Paarth Tandon December 12, 2022

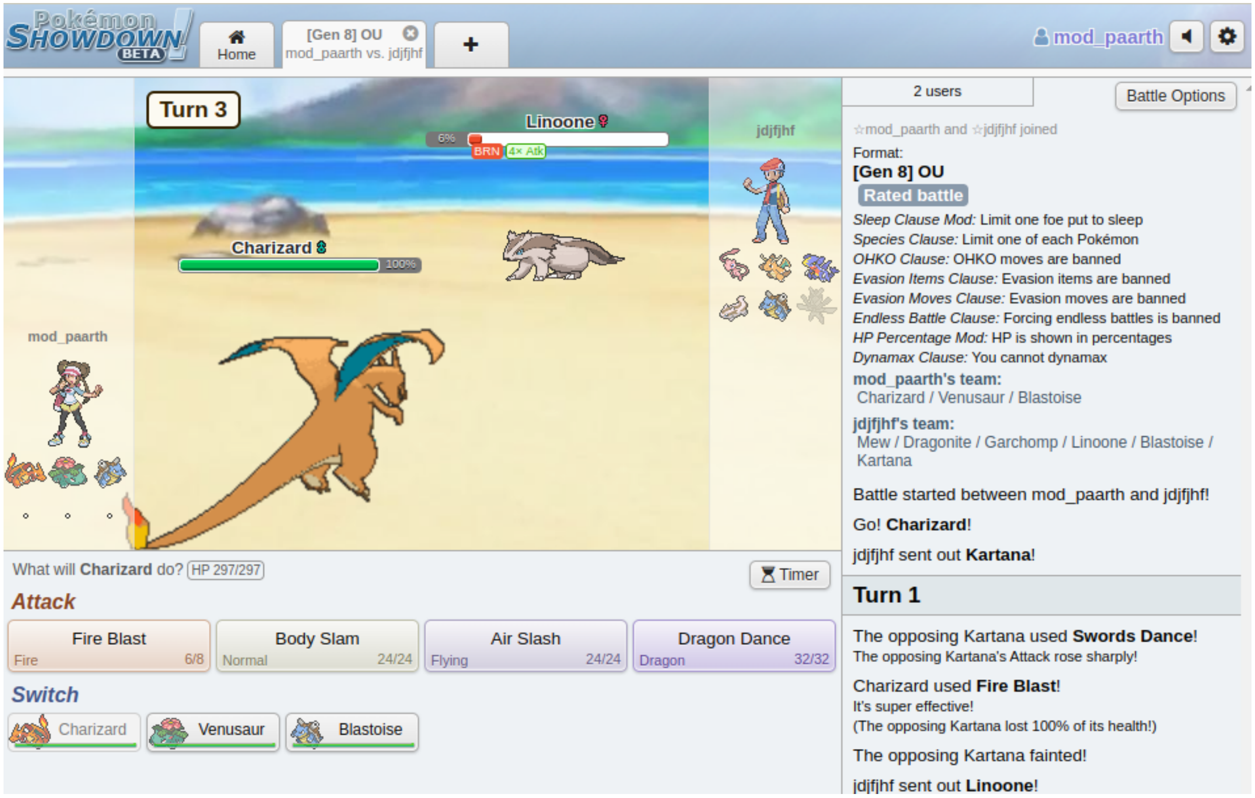
Pokemon: Turn-Based Strategy using RL

1. Preface

I include this preface as a note to any course staff that is reading this project. This has been both the most difficult, yet interesting project I have worked on in my time at UMass. I have poured countless hours into working on it, most likely over 100 hours. While it was very difficult, at the same time it was very entertaining and interesting. Overall, I am proud of my work. Thank you for running this course, as I most definitely plan to continue this project and work on more RL projects in the future.

2. Introduction

a. Pokemon Battles



☆mod_paarth and ☆jdjfjhf joined

Format:
[Gen 8] OU
Rated battle
Sleep Clause Mod: Limit one foe put to sleep
Species Clause: Limit one of each Pokémon
OHKO Clause: OHKO moves are banned
Evasion Items Clause: Evasion items are banned
Evasion Moves Clause: Evasion moves are banned
Endless Battle Clause: Forcing endless battles is banned
HP Percentage Mod: HP is shown in percentages
Dynamax Clause: You cannot dynamax

mod_paarth's team:
Charizard / Venusaur / Blastoise

jdjfjhf's team:
Mew / Dragonite / Garchomp / Linoone / Blastoise / Kartana

Battle started between mod_paarth and jdjfjhf!

Go! **Charizard**!

jdjfjhf sent out **Kartana**!

Turn 1

The opposing Kartana used **Swords Dance**!
The opposing Kartana's Attack rose sharply!

Charizard used **Fire Blast**!
It's super effective!
(The opposing Kartana lost 100% of its health!)

The opposing Kartana fainted!

jdjfjhf sent out **Linoone**!

Pokemon is a turn based strategy game that has been around since 1996. Over the years it has been continuously updated and modernized. The game itself involves traversing a map to collect and battle creatures known as Pokemon. This project focuses on creating an AI to play the battling portion of the game. Pokemon can be thought of as characters that each have different strengths and weaknesses, as well as different abilities. These strengths and weaknesses are known as the Pokemon's "stats", while its abilities are often referred to as its "attacks". Pokemon can also have up to two typings, which affect how much damage they deal and receive. For example, Charizard is a fire and flying type. This means that it will take more damage from water (water is strong to fire) and electric (electric is strong to flying) type attacks. Attacks also have typings, but Pokemon of a certain type can often have attacks of types that are not its own. For example, Charizard is fire and flying, but it can learn a normal type attack, Body Slam. Some attacks deal direct damage, while others can provide buffs and debuffs to a Pokemon's stats. Each turn, both Pokemon will have a chance to attack. If the player chooses, they may also switch out to a different Pokemon instead of attacking. When a Pokemon's health points reach zero, it faints and can no longer be used. Players can have teams of up to six Pokemon. When all of one team's Pokemon faint, that team loses. As you can see, Pokemon is not a simple game, and can often require a high skill level. It is also played competitively. To fully explain all game mechanics would take far too long, so if you are interested in reading it here is a good explanation: Battling Basics.

b. Previous Work

Using RL to battle Pokemon is a very open problem. When searching the internet for previous work, you will find other university students, like me, working on it as a class project, or just for fun. These attempts are usually unsuccessful. However, there is one paper out that demonstrates some success applying self play on a very large model. Doing something like this would be out of my reach, as I am training all my models on a laptop with no GPU. This severely constraints the size of my models and how long I can train them for.

c. Environment

As seen in section three of this report, I created two environments during this project. Since Pokemon is a very complex and highly stochastic game, I opted to use a premade simulator instead of coding the simulation from scratch. This is a very popular simulator that is written in Node.js, called Pokemon Showdown. It is used by millions of people on the official servers to play online against other humans, but can also be downloaded from GitHub and run locally. This step is crucial, as now I can remove any rate limiting and bot protection. This will allow me to simulate battles very fast, under one battle a second.

But there is still a problem: How am I going to use Python to train models on this server? To solve this I used poke-env, a communication layer between the Showdown server and Python. This library allows the user to define OpenAI Gym environments that connect the agent to the Showdown server. Note that the environments still have to be written by the user, defining things like state space, action space, and reward definitions. It simply exposes objects to the user that contain different aspects of the battle running on Showdown.

d. Team Composition

As there are over 1000 available Pokemon in 2022, there is no way that I would be able to create a state representation that accounts for all of them. Because of this, I limited my battles to only use three Pokemon. Each player in the battle will have an identical team as their opponent:

Charizard Ability: Blaze	Venusaur Ability: Overgrow	Blastoise Ability: Torrent

EVs: 252 Atk / 4 SpA / 252 Spe Mild Nature Attacks: - Fire Blast - Air Slash - Body Slam - Dragon Dance	EVs: 4 Atk / 252 SpA / 252 Spe Hasty Nature Attacks: - Giga Drain - Sludge Bomb - Body Slam - Curse	EVs: 252 HP / 252 Atk / 4 SpD Adamant Nature Attacks: - Aqua Tail - Avalanche - Body Slam - Roar

Please refer back to the battling guide if you want to learn more about what these mean.

e. Opponent Agents

The RL agents will be training against three different hard coded agents. These agents were provided by poke-env. The first agent is a random agent. This agent simply picks a random action each turn. The second agent is a max-damage agent. The random agent should be trivial to beat. This agent naively picks the attack which has the highest base damage. This agent acts around the level of a child. The third agent is a rule based agent that uses many heuristics to decide the best attack or switch. This agent plays similarly to an intermediate player of the game. For the current team composition, the heuristic agent is essentially optimal.

f. Algorithms

Discrete-state algorithms implemented: Q-Learning, SARSA
Continuous-state algorithms implemented: DQN (Double DQN), One-Step Actor-Critic

All algorithms mentioned in this report are implemented by me. Note, as per instructor permission, the neural networks in DQN and Actor-Critic are defined using PyTorch.

3. Environment Formulation

a. Discrete State Environment

The first environment I created is the discrete state environment. This is a simple environment that includes minimal information about the game.

The state of this environment is a vector of length four:

[RL Player's active Pokemon ID, RL Player's active Pokemon health,
Opponent's active Pokemon ID, Opponent's active Pokemon health]

The ID is 0 for Charizard, 1 for Venasaur, and 2 for Blastoise. The health is the fraction of health points remaining, uniformly binned into four categories.

The available actions are:

attack 0, attack 1, attack 2, attack 3, switch to Charizard, switch to Venusaur, switch to Blastoise.

The reward is defined as:

(1 * health points damaged) + (-1 * health points lost) + (10 * opponent fainted Pokemon) + (-10 * player fainted Pokemon) + (100 if win) + (-100 if loose)

The reward is awarded to the agent after each action based on this formula.

This is a very simple state representation, and should not be sufficient information to be an advanced player of the game, but is very easy to train. Overally, the Q dictionary holds ~1000 parameters.

b. Continuous State Environment

The second environment I created is the continuous state environment. This environment provided somewhat more detailed information about the game state than the previous discrete environment.

The state of this environment is a vector of length eight:

[RL Player's active Pokemon ID,
RL Charizard's health, RL Venusaur's health, RL Blastoise's health,
Opponent's active Pokemon ID,
Opponent Charizard's health, Opponent Venusaur's health, Opponent Blastoise's health
]

The ID is 0 for Charizard, 1 for Venasaur, and 2 for Blastoise. The health is the fraction of health points remaining, **not** binned this time.

The available actions are the same as before:

attack 0, attack 1, attack 2, attack 3, switch to Charizard, switch to Venasaur, switch to Blastoise.

The reward is defined as:

(1 * health points damaged) + (-1 * health points lost)

The reward is awarded to the agent after each action based on this formula. Note that this reward function is much simpler. It turns out that the gradient based methods used in this environment are better suited for a simpler, smooth reward based on testing.

This is still a simple state representation, and is still not sufficient information to be an advanced player of the game, but is very easy to train on my laptop.

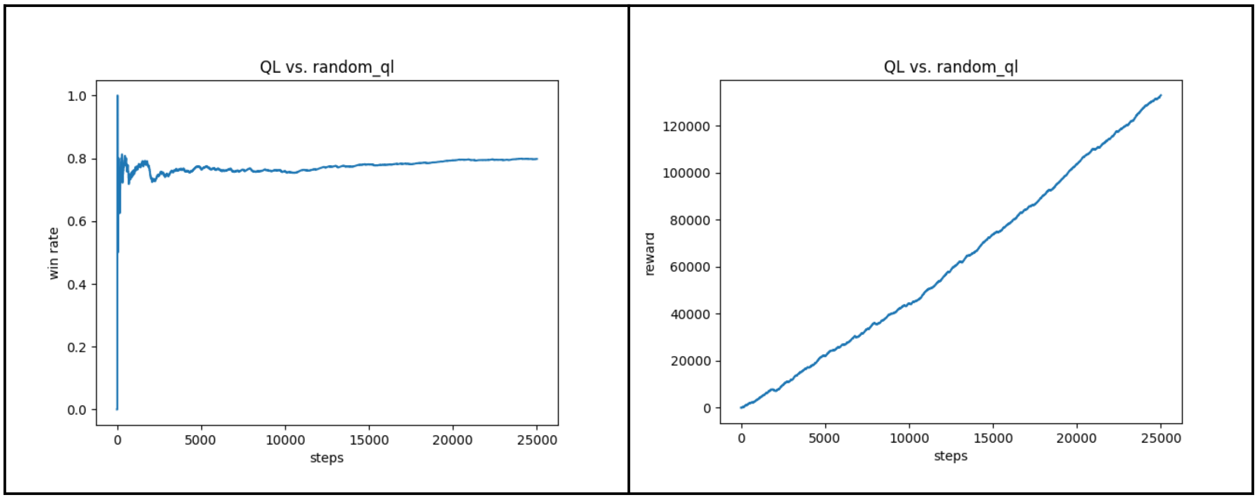
4. Results

a. Discrete State Representation

Q-Learning

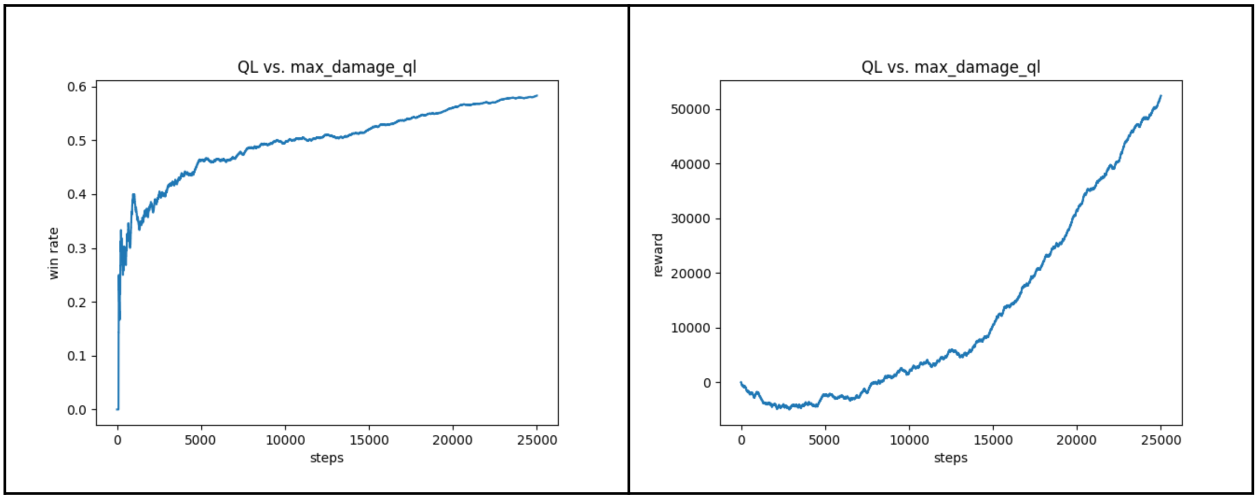
The best hyperparameters found while training were a constant epsilon of 0.1 and a constant learning rate of 0.1.

Results for training against the random agent:



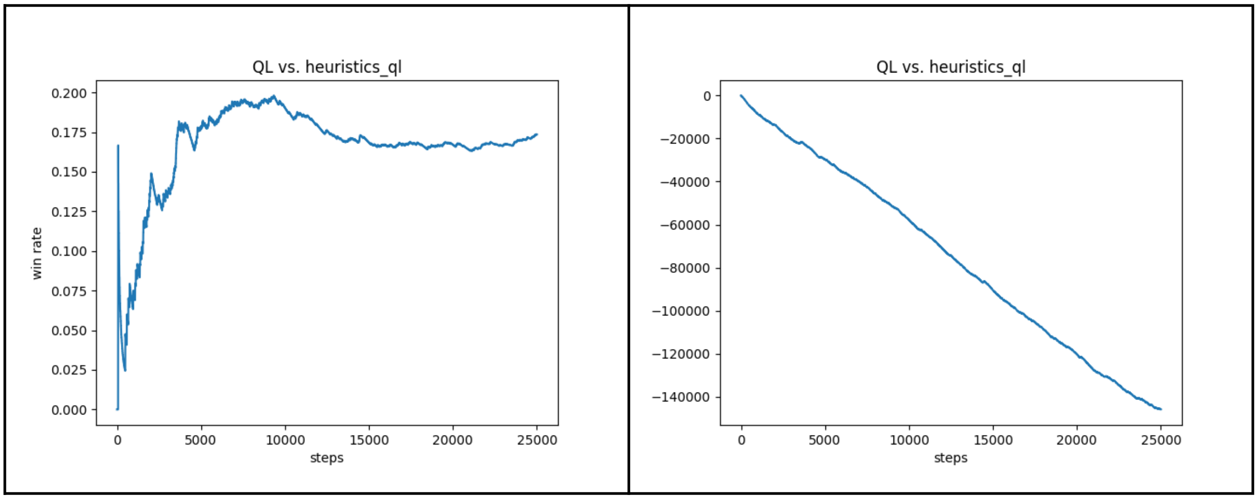
One the left is the win rate of the Q-learning RL model against the random agent. Note that one step is one action in the game. As seen here the model slows learning at around 125,000 actions with a final win rate of 80%. On the right is the cumulative reward of the RL model against the random agent. As seen it is mostly linear.

Results for training against the max-damage agent:



One the left is the win rate of the Q-learning RL model against the max-damage agent. As seen here the model slows learning at the end of training with a final win rate of 60%. On the right is the cumulative reward of the RL model against the max-damage agent. As seen it is higher than linear at first, but is linear after.

Results for training against the heuristic agent:

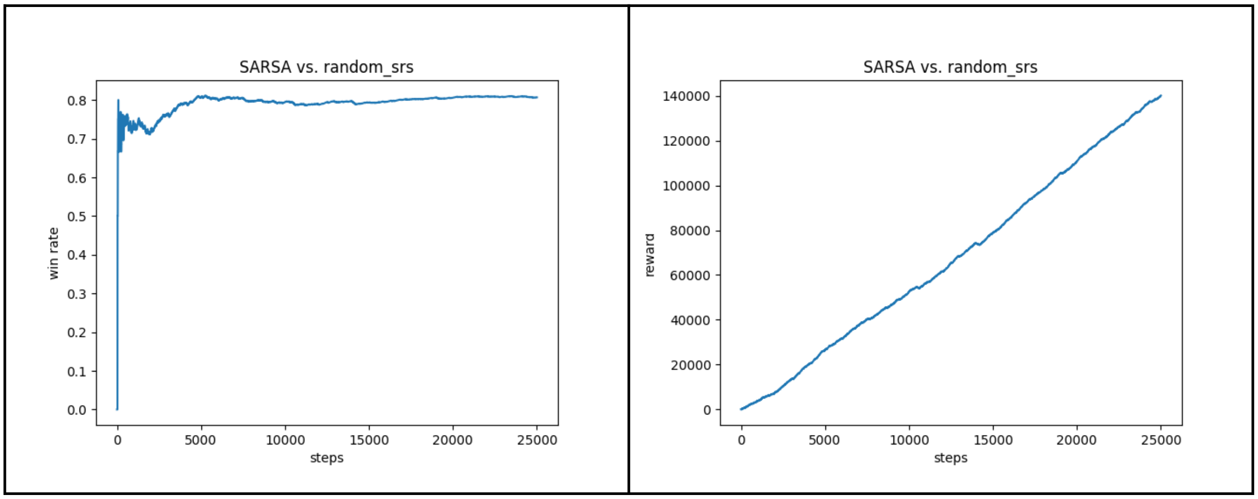


One the left is the win rate of the Q-learning RL model against the heuristic agent. As seen here the model slows learning at around 150,000 steps with a final win rate of 17.5%. On the right is the cumulative reward of the RL model against the heuristic agent. As seen it is negative linear as the model is almost always losing.

SARSA

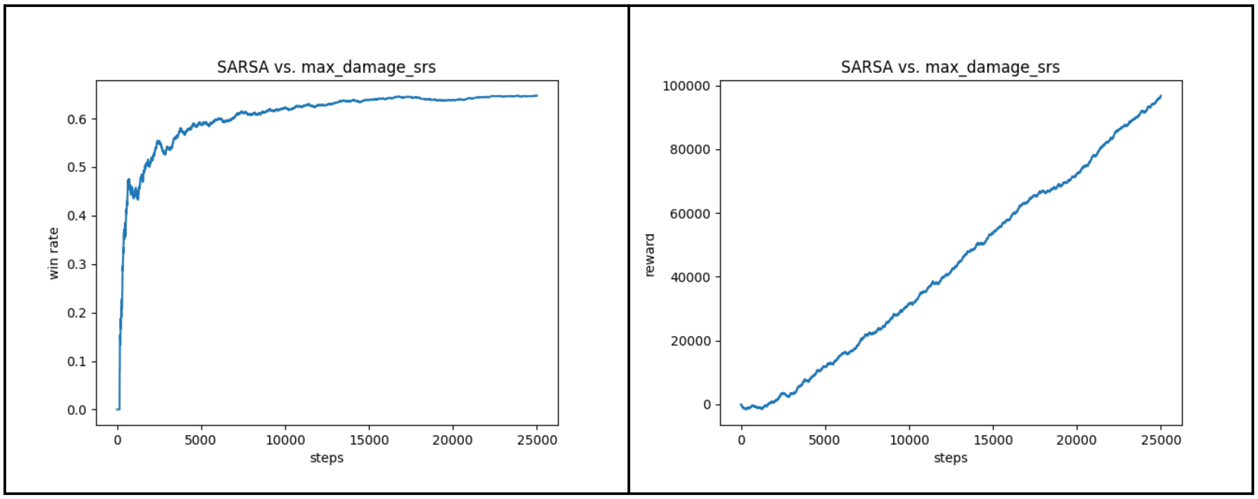
The best hyperparameters found while training were a constant epsilon of 0.1 and a constant learning rate of 0.1.

Results for training against the random agent:



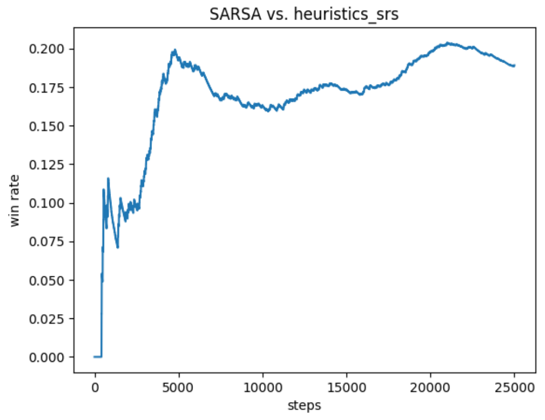
One the left is the win rate of the SARSA RL model against the random agent. Note that one step is one action in the game. As seen here the model slows learning at around 5,000 actions with a final win rate of 80%. On the right is the cumulative reward of the RL model against the random agent. As seen it is linear.

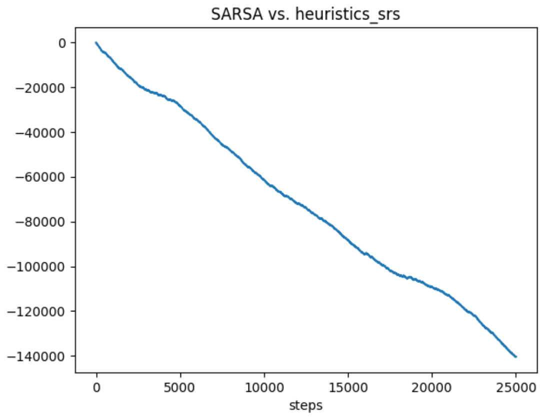
Results for training against the max-damage agent:



One the left is the win rate of the SARSA RL model against the max-damage agent. As seen here the model slows learning at around 150,000 steps with a final win rate of 68%. On the right is the cumulative reward of the RL model against the max-damage agent. As seen it is essentially linear.

Results for training against the heuristic agent:





One the left is the win rate of the SARSA RL model against the heuristic agent. As seen here the model has inconsistent learning with a final win rate of 18%. On the right is the cumulative reward of the RL model against the heuristic agent. As seen it is negative linear as the model is almost always losing.

b. Continuous State Representation

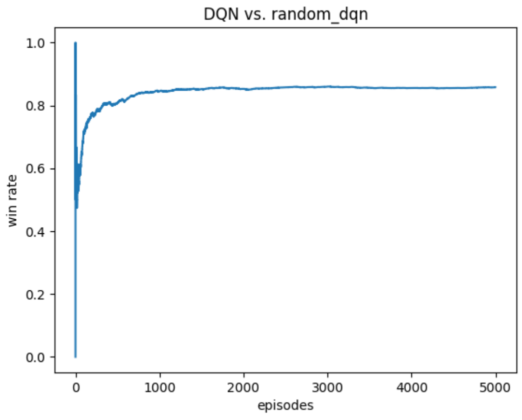
Deep Q Network

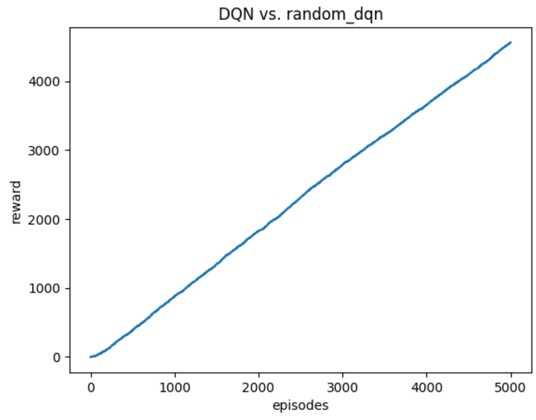
My DQN implementation includes many of the optimizations listed in the original paper, including Replay Memory, mini batching, and Double DQN. For neural network architecture please look at the code.

Hyperparameters chosen based on testing:

memory_size=1e5
minibatch_size=32
reset_Q_steps=400
gamma=0.5
epsilon_start=1
epsilon_min=0.01
epsolon_dec=0.9995
lr=1e-4

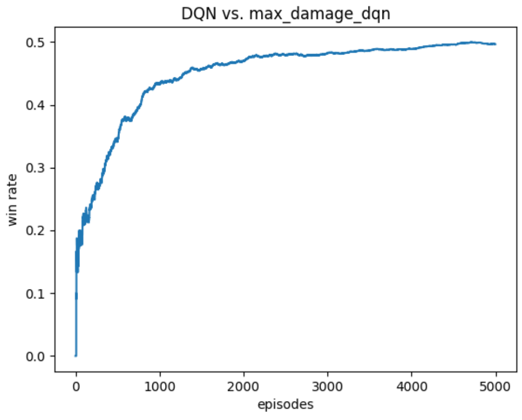
Results for training against the random agent:

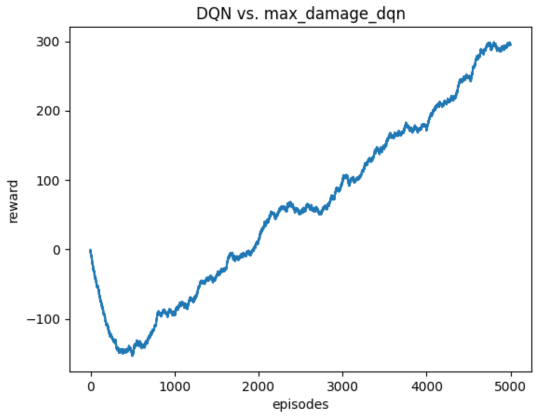




One the left is the win rate of the DQN RL model against the random agent. Note that one episode is one completed battle in the game. As seen here the model slows learning at around 1,000 episodes with a final win rate of 80%. On the right is the cumulative reward of the RL model against the random agent. As seen it is linear.

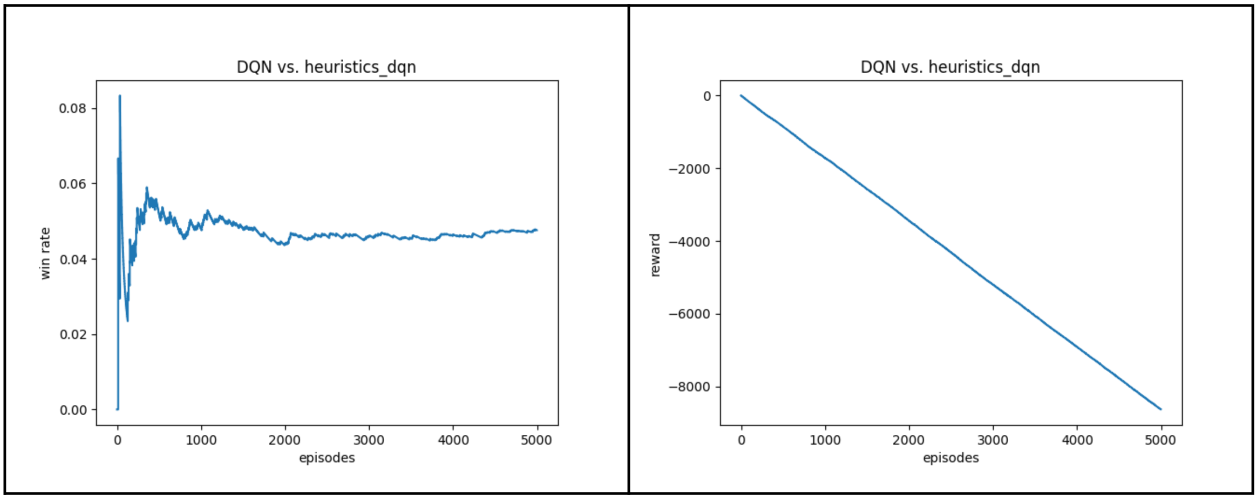
Results for training against the max-damage agent:





One the left is the win rate of the DQN RL model against the max-damage agent. As seen here the model slows learning at around 3,000 episodes with a final win rate of 50%. On the right is the cumulative reward of the RL model against the random agent. As seen it is decreasing at first, but is linear after the agent starts winning.

Results for training against the heuristic agent:



One the left is the win rate of the DQN RL model against the heuristic agent. As seen here the model slows learning at around 2,000 episodes with a final win rate of 5%. On the right is the cumulative reward of the RL model against the random agent. As seen it is decreasing linearly, as the agent always loses.

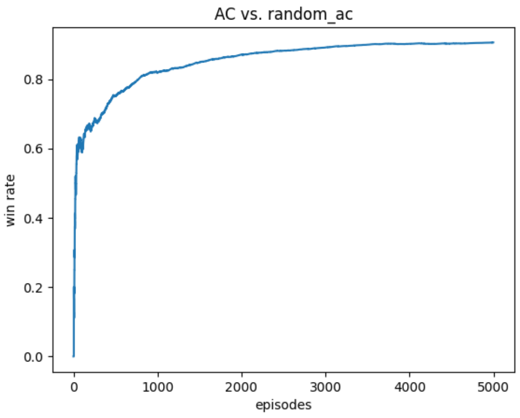
Actor-Critic

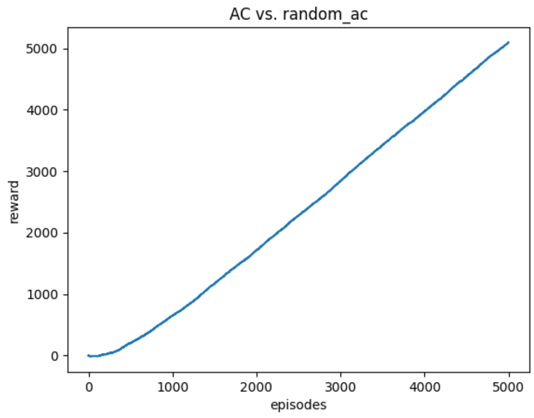
I implemented the One-Step Actor-Critic discussed in class using neural network approximators. For neural network architecture please look at the code.

Hyperparameters chosen based on testing:

gamma=0.5
actor_lr=1e-3
critic_lr=1e-3

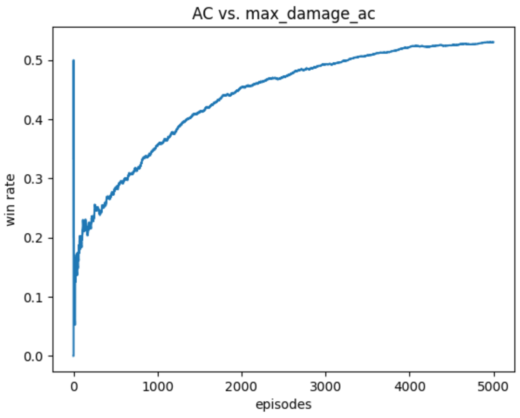
Results for training against the random agent:

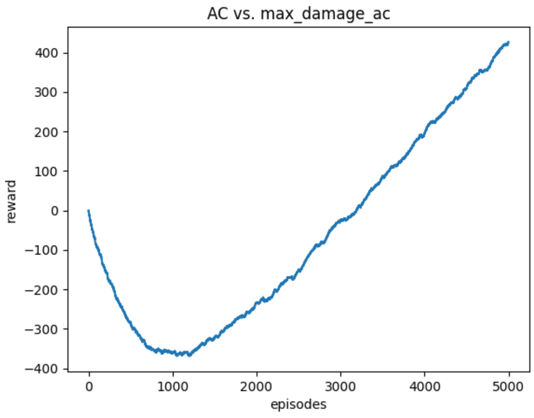




One the left is the win rate of the Actor-Critic RL model against the random agent. Note that one episode is one completed battle in the game. As seen here the model slows learning at around 4,000 episodes with a final win rate of 90%. On the right is the cumulative reward of the RL model against the random agent. As seen it is mostly linear.

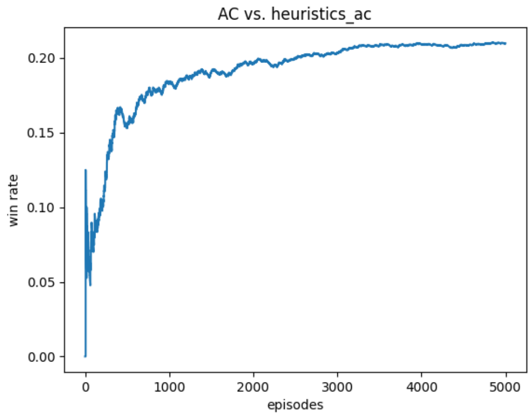
Results for training against the max-damage agent:

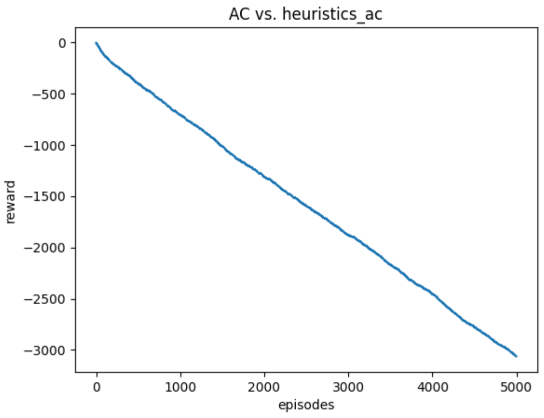




One the left is the win rate of the Actor-Critic RL model against the max-damage agent. As seen here the model slows learning near the end of training with a final win rate of 55%. On the right is the cumulative reward of the RL model against the random agent. As seen it is decreasing at first, but is linear after the agent starts winning.

Results for training against the heuristic agent:





One the left is the win rate of the Actor-Critic RL model against the heuristic agent. As seen here the model slows learning at around 3,000 episodes with a final win rate of 22%. On the right is the cumulative reward of the RL model against the random agent. As seen it is decreasing linearly, as the agent mostly loses.

c. Overall Evaluation

I evaluated the final model of each RL algorithm against each opponent. This was done over 100 battles for each pairing. Here is a summary of the results, including this final evaluation.

Discrete State

Convergence speed (actions):

	Random	Max-Damage	Heuristic
Q-Learning	125,000	Did not converge	150,000
SARSA	5,000	150,000	Did not converge

Evaluation win rate (100 battles):

	Random	Max-Damage	Heuristic
Q-Learning	87%	57%	16%
SARSA	87%	71%	13%

Continuous State

Convergence speed (episodes):

	Random	Max-Damage	Heuristic
DQN	1,000	3,000	2,000
Actor-Critic	4,000	Did not converge	3,000

Evaluation win rate (100 battles):

	Random	Max-Damage	Heuristic
DQN	88%	46%	10%
Actor-Critic	92%	60%	20%

5. Conclusion and Future Work

As seen in the results section, agents performed great against the random agent, alright against the max-damage agent, and horrible against the heuristic agent. This is to be expected, as this matches up with the difficulty of each opponent.

In terms of the discrete state space, it was interesting to see that SARSA was able to converge much faster than Q-Learning against the random agent. This is most likely due to the deeper simulation compared to Q-Learning. Also, SARSA was able to perform much better than Q-Learning against the max-damage agent. I am not sure what the cause of this behavior is.

In terms of the continuous state space, I am not surprised that Actor-Critic performed better than DQN. Having two neural networks learning parameters is a mechanical advantage for Actor-Critic over DQN’s single network approach. I do wish that I saw higher performance against the heuristic agent, but this may be due to lack of parameters or shorter training duration.

It was also interesting that the continuous state models learned the best when only accounting for health points in the reward function. This is strange, as in class we discussed it's usually better for the reward to only give when the agent actually wins. In this case that is not true, as the reward does not account for the win condition. In fact, it did not learn at all when I included the win condition. I hypothesize this is because the networks expect a smooth reward function.

The difficulty in training an AI to play Pokemon is the extreme stochasticity of the game. Moves can miss, damage dealt is a distribution, and status effects add even more randomness. It is a game where picking the right move can still lead to a loss, and picking the wrong move can still lead to a victory. This is probably why performance against the high level heuristics bot was so low.

If I had more time before submission, I would definitely spend more time training my agents. Implementing and debugging all the moving parts of this project took so long that not much time was left to actually train. I believe that this would have prevented the "Did not converge" results, and have possibly even led to higher win rates against the more difficult opponents.

In the future, I would like to experiment with larger models as well. This would allow me to try more advanced methods, such as self-play. I would also like to expand the state space and include all aspects of the battle state. Another future goal would be to implement a state space which can account for much more than three Pokemon.