Assignment: Model-based reinforcement learning
Course: Reinforcement Learning, Leiden University
Written by: Thomas Moerland

Research Question

In this assignment, you will study two model-based reinforcement learning (MBRL) algorithms:

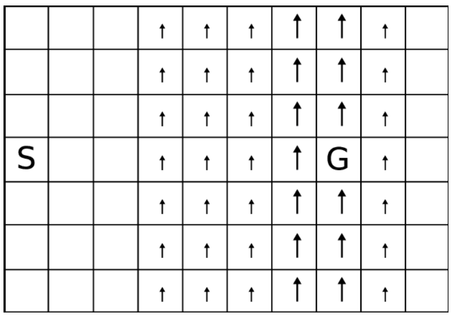
- Dyna (Sutton and Barto, Sec. 8.2) with ϵ-greedy exploration.
- Prioritized sweeping (Sutton and Barto, Sec. 8.4) with ϵ-greedy exploration.

Your goal is to investigate these two algorithms. In particular, you will:

1. Implement these algorithms.
2. For each algorithm investigate the effect of:
   - `n_planning_iterations`: the number of planning iterations each algorithm makes in between a real environment step.
   - `wind_proportion`: the proportion of times the wind blows, i.e., the amount of stochasticity in the environment (further detailed below).

Environment

You will use a variant of the *Windy Gridworld* environment for your studies, based on Example 6.5 (page 130) of Sutton and Barto.



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’.

Transition function A special feature of the environment is that there is a vertical wind. In columns 3, 4, 5 and 8, the wind pushes us up one grid cell per timestep, while in columns 6 and 7, the wind pushes us up two grid cells per timestep. Importantly, **the proportion of times the wind blows can be varied upon environment initialisation**, through the `wind_proportion` parameter. By default, the wind blows on 90% of occasions, which makes the environment transition function *stochastic*: most of the timesteps the agent experiences the effect of the wind, but 10% of times it's actually windstill.

Reward function Reaching the goal 'G' gives a reward of +100, and terminates the episode. Every other step has a default reward of -1.

Preparation

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

Files You are provided with four Python files:

- `MBRLEnvironment.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.
- `MBRLAgents.py`: This file contains placeholder classes for your two MBRL algorithms: `DynaAgent`, and `PrioritizedSweepingAgent`. Currently, each method randomly selects and action, and does not perform any updates. Your goal is to implement the correct `init()`, `select_action()`, and `update()` methods for each class. Run the file and verify that they work for random action selection.
- `MBRLExperiment.py`: In this file you will write your experiment code.
- `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 they do.

Handing in You need to hand in (see the Brightspace schedule for the assignment deadline):

- A **report** (pdf). See `https://irl.liacs.nl/assignments` for further details. Be sure your report:
  - Describes your methods (include equations).
  - Shows results (figures).
  - Interprets your results.
- All **code** to replicate your results. Your submission should contain:
  - The original `MBRLEnvironment.py` and `Helper.py`
  - Your modified `MBRLAgents.py`.

  - Your modified `MBRLExperiment.py`, which upon execution should produce all your plots, and save these to the current folder.

Be sure to verify that your code runs from the command line, and does not give errors!

Remember

- Average your results over repetitions (since each runs is stochastic)!
- In each repetition, really start from scratch, i.e., randomly initialize a new environment, and initialize your policy from scratch.
- Do not fix any seeds within the loop over your repetitions! Each repetition should really be an independent repetition.
- If necessary, apply additional smoothing to your curves to make them interpretable.

1 Dyna

You first decide to study the Dyna algorithm. You proceed in four steps:

a Correctly complete the class `DynaAgent()` in the file `MBRLAgents.py`.

- In `init()`, initialize the means $Q(s,a)$, transition counts $n(s,a,s')$ and reward sums $R_{sum}(s,a,s')$ for each action to 0.
- In `select_action()`, implement the ϵ-greedy policy.
- Most work needs to happen in the `update()` function. Full pseudocode is available in the book, which is repeated below with more detail:

Algorithm 1: Dyna (with ϵ-greedy exploration) pseudo-code.

```
Input: Exploration parameters $\epsilon$, number of planning updates $K$, learning rate $\alpha$,
       discount parameter $\gamma$, maximum number of timesteps $T$
Initialization: Initialize $Q(s,a) = 0$, $n(s,a,s') = 0$, $R_{sum}(s,a,s') = 0 \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$.
for $t = 1...T$ do
    $s \leftarrow$ current state                          /* Reset when environment terminates */
    $a \sim \pi_{\epsilon\text{-greedy}}(a|s)$                                  /* Sample action */
    $r, s' \sim p(r, s'|s,a)$                              /* Simulate environment */
    $\hat{p}(s', r|s,a) \leftarrow \text{Update}(s,a,r,s')$                     /* Update model (Alg.2) */
    $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)]$     /* Update Q-table */
    repeat $K$ times
        $s \leftarrow$ random previously observed state          /* Find state with $n(s) > 0$ */
        $a \leftarrow$ previously taken action in state $s$     /* Find action with $n(s,a) > 0$ */
        $s', r \sim \hat{p}(s', r|s,a)$                                /* Simulate model */
        $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)]$  /* Update Q-table */
    end
end
```

Algorithm 2: Update model $\hat{p}(s', r|s,a)$, split up as a tabular dynamics model $\hat{p}(s'|s,a)$ and deterministic reward function $\hat{r}(s,a,s')$.

```
$n(s,a,s') \leftarrow n(s,a,s') + 1$                              /* Update transition counts */
$R_{sum}(s,a,s') \leftarrow R_{sum}(s,a,s') + r$                         /* Update reward sums */
$\hat{p}(s'|s,a) = \frac{n(s,a,s')}{\sum_{s'} n(s,a,s')}$                         /* Estimate transition function */
$\hat{r}(s,a,s') = \frac{R_{sum}(s,a,s')}{n(s,a,s')}$                           /* Estimate reward function */
```

Verify that your code works by running `MBRLAgents.py`. You can manually execute every step by pressing 'Enter', or complete the full run by pressing 'c'. Look at how the agents learns, how the $Q(s,a)$ value estimates change, and how the optimal policy changes. Is it easy for the agent to learn?

b Write a function `run_repetitions()` in `MBRLExperiment.py`. Your function should repeatedly test the `DynaAgent()` on an instance of `WindyGridworld()`. **Switch plotting off for your repetitions, plotting will slow the run down a lot.**

Make sure to evaluate the agent after every `eval_interval=250` timesteps, by calling the `evaluate` method of the agent (already provided). This function runs greedy evaluation episodes to test what the learned value function at that point is at best capable off (you need to average over multiple test episodes since the environment is stochastic). Set the maximum episode length for your evaluations to `max_episode_length=100`.

- Run a few repetitions of appropriate length (until you see learning behaviour). Store the evaluations along training (i.e., you evaluate after every 250 steps).
- Average the learning curves over your repetitions, pick an appropriate smoothing window to smooth your curves, and plot the result.

Experiment a bit with varying your hyperparameters: `epsilon`, `learning_rate` and `n_planning_updates`. Get a feeling for how they affect your results.

c You decide to run a more structured experiment (which you code in `MBRL_Experiment.py`):

- Test the effect of `n_planning_updates`: `[1,3,5]`, for `epsilon=0.1` and `learning_rate=0.2`.
- Repeat the above experiment, but now set `wind_proportion=1.0` in the initialisation of the environment. This makes the environment deterministic, since the wind always blows.

Run your function from 1b for these different experiments, where you average over `n_rep=20` repetitions of `n_timesteps=10001` steps (we use `10001` so you automatically evaluate at beginning and end, i.e., at `t=0`, `t=250`, ..., `t=10000`: remember Python indexing starts at 0!). Store the performance at each of these steps in a learning curve, average these curves over independent repetitions, and additionally smooth your resulting curve in necessary by setting `smoothing_window` parameter in the function from the Helper file.

Make a nice plot for both of the above experiments: one for the stochastic environment, and one for the deterministic case, where you plot in each case the performance of Dyna for different planning budgets. Add a legend, and label the x and y-axis appropriately. You could use the `LearningCurvePlot()` class for this.

- **To each plot, add a model-free Q-learning baseline**. Hint: you don't need to write new code for this, you can simply set `n_planning_updates=0` in your Dyna code to recover Q-learning.

d Write the first section of your report. Describe:

- Your methodology (with equations).
- Your results (graphs).
- Interpret the results, give possible explanations.

2 Prioritized sweeping

You decide to repeat the above procedure with a different idea, prioritized sweeping, which also plans in the reverse direction to spread the information more quickly over the state space. You should largely be able to reuse your code from the previous experiment.

a Correctly complete the class `PrioritizedSweepingAgent()` in the file `MBRLAgents.py`.

- In `init()`, initialize the means $Q(s,a)$, transition counts $n(s,a,s')$ and reward sums $R_{sum}(s,a,s')$ for each action to 0, and initialize an empty priority queue PQ.
- In `select_action()`, implement the ϵ-greedy policy.
- Most work needs to happen in the `update()` function. Full pseudocode is available in the book, which is repeated below with more detail:

Algorithm 3: Prioritized sweeping (with ϵ-greedy exploration) pseudo-code.

```
Input: Exploration parameters $\epsilon$, number of planning updates $K$, learning rate $\alpha$,
       discount parameter $\gamma$, maximum number of timesteps $T$, priority threshold $\theta$.
Initialization: Initialize $Q(s,a) = 0$, $n(s,a,s') = 0$, $R_{sum}(s,a,s') = 0 \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$,
and prioritized queue $PQ$.
for $t = 1...T$ do
    $s \leftarrow$ current state                          /* Reset when environment terminates */
    $a \sim \pi_{\epsilon\text{-greedy}}(a|s)$                                  /* Sample action */
    $r, s' \sim p(r, s'|s,a)$                                /* Simulate environment */
    $\hat{p}(s', r|s,a) \leftarrow$ Update$(s,a,r,s')$                  /* Update model (Alg.2) */
    p $\leftarrow |r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)|$            /* Compute priority p */
    if p> $\theta$ then
        Insert $(s,a)$ into $PQ$ with priority p          /* State-action needs update */
    end
    /// Start sampling from $PQ$ to perform updates
    repeat $K$ times
        $s, a \leftarrow$ pop highest priority from $PQ$      /* Sample $PQ$, break when empty */
        $s', r \sim \hat{p}(s', r|s,a)$                                /* Simulate model */
        $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)]$   /* Update Q-table */
        /// Loop over all state action that may lead to state $s$
        for each $(\bar{s}, \bar{a})$ with $n(\bar{s}, \bar{a}, s) > 0$ do
            $\bar{r} = \hat{r}(\bar{s}, \bar{a}, s)$                            /* Get reward from model */
            p $\leftarrow |\bar{r} + \gamma \cdot \max_a Q(s,a) - Q(\bar{s}, \bar{a})|$         /* Compute priority p */
            if p> $\theta$ then
                Insert $(\bar{s}, \bar{a})$ into $PQ$ with priority p /* State-action needs update */
            end
        end
    end
end
```

Verify that your code works by running `MBRLAgents.py`. Look at how the agents learns, how the $Q(s,a)$ value estimates change, and how the optimal policy changes. Is it easy for the agent to learn?

b Modify `run_repetitions()` to work for your `PrioritizedSweepingAgent` as well. **Switch plotting off for your repetitions, plotting will slow the run down a lot**. Again, make sure to evaluate the agent every `eval_interval=250` timesteps.

c Run the same structured experiments as before (which you code in `MBRL_Experiment.py`):

- Test `n_planning_updates`: `[1,3,5]`, for `epsilon=0.1` and `learning_rate=0.2`.
- Repeat the above experiment, but now set `wind_proportion=1.0` in the initialisation of the environment. This make the environment deterministic, since the wind always blows. (One could also make the environment deterministic by setting `wind_proportion=0.0`, but this would make the environment very uninteresting – why?)

Make a nice plot for both of the above experiments: one for the stochastic environment, and one for the deterministic case, where you plot in each case the performance of Prioritized Sweeping for different planning budgets. Add a legend, and label the x and y-axis appropriately. You could use the `LearningCurvePlot()` class for this.

- **To each plot, add a model-free Q-learning baseline**. Hint: you don't need to write new code for this, you can simply set `n_planning_updates=0` in your Dyna code to recover Q-learning.

d Write a second section of your report. Describe:

- Your methodology (with equations).
- Your results (graphs).
- Interpret the results, give possible explanations.

3 Comparison

- Make a third graph, both for the deterministic and stochastic versions of the gridworld, where you compare 1) the best performing Dyna model, 2) the best performing Prioritized Sweeping model, and 3) the baseline model-free Q-learning model. Write a small comparison section where you interpret the comparison results, for both the stochastic and deterministic version.

- In addition, create a table where you compare the runtime of each algorithm (Q-learning, Dyna, Prioritized Sweeping), i.e., the average runtime needed for a complete learning instance (a single repetition of your experiment). Interpret your results, and relate them to the comparison graphs you just made.

4 Reflection

Write a dicussion section where you reflect on both algorithms:

- What could be a strength and weakness of model-based RL compared to model-free RL (like Q-learning)?
- How do you compare Dyna and Prioritized Sweeping? Which approach performed better? Which idea do you like better? Could both be combined?
- The assignment focused on new way to update the Q-fuction, but mostly ignored the exploration aspect of RL. You initialized all $Q(s, a)$ value estimates to 0, while the default reward of every step is -1. Do you think this choice has an effect on agent performance, e.g., when it comes to exploration? (Hint: think back of the methods in the bandit assignment. You could also try to set the default reward per step to 0 in the environment definition, and see whether agent performance changes. What could be the explanation?)