Deep Reinforcement Learning for Pokemon Battling

Kevin Zhang
Purdue University
zhan4196@purdue.edu

1. Introduction

Games have long been seen as a test for skill and adaptation, for both humans and artificial algorithms. They vary widely in rule systems and complexity and can serve as test-beds for development of new algorithms. One such game, popular with a wide variety of demographics all over the world, is Pokémon – and more specifically, its system of battling.

Pokémon is among the largest video game franchises in the world, with main gameplay involving players acting as "trainers" leading a team of Pokémon in battle. In a 1v1 Pokémon battle, each player uses up to six Pokémon with each having a set of 6 statistics and one or two typings. Each turn, both players selects one of the four moves to attack the opponent's active Pokémon or may switch their active Pokémon for any of their other non-fainted ones. Once a Pokémon has lost all hitpoints (HP), it is forcibly switched out and remains unusuable for the rest of the battle. Pokémon Battling offers a very interesting combination of strategy, domain knowledge, and luck which has earned it a wide following for human users, and can offer a unique challenge for training artificial strategies.

With over 1000 [1] unique Pokémon and a similar number of moves, plus a variety of methods to change Pokémon statistics, Pokémon battles represent opportunities to learn in a very large, high-dimensional state space. Beyond this, Pokémon Battling features several different properties which make learning strategies difficult for machine learning algorithms. The Pokémon battling environment is multi-agent, where, as a players strategy evolves during the game, so does the opponents'. Additionally, unlike previous work in Chess and Go [10, 12], the environment is only partially observable, with key opponent characteristics such as attack and defense being unknown. Furthermore, the environment is stochastic, with the outcome of certain actions being random, which contrasts it with the environment of similar work for Starcraft II [17] and other such games.

This project is inspired by recent interest in developing machine learning systems for Pokémon battling, and was carried out with the aim of building experience in Reinforcement Learning (RL) techniques to accomplish such a goal. Different algorithms in RL are analyzed, then implemented in PyTorch. The agents are trained and tested in a Pokémon Battling environment which has been integrated into an OpenAI Gym environment.

2. Related Works

Over the past decade, Reinforcement Learning has fully entered the stage as a third major category of Machine Learning, contrasting with Supervised and Unsupervised methods. Within this framework, agents learn a policy to take actions between states depending on the surrounding environment [16]. To capture such information effectively, the value of a specific state or state-action pairing can be modelled as an intermediary to indirectly choose a best policy. In other methods, this policy can be directly learned in a model-free manner. This basic formulation works very well in modelling games, with well-defined states and fully understood transition dynamics.

2.1. Deep Reinforcement Learning

With the advent of Deep Learning and neural networks, their usage in RL has enabled a host of developments towards developing human-level artificial agents. These methods employ neural networks to approximate the value functions, and/or the policy itself.

- Policy gradient methods aim to directly learn the optimal function from state to action. The REINFORCE algorithm [19, 20] is taken as representative for deep policy-gradient methods, and uses a neural network to estimate a stochastic policy- represented by $\pi_\theta(a|s)$. This network is trained via gradient ascent by Monte-Carlo sampling of trajectories and their rewards.
- Bowling [2] introduces GIGA-WoLF, combining the principles of "Win or Lose Fast" and gradient ascent methods. Intuitively, it aims to learn at a faster rate when "losing" as compared to when it is winning. As a policy-gradient based method, GIGA-WoLF keeps track of two

policies, sampling trajectories from one but adjusting its learning based on both. The usage of two policy networks allows for faster updates when the action selection policy receives lower reward compared to the secondary policy. Bowling further proves two key properties: zero expected regret, and convergence in self-play. GIGA-WoLF is implemented to the author's best abilities, with comparison against vanilla REINFORCE policy gradient methods.
- Deep Q-Learning, in comparison with policy based methods, models only the State-Action Value function $Q_\theta(s, a)$ using a neural network. A policy is implicitly determined by maximizing reward over all possible actions at a state. This deterministic greedy policy however, hinders exploration of potential rewards in the environment; and thus an epsilon-greedy exploration strategy is employed. Mnih *et al.* [7] discuss empirical methods for improving convergence and results of Deep Q-Learning, including the use of a target Q-network and replay training for stability. These empirical improvements are included in the implementation of DQN used in this project.
- Actor-Critic methods combine the use of neural networks to estimate both value functions Q and policy π. The policy (actor) network is used to select actions while the value (critic) network "critiques" the actor. Both are trained via Monte-Carlo rollouts, matching policy gradient methods. The inclusion of the critic's predictions serves to stabilize training of the policy by subtraction of a baseline. Async Advantage Actor Critic (A3C) [8] and Generalized Advantage Estimation (GAE) [11] offer two variations of this architecture, and are referenced in implementation of the two methods.

2.2. Other Pokémon Systems

Pokémon battles have multiple properties which make them attractive for exploration [6], lending to the application of Deep RL techniques in studying Pokémon battling.

- Kalose *et al.* [5] offer information and results on their framework for a basic Q-Learning system using epsilon-greedy and softmax exploration techniques. They offer a good starting place for state information of Pokémon and move definition as well as game-state. The agents trained with the methods are compared against random agents and their results show that their agents achieve superiority by win-rate. Such an implementation offers a good starting point for methods and comparison against a similar student project.
- Simoes *et al.* [13] offer a short introduction to distributed versions of GIGA and Weighted Policy Learner (WPL) method. They then implement and test the two training methods for Pokémon battling. Their results indicate agents trained via these methods to be generally effective, and offer analysis of strategies which are learned. However, the authors train agents with randomized Pokémon attributes in nearly deterministic setting (No status, no accuracy $< 100\%$); so an extension of their methods to "real" Pokémon battling scenarios will allow for comparison with other agents.
- Finally, Hu *et al.* present PokeLLMon [4] and detail their adaptation of LLM systems to general strategy games. Most information is focused on the difficulties of this adaptation, including difficulties with memory and hallucinations, which includes analysis of situations where the agent fails. These failures offer insight into potential areas for development in reward or embedding structure. The code is open source on GitHub, allowing for human users to play against what the authors claim is human parity skill. This approach is the only one of the three examined to compete in a realistic Pokémon battle without deterministic constraints.

2.3. Related Methods

The ultimate goal of the project was to develop novel methods examining few shot learning. However, due to time constraints, this was never realized.

- The authors of [18] give an introduction and several examples of few-shot learning techniques in basic OpenAI gym environments and will offer a good starting place for further reading into the subject for future work.
- Sohn *et al.* [15] develop a meta-learning technique for subtask graphs introduces several possible areas for future review, and can offer inspiration for development of other novel techniques.

3. Approach

Poke-env [9] offers an open source battling environment that has been pre-implemented. It offers support for connecting to a Pokemon Showdown backend [14], allowing for local training and potentially access to other players on online servers. It is modified for integration with implementations of RL methods using PyTorch. Based on the introducing papers, the four different methods- REINFORCE, DQN, WoLF-GIGA, and A2C- are reproduced in PyTorch and tested in the environment. Algorithm 1 demonstrates the shared training loop algorithm.

The following subsections describe the formulation of the environment for Reinforcement Learning agents, as well as important derivation and implementation details for the four selected Deep RL methods which are tested in the environment. The code can be found in a GitHub repository at `https://github.com/InvariantProgram/PurdueCS587Proj`

3.1. Environment

Subclassing poke-env's implementation of an OpenAI Gym [3] environment allows for implementations of custom

Algorithm 1 Training Loop

```
Input: Initialized Agent $A$ with (potentially implicit) policy $\pi_A$
Input: Environment $M$
Output: Trained agent $A$
$s \leftarrow$ first state from $M$
for step < n_steps do
    Select action $a \sim \pi_A(s)$
    Execute $a$ in $M$
    Observe reward $r$, new state $s'$, and terminal $d$
    if $A$ is Monte-Carlo (Policy Gradient) then
        $A$ stores $(s, a, r, \log P(a), d)$ in memory
    else
        $A$ stores $(s, a, r, s', d)$ in memory
    end if
    $s \leftarrow s'$
    Train $A$ from memory
    if d then                                  ▷ Terminal State
        $s \leftarrow$ first state from $M$
    end if
end for
```

state embedding and reward functions by modification of the proper functions.

For this experiment a simple embedding and reward structure is used. Given a state in a Pokémon battle, the state is embedded into a vector of length 10. The first four indices of this vector are the base powers of each of the four possible moves for the agent's active Pokémon scaled by a factor of 1/100. If a move does not have a base power- such as a status inflicting move, or Protect, its power is encoded as -1. This leaves the indices in the range [-1, 3], as the highest possible base power for a move is 300. The next four indices encode the effectiveness of the corresponding move's type against the opposing team's active Pokémon. These indices take values $\{0, 0.25, 0.5, 1, 2, 4\}$. The last two indices store the number of remaining non-fainted Pokémon each player has, scaled by a factor of 1/6, such that a full team is encoded to a value of 1 and a player that has just lost has a team value of 0.

The reward calculation is similarly simple. For each Pokémon on the agent's team, an hp value modification is multiplied onto its health and added to the reward. If the Pokémon has fainted, a fainted value modification is subtracted from the reward. If it has any status conditions, a status value modification is subtracted from the reward. A mirrored operation is then carried out using the opponent's team, subtracting their hitpoint scores and adding their fainted and reward scores. Finally, if the agent has won the game it receives a victory reward. If it has lost, it instead loses this victory reward. The parameters used are detailed in Table 1.

Parameter	Value
hp	1.0
fainted	2.0
status	0.0
victory	30.0

Table 1. Reward parameters

From these values it can be seen that the major driving force is maintaining a total hitpoint advantage over the opponent- if you maintain more hitpoints than the opponent at every point in the game, it is impossible to lose. Victory then serves more of a role in adding a signal to potentially exploratory moves with more delayed effects.

Poke-env also includes the implementation of three different baseline agents. They can be labeled as RandomPlayer, MaxBasePowerPlayer, and SimpleHeuristicPlayer with the following behaviors:

- **RandomPlayer**: Randomly selects out of up to 9 valid actions: the four moves of its active Pokémon, and any valid switch to a non-fainted ally Pokémon.
- **MaxBasePowerPlayer**: Always chooses the move with highest base power if possible. If unable to do so, selects a random ally switch.
- **SimpleHeuristicPlayer**: Implements a heuristic-based human-like strategy. It will choose to set up entry hazards (which cause damage or bad effects to opponents passively as they switch in) if possible. It will also choose to boost its own active Pokémon if the opponent's Pokémon has a poor match-up into it. Additionally, it will select moves based on an estimate taking into account both move power, effectiveness, and the physical/special category of the move. These rules combine to create a formidable opponent, even for novice human players.

3.2. REINFORCE

As described in the related works section, vanilla policy-gradient REINFORCE [19, 20] uses a neural network with parameters θ to estimate a stochastic policy:

$$\pi_\theta(s) = P(a \in A|s;\theta) \tag{1}$$

Then the reward $J(\theta)$ of a policy parameterized by θ can be expressed in form of trajectories τ as:

$$J(\theta) = \sum_{\tau} P(\tau;\theta)R(\tau) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{2}$$

Where τ describes the sequence of states, with an associated probability of happening based on the policy π's decisions.

It's total reward, the summation of all rewards received during this trajectory is labeled $R(\tau)$. Then calculating its gradient produces:

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \sum_\tau \pi(\tau;\theta)R(\tau)\\ &= \sum_\tau R(\tau)\nabla_\theta \pi(\tau;\theta)\\ &= \sum_\tau R(\tau)\pi(\tau;\theta)\frac{\nabla_\theta \pi(\tau;\theta)}{\pi(\tau;\theta)}\\ &= \sum_\tau \left(R(\tau)\nabla_\theta \log \pi(\tau;\theta)\right)\pi(\tau;\theta)\\ &= \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)\nabla_\theta \log \pi_\theta(\tau)]\end{aligned} \tag{3}$$

Further, expanding $\pi_\theta(\tau)$-

$$\begin{aligned}\pi_\theta(\tau) &= P(s_0)\prod_{t=1}^{T}\pi_\theta(a_t|s_t)P(s_{t+1}|s_t,a_t)\\ \nabla_\theta \log \pi_\theta(\tau) &= \sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\end{aligned} \tag{4}$$

Allows for the full Policy Gradient Theorem,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T} G_t \nabla_\theta \pi_\theta(a_t|s_t)\right] \tag{5}$$

Where G_t defines the reward of a partial trajectory i.e. the sum of all (potentially discounted) rewards following a specific state within a trajectory. This formulation is readily approximated via Monte-Carlo sampling, and leads to Algorithm 2 describing the implementation of memory training REINFORCE in code.

Algorithm 2 REINFORCE memory training

```
**Input:** Memory $M$ of $(s, a, r, \log P(a), d)$ tuples
**Input:** Agent's policy $\pi_\theta(s)$ and discount rate $\gamma$
**Output:** One step optimization of agent $A$
**if** final tuple in $M$ has $d =$ True **then**
    $L \leftarrow 0$
    **for** t in length $M$ **do**
        **if** t is final timestep **then**
            $G_t \leftarrow r_t$
        **else**
            $G_t \leftarrow r_t + \gamma G_{t+1}$
        **end if**
        $L \leftarrow L + G_t \cdot \log P(a_t)$
    **end for**
    Use PyTorch autograd with loss $-L$ to update $\theta$
    Flush memory $M$                    ▷ autograd minimizes
**end if**
```

3.3. GIGA-WoLF

GIGA-WoLF [2] acts as a modification to vanilla policy-gradient (GIGA) methods. As described previously, the algorithm keeps track of two policy estimators x and z. Actions are taken by the stochastic policy x, while z serves to improve update steps. The update rules are implemented according to the Bowling's three equations

1. A surrogate policy x' is generated by projection [21] of an unconstrained modification of x's policy at state s_t back to a distribution. A gradient update step is then taken for x towards x'.
2. z' is similarly generated from z's policy at s_t. z then also takes a gradient step towards z' but at 1/3rd the step size. Then, a final step size is calculated, taking a value of 1 unless upperbounded by a small step-size taken by z.
3. x is then updated towards the updated z based on the step-size calculated in step 2.

The requirement that a reward be known for a state, and formulation as a modification to policy-gradient methods led to this author's update step occurring at the end of every episode, as in REINFORCE. The implementation of the modified update step is described in psueocode in Algorithm 3.

Algorithm 3 GIGA-WoLF memory training

```
**Input:** Memory $M$ of $(s, a, r, \log P(a), d)$ tuples
**Input:** Agent's action policy $\pi_x(s)$, auxiliary policy $\pi_z(s)$, and discount rate $\gamma$
**Output:** One step optimization of agent $A$
**if** final tuple in $M$ has $d =$ True **then**
    **for** t in length $M$ **do**
        $G_t$ as calculated in REINFORCE training
        $a \leftarrow onehot(a_t) \cdot \log P(a_t) \cdot G_t$
        $x, z \leftarrow \pi_x(s_t), \pi_z(s_t)$
        $x' \leftarrow \_p(x + a)$          ▷ $\_p$ projection in code [13]
        $L \leftarrow ||x' - x||_1$
        Use PyTorch autograd with loss $L$ to update $\pi_x$
        $z' \leftarrow \_p(z + a/3)$
        $L \leftarrow ||z' - z||_1$
        Use PyTorch autograd with loss $L$ to update $\pi_z$
        $z'', x'' \leftarrow \pi_z(s_t), \pi_x(s_t)$
        $\delta \leftarrow \min\left(1, \frac{||z''-z||_1}{||z''-x''||_1}\right)$
        $L \leftarrow \delta \cdot ||z'' - x''||_1$
        Use PyTorch autograd with loss $L$ to update $\pi_x$
    **end for**
    Flush memory $M$
**end if**
```

3.4. Advantage Actor Critic

One issue which policy-gradient methods face is high variance in training. Monte-Carlo estimates for the expectation are very noisy and so can cause much headache in convergence for policy gradient based methods. One solution is the usage of a baseline [19]. From a Monte-Carlo estimation with baseline:

$$\nabla_\theta J(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log \pi_\theta(\tau^{(i)}) \left(R\left(\tau^{(i)}\right) - b \right) \quad (6)$$

With an additional term which is the Monte-Carlo approximation of:

$$\begin{aligned} \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log \pi(\tau;\theta) b] &= \sum_\tau \pi(\tau;\theta) \nabla_\theta \log \pi(\tau;\theta) b \\ &= b \nabla_\theta \left(\sum_\tau P(\tau) \right) \\ &= b \cdot 0 \end{aligned} \quad (7)$$

Factoring b out is valid so long as it does not depend on action in log probability, keeping the gradient estimate unbiased.

One such baseline which does not depend on action is the value of the state $V_\pi(s)$. Thus, Advantage Actor Critic uses an estimate of the Q function to estimate this via the Bellman Equation:

$$Q_\pi(s,a) = r + \gamma V_\pi(s') \quad (8)$$

Thus, this Q estimator serves as a critic to the policy networks actor. The policy seeks to gain ground over the value network's estimation, by maximizing the *advantage*: the actual realized reward of the trajectory minus the Q estimator's reward estimate. This leads to the implementation for this project in Algorithm 4.

Algorithm 4 A2C memory training

```
Input: Memory $M$ of $(s, a, r, \log P(a), d)$ tuples
Input: Agent's policy $\pi_\theta(s)$, value estimator $Q_\omega(s)$, and discount rate $\gamma$
Output: One step optimization of agent $A$
if final tuple in $M$ has $d$ = True then
    $L_p, L_v \leftarrow 0, 0$
    for For $t$ in length $M$ do
        $G_t$ as calculated in REINFORCE training
        $v_t \leftarrow Q_\omega(s_t, a_t)$                ▷ $Q_\omega(s_t)$ indexed at $a_t$
        advantage $\leftarrow G_t - v_t$                ▷ detach $v_t$ here
        $L_p \leftarrow L_p$ + advantage
        $L_v \leftarrow L_v + |G_t - v_t|$
    end for
    Use PyTorch autograd with loss $-L_p$ to update $\theta$
    Use PyTorch autograd with loss $-L_v$ to update $\omega$
    Flush memory $M$
end if
```

3.5. Deep Q Network

Q-Learning [16] seeks to estimate the optimal value of each state action pair, which can then be exploited by selecting the action with optimal action-value at each state. Directly using our estimator $Q_\theta(s,a)$ leads to an update step using the Bellman equations:

$$Q_{target}(s,a) = R(s,a,s') + \gamma \max_{a'} Q_\theta(s',a') \quad (9)$$

Gradient update methods may then be used on Q_θ with this loss formulation to optimize θ in an online manner. Previous policy based methods produced stochastic policies allowing for exploration even as a policy stabilized. In Q-Learning methods, an ϵ-greedy policy is used during training, taking a random action with probability ϵ to explore the action space, while focusing more on actions that are deemed "good."

Algorithm 5 DQN memory training

```
Input: Memory $M$ of $(s, a, r, s', d)$ tuples
Input: Agent's Action-Value estimator $Q_\theta(s)$, target estimator $Q_{\theta'}(s)$, discount rate $\gamma$, exploration rate $\epsilon$, target net update rate target_update
Output: One step optimization of agent $A$
if $M$ has more than batch_size samples then
    $L \leftarrow 0$
    Sample a batch of transitions from $M$
    for $(s, a, s', r, d)$ in Sample do
        $q \leftarrow Q_\theta(s, a)$                ▷ $Q_\theta(s)$ indexed at $a$
        $q' \leftarrow \max_{a'} Q_{\theta'}(s', a')$                ▷ Detach $q'$
        if $d$ = True then
            target $\leftarrow r$
        else
            target $\leftarrow r + \gamma q'$
        end if
        $L \leftarrow L + |\text{target} - q|$
    end for
    Use PyTorch autograd with loss $L$ to update $Q_\theta$
    if target_update steps occurred since last update then
        $\theta' \leftarrow \theta$                ▷ Copy to target
    end if
    Anneal $\epsilon$ towards Agent's minimum $\epsilon$
end if
```

However, Mnih *et al*. [7] note two key considerations. First, the estimator Q_θ is updating towards a moving target-in essence "chasing its own tail." They propose addition of a target estimator $Q_{\theta'}$ to better stabilize training. This better prevents changes in Q_θ from finding vastly different maximization policy. In practice, they find that using a frozen version of Q_θ functions well with little overhead. The second consideration is that rewards are very highly correlated in a single trajectory. Producing a single brilliance in a string of mistakes leading to a loss may cause the algorithm to in fact disprefer such a good move. They propose the usage of a replay buffer in training to break this correlation and show good results. A fixed-length circularly replaced memory is used in this project, and the pseudocode implementation of the method is described in Algorithm 5.

4. Results

For simplicity due to time constraints, simple training experiments were implemented and ran. Environment details for all experiments can be found in Sec. 3.1. The environment embedding and reward structure is held constant for each training method. Only the training opponent is varied for different experiment types. Training results in the environment will include randomization, as the training environment is Generation 5 Random Battles. So, in addition to the stochastic nature of transitions due to chance in moves themselves, each battle will be generated with each agent having different Pokémon. All experiments were trained over 10,000 environment steps, with each training run encompassing approximately 350 games.

Additionally, each method was ran with off-the-shelf hyperparameters. All methods share the same γ value of 0.99. All networks are created with approximately the same architecture: A fully connected network using ReLU activation and a single hidden layer of size 128, though output sizes differ, producing slightly different architectures for value Q networks and policy π networks. Additionally, any optimizer is a default Torch SGD optimizer. Policy gradient methods were evaluated as stochastic policies, sampling actions from a categorical distribution based on output, and did not use other hyperparameters. DQN methods used a circularly replaced replay buffer of size 1000 and batch size 32 and ran target updates every 100 gradient steps. The exploration rate ϵ was annealed at a rate of 1/2000 from initial value 0.9 to minimum value 0.1, with updates occurring on gradient steps.

There were a total of 10 different experiments ran. The first eight are discussed first: Agents trained with each of the four methods (REINFORCE, GIGA-WoLF, DQN, A2C) trained against a RandomPlayer baseline agent, as well as agents trained via each of the four methods against a MaxPowerPlayer baseline agent. Each experiment type was ran 5 times, where after training for 10,000 environment steps, the trained agent was evaluated in the Generation 5 Random Battle format over 50 games against the three baseline agents introduced in Sec. 3.1. The win-rates against these baseline agents are used to discuss the results of the four reinforcement learning strategies and are shown in Sec. 4.

The results clearly show DQN's superiority in training method under these conditions. It is the only method to consistently beat the RandomPlayer in both environment scenarios. It also shows significantly stronger performance than all three other methods in all but 1 format- evaluation against the Heuristic agent when trained against a random player. The DQN agents also outperform Kalose *et al*. [5], reaching a win-rate against random agents of approximately 90% over around 350 games, compared to 70% over 20,000 games.

We examine one such evaluation game between the best performing experiment type: an agent trained with DQN against a MaxPowerPlayer. The agents choices shed insight the difficulties of training for Pokémon battling, and into the results of training.

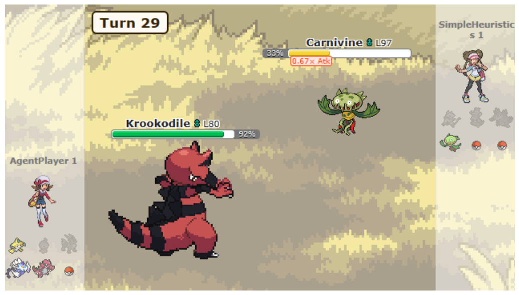


Figure 1. Turn 29: Krookodile (ground, dark type) has just been sent in against Carnivine (grass type)

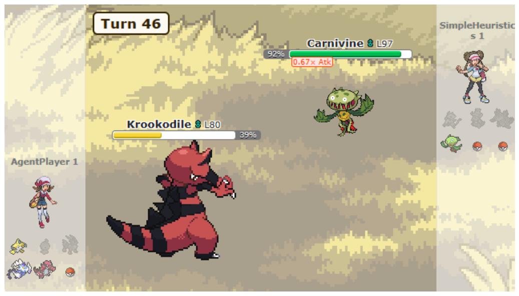


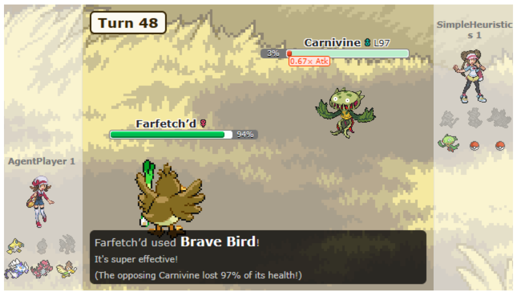
Figure 2. Turn 46: Krookodile and Carnivine remain on the field for 17 turns, with minimal action.

Figures 1 and 2 demonstrate the limitations of a low state space embedding. Over 16 turns on the field, Krookodile repeatedly selected what it believed to be its strongest move against Carnivine, Earthquake. However, Carnivine's ability- Levitate- made it immune, meaning it took no damage (and instead healed over time with its leftovers item). Meanwhile, the SimpleHeuristics agent recognized Carnivine's typing of grass was advantageous into Krookodile's typing of ground, and chose to keep Carnivine in. This cycle was only broken on the next turn when Earthquake ran out of PP (A limitation which is rarely felt by human players), and a random switch was made by the trained agent. This also was the reason Carnivine was doing minimal damage- as its grass type move had already been drained of PP beforehand.

This example, and many others like it, demonstrate that the DQN agent had very successfully learned to repeatedly select the move at its disposal with highest base power, acting almost exactly as the MaxPowerPlayer. This then suggests that the policy-gradient methods: REINFORCE, GIGA-WoLF, and A2C, were unable to capture this simple principle. Simoes *et al.* [13] report success with GIGA. However, they train using over 1.3 million environment steps, and modify the algorithm to use Actor-Critic methods. They also experiment in a simplified environment, using only 5 basic types (Fire, Water, Grass, Normal, Fighting) and a deterministic set of moves.

Thus, it is clear that hyperparameter tuning, and increased training time are necessary to achieve good results with policy-gradient methods, while DQN seemed to be effective almost out-of-the-box.

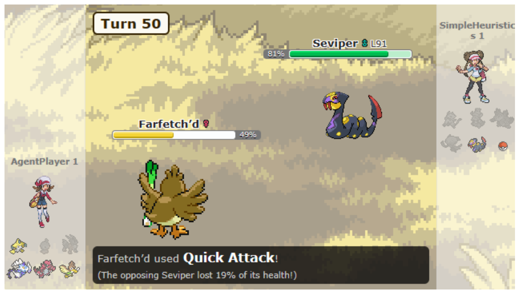
However, beyond this, DQN methods seem to have a win-rate slightly lower against MaxBasePower- while you would expect a win-rate of 50% if their strengths were exactly the same. This is likely due to some artifacts of strategy learning- which can be seen in Figures 3 and 4.



Farfetch'd used **Brave Bird**!
It's super effective!
(The opposing Carnivine lost 97% of its health!)

Figure 3. Turn 48: FarFetch'd very effectively selects Brave Bird (flying) into Carnivine (grass).

The trained agent's FarFetch'd initially selects Brave Bird against the opponent's Carnivine, hitting for very big damage. However, once the opponent switched in Seviper, the agent instead chose to repeatedly use Quick Attack.



Farfetch'd used **Quick Attack**!
(The opposing Seviper lost 19% of its health!)

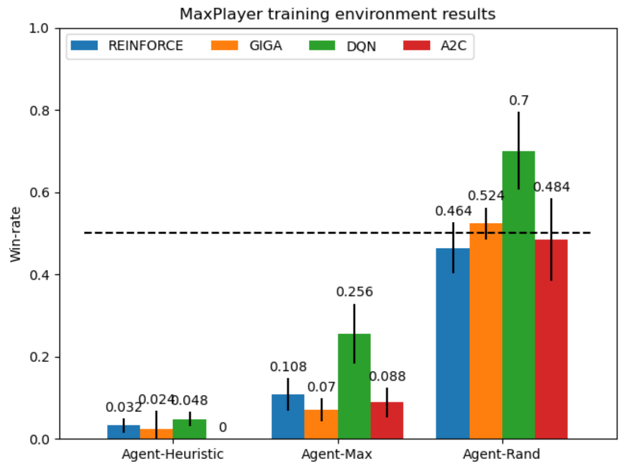
Figure 4. Turns 49 and 50: FarFetch'd opts to use Quick Attack (normal) into Seviper (poison). A quick check shows that Seviper may be faster than FarFetch'd.

An experienced human player may also make such a choice: Carnivine has lower speed than FarFetch'd, so we would easily be able to select our strongest move. However, Seviper may potentially be faster than FarFetch'd; and in this situation, it may be beneficial to simply guarantee a hit via Quick Attack, which (almost) always hits first. We can see here that the DQN agent may in fact be learning some rudimentary strategy, causing it to falter in some situations against the MaxBasePowerPlayer, but occasionally make interesting plays in other situations.

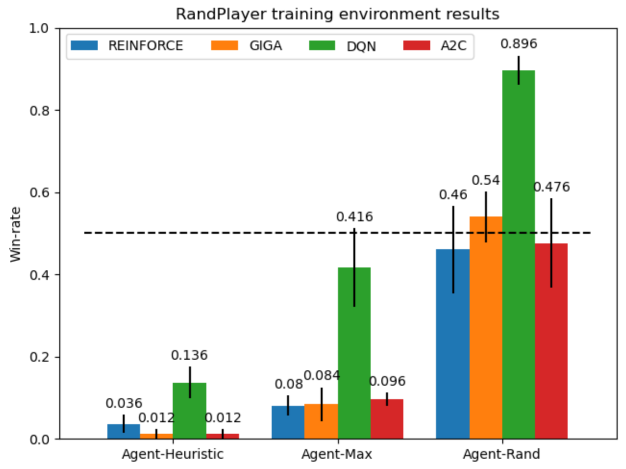
Finally, the final two experiments involve training the DQN agent against a SimpleHeuristicsPlayer, and swapping out the optimizer to a default Torch Adam optimizer, while still training against the MaxBasePower agent. The results are shown below in Fig. 6. They show that generally, the agent trained with SGD performs better as compared to that trained with Adam: this likely relates to the moving target issue of DQN training. The usage of a target network only alleviates this problem, instead of completely solving it. This moving target then makes Adam's momentum potentially problematic. We can additionally see that the agents trained against the MaxPower player performed significantly better than those trained against the SimpleHeuristics agent. This demonstrates further the importance of the opponent in training, and the author hypothesizes that the very strong nature of the opponent reduced reward signals- all towards loss; and hindered learning.

5. Conclusion

Through this project, four different Reinforcement Learning training algorithms: REINFORCE, GIGA-WoLF, DQN, and A2C have been implemented and tested on a realistic Pokémon battling environment. While a very simple embedding and reward structure was used, agents trained via DQN were able to obtain strong superiority over Random Agents, achieving a win-rate of approximately



(a) Results when trained against a MaxPowerPlayer baseline.



(b) Results when trained against a RandomPlayer baseline.

Figure 5. Final win-rate results for a sample size of n=5 runs. Error bars show sample standard deviation. The x-axis corresponds with the opponent in testing, the two graphs shows the results of models trained against different baseline models. The dashed line shows a win-rate of 0.5 for approximate parity in strength.

	Win-rate		
Experiment	Agent-Rand	Agent-Max	Agent-Heuristic
MaxPower DQN	0.896 ± 0.036	0.416 ± 0.095	0.136 ± 0.0385
Heuristic DQN	0.66 ± 0.105	0.276 ± 0.056	0.072 ± 0.023
Adam DQN	0.712 ± 0.107	0.344 ± 0.0385	0.152 ± 0.048

Figure 6. Final win-rate results for a sample size of n=5 runs. Adam DQN is trained with an Adam optimizer against the MaxPower baseline agent. Errors show sample standard deviation.

90% over 10,000 environment steps corresponding to roughly 350 games. The results similarly reflect the DQN training method to be most effective in the setting.

An analysis of the difference in performance is offered, however, the work is hindered by limited scope in hyperparameter exploration, and in low training time; which are both necessary parts of successful Reinforcement Learning, especially in policy gradient methods.

In future work, the author hopes to further explore RL algorithms in this space and develop more complex embedding and reward schemes for testing. The environment and motivation may also serve to produce a new testbed in developing multi-agent learning methods in Pokémon battling.

References

[1] List of pokemon. *Web:* `https://pokemondb.net/pokedex/national`, 2024. 1

[2] Michael Bowling. Convergence and no-regret in multiagent learning. In *Proceedings of the 17th International Conference on Neural Information Processing Systems*, page 209–216, Cambridge, MA, USA, 2004. MIT Press. 1, 4

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 2

[4] Sihao Hu, Tiansheng Huang, and Ling Liu. Pokellmon: A human-parity agent for pokémon battles with large language models. 2

[5] Akshay Kalose, Kris Kaya, and Alvin Kim. Optimal battle strategy in pokemon using reinforcement learning. *Web: https://web. stanford. edu/class/aa228/reports/2018/final151. pdf*, 2018. 2, 6

[6] Scott Lee and Julian Togelius. Showdown ai competition. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 191–198. IEEE, 2017. 2

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518, 2015. 2, 5

[8] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd Interna-*

tional Conference on International Conference on Machine Learning - Volume 48, page 1928–1937. JMLR.org, 2016. 2

[9] Haris Sahovic. poke-env. https://github.com/hsahovic/poke-env, 2024. 2

[10] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588, 2020. 1

[11] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2015. 2

[12] David Silver, Julian Schrittwieser, Karen Simonyan, ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas baker, Matthew Lai, Adrian bolton, Yutian chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature Publishing Group*, 550, 2017. 1

[13] David Simões, Simão Reis, Nuno Lau, and Luís Paulo Reis. Competitive deep reinforcement learning over a pokémon battling simulator. In *2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 40–45, 2020. 2, 4, 7

[14] Smogon. https://github.com/smogon/pokemon-showdown, 2023. 2

[15] Sungryull Sohn, Hyunjae Woo, Jongwook Choi, and Honglak Lee. Meta reinforcement learning with autonomous inference of subtask dependencies. 2020. 2

[16] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. *IEEE Trans. Neural Networks*, 9:1054–1054, 1998. 1, 5

[17] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P Agapiou, Max Jaderberg, Alexander S Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575, 2019. 1

[18] Zhechao Wang, Qiming Fu, Jianping Chen, Yunzhe Wang, You Lu, and Hongjie Wu. Reinforcement learning in few-shot scenarios: A survey. *J. Grid Comput.*, 21(2), 2023. 2

[19] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, 1992. 1, 3, 5

[20] Junzi Zhang, Jongho Kim, Brendan O'donoghue, and Stephen Boyd. Sample efficient reinforcement learning with reinforce. 1, 3

[21] Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. page 928–935. AAAI Press, 2003. 4