CS221 Project Progress
Catch'em all: A Pokémon Go Playing AI

SUNet ID: yulelee
Name: Yue Li

Background

Pokémons Go is a popular game for which the players need to go out and explore. In this project, I'll focus on one aspect of this game: catching Pokémons. The basic rule is: Pokémons are spawned at different places in the real world, if you are near enough to them, they will show up so that you can catch them. As a result, it would be fun to simulate the game and create an AI agent that tries to catch as many Pokémons as possible.

Simulation

The simulation of this game takes 3 components:

(a) Board:
The game is simulated by a $n \times n$ board. Each position is represented by a tuple of coordinates. The board is responsible for spawning Pokémons at various positions. The agent is also at a certain position on the board. The board should implement the `move_agent` method, which takes an action (including "Left", "Right", "Up" or "Down") as argument, and return the new position, the radar information (a list of nearby Pokémon IDs), and the rewards (if there are Pokémons at the new position, the agent could catch them all and get the scores).

(b) Agent:
The agent should implement two methods: (1) `get_action` evaluate the current situation (which includes the current position of agent and the radar list of nearby Pokémons), and make a decision of where to go next. (2) `incorperate_feedback`: for each action and its outcome, the agent has a chance of learning useful things from them. The arguments for this methods includse the old and new position of the agent, the radar information, and the Pokémons being caught (if there is any).

(c) Simulator: A simulator is like a middle man between the board and agent, calling the above methods and also recording all of the useful statistics.

Overall, the simulation is proceeded by iterations, the board might spawn a few new Pokémons according to the statistics from real life data. Each of the newly spawned Pokémon comes with an initial disappearing clock (an integer), which would be decremented for each iteration. As a result, the board also need to check the clock of all the existing Pokémons, if anyone's clock hits 0, the corresponding Pokémon would be deleted. Also, for each iteration, the agent has the chance to move one step on the board, towards the direction it thinks is the best.

Input and Output

For this game, no special input is needed, the game could just start (agent being put to the center of the board). We also don't have an end state, the game will be forced to end after a certain amount of iterations. Two metrics (output) are used to measure the performance of the agent:

(a) Percentage of scores being caught: Given the real life data, each type of Pokémon has been assigned with a score, which is calculate by this formula, for a given type of Pokémon p:

$$\text{score}_p = \frac{K + \sum_i C_i}{C_p + 1}$$

where K is the total types of Pokémons (currently 151), and C_p represent the count of appearance for a Pokémon type p from the dataset. Because the dataset is limited, all types of Pokémons get 1 "free" count, so in my simulation even Mewtwo (a Pokémon that is not currently included in the game) is technically possible to appear. Also, the scores are not revealed to the agent, the agent has to learn from the game which Pokémons are better.

(b) Percentage of Pokémons being caught.

For those two metrics, percentage of Pokémons is used to measure the agent's ability of finding Pokémons, while the percentage of scores is used to measure whether the agent is able to actively prioritize more valuable Pokémons to catch.

MDP

The game is formulated to be a MDP problem with the following components:

(a) States: The states are defined to be a combination of the current agent position, and the list of nearby Pokémons (a list of Pokémons IDs). The agent is initialized to the center of the board, so the start state is the center position plus an empty list of Pokémons. There is no end state.

(b) Actions: The actions includes the four possible directions the agent can move, "Left", "Right", "Up", "Down". A legal action is any of those actions that does not move the agent out of the board. Currently, there is no "Stay" action being incorporated.

(c) Transitions: Since there are 2 parts within the states, the agent position can be determined by the action, but the nearby Pokémons list is controlled by the board and therefore by randomness. The Pokémons are randomly generated as described in project proposal.

Base line agent

The state space of this game is huge. On an 50×50 board, there are 2500 different positions, but the real challenge comes with the list of nearby Pokémons. If each Pokémons could only appear once in the list, the overall number of different combinations would be 2^{151}, but still, since there could be 2 Pidgeys around at the same time, the number of combinations is in fact infinite.

The base line agent resolve this problem by utilizing Q-learning algorithm with function approximation. The weight vector w is initialized to be 0, and updated in the `incorperate_feedback` method:

$$w^{(t+1)} := w^{(t)} - \eta[\hat{Q}_{opt}(s, a, w) - r - \hat{V}_{opt}(s')]\phi(s, a)$$

More specifically:

(a) s is the old state, s' is the new state, and a is the action transit from s to s'

(b) η is the learning rate, computed by $\eta = \frac{1}{\sqrt{t}}$ where t is the current number of iterations

(c) r is the reward, equals the real scores issued by the board for catching Pokémons.

(d) The feature extractor ϕ constructs indicator features for all of the information from the current state. For example, if the current state is: {'agent_position': (20, 25), 'radar': [1,4]}, then the feature vector would be {'agent_at_x20': 1, 'agent_at_y25': 1, 'pokemon_1': 1, 'pokemon_4': 1}.

(e) $\hat{Q}_{opt}(s,a)$ is the estimation of the optimal value if take action a from state s, and $\hat{V}_{opt}(s)$ is the estimate of the optimal value from state s, they are computed by:

$$\hat{Q}_{opt}(s,a,w) = w^{(t)\top}\phi(s,a)$$

$$\hat{V}_{opt}(s') = \max_{a' \in \text{Actions}(s')} \hat{Q}_{opt}(s',a')$$

For the `get_action` method, the agent compute the $\hat{Q}_{opt}(s,a)$ for all the legal actions from the current state, and returns the best one. The agent maintains an exploration rate of 0.2 to take a random move.

Improved Agent

One major problem with the base line is that the reward comes too late to be useful. For example, the agent has to take 10 consecutive "good" moves in order to reach a Pokémon, but the reward would only be issued to the last move. The improved agent tries to tackle this problem by letting the agent issue some reward to itself to encourage "seemingly good" moves. Most of the infrastructure of the improved agent is exactly the same as the base line agent, except the following:

(a) The agent now keeps track of the rareness of the Pokémons, by counting the Pokémons it has seen. A counter has been setup for each type of Pokémons, and is incremented whenever the agent sees one Pokémon of that type. To "see" a Pokémon, the agent don't need to actually catch it, the counter would be incremented as long as the Pokémon is in the radar.

(b) In the `incorperate_feedback` method, the reward is modified by:

$$r' = r + \gamma \sum_{R_i} \frac{1}{C_{R_i}} \mathbf{I}\{R_i \in P\}$$

where r is the score issued by the board (most of the time equals 0), C_{R_i} is the counter for the ith Pokémon in the current radar R. P is the Pokémons in the previous radar, which should be saved and updated for every iteration. Here C_{R_i} would never be 0, because if we've seen R_i in the last radar, C_{R_i} would be at least 1. As a result, this means that now we're encouraging the agent to keep the Pokémons in the radar, instead of loosing track of them. Lastly, γ is just a constant, by which we can use to adjust how much do we care about this, compared with the real scores.

Reflect Agent

Reflect agent keeps track of the locations of actual Pokémons, and based on the history to find Pokémons in the radar. The intuition behind this is that if I've caught a Growlithe at the back of my apartment, the next time when I see a Growlithe on the radar, that might be the first place I want to go and check. As a result, instead of seeing the game as a MDP, the reflect agent keep track of a list of "centers" for different types of Pokémons, and when a Pokémon shows up in the radar, the agent goes to the corresponding center. Here's details of implementation:

(a) The reflect agent also keep track of the rareness of Pokémons, using the same method as the improved agent, so we have C_i equals how many times we've seen the Pokémon (of ID i).

(b) In the `incorperate_feedback` method, we update the rareness count as before, and also, update the "centers". For each Pokémon in the radar, we sample a random point on the board near the agent from a Gaussian distribution, let's call this point P. If this is the first time the agent sees this Pokémon, the center O_i would be set to this point, otherwise, the x and y coordinates of the center would be updated by this rule:

$$O_i := \frac{O_i(C_i - 1) + P}{C_i}$$

If the agent has actually caught a Pokémon, instead of sampling random nearby position, we just set P to be the current agent position and update by $O_i = \frac{O_i+P}{2}$. The intuition is that we're giving higher weight to the actual catching positions, because we're more confident about its usefulness.

(c) In the `get_action` method, the agent makes the decision by considering the "centers" for all the Pokémons in radar. For a given Pokémon with ID i, if the center O_i is on the upper right of the current agent position, this Pokémon would vote for "Up" (or "Rgiht") with voting power $\frac{1}{C_i}$, the direction with the highest total votes wins. Still, there is a 0.2 chance for the agent to make a random move.

Current results and Discussion

With 30000 iterations, on the "regional board" (50×50), the average result for 10 runs is presented:

Agent	Percentage of Scores	Percentage of Pokémons
Random Agent	0.12%	0.10%
Base line Agent	0.12%	0.12%
Improved Agent	0.43%	0.52%
Reflect Agent	1.13%	0.98%
Oracle Agent	49.2%	17.9%

Currently, the best agent is the reflect agent, however, we've made a huge assumption for the reflect agent, which is each type of Pokémons only have one "center". The key parameters for controlling this includes `board.nearby_variance`, which controls the position variance of a certain type of Pokémons, and `pokemon_selector.neighbor_rate`, which controls the probability of adding neighbors for an initial spawn (the mechanism for the `pokemon_selector` is described in the proposal). For an extreme case, if `neighbor_rate = 0` and `nearby_variance = 0.01`, the performance of reflect agent could be really great (6% score and 8% Pokémon), but in the ordinary setting, where `neighbor_rate = 0.15` and `nearby_variance = 0.25 * board.size`, the result is listed in the above table.

Overall, some progress has been made but there's a still a lot of space to improve. For the next step, I want to further generalize based on the reflect agent, constructing features for every position on the board. The feature values should be related with the probability of finding Pokémons on that spot. From the perspective of the agent, seeing a Pokémon in the radar (though not knowing the exact position), still increases the probability of all of the nearby positions to hold that Pokémon.