

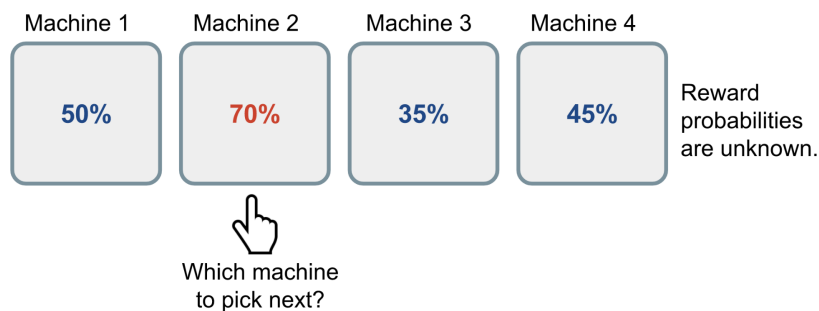
Computational Intelligence WS24/25

Exercise Sheet 6 (Solution) — January 9th, 2025

Thomas Gabor, Maximilian Zorn, Claudia Linnhoff-Popien

1 Multi-Armed Bandit

Imagine you are in a casino facing multiple slot machines and each is configured with an unknown probability of how likely you can get a reward at one play. The question is: What is the best strategy to achieve highest long-term rewards, assuming you can play as long as you want?



More formally: Let $\mathcal{A} = \{1, \dots, K\}$ for $K \in \mathbb{N}$ be a set of actions. Let $\mathcal{R} : \mathcal{A} \rightarrow \mathbb{R}$ be a probabilistic reward function. A Bernoulli multi-armed bandit of K machines with respective reward probabilities $\sigma_1, \dots, \sigma_K$ can be described as a tuple $(\mathcal{A}, \mathcal{R})$ where the value of action $a \in \mathcal{A}$ is the expected reward of pulling the lever on machine a described by $Q(a) = \mathbb{E}[r_t | a_t = a] = \sigma_a$. We observe a reward r in a stochastic fashion. At the time step t , $r_t = \mathcal{R}(a_t)$ may return reward 1 with probability $Q(a_t)$ or 0 otherwise.¹

The goal is to *maximize* the cumulative reward $\sum_{t=1}^T r_t$. To make things easier, we approximate the true action probabilities Q as

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{t=1}^T r_t \mathbf{1}[a_t = a],$$

¹taken from: <https://lilianweng.github.io/posts/2018-01-23-multi-armed-bandit>

where $\mathbf{1}[\cdot]$ is the binary indicator function and $N_t(a)$ is the count how many times the action a has been selected so far. Consider the following three (agent) strategies for estimating the optimal action \hat{a}_t^* at time step t :

$$A : \quad \hat{a}_t^* = \arg \max_{a \in \mathcal{A}} \hat{Q}_t(a),$$

$$B : \quad \hat{a}_t^* = \begin{cases} a \sim \mathcal{A} & \text{if } R \leq \varepsilon \text{ for some } \varepsilon \in [0; 1] \text{ and } R \sim [0; 1], \\ \arg \max_{a \in \mathcal{A}} \hat{Q}_t(a) & \text{otherwise,} \end{cases}$$

$$C : \quad \hat{a}_t^* = \arg \max_{a \in \mathcal{A}} \hat{Q}_t(a) + \hat{U}_t(a) \quad \text{where} \quad \hat{U}_t(a) = \sqrt{\frac{2 \log t}{N_t(a)}}$$

Read, understand, and explain, in your own words, what ideas each of the estimations of A , B , and C employ as their strategies. In which cases would each of these strategies perform the best? How would each of these strategies be categorized in their respective utility for *exploration* or *exploitation* of (known) action probabilities? (You can, of course, consult the source linked above for more information.)

Summary:

- Shown above are the basics of *Q-Learning*, where we estimate the best action to draw next over a distribution of reward/payoff probabilities (the Q-values). Since the true expected reward for machine (bandit) i , ($\mathbb{E}[r|a] = \theta_i$), is hidden/unknown we estimate the probability with a $\hat{Q}(a)$ for any action a .
- In the estimated $\hat{Q}(a)$ shown here, we simply ‘remember’ all rewards we have received from each bandit so far (and count along how often we have ‘pulled each lever’/taken each action).
- The three strategies listed here, A, B, C represent the *greedy-selection*, ε -*greedy-selection* and *UCB1* (‘uncertainty bound 1’) strategies respectively.
- *Greedy-selection* A will always take the highest recorded reward/action pair. Depending on where we start (or where we receive the first reward), we will **always**, blindly continue to choose that action. Zero Exploration, Maximum Exploitation (most certainly bad one). Best in rare case where we luckily score on the best bandit in the very first attempt and can exploit advantage from the bandit.
- ε -*greedy-selection* on the other hand will choose the currently best estimated Q-value in **almost** all cases, but with a random chance of ε the agent draws a random action instead. Exploration with probability ε , Exploitation with probability $(1-\varepsilon)$. Best for simple but robust ‘naïve’ initial strategy.
- Strategy C also takes the greedy approach, but instead of the pure reward / action counts so far, the Q-value is augmented with a uncertainty bound U . The uncertainty bound function is dependent on both the time steps taken t and the count of the action. If the count stays 1, the function will increase with time (making it more likely to be selected) but converges faster with higher action count (i.e., the more often the action is taken, the smaller the uncertainty-bound for the Q-value). Encourages Exploration for lesser taken actions, Focuses on greedy Exploitation for well observed action results. Better than A, B as elegant initial strategy.

2 Scientific Reading: Inverse Reinforcement Learning

With reading and understanding formal definitions (and their papers) as an integral part of the lecture, let's test that ability by reading and understanding the following (correct) explanation of the novel technique *inverse reinforcement learning*, which we did not discuss in the lecture, and answer the questions afterwards.

CoIn Team:

Give a full-page explanation of inverse reinforcement learning suitable for computer science graduate students and include historical remarks and mathematical definitions.

ChatGPT Assistant:

Inverse Reinforcement Learning (IRL) is a subfield of Reinforcement Learning (RL) that aims to infer a reward function from observed behavior of an expert in an environment. The idea is to learn a reward function that would have generated the observed behavior, rather than learning a policy directly from interaction with the environment. The history of IRL can be traced back to the 1990s when the concept of reward-based learning was first introduced in the field of RL. However, the field of IRL started gaining traction in the early 2000s with the introduction of algorithms such as Maximum Entropy IRL and Bayesian IRL. These algorithms made it possible to learn reward functions from observed behavior, and they paved the way for further development in the field of IRL.

IRL is motivated by the observation that in many real-world RL problems, it is difficult to specify a reward function that accurately captures the desired behavior. For example, in a robotics problem, it may be difficult to specify a reward function that encourages a robot to move smoothly and avoid obstacles. By observing the behavior of an expert in the environment, IRL allows for the inference of a reward function that

would have generated the expert's behavior.

In mathematical terms, let π_E be the policy of the expert and π_θ be a policy parameterized by θ that is learned from interaction with the environment. The goal of IRL is to find the reward function, R , that best explains the observed behavior of the expert, π_E :

$$R^* = \arg \min_R D(\pi_E, \pi_\theta(R))$$

where D is a distance metric that measures the difference between the expert's behavior and the behavior of the learned policy. The learned reward function can then be used as a guide for learning a policy that mimics the expert's behavior.

There are several algorithms for IRL, including Maximum Entropy IRL, Bayesian IRL, and Gradient-Based IRL. Each of these algorithms has its own strengths and weaknesses, and the choice of algorithm will depend on the specific problem being solved.

Maximum Entropy IRL is a popular approach that seeks to find the reward function that maximizes the entropy of the policy that is optimized with respect to the reward function. This approach has been shown to be effective in a variety of domains, including robotics and gaming.

Bayesian IRL is another popular approach that uses Bayesian inference to learn a reward function from observed behavior. This approach has the advantage of allowing for uncertainty in the learned re-

ward function, which can be useful in problems where the expert's behavior is ambiguous.

Gradient-Based IRL is an approach that uses gradient-based optimization to learn a reward function from observed behavior. This approach has the advantage of being computationally efficient, and it has been used to learn reward functions in large and complex environments.

IRL has been applied in a variety of domains, including robotics, computer vision, and gaming, and it has the potential to make it easier to apply RL to real-world problems by allowing the use of human-defined reward functions. It is an active area of research with many open questions and opportunities for further development.

In conclusion, IRL is a promising sub-field of RL that offers new and innovative ways to learn reward functions from observed behavior. It has the potential to greatly expand the scope of RL by making it easier to apply to real-world problems,

and it is an area of ongoing research and development.

CoIn Team:

What basic exam question could you ask about inverse reinforcement learning?

ChatGPT Assistant:

A basic exam question about Inverse Reinforcement Learning could be:

What is Inverse Reinforcement Learning and why is it used in Reinforcement Learning?

[...]

CoIn Team:

Alright, let's talk about squirrels...

ChatGPT Assistant:

Sure, I'd be happy to provide more information about squirrels! What specific aspect of squirrels would you like to know more about? Their behavior, habitat, biology, or something else?

(i) Answer the question suggested by ChatGPT: What is Inverse Reinforcement Learning and why is it used in Reinforcement Learning?

As ChatGPT elaborates:

This question tests a student's understanding of the basic concept of IRL and its motivation. The answer should reflect an understanding of the definition of IRL and its purpose, which is **to infer a reward function from observed behavior of an expert in an environment**. The answer should also briefly mention **the limitations of specifying a reward function in RL problems and the potential benefits of using IRL to address these limitations**.

(ii) In the mathematical formula for the optimal reward function R^* , ChatGPT introduces an expert policy π_E that can be used with the type signature we are used to, i.e., $\pi_E : \mathcal{O} \rightarrow \mathcal{A}$. It also introduces a trained policy π_θ that is defined by a (network) parameter vector $\theta \in \Theta$ and is parametrized on a given reward function R that it is trained on, i.e., $\pi_\theta : \mathcal{R} \rightarrow \mathcal{O} \rightarrow \mathcal{A}$ where \mathcal{R} is the space of all possible reward functions.

Why does the expert policy π_E not need parameters (like θ)? Given a set of training observations \mathbb{O} , how can the behavior of two policies (like π_E and π_θ) be compared? Describe a possible implementation for a function D as discussed in the text.

Hint: You do not need to give a full definition for D but just sketch a possible approach.

The expert policy π_E is not given via an algorithm (with possible parameters) but via a data set $\mathbb{D} \subseteq \mathcal{O} \times \mathcal{A}$ of expert actions $a \in \mathcal{A}$ given observations $o \in \mathcal{O}$ with $(o, a) \in \mathbb{D}$.

We could define a distance function D on two policies, e.g., by counting the times they suggest different actions given the same observations. (As a concrete suggestion, for some action similarity relation \approx , let

$$D(\pi, \pi') = \sum_{o \in \mathbb{O}} \begin{cases} 0 & \text{if } \pi(o) \approx \pi'(o), \\ 1 & \text{otherwise,} \end{cases}$$

if it is possible to iterate through all of \mathbb{O} efficiently.)

(iii) In the text it is briefly mentioned that Gradient-Based IRL might be computationally more efficient compared to other variants of IRL. Give a reasonable explanation for this statement.

There are several learning and optimization techniques (cf. gradient descent, e.g.) that can use (helpful) gradient information for more efficient computation by gaining a rough estimate on the right direction in the search space to look into next. (This is reflected in gradient-based goals sitting in a higher level of the goal hierarchy introduced in the lecture. ;D)

3 Running Example: Vacuum World (reinforcement learning)

With the implementation of the *Vacuum World* example from Exercise 2, we now extend the code to act as a reinforcement learning environment. However, instead of two rooms to clean, let's make the environment more challenging to learn for the agent by

extending it to a N -room Vacuum World environment (with $N \gg 2$).

(i) How would we need to change our formal definition to accommodate N rooms? Since we will be implementing this change, it may be helpful to think about input and output formats suitable for a coded agent in advance!

We have previously defined the MDP tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, P, R)$ as follows (cf. Exercise 05):

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

For the N -room MDP tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, P, R)$ we could define, e.g.;

$$\begin{aligned}\mathcal{S} &= (s_0, s_1) = [0; N-1] \subset \mathbb{N} \times \{\mathbb{B}^N\} \\ \mathcal{A} &= \{\text{move left}, \text{vacuum}, \text{move right}\}, \\ \mathcal{T} &= \mathbb{R}, \\ P(s, a, s') &= \begin{cases} 0 & \text{if } s \in \{(0, -)\} \text{ and } a = \text{move left}, \\ 0 & \text{if } s \in \{(N-1, -)\} \text{ and } a = \text{move right}, \\ 1 & \text{otherwise} \end{cases} \\ R(s, a, s') &= \begin{cases} +10 & \text{if } s_0 = x \text{ and } s_1[x] == 1 \text{ and } a = \text{vacuum}, \\ -1 & \text{if } s_0 = x \text{ and } s_1[x] == 0 \text{ and } a = \text{vacuum}, \\ 0 & \text{otherwise,} \end{cases} \end{aligned}$$

where $s_1[x]$ identifies the x -th digit (from the left) in the binary number s_1 .

(ii) Convert your previous Vacuum World code to match your new definition with N -rooms, e.g., with $N = 10$. Additionally, reformat your code to be `gym`-compliant, i.e., providing an environment class with matching `step()`, `render()` and `reset()` methods, similar to the `Rooms` environment of Exercise 5.3. Instead of running indefinitely, also terminate (and reset) the current episode at a maximum timestep $T = 50$.

See `vacuum_world_rl_solution.zip` for a solution example.

(iii) Transfer the `PolicyNetwork` agent from Exercise 5.3 to your adapted N -room Vacuum World and observe its performance.

See `vacuum_world_rl_solution.zip` for a solution example.

(iv) (Bonus) Additionally, you may implement (with the attached `sarsa.agent.py`) the tabular reinforcement learning agent *SARSA*.

For this, implement the method `update` with the parameters `state`, `action`, `reward`, `next_state`, `done` in which you update the Q -value for a given state-action pair entry `self.q_values[state][action]` according to the SARSA learning rule.

Hint: You get the Q -array for `state` by calling the method `self.q_table(state)` and a_{t+1} by calling the method `self.policy(next_state)` method).

The SARSA learning rule update for its internal Q -values is given by:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

See `sarsa_agent_solution.py` for a solution example. (In our solution we have added a `run()` method to the `SarsaLearner` to be uniform with the `PolicyAgent` from last exercise, but you could also directly use the agent without the helper methods.)