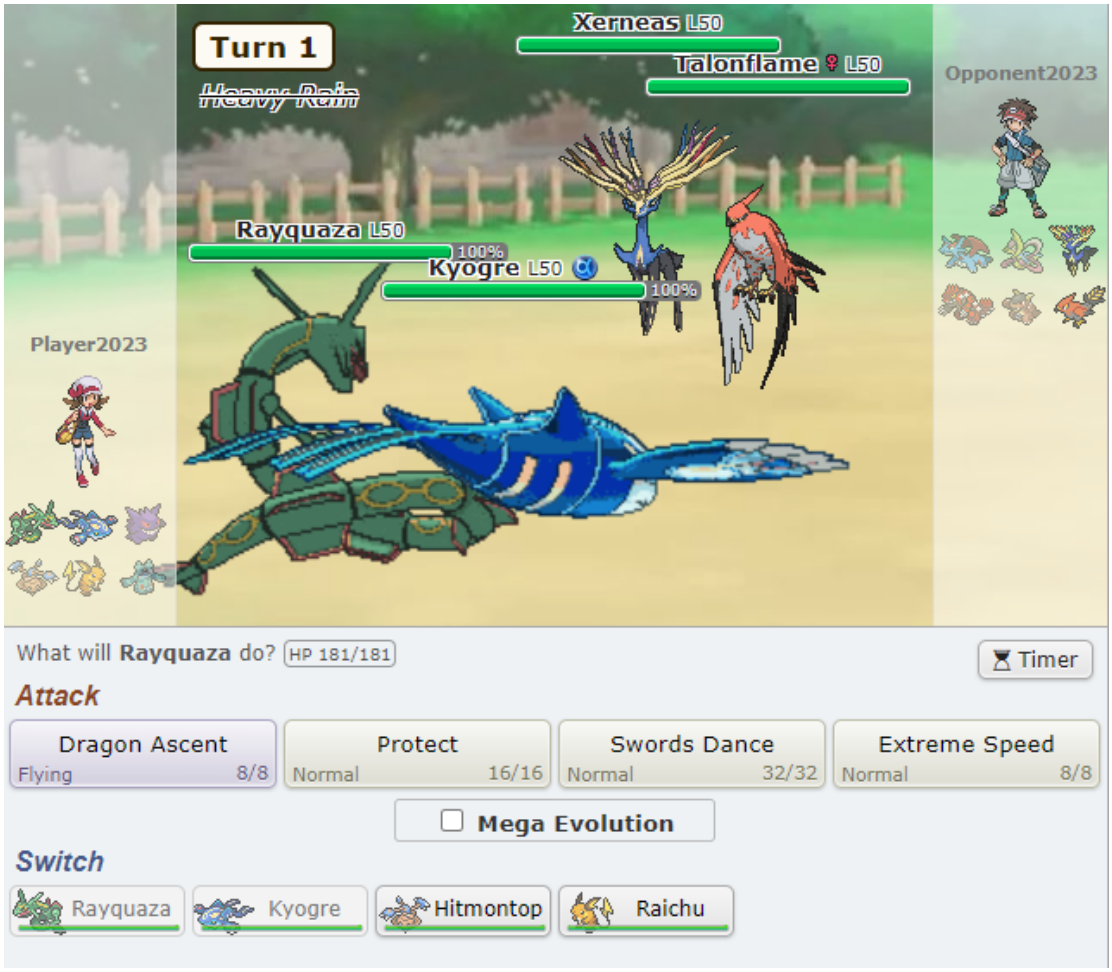

Figure 1.2: A turn in a game of Pokémon on the battle simulator "Showdown", most of the information on the game is available to players via this interface

- Special Defense, that will determine damage reduction for special moves.
- Speed, the value that controls the order in which each Pokémon acts within the battle, the higher the value the earlier it can move in a turn.
- Health Points (HP), the value that gets subtracted during the battle, once it reaches zero the Pokémon is defeated.

Each Pokémon interacts within the game with up to four different moves, identified by a unique name and with set values for power, accuracy and number of times they can be used during a match. Most moves have additional side effects that users can learn by reading the associated description in the game: this comes especially useful when dealing with non damaging moves (status moves). Damaging

moves are further divided into two categories, *physical* and *special*, resulting in different statistics (described above) required for the move to perform optimally.

Due to the recurring nature of these events and the ongoing support and development of new video games, the selected set of rules was chosen for convenience to be the VGC16 rules (from tournaments which took place in 2016): the reason behind this choice is that older sets of rules usually have established ranking information for teams, moves and strategies at the expense of a less active player base. The main aspects of the VGC16 [12] rules are:

- Battles are to be played 2v2, meaning every player must use two Pokémon simultaneously whenever possible for the duration of the game.

- The match is played up to three times: once one of the players reaches two wins, they are declared winner; draws are rare but possible: in the event a match ends with an even outcome, additional games are to be played until one of the two players wins.

- Players can use only up to four members of their team per game, selected independently at the beginning of each game.

- Pokémon species within the team must be unique, meaning that players cannot use multiple copies of the same Pokémon in the same team.

- The team can contain at most two Pokémons included in a specified list, in order to prevent players from using teams uniquely composed by the most powerful species in the game, often referred to as *Ubers*, from their status in the tier lists, or *Legendaries*, the label usually employed within the game story.

- Items used by members of the team must be unique.

- Levels will be normalized at 50, to reduce overall battle times by limiting the maximum available values for statistics.

- Matches have a 15 minute overall timer, at the end of which the team with the most undefeated Pokémons is declared winner; in the event of a tie, HP count will determine the winner.

- Players have 90 seconds to select the four Pokémons to be used in the match.

- Players have 45 seconds to select the moves to be used in any turn.

1.4.1 Team Preview

Competitive players refer to "team preview" as the initial 90 seconds of a match in which contenders have to select the four Pokémons they are going to use for the rest of the battle. When performing that choice, it is a common behavior to analyse the opponent's team, trying to anticipate the possible tactics they are going to employ, how to disrupt them and trying to imagine possible counterattacks for their own, while planning a reasonable strategy and backup for their team as well. Definitely a complex task that requires experience, as seeing many different games from the same player, or team, highlights their best options and weaknesses, as well as knowledge of both their team and the opponent's, to more accurately preview how some moves may play out in optimal conditions and exploit the advantage for a more precise plan. Selecting four out of six Pokémons, often allows players to reverse the odds in their favor with a different strategy: for this reason matches are played in a "best-of-three" format, where players need to win two separate games to win the match, similarly to tennis sets, to allow players to recover from poor starts and even out possible random occurrences.

1.4.2 Moves

Every turn, players must select a move to progress the battle until it reaches its conclusion; failing to do so results in a random move selected among the available ones. Moves usually have the most diverse effects that may help achieving victory in different ways; however, building a team with a single effect in mind is not always a successful strategy, which is why players usually need to find good balance around the main move types.

Damaging attacks, usually described with a name, damage value and accuracy, are the primary way to reduce the opponent's health to zero, which leads to victory in case the whole team is defeated in this way. These moves are the main focus of Pokémon whose purpose is to push towards the conclusion of the game (often called "sweepers", due to their role to remove foes from the game by quickly defeating them), which have many different attacks as the ones described to choose from. This kind of move usually comes with a trade-off, with more powerful moves come more penalties; for example, a high damage attack may have 90% as accuracy value and/or an additional effect that temporarily reduces the Pokémon's statistics.

Status moves, often referred to as support, are moves that do not excel in damage but are very useful when their side effects activate, potentially disrupting the opponent's strategy or strengthening the player's position in a game. Such attacks are often associated to very defensive Pokémons that need to stay on the field for as long as possible to support the rest of the team, usually not as bulky as them, and ensure the correct execution of a strategy. Since their damage is mostly negligible, it is usually considered as "chip damage", that is welcome in case of need

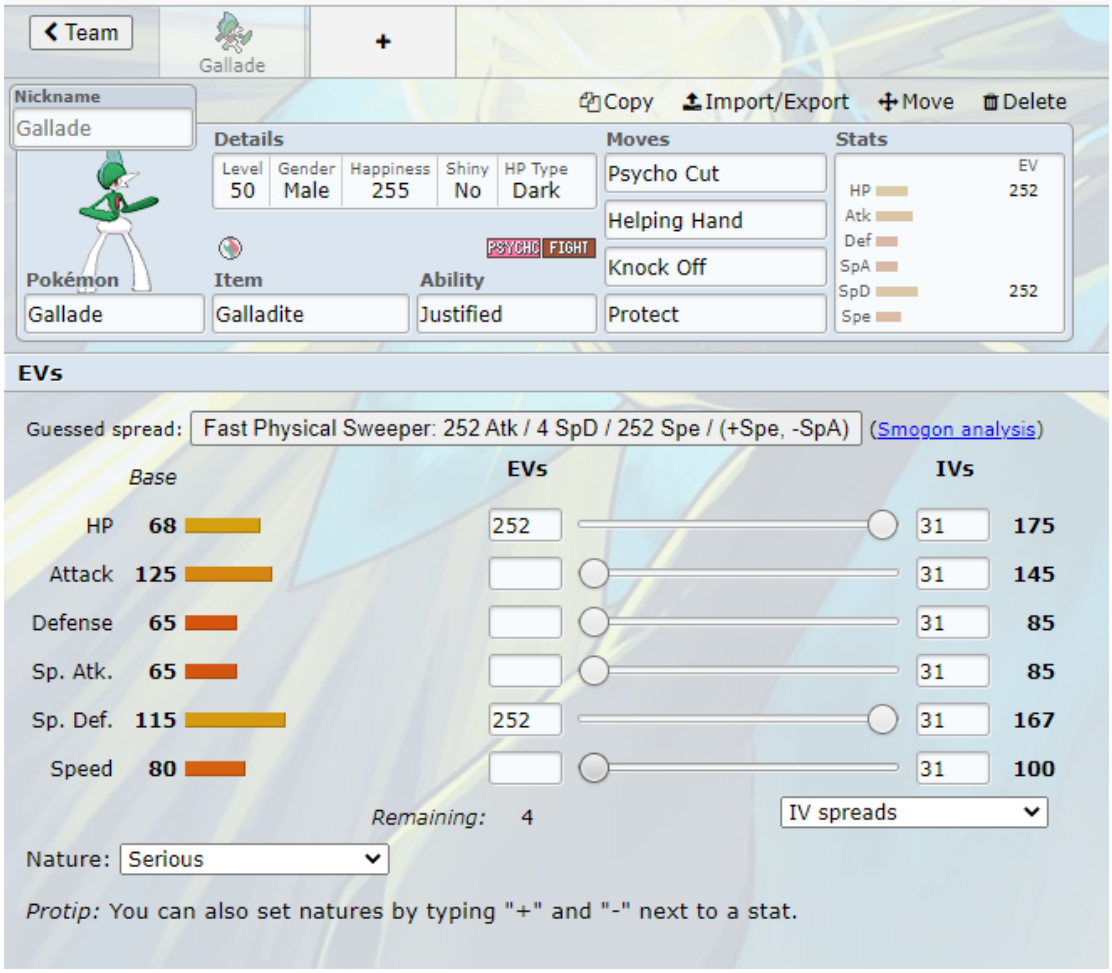

Figure 1.3: Interface for customization of Pokémon, the four moves are on top and the sliders are used to customize the statistics of the selected Pokémon

but it is rarely expected to be the main solution when defeating opponents. The longer the game, however, the more it becomes relevant, since Pokémons that have already suffered damage are not guaranteed to be able to sustain more powerful attacks from sweepers: an additional reason why players usually need to keep in consideration their opponent's defensive tactics. When a status move has very little damage, or no damage at all, usually it comes with a persistent side effect that benefits players for a prolonged period of time, allowing strategies to be more and more complex in the long run and discouraging greedy, and rather easily blocked, tactics.

Protect moves are the easiest way to avoid taking damage for a single turn, often used to bait attacks or to scout the opponent's strategy while maintaining the

position on the field. Pokémons who protect themselves however are not allowed to move for the rest of the turn, resulting in a move effective for stalling turns while waiting for a side effect to subside or to block a very powerful attack in unfavorable conditions. To prevent players to spam the move and abuse this mechanism, the game progressively reduces the chance of protection moves to correctly apply for each following turn the move is used successfully (n), according to the formula $\frac{1}{1+n}$, where n is reset after a different move is selected or the protection move fails.

Considering that any Pokémons can learn up to four distinct moves, this allows players to customize their team to account for a wide range of tactics to produce a win condition for any game; however, not every species can learn every move, therefore opponents can make educated guesses on which Pokémon covers which role, a key aspect when the game starts and contenders have to select the four members for the game.

In addition to attacks, since players have up to two reserves when entering a game, switching a Pokémon is an available move in most cases (there are conditions, moves and Pokémon abilities that prevent some or all other Pokémons from leaving the field), often employed to reposition the team in a defensive configuration or, usually when expecting a similar move from the opponent, to maintain a favorable position when setting up the strategy. Players may also switch in order to prevent damage, either by removing from the field a weakened Pokémon or by sending out a bigger threat to the opponent's tactics. Overall, switches are another available move that adds some depth to the battles, forcing players to take into account a wider array of moves and tactics rather than focusing on what is visible at any given turn, from the team preview up to the final turn of a battle.

1.4.3 Turn resolution

After both players have selected their moves, the new turn is resolved following some rules and players can analyze the results before selecting new actions. Since many aspects of the opposing team are unknown until shown during the game, contenders pay particular attention to the many details hidden in the resolution phase of turns.

First, the order of movement is one of the main points of interest of many players, since attacking first is usually a significant advantage that may lead to defeat opponents before they can act: each Pokémon has a Speed stat, that determines the initial order of action by ranking every Pokémon on the field by the value of such stat in reverse order. Top level players build their team to have a very precise Speed to scout the opponent's configurations earlier than others, usually by selecting values that differ by a single point rounding either up or down. Skills are also activated following this order, which will be the primary indicator to observe right before turn 1, where Pokémons are sent out after the team preview

phase but before acting, since players still need to select their first actions. In case two (or more) Pokémons have the exact same value of *Speed*, the order is randomly selected.

Some moves, however, ignore the order stated by *Speed* and are usually marked in their description by strings such as "usually goes first" or "will always go first"; these moves are called "priority moves", due to their property that allows them to act earlier. Priority is basically the value that indicates when a move will act compared to the regular *Speed*-based resolution. Ranging from a value of +6 to -5, where 0 is the value for standard moves, a higher value will result in an early movement, while a lower one will force moves to act later; for negative values, the move will activate after the standard turn order. In case multiple moves with the same priority are selected within the same turn, the final order will follow the rules for regular (0 priority) moves.

Another important aspect to keep under control is how damage is applied by attacks, due to some random variables involved in the calculations: the formula that produces the value of the damage applied by a move covers many aspects of the game state at any turn, as well as some multipliers that may apply under specific circumstances, and expands as follows:

$$\text{Damage} = \left(\frac{\left(\frac{2}{5} \cdot \text{L} + 2\right) \cdot \text{P} \cdot \frac{\text{A}}{\text{D}}}{50} + 2\right) \cdot \text{STAB} \cdot \text{T} \cdot \text{C} \cdot \text{R} \cdot \text{Other}$$

where

- L is the level of the attacking Pokémon, the maximum available value is 50.
- P is the power of the attack, the main source of damage of a move before multipliers are added to the formula.
- A is the attack of the Pokémon performing the move, may be either Attack or Special Attack depending on the move and its effects.
- D is the defense of the target Pokémon, may be either Defense or Special Defense depending on the move and its effects.
- STAB or Same Type Attack Bonus is an additional multiplier of value 1.5 when the move performed is of the same type as the Pokémon performing it, otherwise it is 1.
- T is the type effectiveness of the move against the type(s) of the defending Pokémon.
- C or Critical is the multiplier for critical hits, that may randomly take the value of 1.5 with a chance of about 6.25% (4.17% in newer versions of the game) under normal circumstances, but can be increased up to 50%.

- R is a value that ranges from 0.85 to 1 that is generated for each attack performed by any Pokémon on the field that determines a degree of randomness to the final value.
- Other covers a larger number of multipliers that may be applied under specific circumstances but ultimately is not useful for the discussion.

The random value intrinsic in the formula is mostly negligible: its magnitude is fairly limited as opposed to the other factors and players build their team taking into account worst case scenarios in which damage is always rounded down to the lowest value possible. However, critical damage may result in an unexpected turn in the original strategies, producing an advantage or disadvantage that needs to be taken into account: for example, if an attack's expected damage is about 40% of the health of a target, suddenly receiving 60% means that the minimum required hits to be defeated decreases from being guaranteed three to possibly being two, depending on the random variable multiplier, forcing the player to adapt the strategy according to the new state, less likely than the expected outcome but still plausible [13].

Chapter 2

Proposed approach

2.1 Motivations

Reinforcement learning is a very useful tool, but what is the reason behind its adoption as opposed to other algorithms? Many other algorithms have the means to tackle episodic tasks and learn how to complete a series of actions in the best possible way; tree search, for example, is an easy way to represent problems in a human readable way and provides a more accurate image of the task at hand. Moreover, some evolutionary algorithms are specialized in graph traversal and tree search, proving other options are available when dealing with this kind of scenarios.

However, the main issue with tree traversal is not in the algorithms, but in the trees structures themselves: for simple tasks, a tree might have a limited amount of nodes and a proportional number of edges, therefore it may seem reasonable, but increasing the number of nodes leads to a messy representation with long traversal times, combined with the fact that edges must convey the required information for the algorithm to evaluate the available option at any given state.

In addition, reinforcement learning can be set up to manage the simulation autonomously, requiring little to no intervention by the user both during training and deployment, which proves particularly useful when dealing with complex systems with variables that experienced users may give for granted when taking decisions. Reinforcement is key in this case since the algorithm is able to understand the given task by the reactions of the environment rather than from supervision, allowing for a completely unsupervised code that may be executed in several different fields like content filters or search engine support algorithms: for example, spam filters [14], email, SMS and calls can be automatically managed as the user is part of the environment, therefore unwanted content can be marked and, with a large enough set of data, the code may be able to predict what to filter, whereas for search engines it is possible to redirect the user towards more relevant results

according to preferences and previous keywords.

Given all the minute details to be accounted for described in the previous sections, it is highly unlikely that a player can luck their way out of the competition for long, especially against more experienced opponents that may know better move combinations and counter strategies to severely limit the available options during the game. For this reason, players planning to take part in tournaments spend a significant amount of time building their teams, developing strategies against the most common threats and, in general, preparing for the competition.

To this end, players usually have to take part in a large number of private games to verify possible flaws in their approach to the game or in their team, resulting in a proper training session, similar to what a reinforcement learning algorithm may undergo: with this premise, the idea of the project is to verify that, provided that a sufficient training is conducted, a reinforcement-learning-based algorithm may result in a competent behavior in the context of the game.

In particular, human players exploit these test matches to gather information about their team, in particular regarding the damage dealt and received by the various Pokémons, the impact of randomness of some aspects of the game on the strategy, as well as the psychological training to be prepared during both easier and harder games, to prevent mistakes due to anxiety or distractions.

As the project is not capable of human emotions, the main goal behind the training for the algorithm is to gather information; however, since the code does not rely on the descriptions attached to moves or on in-depth knowledge of the game and the team it is using, the required data is going to be significantly larger compared to a human player.

With the inclusion of Pokémon into IEEE competitions and the increasing interest in the field of machine learning, some research has been published discussing various aspects of this game as a benchmark for future studies; however, due to the sheer number of available formats and the ongoing development of new games, the research focus is often limited to older games [15].

2.2 Issues

In order to train the algorithm, a sufficient number of games must be played; the best way to achieve this feat is to deploy the agent online and let it play against the largest number of available opponents; nevertheless, there are a few issues with this approach that may alter the experiments' results. It is a known fact that the competitive scene in any kind of game has to take into account many unorthodox behaviors on behalf of the player base like *smurfing* and gimmick exploitation only to name a few. *Smurfing* is the practice of creating a new account as an experienced player, relevant mostly in ranking-based matchmaking, to win easily

by pairing with newer users, whereas gimmick exploitation is the practice of players disregarding common approaches to games only to win using trick strategies that work only against oblivious opponent.

Moreover, a human player may behave differently when playing against non-human agents or try to tamper with the algorithm by playing non optimally to win in future matches. The trade off of playing with human opponents is to significantly speed up the training process sacrificing the exploration of some solution space an is probably not worth the risks presented above; in addition, the chosen rule set is (at the time of writing) already seven years old: it is therefore unlikely to find players that fall under the requirements for an optimal training routine.

Finally using a private server for training helps circumventing all the security measures placed by developers to prevent spamming and flooding, while giving administrators the right tools to find any possible users with malicious intents, since the environment would be used only by non-human agents on a local machine, possibly preventing access to third parties [5].

2.3 Tests

The required tests for the agent follow the described guidelines in the above sections; in particular, the algorithm is expected to be able to react to players at different levels of skill, as well as to adapt to new opponents or strategies.

The reason behind the tests presented hereafter is to verify that the algorithm is actually capable of making choices according to the provided environment and the past experience; in particular, the main reference will be the comparison between trained and untrained agents against various forms of opponents, designed to mimic players at different skill levels [5][16].

However, the agent should be trained as a general purpose player rather than to actively counter specific tactics: therefore, a training routine against itself with hyper-parameters set up for exploration might prove useful when the code has not any experience of the game [17], since the code has no understanding of the moves' descriptions, which usually suggests to human players how to effectively use a move and its side effects. For instance a move such as "Icy Wind" is very weak compared to other damage-oriented attacks, with a power of 65 it is exactly half as powerful as Groudon's "Precipice Blades" which has 130 power; however, the purpose of this move is to slow down opponents, reducing their Speed stat and allowing allies to attack first in the next turns, providing a significant tactical advantage over the opponent's damage-dealing Pokémons. In addition, Pokémons can have at their disposal multiple moves with similar side effects aimed at helping the team, meaning that the agent must have a reasonable understanding of how these moves work and how to include them in a more complex strategy.

2.3.1 Test 1

The first and possibly most common test is to compare the trained version of the agent against itself in various different ways to see how the training impacts on performance; in particular, the focus should be to understand how effective is the strategy of the trained model and how quickly the untrained model is able to adapt to a new opponent. The objective of this comparison is to show how the trained model is able to devise a strategy taking into account many possible aspects within the game, random and not, balancing risks and rewards to achieve victory, while still adapting to the new possible game states provided by untrained models or new opponents.

The aim of this test is to simulate the interactions that many players have to face, i.e. different players using the same team with different strategies: in the final ranking of the world tournament held in San Francisco, California, the same team classified first, third and eighth out of the top 16. While the answer for this kind of occurrence is usually a different combination of factors such as luck,

strategy, skill and experience, the tournament results prove that different strategies are possible and their effectiveness varies: the code should therefore be able to find a way to beat opponents with a consistent strategy accounting for risks, but still it is required to switch between known strategies seamlessly according to the state and to known results.

The most common occurrence for this kind of scenario is usually after an international tournament, when the participants give to the public most of the information about their teams due to the larger audience and more focused interest. With this data, any player can use a top level team for testing matches, improving strategies with and against said team; however, even less experienced players will possibly start using a winning team to improve themselves rather than the available tactics, often trying different approaches than the ones proposed by others.

2.3.2 Test 2

An additional test to be conducted is how effectively the agent is able to react to different strategies against the same team, such as a novice's in contrast to an experienced player: how much the training will be of use or, conversely, be an obstacle. Theoretically, the code should have a reasonable amount of data about which moves to use in many potential scenarios involving members of any two given teams; however, the initial games may lead to a number of defeats linked to the multi-turn setup of more complex strategies, which require users to use suboptimal moves during the first turns in order to have a stronger position at a later stage of the game.

This approach to the game relies on the idea that the opponent is trying to set up a strategy of their own and is able to possibly counter simpler moves, which may result in poor performance against new players who are just unaware of more complex interactions. Depending on the way training has been conducted and on how effective the agent is with a given pairing of teams, the results should be a helpful insight of how unexpected behavior is handled and how much the strategies of a trained algorithm must be adapted when changing opponent, particularly relevant in the context of real life tournaments, which (as described above) may host different players with the same team.

While trained players may find some difficulties when dealing with a suboptimal opponent, a completely novice one should be able to adapt rather quickly to novice strategies and to develop a strategy of their own to effectively counter the only one the agent ever faced. Although it may seem pointless to develop a strategy against a non-adaptive agent, this may help point out how quick and effective the code is to find a response to certain strategies, be it more advanced or simpler, in particular when usually dealing with the most common optimal ones.

Usually human players build their team around the most used teams and

combinations, as well as personal preference in Pokémons and strategies, but the shared experience is usually built upon official competitions and their results, working as a test bench for future competitions: in fact, it is rare for players to take part in multiple major tournaments with the same team, unless it is good enough to remain relevant after some time, which is uncommon as well.

Players also prefer to use the same team for longer periods, as they usually adapt the strategies to new opponents in a similar way a reinforcement learning algorithm would do: the more they play, the more information they gather on how the team interacts with new combinations and new moves, with little to no adjustments, ending up in more consistency in terms of results as well, when possible.

2.3.3 Test 3

Another important aspect to consider is to verify the code flexibility when facing unorthodox strategies, keeping in mind the main problems involving low ranking players: some opponents might use unusual tactics to catch inexperienced opponents off guard, resulting in easy victories with possibly very weak teams. In order to verify this feature, the algorithm must face a player performing very similar moves in the same conditions or suboptimal strategies.

Players using these kind of moves are often trying to exploit particular game mechanics or combinations to their advantage and are mostly well known among experienced human players, who usually have a simpler strategy to manage such opponents. They are also recognizable because of non-standard team composition and the moves used they use; they are therefore rather easy to predict compared to the tactics employed in international tournaments, which hardly count such teams in their final rankings.

The aim of this test is to verify how quickly the code is able to adapt to this kind of tactics and how effective the resulting strategy is.

2.4 Implementation

The project revolves around a custom distribution of the popular battle simulator Pokémon Showdown, a web based application capable of setting up, simulate and record matches between players in many different formats. The server side is developed in Typescript and can be easily configured thanks to the public GitHub repository that allows free access to all of the source code files; however, the only file that has been edited is the configuration file "*server/config.ts*" to lift limitations on the allowed number of messages to the server per second. This security measure was added to prevent spam from users and bots alike and help the developers keep in check the system for DOS attacks, but since this instance of the server is used solely for the training of the code and no outside connection will be possible, this small tweak allows the project to definitely improve on performance while remaining safe for usage; besides that, the environment is exactly the same as human players use.

Usually, the communication between server and client is carried out via a web application which handles all the server messages, that mostly consist in pipe-delimited plain text or sometimes JSON snippets, to display the user available information in a Graphical User Interface (GUI) similar to the one used in official games for easier understanding. Since most of the information is transferred via text over a bidirectional socket channel, the code does not need to access the GUI or wait for the information rendering to start working on the game. Combined with the configuration changes described above, this helped greatly during testing.

The core of the project is written in Python, connections to the server are managed via a socket in the same way as the public distribution to allow the code to run seamlessly in any server compliant to the original communication protocol [18]. The interaction between server and the project is therefore managed by means of an interface that acts as parser for the code and converts all server inputs from string to Python data structures, which are later normalized to ensure that the code only handles data in a predictable way. Because of the initial (lack of) knowledge of the project, additional redundancy measures have been included to prevent locks in case of errors and, when possible, try to recover to proceed with execution, thanks to recognizable patterns in received server messages.

The code handles single games separately and independently to emulate the web application, as well as allowing multiple simultaneous games with the same settings for a team. Once a team has been selected and loaded into memory, the code tries to access previous learning memory by querying the shelf dedicated to the team, when available, it loads the table in memory as well, otherwise it creates a new record to start training: this option is needed as, due to the match-up oriented nature of the game, different team combinations are handled separately to reduce the overall memory requirements at run-time.

Chapter 3

Experiments and Results

3.1 Setup

First, the agent requires information about the team it is using and its moves to be able to determine the effectiveness of each move in the current state. In order to achieve this goal, the initial simulations were set up to explore as much as possible the solution space, so that each new move used by either side of the game would be added to the internal database for future reference. This solution reduced the required time for each turn by decreased the number of messages sent to the server, saving up to 50 ms per message, greatly improving the performance since each request to the server usually has to handle up to three messages as response.

Following the initial setup stage, training has been conducted with hyper-parameters set to $\alpha = 0.8$, $\epsilon = 0.5$, $\gamma = 0.8$ to provide a wide enough variety for many different states of the game, while maintaining the chance for on-policy plays to avoid unbalanced values across the table, potentially resulting in a strong enough local optimum that would prevent the exploration of more successful strategies when increasing the chance of on-policy plays.

The resulting memory was then backed up for consistency across different experiments, providing a common starting condition for all instances of the trained agent.

Many training sessions were performed before producing a satisfactory learning table for the agent: this was mainly due to the randomness involved in the process, leading to larger values for suboptimal values caused by some very beneficial coincidences during the exploration. Such values were slowly decreasing to more balanced rewards; however, the results would eventually indicate a large portion of games lost to fix the initial value of a single move when playing on-policy.

When referring to learning tables or simulations as “balanced”, it means that the agent is not actively exploiting a single strategy due to initial games influencing

heavily the following batch. The need for this distinction is due to the wide solution space of a single match-up between two teams: exploiting a single pattern might therefore prove useful when testing a single strategy but overall fails the purpose of this project; to this end, initial training simulations were filtered to verify the effectiveness of the agent in many possible scenarios.

3.2 Experiments

Keeping in mind the scenarios described in 2.3, testing has been conducted by maintaining a copy of the starting learning table and customizing the various opponents starting from a common code base. The different instances of the agent would still use predefined teams, whose information has been gathered from team reports of the original players who created them, rather than building them from scratch, to keep the simulation environment as close as possible to the real tournament scenario.

The simulation parameters were chosen to be $\alpha = 0.8$, $\epsilon = 0.8$, $\gamma = 0.2$ to allow the code to still learn and adapt to the new specialized tactics that opponents might use, since the goal is to test the strategies and verify how effective they are when playing to win. However, since the value for new moves is still considered, new strategies may still be discovered, but the amount of games required to be taken into consideration by the agent is going to be slightly larger in case of completely new moves. Finally, the reason why the optimal moves rate is 80% is to prevent local optima from stalling the simulation and encourage exploration up to some degree, in addition to the fact that across multiple games the agent would be prevented from mindlessly repeating moves in case of favorable states.

Usually, players use the Pokémon Showdown simulator to test teams before competitions, due to how quickly it is for users to set up a team and start games, allowing for multiple simultaneous games to be played. The number of games for a human to determine whether the team is viable and which tactics work better in different occasions varies, since the team building phase usually starts from a core strategy to be developed with some variations, depending on how opponents are expected to react. Moreover, testing helps when collecting information about new strategies and teams, and provides details on how to further tweak the team composition to increase effectiveness. Finally, with the gathered data they would eventually build the team on the official game, which is a slower process compared to Showdown, and start testing on the platform used for the competition, where players are usually more similar to the expected opponents.

However, the teams used during the simulations are the two teams which reached the finals of the 2016 Pokémon World Championship: their expected performance is therefore mostly known and the code has access to some of the best tools and

strategies available in the VGC16 format. This solution was adopted to focus on the learning and decision-making phases of the agent, expecting the two teams to be well built and overall an effective starting point for the simulations.

3.2.1 Preliminary results

Balanced training simulations proved that both instances of the code had all the means to prevent either agent to gain a significant lead, usually remaining within 40 wins between the two versions, indicating not only that the two can actually adapt to each other, but also that, with the resulting table, the agents would still be able to employ a couple of viable strategies to win.

However, the final results proved that, although balanced in terms of performance, a slightly early start during training leads to a significant advantage in terms of victories: even a simple 100 games jump start resulted in a 6% increase in victory rate, possibly linked to the, even shallow, experience of the first few turns of a game, at the expense of a more specialized training for the instance that starts at disadvantage.

Moreover, after training, the two files containing the two agents' learning tables had respectively a size of 4.58 GB and 4.45 GB, slowly increasing along with the number of training games; given the remarkable disk space required by these files, they are monitored for the duration of the experiments.

3.2.2 Simulation 1

Following the testing program presented in section 2.3, the first conducted experiment involved a completely untrained instance of the agent against a trained version of itself: the expected results should suggest that the trained agent be more effective initially, while the untrained agent should be able to slowly discover better ways to counter the most effective strategies. A useful aspect to take into consideration compared to peer training is that the opponent will mostly use near optimal moves, maximizing the penalties for any very disadvantageous move; however, since the only difference between the two agents is their experience, effective moves would still be awarded reasonable rewards, meaning that the trained code has access to optimal moves but is not overwhelmingly stronger, since both teams have all the right means to have balanced games.

The simulation is executed over 1000 games, in which agents will use the same teams as during training with parameters set as $\alpha = 0.8$, $\epsilon = 0.8$, $\gamma = 0.2$ for both agents in order to provide a challenge for the untrained agent while being able to explore solutions across several different game states. The untrained instance of the code is allowed to update its learning table to adapt to the strategies from the trained code.

3.2.3 Simulation 2

When players first log into the official Pokémon Showdown server, they are all given a score of 1000 points, the baseline for ranking players based on their performance. As all users start from the same score, even experienced players must at first adapt to low ranking plays and strategies, even with very specialized teams, usually built for top level competitions. To emulate such scenario, the code will play against an agent playing random moves; the reason behind this decision lies with the need for many different potentially suboptimal (if not wrong) moves across many different matches.

Since the goal is to test the code under low ranking conditions, the opponent is set up to not learn from previous games and force it to play only with random moves over the required 1000 games with the same hyper parameters as above, $\alpha = 0.8$, $\epsilon = 0.8$, $\gamma = 0.2$ due to the need for potentially new strategies and to better adapt to the opponents.

3.2.4 Simulation 3

Along with higher rankings come more effective tactics, as well as more experienced opponents, so that testing teams against them may be more precise and potentially interesting. These opponents are expected to have developed reliable strategies to win enough matches to progress towards higher scores; however, due to the same initial conditions described in point 3.2.3, these tactics may be potentially suboptimal because they were developed against low-ranking players.

To emulate this scenario, the agent is set up against a similar code as in 3.2.3, but the opponent selects the best four moves in a greedy fashion as a pool for the random selection, resulting in a greedy-like approach but not as repetitive as a completely greedy algorithm, in order to prevent repetition and therefore predictable by the tested agent. Again, the simulation is executed over 1000 games with hyper parameters set up as in the previous points to $\alpha = 0.8$, $\epsilon = 0.8$, $\gamma = 0.2$.

3.3 Results

The overall results pointed out some expected factors that heavily influenced performance: namely, how states are defined and how training is conducted, as hinted at in the previous sections, play a significant role in how the agent selects moves and plans for strategies during games.

First, the amount of details recorded when defining a new state helps the algorithm when elaborating a strategy in the following games, since the conditions for the strategy to work are well defined and the code is capable of replaying the steps required to end up in an advantageous state. Moreover, since some moves react differently to different targets or scenarios (weather, statistics modifiers etc.), the code is capable of exploiting information gathered during previous runs to adapt to the opponent's counter moves, which often lead to conditions similar to those seen in other games, but with some subtle difference that may drastically reduce the effectiveness of the optimal strategy. However the trade-off for a higher level of detail is a notable increase in disk space usage for the learning tables due to the granularity of states and the sheer number of variables that define them; to limit the file size to a reasonable value, the minimum amount of data to be stored proved to be a combination of information about the field and previous moves to achieve a satisfactory performance. In addition to the state definition, training had to be conducted without supervision; however, the code would be stuck in local optima if during some games at the beginning one of the agents developed a predictable strategy due to a very unsafe move, easily exploited by the opponent and resulting in a learning table heavily suggesting said action, expecting the opponent to make the same mistake. This often leads to a hiatus that could extend for several games before the code could develop new strategies and break free from the initial optimum. There is not much to do to prevent this kind of behavior due to the initial randomness of the training, however combinations of moves such as described are not common, so restarting the simulation from scratch proved to be a quite effective solution to save disk space and time for training. This is a common issue when dealing with reinforcement learning in large state-action spaces, leading to agents exploring the solution space without ever reaching a satisfactory conclusion or to instances of very short episodes due to early defeats for the code [19].

3.3.1 Game Analysis

In order to reasonably analyse the results of games, many different approaches have been proposed; however, due to the variety of strategies and randomness involved in games, trying to reduce the evaluation to a mere numerical value would overly simplify the evaluation and planning processes. As mentioned in the previous chapters, high profile players usually produce a "team report", a document (be it written or on a video) that describes in detail all the decisions taken during team building and the general strategies they expected to employ in different occasions, with some references to nuances and possible adjustments needed in case of unexpected occurrences (random critical hits, activating side effect to name a few). Players may be able, with the help of such documents, to reproduce the team and its general behavior without the need to go through the process of building it and planning for strategies, resulting in an effective tool to study the strengths and weaknesses of the best teams, as well as effective strategies in the chosen format. This information was particularly useful even years after the tournament, as it provided a fair scale to compare the algorithm to a real player side by side in similar scenarios, supporting the evaluation process of the games played across the various simulations. In particular, sample games have been selected approximately every 25 matches to evaluate the strategies employed by either agent, with a close attention to the code with learning options. Lastly, since the teams used during the simulations were the two from the 2016 VGC World Tournament finals, a direct record of the game is available as additional reference.

The main way to evaluate competitors, due to the randomness involved in the game environment, is to measure the amount of victories against opponents and, according to rating, a score will be added to the winner depending on the difference of rating (ELO rating system). This method however relies on the premise that players have equal chances of victory when entering a game (the original rating system was designed for chess competitions); therefore, the actual rating may suffer from match-up oriented games such as Pokémon. To compensate this issue, official tournaments expect player to win the most out of three games against the same opponent to record the match as a win, focusing more on the amount of victories during the competition to produce a rating.

To evaluate the agent's performance, a mix of the two proposed solutions has been employed, taking into account that both players have access to top level teams and strategies, being the best results of an international competition, hence having comparable chances of victory. Either method alone would not be sufficiently accurate to describe how the two agent approached the games.

3.3.2 Trained against Untrained

When playing against a trained version of the code, an untrained agent is required to understand the opponent's strategy and elaborate a counter plan to effectively limit their moves and learn as much as possible from the limited number of games. The results of this test accurately proved this assumption, providing significant insights on how trained agents react to the provided situation as well.

Starting from mere results, the trained agent won 57% of the times, an expected result since the initial games were easily won due to its advantage; however, the untrained code was able to catch up on the opponent's strategy rather quickly.

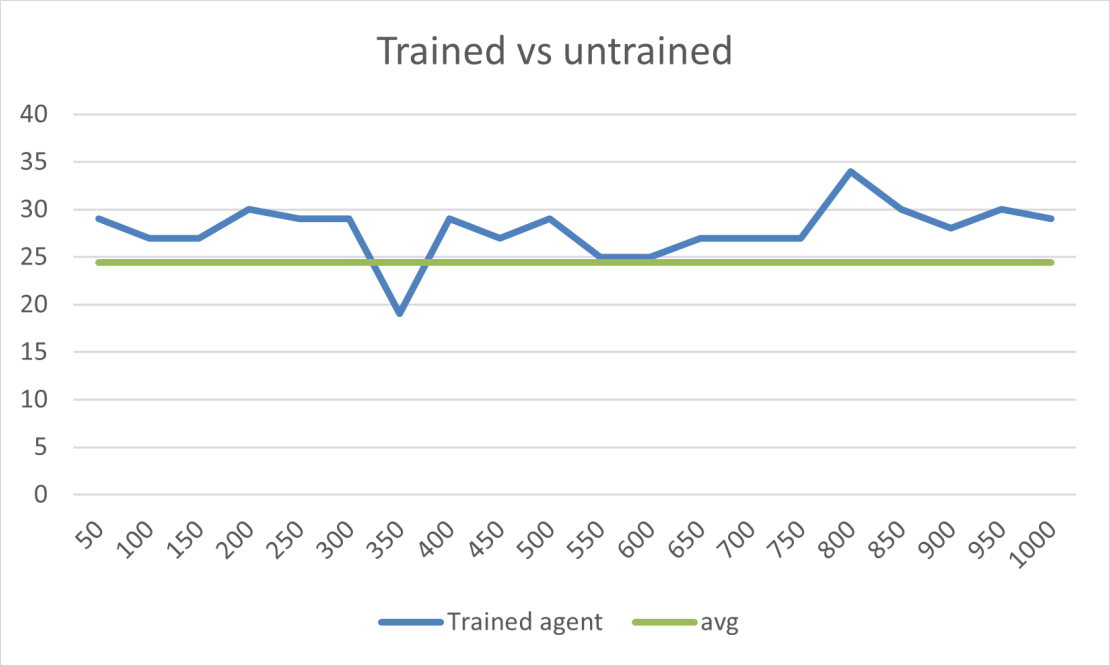

Figure 3.1: Plotting the number of wins every 50 games, the average is the axis of symmetry for the two players

Looking in detail how the simulation unfolded, the initial few games are generally won by the trained code, exploiting the lack of experience until around game 300 when the untrained code started developing viable tactics which led to a notable number of victories and overall balance between the two, until the trained algorithm adjusted the strategy to gain a major advantage for the last hundreds of games.

Looking at the above data, it is clear that the initial experience from training proved to be useful to maintain a consistent advantage over the opponent, in particular when the games required a different approach: the larger learning table suggested the agent a wider variety of moves to avoid repetition and exploitation by part of the untrained algorithm. On the other side however, using a specialized

opponent not only helped during training, resulting in a substantially smaller table size, but also provided a competent agent, capable of dealing with a far more effective player, at the expense of some exploration but ultimately proving to be a suitable training routine for the algorithm.

As an additional remark, the file containing the final learning table for the untrained agent was significantly smaller than the opponent's, due to the limited amount of exploration allowed against a player employing near optimal strategies, eventually settling at 809 MB. The fact that the untrained agent was able to catch up on the opponent rather quickly may be due to the larger overall penalties for moves that do not counter the opponent's tactics efficiently, leading to a relatively swift defeat, producing a smaller state space [20][19].

3.3.3 Games against random agent

To determine the effectiveness and robustness of the agent's planning, playing against opponents who approach games in a suboptimal way is an interesting test bench. The player should be able to recognize various game states, potentially discovering new viable tactics for unorthodox players, however it is a good way to overall understand how the agent handles opponents who develop different strategies than the ones it was prepared for during the training.

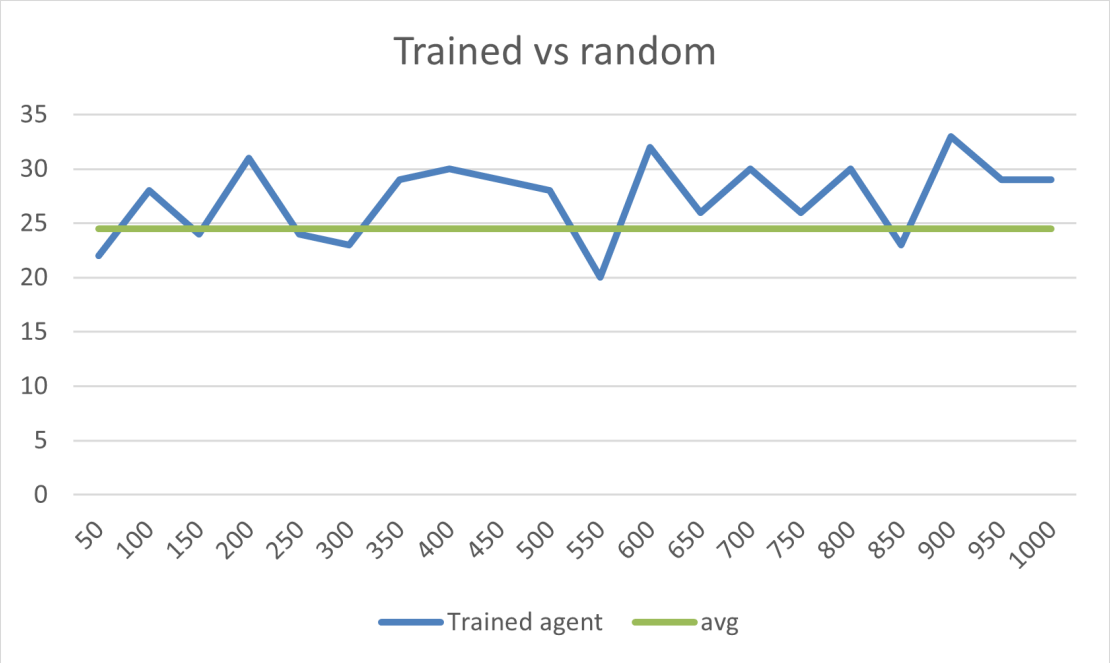

Figure 3.2: Plotting the number of wins every 50 games, the average is the axis of symmetry for the two players

The results are as expected at the beginning, since the agent is planning its moves expecting the opponent to develop a strategy of their own with the given team; however, conventional tactics are not meant to be adopted against such kind of players and they are very likely to fail if the moves from the other team are not considering the bigger picture. For example, in a very disadvantageous situation an experienced player tries to protect the team and switches out the most important assets to preserve their win conditions, while a random player could try to attack mindlessly: expecting the former move a trained agent may try to set up for the following turn and maintain the advantage, while the optimal move in the latter case is to attack right away exploiting said advantage.

The agent is very close to the average in the initial games, meaning that the number of wins is even during the first few hundred games; however, as the simulation progresses the algorithm is capable of developing a solid strategy that allows the code to achieve a large amount of victories for the rest of the simulation, with some exceptions around games 550 an 900, possibly due to a critical flaw in the strategy that arose in some selected scenarios.

The simulation ended with a win rate of 56.6% for the agent, slightly inferior compared to the performance against an adaptive agent (described in the previous section), linked to the key factor that the opponent was not adjusting a strategy, but rather kept playing random moves, therefore the algorithm could only refine its tactics based on random data.

3.3.4 Games against greedy agent

Greedy agents are a common way to program algorithms when first approaching a task, either because they are easier to code or due to their faster convergence. In the context of Pokémon, a greedy player should be a reasonable approximation for new players or users trying a team from other peers: the latter in particular is a common scenario for player who want to learn but cannot build a strong enough team on their own, usually trying basic strategies to check if they are effective. The agent should be able to adapt rather quickly to such behavior and develop a consistent counter strategy to maintain a significant advantage.

Moreover in this case, both a trained and untrained agent should be able to produce relevant performances in terms of both strategy and results; therefore, each of them were tested as additional comparison between the two instances.

The results show that the trained agent struggles during the first few games but overall it has the upper hand over the greedy algorithm, however when trying to optimize the strategy to be more effective, the states it moves in are different enough to change the opponent's pattern and therefore restart the optimization process, although the frequency of these occurrences is decreasing along the simulation as the only moments where the agent has a negative win rate are around games 250,

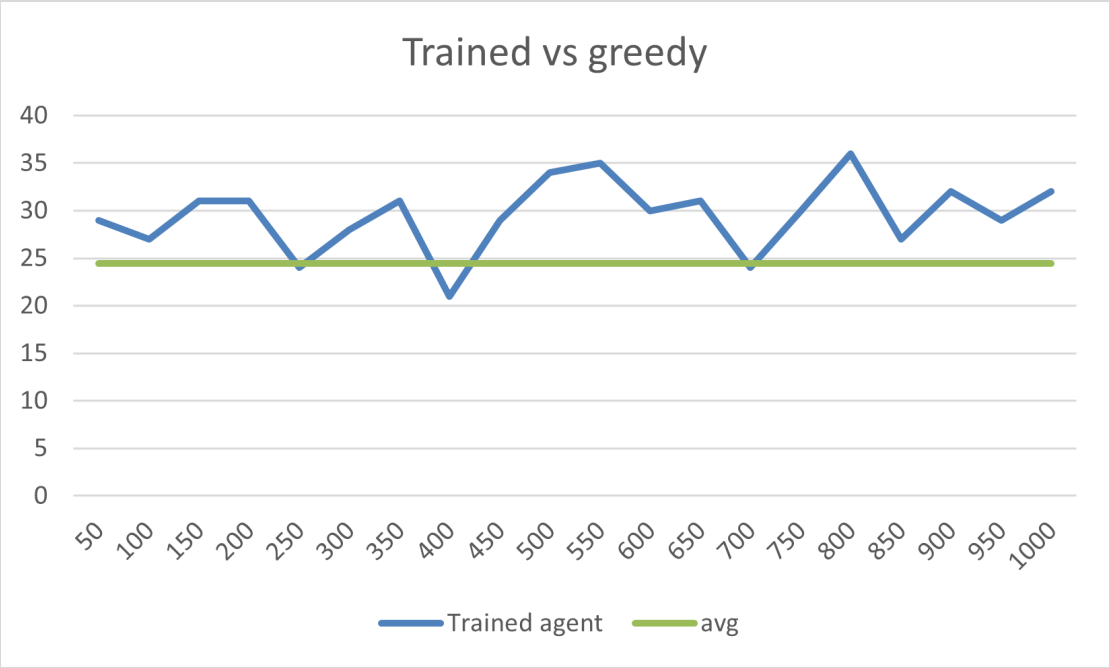

Figure 3.3: Plotting the number of wins every 50 games, the average is the axis of symmetry for the two players

400 and 700, possibly due to better optimization of past strategies. Again the training is likely the reason behind this behavior, as the code expects more optimal plays rather than greedy solutions, however, since the other player is predictable, results are on average better than the ones from the random opponent, rounding up with a win rate of 60.4%.

On the other side the untrained version of the code has a stronger start and overall better consistency since not having previous knowledge encourages the algorithm to exploit the best strategy at the maximum of its potential, however due to the non negligible probability to explore new solutions left by design, the code has still room for improvement, as suggested by the minima at around games 250 and 350, possibly linked to some attempts at optimization exploited by the opponent. The final better consistency with respect to the trained instance is remarkable, not only for the adaptation itself, which led to a comparable number of victories as the trained version, but also for the final result, which ended up as the best simulation with a 62% of win rate.

The code had to face a predictable opponent with a simplified decision making process; nevertheless, both players have very well built and all around balanced teams, meaning that such a result was far from obvious. Moreover the comparison between the two instances pointed out that this algorithm is suitable for playing

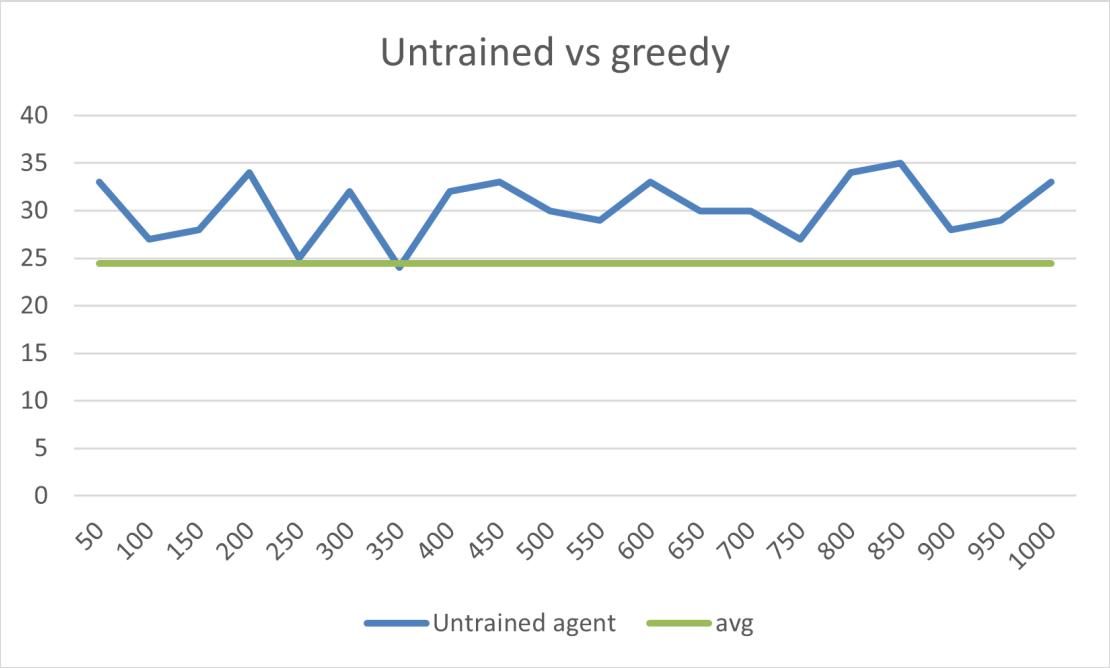

Figure 3.4: Plotting the number of wins every 50 games, the average is the axis of symmetry for the two players

against human players at high level, since the strategies are designed to follow a general pattern, with training providing a fair understanding of the game environment to reproduce competent tactics.

Chapter 4

Conclusions

Reinforcement learning proved to be a very effective tool to handle the proposed environment, efficiently learning the various options in the game and their consequences with little to no external intervention; however, the proposed solution comes with a number of caveats and issues that further highlight the limits of simple reinforcement learning. The first issue encountered when setting up the algorithm was how states were encoded and recognized: while it is rather easy to develop a parser to allow the code to gather more and more information on the game state and memorize a large amount of data to elaborate the strategy, the early implementation of the code struggled with symmetries and had to handle technically identical situations as separate occurrences, potentially with different strategies as well. Although this issue was patched later in development, it clearly showed a potential limitation in basic reinforcement learning algorithms, leading to additional work towards state recognition and modeling. This relatively smaller problem set back the initial training simulations for the mentioned reasons; however, additional states led to a significant increase in disk space requirements, more than doubling the size of the final files used for memory.

An additional aspect to be taken into account is how memory is managed in the code, in particular, as mentioned above, the way the algorithm saves each state is key for the performance of the agent; however, the amount of details stored when describing game states determines the final disk space required to store the memory table. A compromise had to be made between accuracy when recording a new state and memory requirements, affecting performance and containing the maximum learning table file size.

Finally, many more minor details may affect the overall performance of the algorithm, from the regular expressions used in the parser up to internal data structures and their management can be crucial when trying to improve performance, nevertheless reaction time is still acceptable and further optimization is due only to the data saving process, that may take many seconds to complete.

This project is an effective example of how reinforcement learning algorithms approach decision-making in uncertain conditions, however, as mentioned above, it suffered from some of the most well known issues of this approach[8]: bloating file sizes to store training and experience as well as pattern recognition issues that, while easily fixed in some cases, usually require a much more complex coding effort to correctly handle similar situations and potentially save some disk space. Overall the produced results were in line with expectations and they surely open to some improvements starting from the code written to tackle this problem, to the adoption of neural networks to a more efficient data management many ideas can be developed to further boost performance and potentially rival with experienced human players.

Moreover, due to the ongoing development of new games, together with the addition of new mechanics and rules, the software can be expanded to recognize the format of the game(s) it is playing and adapt to the increasing number of different categories where a variety of players may compete. This may lead the resulting algorithm to a deeper understanding of the game and potentially to some more accurate strategies in various scenarios proposed by each of the rule-sets used in separate competitions.

In addition to the proposed improvements, a very useful change that might be added for future testing is parallelism: with little modifications, by using multi-threading, the code might be able to process a larger number of agents during simulations, allowing for a further increase in performance during testing as well as giving the opportunity for multiple games on a public server, to improve the experience with real-life data combined with the original training.

Lastly, because of the very limited human intervention during training and across the many simulations, a similar approach might be exploited for different fields, involving random occurrences in the environment and non-deterministic actions during simulations; the key role for this transfer lies in the interface between the code and the environment, which requires a sufficiently large amount of control variables to accurately recognize and process states, actions and consequences.

Bibliography

[1] Peter Dayan and Yael Niv. «Reinforcement learning: The Good, The Bad and The Ugly». In: *Current Opinion in Neurobiology* 18.2 (2008), pp. 185–196. ISSN: 0959-4388. DOI: https://doi.org/10.1016/j.conb.2008.08.003. URL: https://www.sciencedirect.com/science/article/pii/S0959438808000767 (cit. on pp. ii, 2).

[2] Mark McKenzie, Peter Loxley, William Billingsley, and Sebastien Wong. «Competitive Reinforcement Learning in Atari Games». In: Aug. 2017. ISBN: 978-3-319-63003-8. DOI: 10.1007/978-3-319-63004-5_2 (cit. on p. ii).

[3] Georgios Yannakakis and Julian Togelius. «A Panorama of Artificial and Computational Intelligence in Games». In: *IEEE Transactions on Computational Intelligence and AI in Games* 7 (Jan. 2014), pp. 1–1. DOI: 10.1109/TCIAIG.2014.2339221 (cit. on p. ii).

[4] Scott Lee and Julian Togelius. «Showdown AI competition». In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. 2017, pp. 191–198. DOI: 10.1109/CIG.2017.8080435 (cit. on pp. iii, 10).

[5] Dan Huang and Scott Lee. «A Self-Play Policy Optimization Approach to Battling Pokémon». In: *2019 IEEE Conference on Games (CoG)*. 2019, pp. 1–4. DOI: 10.1109/CIG.2019.8848014 (cit. on pp. iii, 20, 21).

[6] David Silver et al. «A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play». In: *Science* 362.6419 (2018), pp. 1140–1144 (cit. on pp. iv, 5).

[7] Kush Khosla, Lucas Lin, and Calvin Qi. «Artificial Intelligence for Pokemon Showdown». PhD thesis. PhD thesis, Stanford University, 2017 (cit. on p. iv).

[8] Runjia Tan, Jun Zhou, Haibo Du, Suchen Shang, and Lei Dai. «An modeling processing method for video games based on deep reinforcement learning». In: *2019 IEEE 8th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*. 2019, pp. 939–942. DOI: 10.1109/ITAIC.2019.8785463 (cit. on pp. iv, 37).

[9] Cameron B Browne et al. «A survey of monte carlo tree search methods». In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43 (cit. on p. 8).

[10] Matsumoto Shimpei, Hirosuey Noriaki, Itonagaz Kyohei, Yokooz Kazuma, and Futahashiz Hisatomo. «Evaluation of Simulation Strategy on Single-Player Monte-Carlo Tree Search and its Discussion for a Practical Scheduling Problem». In: *Lecture Notes in Engineering and Computer Science* 2182 (Mar. 2010) (cit. on p. 8).

[11] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018 (cit. on p. 10).

[12] *Pokemon.com, Official rules for tournaments*. URL: `https://www.pokemon.com/us/play-pokemon/about/tournaments-rules-and-resources/` (cit. on p. 12).

[13] *Bulbapedia main page*. URL: `https://bulbapedia.bulbagarden.net/wiki/Main_Page` (cit. on p. 17).

[14] Jinsung Yoon, Sercan Arik, and Tomas Pfister. «Data valuation using reinforcement learning». In: *International Conference on Machine Learning*. PMLR. 2020, pp. 10842–10851 (cit. on p. 18).

[15] Kevin Chen and Elbert Lin. «Gotta Train 'Em All: Learning to Play Pokemon Showdown with Reinforcement Learning.» In: (2018). URL: `http://cs230.stanford.edu/projects_fall_2018/reports/12447633.pdf` (cit. on p. 19).

[16] Joseph Flaherty, Aaron Jimenez, Bahareh Abbasi, B Abbasi, J Flahery, and A Jimenez. «Playing Pokemon Red with Reinforcement Learning». In: (2021) (cit. on p. 21).

[17] Yu Bai and Chi Jin. «Provable self-play algorithms for competitive reinforcement learning». In: *International conference on machine learning*. PMLR. 2020, pp. 551–560 (cit. on p. 21).

[18] *Repository for Pokémon Showdown*. URL: `https://github.com/smogon/pokemon-showdown` (cit. on p. 24).

[19] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. «Reinforcement learning: A survey». In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285 (cit. on pp. 29, 32).

[20] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. «A unified game-theoretic approach to multiagent reinforcement learning». In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 32).