Deep Q-Learning for Pokémon Showdown

Tayyab Hussain

Supervisor: Tom Chothia
BSc Computer Science 2019/2020

1 Abstract

Pokémon Showdown is an open source battle simulator for the hit video game series Pokémon, used by thousands of players to practice their skills in preparation for tournaments or just for fun. Pokémon is an incredibly diverse and difficult game to master as it has a very large possible state space and requires an incredible amount of game knowledge and human prediction skills. Deep Q-learning techniques were used to see if this game is solvable by a machine or whether the problem is intractable for computers. There are many features that make Pokémon a potentially difficult game for machine learning to solve. For example, move trees cannot be easily created due to an inherent randomness factor to every move, such as missing a move, critical hits or damage rolls. This increases the state space drastically therefore simple expectimax algorithms will not produce the best results, and simple Q-learning is insufficient. The very top players in Pokémon Showdown predict their opponent's moves based on human psychology. There is not always an obviously correct move so choices cannot always be easily predicted, leading to moves based on "gut feeling". However, deep learning has shown extremely impressive results in games such as Go that also have an element of human psychology to them, which is why it was chosen as the preferred technique to solve this game.

Contents

2 Introduction

2.1 The Problem

Pokémon Showdown is an unofficial, open source battle simulator client that thousands of people use daily to play competitively. The appeal is that fans are able to play Pokémon for free without paying for and owning any of the other games. It offers very useful features such as sending commands via web sockets. As the game is open source, the code for the client is free to be modified.

Pokémon Showdown is a turn-based game where the aim is to knock out all 6 of the opponent's Pokémon by reducing each of their hit points (HP) to 0. Only one Pokémon is active for each player per turn.

There are over 800 different Pokémon to choose from, each with their own moves, statistics (more commonly referred to as "stats") and abilities. The player chooses 4 moves for each Pokémon to use from a given selection (there may be hundreds). Stats include HP, attack, defence, special attack, special defence and speed. Abilities are unique to each Pokémon, such as "intimidate" which lowers the attack stat of the opponent's current Pokémon.

Each Pokémon may have either 1 or 2 types. Types are based on elements such as water, fire, ground, electric, grass, etc. There is a total of 18 different types. Some types are strong against other types, whilst some types are weak against other types. As well as the Pokémon, each move has a type. For example, if a Pokémon uses a water move against a Pokémon of a fire type, double the amount of damage is dealt because the water type is strong against fire. Figure 1 shows a type chart that indicates all of the type weaknesses and strengths in the game. Humans can reason and quickly understand that water would theoretically beat fire however machines would have to understand this through playing the game and experimentation. This is just one of the problems that makes Pokémon Showdown difficult for artificial intelligence (AI) to tackle.

With enough exploration an AI could learn these type weaknesses, however, it would take a long time. A problem that AI struggles to deal with is understanding human choices. The game does not always obviously present what the best choice would be. This means there is a lot of room for human psychology, and this is incredibly prevalent at the top levels of play. There are also many luck-based elements. For example, some moves can miss or there may be a small chance of inflicting a critical hit which does more damage that intended. All damage dealt has a slight variance, making the game unpredictable at times. Intuitively, AI would struggle since it is unable to comprehend how humans think. However, current AI techniques are very good at spotting patterns. If the AI can reduce the game down to human patterns, then it could become better than the average player.

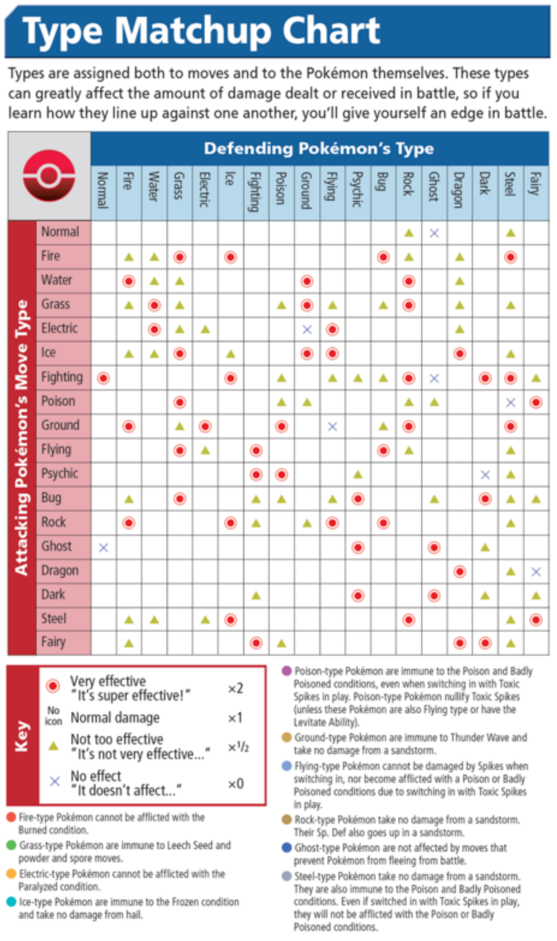
Pokémon Showdown also offers the ability to play random battles where the Pokémon on the player's team are randomly generated and balanced by level. By choosing this option, the AI can focus on learning to play the game rather than make teams, and will not have inaccuracies based on how good the team is.

The wide variety of options available to players means that there is a potentially enormous game state. Given this, an interesting question to ask is whether computers can perform better than human players.

2.2 Challenges

There are a number of challenges in trying to create an AI that learns to play Pokémon. The first is interfacing with the Pokémon Showdown client. The public servers are off-limits to non-developers and therefore some client code analysis and potential modifications will be required. Getting the game state might prove difficult since some of the data may not be stored on the client and server requests could be required. Since Pokémon Showdown is a browser application written in JavaScript, there could also be a problem passing the game data to the AI.

Pokémon Showdown is a game that requires a lot of game knowledge, which can only come through exploration. Given that there are so many different Pokémon (over 800) and many more possible move combinations and types, an AI would have to play



Type Matchup Chart

Types are assigned both to moves and to the Pokémon themselves. These types can greatly affect the amount of damage dealt or received in battle, so if you learn how they line up against one another, you'll give yourself an edge in battle.

	Defending Pokémon's Type																	
Attacking Pokémon's Move Type	Normal	Fire	Water	Grass	Electric	Ice	Fighting	Poison	Ground	Flying	Psychic	Bug	Rock	Ghost	Dragon	Dark	Steel	Fairy
Normal													▲	×			▲	
Fire		▲	▲	●		●						●	▲		▲		●	
Water		●	▲	▲					●				●		▲			
Grass		▲	●	▲				▲	●	▲		▲	●		▲		▲	
Electric			●	▲	▲				×	●					▲			
Ice		▲	▲	●		▲			●	●					●		▲	
Fighting	●					●		▲		▲	▲	▲	●	×		●	●	▲
Poison				●				▲	▲				▲	▲			×	●
Ground		●		▲	●			●		×		▲	●				●	
Flying				●	▲		●					●	▲				▲	
Psychic							●	●			▲					×	▲	
Bug		▲		●			▲	▲		▲	●			▲		●	▲	▲
Rock		●				●	▲		▲	●		●					▲	
Ghost	×										●			●		▲		
Dragon															●		▲	×
Dark							▲				●			●		▲		▲
Steel		▲	▲		▲	●							●				▲	●
Fairy		▲					●	▲							●	●	▲	

Key		
●	Very effective "It's super effective!"	×2
No icon	Normal damage	×1
▲	Not too effective "It's not very effective..."	×1/2
×	No effect "It doesn't affect..."	×0

● Fire-type Pokémon cannot be afflicted with the Burned condition.
● Grass-type Pokémon are immune to Leech Seed and powder and spore moves.
● Electric-type Pokémon cannot be afflicted with the Paralyzed condition.
● Ice-type Pokémon are immune to the Frozen condition and take no damage from hail.
● Poison-type Pokémon are immune to the Poison and Badly Poisoned conditions, even when switching in with Toxic Spikes in play. Poison-type Pokémon nullify Toxic Spikes (unless these Pokémon are also Flying type or have the Levitate Ability).
● Ground-type Pokémon are immune to Thunder Wave and take no damage from a sandstorm.
● Flying-type Pokémon cannot be damaged by Spikes when switching in, nor become afflicted with a Poison or Badly Poisoned conditions due to switching in with Toxic Spikes in play.
● Rock-type Pokémon take no damage from a sandstorm. Their Sp. Def also goes up in a sandstorm.
● Ghost-type Pokémon are not affected by moves that prevent Pokémon from fleeing from battle.
● Steel-type Pokémon take no damage from a sandstorm. They are also immune to the Poison and Badly Poisoned conditions. Even if switched in with Toxic Spikes in play, they will not be afflicted with the Poison or Badly Poisoned conditions.

Fig. 1: Pokémon type chart *Pokemon Type Chart - Sword and Shield - Pokemon Sword and Shield Wiki Guide* (2019)

many thousands of games to even become remotely good at the game. This would require the AI constantly training for days at a time. The game would have to run without any problems such as network issues, which cannot be guaranteed, therefore the AI would have to be able to pause training and continue it at a later point if needed.

The main problem, however, will be the learning. To become good at the game, the bot will need to be able to make a "good" move. To make a good move, it needs to be able to reason based on both what the opponent and the bot has available. This is difficult for even humans to do. A good move is difficult to characterise. Something that seems good now may not be good in a few turns and vice versa. Reinforcement learning has proved very good at spotting patterns that increase its reward so the AI may potentially find some optimal pattern of opening moves to maximise its score. For this pattern to be good, the reward function (the algorithm which gives reward for doing good actions) needs to capture the game well enough to lead to eventual victory. This is not easy as there is not an obviously correct move at each stage. Deep Q-learning is particularly good at finding optimal patterns however, so this may not be a huge problem after a lot of training.

2.3 Importance of the Solution

More and more problems are being solved by some application of AI techniques. It is very important to understand what AI can be applied to, and the limits of various techniques. Once a solution has been found for one type of problem, a similar approach can be taken for other analogous problems. Pokémon is a very high state space, prediction dependant and knowledge-based problem. If an AI can be shown to solve this problem, then this can be applied to similar games such as YuGiOh. Beyond gaming, a similar solution can be applied to motor functions for human-like movement or a high state space, prediction based problem.Mowforth & Bratko (2009). It could even suggest semi-optimal hyperparameters for the AI to solve those related problems.

Beyond real-life applications, AI can help show new strategies that human players did not previously know about or consider. This would change the way thousands of players look at the game. Pokémon Showdown is, by no means, a solved game. This means that an AI could shed some light on alternative strategies to win or show a method that has never been seen.

3 Literature Review

3.1 Deep Q-Learning

There are many different techniques to allow for machines to learn, however, one of the most recent developments and successes in the field of machine learning is deep Q-learning. This is a model that is based on reinforcement learning Sutton & Barto (1999) and Q-learning Watkins & Dayan (1992). Q-learning is a rudimentary solution for solving small state space problems effectively. Q-learning is also extremely good at solving solved games such as tic-tac-toe, however, most games are not like this. Deep Q-learning is a model that works with infinitely large state spaces. It uses the fundamental idea of Q-learning to maximise quality values, referred to as Q-values. These are values that are assigned to actions in a game or problem. The better an action is perceived to be in the game, the higher the given Q-value for the action. The process of setting up a system that uses neural networks to predict an action's given Q-value is laid out by Li (2017). They summarise how deep Q-learning has been applied to a wide variety of different problems, both gaming and non-gaming related. They layout the fundamentals of reinforcement learning. Reinforcement learning is represented in the following way.

t is the current time step in the problem. s_t is the game state in the current time step. a_t is the action taken at a given time step. $\pi(a_t|s_t)$ represents the policy which the AI will follow to map a state to an action. r_t is the reward given at a time step as a result of following the policy at that time step. The reward is accumulated and discounted by a discount factor γ that is bound between 0 and 1 thus

$R_{t+k} = \sum_k^\infty \gamma^k r_{t+k}$

This discounting is done to ensure that longer term solutions that give more reward over time will outweigh solutions that give points in the short term. They succinctly show that the optimal policy decomposes into the Bellman Equation

$q_*(s,a) = \sum_{(s',r)} p(s',r|s,a)[r + \gamma max_{a'} q_*(s',a')]$

where s' and a' are the following state and the following action respectively and $q*$ represents the optimal Q value.

This equation is the cornerstone of reinforcement learning and is used in almost every reinforcement learning application.

The aim of a deep Q-network is to use a neural network that takes input from a state. This is then fed forward through the network and the output action with the highest Q-value is chosen. However, this does not achieve much if the neural network cannot improve over time. There needs to be some way to find out what the optimum Q-value would be for every action. This is the right side of the Bellman Equation. This can be done by passing in the next state S' into the network and obtaining the maximum Q-value.

$Loss = q_*(s,a) - q(s,a)$

$Loss = E[R_{t+1} + \gamma max_{a'} q_*(s',a')] - \sum_k^\infty \gamma^k r_{t+k+1}$

This way, the loss can be calculated, and the neural network can maximise accuracy over time. This method does, however, have a problem. Using the same network to both estimate the Q-values and determine the maximum Q-value to calculate the loss is problematic. As the same weights are being used to calculate both the Q-value and the target Q-value, when the Q-values converge on the target values, the target values get further away. This is because the weights are adjusted. This can lead to many issues such as overestimation of Q-values in the long term and other instabilities. Hasselt et al. (n.d.) explains a solution to this by using a second network and various other optimisations to improve performance. Instead of using the same network for the target Q-values, they use a second network that they do not update every episode. This use of a target network has been shown to have excellent results when learning. The frequency of updates is a hyperparameter that can be tuned for accuracy.

3.2 AI for Perfect and Imperfect Information Turn Based Games

In recent years, developing AI to solve turn-based games, and games in general, has proven extremely popular and insightful. Numerous large companies invest a considerable amount of money into AI research. Google's DeepMind has been at the forefront of a large number of breakthroughs.

Go is a turn based game similar to chess (it is played on a board with pieces) however it has a significantly larger possible state space than chess does at 250^{150}, according to Li (2017). Go seemed to be a game that historically computers have struggled with. Silver et al. (2016) shows how successful the correct reinforcement learning techniques can be when applied to even the largest of state spaces. AlphaGo was designed using reinforcement learning and tree searches in order to master Go. Go is notorious for being a game in which it is difficult to determine what a "correct" move would look like. Many professional and top players even describe making moves based on a "gut feeling". In 2015, AlphaGo was able to beat the European champion in a non-handicapped, full-sized 19x19 board match. This is the first time a computer had beaten a professional human Go player. It went on to defeat the world champion Lee Sedol, an 18-time world champion of the game.

This highlights the potential of reinforcement learning techniques, such as deep Q-learning, on very high state space problems. However, Go is a perfect information game. Both players have all the information about the game state at all times. This makes it significantly easier to make decisions and is therefore easier for an AI to play.

Imperfect information games are those in which certain variables are either hidden or not available to the player. Games like these can be extremely difficult to play as there is a lack of information, so the game state isn't as informative. This causes a huge detriment to traditional AI techniques. However, Heinrich et al. (2016) explored the use of self-play with reinforcement learning to increasingly learn to be better than the opponent. This is extremely effective as, in these types of games, outwitting and outplaying the opponent is almost always the goal. This would be difficult to do without some training against opponents. Self-play is very advantageous as any patterns in either bot will improve the AI overall.

The game chosen by Heinrich et al. (2016) is poker. In poker, both the opponent's hand and any shared cards are hidden until the end of the round. This varies in different forms of the game. The player is forced to predict how good their hand is compared to the opponent's hand based on the opponent's behaviour. This seems like a psychological problem at first however spotting patterns in players is inherently a very good fit for neural networks. Moravcik et al. (2017) developed an AI to play heads-up no-limit hold 'em poker. It became the first computer to beat a professional player without any preconceived strategy. It uses a technique known as counterfactual regret minimisation (or CFR) that utilises self-play for imperfect games to increasingly reach a point of equilibrium known in game theory as the Nash Equilibrium. Self-play is an extremely powerful tool for imperfect information games, as these games typically rely on the opponent's behaviour.

The reward function is a very important part of reinforcement learning. If a good reward function isn't found, then the AI will not be aiming to learn the best features. For certain problems, there are not many behaviours that can lead to reward, therefore creating a sparse reward function. This is an issue for many games that are not continuous, such as most turn-based games. To clarify, if it is hard to give rewards, then it is hard for the AI to be able to learn. To combat this Jaderberg et al. (2016) demonstrated that using auxiliary tasks to "drip feed" the AI into doing the actions leads to high rewards and strong results. In essence, having additional actions and states that lead to bigger rewards should be rewarded but to a lesser degree. The paper outlines a labyrinth containing items that the AI must find in order to gain points. They solved the issue of the sparse reward function (and therefore created a non-sparse reward function) by giving rewards for various other items that would enable the AI to get to the items with the most reward. This technique is useful for all kinds of sparse reward functions however the auxiliary functions are specific to each individual problem.

4 Methodology

4.1 Pokémon Showdown Interface

4.1.1 How Pokémon Showdown Works

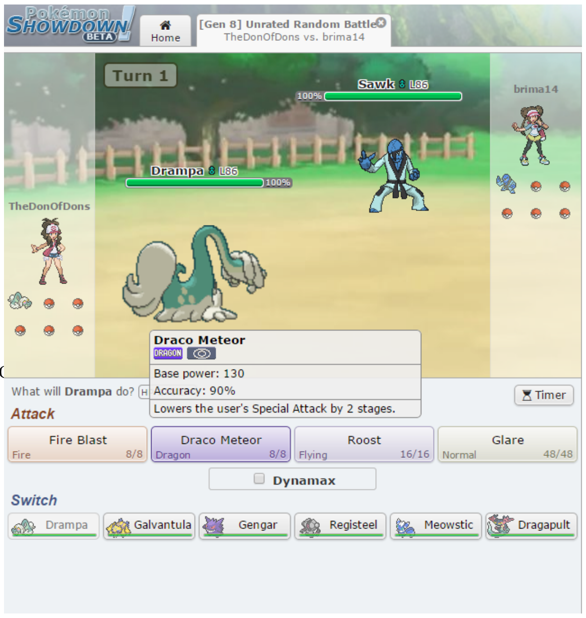
Pokémon Showdown is an unofficial, open source battle simulator, created to facilitate people who wish to play Pokémon competitively and for free. The Pokémon games themselves are adventure games in which the player must collect creatures known as Pokémo There are over 800 different Pokémon, each with their own stats, types and abilities. In the official Pokémon games, it can be difficult to collect rare Pokémon which have very specific moves and perfect stats. Therefore, the Pokémon Showdown client was created to give players a free and open place to make a team of Pokémon of their choice, allowing players to adjust any stats. The process of creating a team is a popular aspect in the competitive scene for the game.

The client was created in Node.js and has dedicated servers for people to play 24/7. These servers track each player's win/loss ratios as well as ratings for ranked modes and replays for all public matches. Pokémon Showdown's public GitHub repository also provides tools to create custom and private servers. Smogon (2020). This ensures that the tools for breaking down this client are readily accessible.

4.1.2 Sending Commands

There are numerous ways to send commands to the Pokémon Showdown client. The first is by pressing the buttons available on the client's graphical user interface (GUI). The client is a typical Node.js application. When in a battle, multiple choices can be seen representing what moves can be selected. Figure 2 shows that in the battle each Pokémon has 4 moves to choose from. If the players hovers over a potential move, some information about it is displayed. This includes the move's power (i.e. how strong the move is in terms of damage) and if there are any secondary effects when using the move.

There is a checkbox with the word "dynamax". In Pokémon, dynamaxing is something that can only



Base power: 130
Accuracy: 90%
Lowers the user's Special Attack by 2 stages.

Fig. 2: Pokémon Showdown's user interface

be done once per battle. It powers up a Pokémon for the duration of 3 turns, giving the Pokémon twice the usual HP and stronger moves. Both players may only use this once per game. Below this checkbox, the GUI presents the opponent's team of Pokémon. During the battle, the current Pokémon may be changed to another, at the cost of one turn. This is important in competitive play because it may be advantageous to swap to a resisting Pokémon, as discussed on the type-chart earlier, in order to reduce the damage taken. Finally, on the right of the GUI is a chat log of all the messages sent, moves completed and damage percentages so far throughout the battle. This is useful for referencing information about the battle so far, such as how much damage a specific previous move had dealt.

Another way of sending data to the client is via WebSocket commands. The developers of Pokémon Showdown have implemented a system in which users may send commands directly to the server for a range of tasks such as logging in, sending messages and playing matches. This is the primary method of communication for the AI.

4.1.3 Retrieving Data

One challenge is to retrieve the data from Pokémon Showdown that informs the AI's decision. There are multiple ways of retrieving this information. The first method captures the frame and passes the image to a convolutional neural network. This could be a good way for the program to learn as it sees the screen in exactly the same way as a human player does. This method would lead to a more realistic AI as it reacts to the same stimuli as human players would. However, this method also has some drastic downsides. Human players are able to hover over moves and Pokémon to get additional information about the game state. The AI would not be able to do this, as there is no implementation of this via WebSocket commands in Pokémon Showdown. There is also no need for image data to be passed as this isn't a motion-based or real-time game. It is an information focussed, turn-based game. Image data would make the training process more difficult than is necessary. Pokémon Showdown is a Node.js application and runs in the browser, making it difficult to directly retrieve data from the client and pass it to the AI. This means some client modification is necessary in order to force the client to send current the game state data to a local server, which can then pass the data to the AI. While this is a convoluted approach, it does allow fine tuning of the precise data being sent to the AI. It also requires decoding and understanding of how Pokémon Showdown works.

4.2 Analysis of the Game

4.2.1 Pokémon

Pokémon Showdown is a turn-based game that requires various skills in order to win. Such skills include predicting the opponent's behaviour, understanding the effects of different moves and planning ahead. In order to be successful, the AI must learn patterns that achieve all these skills.

Each Pokémon has 6 stats, each with unique values that are based on the Pokémon. These stats are HP (hit points), attack, defence, special attack, special defence and speed. HP is the health of the Pokémon. At the beginning of each battle, the Pokémon with the highest speed stat takes the first turn. Attack is used for damage calculation. In Pokémon, there are 2 different types of damaging attack - physical and special. The attack is used to calculate damage for the physical attacks, whereas the special attack stat is used to calculate damage from special attacks. The opponent's defence and special defence are also used in this calculation. The formula for damage is shown in Figure 4.2.1 where A is the attack stat of the player's Pokémon and D is the defence stat of the opponent's Pokémon.

In online random battles, the level of each Pokémon is an integer usually between 70 and 90, depending on how strong the Pokémon is. The stronger the Pokémon, the lower the level. The weaker the Pokémon, the higher the level. This is how Pokémon Showdown balances very powerful Pokémon against weaker ones.

Some moves give the player's Pokémon boosts. An example is "Swords Dance" which doubles the Pokémon's attack stat, or "Fishious Rend " which doubles damage dealt if the Pokémon goes first. Boosts may be accounted for differently by the damage equation shown in Figure 4.2.1/. The effect of Swords Dance would double the A value. The effect of Fishious Rend would be included in the modifier value. Boosts last until the Pokémon is knocked out or switched with another.

Boosts are an integral part of strategy in Pokémon Showdown because they enable the player to deal more damage or reduce the amount of damage taken. The modifier in the damage equation has the most impact on the damage dealt and therefore is one of the most important factors for the AI to learn. These behaviours would be difficult for the AI to learn, especially without any context. Therefore, some hints must be given to the AI via reward from the reward function to encourage the behaviour that enables the AI to knockout the opponent's Pokémon and gain more reward.

Each Pokémon has an ability. Each ability is active when either its Pokémon is active, or the ability's conditions are met. Whilst abilities don't usually a play a significant role in Pokémon Showdown, they may sometimes mean the difference between a win and a loss if not correctly accounted for. An example of an ability is "Moxie". A Pokémon with this ability has its attack increased for each time it knocks out an opponent's Pokémon. In this case, it would be wise to avoid allowing an opponent with this ability to repeatedly knockout the player's Pokémon. This would lead to a very large attack stat which would allow the opponent to knockout future Pokémon very easily. The AI must learn by experimentation which ability every Pokémon has and how to account for them. This takes many games and a lot of training, as there are over 200 abilities available.

Each Pokémon can be uniquely identified by its Pokédex number. The Pokédex is an official online database containing information about every Pokémon. The AI accesses the Pokédex in order to differentiate between different Pokémon and their abilities.

$$Damage = \left(\frac{\left(\frac{2 \times Level}{5} + 2\right) \times Power \times A/D}{50} + 2 \right) \times Modifier$$

Figure 4.2.1

4.2.2 Battle

In each battle of Pokémon Showdown, both players have 6 Pokémon on their team. Both players have only 1 Pokémon in play at any time. The aim is to knockout (reduce the HP to 0) all 6 of the opponent's Pokémon. This can be achieved through damaging or non-damaging moves. Moves that deal damage have a set power and type. Moves that deal no damage have special abilities which may boost stats or inflict special status conditions on the opponent. Each Pokémon may have a maximum of 1 status condition at any given time. These are listed below.

- Paralysis: Halves the Pokémon's speed and there is a 25% chance that the Pokémon will not be able to perform a move.
- Burn: Halves the attack stat of the Pokémon and the Pokémon is dealt 1/8 of their maximum HP in damage each turn.
- Freeze: While frozen, the Pokémon cannot perform a move. For each turn the Pokémon is frozen, the Pokémon has a 20% chance of thawing. This may happen during the initial turn that it is frozen.
- Poison and Bad Poison: If a Pokémon is poisoned, it is dealt 1/8 of their maximum HP in damage each turn. If a Pokémon is badly poisoned, then the damage dealt starts at 1/16 of their maximum HP and increases by 1/16 each turn (i.e. 1/16, then 2/16, then 3/16, etc.).
- Sleep: Whilst asleep, the Pokémon cannot perform a move. The Pokémon awakes at random within the duration of 3 turns.

Status conditions are not intuitive for an AI to learn. The AI must learn that both boosts and status conditions may lead to reward (and the knockout of an opponent's Pokémon) in both the short term and the long term. In other words, these may not deal a large amount of damage immediately but the effects of these will allow the AI to knockout the opponent's Pokémon sooner.

A very important factor in encouraging this behaviour is the reward function. The AI uses an epsilon-greedy strategy. At first, the AI will select moves and gain reward mostly at random. Over time, the AI uses inferred knowledge more. If a small reward is given after the use of moves involving status conditions or boosts, and this leads to a knockout or large reward, then the AI will learn that this behaviour gains more reward in the long term (despite the initially small reward) and that these moves are encouraged.

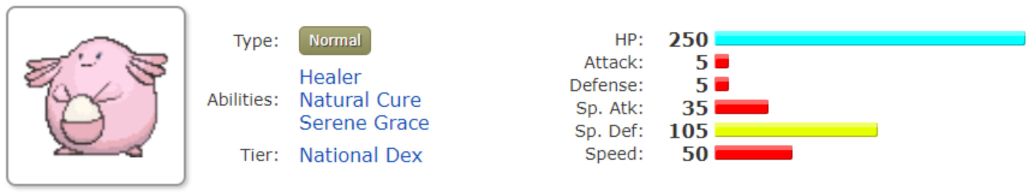
The speed stat of each player's initially selected Pokémon determines which player has the first turn in the battle. The Pokémon with the highest speed stat moves first. However, there are some moves that have priority and will move faster regardless of speed, enabling the opponent to perform a move first in this case. This is another factor that the AI must consider via experimentation. Given all the factors that the AI must consider, a very large neural network is required to encode all the details required when making a decision on which move has the highest Q-value.

When selecting the initial Pokémon, both players are unable to see each other's teams. Each team becomes visible only when a Pokémon is switched with another or when a move is performed. Pokémon is therefore an example of an imperfect information game (IIG). The AI does not have all the information about the game when playing. As discussed earlier, this can be a difficult problem to solve, but Moravcik et al. (2017) showed that self-play is an extremely good technique to solve IIGs. The idea here is that if an AI plays against itself many times then it learns optimal patterns of play to outwit itself, which could then be applied in practice against human opponents. Self-play also has many additional benefits, such as faster training compared to playing against humans (due to the slow nature of human decisions) and testing in a closed non-influenced state. Self-play means that the AI would have no knowledge of how humans play the game. This may be advantageous as the AI may discover new, previously unthought-of strategies to win. This may also be disadvantageous as the AI may not learn strategies played by human players. The CFR technique discussed in the paper would not be applicable since Pokémon is not a zero-sum game and has no Nash Equilibrium.

4.2.3 In-Game Strategy

Within Pokémon Showdown, there are many set strategies that players will employ depending on their team. Some Pokémon have stats that allow a particular play style whereas others don't. This depends on the Pokémon's moves and abilities and whether certain play styles would be effective. An example of a Pokémon having a specific play style is Chansey, show in Figure 4.2.3.

As shown in Figure 4.2.3, Chansey has very low stats in general, apart from a significantly high HP. It has the 2nd highest HP stat of all Pokémon, after Blissey who's evolution has 5 more HP. Chansey is considered better than Blissey as it has access to an item called eviolite. Chansey is considered to be a "wall" Pokémon. Some Pokémon are referred to as walls as they are incredibly difficult to break. Their aim and strategy is to stall the game. Chansey is considered a wall due to its pool of healing and status moves. The AI must know when to switch Pokémon and which Pokémon to switch to, considering whether a wall Pokémon would be the best strategy. It is very important for the AI to learn strategies concerning switching Pokémon. This sort of game knowledge will only be learned after training on thousands of games. An adequate reward function must be made to both account for these strategies and to ensure fast learning.



Type:	Normal
Abilities:	Healer Natural Cure Serene Grace
Tier:	National Dex

HP:	250
Attack:	5
Defense:	5
Sp. Atk:	35
Sp. Def:	105
Speed:	50

Figure 4.2.3

5 Implementation

5.1 Preparation and Data

5.1.1 Pokémon Showdown Modification

For an AI to be able to play Pokémon Showdown, a game state is required for each player after any given move has been performed. For normal video games, this information would be obtained from a frame or collection of frames from the game which would then be put through a convolutional neural network. However, since this game isn't a movement- or real-time-based game, there is no need for this. If the AI was to obtain game state data through frames, two issues are presented. The AI must learn features in order to obtain the data (which would take more time) and some information can only be obtained by hovering over specific areas of the screen. This would then have to incorporate mouse movements which adds another layer of unnecessary complexity. Instead of this, it would be much simpler and more efficient to modify the Pokémon Showdown client to send the raw data of the current state of the game.

The Pokémon Showdown Client is comprised of various TypeScript files which represent different aspects of the game. They are all tied together with a HTML and JavaScriptCore. Smogon (2020). The Battle.ts file manages the current battle that is taking place in the client. This is split up into multiple subclasses with "Battle" being the highest-level class holding the overall core methods relating to calculations, graphics and the user interface. Within this class there is the subclass "Side" which is a container for all information concerning the Pokémon and the player (the AI in this case). Within this, there is a Pokémon subclass that stores all the current known information about the Pokémon in the battle as well helper methods that can access any data about any Pokémon. This is used to get the Pokémon's initial stats (see appendix A).

The Pokémon Showdown client was modified to send the game state data as a JSON array to a Flask server running on a local machine. This was necessary there is no simple way of writing data from the browser in Node.js to a file on the machine. The Flask server then writes this JSON array to a file after each move has been performed to ensure the file contains the latest game state. JSON makes it easy to access specific elements of the game for pre-processing before being input to the neural network. An example of a JSON array element (a JSON object) is the following.

```
{species: Pok mon Dex Number,
type:[type1, type2]
stats:[hp, attack, defence,
special attack,special defence, speed],
move1:move1,
move2:move2,
move3:move3,
move4:move4,
hp:current HP
status: current Status effect
boosts: [b1,b2,b3,b4,b5,b6]
fainted: Boolean is Pok mon fainted?
active: Boolean is Pok mon active?
}
```

The JSON array is comprised of 12 of the above JSON objects and there is one JSON object for every Pokémon in the current game. The Pokémon Showdown client only has a limited amount of data however and therefore much of the above data is a null or 0 value. This null data is the result of Pokémon Showdown being an imperfect information game. All null values are changed to 0s as JSON does not send null values.

The code uses the built-in helper methods in Pokémon Showdown to retrieve this data and send it to the local server (see appendix B). The server then receives this data and writes it to a file. This JSON object is loaded from the file and prepared for deep learning.

5.1.2 Data Pre-Processing

The data pre-processing is done when the game state is requested by the agent. Firstly, the JSON array is loaded from the file. Then, the species numbers are loaded. As the 12 species are not available at the start of a match (as the opponent's Pokémon are not immediately visible), these must be padded with 0s until all 12 Pokémon are revealed. The species number is the unique identification number for each Pokémon. As there are over 800 Pokémon, this ID can get significantly large. This value must be normalised otherwise it would heavily influence the learning. Each species number is divided by 1500 to keep this value between 0 and 0.75. It was decided to not normalise the value to be between 0 and 1 as it makes sense intuitively that the species number has less weighting than other, more important values such as the Pokémon's type and stats. The aim is for the AI to learn features of the game itself rather than memorise specific Pokémon. These normalised IDs are stored in an array called "Pokémonids".

The next element to prepare for learning is the Pokémon types. Each Pokémon may have up to 2 types. The types of each opponent's Pokémon are hidden at the start of the battle. A Dictionary object contains all 19 possible types. Each type is given an ID which is then divided by 19 in order to normalise each value to be between 0 and 1. This method of normalisation is not perfect as certain types will have a higher value than others, which could be construed as these being "worth more". The solution for this could be to use a technique such as one-hot encoding (OHE). However, this would cause the state to become unnecessarily large and would affect learning. For states of this size, it is more suitable to keep the original normalisation method. A normalised value of between 0 and 1 is necessary to avoid saturating the neural network. Each Pokémon has an array of 2 type IDs that represents their typing. This is stored in an array called "typesarray".

The next step is to parse status conditions. Each Pokémon may only have 1 status condition at any given time. There are 7 different status conditions. A similar approach to the preparing the types can be taken. A Dictionary object is created, with the ID numbers of all the possible status conditions. Pokémon without a status condition are given a value of 0. At the start of the battle, every Pokémon's status condition has a value of 0. To normalise each status condition, each ID is divided by 7. These values are stored in an array called "statusEffects".

The next set of data required for the game state is the current HP of each Pokémon. These are automatically given as a number between 0 and 1 therefore no normalisation is required. At the start of the battle, all hidden Pokémon must have full HP and therefore this value is automatically 1. All Pokémon that have been knocked out have an HP value of 0. HP values are stored in an array called "hp".

All moves must also be prepared for learning. There is a significant amount of data involved here. Each move has a name, specific type and a power value. These are all important and each must be taken into account before a decision is made. The opponent's moves are hidden by default until a specific move has been performed and only this move is made visible to the player. For each Pokémon, the moves are stored in the following way.

```
[Move 1 ID, Move 1 Type,
Move 1 Power, Move 2 ID,
Move 2 Type, Move 2 Power,
Move 3 ID, Move 3 Type,
Move 3 Power, Move 4 ID,
Move 4 Type, Move 4 Power]
```

This data is stored for all 6 of the player's Pokémon. This data cannot be stored for the Pokémon as this information is not available on the client and is intended to be hidden. Despite this limitation, this information is not largely relevant when making a move decision and so it won't significantly hinder the AI. Each item of data in the array is normalised. Each move ID is divided by 1000 so that they are all approximately between 0 and 1. Each move's type is divided by 19 (which is the same normalisation for each Pokémon's type or types). Each move's power is divided by 250 to be between 0 and 1. This is because maximum power value is 250. This data is added to an array called "teammoves".

Boosts are very common and very important to keep track of. There are many boosts which may increase or lower certain stats and therefore they form a very important part of the game state and any decision making. During the battle, boosts may change the following stats: attack, defence, special attack, special defence and speed. Stats can be increased by 6 positive stages or decreased by 6 negative stages. Where this is no boost to stats, the boost can be thought of as being a multiplier of 1 or 2/2. For each stage increase to this value, the multiplier increases by 1/2 capping out at stage 6 (known as +6) which is a multiplier of 4. For each stage decrease however, the following formula is used:

$$Multiplier = \frac{2}{-stage + 2} \tag{1}$$

Therefore, if the stage is -2 then the Pokémon's stat will be 2/4 or half its previous value. This caps out at -6. The Pokémon Showdown client sends this data through as stages of boost (-6 to 6). These are stored in an array for each Pokémon and are normalised by dividing each by 6. These are stored in an array called "boosts".

Finally, the AI must know which of the 12 Pokémon that are currently active. This allows the AI to make decisions based on which Pokémon are in play. Therefore, is Pokémon is associated with a Boolean value. A value of 0 means that the Pokémon isn't currently active and a value of 1 means that it is. It is important for the AI to know which Pokémon it is attacking as well as to learn patterns in any switches. This data is stored in an array called "activeArray".

All data must be fed into a tensor to be processed by the neural network. First, the aforementioned arrays are concatenated to produce a large NumPy array of all the data in the game state.

```
obs_state =
[Pok monids, typesarray, hp,
statusEffects, boosts, teammoves,
activeArray]
```

This creates an array of 276 elements that can be passed into the nodes of a neural network for Q-value estimation.

5.2 AI

5.2.1 Deep Q-Learning

A modified version of deep Q-learning was chosen as the primary method of learning. This is because deep Q-networks (DQNs) have shown fantastic results in the realms of both agents in video games and in turn-based games. Deep Q-learning is flexible and allows for both self-play and online vs. player learning. It can run in the background with minimal use of resources unlike certain AI and expectimax algorithms which rely on heavy computation. The aim is to use DQNs to estimate the Q-value of any given move. This Q-value will represent the "quality" of performing a move. Over time, this DQN should increasingly improve until it is near perfect in choosing the move with the highest Q-value.

For this project, a double deep Q-network setup was chosen as it has been shown to optimise results and avoid the common problem of overestimation. The first network is referred to as the policy network and is responsible for establishing the optimal policy which should be followed to produce the best Q-values. The second network is referred to as the target network. This is used to estimate the second part of the Bellman Equation as discussed earlier. If both sides in the equation use the same network to calculate loss, this would result in overestimation. Using only one network means that when the policy weights are changed in order to move closer to the target, the target weights themselves change and move further away. Both the Q-value estimation and the target to move at the same time, leading to unstable learning. A separate target network is therefore used. The target network is periodically loaded with the same weights as the policy network but allows for the policy network to learn for the duration of a few episodes before taking the weights for much better results.

In order to learn from previous experiences, all actions taken as well as their rewards must be stored. This is called replay memory. After every action, an object is stored in replay memory which contains the action taken, the state before the action, the state after the action and the reward for the action. These objects are randomly sampled and fed through the policy network as a batch of multiple memories. The size of this batch has a significant impact on learning and is a hyperparameter which must be carefully chosen. A small batch size converges quickly but can lead to much more noise. A large batch size more accurately represents the reward function but takes longer to converge. Bengio & Yoshua (2012).

Using this information, a new technique was trialled. This technique involves performing self-play for the majority of training time with a high batch size and then doing online training with a small batch size. This aims to reduce any noise arising from initial exploration of the game and then learn quickly against human players.

The main training loop continues looping until a set number of episodes have been met. Within the episodic loop, there is a loop that loops over every time step within the episode. At each time step, an action is chosen by either taking a random action or by using the network to infer the action with the highest Q-value. The epsilon-greedy strategy determines whether the action taken is random or not. This is discussed later.

5.2.2 Network Structure

Designing an optimal network structure is another important factor in efficient learning. Factors such as network shape, the number of nodes and the number of layers can be the difference between an efficient network and a sub-optimal one. Here, the input layer has 276 nodes as this is the number of inputs from the game state. The number of hidden layers has been chosen to be two since two layers can represent any decision boundary to an arbitrary accuracy and can approximate a smooth mapping to almost any accuracy. This is thought to be more than sufficient for this problem.

5.2.3 Reward Function

The reward function defines what the AI will be rewarded for and therefore what the AI should be aiming to achieve. The reward function also defined what the AI is penalised for. This influences the AI's behaviour and therefore the reward function must drive the AI to try to win efficiently. In Pokémon Showdown, the win condition is to knockout all the opponent's Pokémon. Knocking out one of the opponent's Pokémon can be considered 1/6 th of a win. This isn't strictly true as knocking out certain Pokémon may be worth more than knocking out others. Knockouts lead to wins, but these are not a common occurrence and are difficult to achieve without guidance. This is why good behaviour is "drip fed" into the AI by giving it a small reward for moves that are going to help secure a knockout. Jaderberg et al. (2016). Super effective (SE) hits are moves that deal extra damage due to having a good type matchup, as shown in Figure 1. Generally, this is a good tactic to aim for in order to secure a knockout. This strategy should therefore be given a small amount of reward to incentivise the AI in doing this. The same logic applies for penalising "bad" moves. If the AI chooses to perform a bad move that could lead it further away from securing a knockout, this must be penalised slightly to discourage it from happening in the future.

```
Win: +200
Pok mon gets a knockout: +50
Pok mon gets a boost: +2
Pok mon deals a status condition: +2
Pok mon deals damage: +2
Pok mon lands an SE hit: +4
Pok mon faints: −10
Pok mon takes an SE hit: − 4
Pok mon performs a resisted hit: −4
Pok mon attempts an invalid move: −5
Loss: −100
```

Winning the battle rewards the AI with 200 points as this is what the AI should be aiming towards. 200 points is the largest reward and it should ensure that once the AI has experienced its first win, it will try hard to win again. Knockouts are the most important factor leading to a win which is why it rewards 50 points. Super effective hits, boosts and status inflictions are also "good" ways of leading to a knockout. These therefore reward 4 points, 2 points and 2 points respectively. These values were chosen to approximate the value of these actions in the game. This is difficult to choose, however, and is a rough approximation at best since there is no set way to quantitively calculate what each move is worth.

5.2.4 Exploration vs. Exploitation

As the AI plays against players, it is important that it explores as many actions as possible so that it can learn which moves are optimal and which are not. However, in an online match the AI must also win. There is a balance that must be struck between inferring the best move from the neural network or exploring new moves randomly to discover what they do. Clearly, at the start of training more exploration must be done as the neural network hasn't learned from any moves yet. Later on in the training cycle (once a lot of exploration has occurred) the AI should move on to using what it has learned in order to win matches, rather than exploring moves at random.

To achieve this, an epsilon-greedy strategy is employed with the following formula.

$$\epsilon_e + (\epsilon_s - \epsilon_e) \cdot e^{-x\epsilon}$$

where ϵ_s is the start value, ϵ_e is the end value and ϵ is the epsilon decay. x is the current episode. At each step, a random number is generated between 0 and 1. If the current epsilon value is larger than the random number, then a random action is taken. If the current epsilon value is less than the random number, then an inferred action is taken. Changing the epsilon start and epsilon end values clamp the possible decay rate between the start and the end values. Changing the epsilon decay value adjusts the rate at which epsilon will decay (see appendix C).

The epsilon value must be tweaked depending on the number of episodes in training. The initial training cycle is 1000 episodes of self-play and therefore an epsilon value of 0.003 is chosen. This value is chosen as after 750 games have been completed, there is approximately a 10% chance the AI will perform a random move. Once 750 games have been completed, a significant portion of learning will have been done and so the focus should be on using inferences to win the game.

6 Results

6.1 Tuning Hyperparameters

6.1.1 Learning Rate

The learning rate affects how much the Q-value estimation is changed by new data given to the neural network. It is important that the learning rate is not too high, or the network will not be able learn correctly and the loss values will be extremely wild. They may not even be able to converge. If the learning rate is too small, then convergence will be very small to the point where the loss may be stuck permanently. To ensure a good learning rate was chosen, 100 test games of self-play were played with rates of 0.001 and 0.0001.

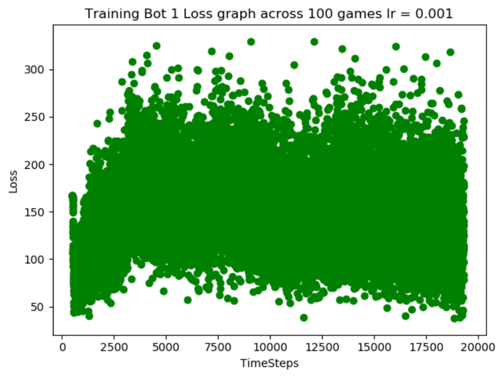


Fig. 3: Learning rate of 0.001

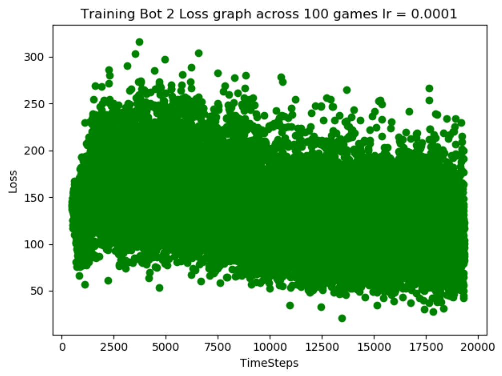


Fig. 4: Learning rate of 0.0001

From these results, it is clear that a learning rate of 0.001 produces wild and noisy results, whereas a learning rate of 0.0001 shows the formation of an ideal converging pattern. This indicates that 0.001 is a more optimal learning rate. Interestingly, for each battle taking place, the loss decreased dramatically as the game progressed. This may be because once the initial uncertainty of the hidden Pokémon ends, the game becomes much easier to predict.

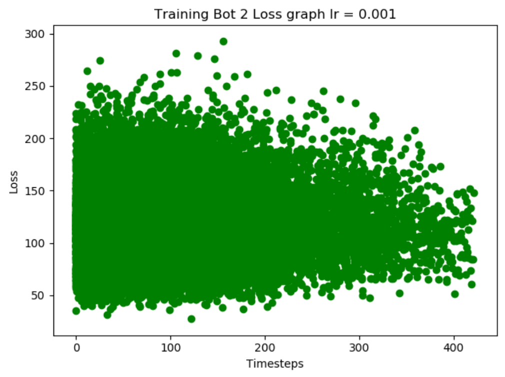


Fig. 5: Learning rate of 0.001 in each episode

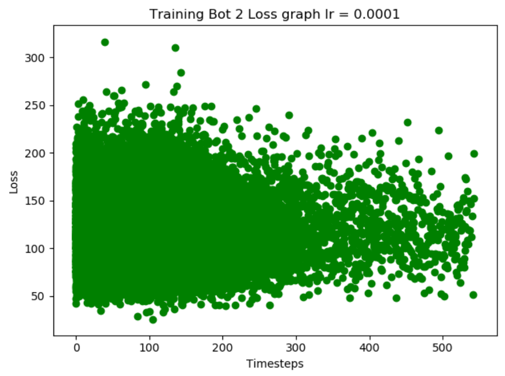


Fig. 6: Learning rate of 0.0001 in each episode

Figure 6 shows that a learning rate of 0.0001 results in faster convergence within each battle. Additionally, on average it took less time steps to reach a lower loss within each episode. This shows that the lower learning rate is better at predicting Q-values for moves in a game. For this reason, a learning rate of 0.0001 was used for further learning.

Since both learning rates do show convergence, they both could potentially be used in conjunction with different batch sizes to further improve results. The most optimal hyperparameters are problem specific and require significant experimentation when defining them.

6.1.2 Batch Size

The batch size plays a very important role in learning. A batch is a sample taken from replay memory which is put through the neural network for learning. If a big batch is used, learning and convergence is slower, but the results will not include as much noise. If a smaller batch size is used, the convergence will be much faster however there will usually be a lot of noise and wild loss results.

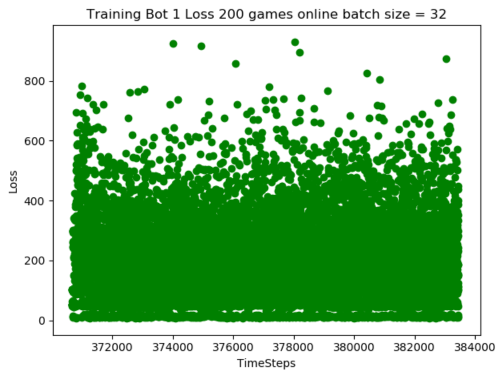


Fig. 7: 200 games with batch size 32

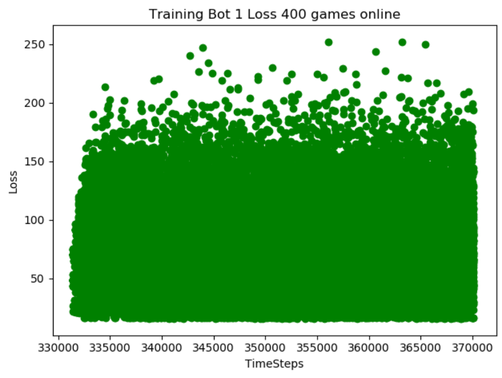


Fig. 8: 400 games with batch size 64

Figure 8 has much higher loss than Figure 7 but does contain much more noise. A batch size of 32 shows slightly more convergence but this is to be expected. Since both contain a considerable amount of noise, a higher batch size over a much longer training period is the most optimal way to train.

6.2 Performance Against Human Players

6.2.1 Win Rates

Win rates are the most important statistic when determining the strength of an AI agent in any game. In Pokémon Showdown, a win rate over 15% is impressive as the game is so vast, hidden and player dependant that even the best human players of the game struggle to keep their win rate above 70%. Furthermore, human decision-making must be predicted in order to win and this isn't an easy task for any AI as each human player is different.

After 1000 games of self-play with a high batch size of 500, the following results were achieved after 203 games against online human players.

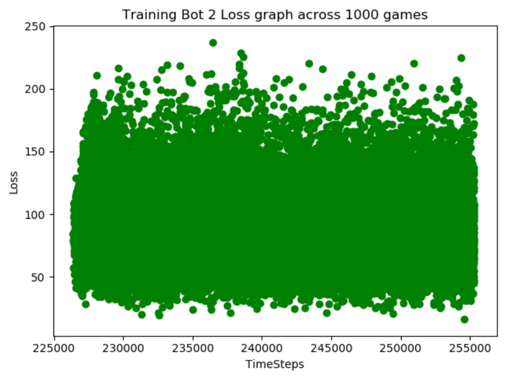
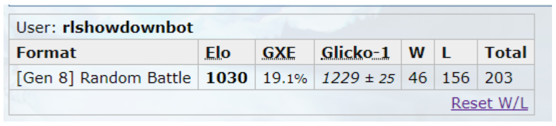


Fig. 9: Final 200 games of 1000 self-play games



User: **rlshowdownbot**

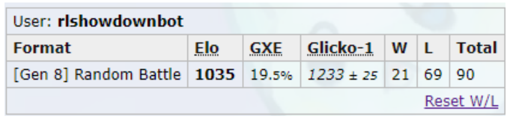
Format	Elo	GXE	Glicko-1	W	L	Total
[Gen 8] Random Battle	**1030**	19.1%	*1229 ± 25*	46	156	203

Reset W/L

Fig. 10: Win rate after 203 games

The graph in Figure 9 shows extremely slow convergence. This is to be expected as the batch size is very large (500). This was done to get more accurate results over a large number of games. A win rate of 22.6% was achieved which means that the AI won almost 1 in every 4 of its games with this method of training. This is an impressive result given the inherent human influence of a game like Pokémon Showdown and that only 1000 games were used to train. Typically, to achieve extremely good results with turn-based games, hundreds of thousands of games must be played.

There is a possibility that a lower batch size could provide better results in a shorter time despite being noisy. The following Figure shows the win rate with a batch size of 64 after only 200 games of self-play training and 90 games played online.



User: **rlshowdownbot**

Format	Elo	GXE	Glicko-1	W	L	Total
[Gen 8] Random Battle	**1035**	19.5%	*1233 ± 25*	21	69	90

Reset W/L

Fig. 11: Win rate after 90 games with batch size 64

A win rate of 23.3% was achieved out of 90 games. This shows that a lower batch size can lead to even better results than a well-trained agent with a higher batch size. Given more training this technique can improve the performance of the AI against online opponents significantly.

6.2.2 Patterns of Play

When observing the AI's behaviour during battles, the bot used some interesting techniques to help it achieve its victories. The first of which was the AI learning to dynamax early on in a game. The dynamax mechanic can only be used once in a match by each player and makes a Pokémon have double the HP and access to stronger moves for the duration of 3 turns. In particular, these stronger moves share secondary abilities that can boost the Pokémon's stats. As the Pokémon has double its usual HP, this allows for the Pokémon to "setup" with big boosts which attack the opponent's stats and sweep the opponent. The dynamax mechanic was criticised heavily by the competitive community for its ability to allow Pokémon to snowball into a victory. Since the AI would dynamax early, it was able to take advantage of the boosts and the added power to moves to get as much reward as possible. This also highlights the fact that the reward function works as intended.

Furthermore, the AI intelligently noticed a strategy that isn't commonly employed by many human players. Here, the AI switches Pokémon repeatedly to 2 or 3 different Pokémon in the team. Normally, this would cause the opponent to also switch their Pokémon to an optimal counter. This strategy reveals the Pokémon in the opponent's team. This information is extremely useful for decision making. Most human players don't understand the advantages of doing this as they are likely to take damage as they switch their Pokémon out. Despite this, the AI seemed to value the information gained much more than the damage taken.

Early on in training, the AI exploited an oversight in the reward function. Formerly, the reward function would only penalise an invalid move with a reward of -1. If the Pokémon were to perform an ineffective a move (due to typing) then the reward would be -2. A situation arose where the Pokémon was locked into performing only one move due to an item. This move would be ineffective against the opponent's Pokémon. The Pokémon could not be switched as the other Pokémon had fainted. In this case, the only reasonable action was to use the only available move. However, the reward function meant that the AI would repeatedly attempt an invalid move as the reward was better in this case. This showed the AI trying to maximise its reward correctly, despite no progress being made in the battle. This oversight was fixed, and the reward function was changed to reward -5 points for an invalid move.

7 Evaluation

The more training that is done, the better the accuracy and performance of the AI. Playing 1000 games is not enough training to produce an AI that is capable of beating the best players. Most turn-based games require hundreds of thousands of games and endless hyperparameter optimisations to reach that point. However, training itself takes a huge amount of time and resources. The AI is capable of playing around 200 online games a day at maximum capacity. This makes collecting results extremely slow and difficult. Obtaining 10,000 games worth of training would take 50 days of non-stop training which is impractical.

The results show interesting features in various aspects of the AI. Figure 8 shows 400 games worth of training with a batch size of 64 against Figure 7 showing 200 games with a batch size of 32. A batch size of 32 will converge much more quickly than a batch size of 64. However, this isn't necessarily a fair comparison. It would have been more accurate to compare these with the same number of games (ideally 400) as the more games are played, the more accurate the results will be. The hyperparameters in all these training cases can be tweaked and optimised. As well as a very long training time, it takes a large amount of trial and error to find good hyperparameters for any reinforcement learning problem, let alone deep Q-learning. The batch size, discount factor, learning rate, target update frequency and network structure can all be further optimised to produce much better and faster results, but this isn't easy to do as there is no definitive method to determine optimal parameters. They are specific to any given problem and usually require automated testing and trial and error or even heuristics to find.

Win rates are a good measure of how good an AI is at playing a game but looking at replay footage and understanding the process and decisions that an AI takes is a very good indicator as to how attempts to exploit the reward function. The replays on Pokémon Showdown must be manually scraped and these are reset every day. Without dedicated code, it can be difficult to obtain these replays.

Self-play may not always be the best way to train an AI. It has many advantages such as being self-contained and faster learning. However, if the task is to beat human players, then it may be much more advantageous to train online against human players. If the AI trains against human behaviours and patterns, this could make it better at the given task. There is no way to know which is better without extensive testing.

8 Conclusion

Self-play offers faster learning with advantages such as not being influenced by human players to help it learn the game better. Using two neural networks reduces the overestimation of Q-values to further improve accuracy. An epsilon-greedy strategy was employed to encourage exploration of the game state early on in the training cycle, but then encouraged more inference later on in order to learn optimal patterns of play. Taking the game state as just numbers in a tensor rather than passing a frame through convolutional layers also helped in terms of performance as this is already a large problem to attempt to solve.

An appropriate reward function has been exploited successfully by the AI and, as more games are played and more training is done, the AI will only get better at the game. The results show slow convergence, which indicates that more training is required.

Turn-based games such as Go have been shown to be beaten through AI techniques but for games with much larger possible state spaces, luck and human psychology is involved. It can be a difficult task to train an AI to beat such games. Pokémon Showdown is an example of such a game, requiring human prediction, game knowledge and long term thinking to win. Double deep Q-learning with self-play has shown to be a very promising, with win rates of 23% with only 400 games of training. Tuning hyperparameters, playing more games and better training techniques could easily bring this number up to more than 50% with better loss convergence and less noise.

Overall, the results produced have shown the project to be a success in training an AI to play Pokémon Showdown at a reasonable level. Playing more games and further tuning parameters is required to improve the ability of the AI further. However, results obtained show that a simple increase in the number of games played will improve the AI further with no obvious changes to the existing implementation.

References

Bengio & Yoshua (2012), 'Practical recommendations for gradient-based training of deep architectures'.
URL: *https://arxiv.org/abs/1206.5533*

Hasselt, H. V., Guez, A. & Silver, D. (n.d.), 'Deep reinforcement learning with double q-learning'.
URL: *https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389/11847*

Heinrich, Johannes & David (2016), 'Deep reinforcement learning from self-play in imperfect-information games'.
URL: *https://arxiv.org/abs/1603.01121*

Jaderberg, Max, Mnih, Volodymyr, Czarnecki, Marian, W., Schaul, Tom, Leibo, Z, J. & et al. (2016), 'Reinforcement learning with unsupervised auxiliary tasks'.
URL: *https://arxiv.org/abs/1611.05397*

Li, Y. (2017), 'Deep reinforcement learning: An overview'.
URL: *http://arxiv.org/abs/1701.07274*

Moravcik, M., Schmid, M., Burch, N., Lisý, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M. & Bowling, M. (2017), 'Deepstack: Expert-level artificial intelligence in heads-up no-limit poker', *Science* **356**(6337), 508–513.
URL: *https://science.sciencemag.org/content/356/6337/508*

Mowforth, P. & Bratko, I. (2009), 'Ai and robotics; flexibility and integration: Robotica'.
URL: *https://www.cambridge.org/core/journals/robotica/article/ai-and-robotics-flexibility-and-integration/9B7732C5E1B827A0B1866BAE940782C3*

Pokemon Type Chart - Sword and Shield - Pokemon Sword and Shield Wiki Guide (2019).
URL: *https://uk.ign.com/wikis/pokemon-sword-shield/*$Pokemon_Type_Chart\_{}_Sword_and_Shield$

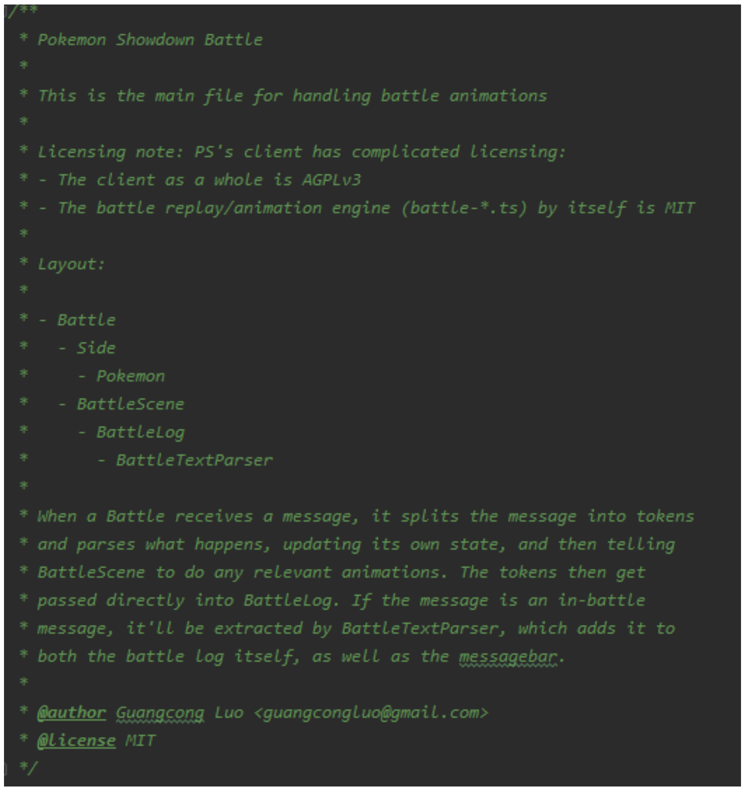
Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G. & Schrittwieser, J. (2016), 'Mastering the game of go with deep neural networks and tree search', *Nature* **529**(7587), 484–489.
URL: *https://doi.org/10.1038/nature16961*

Smogon (2020), 'Pokemon showdown'.
URL: *https://github.com/smogon/pokemon-showdown-client*

Sutton, R. & Barto, A. (1999), 'Reinforcement learning: An introduction'.
URL: *https://www.cell.com/trends/cognitive-sciences/fulltext/S1364-6613(99)01331-5*

Watkins, C. J. C. H. & Dayan, P. (1992), 'Q-learning', *Machine Learning* **8**(3), 279–292.
URL: *https://doi.org/10.1007/BF00992698*

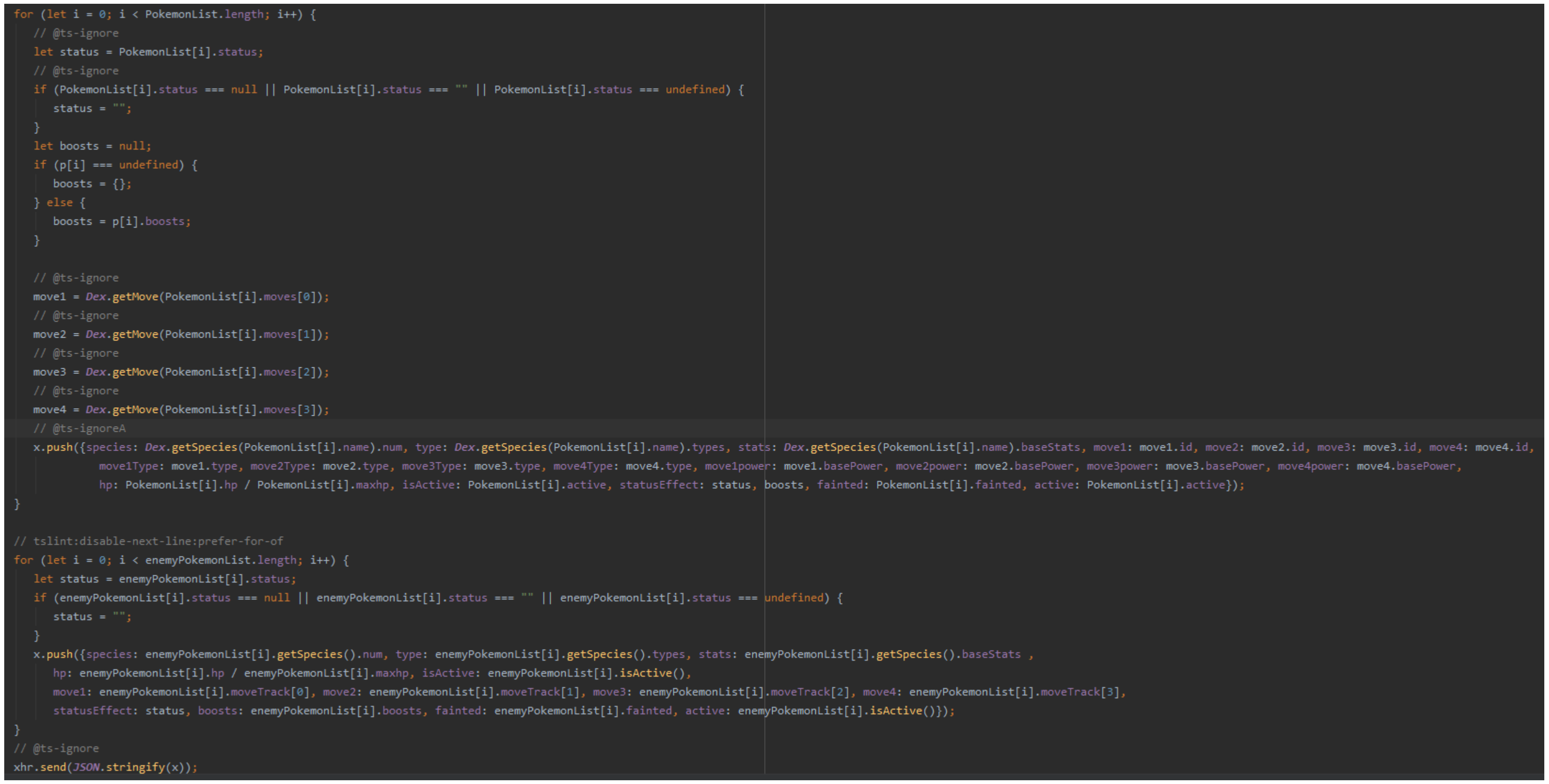
A Pokémon Showdown Structure



```
/**
 * Pokemon Showdown Battle
 *
 * This is the main file for handling battle animations
 *
 * Licensing note: PS's client has complicated licensing:
 * - The client as a whole is AGPLv3
 * - The battle replay/animation engine (battle-*.ts) by itself is MIT
 *
 * Layout:
 *
 * - Battle
 *   - Side
 *     - Pokemon
 *   - BattleScene
 *     - BattleLog
 *       - BattleTextParser
 *
 * When a Battle receives a message, it splits the message into tokens
 * and parses what happens, updating its own state, and then telling
 * BattleScene to do any relevant animations. The tokens then get
 * passed directly into BattleLog. If the message is an in-battle
 * message, it'll be extracted by BattleTextParser, which adds it to
 * both the battle log itself, as well as the messagebar.
 *
 * @author Guangcong Luo <guangcongluo@gmail.com>
 * @license MIT
 */
```

Fig. 12: Structure of Battle.ts

B Pokémon Showdown Modifications

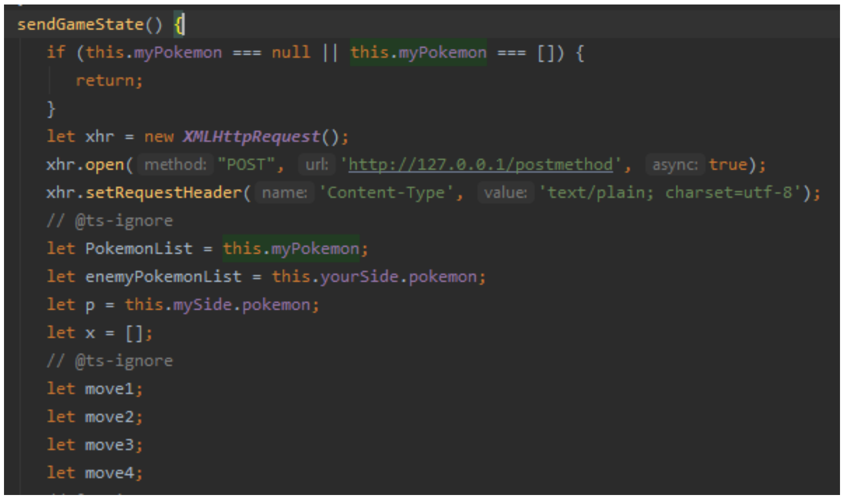


```
for (let i = 0; i < PokemonList.length; i++) {
   // @ts-ignore
   let status = PokemonList[i].status;
   // @ts-ignore
   if (PokemonList[i].status === null || PokemonList[i].status === "" || PokemonList[i].status === undefined) {
      status = "";
   }
   let boosts = null;
   if (p[i] === undefined) {
      boosts = {};
   } else {
      boosts = p[i].boosts;
   }

   // @ts-ignore
   move1 = Dex.getMove(PokemonList[i].moves[0]);
   // @ts-ignore
   move2 = Dex.getMove(PokemonList[i].moves[1]);
   // @ts-ignore
   move3 = Dex.getMove(PokemonList[i].moves[2]);
   // @ts-ignore
   move4 = Dex.getMove(PokemonList[i].moves[3]);
   // @ts-ignoreA
   x.push({species: Dex.getSpecies(PokemonList[i].name).num, type: Dex.getSpecies(PokemonList[i].name).types, stats: Dex.getSpecies(PokemonList[i].name).baseStats, move1: move1.id, move2: move2.id, move3: move3.id, move4: move4.id,
            move1Type: move1.type, move2Type: move2.type, move3Type: move3.type, move4Type: move4.type, move1power: move1.basePower, move2power: move2.basePower, move3power: move3.basePower, move4power: move4.basePower,
            hp: PokemonList[i].hp / PokemonList[i].maxhp, isActive: PokemonList[i].active, statusEffect: status, boosts, fainted: PokemonList[i].fainted, active: PokemonList[i].active});
}

// tslint:disable-next-line:prefer-for-of
for (let i = 0; i < enemyPokemonList.length; i++) {
   let status = enemyPokemonList[i].status;
   if (enemyPokemonList[i].status === null || enemyPokemonList[i].status === "" || enemyPokemonList[i].status === undefined) {
      status = "";
   }
   x.push({species: enemyPokemonList[i].getSpecies().num, type: enemyPokemonList[i].getSpecies().types, stats: enemyPokemonList[i].getSpecies().baseStats ,
      hp: enemyPokemonList[i].hp / enemyPokemonList[i].maxhp, isActive: enemyPokemonList[i].isActive(),
      move1: enemyPokemonList[i].moveTrack[0], move2: enemyPokemonList[i].moveTrack[1], move3: enemyPokemonList[i].moveTrack[2], move4: enemyPokemonList[i].moveTrack[3],
      statusEffect: status, boosts: enemyPokemonList[i].boosts, fainted: enemyPokemonList[i].fainted, active: enemyPokemonList[i].isActive()});
}
// @ts-ignore
xhr.send(JSON.stringify(x));
```

Fig. 13: Structure of Battle.ts



```
sendGameState() {
    if (this.myPokemon === null || this.myPokemon === []) {
        return;
    }
    let xhr = new XMLHttpRequest();
    xhr.open( method: "POST",  url: 'http://127.0.0.1/postmethod',  async: true);
    xhr.setRequestHeader( name: 'Content-Type',  value: 'text/plain; charset=utf-8');
    // @ts-ignore
    let PokemonList = this.myPokemon;
    let enemyPokemonList = this.yourSide.pokemon;
    let p = this.mySide.pokemon;
    let x = [];
    // @ts-ignore
    let move1;
    let move2;
    let move3;
    let move4;
```

Fig. 14: Structure of Battle.ts

C Epsilon Decay

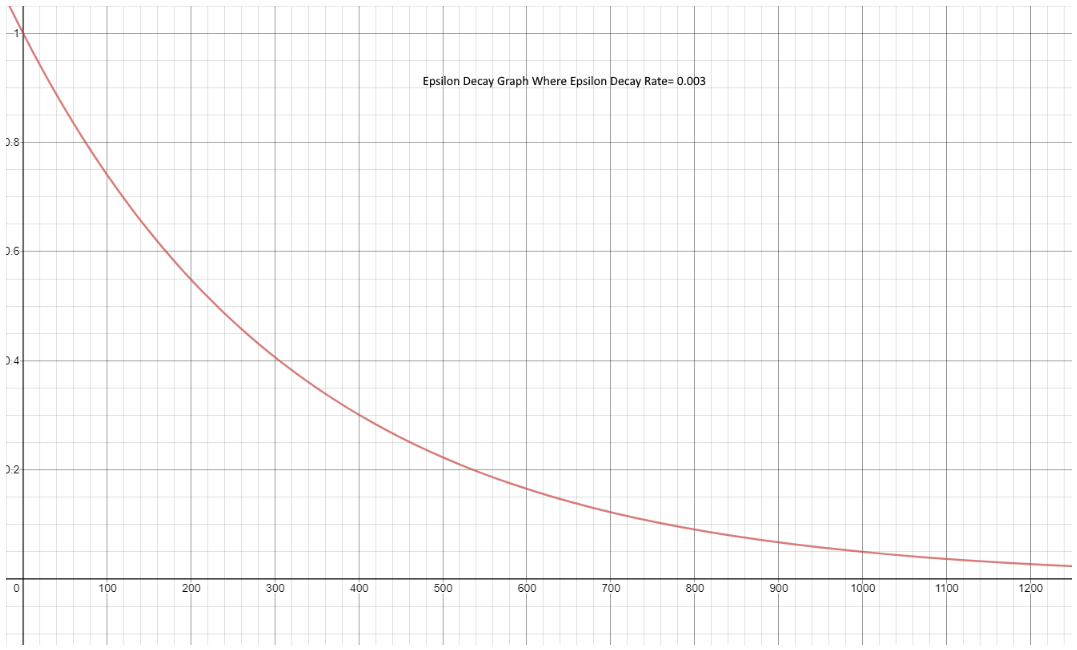


Fig. 15: Epsilon Decay Rate=0.003

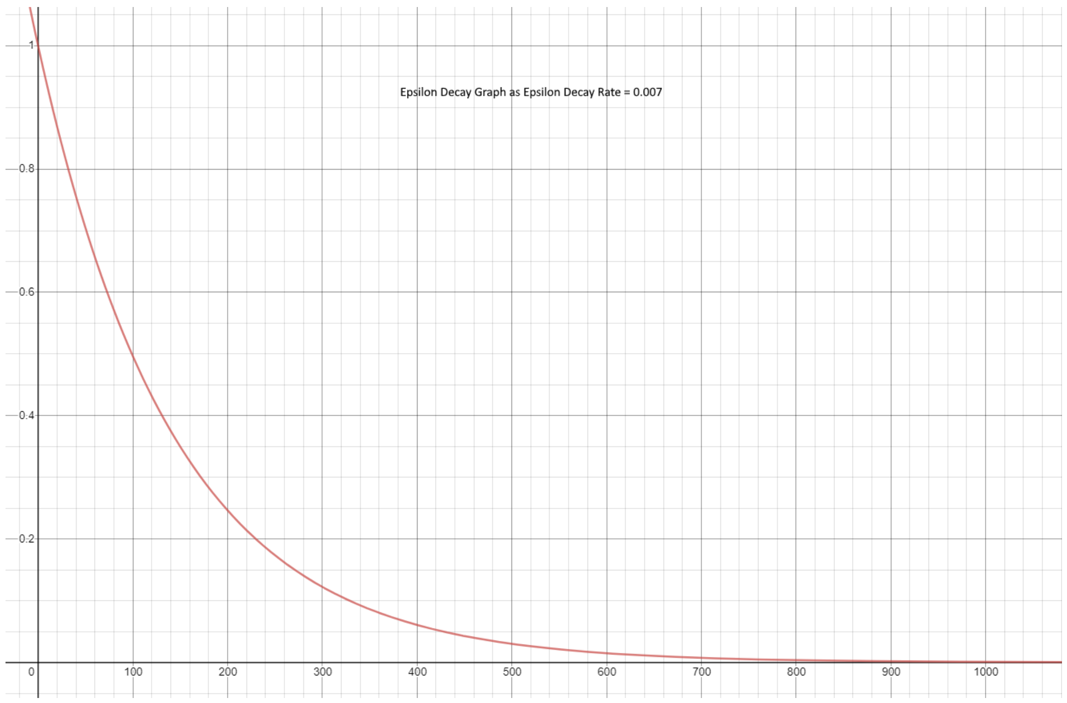


Fig. 16: Epsilon Decay Rate=0.007