Prof. Dr. Jochen Garcke
Jannik Schürg

5 REINFORCEMENT LEARNING

Send your solutions to this chapter's tasks until
July 2nd.

So far we have seen examples for supervised learning (e.g., SVMs or neural networks) and unsupervised learning (PCA). In this chapter we look into *reinforcement learning*, which is often associated with *artificial intelligence* (AI). Instead of predicting targets, reinforcement learning tries to make *sequential decisions* in a given environment. A decision has to be made among a given set of *actions*. The goal is to obtain a so called *agent*, a map which tells the environment which action to apply given the current state. The agent is guided[1] in its decisions through rewards (possibly negative) which are manually chosen. An important aspect of reinforcement learning is the fact that no details about the environment are exposed to the agent, the agent can only learn by applying actions and observing rewards.

Reinforcement learning has been prominently applied to games. For games typically a positive reward is given for a win and a negative reward for a loss, all other rewards are zero. One might be tempted to give non-zero reward for subjectively good or bad events (like losing a valuable piece in chess, for example), but by not designing a more sophisticated reward, the agent can come up with original strategies, which are often superior to humans. In this way, the agent is only told the ultimate goal, but not how to achieve it. To give an example, the set of actions for a game could consist of the allowed moves (in a board game) or buttons to press (in a video game).

In many applications, at the beginning the agent faces an enormously large environment with rewards "hidden" deep down a sequence of steps, similar to a labyrinth. It is unknown where a decision will lead, also the outcome might follow a probability distribution, i.e., the same decision in the same state (situation in which a decision has to be made) can lead to different new states. A path through the state space, a sequence of states, actions and rewards, is called a *trajectory*. The agent might not only be rewarded late in a trajectory, the immediate rewards (rewards early in the trajectory) might be lower or negative, too. Decisions are often based on an estimated value of a state. The estimate is updated from observations. This mutual influence makes analysis, and reinforcement learning in general, hard.

An early success is the application of reinforcement learning to the board game backgammon. The resulting agent, *TD-Gammon* (1992), did beat world class players. Computer programs playing games have

1 In the spirit of AI, the agent is often personalized in texts.

a long tradition. The chess program DeepBlue is very famous, another successful example is Chinook [8], a hand-built program for the game checkers (draughts). It was beaten in 1990 by Marion Tinsley, who is considered the best checkers player of all-time, being significantly ahead to his peers. After the loss, Chinook was improved and in a rematch in 1994 Mario Tinsley had to withdraw after the first games due to health problems. He later died. It seemed Jonathan Schaeffer, the lead developer of Chinook, should never know whether his program could have beaten Tinsley. In 2007, after over ten years of computation, Schaeffer and his team weakly solved checkers, erasing any doubt whether Tinsley could have won. Marion Tinsley has lost seven games in his full career, two of those were against Chinook.

In 2015 the computer program AlphaGo, partly based on reinforcement learning, did beat a world-class Go player, an event predicted not to happen in the next ten years or more. One could say reinforcement learning managed to become persona non grata of several communities by beating human intelligence. More to come[2].

5.1 PROBLEM FORMULATION

Let us describe the mathematical setting which we use for reinforcement learning. First, we want to model dynamic systems or *environments*. Let S be a set of states. In each state $s \in S$ we can choose an *action* a from a non-empty set A_s. If we apply in state $s \in S$ an action $a \in A_s$ our environment transitions to a new state $s' \in S$. In order to specify this transition let

$$\mathcal{P} \colon \{(s,a) \mid s \in S,\ a \in A_s\} \times \mathcal{B}(S) \to [0,1]$$

be a *stochastic kernel*, where $\mathcal{B}(S)$ is the Borel σ-algebra of S (assume that S is Polish). The exact definition of a stochastic kernel is, among other things, omitted here, the most important property for our purposes is, that $\mathcal{P}((s,a),\cdot)$ is a probability measure which we use to model how likely a state s' is as the next state, given we are in state s and apply action a.

For simplification, we assume our environment to be deterministic, i.e., $\mathcal{P}((s,a),\cdot)$ is a point measure. To ease notation in this case, $\mathcal{P}$ is considered to be a map from $\{(s,a) \mid s \in S,\ a \in A_s\}$ to S.

EXAMPLE For no particular reason let us consider photocopiers as an environment. To describe the state of a photocopier, we could pick some very general states, for example "working" and "not working", or we might position a camera pointed at the photocopier and then consider a picture of the photocopier as its state. The set of actions could include pressing the buttons, fixing a paper jam, or to verbally threaten it. After choosing an action, the photocopier transitions into a new

2 You think you are an awesome car driver? Well, be ready.

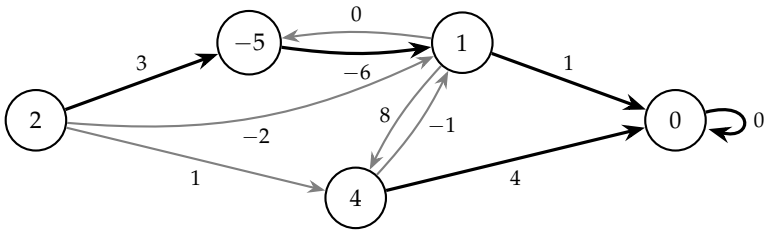


Figure 5.1: A abstract example for a deterministic environment. The environment is represented by a graph, every state is a node, every action an edge. The reward is written on the edges. A policy is visualized by bold edges. For this policy, the value V^π is denoted inside the nodes (with $\gamma = 1$).

state. Depending on your beliefs about photocopiers, this transition is deterministic. The state can also stay the same: the photocopier's metaphysical condition which is affected for example by verbal threats is not reflected in the state space usually.

Our goal is to find an optimal *policy* $\pi : S \to A$ that tells us which action to pick in a state. A policy is optimal, if it maximizes the total reward received. The (positive or negative) reward is defined by utilizing a reward function

$$\mathcal{R}\colon \{(s,a) \mid s \in S,\ a \in A_s\} \to \mathbb{R},$$

which tells us the reward we get for picking action a in state s. Then, let $s_0, s_1, \ldots$ be the sequence of states (a *trajectory*) we obtain when following π with an initial state s_0, i.e., $s_1 = \mathcal{P}(s_0, \pi(s_0))$ and so on. The total reward received for a given policy π is then defined by the *accumulated discounted reward*

$$V^\pi(s_0) := \sum_i \gamma^i \mathcal{R}(s_i, \pi(s_i)), \tag{5.1}$$

where $\gamma \in (0,1]$ is a constant *discount factor* which diminishes delayed rewards. For $\gamma = 1$ we speak of an *undiscounted* problem, otherwise the problem is called *discounted*. The analysis for discounted problem is usually easier.

See fig. 5.1 for an abstract example.

The sum in eq. (5.1) is sometimes cut-off after a fixed number of steps in which case we call the problem a *finite horizon* problem.

Often, it is natural to consider environments with *terminal* states. Those are states which have no actions (e.g., a game ends, a car crashes). To fit them into the mathematical model those states get a dummy action which leads back to the same terminal state with zero reward. You may also think of those states as absorbing states. If the policy reaches a terminal state, the sum is effectively finite because all remaining terms are zero.

Remark. The tuple $(S, \{A_s\}_{s\in S}, \mathcal{P}, \mathcal{R})$ is called a *Markov decision process* (MDP). Combined with a policy we obtain a Markov chain. In a more

general form, policies π are allowed to be non-deterministic or time-dependent (non-stationary). Infinite action spaces and continuous time are natural extensions. See the books [4, 7] for a detailed description.

The reinforcement learning problem can now be stated as the following optimization problem: Find an optimal policy

$$\pi^* := \arg\max_{\pi} V^{\pi}(s_0)$$

for some s_0. The accumulated discounted reward for the optimal policy π^* is called *value function* and is denoted by $V(s) := V^{\pi^*}(s)$.

EXAMPLE We continue our photocopier example. Assume we want to copy a set of sheets of paper while combining two (!) on each new paper. Denote this state (having a copy with two pages combined on every new page) as terminal and define the reward to be -1 (or any other fixed negative number) unless we reach the terminal state. An optimal policy then is a sequence of steps which achieves this goal as fast as possible.

BELLMAN EQUATION We can split up the sum eq. (5.1) in the first and remaining terms and then obtain

$$V^{\pi}(s_0) = \mathcal{R}(s_0, \pi(s_0)) + \gamma V^{\pi}(s_1),$$

and further, the value function fulfills the so called Bellman equation

$$V(s_0) = \max_{a \in A_{s_0}} \big(\mathcal{R}(s_0, a) + \gamma V(s_1)\big), \tag{5.2}$$

where s_1 is again given by $\mathcal{P}(s_1, a)$. One can show, that a fixed-point of eq. (5.2), so a function V which fulfills eq. (5.2) for every $s \in S$ is a value function. We can obtain a policy by choosing a maximizing action in each step,

$$\pi^*(s) := \arg\max_{a} \big(\mathcal{R}(s, a) + \gamma V(\mathcal{P}(s, a))\big). \tag{5.3}$$

Such a policy maximizes $V^{\pi^*}(s)$ for every s.

The idea behind the Bellman equation is the *Bellman optimality principle*: Suppose π^* is an optimal policy and $s_0, s_1, \ldots$ is the sequence of states when following π^*. Then, the Bellman optimality principle states, that π^* is also an optimal policy if we start in any later s_i of the sequence (trivial, argue by contradiction). This is also known as the *dynamic programming principle*.

5.1.1 *Value iteration*

If the operator which applies eq. (5.2),

$$\begin{aligned} \mathcal{T} \colon \{f \colon S \to \mathbb{R}\} &\to \{f \colon S \to \mathbb{R}\} \\ f &\mapsto \big(s \mapsto \max_{a \in A_s} \big(\mathcal{R}(s, a) + \gamma f(\mathcal{P}(s, a))\big)\big), \end{aligned}$$

is a contraction on some Banach space, we get from Banach's fixed-point theorem that there exists a unique fixed-point which can be computed iteratively (fixed-point iteration). For finite state spaces S and finite action spaces A, the contraction property can be shown quite easily, especially if $\gamma < 1$.

This idea leads to an algorithm called *value iteration* which is described in algorithm 5.1. Convergence can be shown under a variety of assumptions.

Algorithm 5.1 Value iteration.

function ValueIteration(S, $\{A_s\}_{s\in S}$, $\mathcal{R}$, $\mathcal{P}$, γ)
  Initialize $V(s) \leftarrow 0$ for every $s \in S$.
  repeat
    $V \leftarrow \mathcal{T}(V)$
  until V did not change.
  return V.
end function

5.1.2 *Policy iteration*

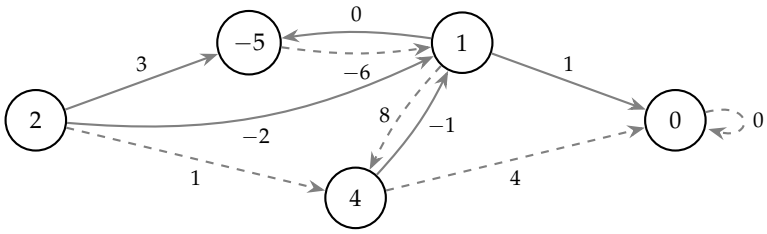
Instead of storing V we can also try to compute an optimal policy directly. A similar algorithm, *policy iteration*, is based on eq. (5.3) and is described in algorithm 5.2.

Algorithm 5.2 Policy iteration.

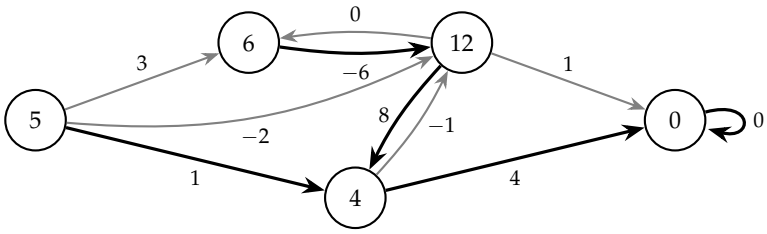
1: **function** PolicyIteration(S, $\{A_s\}_{s\in S}$, $\mathcal{R}$, $\mathcal{P}$, γ)
2: Initialize $\pi(s) \leftarrow a$ for an action $a \in A_s$ and every $s \in S$.
3: Compute V^π.
4: **repeat**
5: $\pi(s) \leftarrow \arg\max_a(\mathcal{R}(s,a) + \gamma V^\pi(\mathcal{P}(s,a)))$ for every $s \in S$.
6: Update V^π.
7: **until** V^π did not change.
8: **return** π.
9: **end function**

Hint!
Exploit for terminal states s that $V(\mathcal{P}(s,a)) = 0$ in algorithm 5.1, as well as $V^\pi(\mathcal{P}(s,a)) = 0$ in algorithm 5.2. Furthermore, check for cycles if you compute V^π in algorithm 5.2 by following π.

In line 6 we compute the value function V^π for the current policy π. This step is called *policy evaluation*. The value function can be computed by following the policy, or by using a fixed-point iteration. After V^π is computed, we update π greedily in line 5, which is called *policy improvement*. See fig. 5.2 for an example.



(a) Policy improvement.



(b) Policy evaluation.

Figure 5.2: A step of policy improvement and evaluation for the example from fig. 5.1. The new action is shown using a dashed edge. The next iteration will yield an optimal policy.

One can show, that policy iteration converges not slower than value iteration. Policy iteration can also be seen as a form of Newton's method.

5.1.3 *General remarks*

Value and policy iteration are basic building blocks for reinforcement learning algorithms. They themself are not considered reinforcement learning, though. The reason is, that for reinforcement learning the learning algorithm must not explicitly use the model of the system ($\mathcal{P}$ in this case). While *model based* reinforcement learning exists, there, the algorithm *learns* the model from observations, it is not given as an input. Often, software simulations of environments are used to test reinforcement learning algorithms. There, the model used for the simulation must not be available to the learning algorithm. In practical applications this requirement of an a priori unknown model might be weakened.

In the form described here, value and policy iteration are only applicable with small, finite state and action spaces. For infinite or large state spaces we can neither iterate through all states, nor store V^π. In this case one can use supervised learning (function approximation) in order to represent V^π or π using samples. We will see an example for such a method below. For linear function approximations there is theory available reaching back to the 80s.

5.2 ALTERNATING MARKOV GAMES

Related: Stochastic Games, Competitive Markov Decision Processes

In this section we will apply value and policy iterations to a popular problem class: games. In AI, board games were one of the first things people tried to tackle. Many board games can be modeled as a simple dynamic system which is easy to implement, fast to run, fun to play, but at the same time hard to master. Some examples are chess, checkers (English draughts), and Go.

In a game with N players which pick their action one after another, we assign a policy π_i to each player. A policy π_i is called a *best response* if it maximizes the total reward received when playing against previously fixed policies of all other players. To compute a best response fix all policies except for one, the resulting game is then an example for an environment as described in section 5.1. Usually, we want to find a set of policies $\pi_1^*, \ldots, \pi_N^*$ such that every policy π_j^* is a best response to the other policies $\pi_1^*, \ldots, \pi_{j-1}^*, \pi_{j+1}^*, \ldots, \pi_N^*$. This is called a *Nash equilibrium*[3] and its existence is known for the class of games we consider. Assuming that all opponents always have a best-response, a perfect player will always want to follow π_j^* as any deviation leads to a reduced reward.

We only consider games for two payers, black and white, which take actions in alternating order. The game state is fully observable (perfect information) by each player, this means there are, for example, no hidden cards. Each player has a reward function $\mathcal{R}_i$ and we assume that

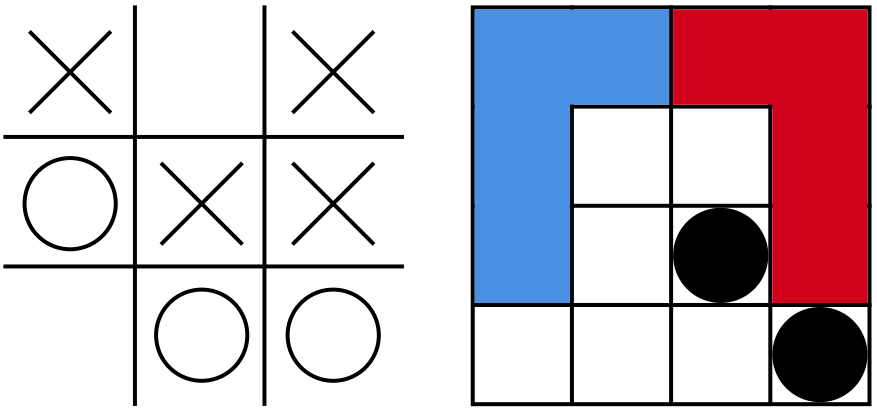
$$\mathcal{R}_1(s, a) = -\mathcal{R}_2(s, a) \quad \text{for all } s \in S,\ a \in A_s. \tag{5.4}$$

A state encodes which player has to pick an action, so eq. (5.4) implies that the reward black receives for an action of white has the opposite sign as the reward white receives for her own move. With other words, we assume a *zero-sum game*.

STATES OF EQUAL VALUE Often one can tell a priori which states will have equal value by using symmetries. The value of a Tic-tac-toe (noughts and crosses) board state for example is invariant under rotation and reflection if both players are perfect. To save computation we therefore consider equivalence classes of states which are known to have equal value. The accompanying material contains the implementation of two games and both implementations provide a function which maps any game state to a unique representative of its equivalence class. This representative is called the *normalized state* in the code.

Because of the zero-sum assumption we only need to compute one value function V which maps a state s to the total reward received for the current player. Similarly, only one reward function $\mathcal{R}$ is needed.

3 Go watch the (not so biographic) movie "A Beautiful Mind" (not *now*).



(a) Tic-Tac-Toe (b) L-Game

Figure 5.3: Two visualized states of the board games Tic-Tac-Toe and the L-Game.

The reward $\mathcal{R}(s, a)$ shall give the reward for the player who has to pick an action in state s. With other words, V and $\mathcal{R}$ always return the value and reward for the current player.

5.2.1 *Value iteration for alternating games*

Unless there is no particular reason one typically wants to find a policy which is optimal against a perfect player. But neither π_1^* nor π_2^* are available at the beginning. The Bellman equation 5.2 applies also for the two-player game and value iteration is known to converge in our case. In the following task the value iteration algorithm has to be adapted to our representation of the value function and reward in the two-player game setting.

Task 5.1. *Use the provided notebook as a template.*

(b) Modify algorithm 5.1 and implement it for alternating games.

(c) Test your implementation for Tic-tac-toe and the L-Game. Use a ± 1 reward on terminal states, depending on whether the current player lost or won, and a zero reward otherwise.

Hint!
For task 5.1 (b) apply a minor change to $\mathcal{T}$. Take the normalized states and how we use V and $\mathcal{R}$ into account.

Remark. Tic-tac-toe has 5,478 states and only 765 up to reflection and rotation. The L-Game has 18,368 states and 2,296 up to reflection, rotation and player's switched.

We will now use the value function of the L-Game to analyze it and get an improved policy.

Task 5.2.

(a) Compute the number of states with negative value for the L-Game. How many of those are terminal? What does it mean that a state has negative value but is not terminal (given our specific reward function)?

(b) Let the agent play against itself several times from the non-terminal states with negative value. Report the number of moves until the game finishes.

(c) Create a new reward based on the behavior from (b) to improve the L-Game policy. Be careful to preserve the zero-sum property.

5.2.2 *Computing π_1^* and π_2^* through self-play*

The policy iteration algorithm for alternating games is less straightforward. The major difference is that we have to iterate over two polices until their value functions are equal (more precisely, until they assign values of opposite sign to each state). While policy evaluation is easily adapted, policy improvement is not clear. The following steps lead to convergence:

1. Update π_1 greedily (as before).
2. Compute π_2 as a best response to the updated π_1.

The best response can be computed using policy iteration as in section 5.1 because the opponent's policy is fixed.

We now implement a variant of policy iteration for games.

Task 5.3.

(a) What is the environment for black? What is the environment for white? Why is the assumption on the environment made in section 5.1 no longer fulfilled when both policies are updated (self-play)?

(b) Implement the adapted policy improvement and policy evaluation.

(c) Why does the provided code compute two value functions in each iteration?

*(d) (**Optional bonus task**) It is likely that your policy improvement implementation gets trapped in an infinite loop for the L-Game. Why is that? Can you fix it?*

Hint!
When implementing policy evaluation make sure to sum up the rewards for a trajectory *correctly*.

5.3 DEEP Q-LEARNING

As reinforcement learning algorithms do not have a model a priori, for example, the update

$$V(s) \leftarrow \max_{a \in A_s} \big(\mathcal{R}(s,a) + \gamma V(\mathcal{P}(s,a)) \big)$$

is not possible since $\mathcal{P}(s,a)$ is not available. The algorithm can only interact with the environment by

- reading the current state and reward, or by
- applying an action.

When an action is applied, the environment returns a reward and the new state. The system dynamics $\mathcal{P}$ are not visible, the environment is a *black box*. With other words, the agent can only observe trajectories in the state space.

A very popular idea is to store the value instead of states for state-action pairs (s,a) in a new function $Q\colon S \times A \to \mathbb{R}$. A variant of value iteration works like this, we initialize Q and then iterate:

1. Let s be the current state, apply an action a, for example $a = \arg\max_{a \in A_s} Q(s,a)$ and observe a reward r and the new state s'.
2. Update $Q(s,a) \leftarrow r + \gamma \max_{a' \in A_{s'}} Q(s',a')$.

Observe that this scheme does not use $\mathcal{P}$ explicitly. The first step can be interpreted as a single step of policy evaluation, and the second as single step of policy improvement. We refer to this algorithm as *Q-learning*. The immediate switch between policy improvement and evaluation is very closely related to the so called SARSA algorithm, which got its name after the five variables, s, a, r, s', and a'.

Q-VALUE APPROXIMATION We now assume that S is too large to store and replace $Q(\cdot)$ by an approximation $Q_{\boldsymbol{\theta}}(\cdot)$ which is parameterized by a finite set of real values $\boldsymbol{\theta}$. A common choice is a linear approximation

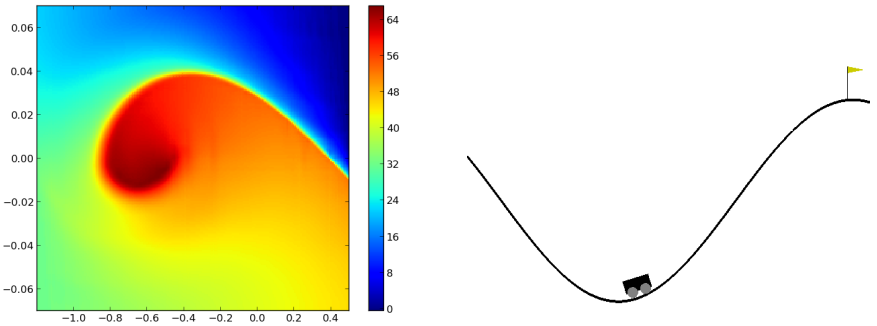
$$Q_{\boldsymbol{\theta}}(\cdot) = \sum_k \theta_k f_k(\cdot)$$

for a set of (feature) maps f_k. The update of Q

$$Q(s,a) \leftarrow r + \gamma \max_{a' \in A_{s'}} Q(s',a')$$

is then replaced by a step of gradient descent

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \Big(\big[r + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s',a') \big] - Q_{\boldsymbol{\theta}}(s,a) \Big) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a),$$



(a) Value function (b) Initial state

Figure 5.4: The value function of the mountain car environment (computed with sparse grids [3]) and a visualization of the state. A car which is under-powered has to reach the top of the mountain right. The state consists of a position in $[-1.2, 0.6]$ and a velocity in $[-0.07, 0.07]$.

with a step size α. The objective in this case was the mean-squared error

$$\frac{1}{2}\Big(y - Q_\theta(s,a)\Big)^2, \quad \text{with target } y = r + \gamma \max_{a'} Q_\theta(s', a').$$

For linear approximations convergence can be shown under certain assumptions [11]. It is known for a long time that nonlinear approximations are tricky (divergent) [11].

ε-GREEDY-POLICIES Since $\max_a(\mathcal{R}(s,a) + \gamma \max_{a'} Q(\mathcal{P}(s,a), a'))$ is no longer feasible, we need to ensure during training, that we get observations of transitions for every actions a in a state s. If a is always picked using the maximum, we could miss a trajectory/observation for a lucrative action. In other words, it is important to *explore* the state system and try actions which under the current approximation are estimated to have lower value. The balance between trying those actions and following the best action is called *exploration vs. exploitation*. We will use a so called ε-greedy-policy, which picks a random action with probability ε, and a maximizing action otherwise. During training, ε is decreased steadily.

5.3.1 *Q-Networks*

With the seminal achievements of deep neural network, the idea to try a neural network as an approximation Q_θ, where $\boldsymbol{\theta}$ denote the weights, was popularized by a very influential paper from 2015, published in Nature [6]. In this work, the authors managed to find a single DNN architecture and a reinforcement learning algorithm (with fixed hy-

perparameters[4]) which could achieve superhuman performance on 49 Atari games[5]. As a true reinforcement learning, no rules of the game (system dynamics) were available to the agent. In fact, a state consisted only of the last four frames[6] of the game, so the agent sees a visual representation of the game. Two main modifications were made to the approximated Q-learning algorithm described above. The resulting algorithm is called DQN (Deep Q-network). Atari games are now a benchmark in reinforcement learning research (called "Arcade Learning Environment").

In [6] three problems are named with classic Q-learning which they tried to tackle with DQN:

- The observed transitions s, a, r, s', a' are correlated (theory, for example Monte Carlo, needs independence).
- The targets y for updating $Q_{\boldsymbol{\theta}}(s,a)$ are computed using $Q_{\boldsymbol{\theta}}$ itself. This leads to oscillations.
- Small changes in $Q_{\boldsymbol{\theta}}$ can change policies significantly. With oscillations this can then cause divergence.

EXPERIENCE REPLAY To get less correlated transitions, a technique called *experience replay* is used. During training the observed transitions (s, a, r, s') are collected in a set, which is referred to as *replay memory*. When the network is updated, transitions are sampled from this memory and a gradient step is performed for this batch. In the original work the samples were drawn uniformly, but a subsequent paper [9] proposes to draw the samples using, for example, $|y - Q(s,a)|$ as weights. With a weighted sampling, a batch is more likely to contain transitions where $Q_{\boldsymbol{\theta}}$ has a high error. This method is called *prioritized experience replay*, an implementation is provided in the material.

DECOUPLED TARGETS Instead of computing the targets y using $Q_{\boldsymbol{\theta}}$, a second instance of *target weights* $\hat{\boldsymbol{\theta}}$ is used:

$$y = r + \gamma \max_{a'} Q_{\hat{\boldsymbol{\theta}}}(s', a').$$

Periodically, the update $\hat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}$ is performed. Only $\boldsymbol{\theta}$ is updated by gradient descent. Q-learning is known to overestimate values (non-uniformly). An existing solution for this issue was adapted to DQN resulting in *Double DQN* [14], abbreviated as DDQN. It boils down to computing the targets by

$$y = r + \gamma Q_{\hat{\boldsymbol{\theta}}}\big(s', \arg\max_{a'} Q_{\boldsymbol{\theta}}(s', a')\big),$$

4 For example, the step size is a hyperparameter, or the ε used for the ε-policy, or the batch size, training steps, preprocessing, etc. The weights of the network were trained new for each game.

5 Old video games, in the style of Tetris. You know Tetris, right?

6 A single picture of the game's screen. Typical games show 30 frames or more per second.

i.e., the maximizing action a' is selected using the (online) weights θ, and the Q-value is then computed with $\hat{\theta}$.

The training for DDQN is shown in algorithm 5.3. The parameters listed there are

- The maximal number of steps $n_{\max}$.
- The number of steps n_{replay} after which each time a gradient step is performed on θ using a batch from the replay memory.
- The number of steps n_{update} after which the target weights are updated each time.
- A preprocessing map ϕ which maps S to a (possibly equal) state space $\hat{S}$. For example, ϕ could normalize the data.
- The parameters for the ε-policy. The initial value ε_0 (e.g., 1.0), the final value ε_∞ (e.g., 0.01), and the number of steps n_ε between which ε is linearly decreased from ε_0 to ε_∞.

Often, an environment has terminal states. This is not explicitly handled in algorithm 5.3. When s' is terminal, the environment is reset. The time between such terminal states is called an *episode*.

Another inaccuracy due to simplification is that for prioritized experience replay sample weights are returned by the replay memory. Those sample weights are used in the mean-squared loss. See the material for details.

5.3.2 *Car Racing*

The Python package GYM by OPENAI provides implementations of environments for reinforcement learning. One such environment is the *car racing environment*. The agent controls a racing car on a track. The simulation is done in 2D, the state space is the bird's-eye view of the car and its closer surroundings, as well as the current velocity and momentum[7].

The reward function gives a +5 reward for every track tile touched with the applied action, as well as -0.1 no matter what. An optimal policy is expected to touch as many track tiles as fast as possible.

The car simulation is relatively sophisticated, it incorporates, for example, friction and inertia. It is possible to drift.

Task 5.4. *Implement DDQN and test it on environments.*

(a) Implement algorithm 5.3. The accompanying material contains a lot of help and many parts are already implemented or sketched.

7 We modified the environment, those measurements are now returned as numbers. Previously, they were added as a plot to the frame.

Algorithm 5.3 Double Deep Q-Learning

function TRAINDDQN($n_{\max}$, n_{replay}, n_{update}, n_{batch}, ϕ, ε_1, ε_∞, n_ε)
  Initialize the replay memory using a random policy.
  Initialize the target weights $\hat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}$.
  for step $i \leftarrow 1$ **to** $n_{\max}$ **do**
    $s \leftarrow$ current state, $a \leftarrow$ EPSILONPOLICY$(s, i, \varepsilon_1, \varepsilon_\infty, n_\varepsilon)$.
    Apply a, observe reward r and the new state s'.
    Store the transition $(\phi(s), a, r, \phi(s'))$ in the replay memory.
    if $i \equiv 0 \mod n_{\text{replay}}$ **then** ▷ Train with batch from memory.
      Sample a batch $(\phi_j, a_j, r_j, \phi'_j)_{i=1}^{n_{\text{batch}}}$ from the replay memory.
      $$y_j = \begin{cases} r_j & \text{if } \phi'_j \text{ is terminal,} \\ r_j + \gamma Q_{\hat{\boldsymbol{\theta}}}(\phi'_j, \arg\max_{a'} Q_{\boldsymbol{\theta}}(\phi'_j, a')) & \text{else.} \end{cases}.$$
      Perform a gradient descent step on
      $\frac{1}{n_{\text{batch}}} \sum_{j=1}^{n_{\text{batch}}} (y_j - Q_{\boldsymbol{\theta}}(\phi_j, a_j))^2$.
    end if
    if $i \equiv 0 \mod n_{\text{update}}$ **then**
      Update the target weights $\hat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}$.
    end if
  end for
  return $\boldsymbol{\theta}$.
end function
function EPSILONPOLICY(s, i, ε_1, ε_∞, n_ε)
  $\varepsilon \leftarrow \frac{\varepsilon_\infty(\min\{i,n_\varepsilon\}-1)+\varepsilon_1(n_\varepsilon-\min\{i,n_\varepsilon\})}{n_\varepsilon-1}$. ▷ Decrease ε linearly.
  return $\begin{cases} \text{random action} & \text{with probability } \varepsilon, \\ \arg\max_a Q_{\boldsymbol{\theta}}(\phi(s), a) & \text{else.} \end{cases}$
end function

(b) *Check your implementation with the mountain car environment. Pick a design for the Q-network, choose hyperparameters, train the agent. Visualize the final policy and report the reward. An implementation of the environment is included in the* GYM *package.*

(c) (***Optional bonus task***) *Do the same as in (b) for the car racing environment. To simplify the task, the racing track is fixed. You can change this by removing the random number generator seed.*

Attention!
The car racing environment needs a large replay memory (we tried 100k to 1M transitions), and at lot of steps (we got good results with 1M and more). So, on the hand you need roughly 10 GB of memory (a laptop might not have this much), and on the other hand, training will take several hours (easily 10h or more). The batch size, network size, and frequency of evaluations also has an influence. With other words: Do not start the evening before the deadline if you want us to see your results.

Hint!
The network architecture used in the DQN paper [6] is a CNN where the first hidden layer convolves 32 filters (size 8×8, stride 8) and ReLU. Then, 64 filters (size 4×4 , stride 2) plus ReLU follow. The third hidden layer consists of 64 filters (size 3×3, stride 1) plus ReLU. A fully-connected layer with 512 and ReLU is added before the output layer. The input was preprocessed by extracting the luminance from RGB and scaling it to 84×84. The appendix of [6] contains very detailed and nicely structure information which are invaluable for reproducing the results. In fact, the whole paper is excellent and reads nicely.

5.3.3 *Room for improvement*

Algorithm 5.3 was a milestone for reinforcement learning but it does not reflect the state-of-the-art for the Arcade Learning Environment. Similar to double Q-learning and prioritized experience replay, more modifications were proposed. In [5] several of those improvements were combined and their attribution tested. There, one major improvement was attributed to *n-step learning*: Instead of considering the reward of one step as a target, $y = r + \gamma V(s')$, n steps are used to compute y, $y = \sum_{i=1}^{n} \gamma^{i-1} \mathcal{R}(s_i, a_i) + \gamma^n V(s_{n+1})$. The idea is not new, instead it was adapted for deep Q-learning.

Sometimes so called *exploring starts* are employed. The idea is to start the environment not in its initial state, but to choose a random state. For the car racing environment, this could be restricted to start randomly on the track, or slightly off the track, or with different speed.

In the current state space of the car racing environment the agent can only see a portion of the track. Unless the track is fixed, it is likely that a better policy is possible if, for example, the next curve can be seen. This could be achieved by zooming out and increasing the resolution.

The action space for the car racing environment has to be discretized for DQN. The actual action space is continuous, so a policy which can use the full, continuous space is potentially better. There are reinforcement learning algorithms for continuous action spaces, for example the *policy gradient* methods [12], one recent example is PPO [10]. The basic idea of policy gradient methods is to approximate π using a, say, neural network $\pi_{\boldsymbol{\theta}}$. Then, a measure for the reward is maximized w.r.t. to $\boldsymbol{\theta}$.

5.4 FIXING WRONG IMPRESSIONS

After reading this chapter you might have thought the following statements are true. This section tries to prevent that.

- A value function does always exist, furthermore, it is unique and continuous. (none of those claims are true)

- There is always a *stationary* policy, which is optimal. (not true, cf. finite horizon problems)
- There is always a *deterministic* policy, which is optimal. (not true, cf. partially observed problems)
- The cost function has to be a *sum* of rewards. (no, but most common case, though)
- A Nash equilibrium does always exist and it is unique. (no)
- The mean-squared error is optimal. (not clear, in fact, DQN is using a clipped variant)
- There is a strong, theoretical analysis of popular reinforcement learning algorithms. (no, "more research is needed")
- You can compute state-of-the-art StarCraft policies on your laptop in a day. (no, it takes weeks or months on big, specialized clusters)

5.5 CLOSING REMARKS

In this chapter you got an impression of what reinforcement learning does and how. Reinforcement learning is mixture of different fields, for example supervised learning, game theory and planning.

Research is done to scale reinforcement learning by searching for parallel and distributed algorithms, i.e., to find ways such that an agent can be computed utilizing multiple machines (many CPUs, GPUs, etc.). Also, a major concern is sample efficiency: The enormous training steps do not seem justified, there is hope to lower the samples needed, which would make reinforcement learning also more available. Another reason to search for more sample efficient algorithms is that on real world environments one step is more costly compared to a simulation. There are several works which try to pretrain an agent using a simulation and transferring the knowledge to the real world, in those examples people use robotic arms.

In games, most recently an agent was presented to beat professional strategic video game StarCraft II players under somewhat realistic conditions. A real time strategy game like StarCraft II is considered harder than Go, for example: The more complicated partially observed state, the much bigger action space, the much higher number of steps for a typical game (60 frames per second vs. a dozen total moves in a Go game).

On the theoretic side, the books by Bertsekas are interesting [1, 2] (see also all his books), as well as the book by Szepesvári [13][8]. The

8 A regularly updated draft is available online at https://sites.ualberta.ca/~szepesva/papers/RLAlgsInMDPs.pdf.

same author publishes a freely available draft of a book[9] on n-armed bandits, which are closely related to reinforcement learning. More generally, the already mentioned book by Sutton and Barto is a classic [11][10] and recommended for the beginning.

REFERENCES

[1] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. 4th ed. Vol. 2. Athena Scientific, 2012.

[2] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. 4th ed. Vol. 1. Athena Scientific, 2017.

[3] Jochen Garcke and Irene Klompmaker. "Adaptive Sparse Grids in Reinforcement Learning." In: *Extraction of Quantifiable Information from Complex Systems*. Ed. by S. Dahlke, W. Dahmen, M. Griebel, W. Hackbusch, K. Ritter, R. Schneider, C. Schwab, and H. Yserentant. Vol. 102. Lecture Notes in Computational Science and Engineering. Springer, 2014, pp. 179–194. DOI: 10.1007/978-3-319-08159-5_9. URL: http://www.springer.com/mathematics/dynamical+systems/book/978-3-319-08158-8.

[4] Xianping Guo and Onésimo Hernández-Lerma. "Continuous-time Markov decision processes." In: *Continuous-Time Markov Decision Processes*. Springer, 2009, pp. 9–18.

[5] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. "Rainbow: Combining improvements in deep reinforcement learning." In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. "Human-level control through deep reinforcement learning." In: *Nature* 518.7540 (2015), p. 529.

[7] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[8] Jonathan Schaeffer. *One Jump Ahead. Challenging Human Supremacy in Checkers*. Springer-Verlag New York, 1997.

[9] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. "Prioritized experience replay." In: *arXiv preprint arXiv:1511.05952* (2015).

9 https://tor-lattimore.com/downloads/book/book.pdf

10 A regularly updated draft of the second version is available online at http://incompleteideas.net/book/the-book-2nd.html.

[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal policy optimization algorithms." In: *arXiv preprint arXiv:1707.06347* (2017).

[11] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*. 2nd. MIT Press, 2018.

[12] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. "Policy gradient methods for reinforcement learning with function approximation." In: *Advances in neural information processing systems*. 2000, pp. 1057–1063.

[13] Csaba Szepesvári. "Algorithms for reinforcement learning." In: *Synthesis lectures on artificial intelligence and machine learning* 4.1 (2010), pp. 1–103.

[14] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." In: *Thirtieth AAAI Conference on Artificial Intelligence*. 2016.