CS-456 Mini Project: Dealing with Sparse Reward in Mountain Car Environment

Marija Zelic
Section of
Life Sciences Engineering

Nicolas Moyne
Section of
Life Sciences Engineering

I. Introduction

In this report, we will try to develop reinforcement learning methods to solve the Mountain Car problem, a deterministic Markov decision process, present in the Gymnasium environment [1]. The problem setting is the following: the car is placed between two hills and aims to strategically accelerate to the left or to the right to get out of the valley depicted in Figure 1. The goal is formulated as reaching the flag on top of the right hill (coordinate of the flag being 0.5) as quickly as possible, as such the agent is penalised with a reward of -1 for each timestep. The episode ends if either *termination* - the goal is reached or *truncation* - the length of the episode is 200 steps, occur.

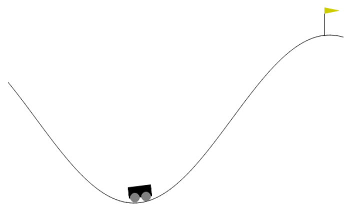


Fig. 1: Mountain Car environment.

In every state, we are provided with information on the car's position and velocity, therefore the problem is two-dimensional. The action space is discreet, with 3 possible actions at every state: accelerate forwards, accelerate backwards, or do nothing. More information about the observation space can be found in the Table I.

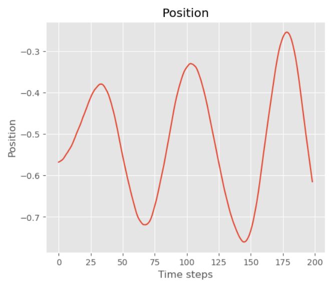
Num	Observation	Min	Max	Unit
0	position of the car along the x-axis	-1.2	0.6	position (m)
1	velocity of the car	-0.07	0.07	velocity (v)

TABLE I: Observation space.

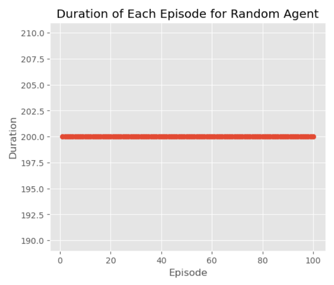
To get familiar with the environment, we will first observe the behaviour of the Random Agent. Later, we will delve into two strategies for solving the problem: a model-free algorithm, Deep Q-Network and a model-based one, Dyna.

II. Random Agent

We started gaining intuition on the task at hand by running a Random Agent, i.e. an agent which selects actions randomly. Figure 2a displays the position of the car as a Random Agent during one episode. We can notice that position changes according to sinusoidal law, which is expected owing to the transition dynamics. If we run the agent for 100 episodes, with the environment initialized using a different randomly sampled seed each time, we can track the duration of every episode, depicted in Figure 2b. Since the length of the episode is 200 steps, truncation occurs at every run, meaning that the agent never reaches the goal. Even though the position plot indicates the amplitude of the sinusoid getting larger, implying the car potentially getting out of the valley, 200 steps are not sufficient for that to happen. Therefore, we need to explore other approaches to help the agent learn faster.



(a) Position of the car as a Random Agent during one episode.



(b) Duration of the episode for Random Agent.

Fig. 2

III. Deep Q-Network (DQN)

A. No Auxiliary Reward

First, we will try to solve a problem at hand with the "basic" Deep Q-Network (DQN) algorithm. DQN agent uses the feed-forward network to estimate Q-values and ϵ-greedy policy to take actions. With DQN being an off-policy reinforcement learning algorithm, we implement a replay buffer that stores observed transitions and allows to sample batches of transitions for network updates.
As a feed-forward network, we employ two hidden layers multi-layer perceptron (MLP), with each layer having 64 neurons. Other parameters we set for the DQN agent are discount factor for the Q-values γ - 0.99, ϵ for ϵ-greedy policy - scheduled exponential decay from the maximum value of 0.9 to the minimum value of 0.05, size of the replay buffer - 10000, batch size - 64, and optimizer - AdamW. We train the

DQN agent with such a configuration for 1000 episodes and report its loss and cumulative reward per episode in Figure 3.

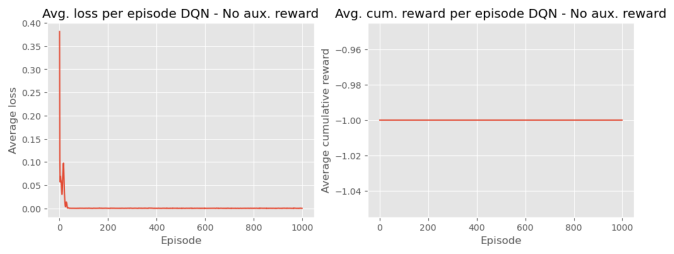


Fig. 3: Average loss (left) and average cumulative reward (right) per episode during training for DQN agent without auxiliary reward.

As we can observe from the plots, the average cumulative reward per episode is -1 and we report that the agent never manages to achieve the goal, i.e. in each of the 1000 runs, the end of the episode is due to the truncation after 200 steps. The cause for this mainly relies on the sparsity of the reward accumulated over the steps towards the goal. Essentially, with each taken step, the agent is punished for going further, without any indication of whether it is approaching the goal or not. We can try to improve its learning process by the strategy of *reward reshaping*, i.e. changing the reward function in such a way that will allow the agent to get a sense of an environment.

B. Heuristic reward function

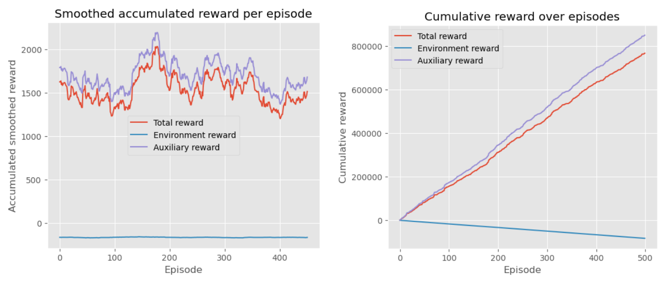
We are supposed to create an auxiliary reward function to assist the agent in learning better behaviour. For that purpose, we refer to the suggestions found in [2], where the authors recommend assigning a potential value to each state and adding to the reward of a transition the difference of potentials. Specifically, as a potential function, we consider:

$$\Psi(next\_state\_position) = 5|next\_state\_position - starting\_state|.$$

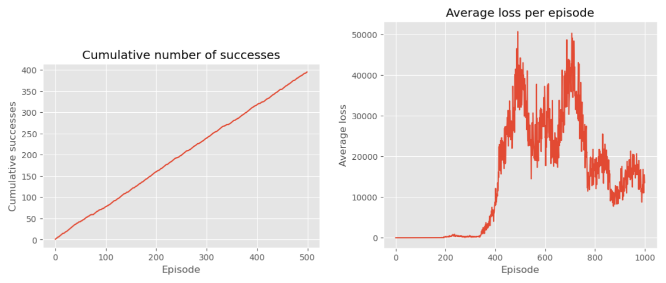
The rationale behind this potential function is to reward the agent for a large step out from the initial state. Since the initial state is uniformly randomly chosen from the range $[-0.6, -0.4]$, its mean value is -0.5, which is used as the starting state value in the formula. Additionally, we provide the agent with a substantial reward if it approaches the goal, i.e. we give 150, 100, and 50 for the position of the car larger or equal to 0.5, 0.3 and 0.1, respectively. DQN agent with the same parameter configuration as in the section III-A and this heuristic for the reward, run for 1000 training episodes, achieves testing results displayed in Figure 4.

With this approach, we can observe that the agent manages to successfully solve the task in almost 400 out of 500 testing episodes. A similar conclusion can be drawn from the length of an episode - compared to the Random Agent where each run was truncated, here we notice runs ending in the termination after approximately 150 steps. Accumulated reward has a large positive value owning to the implemented auxiliary reward.

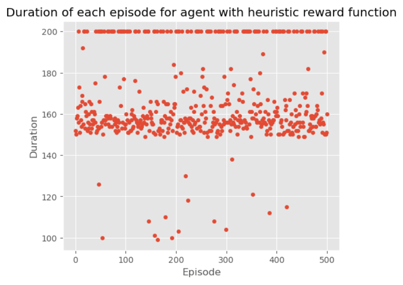
Contrasting to the cumulative environment reward, total and auxiliary cumulative rewards gradually increase over episodes.



(a) Accumulated reward per episode. (b) Cumulative reward over episodes.



(c) Cumulative number of successes over episodes. (d) Average training loss per episode.



(e) Duration of each episode.

Fig. 4: Plots of relevant metrics for DQN agent with heuristic reward function.

This setup is substantially more stimulating for the agent since the reward is not sparse.

A somewhat unusual result can be observed in Figure 4d, where the average training loss per episode is displayed. Namely, we cannot recognize the general decreasing trend in the loss function, instead, we notice atypical spikes. We hypothesize that this can be attributed to the fact that over time, the agent explores and stores different transitions in the replay buffer and tries to adjust network parameters so that they accurately represent every state-action pair. However, due to the large state space and reality of DQN being more accurate for the state-actions pairs the agent is supposed to encounter rather than less visited ones, i.e. agent makes mistakes on less frequent trajectories, therefore we have rapid increases in the loss function. Having a network with more parameters would potentially be able to capture more precise Q-Values for state-action pairs. Nevertheless, this solution is adequate since the agent successfully finds the goal.

We will briefly address the impact of smaller auxiliary reward on the model. If we discard the additional reward given when the goal is approached, i.e. we only keep the potential function as a reward, the agent manages to solve the task on training but fails during the testing phase. Therefore, a smaller reward is not sufficient for the agent to learn the environment and reach the goal.

C. *Non-domain specific reward*

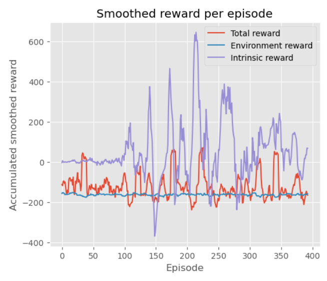
Well-defined heuristic reward functions work well but have the undesirable property of requiring knowledge about the specific environment that they are designed for and therefore not necessarily generalizing to different ones. Therefore, we implement Random Network Distillation (RND) [3], which encourages the agent to explore the environment by rewarding it highly for reaching less-frequently encountered areas and giving lower rewards for reaching well-known states.

This technique involves a target network (with fixed random weights) and a predictor network (with adaptable weights), both MLPs taking a state as input and outputting a single number, with the predictor network trained to match the target network on the states that are encountered. We define the RND intrinsic reward as the squared difference between the predictor and the target applied to a specific state, with some normalization - *reward and observation normalization*. The scale of such an intrinsic reward can vary largely for different environments and steps in episodes in training, therefore its normalization is imperative. To obtain the reward it is necessary to input the state, so it is also important to keep their scale consistent by normalization of observations. Both are achieved by subtracting a running mean and dividing by the standard deviation of respective parameters, with additional clipping after reward normalization.

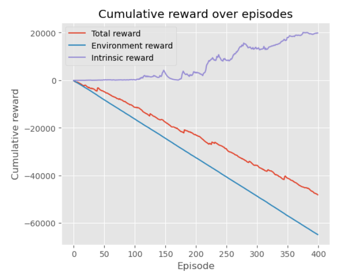
Another important parameter of DQN with the RND model is *reward factor* that balances intrinsic reward and the one received from the environment. Value selection for this parameter boils down to whether we want to favour exploration or exploitation. Since we already proved environment's reward is not suitable for the learning process and we want to push the agent to explore the environment more, we believe *reward factor* should have a value large enough to make the intrinsic reward larger than the environment's.

We report the 400-episode testing results for the DQN model with the same parameter configuration as in the section III-A and RND as two-layer MLP and reward factor - 100, run for 400 training episodes in Figure 5.

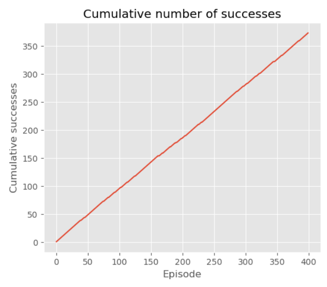
Based on the plot in 5c, we can observe that the agent successfully solves the task in over 375 out of 400 testing episodes, which is an impressive achievement rate. This agent takes around 10 steps more on average to find the goal, according to Figure 5e. In the first 100 training episodes, there is a very low accumulated reward per episode, which can be attributed to an insufficient amount of observations and rewards for normalization and therefore inaccurate representation. The total cumulative reward over episodes is negative for this agent, with intrinsic reward increasing over time, making it able to learn. Since this agent is in its essence DQN model,



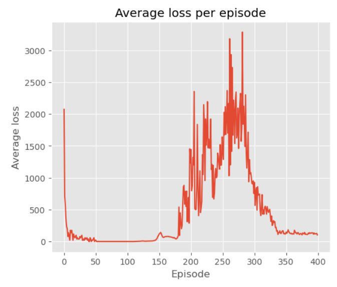
(a) Accumulated reward per episode.



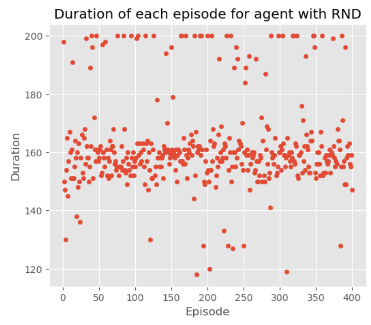
(b) Cumulative reward over episodes.



(c) Cumulative number of successes over episodes.



(d) Average training loss per episode.



(e) Duration of each episode.

Fig. 5: Plots of relevant metrics for DQN agent with RND model for intrinsic reward.

we notice a similar pattern of spikes in training loss in Figure 5d as in 4d, with however, more evident convergence towards the end of testing.

IV. Dyna

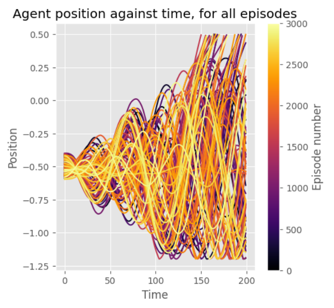
A. *Implementation and first results*

After considering a DQN to solve the problem, we will build a Dyna agent to work out the problem. Dyna's algorithm uses observations to construct a model of the environment which is then used for learning as explained in [4]. The appropriate modelisation for environment learning is using a discrete tabular state space and estimating the transition probability. We therefore discretize the environment using fixed bin sizes (which gives us two hyperparameters: bin size for acceleration and bin size for speed).

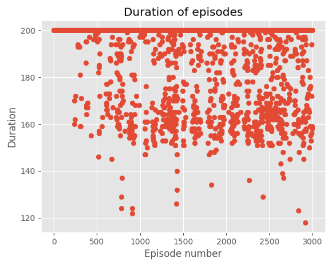
During training, the agent constructs a model of the environment composed of two main components: an estimation of transition probabilities, denoted as $\hat{P}_{s,a}(s')$, and an estimation of the reward for each state-action pair, denoted as $\hat{R}(s,a)$. The term $\hat{P}_{s,a}(s')$ indicates the expected probability of transitioning to state s' after performing action a in state s, whereas $\hat{R}(s,a)$ represents the expected reward for taking action a in state s. These estimates are continually refined with each new observation. $\hat{P}_{s,a}(s')$ is also used at each training episode to simulate k new transitions and refine $\hat{R}(s,a)$.

For the most part in this section we will use $n = 3000$ episodes, $k = 20$, a discretization bin size $d = [0.025, 0.005]$, a discount factor $\gamma = 0.95$ and epsilon decay from $\epsilon_{max} = 0.9$ to $\epsilon_{min} = 0.05$ with a decay factor of 0.997.

After training the agent with the parameters above, we get a training progression as in Figure 6a, where we can see that more and more trajectories succeed in reaching the goal as training progresses. This is confirmed by the scatter plot in Figure 6b.



(a) Car trajectories during training for 3000 episodes.



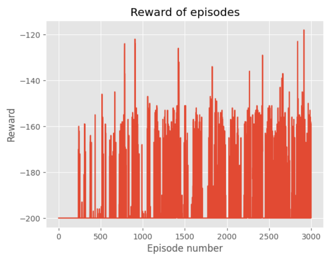
(b) Episode duration for Dyna agent during training.

Fig. 6: Dyna agent's performance progression during training.

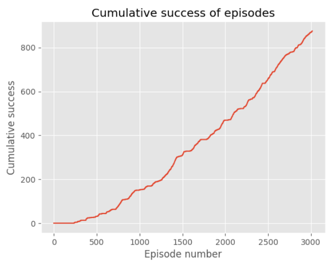
The agent after training succeeds with a probability of 0.608.

B. Training progression

We will now concentrate on the way the training progresses. First, we visualize the reward per episode in 7a and the cumulative success over episodes in 7b. At first glance, we do not observe anything of interest,i.e. no particular pattern except for the fact that the training indeed works (number of successes gradually increases towards the end).



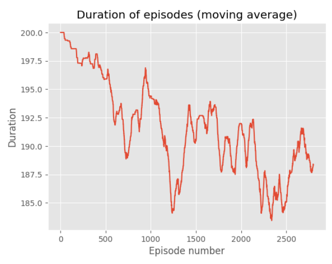
(a) Rewards obtained during training for 3000 episodes.



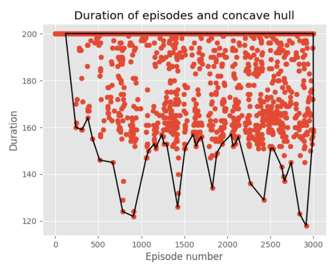
(b) Cumuative success during Dyna agent's training.

Fig. 7: Agent's success progression during training.

Nevertheless, the density of high rewards seems less important at the end of the training than at the beginning. To verify this, we will perform further analysis. In Figure 8a, we notice that the training's progression is not at all linear and that the performance oscillates after around 800 episodes, while still globally improving. This is confirmed in Figure 8b which shows a weighted concave hull of episode duration using the *alphashape* library, which can be interpreted as an optimistic evaluation of the model's performance.



(a) Smoothed average of episode duration during training.



(b) Episode duration's convex hull for Dyna agent.

Fig. 8: Agent's performance evolution during training.

C. Impact of bin size

We will now try to assess the impact of the bin size. We see its influence on the two main factors: performance and time. We take as a baseline the default bin size described earlier and provide in Table II the performance and training time for various bin sizes.

Size ratio	Bin size	Test performance	Training time
0.5	$[0.0125, 0.0025]$	1.0	64 minutes
1	$[0.025, 0.005]$	0.9	25 minutes
2	$[0.050, 0.010]$	0.4	20 minutes
4	$[0.100, 0.020]$	0.02	20 minutes

TABLE II: Test performance and time for different size ratios and bin sizes.

We chose to stay with our default-chosen bin size and not move up to half this bin size as 0.6 success probability is enough as proof of concept, while the difference between 1 hour of computing time and 25 minutes is the limit between being able to tweak our agent's code without worrying of having to retrain it and just not being able to do much change in the code.

D. Q-Values learning and trajectories

During training, the agent learns the Q-Values for each state-action pair, which are then used to make decisions using an ϵ-greedy policy. The Figure 9 displays the Q-Values during training.

More precisely, for a state s, we display $\max_a Q(s,a)$. We can see that the Q-values map expands during training. We initialized the Q values at 0 to favour exploration. The final map seems to reflect well the situation: states with high speeds or near the top of the mountain have the highest Q-value, while states in the valley with low speeds have substantially lower Q-Values.

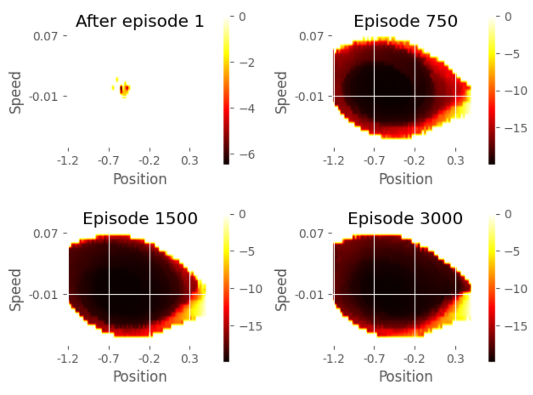


Fig. 9: $\max_a Q(s,a)$ during training

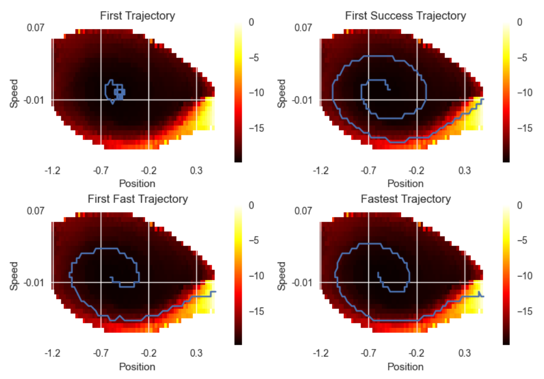


Fig. 10: Trajectories during traing

Finally, we visualize in Figure 10 the way the Q-Values reflect on the trajectories taken by the agent. We want to visualize relevant trajectories to grasp the agent's evolution, so we chose various important moments during training :

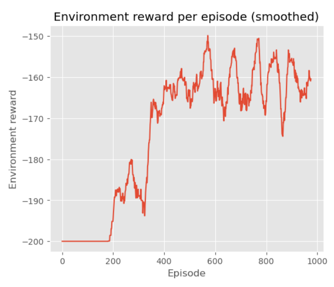
- First trajectory
- First trajectory to succeed
- First trajectory to succeed under 120 steps
- Fastest trajectory

The Q-Values displayed are the ones learned after 3000 episodes.

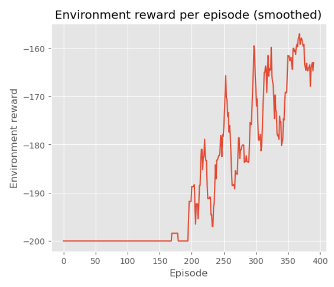
We denote that the model first learns that to get to the top of the hill, he has to balance back and forth (i.e. make spirals on the figure, which from a physical point of view is a phase diagram) and then learns to optimize this. This can be seen by the fact that on the first successful trajectory there is a position where the agent passes 4 times while on the fastest trajectories, the maximum number of times a position is crossed is 3.

V. Comparison of Dyna and DQN

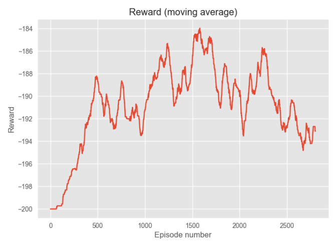
To compare the training performance of Dyna, DQN with RND and DQN with auxiliary reward, we trained those models for the appropriate number of episodes: 3000 for Dyna, 1000 for DQN with auxiliary reward and 400 for DQN with RND. Figure 11 shows the difference of treatment regarding the environment reward for DQNs methods compared to Dyna: for the latter, the reward obtained evolves fast with training (it is the objective) while for DQNs, it is more the total reward (environment + auxiliary of environment + intrinsic) that changes through training.



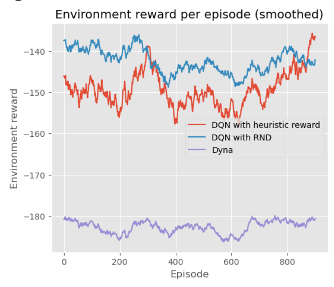
(a) DQN with auxiliary reward - 400 episodes



(b) DQN with RND - 400 episodes



(c) Dyna, 3000 episodes



(d) Testing performance

Fig. 11: Smoothed training and testing performance.

We can then compare the testing performance of the three algorithms in various ways :

- Task solving: all models solve the task. Dyna is between 30 to 40 steps slower than DQNs.
- Consistency: while Dyna has the worst performance, it is the most consistent model.
- Training: Dyna's short training time (25 minutes vs. 1 hour for DQN with RND) makes it a still relevant model.

VI. Conclusion

In this study, we contrasted the way Dyna and Deep Q-Network (DQN) algorithms manage to address the Mountain Car problem. Initially, we showed DQN's inability to succeed without a heuristic reward, but the task is solved with its inclusion. Next, we demonstrate how Dyna, a model-based approach, obviates the necessity for a heuristic reward. Finally, we established that DQN methods yield approximately the same result (DQN with RND being slightly better), while Dyna solves the task with a smaller environmental reward, but with greater consistency.

References

[1] Gymnasium documentation - mountain car. [Online]. Available: https://gymnasium.farama.org/environments/classic_control/mountain_car/

[2] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Icml*, vol. 99, 1999, pp. 278–287.

[3] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, "Exploration by random network distillation," 2018.

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018, ch. 8.2.