Computational Intelligence WS24/25

Exercise Sheet 5 (Solution) — Dezember 17th, 2024

Thomas Gabor, Maximilian Zorn, Claudia Linnhoff-Popien

1 Multi-Cow Systems

We consider a multi-agent system made up of $n = 20$ cows and a mowing robot called *MowBot*. All these 21 agents live on an infinite meadow which we model as an infinite 2d plane of positions $l \in \mathbb{R}^2$. The agent $G^{[0]}$ is the *MowBot* and all other agents $G^{[1]}, ..., G^{[20]}$ are cows.

The evolution of the multi-agent system is encoded as a series of states $\langle s_t \rangle_t$ of unspecified length for states $s_t \in \mathcal{S}$ at time t, where $\mathcal{S} = \mathbb{N} \times (\mathbb{R}^2)^{21}$ is the (in total 43-dimensional) state space, as well as a series of joint observations $\langle o_t \rangle_t = \langle (o_t^{[0]}, ..., o_t^{[20]}) \rangle_t$ made by the agents and a series of joint actions $\langle a_t \rangle_t = \langle (a_t^{[0]}, ..., a_t^{[20]}) \rangle_t$ executed by the agents.

Our system is fully observable. Thus, each agent $G^{[i]}$ makes observations

$$o_t^{[i]} = s_t = (t, l_t^{[0]}, ..., l_t^{[20]}) \in \mathcal{S}$$

at each time step t where $l_t^{[i]} \in \mathbb{R}^2$ is the position of the agent $G^{[i]}$ at time $t \in \mathbb{N}$. Furthermore, any agent $G^{[i]}$ uses the identical action space $\mathcal{A}^{[i]} = \{$`do_nothing`, `go_northeast`, `go_east`, `go_southeast`, `go_south`, `go_southwest`, `go_west`, `go_northwest`, `go_north`, `eat/mow`$\}$ with $|\mathcal{A}^{[i]}| = 10$ where `do_nothing` has the agent not do anything, all actions starting with `go_` cause the agent to move for a certain length in the respective direction, and `eat/mow` causes any cow agent to eat the grass and any *MowBot* agent to mow the grass.
You can use a 2d geometric distance function $dist(l, l') = \sqrt{(x - x')^2 + (y - y')^2}$ where $l = (x, y) \in \mathbb{R}^2$ and $l' = (x', y') \in \mathbb{R}^2$.

(i) Give a goal predicate $\gamma_1 : \langle \mathcal{S} \rangle \times \langle \mathcal{A} \rangle \to \mathbb{B}$ so that γ_1 holds iff the *MowBot* has never (at full time steps) been closer than a distance of 1 to any of the cows.

$$\gamma_1\Big(\langle (t, l_t^{[0]}, ..., l_t^{[20]}) \rangle_t, \langle (a_t^{[0]}, ..., a_t^{[20]}) \rangle_t\Big) \iff \forall t : \forall j \in [1; 20] \subset \mathbb{N} : dist(l_t^{[0]}, l_t^{[j]}) \geq 1$$

(ii) Let $(\mathcal{S}, \mathcal{A}, \mathbb{R}, P, R)$ be a Markov decision process (MDP, cf. Definition 10 on the def sheet) with state space $\mathcal{S}$ and joint action space $\mathcal{A}$ from the scenario above, target space $\mathcal{T} = \mathbb{R}$, and a given transition probability function P. Give — from our *MowBot*'s perspective — a reward function R that both incentivizes to (a) fulfill γ_1 and (b) execute `mow/eat` as often as possible.

$$R(s_t, a_t, s_{t+1}) = \begin{cases} -10 & \text{if } \exists j \in [1;20] : dist(l_{t+1}^{[0]}, l_{t+1}^{[j]}) < 1, \\ +1 & \text{else if } a_t = \texttt{eat/mow}, \\ 0 & \text{else.} \end{cases}$$

(iii) Now, in any run at time step $t = 26$ all cows suddenly change their policy to a specific new (from then on again fixed) policy.
Can R still be learned as the reward function of an MDP? Briefly state your reasoning.

Yes, because the time is part of the state space and time dependent changes can thus be recognized when learning.

(iv) Let $\gamma_2 : \langle \mathcal{S} \rangle \times \langle \mathcal{A} \rangle \to \mathbb{B}$ be goal predicate so that γ_2 holds iff the *MowBot* has never executed the action `eat/mow` within a radius of 1 from any point where any cow has ever executed `mow/eat`. Assume that the cows' policies are fixed.
Can the *MowBot* learn to fulfill γ_2 effectively for an MDP with the given state space $\mathcal{S}$ for some reward function R'? Briefly state your reasoning.

No, because past eaten patches are not encoded in the state, so if the *MowBot* implements an MDP, it can never be certain where cows might have previously eaten grass.

2 Markov's Vacuum

Recall our Vacuum World running example from the lecture. If the agent chooses to vacuum, it receives a reward of $+10$ if the room it is in was dirty or -1 if it was not dirty. Other actions inflict no reward nor cost. The initial state of the Vacuum World is given as the agent being positioned in room A (clean), and room B being dirty.

(i) Provide a suitable definition for all elements in an MDP tuple to define the initial Vacuum World of the abovementioned setup and a randomly acting agent. Then, also define all additional (initial) elements in an MDP run tuple for completeness.

For the MDP tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, P, R)$ we need:

$$\begin{aligned}
\mathcal{S} &= \{A, B\} \times \{\textit{dirty}, \textit{clean}\} \times \{\textit{dirty}, \textit{clean}\},\\
\mathcal{A} &= \{\textit{switch}, \textit{vacuum}\},\\
\mathcal{T} &= \mathbb{R},\\
R(s, a, s') &= \begin{cases} +10 & \text{if } s \in \{(A, \textit{dirty}, _), (B, _, \textit{dirty})\} \text{ and } a = \textit{vacuum},\\ -1 & \text{if } s \in \{(A, \textit{clean}, _), (B, _, \textit{clean})\} \text{ and } a = \textit{vacuum},\\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

For the MDP run we additionally need:

$$\begin{aligned}
\pi(s) &= a \sim \mathcal{A},\\
\langle a_t \rangle &= [],\\
\langle s_t \rangle &= \langle (A, \textit{clean}, \textit{dirty}) \rangle,\\
\tau(\pi) &= \sum_{t \in \mathcal{Z}} R(s_t, a_t, s_{t+1})
\end{aligned}$$

(ii) Consider a fixed policy π for our robot vacuum agent, which executes the actions *vacuum*, *switch*, *vacuum*, *vacuum* in that order and then stops execution. Execute π and for each generated state s during the execution (including the initial state $s = s_0$) compute its value $V^\pi(s)$ given a fully deterministic state transition function P and $\gamma = 1$. To this end, consider the Bellman equation for computing the value function via:

$$V^\pi(s) = R(s, \pi(s), s') + \gamma \cdot \sum_{s' \in S} P(s' | s, \pi(s)) \cdot V^\pi(s').$$

$$
\begin{aligned}
s_0 &= \{A, \mathit{clean}, \mathit{dirty}\} \\
s_1 &= \{A, \mathit{clean}, \mathit{dirty}\} \\
s_2 &= \{B, \mathit{clean}, \mathit{dirty}\} \\
s_3 &= \{B, \mathit{clean}, \mathit{clean}\} \\
s_4 &= \{B, \mathit{clean}, \mathit{clean}\} \\
V^{\pi}(s_4) &= 0 \\
V^{\pi}(s_3) &= -1 \\
V^{\pi}(s_2) &= 9 \\
V^{\pi}(s_1) &= 9 \\
V^{\pi}(s_0) &= 8
\end{aligned}
$$

3 Rooms Policy Network

(a) (b)

Figure 1: The *rooms* domain and its corresponding state value map. Top left is the agents initial starting position, bottom right is the goal position.

Consider the *state values* noted in Fig. 1b as being produced by an optimal policy π^* according to $\pi^*(s) = \arg\max_{a \in A} V^*(s')$, where $s' \sim P(s'|s_t, a)$. For your computation assume $\gamma = 0.99$ and a reward function

$$
R(s, a, s') = \begin{cases} 1 & \text{if } \mathit{goal}, \\ 0 & \text{otherwise}, \end{cases}
$$

to be maximized over a given time horizon T via the *discounted reward*

$$
G_T = \sum_{t=0}^{\text{T-1}} \gamma^t \cdot r_t.
$$

(i) Mark in Fig. 1a one optimal policy π^* that would produce the value map seen in Fig 1b.

One example for π^* is the route along the left bottom wall, i.e., `go_down`$*6$, `go_right`$*6$.

(ii) Compute the *discounted reward* (also defined in Def. 10 (MDP)) for your optimal policy π^*. (For the sake of computation we assume that the agent stops upon reaching the goal state.)

There are multiple optimal policies that reach the goal in 12 steps. Remember, the max operator is not deterministic when multiple transition probabilities are the same, instead a random action is sampled from the set of equally optimal actions. With the above example for π^*, i.e., `go_down`$*6$, `go_right`$*6$.
Discounted reward:

$$\begin{aligned} R(\pi^*) &= r_1 + \gamma r_2 + \cdots + \gamma^{11} r_1 2 \\ &= 0 + 0.99^1 * 0 + 0.99^2 * 0 + \cdots + 0.99^{11} * 1 \\ &= 0.99^{11} \approxeq 0.895 \end{aligned}$$

Building on the concept of neural networks, we will now implement a so-called "policy network" as a decision making agent. This network will iteratively learn a probability distribution over the various actions in a given state, thus learning from experience (i.e., by visiting states in the rooms domain). This approach makes it possible to refine decision making in complex scenarios and better identify optimal actions.

We will test our agent on the *rooms* domain, implemented in the `./rooms` folder, which is instantiated with the `load_env` function. The Rooms domain allows four actions (NORTH = 0, SOUTH = 1, WEST = 2, EAST = 3) with which the agent can move in all directions. All actions are represented by integer values. If the agent tries to navigate against a wall, nothing happens. An `episode`, i.e., a sequence of actions, ends when the agent has reached the goal or 100 time steps have elapsed.

The environment has implemented the reward function

$$R(s, a, s') = \begin{cases} 1 & \text{if action } a \text{ reached the goal position,} \\ 0 & \text{otherwise,} \end{cases}$$

which our agent will use to calculate its *return* G_T of a run, given via

$$G_T = \sum_{t=0}^{\text{T-1}} \gamma^t \cdot r_t,$$

where $\gamma = 0.99$ is a *time discount factor*, for the given time horizon $T = \min\{100, t_{reached\_goal}\}$.

(iii) Given that our agent learns from the evaluation of its policy via the return G_T, the discount factor γ might seem counterintuitive at first. Briefly argue for the functionality of this discount factor γ^t and how it might affect the learning result.

Time discounted rewards ensure that actions taken very 'far' from the initial position are weighted less, thus prioritizing rewards taken early in the episode and close to the agent.

(iv) Using the ML building blocks we have introduced in Exercise Sheet 4 (network, optimizer, `forward()`, `backward()`, and optimizer update), code in the blank `TODO`s in `agent.py`. Complete

0. Uncomment the `PolicyAgent` in the main.py.
1. Initializing the Adam optimizer for the parameters of this network with the learning rate of $\alpha = 0.01$.
2. Computing the forward pass of an input through the network.
3. Calculate the loss in `agent.py`, by summing all elements in the output tensor. The loss $\mathcal{L}$ is calculated as

$$\mathcal{L} = -\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot G_t.$$

4. Compute a backward step and let the optimizer update the network parameters.

Example implementation of the TODOs see solution

(v) Observe the trained behavior in the `rooms9-fixed` environment. How does the agent perform after 1500, 2500, 3500 etc. steps?