Homework Assignment

Course: Deep Reinforcement Learning

Sophie Haas, Johanna Linkemeyer, Tjorven Weber (Group 9)

April 30, 2022

1 Task 1

You are tasked with creating an AI for the game of chess. To solve the problem using Reinforcement Learning, you have to frame the game of chess as a Markov Decision Process (MDP). Describe both the game of chess formally as a MDP, also formalize the respective policy.

Solution: The Markov Decision Process consists of the set of states S, the set of Actions A, the state dynamics and the reward dynamics. In the example of chess, the **states** $s \in S$ are described by the board configurations. The **actions** $a \in A$ are the movements of one of the own pieces within the boundaries of the board and the allowed movement patterns defined for each type of piece (e.g. pawns: move one or two fields to the front; move to the front left or right diagonal if occupied by an opponents piece). The **state dynamics** are deterministic, i.e. when taking the same action in the same current state, the same next state will be reached. The **reward dynamics** are described by a reward of $+1$ for winning, and -1 for losing. Related to the MDP are the **agent**, which in the case of the chess is the player, and the policy that it uses. The **policy** $\pi(a|s)$ is a joint probability distribution over all of the categorical distributions for each chess figure, where the categories are the respective actions (i.e. possible moves) given the current state (i.e. board configuration).

2 Task 2

Check out the LunarLander environment on OpenAI Gym: Check out this Link!. Describe the environment as a MDP, include a description how the policy is formalized.

Solution: The Markov Decision Process consists of the set of states S, the set of Actions A, the state dynamics and the reward dynamics. In the example of the LunarLander, the **states** $s \in S$ are described by the pixel space. The **actions** $a \in A$ are: do nothing, fire left orientation engine, fire main engine, and fire right orientation engine. The **state dynamics** are a deterministic physics simulation. The **reward dynamics** are the following: "Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the

lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points." Related to the MDP are the **agent**, which in the case of the LunarLander is the lander, and the policy that it is using. The **policy** is a categorical distribution, where the categories are the possible actions (i.e. do nothing, fire left orientation engine, fire main engine, and fire right orientation engine).

3 Task 3

Discuss the Policy Evaluation and Policy Iteration algorithms from the lecture. They explicitly make use of the environment dynamics (p(s',r—s,a)).

- *Explain what the environment dynamics (i.e. reward function and state transition function) are and give at least two examples.*
- *Discuss: Are the environment dynamics generally known and can practically be used to solve a problem with RL?*

Solution: Both the **policy evaluation** and the **policy iteration algorithm** are part of the dynamic programming approach, which uses the Bellman Equation as update rule to approximate the value-state function, and in the case of the policy iteration, subsequently improve the policy. Both require full access to the environment dynamics.

The **policy evaluation algorithm**, as the name says, evaluates the "goodness" of the current policy, that is, the value of each state when following a policy π. After initializing all state values arbitrarily, this is done iteratively, by looping over every state and updating the values with the Bellman Equation:

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) * [r + \gamma * V(s')]$$

The equation computes the expected return for all actions and possible successor states, weighted by (1) the probability of action a given s (defined through policy π) and (2) given state s and action a, the probability of reaching successor state s' with reward r (defined through the state transition function). In each iteration, the error Δ is tracked, which is the max of the previous error and the difference between the old and and the new state value. The loop runs, until Δ is below a predefined threshold ϵ.

The **policy iteration algorithm** converges to an optimal policy (i.e. the policy that maximizes the **return**) by iterating over the two steps policy evaluation and policy improvement. The policy is evaluated with the policy evaluation algorithm as described above. Following that, in each iteration the policy can be improved based on the new state values. This is done by going over all states and assigning to π(s) the action a that maximizes the expected return

$$\pi(s) \leftarrow argmax_a \sum_{s',r} p(s', r|s, a) * [r + \gamma * V(s')]$$

As the evaluation of the policy (obviously) depends on the policy, and the improvement policy on the previous evaluation, both steps are repeated until the policy does not change any further.

Regarding the **terms**: The **reward** is what we want to optimize. The **reward function** describes how an agent should optimally behave, i.e. it defines good and bad events (only immediate, not considering the long run). One example would be that of the racecar environment. Here, there is a negative reward for each frame and a positive reward for every track tile visited, ultimately creating the goal to finish as fast as possible. The second example is that of a pick up and place task of a robotic arm where there is a positive reward for successfully picking up an object and placing it in a desired place and a negative reward for dropping the object, or placing it in the wrong place. The **state transition function** describes the transition to the next state (and receiving a reward) based on the current state of the environment and the action the agent took previously, i.e. the mapping of actions to states. The state transition function can be probabilistic, e.g. in a weather scenario, but there are also deterministic (0 or 1) sate transition functions, e.g. for chess when taking the same action in the same current state, the same next state will be reached.

Regarding the **applicability**: In almost all real-world problems the environment dynamics are not completely known, as they depend on other agents (e.g. in chess, the opponent). And even if the complete dynamics are accessible, as e.g. in Backgammon, the computing power that it takes to compute the optimal policy and value function is so large that it is still not practical for real life application. Instead, reinforcement learning usually tries to approximate the optimal value function and policy instead of computing exactly based on the transition dynamics.