ASSIGNMENT 3: MODEL-BASED REINFORCEMENT LEARNING

Daniël Zee
s2063131

Noëlle Boer
s2505169

1 INTRODUCTION

For this assignment we studies two model-based reinforcement learning algorithms: Dyna and Prioritized sweeping. Each algorithm was implemented in Python and our implementations were used to study their performance of a finite Markov Decision Process.
In finite Markov Decision Processes, or MDPs, taking an action in a state will bring us in a new state with a certain reward. We define the state space $\mathcal{S}$ as the set of states we can be in and $\mathcal{A}$ as the set of actions that are possible. The dynamics of an MDP are therefore defined by a transition function $p(s'|s,a)$ and a reward function $r(s,a,s')$, where $a \in \mathcal{A}$ and $s \in \mathcal{S}$. These dynamics form the environment on which our algorithms are able to act. As we have seen in model-free methods, the transition and reward function of our environment are unknown to our agent. We can only irreversibly act on our environment to learn the Q-values, meaning that we are only able to learn from real experience. This process is called *direct reinforcement learning* (direct RL). The insight of model-based reinforcement learning is that we can also use our real experience with the environment to learn a model of the environment dynamics. The model is then used to simulate the environment and we can act on it to produce simulated experiences which can then also be used to update our Q-values. This process is called *planning*. The advantage of planning is that we have reversible access to the environment through our model which means we can more easily back-up changes to our Q-values to previously visited states as we can act on those states without the agent being there in the real environment. Planning can be added to one-step temporal-difference learning, also called TD(0), resulting in the following general steps:

1. Provide the policy π to be evaluated
2. Initialize $Q(s,a)$ and $\hat{p}(s',r|s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
3. Initialise the starting state s
4. Repeat for each timestep
   - 4.1. Sample an action a from $\pi(s,a)$
   - 4.2. Observe the reward r and next state s' after taking action a in state s in the environment
   - 4.3. Update $Q(s,a)$ based on r combined with the attractiveness of being in s'
   - 4.4. Update $Model(s,a)$ to reflect the possibility of receiving r and ending in s'
   - 4.5. Repeat for each planning update
     - 4.5.1. Choose some previously observed state s and action a taken in s.
     - 4.5.2. Observe the reward r and next state s' by sampling from $\hat{p}(s',r|s,a)$
     - 4.5.3. Update $Q(s,a)$ based on R combined with the attractiveness of being in s'

where $\hat{p}(s',r|s,a)$ is our learned model, split up as the estimated transition function $\hat{p}(s'|s,a)$ and reward function $\hat{r}(s,a,s')$. Steps 4.3, 4.4 and 4.5 implement direct RL, model-learning and planning respectively.

2 METHODOLOGY

We were provided with a custom environment, called the *Windy Gridworld* environment, based on Example 6.5 (page 130) of Sutton & Barto (2018). The environment consists of a 7x7 grid, where the agent can move to adjacent squares. The agent stays in the same square when it tries to move outside of the grid. The agent starts at location (0,3) and our goal is to move to location (7,3).

Figure 1 shows a visualisation of the environment. The arrows indicate a vertical wind which is present in the environment. In columns 3, 4, 5, and 8, the wind pushes the agent up one cell per timestep, while in column 6 and 7, the wind pushes the agent op two cells. The proportion of times the wind blows can be set upon environment initialisation through a parameter. By default this is set to 0.9, making the environment transition function stochastic. The reward function gives the agent a reward of +100 when it reaches the goal, ending the episode. Every other step gives a reward of -1. The agents for each algorithm were implemented in a class containing three methods corresponding to the general steps of a TD(0) algorithm with model-based planning for which the agent is responsible: The `__init__` method for the initialisation of the Q-values and the model (step 2), the `select_action` method for defining the policy function and using it to sample actions given a state (steps 1 and 4.1) and the `update` method for direct RL, model-learning and planning (steps 4.3, 4.4 and 4.5). To experiment on these agents, we wrote an experiment function that repeatedly tests the agent on an instance of the WindyGridworld environment. This function implements all the remaining algorithm steps which are independent of the used agent. The function runs the agent for the algorithm specified by the user for a specified number of timesteps and repetitions. Then after each interval of timesteps specified by the user we run multiple greedy evaluation episodes on the agent and average over them to see what the learned value function at that point is at best capable of. The greedy evaluation returns are then averaged over all the repetitions of the experiment function. We use these results to plot the learning curves of each algorithm on the default stochastic Windy Gridworld environment and on a deterministic version of the environment for different settings for the number of planning updates at each timestep. We compare all results with the learning curves for Q-learing to see the impact the planning process has compared to model-free learning.

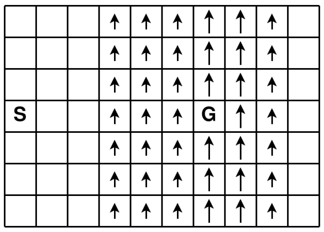

Figure 1: Visualisation of the WindyGridworld environment

3 Dyna

The Dyna algorithm uses a combination of direct RL, model-learning and planning, meaning that we use our real experience with the environment and our simulated experience using the model to update our Q-values. We have seen for TD(0) algorithms that there as multiple approaches for updating the Q-values. On-policy approaches learn the Q-values for the policy you are acting on while an off-policy approach like Q-learning tries to learn the Q-values for the optimal policy while acting on some other policy. Dyna simply adds the model-learning and planning steps to one of these existing TD(0) algorithms. During the planning step of Dyna, we randomly pick a state-action pair that previously has been experienced, sample a possible reward and next state from our model and use this simulated experience to update the Q-values using the same updating rule used for the real experience in the learning step. Because we are comparing our model-based algorithms to Q-learning and we are interested in maximizing the greedy evaluation returns, we will implement the Dyna algorithm using the Q-learning update function with ϵ-greedy action selection, resulting in the Dyna-Q algorithm.

3.1 Methodology

We implicitly derive our learned model $\hat{p}(s', r|s, a)$ by keeping track of of the transition counts $n(s, a, s')$, where we store the number of times we observed state s' after taking action a in state s and the reward sums $R_{sum}(s, a, s')$, where we store the total sum of rewards obtained for transition s, a, s'. We can then easily derive our estimated transition and reward function:

$$\hat{p}(s'|s,a) = \frac{n(s,a,s')}{\sum_{s'} n(s,a,s')} \qquad \hat{r}(s,a,s') = \frac{R_{sum}(s,a,s')}{n(s,a,s')} \tag{1}$$

For the Dyna agent we implemented the methods for the agent class as follows:

- `__init__`: Initialize $Q(s, a) = 0$, $n(s, a, s') = 0$, $R_{sum}(s, a, s') = 0 \quad \forall(s) \in \mathcal{S}, a \in \mathcal{A}$

- `select_action`: We implemented the following ϵ-greedy policy:

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

- `update`: We update the model:

$$n(s,a,s') \leftarrow n(s,a,s') + 1 \qquad R_{sum}(s,a,s') \leftarrow R_{sum}(s,a,s') + r \tag{3}$$

  We implemented the Q-learning update equation and update $Q(s,a)$ for the real experience:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{4}$$

  where α is the learning rate and γ is the discount factor. For every planning update we do the following:

$$s \leftarrow \text{random state with } n(s) > 0 \qquad a \leftarrow \text{random action with } n(s,a) > 0 \tag{5}$$

$$s', r \sim \hat{p}(s', r|s,a) \tag{6}$$

  followed by the Q-learning update in equation 8 for this simulated experience.

We ran our experiment function for this agent for 10 repetitions of 10001 timesteps with an evaluation interval of 250 timesteps. Every evaluation averages over 30 episodes with a maximum length of 100 timesteps. This was done for the following number of planning updates: $[1, 3, 5]$ on the WindyGridworld environment with wind proportions 0.9 and 1.0. $\epsilon = 0.1$, $\alpha = 0.2$ and $\gamma = 1.0$ were used for all experiments. All learning curves were smoothed with a smoothing window of 5.

3.2 Results

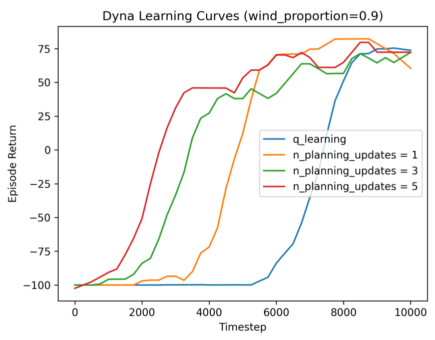

Figure 2: Greedy evaluation of the Q-values learned by Dyna on the stochastic WindyGridworld environment.

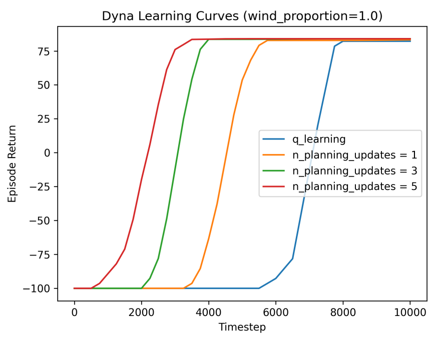

Figure 3: Greedy evaluation of the Q-values learned by Dyna on the deterministic WindyGridworld environment.

We can see that increasing the number of planning updates at each timestep results in better episode returns at earlier timesteps. This can be explained by the fact that without planning it taking a long time for the reward given by reaching the goal to propagate back to the states around the start of an episode. For example, after finishing the first episode, only the Q-value of the state-action pair which led to the goal will be updated to reflect the high reward. When we add planning however, we can use our model to propagate the rewards back to the earlier states using our simulated experience. While this does lead to a better value function at early timesteps, it is important to note that the Q-values here do yet reflect the real environment as our model is still only an incomplete approximation. When our model is wrong about the probability of certain transitions, it can move our Q-values in the wrong direction. This can be seen in figure 2 as the episode returns sometimes plateau or even decrease in quality, indicating that the simulated experiences do not represent the real environment dynamics well. This is however counteracted in the long run by the combination of the direct RL step, which moves the Q-values based on real experience, and the fact that our models keeps improving after every interaction with the environment. For the deterministic environment in figure 3 this is not a problem, as can be seen by the earlier convergence to the optimal episode returns by increasing the number of planning updates. This is because the transition function for every state-action pair can be fully learned by only a single visit from real experience, meaning that the model is never wrong about the probabilities of the transition function, leading to perfect updates to the Q-values compared to direct RL.

4 PRIORITIZED SWEEPING

The second algorithm we studied is called prioritized sweeping. The planning updates in the Dyna algorithms were performed on randomly selected state-action pairs that were previously visited from real experience. This uniform selection is usually not the best as not all state-action pairs require updating after each previous update, meaning that wasteful updates will be made. Prioritized sweeping addresses this by keeping track of the state-action pairs for which updating the Q-values will be most informative, which are usually the ones leading up to state for which a previous update resulted in a big change in the Q-value. To do this we do not update our Q-values using direct RL (meaning that we skip step 4.3 of the general process) but instead only calculate how much the Q-value would change with the newly gained real experience from the environment. If this difference is greater than a specified threshold, we add this state-action pair to a priority queue with the priority being the size of the change. In the planning step we then select the state-action pair with the highest priority to sample a reward and next state from our model from and use this simulated experience to update its Q-value, again using the Q-learning update rule. Before moving on to the next planning update we first use our model to find all state-action pairs that have previously led to the state we just updated, sample a reward from the model for each, and calculate how much their Q-values would change after an update. If this is again higher than the threshold these state-action pairs are also added to the queue. This results in a planning strategy that works backwards from previous influential updates.

4.1 METHODOLOGY

We derive our learned model $\hat{p}(s', r|s, a)$ in the same as for the Dyna algorithm shown in equation 1. The only addition is that we are now also interested in the *reverse model* during planning to find which states and action have previously led to a specific nest state:

$$\hat{p}(s, a|s') = \frac{n(s, a, s'}{\sum_{s,a} n(s, a, s')} \tag{7}$$

For the prioritized sweeping agent we implemented the method for the agent class as follows:

- `__init__`: 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 priority queue PQ.
- `select_action`: We implemented the ϵ-greedy policy as shown in equation 2.
- `update`: We update the model as shown in equation 3. We calculate the update priority as follows:

$$\text{p} \leftarrow |r + \gamma \max_{a'} Q(s', a') - Q(s, a)| \tag{8}$$

  and insert (s, a) into PQ with priority p if p $> \theta$, where θ is the priority threshold. For every planning update we do the following:

$$s, a \leftarrow \text{pop highest priority from PG} \tag{9}$$

$$s', r \sim \hat{p}(s', r|s, a) \tag{10}$$

  followed by the Q-learning update in equation 8 on this simulated experience. Before the next planning update we do the following for all $(\bar{s}, \bar{a})$ with $\hat{p}(\bar{s}, \bar{a}|s) > 0$:

$$\bar{r} = \hat{r}(\bar{s}, \bar{a}, s) \tag{11}$$

$$\text{p} \leftarrow |\bar{r} + \gamma \max_{a} Q(s, a) - Q(\bar{s}, \bar{a})| \tag{12}$$

  and insert $(\hat{s}, \hat{a})$ into PQ with priority p if p $> \theta$.

We ran the same experiments for this algorithm as for the Dyna algorithm using the same settings for both configurations of the WindyGridworld environment. $\theta = 0.01$ was used for all experiments.

4.2 Results

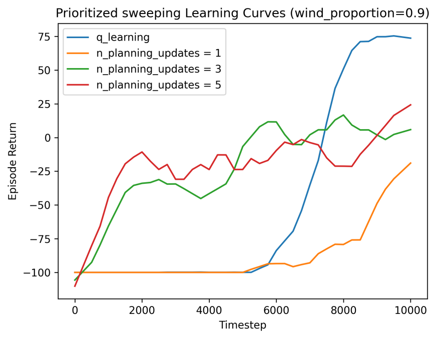

Figure 4: Greedy evaluation of the Q-values learned by prioritized sweeping on the stochastic WindyGridworld environment.

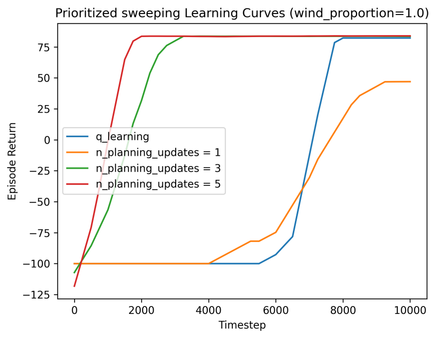

Figure 5: Greedy evaluation of the Q-values learned by prioritized sweeping on the deterministic WindyGridworld environment.

Looking at the results on the deterministic environment in figure 5 we again see that increasing the number of planning steps leads to better episode returns at earlier timesteps. An interesting observation however is that we also see an increase in the slope of the learning curves, which we did not observe for Dyna. With only one planning update Q-learning even seems to overtake prioritized sweeping in performance. This behaviour can be explained by the order in which the algorithms update their Q-values. Q-learning only uses direct RL, which results in the positive reward of reaching the goal first getting incorporated in the Q-values of the state-action pairs close to the goal and only after many successive episodes will those values reach the state-action pairs at the start, resulting in those Q-values being the last to be optimized. So when this does finally happen, most Q-values further along the path will already be close to their optimal values, meaning it takes relatively few timesteps to reach the optimal greedy evaluation return. Prioritized sweeping on the other hand only uses planning and updates the Q-values in order of the most influential updates. This means that after reaching the goal, the positive reward gets propagated back to the start much faster. The downside however is that while we find a greedy path to the goal more quickly, it takes more updates compared to Q-learning to reach the optimal path because of the exploitative nature of our ϵ-greedy action selection. Increasing the number of planning updates speeds this up, increasing the slope of the learning curves. For the stochastic environment in figure 4, prioritized sweeping does not come close to the performance of Q-learning, and increasing the number of planning update steps from 3 to 5 does not result in significantly better performance. This can again be explained by the fact that our model is only an incomplete approximation of the real environment and prioritized sweeping does not use direct RL, meaning that the Q-values we are learning also do not represent the real environment. Adding more planning updates will not change that fact and the only way to increase the performance is to improve the model by trying more actions in the real environment.

5 Comparison

Figures 6 and 7 show a comparison between the best performing models for both algorithms and the Q-leaning baseline for the stochastic and deterministic versions of the WindyGridworld environment respectively. The best performing models were defined as the learning curves with the greatest area under the curve. We see that prioritized sweeping outperforms Dyna and Q-learning at earlier timesteps. This is to be expected as prioritized sweeping prioritizes the most influential updates in the planning step, resulting in less wasteful updates. On the stochastic environment however we see that Dyna quickly overtakes prioritized sweeping in performance and converges to the optimal episode return, which prioritized sweeping does not. This is a clear indication that the direct RL step, which is absent in prioritized sweeping, is more influential in a stochastic setting than the difference in planning strategies. This intuitively makes sense as a more efficient planning strategy does not help to fill the gaps where the learned model is lacking information. This is again not an issue on the deterministic environment, where the model perfectly represents the real transition dynamics, leading to convergence for all algorithms.

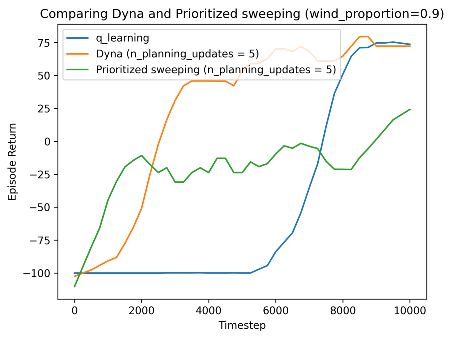

Figure 6: Comparison between the Q-learning baseline and the best performing Dyna and prioritized sweeping models on the stochastic WindyGridworld environment.

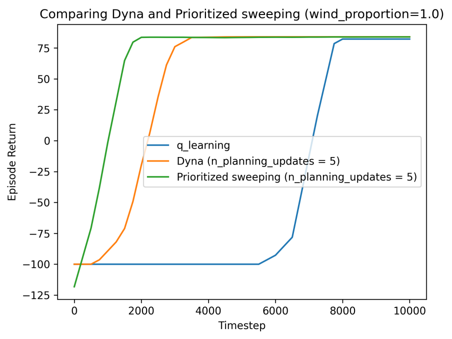

Figure 7: Comparison between the Q-learning baseline and the best performing Dyna and prioritized sweeping models on the deterministic WindyGridworld environment.

6 Discussion and Conclusion

In comparing our two model-based reinforcement learning algorithms we have seen strengths and weaknesses compared to model-free approaches, like Q-learning. A great strength is that we converge to optimal Q-values in fewer timesteps, assuming a deterministic environment. In a deterministic environment, the model learned by the algorithm is perfectly coherent to the environment, which makes it a reliable resource to simulate experience. In real-world problems, taking steps in the environment could come with a cost, either monetarily or in terms of time. When we exploit every step taken in the environment by building a model and plan from this model, we accelerate the learning process and this will could save us actions taken in the real environment. A weakness of model-based reinforcement learning lies within the learning of the model. In a stochastic environment, we need to try state-action pairs multiple times before our model becomes a good enough approximation to do meaningful planning on. When the algorithm learns a wrong or sub-optimal model, it will not make the best planning decisions and will not perform well. This aspect of model-based learning was highlighed when comparing Dyna and prioritized sweeping. Both used planning but only Dyna combined it with direct RL and therefore it performed better in the stochastic environment. Prioritized sweeping on the other hand performed better in a deterministic environment, because of the more efficient planning strategy. The question therefore arises if the best aspects of both algorithms could be combined by adding a direct RL step to prioritized sweeping in order to make it more resilient to randomness in the environment. This could be an interesting topic for further study. In our implementations we always used ϵ-greedy action selection to balance exploration and exploitation, which might not be the smartest approach for model-based learning. As stated before, the performance of our algorithms on stochastic environments is very dependent on the accuracy of the learned model, so ideally you would like an exploration approach that encourages taking actions for which our model does not yet have much information. We have seen such an approach before in multi-armed bandit problems, which was the upper confidence bound (UCB) action selection described in equation 2.10 in Sutton & Barto (2018). This method takes into account the uncertainty of the estimated Q-values by giving a bonus to actions which we have not taken many times before in the real environment. Because we already keep track of the transition counts to derive out model, this is an easy extension to make and could possible lead to faster and more targeted exploration. Another source of exploration is the choice for the initial Q-values, which we have set to 0. When the agent takes a few steps in the environment, but does not reach the goal, the Q-values on these steps will be negative as the default reward is -1. The Q-values for the untried actions therefore stay at 0 and this means that the agent will be directed more towards the untried actions. Overall, we have observed that balancing exploration and exploitation in model-based reinforcement learning is not only important for learning the Q-values , but also for added model-learning step.

References

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.