Reading Assignment One

Amol Kapoor, Adam Cobb
Worcester College, Oxford University

May 11, 2017

Exercise 1.1: Self-Play. Suppose, instead of playing against a random opponent, the reinforcement learning algorithm described above played against itself. What do you think would happen in this case? Would it learn a different way of playing?

The AI will eventually converge on the optimal choices for each state in the game. This is because both the AI and it's opponent (another version of the AI) are built to exploit mistakes and learn by bootstrapping future state values. If one AI makes a mistake in a given state, it will eventually learn to correct it. It should eventually learn to play better against itself than against an imperfect player, though if the imperfect player does not make the same mistakes over and over again, with infinite replays both AI's will converge to the same decision tree.

Exercise 1.2: Symmetries. Many tic-tac-toe positions appear different but are really the same because of symmetries. How might we amend the reinforcement learning algorithm described above to take advantage of this? In what ways would this improve it? Now think again. Suppose the opponent did not take advantage of symmetries. In that case, should we? Is it true, then, that symmetrically equivalent positions should necessarily have the same value?

If symmetries are treated as equivalent states, actions (and updates) taken from one state can be used to back up multiple states simultaneously, allowing the algorithm to converge faster (more specifically, one could keep a map of which states are equivalent to other symmetric states, and then for each update step for V(s) update all equivalent states as well). Since there are numerous axis of symmetry, the AI could speed up the algorithm significantly. In the case where our opponent does not use

symmetries, it would likely be better in the long run to still use symmetries (more data applied to the same state would likely lead to faster convergence) but would result in more mistakes/losses in the short term (the AI will try to cross apply experiences from one symmetric state that won't translate because the opponent will behave differently). Additionally, optimal play may be slower than taking advantages of mistakes that the opponent may make.

Exercise 1.3: Greedy Play. Suppose the reinforcement learning player was greedy, that is, it always played the move that brought it to the position that it rated the best. Would it learn to play better, or worse, than a nongreedy player? What problems might occur?

It would play worse. Without any exploration, the RL player would simply settle on whichever route leads to positive reward first. The odds that this route is the optimal one is very low. Further, if the opponent learns, the greedy player will not be able to adapt. Depending on the reward scheme and how the step size is reduced, theoretically even a greedy player should converge to the optimal solution, though it would take a lot longer, and would only occur if the opponent does not continuously make the same mistakes.

Exercise 1.4: Learning from Exploration. Suppose learning updates occurred after all moves, including exploratory moves. If the step-size parameter is appropriately reduced over time, then the state values would converge to a set of probabilities. What are the two sets of probabilities computed when we do, and when we do not, learn from exploratory moves? Assuming that we do continue to make exploratory moves, which set of probabilities might be better to learn? Which would result in more wins?

Backing up exploratory moves gives an incorrect value for the state being updated, as the future state that is being used for the update is not the optimal choice and therefore not representative of the future state's actual value. Over time, then, we could expect the exploratory-learning AI to converge on a set of probabilities that are less optimal than the base AI - the final state values would include the expected value of the exploration, not the actual value of the state. If we continued to explore, the state values that included the exploration expected value would actually be more accurate representations of what would happen in a given state. With infinite games, playing greedily with respect to accounting for the exploration would lead to more wins using that algorithm, as the AI would better understand its likelihood of winning while still exploring.

Exercise 1.5: Other Improvements. Can you think of other ways to improve the reinforcement learning player? Can you think of any better way to solve the tic-tac-toe problem as posed?

Tic-tac-toe is a learn-able game with an optimal solution. However, the AI as written will continue to explore; even if the exploration likelihood degrades over time, this can lead to a lot of sub-optimal games. Because the player is trying to maximize wins, a RL algorithm that removed the exploration constant entirely once a state with a certain win likelihood was reached would play better. This could be implemented with a simple threshold check before running any exploration likelihood.

The current AI considers ties the same as a loss for simplicity. In reality, a tie and a loss are very different, and the AI should prefer a tie state to a loss state. Right now state values correspond to likelihood of winning, but they do not have to. If a tie was given an arbitrary value between 0 (loss) and 1 (win) instead of just 0, the AI would eventually learn to prefer ties over losses.

Exercise 2.2 (programming). How does the soft max action selection method using the Gibbs distribution fare on the 10-armed testbed? Implement the method and run it at several temperatures to produce graphs similar to those in Figure 2.1. To verify your code, first implement the -greedy methods and reproduce some specific aspect of the results in Figure 2.1.

It should perform better as it is not weighting all exploration selection equally. See attached for code and image.

Exercise 2.4 Give pseudocode for a complete algorithm for the n-armed bandit problem. Use greedy action selection and incremental computation of action values with a = 1/k step-size parameter. Assume a function bandit(a) that takes an action and returns a reward. Use arrays and variables; do not subscript anything by the time index t (for examples of this style of pseudocode, see Figures 4.1 and 4.3). Indicate how the action values are initialized and updated after each reward. Indicate how the step-size parameters are set for each action as a function of how many times it has been tried.

```
//assumes that the max output for a bandit is 1
k = 0
Q = [1] //array of bandit averages initialized to 1
```

```
        // this forces initial exploration of all bandits
eps = .1

while true:
    k++

    //get a random bandit to explore based on exploration
    //factor, otherwise choose greedily
    a = random(Q) if random < eps else argmax(Q)

    Q(a) = Q(a) + 1/k * [bandit(a) - Q(a)]

    //decay eps greedy exploration
    eps = decay(eps)
```