Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. Matthias Schubert
Sebastian Schmoll, Sabrina Friedl

Munich, 21.01.2019

Deep Learning and Artificial Intelligence

WS 2018/19

Exercise 12: Function Approximation

Exercise 12-1 Value Function Approximation

(a) Name two problems that come along with large MDPs!

(b) Given an approximation function $\hat{U}(S, w) \approx U_\pi(S)$, parameterized by a weight vector w. What is the prediction objective (the error) we want to minimize?

(c) Write down the gradient of this objective with respect to the parameter, i.e. $\nabla_w \overline{\text{VE}}(w)$!

(d) Gradient descent adjusts w in the direction of the negative gradient by the update rule $w \leftarrow w + \Delta w$ with $\Delta w = -\frac{1}{2}\alpha \nabla_w \overline{\text{VE}}(w)$, where α is the learning rate.

Write down the update rule for Stochastic Gradient Descent (SGD), i.e. for a single sample S (without weighting).

(e) What is the SGD update rule for a linear approximation function $\hat{U}(S, w) = x(S)^T w$, where $x(S)$ is a feature vector of the same dimensionality as w?

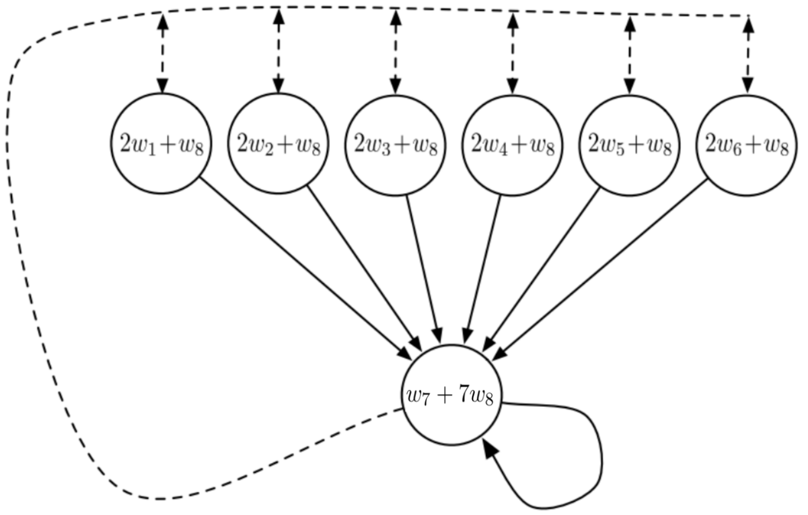
(f) In practice, we do not have the true value function $U_\pi(S)$, but only rewards. Write down Δw when using the targets from Monte-Carlo (MC)- and one-step Temporal Difference (TD(0)) prediction respectively!

(g) Why is TD called a semi-gradient method?

Exercise 12-2 Baird's Counterexample

Consider the episodic seven-state, two-action MDP shown below. The dashed action takes the system to one of the six upper states with equal probability, whereas the solid action takes the system to the seventh state. In state 7, the behavior policy μ selects the dashed and solid actions with probabilities $\frac{6}{7}$ and $\frac{1}{7}$, so that each state has an equal probability of being next. In states 1 to 6, μ takes the solid action (with probability 1). The target policy π always takes the solid action in all states. The starting distribution for each episode is the uniform distribution. The discount rate is $\gamma = 0.9$.

Each state value $U(s)$ is approximated by a linear function $\hat{U}(S, w) = x(S)^T w$ where $x, w \in \mathbb{R}^8$ (both indices start at 1). For example, the feature vector for the first state is $x(1) = (2, 0, 0, 0, 0, 0, 0, 1)^T$ and thus $\hat{U}(1, w) = 2w_1 + w_8$. The reward is zero on all transitions, so the true value function is $U_\pi(s) = 0$ for all s, so an exact approximation would be $w = \vec{0}$. This seems like a problem that should be easy to solve with the techniques we learned so far. In this exercise, you will see that – as Baird showed – the weights will diverge to infinity with semi-gradient off-policy methods.



$\pi(solid \mid \cdot) = 1$
$\mu(dashed \mid \cdot) = 6/7$
$\mu(solid \mid \cdot) = 1/7$
$\gamma = 0.9$

(a) Let the weight vector be initialized as $w = (1, \dots, 1)^T$. For each of the states $s = 1, 2, \dots, 7$ calculate Δw for the transition (s, r, s') from s to state $s' = 7$ successively, i.e. calculate

$$\Delta w = \alpha(r + \gamma U(s') - U(s))\nabla_w U(s).$$

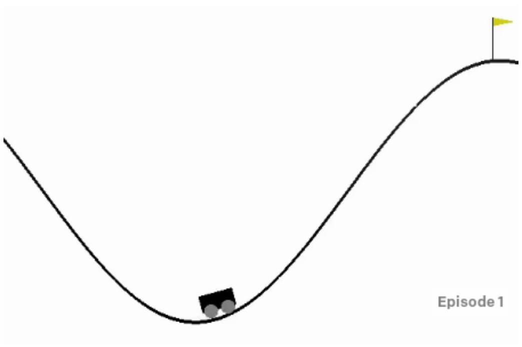
Update $w \leftarrow w + \Delta w$ after each of the transitions! You can do this by hand or write a short program to do the calculations.

(b) How do the different transitions affect the components of the weight vector and the state value $U(7)$?

(c) What would happen if we repeat this sequential update procedure over and over?

Exercise 12-3 Mountain Car

Mountain Car is a well known benchmark for reinforcement learning algorithms. As you can see in the figure below, the domain contains a car located in a valley between two mountains.



The goal is to drive the car to the top of the right mountain. However, the car's engine is not strong enough to get up the mountain, so the only way to achive the goal is to drive back and forth in order to gain some momentum. The OpenAi Gym framework provides a convenient Python API for this (and other) benchmarks.

On the lecture website, you will find an ipython notebook that uses the Gym library to generate an environment for the mountain car problem in which an agent can be trained. In the "Main" part, the function `episode` simulates an episode in which the agent tries to learn how to reach the goal and gives back the (undiscounted) return. For every action, the method env.step() provides the next state and the reward. The learning is implemented on-policy, i.e. the policy is updated after each step (agent.update()). An episode terminates when the goal is reached or after 200 steps. Each time step receives a reward of -1, until the goal position is reached. The function `train` executes a given number of episodes and decreases the parameter ϵ after each episode.

(a) In the class `AbstractQLearn`, implement a general ϵ-greedy policy that selects a random action with probability ϵ and otherwise chooses the action with maximal Q-value (use the abstract get_q_value method).

(b) In the class `QLearningAentTabular`, implement the missing methods for the table-based Q-learning agent. Note that the state space is a two dimensional continuous vector. The method `_discretize` provides a discretization such that each dimension has 10 intervals.

(c) In the class `QLearningAgentApproximator`, implement semi-gradient Q-learning for linear action-value function approximation. The script already provides an appropriate feature vector based on radial basis functions.