Nim

Description

Write an AI that teaches itself to play Nim through reinforcement learning.

Introduction

Recall that in the game Nim, we begin with some number of piles, each with some number of objects. Players take turns: on a player's turn, the player removes any non-negative number of objects from any one non-empty pile. Whoever removes the last object loses.

There's some simple strategy you might imagine for this game: if there's only one pile and three objects left in it, and it's your turn, your best bet is to remove two of those objects, leaving your opponent with the third and final object to remove. But if there are more piles, the strategy gets considerably more complicated. In this problem, we'll build an AI to learn the strategy for this game through reinforcement learning. By playing against itself repeatedly and learning from experience, eventually our AI will learn which actions to take and which actions to avoid.

In particular, we'll use Q-learning for this project. Recall that in Q-learning, we try to learn a reward value (a number) for every `(state, action)` pair. An action that loses the game will have a reward of -1, an action that results in the other player losing the game will have a reward of 1, and an action that results in the game continuing has an immediate reward of 0, but will also have some future reward.

How will we represent the states and actions inside of a Python program? A "state" of the Nim game is just the current size of all of the piles. A state, for example, might be `[1, 1, 3, 5]`, representing the state with 1 object in pile 0, 1 object in pile 1, 3 objects in pile 2, and 5 objects in pile 3. An "action" in the Nim game will be a pair of integers `(i, j)`, representing the action of taking `j` objects from pile `i`. So the action `(3, 5)` represents the action "from pile 3, take away 5 objects." Applying that action to the state `[1, 1, 3, 5]` would result in the new state `[1, 1, 3, 0]` (the same state, but with pile 3 now empty).

Recall that the key formula for Q-learning is below. Every time we are in a state `s` and take an action `a`, we can update the Q-value `Q(s, a)` according to:

```
Q(s, a) <- Q(s, a) + alpha * (new value estimate - old value estimate)
```

In the above formula, `alpha` is the learning rate (how much we value new information compared to information we already have). The `new value estimate` represents the sum of the reward received for the current action and the estimate of all the future rewards that the player will receive. The `old value estimate` is just the existing value for `Q(s, a)`. By applying this formula every time our AI takes a new action, over time our AI will start to learn which actions are better in any state.

Understanding the Environment

First, open up `nim.py`. There are two classes defined in this file (`Nim` and `NimAI`) along with two functions (`train` and `play`). `Nim`, `train`, and `play` have already been implemented for you, while `NimAI` leaves a few functions left for you to implement.

Take a look at the `Nim` class, which defines how a Nim game is played. In the `__init__` function, notice that every Nim game needs to keep track of a list of piles, a current player (0 or 1), and the winner of the game (if one exists). The `available_actions` function returns a set of all the available actions in a state. For example, `Nim.available_actions([2, 1, 0, 0])` returns the set `{(0, 1), (1, 1), (0, 2)}`, since the three possible actions are to take either 1 or 2 objects from pile 0, or to take 1 object from pile 1.

The remaining functions are used to define the gameplay: the `other_player` function determines who the opponent of a given player is, `switch_player` changes the current player to the opposing player, and `move` performs an action on the current state and switches the current player to the opposing player.

Next, take a look at the `NimAI` class, which defines our AI that will learn to play Nim. Notice that in the `__init__` function, we start with an empty `self.q` dictionary. The `self.q` dictionary will keep track of all of the current Q-values learned by our AI by mapping `(state, action)` pairs to a numerical value. As an implementation detail, though we usually represent `state` as a list, since lists can't be used as Python dictionary keys, we'll instead use a tuple version of the state when getting or setting values in `self.q`.

For example, if we wanted to set the Q-value of the state `[0, 0, 0, 2]` and the action `(3, 2)` to `-1`, we would write something like

```
self.q[(0, 0, 0, 2), (3, 2)] = -1
```

Notice, too, that every `NimAI` object has an `alpha` and `epsilon` value that will be used for Q-learning and for action selection, respectively.

The `update` function is written for you, and takes as input state `old_state`, an action take in that state `action`, the resulting state after performing that action `new_state`, and an immediate reward for taking that action `reward`. The function then performs Q-learning by first getting the current Q-value for the state and action (by calling `get_q_value`), determining the best possible future rewards (by calling `best_future_reward`), and then using both of those values to update the Q-value (by calling `update_q_value`). Those three functions are left to you to implement.

Finally, the last function left unimplemented is the `choose_action` function, which selects an action to take in a given state (either greedily, or using the epsilon-greedy algorithm).

The `Nim` and `NimAI` classes are ultimately used in the `train` and `play` functions. The `train` function trains an AI by running `n` simulated games against itself, returning the fully trained AI. The `play` function accepts a trained AI as input, and lets a human player play a game of Nim against the AI

Task Specification

Complete the implementation of `get_q_value`, `update_q_value`, `best_future_reward`, and `choose_action` in `nim.py`. For each of these functions, any time a function accepts a `state` as input, you may assume it is a list of integers. Any time a function accepts an `action` as input, you may assume it is an integer pair `(i, j)` of a pile `i` and a number of objects `j`.

The `get_q_value` function should accept as input a `state` and `action` and return the corresponding Q-value for that state/action pair.

- Recall that Q-values are stored in the dictionary `self.q`. The keys of `self.q` should be in the form of `(state, action)` pairs, where `state` is a tuple of all piles sizes in order, and `action` is a tuple `(i, j)` representing a pile and a number.
- If no Q-value for the state/action pair exists in `self.q`, then the function should return `0`.

The `update_q_value` function takes a state `state`, an action `action`, an existing Q value `old_q`, a current reward `reward`, and an estimate of future rewards `future_rewards`, and updates the Q-value for the state/action pair according to the Q-learning formula.

- Recall that the Q-learning formula is: `Q(s, a) <- old value estimate + alpha * (new value estimate - old value estimate)`
- Recall that `alpha` is the learning rate associated with the `NimAI` object.
- The old value estimate is just the existing Q-value for the state/action pair. The new value estimate should be the sum of the current reward and the estimated future reward.

The `best_future_reward` function accepts a `state` as input and returns the best possible reward for any available action in that state, according to the data in `self.q`.

- For any action that doesn't already exist in `self.q` for the given state, you should assume it has a Q-value of 0.
- If no actions are available in the state, you should return 0.

The `choose_action` function should accept a `state` as input (and optionally an `epsilon` flag for whether to use the epsilon-greedy algorithm), and return an available action in that state.

- If `epsilon` is `False`, your function should behave greedily and return the best possible action available in that state (i.e., the action that has the highest Q-value, using 0 if no Q-value is known).
- If `epsilon` is `True`, your function should behave according to the epsilon-greedy algorithm, choosing a random available action with probability `self.epsilon` and otherwise choosing the best action available.
- If multiple actions have the same Q-value, any of those options is an acceptable return value.

Implementation Hints

If `lst` is a list, then `tuple(lst)` can be used to convert `lst` into a tuple.

Submission Guidelines

You should not need to modify anything else in `nim.py` other than the functions the specification calls for you to implement, though you may write additional functions and/or import other Python standard library modules. You may also import `numpy` or `pandas`, if familiar with them, but you should not need to use any other third-party Python modules. You may modify `play.py` to test on your own, but when grading your peers please use the original file.

Peer-grading Guidelines

Open `play.py` in an editor and change the value of the variable `peer_grading_mode` to `True`. Run `play.py` which will start by training the AI agent of your peer student, then will allow you to play 5 games against it. Try to play to beat the AI:

The number of training episodes is set at 10,000 by default. Do not change that value when doing peer-grading. The training will take some time, but it should not exceed 5 minutes.

1- If the AI beat you on all of them, give full points.
2- If you were able to beat the AI on some games but not others, give half points.
3- If you won all games or if the AI takes longer than 1 minute to think of its move, give zero points.

Note: After peer-grading is done, consider playing against your peer student's AI with output produced by your own AI.