Assignment: Tabular Reinforcement Learning

Course: Reinforcement Learning, Master CS, Leiden University

Written by: Thomas Moerland

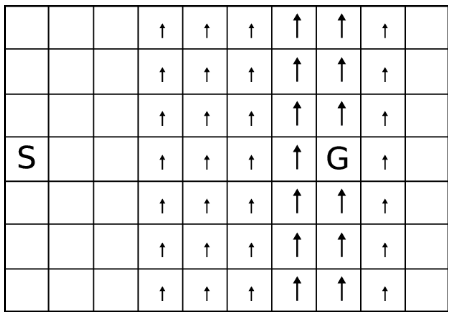
Research Question

In this assignment, you will study a range of basic principles in tabular, value-based reinforcement learning. They serve as a primer for the rest of the course. In particular, we will study the following topics:

- **Dynamic Programming** (DP) (Part 1):
  We first focus on dynamic programming, which is a bridging method between planning and reinforcement learning. DP assumes full access to a model of the environment, i.e., we can get $p(s'|s,a)$ and $r(s,a,s')$ for any state s and action a. DP is guaranteed to find the optimal solution, but it 1) requires a model (which is not always available) and 2) suffers from the curse of dimensionality (which all tabular methods actually do, and to which we get back later in the course).
- **Model-free RL**: We next switch to the reinforcement learning setting, where we do not have access to a model, but can only permanently execute actions from a state, and have to continue from the resulting next state.
  - **Exploration** (Part 2) The first issue this brings up is exploration versus exploitation: we need to sometimes try novel things, but at some point also exploit what we know works well. We will compare two simple ways to ensure exploration: ϵ-greedy and a softmax/Boltzmann policy.
  - **Back-up**: The second main aspect of any RL algorithm is the back-up. We acquired new information, and want to construct a new estimate of the value of a certain state-action pair s, a. There are two important considerations when constructing this back-up:
    * **Off-policy versus on-policy** (Part 3): This difference is best illustrated for one-step back-ups, for which we will compare Q-learning (off-policy) to SARSA (on-policy).
    * **Depth** (Part 4): We can also compute deeper back-ups, where we sum more rewards in a trace. We will compare 1-step back-ups, n-step back-ups, and Monte Carlo back-ups.

Environment

You will study these methods on the *Stochastic Windy Gridworld*, an adapted version of Example 6.5 (page 130) in *Reinforcement Learning: An Introduction* (second edition) by Sutton and Barto (see figure).



The environment consists of a 10x7 grid, where at each cell we can move up, down, left or right. We start at location (0,3) (we start indexing at 0, as is done in Python as well), indicated in the figure by 'S'. Our goal is to move to location (7,3), indicated by 'G'. However, a special feature of the environment is that there is a vertical wind. In columns 3, 4, 5 and 8, we are pushed one additional step up, while in columns 6 and 7, we move up two additional steps. The wind does not always blow, but is randomly present on 90% of the occasions (which makes the environment stochastic!). The reward of the agent at each step is -1, while reaching the goal gives a reward of +100, and terminates the episode.

Preparation

Python You need to install Python 3, the packages `Numpy`, `Matplotlib`, `SciPy` and an IDE of your choice.

Files You are provided with the following Python files:

- `Environment.py`: This file generates the environment. Run the file to see a demonstration of the environment with randomly selected actions. Inspect the class methods and make sure you understand them. With `render()` you can interactively visualize the environment during execution. If you provide `Q_sa` (a Q-value table), the environment will also display the Q-value estimates for each action in each state, while toggling `plot_optimal_policy` will also show arrows for the optimal policy. Play around with these settings, and make sure you understand them.

- `Dynamic_Programming.py`: This file contains placeholder classes and functions for your Dynamic Programming experiments (Part 1). Your goal is to complete these classes and functions.

- `Agent.py`: This file contains the Agent baseclass, in which you will implement the exploration methods (`select_action`) and you are provided with an `evaluate` method to run greedy evaluation episodes throughout training. The `update` method needs to be overwritten in each specific back-up method (files below).

- `Q-learning.py`: This file contains placeholder classes and functions for your Q-learning implementation.
- `SARSA.py`: This file contains placeholder classes and functions for your SARSA implementation.
- `MonteCarlo.py`: This file contains placeholder classes and functions for your Monte Carlo RL implementation.
- `Nstep.py`: This file contains placeholder classes and functions for your n-step Q-learning implementation.
- `Experiments.py`: In this file you will write all your code for the reinforcement learning experiments (Part 2, 3 and 4).
- `Helper.py`: This file contains some helper classes for plotting and smoothing. You can choose to use them, but are of course free to write your own code for plotting and smoothing as well. Inspect the code and run the file to verify that your understand what the functions do.

Matplotlib rendering Depending on your local software setup and the way you run your code (e.g., from the command line, or within an IDE), you may need to change the Matplotlib backend to allow for interactive rendering. For example, when your code does not give interactive rendering in PyCharm, you may add the following two lines to the top of `Environment.py`:

```
import matplotlib
matplotlib.use('Qt5Agg') # or TkAgg
```

Depending on you own software setup, play around with the backend settings until you find the plot being interactively updated (or run it from the command line, outside of your IDE).

1 Dynamic Programming

You first study Dynamic Programming, in particular the Q-value iteration algorithm (Alg. 1). In this algorithm you sweep through all state-action pairs, each time updating the estimate of a state-action value based on the following equation:

$$Q(s,a) \leftarrow \sum_{s'} \Big[p(s'|s,a) \cdot \big(r(s,a,s') + \gamma \cdot \max_{a'} Q(s',a') \big) \Big] \tag{1}$$

In DP, you have access to the full model of the environment dynamics. Therefore, you may use the `StochasticWindyGridworld.model()` function in your experiments.

You proceed with the following steps:

a) **Implement**:

- Correctly complete the class `QValueIterationAgent()` in the file `DynamicProgramming.py`.
  - In `init()`, initialize a table with means $Q(s,a)$ to 0.
  - In `select_action()`, implement the greedy policy: $\pi(s) = \arg\max_a Q(s,a)$.
  - In `update()`, implement the Q-iteration update, shown in Eq. 1. Make sure you print the maximum absolute error after each full sweep (i.e., each time after you visited each state-action pair once).
- Correctly complete the function `Q_value_iteration()` in the file `DynamicProgramming.py`. This function should execute Q-value iteration, as shown in Algorithm 1. It first initializes an agent, and then sweeps through the state space, each time calling the model and then updating the agent, until convergence.

b) **Experiment**: Verify that your code works by running the file. You should see a visualization of all the Q-value estimates during execution of the algorithm. **Closely inspect the values, how they change, and how they converge**. (You may need to increase the value of `step_pause` to make plotting slower). Do you understand the final values in each cell, and can you interpret them?

Algorithm 1: Tabular Q-value iteration (Dynamic Programming)

```
Input: Threshold $\eta \in R^+$.
Result: The optimal value function $Q^\star(s,a)$ and/or associated optimal policy $\pi^\star(s)$.
Initialization: A state-action value table $\hat{Q}(s,a) = 0 \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$
repeat
    $\Delta \leftarrow 0$
    for each $s \in \mathcal{S}$ do
        for each $a \in \mathcal{A}$ do
            $x \leftarrow Q(s,a)$                                   /* Store current estimate */
            $\hat{Q}(s,a) \leftarrow \sum_{s'} \Big[ p(s'|s,a) \cdot \big( r(s,a,s') + \gamma \cdot \max_{a'} Q(s',a') \big) \Big]$   /* Eq. 1 */
            $\Delta \leftarrow \max(\Delta, |x - \hat{Q}(s,a)|)$          /* Update max error */
        end
    end
until $\Delta < \eta$;
$Q^\star(s,a) = \hat{Q}(s,a)$                     /* Converged at optimal value function */
$\pi^\star(s) = \arg\max_a Q^\star(s,a) \quad \forall s \in \mathcal{S}$           /* Optimal policy is greedy */
Return $Q^\star(s,a)$ and/or $\pi^\star(s)$.
```

2 Exploration

You will now switch to the (model-free) reinforcement learning setting. In this case, you no longer have access to the model (like the real world, where executing an action permanently brings you to the next state). You therefore no longer have access to the `StochasticWindyGridworld.model()` function, and you can no longer sweep through all states. **Instead, you have to move forward from the current state, where you use the `StochasticWindyGridworld.step()` function.**

Since you cannot sweep through all states anymore, you will proceed in episodes from the start state. Compared to sweeping through the state space, you are now no longer guaranteed to visit all states under a greedy policy. You therefore need to introduce *exploration* into your action selection, to sometimes try something novel.

1. This first crucial step of any RL algorithm is the **action selection**. You decide to compare two types of policies:

   - The **ϵ-greedy policy**:

   $$\pi(a|s) = \begin{cases} 1.0 - \epsilon \cdot \frac{|\mathcal{A}|-1}{|\mathcal{A}|}, & \text{if } a = \arg\max_{b \in \mathcal{A}} \hat{Q}(s,b) \\ \epsilon/(|\mathcal{A}|), & \text{otherwise} \end{cases} \tag{2}$$

     In words, we select with small probability ϵ a random action, which ensures exploration, and otherwise take the greedy action. The parameter ϵ allows you to scale the amount of exploration ($\epsilon = 0$ gives a greedy policy, $\epsilon = 1$ gives a uniform/random policy).

   - The **Boltzmann policy**:

   $$\pi(a|s) = \frac{e^{\hat{Q}(s,a)/\tau}}{\sum_{b \in \mathcal{A}} e^{\hat{Q}(s,b)/\tau}} \tag{3}$$

     where $\tau \in (0, \infty)$ denotes a temperature parameter. This approach gives a higher probability to actions with a higher current value estimate, but still ensures exploration of other actions than the greedy one. The temperature τ allows you to scale the amount of exploration: for $\tau \to \infty$ the policy becomes uniform/random (why?), and for $\tau \to 0$ the policy becomes greedy.

2. The second crucial step of an RL algorithm is the **update**. After executing an action, the environment gives you new data, in the form of the observed reward and next state. Therefore, after timestep t you have observed data $\langle s_t, a_t, r_t, s_{t+1} \rangle$. In the next part of the assignment we will compare different ways to use this data to compute a new estimate for the state-action value at s_t, a_t, but in this assignment we will use the 1-step Q-learning update. We first compute the new **back-up estimate/target** G_t as

   $$G_t = r_t + \gamma \cdot \max_{a'} \hat{Q}(s_{t+1}, a') \tag{4}$$

   and then apply the **tabular learning update**

   $$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)] \tag{5}$$

   where $\alpha \in (0, 1]$ denotes the learning rate. For $\alpha \to 0$ learning is slow but stable, while $\alpha \to 1$ makes learning fast but less stable. The optimal learning rate typically lies somewhere in between. The learning rate is an important parameter in any learning experiment, and typically needs to be tuned extensively.

You proceed with your experiments as follows:

a) **Implement**:

- Correctly complete the class `Agent()` in the file `Agent.py`.
  - In `init()`, initialize a table with means $Q(s, a)$ to 0.
  - In `select_action()`, implement the above ϵ-greedy policy and softmax policy. Note: we already provided `argmax()` and `softmax()` functions for you in `Helper.py`, which are already imported into the file.
- Correct complete the class `QLearningAgent()` in the file `Q_learning.py`.
  - In `update()`, implement the Q-learning update shown above.
- Correctly complete the function `q_learning()` in the file `Q_learning.py`. This function should execute Q-learning, as shown in Algorithm 2. Make sure you run independent evaluation episodes (through the `evaluate()` method of the base agent) after every `eval_interval` steps. The function should return the mean return of the greedy policy at these evaluation moments, and the timesteps of evaluation.
- Verify that your code works by running `Q_learning.py`. Observe how the agent explores and learns. Plots the value estimates during execution, and observe how they change.

b) **Experiment**: You decide to perform a more systematic experiment, comparing ϵ-greedy and Boltzmann policies with different settings for the exploration parameters (respectively ϵ and temperature parameter τ).

- Write your experiment code in `Experiment.py`, using the `q_learning()` function you wrote above.
- Try ϵ-greedy with `epsilon = [0.03,0.1,0.3]` and softmax with `temps = [0.01,0.1,1.0]`.
- **Use runs of `n_timesteps = 50001` where you evaluate every `eval_interval = 1000`. For each setting, average the results over 20 repetitions. Smooth your learning curves if necessary**. Plot the learning curves for each setting in the same graph. Add a clear legend!

Algorithm 2: Tabular Q-learning.

```
Input: Exploration parameter, learning rate $\alpha \in (0,1]$, discount parameter $\gamma \in [0,1]$,
       total budget.
$\hat{Q}(s,a) \leftarrow 0, \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$.              /* Initialize Q-value table */
$s \sim p_0(s)$                                                  /* Sample initial state */
while budget do
    $a \sim \pi(a|s)$                          /* Sample action, e.g., $\epsilon$-greedy, softmax */
    $r, s' \sim p(r, s'|s,a)$                                    /* Simulate environment */
    $\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)]$      /* Q update */
    if $s'$ is terminal then
        $s \sim p_0(s)$                                          /* Reset environment */
    else
        $s \leftarrow s'$
    end
end
Return: $\hat{Q}(s,a)$
```

3 Back-up: On-policy versus off-policy target

The second important part of any RL algorithm is the way we back-up information. A major distinction is between off-policy back-ups (like Q-learning) and on-policy back-ups (like SARSA). We will first focus on the one-step case. The back-up equation for Q-learning was already implemented in the previous assignment:

$$G_t = r_t + \gamma \cdot \max_{a'} \hat{Q}(s_{t+1}, a') \tag{6}$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)] \tag{7}$$

The back-up equation for SARSA, given observations $\langle s_t, a_t, r_t, s_{t+1}, a_{t+1} \rangle$ is

$$G_t = r_t + \gamma \cdot \hat{Q}(s_{t+1}, a_{t+1}) \tag{8}$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)] \tag{9}$$

The major difference between these two is the value they bootstrap at the next state. Q-learning plugs in the value of the *best possible action at the next state*, and thereby attempts to learn the value of the optimal policy. SARSA backs up the value of the action we actually take (which may be exploratory and not the one with the currently optimal estimate). SARSA therefore learns the value function of the policy we actually execute (which includes exploration). Both approaches have their benefits and problems (see the textbook). Look closely at both above equations to understand this difference.

- **Implement**:
  - Correctly complete the class `SarsaAgent()` in the file `SARSA.py`.
    - In `update()`, implement the SARSA update shown above.
  - Correctly complete the function `sarsa()` in the file `SARSA.py`. This function should execute SARSA, as shown in Algorithm 3. Make sure you run independent evaluation episodes (through the `evaluate()` method of the base agent) after every `eval_interval` steps. The function should return the mean return of the greedy policy at these evaluation moments, and the timesteps of evaluation.
  - Run `SARSA.py` to verify that your implementation works. Plots the value estimates during execution, and observe how they change.
- **Experiment**: You decide to perform a more systematic experiment, where you compare Q-learning and SARSA for different learning rates.
  - Write your experiment code in `Experiment.py`, using the `q_learning()` and `sarsa()` functions you wrote above.
  - Try both methods for `learning_rates = [0.03,0.1,0.3]`.
  - **For each setting, average your results over 20 repetitions. Smooth your learning curves if necessary**. Plot the learning curves for all settings (Q-learning and SARSA for each of the above learning rates) in the same graph. Add a clear legend!

Algorithm 3: Tabular SARSA.

```
Input: Exploration parameter, learning rate $\alpha \in (0,1]$, discount parameter $\gamma \in [0,1]$,
       total budget.
$\hat{Q}(s,a) \leftarrow 0, \forall s \in \mathcal{S}, a \in \mathcal{A}$.        /* Initialize Q-value table */
$s \sim p_0(s)$                                                /* Sample initial state */
$a \sim \pi(a|s)$                  /* Sample action, e.g., $\epsilon$-greedy or softmax */
while budget do
    $r, s' \sim p(r, s'|s, a)$                                 /* Simulate environment */
    $a' \sim \pi(a'|s')$                          /* Sample action, e.g., $\epsilon$-greedy */
    $\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha \cdot [r + \gamma \cdot \hat{Q}(s',a') - \hat{Q}(s,a)]$          /* SARSA */
    if $s'$ is terminal then
        $s \sim p_0(s)$                                        /* Reset environment */
        $a \sim \pi(a|s)$
    else
        $s \leftarrow s'$
        $a \leftarrow a'$
    end
end
Return: $\hat{Q}(s,a)$
```

4 Back-up: Depth of target

The other important aspect of the back-up is its depth. So far, we have only looked at 1-step method, which directly bootstrap a value estimate after one transition. Instead, we can also sum multiple rewards in a trace before we bootstrap, which leads to *n-step methods* (n-step Q-learning or n-step SARSA, depending on the way you bootstrap). You will use n-step Q-learning, which computes the following target:

$$G_t = \sum_{i=0}^{n-1} (\gamma)^i \cdot r_{t+i} + (\gamma)^n \max_a Q(s_{t+n}, a) \tag{10}$$

and again standard tabular update

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)] \tag{11}$$

Not that, although it is called n-step Q-learning (due to the maximization over the last action), it is not a full off-policy method, since the first n reward are of course sampled from the current policy (and the target therefore mostly follows our behavioral policy).

On the other extreme, we can also omit bootstrapping altogether, and simply sum all rewards up to the end of the episode (or up to some maximum timestep after which we terminate the episode). The gives a *Monte Carlo update*:

$$G_t = \sum_{i=0}^{\infty} (\gamma)^i \cdot r_{t+i} \tag{12}$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)] \tag{13}$$

You will run experiments to compare different depths of the back-up target, from one-step up to Monte Carlo targets.

- **Implement**:
  - Correctly complete the class `NstepQLearningAgent()` in the file `Nstep.py`.
    - In `update()`, implement the n-step Q-learning update for each state-action pair in `states` and `actions`.
  - Correctly complete the function `n_step_Q()` in the file `Nstep.py`. This function should execute n-step Q-learning, as shown in Algorithm 4. Make sure you run independent evaluation episodes (through the `evaluate()` method of the base agent) after every `eval_interval` steps. The function should return the mean return of the greedy policy at these evaluation moments, and the timesteps of evaluation.
  - Run `Nstep.py` to verify that your method works. Observe the agent learning.
  - Correctly complete the class `MonteCarloAgent()` in the file `MonteCarlo.py`.
    - In `update()`, implement the Monte Carlo update for each state-action pair in `states` and `actions`.
  - Correctly complete the function `monte_carlo()` in the file `MonteCarlo.py`. This function should execute Monte Carlo RL as shown in Algorithm 5. Make sure you run independent evaluation episodes (through the `evaluate()` method of the base agent) after every `eval_interval` steps. The function should return the mean return of the greedy policy at these evaluation moments, and the timesteps of evaluation.

Run `MonteCarlo.py` to verify that your method works. Observe the agent learning, while you plot the value estimates and optimal policy. Do you see a difference with the previous RL methods?

- **Experiment**: You decide to perform a more systematic experiment, comparing different back-up depths.
  - Write your experiment code in `Experiment.py`, using the `n_step_Q()` and `monte_carlo()` functions you wrote above.
  - For `n_step_Q()`, try different back-up depths: `ns = [1,3,10]`. Also run the Monte Carlo method.
  - Plot the learning curves for each setting in the same graph. **For each setting, average your results over 20 repetitions. Smooth your learning curves if necessary**. Add a clear legend!

Algorithm 4: Tabular n-step Q-learning

```
Input: Exploration parameter $\epsilon \in (0,1]$, learning rate $\alpha \in (0,1]$, discount parameter
       $\gamma \in [0,1]$, maximum episode length $T$, target depth $n$.
$\hat{Q}(s,a) \leftarrow 0, \forall s \in \mathcal{S}, a \in \mathcal{A}$.                /* Initialize Q-value table */
while budget do
    $s_0 \sim p_0(s)$                                            /* Sample initial state */
    /// Collect episode
    for $t = 0...(T-1)$ do
        $a_t \sim \pi(a|s_t)$                              /* Sample action, e.g., $\epsilon$-greedy */
        $r_t, s_{t+1} \sim p(r, s'|s_t, a_t)$                         /* Simulate environment */
        if $s_{t+1}$ is terminal then
            break                                            /* Episode terminates */
        end
    end
    $T_{ep} \leftarrow t + 1$                                 /* $T_{ep}$ stores episode length */
    /// Compute n-step targets and update
    for $t = 0...(T_{ep}-1)$ do
        $m = \min(n, T_{ep} - t)$                 /* $m$ is number of rewards left to sum */
        if $s_{t+m}$ is terminal then
            $G_t \leftarrow \sum_{i=0}^{m-1} (\gamma)^i \cdot r_{t+i}$             /* n-step target without bootstrap */
        else
            $G_t \leftarrow \sum_{i=0}^{m-1} (\gamma)^i \cdot r_{t+i} + (\gamma)^m \cdot \max_a \hat{Q}(s_{t+m}, a)$        /* n-step target */
        end
        $\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)]$              /* Update Q-table */
    end
end
Return: $\hat{Q}(s,a)$
```

Algorithm 5: Tabular Monte Carlo reinforcement learning.

```
Input: Exploration parameter $\epsilon \in (0,1]$, learning rate $\alpha \in (0,1]$, discount parameter
       $\gamma \in [0,1]$, maximum episode length $T$.
$\hat{Q}(s,a) \leftarrow 0, \forall s \in \mathcal{S}, a \in \mathcal{A}$.          /* Initialize Q-value table */
while budget do
    $s_0 \sim p_0(s)$                                  /* Sample initial state */
    for $t = 0...(T-1)$ do
        $a_t \sim \pi(a|s_t)$                      /* Sample action, e.g., $\epsilon$-greedy */
        $r_t, s_{t+1} \sim p(r, s'|s_t, a_t)$                /* Simulate environment */
        if $s_{t+1}$ is terminal then
            break                                     /* Episode terminates */
        end
    end
    $G_{t+1} \leftarrow 0$                         /* Start reward summation from 0 */
    for $i = t...0$ do
        $G_i \leftarrow r_i + \gamma \cdot G_{i+1}$          /* Compute Monte Carlo target at each step */
        $\hat{Q}(s_i, a_i) \leftarrow \hat{Q}(s_i, a_i) + \alpha \cdot [G_i - \hat{Q}(s_i, a_i)]$           /* Update Q-table */
    end
end
Return: $\hat{Q}(s,a)$
```