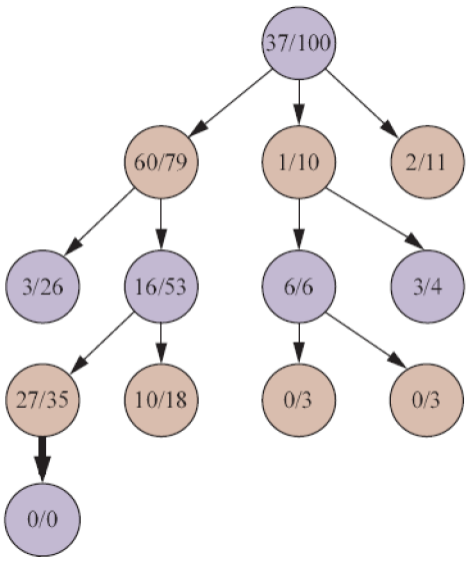
COMS W4701: Artificial Intelligence, Summer 2022

Homework 3

Instructions: Compile all solutions to the written problems on this assignment in a single PDF file. **Show your work by writing down relevant equations or expressions, and/or by explaining any logic that you are using to bypass known equations**. Coding solutions may be directly implemented in the provided Python file(s). **Do not modify the filename or any code outside of the indicated sections.** When ready, follow the submission instructions to submit all files to Gradescope. Please be mindful of the deadline and our late policy, as well as our course policies on academic honesty.

Problem 1: MCTS Practice (12 points)

The partial game tree below was discussed in class on the topic of Monte Carlo tree search. Each node shows the *win rate*: number of playout wins / total number of playouts from that node. The leaf node labeled 0/0 was just expanded in the middle of a MCTS iteration.



(a) Suppose that a rollout is performed and the player corresponding to the purple nodes (root, third layer, and newly expanded leaf) wins. Sketch a copy of the tree and indicate all win rates after backpropagation, whether updated or not.

(b) Using the **new** win rates from your tree in (a) above and the exploration parameter $\alpha = 1$, compute the UCT values of each of the nodes in the second layer of the tree (immediate children of the root node). Which of these three nodes is traversed by the selection policy in the next MCTS iteration?

(c) We will refer to the child node you found above as n. Suppose that the player corresponding to the second layer (orange) always loses in simulation. After how many iterations of MCTS will a child node of the root *different* from n be selected? (You may use a program like WolframAlpha to solve any nonlinear equations.)

(d) Starting once again from your tree in (a), solve for the minimum value of α for which a child node of the root different from n be selected in the *next* MCTS iteration. Explain why we need a *higher*, not lower, α value in order to select one of the other two nodes.

Problem 2: Mini-Blackjack (12 points)

We will model a mini-blackjack game as a MDP. The goal is to draw cards from a deck containing 2s, 3s, and 4s (with replacement) and stop with a card sum as close to 6 as possible without going over. The possible card sums form the states: 0, 2, 3, 4, 5, 6, "done". The last state is terminal and has no associated actions. From all other states, one action is to **draw** a card and advance to a new state according to the new card sum, with "done" representing card sums of 7 and 8. Alternatively, one may **stop** and receive reward equal to the current card sum, also advancing to "done" afterward.

(a) Draw a state transition diagram of this MDP. The diagram should be a graph with seven nodes, one for each state. Draw edges that represent transitions between states due to the **draw** action only; you may omit transitions due to the **stop** action. Write the transition probabilities adjacent to each edge.

(b) Based on the given information and without solving any equations, what are the optimal actions and values of states 5 and 6? You may assume that $V^*(\text{done}) = 0$. Then using $\gamma = 0.9$, solve for the optimal actions and values of states 4, 3, 2, and 0 (you should do so in that order). Briefly explain why dynamic programming is not required for this particular problem.

(c) Find the largest value of γ that would possibly lead to a different optimal action in state 3 (compared to those above) but leave all others the same. Is there any nonzero value of γ that would yield a different optimal action for state 0? Why or why not?

Problem 3: Dynamic Programming (12 points)

Let's revisit the mini-blackjack game but from the perspective of dynamic programming. You will be thinking about both value iteration and policy iteration at the same time. Assume $\gamma = 0.9$.

(a) Let's initialize the time-limited state values: $V_0(s) = 0$ for all s (we will ignore "done"). Find the state values of V_1 after one round of value iteration. You do not need to write out every calculation if you can briefly explain how you infer the new values.

(b) Coincidentally, $V_0 = 0$ are also the values for the (suboptimal) policy $\pi_0(s) =$ **draw** for all s. If we were to run policy iteration starting from π_0, what would be the new policy π_1 after performing policy improvement? Choose the **draw** action in the case of ties.

(c) Perform a second round of value iteration to find the values V_2. Have the values converged?

(d) Perform a second round of policy iteration to find the policy π_2. Has the policy converged?

Problem 4: Reinforcement Learning (12 points)

Let's now study the mini-blackjack game from the perspective of using reinforcement learning given a set of game episodes and transitions. This time, assume $\gamma = 1$. We initialize all values and Q-values to 0 and observe the following episodes of state-action sequences:

- 0, draw, 3, draw, done (reward = 0)
- 0, draw, 2, draw, 4, draw, done (reward = 0)
- 0, draw, 4, draw, 6, stop, done (reward = 6)
- 0, draw, 3, draw, 5, stop, done (reward = 5)
- 0, draw, 2, draw, 5, stop, done (reward = 5)

(a) Suppose that the above episodes were generated by following a fixed policy. According to Monte Carlo prediction, what are the values of the six states other than the "done" state? Explain whether the *order* in which we see these episodes affects the estimated state values.

(b) Suppose we use temporal-difference learning with $\alpha = 0.8$ instead. Write out **each** of the updates for which a state value is changed. Again explain whether the *order* in which we see these episodes affects the estimated state values.

(c) Now suppose we had generated the above transitions using Q-learning, starting with all Q-values initialized to zero. Which Q values are nonzero after all episodes are complete? Assuming that **draw** is the default "exploit" action in the case of equal Q values, which transitions would be considered exploratory?

Problem 5: "Real" Blackjack (52 points)

We will now extend mini-blackjack from 6 to 21 and add in a few other twists. We have 22 numbered states, one for each possible card sum from 0 to 21, in addition to the "done" terminal state. From each non-terminal state we can either **stop** and receive reward equal to the current card sum, or we can **draw** an additional card.

When drawing a card from the deck, any of the standard set of cards may show up, but the jack, queen, and king cards are treated as having value equal to 10 (aces will just be treated as 1s). We will still be drawing cards with replacement, so the probability of **drawing** a card with a value 1 through 9 is $\frac{1}{13}$ each, while the probability of obtaining a 10 value is $\frac{4}{13}$. Lastly, we will add in a constant living reward that is received with every **draw** action.

Given this information, we can thus model the problem as a Markov decision process and solve for the optimal policy and value functions. You will be implementing several of the dynamic programming and reinforcement learning algorithms in the provided blackjack Python file.

5.1: Value Iteration (12 points)

Implement `value_iteration` to compute the optimal values for each of the 22 non-terminal states. The first argument is the initial value function V_0 stored in a 1D NumPy array, with the index corresponding to the state. The other arguments are the living reward, discount factor, and stopping

threshold. As discussed in class, your stopping criterion should be based on the *maximum* value change between successive iterations. Your value updates should be **synchronous**; only use V_i to compute V_{i+1}. When finished, your function should return the converged values array V^*.

5.2: Policy Extraction (10 points)

An agent would typically care more for a policy than a value function. Implement `value_to_policy`, which takes in a set of values, living reward, and discount factor to compute the best policy associated with the provided values. Return the policy as a NumPy array with 22 entries (one for each state), with value 0 representing **stop** and value 1 representing **draw**.

5.3: DP Analysis (9 points)

Now that you can generate optimal policies and values, we can study the impact of living reward and discount factor on the problem. For each of the following experiments, you can simply use an initial set of values all equal to 0.

(a) Compute and plot the values V^* and policy π^* for living reward $lr = 0$ and $\gamma = 1$. You should see that V^* consists of three continuous "segments". Briefly explain why the discontinuities (which show up as "dips") between the segments exist, referring to the optimal policy found and game rules.

(b) Experiment with decreasing the discount factor, e.g. in 0.1 decrements. For sufficiently low values of γ you should see that the three segments of V^* merge into two. Show the plots for an instance of this effect and report the γ value you used. Briefly explain the changes that you observe in V^* and π^*.

(c) Reset γ to 1 and experiment with changing the living reward, e.g. using intervals of 1 or 2. Which segments of V^* shift for negative living rewards or slightly positive living rewards? In which direction do these values shift in each case? How does π^* change as these state values shift? Find approximate thresholds of the living reward in which π^* becomes **stop** in all states, and alternatively **draw** in all states.

5.4: Temporal-Difference Learning (12 points)

You will now investigate using reinforcement learning, and in particular the Q learning algorithm, to learn the optimal policy and values for our blackjack game purely through self-play. We will need to keep track of Q values; we can use a 22×2 NumPy array `Q`, so that we have `Q[s,a]`$= Q(s, a)$. `a=0` corresponds to the **stop** action and `a=1` corresponds to the **draw** action. To allow for exploration, we will use ε-greedy action selection.

The `Qlearn` function takes in an initial array of Q values, living reward lr, discount factor γ, learning rate α, exploration rate ε, and the number of transitions N. In addition to updating `Q`, your procedure should keep track of the states, actions, and rewards seen in a $N \times 3$ array `record`. After initializing the arrays and initial state, the procedure within the simulation loop is as follows:

- Use the ε-greedy method to determine whether we explore or exploit.
- If **stopping**, the reward is $r = s$, where s is the current state. If **drawing**, the reward is $r = lr$; also call the `draw()` function so that you can compute the successor state s'.

- Update the appropriate Q value, as well as `record` with the current state, action, and reward. You should use 0 for the "Q value" of the "done" state.
- Reset s to 0 if we either took the **stop** action or $s' > 21$, else set $s = s'$.

After finishing N transitions, return `Q` and `record`.

5.5: RL Analysis (9 points)

After you implement `Qlearn`, you should ensure that it is working correctly by comparing the learned state values (the maximum values in each row of `Q`) with those returned by value iteration. A suitable default set of learning parameters would be to have a low α (e.g., 0.1), low ε (e.g., 0.1) and high N (e.g., 50000).

Once you are ready, call the provided `RL_analysis` function (no additional code needed) and answer the following questions using the plots that you see. Your `Qlearn` **must** return the two arrays as specified above in order for this to work properly.

(a) The first plot shows the number of times each state is visited when running `Qlearn` with $lr = 0$, $\gamma = 1$, $\alpha = \varepsilon = 0.1$, and N ranging from 0 to 50k in 10k intervals. Explain how the value of N is important to ensuring that all states are visited sufficiently. You should also note that the two most visited states are the same regardless of N; explain why these two states attract so much attention.

(b) The second plot shows the cumulative reward received for a game using the same parameters above, except with $N = 10000$ and ε ranging from 0 to 1 in 0.2 intervals. Explain what you see with the $\varepsilon = 0$ curve and why we need exploration to learn. Looking at the other curves, how does increasing exploration affect the overall rewards received and why?

(c) The third plot shows the estimated state values using the same parameters above, except with $\varepsilon = 0.1$ and α ranging from 0.1 to 1. While the set of curves may be hard to read, each should bear some resemblance to the true state values V^*. What is a problem that arises when α is too low, particularly with less visited states? What is a problem that arises when α is too high? Try to compare the stability or smoothness of the different curves.

Submission

You should have one PDF document containing your solutions for problems 1-4, as well as the information and plots for 5.3 and 5.5. You should also have a completed Python file implementing problem 5; make sure that all provided function headers and the filename are unchanged. Submit the document and `.py` code file to the respective assignment bins on Gradescope. For full credit, you must tag your pages for each given problem on the former.