Prof. Dr. Jochen Garcke
Arno Feiden

5

REINFORCEMENT LEARNING

Send your solutions to this chapter's tasks until
July 24th.

Reinforcement Learning is the third main category in modern *artificial intelligence* (AI) conceptually in between supervised and unsupervised learning. Unsupervised learning describes techniques that work on some data X, while supervised learning describes techniques on labeled data (X, y). These labels helps the learner evaluate if it makes a correct or incorrect decision. Reinforcement learning is concerned with problems that come with a quantitative feedback, a measure of how good or bad a decision has been. It is influenced by ideas of Behaviorism that attribute learning to a feedback of positive and negative reinforcement.

Consider how a behaviorist *agent* may learn through interaction with its *environment*. They observe the world, interact with it in a meaningful way, and observe how their actions changed the world. But there also needs to be some kind of feedback to assess, if they "liked" doing what they did and are satisfied with the outcome. If they do, they might do it again. If not, they might avoid taking that action again or even try to avoid to end up in the situation that in which they took that action.

While it is extremely difficult to fully explain what drives the behaviour of a living being, we can sometimes observe that positive and negative reinforcement shapes it. For example when training an animal: When it does something we want it to do, we may use positive reinforcement like a treat to encourage this behaviour or negative reinforcement like shouting at it to discourage it. That is the kind of *reinforcement* that gives this technique its name. Past behaviour is reinforced by positive feedback or discouraged (negatively reinforced) by negative feedback.

Reinforcement learning has been prominently applied to games. A nice training ground for these algorithms have been a collection of Atari games. Take for example Ms. Pac-Man. At any point during the game, the actual *state* of the game is the RAM. The rendered frame on the screen is the *observation* an agent relies on. What happens next is mostly unknown to the agent, but it is guided by their *actions* and it is well defined by the software of the game. Reinforcement of good plays is given through the *reward* of the having a high score and the emotions that stirs in us. This agent is usually a human player. But since the actions taken in the game are defined by pressing a limited set of

(a) Ms. Pac-Man (b) StarCraft AlphaStar vs. Serral

Figure 5.1: Increasingly complex games are being tackled with reinforcement learning. For all Atari games instances with superhuman performances have been trained. In StarCraft, despite strong efforts, humans still dominate.

buttons, it can also easily be played by some algorithm that decides which virtual button to press next.

If we apply this framework to games like Chess or Backgammon that only end in a win, loss or draw, a positive point is given for a win and a negative point for a loss and everything else is neutral. One might be tempted to give feedback for subjectively good or bad events, like losing a valuable piece in chess, for example. This is called *reward engineering*. While most practical application make heavy use of this strategy, from a conceptual standpoint we want to avoid this: The agent is only told the ultimate goal, but not how to achieve it and it can come up with original strategies.

In most applications, the agent faces an enormously large environment with rewards "hidden" deep down a sequence of steps, similar to a maze. It is unknown where a decision will lead, especially if the environment is complicated or has random components. A decision that may lead to bountiful reward later, might yield no or even negative reward at the time of making it. Decisions need to be based not on the immediate reward but on their long-term *value*, or an estimation of it.

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 a long tradition. The chess program DeepBlue is very famous, another successful example is Chinook [7], 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.

More current developments are the attempt to train very efficient agents in newer video games such as StarCraft and DotA. DotA and the card game Hanabi are also of interest in the frontier of cooperative play of autonomous agents.

5.1 PROBLEM FORMULATION

5.1.1 *Markov decision process*

Reinforcement learning is concerned with the interaction of an *agent* and its *environment*. Based on the *observations* of the *state* of the environment, the agent takes an *action*, that in turn lead to a *transition* in the state of the environment and incurs a *reward*. This is formalized through the blueprint of the *Markov decision process*.

Attention!

Definition (Markov decision process). Let S be the state space, A the action space, $T : S \times A \to S$ the transition function, and $R : S \times A \to \mathbb{R}$ the reward function.

The tuple (S, A, T, R) is called a *Markov decision process* (MDP).

A mapping $\pi : S \to A$ is called an *agent* or a *policy* while the entirety of the MDP is called the *environment*.

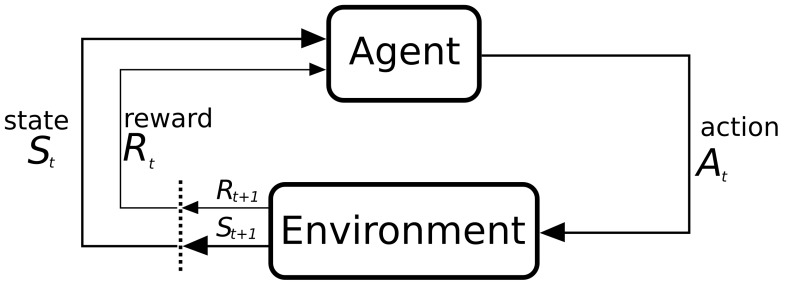
The transition and the agent are sometimes *stochastic*, i.e. non-deterministic.

The possible actions may be dependent on the state the environment is in.

If the transition function of a MDP is stochastic, the transition function maps every state-action pair (s, a) to a probability distribution $p_{(s,a)}(s', r)$ of target states s' and rewards r. A stochastic policy maps every state s to a probability distribution $\pi_s(a)$ of actions a. We might abuse notation to write those as $p(s', r|s, a)$ and $\pi(a|s)$.

If the possible actions are dependent on the state of the environment, for every $s \in S$, we have a set A_s of possible actions with transition and reward function defined accordingly.

Note, that the deterministic case is a special case of the stochastic case, by defining the probability measures as point-measures. Also trivially, the case of having a fixed action space A is a special case of having state-depended actions.



(a)

Figure 5.2: Sketch of a Markov decision process

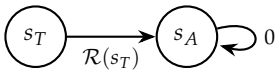


Figure 5.3: Adding a neutral, endless loop at a terminal state allows mathematically to aggregate the continuing case with the terminating case.

We call one "play-through" of a MDP an *episode* or if we want to explicitly mention the policy used a *rollout* of that policy. The resulting time-series of state-action pairs are the *trajectory* through the state-action space.

With this we implicitly assume that an MDP ends. This terminating case, in the contrast to the continuing case, is defined by the existence of *terminating states*. Mathematically, we aggregate terminating MDPs with continuing MDPs by adding a zero-reward dummy action to terminating states that does not change the state. From a software engineering standpoint every MDP terminates sometime, after some amount of steps regardless if a terminating state is reached.

We give these processes the “Markov” title, because we assume they fulfill the *Markov property*: That what happens next always only depends what is now, not how we got here. Specifically that means that the transition function T at time step k depends on the state-action (s_k, a_k) but not on any previous state or action. If that is always true in reality is more a philosophical question, but mathematically we can make any process “Markov” by simply adding to the current state any past events that are relevant for the now.

It can be useful to allow a single non-zero reward in a terminal state s_T. This reward is sometimes denoted by $R(s_T)$. This still fits into the model: Add a single dummy action in such a terminal state with reward $R(s_T)$, the next state s_A then always is an absorbing state with zero rewards. In this case the terminal and absorbing state are different.

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 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.

5.1.2 *Optimization and the Discount Factor*

An MDP implies an optimization problem to find a reward-maximizing policy π^*. We assume any MDP to be continuing, with terminal states having dummy actions. If we assume everything is deterministic, any policy π will rollout from starting state s_0 as a specific trajectory $(s_i, a_i)_{i\in\mathbb{N}}$ where $a_i = \pi(s_i)$ and $s_{i+1} = T(s_i, a_i)$. Conceptually, the value of a state for a policy π is the expected future reward starting from that state. Due to the Markov property, it does not matter how the agent environment arrived in that state. We may be tempted now to try to maximize the sum of rewards $\sum_{i\in\mathbb{N}} R(s_i, a_i)$ for any starting state.

It is easy to see, that even if R is bounded, this formulation may not by. To solve this problem we introduce the *discounting factor* $\gamma \in (0, 1)$ by which we reduce the value of future rewards. Because this is essential for the optimization problem, MDPs are sometimes defined as the 5-tuple (S, A, T, γ, R). For any policy π we can now define the *cumulative discounted (future) reward*

$$V^\pi(s_0) = \sum_{i\in\mathbb{N}} \gamma^i R(s_i, a_i). \tag{5.1}$$

Adding a discounting factor for future rewards has the nice side effect that in reality nothing goes on forever so we want to favour good things to happen rather sooner than later. In general, γ may be 1, if the MDP has a *finite horizon*, i.e. is guaranteed to finish by some mechanism and therefore V_π is always bounded.

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

$$\pi^* := \arg\max_\pi V^\pi(s) \tag{5.2}$$

for all $s \in S$. The cumulative discounted future reward of an optimal policy π^* is called the *(optimal) value function* and is denoted by $V(s) := V^{\pi^*}(s)$.

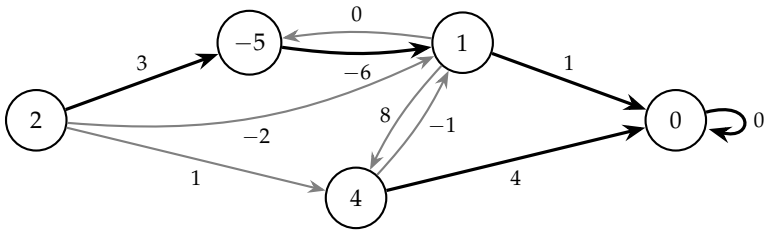


Figure 5.4: A abstract example for a deterministic environment. The environment is represented by a graph, every state is a node, every action an edge. The reward is written on the edges. A policy is visualized by bold edges. For this policy, the value V^π is denoted inside the nodes (with $\gamma = 1$).

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.

5.1.3 *Bellman Optimality Equation*

From the general definition of cumulative discounted reward (eq. (5.1)), we can see it is formulated in a recursive way. Split up the sum into the first term and the remaining sum is itself again a representation of the value function:

$$\begin{aligned} V^\pi(s_0) &= \sum_{i=0}^{\infty} \gamma^i R(s_i, a_i) \\ &= R(s_0, a_0) + \sum_{i=1}^{\infty} \gamma^i R(s_i, a_i) \\ &= R(s_0, a_0) + \gamma V^\pi(s_1) \end{aligned}$$

This yields the recursive formulation of cumulative discounted reward of π:

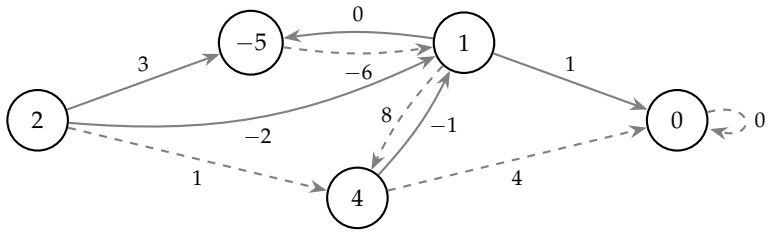
$$V^\pi(s) = \big(R(s, a) + \gamma V^\pi(s')\big), \tag{5.3}$$

where $a = \pi(s)$, and $s' = T(s, a)$. If π is an optimal policy, this equation for the value function is called the *Bellman (Optimality) Equation*:

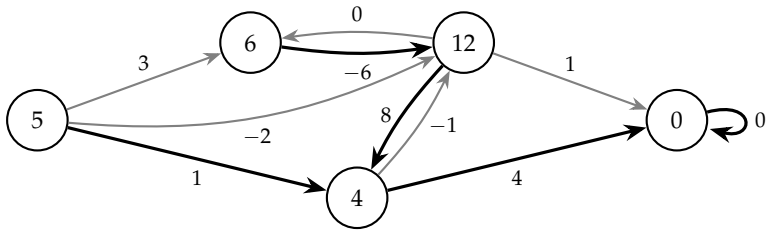
$$V(s) = \big(R(s, a) + \gamma V(s')\big), \tag{5.4}$$

Specifically, this yields an equation for the optimal policy in terms of the value function:

$$\pi^*(s) := \arg\max_a \big(R(s, a) + \gamma V^*(s')\big). \tag{5.5}$$



(a) Policy improvement.



(b) Policy evaluation.

Figure 5.5: A step of policy improvement and evaluation for the example from fig. 5.4. The new action is shown using a dashed edge. The next iteration will yield an optimal policy.

The non-deterministic formula looks a little more complicated, but we just take the expected value over the uncertainties:

$$V^*(s) := \max_a \sum_{s',r} p(s', r|s, a)\big(r + \gamma V^*(s')\big), \tag{5.6}$$

5.1.4 *Iterative Policy Evaluation and Value Evaluation*

Algorithm 1 Iterative Policy Evaluation (see [10])

function PolicyEvaluation($(S, A, T, \gamma, R), \pi, \delta > 0$)
  Initialize $V, V' : S \to \mathbb{R} \cong \mathbb{R}^{|S|}$ arbitrarily, but $V(s_{\text{term}}) \leftarrow 0$.
  repeat
    $V' \leftarrow V$
    for $s \in S$ **do**
      $V(s) \leftarrow R(s, \pi(s)) + \gamma V'(T(s, \pi(s)))$
    end for
  until $\max_{s \in S} |V - V'| \leq \delta$
  $V_\pi \leftarrow V$
  return V_π
end function

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*.

In algorithm 1 we compute the value function V^π for the current policy π. This is called *policy evaluation*. By repeatedly applying the Bellman equation to the whole state space of a value function of a policy π, the information of good and bad moves propagates through the whole space.

In algorithm 2 we both compute the value function for the current policy π and π is the most optimal policy assuming V is optimal. This iteratively moves V to V^* and π to π^*.

From 5.3 and 5.4, one can see that V^π and V are fixed points of the operators applied in line 6 of both algorithms respectively. If this operator is also a contraction, we get from Banach's fixed-point theorem that there exists a unique fixed-point which can be computed through fixed-point iteration. For finite state spaces S and finite action spaces A, under certain condition e.g. $\gamma < 1$ this can be shown. See fig. 5.5 for an example of value iteration.

Algorithm 2 Value Iteration (see [10])

1: **function** ValueIteration($(S, A, T, \gamma, R), \delta > 0$)
2: Initialize $V, V' : S \to \mathbb{R} \cong \mathbb{R}^{|S|}$ arbitrarily, but $V'(s_{\text{terminal}}) \leftarrow 0$.
3: **repeat**
4: $V \leftarrow V'$
5: **for** $s \in S$ **do**
6: $V'(s) \leftarrow \max_{a \in A_s} R(s,a) + \gamma V(T(s,a))$
7: **end for**
8: **until** $\max_{s \in S} |V - V'| \leq \delta$
9: $\pi^* \leftarrow \left\{ s \to \arg\max_{a \in A_s} \left(R(s,a) + \gamma V(T(s,a)) \right) \right\}$
10: **return** π^*
11: **end function**

Nitpicking

The algorithms 1 and 2 are formulated with both deterministic environment and policy in mind. For the general case we have to account for the uncertainty. Replace line 6 in algorithm 1 with

$$V'(s) \leftarrow \sum_a \left(\pi(a|s) \sum_{(s',r)} p(s',r|s,a)(r - \gamma V(s')) \right)$$

and in algorithm 2 with

$$V'(s) \leftarrow \max_a \left(\sum_{(s',r)} p(s',r|s,a)(r - \gamma V(s')) \right)$$

Where $\pi(a|s)$ denotes the probability that π picks action a given state s and $p(s', r|s, a)$ denotes the probability of taking action a in state s to result in arrive in state s' with reward r under the dynamics of the environment.

5.1.5 *General remarks*

Value and policy iteration are basic building blocks for reinforcement learning algorithms. They themselves 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. 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 function approximation in order to represent V^π or π using samples. We will see an example for such a method later. For linear function approximations there is theory available reaching back to the 80s.

Hint!

An avid scholar of reinforcement learning techniques may recognize degenerate Markov decision processes in a lot of decision processes. An important class of decision problems are those, where there is only one state, one timestep, or actions do not change the state. This simplifies the problem drastically and is researched under the concept of n-bandits - Like a one-armed bandit but with n arms, where each arm represents a possible action and may result in different outcomes. Consider for example a recommendation algorithm for a music playlist: We get one shot of recommending a song, we want the listener to not skip or turn off the music, and recommending a song does not (immediately) change the listeners taste in music. If we additionally know some information about the listener, this could be described as a *contextual bandit*. Refer to "Bandit Algorithms" [a], a freely available, exhaustive, and informative book for more information on n-armed bandits.

[a] https://tor-lattimore.com/downloads/book/book.pdf

5.1.6 *Tasks*

Task 1. *Get familiar with gym and the ideas of value iteration. Refer to the* Jupyter notebook*!*

(a) *Experimentally estimate the expected value of the random policy on the FrozenLake-v1 environment with* `is\_slippery` *first* `False` *then* `True`. *What do you observe and why?*

(b) *Implement the missing piece of iterative policy evaluation to calculate the value function for the random policy and $\gamma = 0.9$ on the FrozenLake-v1 environment with* `is\_slippery` *is* `False`*. What is the value function if* `is\_slippery` *is* `True`*?*

(c) *Implement the missing piece of value iteration to calculate an optimal policy for the FrozenLake-v1 environment where* `is\_slippery` *is* `False`*. Experimentally estimate the expected return for these optimal policies.*

5.2 CLASSIC REINFORCEMENT LEARNING

Artificial intelligence promises to cope with problem that are large, complex, and unclear. So far, we have dealt with problems that are so simple we can understand them completely. But even in a moderately simple game like checkers, the space of all conceivable states is enormous. The transition function is hard to pin down, because it is influenced by the move the opponent makes. In general, the dynamics of a problem are unknown.

In line with the conceptual idea of mimicking learning through positive and negative reinforcement, we let the agent learn from experience. We look at the *Temporal Difference* (TD) approach, because it has been historically an important milestone and it serves as an instructive example.

5.2.1 *Q-Function for State-Action Value*

To mitigate the problem of unknown dynamics of the system, one approach is to store the value for state-action pairs (s, a), instead of storing the value only for states. Given just the value function V, the value of an action is determined by the incurred reward and the resulting state. Considering state-action pairs allows the agent to estimate the value of an action a given state s even if we do not have a predefined notion which state will occur after taking an action. We define for a policy π the corresponding action-value function Q_π, its *Q-function*, as:

$$\begin{aligned} Q^\pi \colon \{(s,a) \mid s \in S, a \in A_s\} &\to \mathbb{R} \\ (s_0, a_0) &\mapsto R(s_0, a_0) + \sum_{i=1}^{\infty} \gamma^i R(s_i, \pi(s_i)), \end{aligned}$$

Just as for the value function 5.1, for simplicity we define this equation first for a deterministic case, where $(a_i, s_i)_{i \in \mathbb{N}}$ is the trajectory of π starting in (s_0, a_0) through the state-action space. The state-action values $Q(s, a)$ are sometimes called Q-values.

Now if π^* is an optimal policy, then we call $Q := Q^* := Q^{\pi^*}$ *the Q-function* (of the MDP) and we can induce the value function from it:

$\max_{a \in A_s} Q(s,a) = V(s)$ and an optimal policy can then be obtained by

$$\pi^*(s) := \arg\max_{a \in A_s} Q(s,a). \tag{5.7}$$

Compared to eq. (5.5) we do no longer need R or T here, which represent the dynamics of the system. The drawback is that more values need to be stored.

Again, we can find a recursive relationship between a policy π and the associated Q-function:

$$Q^\pi(s,a) = R(s,a) + \gamma Q^\pi(s',a'), \tag{5.8}$$

where $s' = T(s,a)$ and $a' = \pi(s')$. We utilized both these equations to formulate iterative optimization schemes in the similar to the idea of value iteration (2). Now we first note, that we cannot iterate over every state, but instead learn from experience. So instead of iterating over every state, we step by step gain new information by interacting with the environment:

1. In state s we take an action a and observe a reward r and the new state s'.
2. Based on $r + \gamma \max_{a' \in A_{s'}} Q(s',a')$ modify our idea of $Q(s,a)$.

By tuning $Q(s,a)$ to the experienced $r + \gamma \max_{a' \in A_{s'}} Q(s',a')$, we push our estimate of Q toward the Q-function of the optimal policy.

5.2.2 *Sarsa*

Sarsa or SARSA stands for state-action-reward-state-action. And refers to the five variables, s, a, r, s', and a'. It follows that described idea of iteratively learning Q from experience. In each learning step, our knowledge of $Q(s,a)$ improves through the incurred reward r, our current estimate of the value of the next state s', and the next action a'. We call the estimation of on value by neighbouring values *bootstrapping*.

Sarsa is a *Temporal Difference* algorithm. Whenever we get a new estimate for $Q(s,a)$ from experience, instead of overwriting, we nudge the current estimate toward the observed value. So for $\alpha \in (0,1)$, this is

$$\begin{aligned} Q_{\text{new}} &\leftarrow (1-\alpha) Q_{\text{old}} + \alpha Q_{\text{observed}} \\ &= Q_{\text{old}} + \alpha (Q_{\text{observed}} - Q_{\text{old}}). \end{aligned}$$

This difference $(Q_{\text{observed}} - Q_{\text{old}})$ is the eponymous temporal difference.

The goal of Sarsa is learning a Q-function from which we derive the estimate of an optimal policy. We call Sarsa an *on-policy* algorithm,

because we always keep our current estimate of this optimal policy and use it to generate experience and to bootstrap.

It is not necessary that the policy that generates the experience and the policy being learned are identical. However, in general a strong alignment between those two improves learning: It is easier to learn something, if you try it yourself, then just by watching someone else do it. It enables you to challenge your own believes, try out paths you deem promising, and get more experience in those situations that are relevant to your behaviour.

Algorithm 3 Sarsa (see [10])

function SARSA($(S, A, T, \gamma, R), N > 0, \alpha$)
  Initialize $Q(s, a)$ arbitrarily, but $Q(s_{\text{term}}, a) \leftarrow 0$.
  for $i \in \{0, ..., N\}$ **do**
    Initialize a starting state $s \in S$ randomly.
    a is determined for s from Q
    repeat
      Take action a, observe reward r and next state s'
      a' is determined for s' from Q
      $Q(s, a) \leftarrow Q(s, a) + \alpha\left(r + \gamma Q(s', a') - Q(s, a)\right)$
      $s \leftarrow s'; a \leftarrow a'$
    until s is terminal
  end for
  Policy π is derived from Q
  $Q^\pi \leftarrow Q$
  return π, Q^π
end function

Finally we note that this algorithm is open-ended and we need to provide an end condition to define when learning is finished. For our purpose, we simply define a runtime of N rollouts of the policy.

Hint!

The statement "a' is determined for s' from Q" in 3 is a bit vague and will repeat in similar forms in later algorithms. We imagine most of the time that this is done by the *greedy* rule 5.7. For most practical applications however, we need to adjust this rule to be less greedy and more curious. The specific implementation may vary. More on that in the later paragraph on "Exploration-Exploitation".

5.2.3 *Q-Learning*

While very similar to Sarsa, this is classified as an off-policy algorithm. It is virtually the same as Sarsa, with a subtle difference in the temporal difference. The bootstrapping is not done by the actually chosen a' of the current policy but rather by the most promising a' derived from the current estimate of the Q-function.

Algorithm 4 Q-Learning (see [10])

function Q_LEARNING($(S, A, T, \gamma, R), N > 0, \alpha$)
  Initialize $Q(s, a)$ arbitrarily, but $Q(s_{\text{term}}, a) \leftarrow 0$.
  for $i \in \{0, ..., N\}$ **do**
    Initialize $s \in S$
    repeat
      a is determined for s from Q
      Take action a, observe reward r and next state s'
      $Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_a Q(s', a) - Q(s, a)\right)$
      $s \leftarrow s'$
    until s is terminal
  end for
  $Q^* \leftarrow Q$
  return Q^*
end function

This is a meaningful distinction. If we greedily pick the most promising option during learning, we might not find the actual best option, because we never try new pathways. We usually address this problem, by deriving policies from Q that are a little suboptimal in relation to Q in favour of trying new things.

5.2.4 *Exploration-Exploitation*

Coupling the behaviour of the policy to the exploration of possible state-action pairs and their consequences is in general not necessary to find an optimal policy or the optimal Q or V function. It is very practical as it focuses the limited learning resources on those situations that happen often and naturally ignores imaginable but impossible states.

If we imagine an agent that keeps on learning forever, we want them to incur good rewards as soon as possible while continuing to learn. 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 in a *greedy* manner, using the maximum, we could miss a path for a more lucrative action. This problem of balancing curiosity and productivity is called the trade-off or dilemma of *exploration* and *exploitation*.

From a theoretical standpoint, we even need to guarantee that in an infinity amount of training steps, every state-action pair is visited an infinity amount of time to derive convergence.

We will use the so called *ε-greedy* policy, which picks a random action with probability $\varepsilon > 0$, and a maximizing action otherwise. Of course this means, that throughout the training process we will estimate the Q-function of a suboptimal policy. So sometimes $\varepsilon \to 0$ during the training process. We give two more, less frequently used examples:

Optimistic initialization initializes Q with high values, favouring actions that have never or rarely been picked before. This requires some knowledge about the MDP, specifically what value of Q is appropriately high to balance exploration and exploitation.

Softmax exploration replaces the arg max to pick an action with drawing from the *softmax* function

$$\sigma : \mathbb{R}^n \to \mathbb{R}^n$$
$$\sigma(z)_i = \frac{e^{z_i}}{\sum_i e^{z_i}}.$$

The softmax function takes any vector in $\mathbb{R}^n$ and transforms it into a vector in $(0,1)^n$ that sums to one, preserving the ordering. This vector is than treated as a probability distribution from which we can draw the index of the action.

Decaying ε strategies use ε-greedy policies that start the learning process with a high ε value that decays over the course of training. This is one of the most popular approaches, as it is easy to implement and very powerful.

5.2.5 *Binning*

We have been exploring discrete state- and action-spaces. In the next chapter we will progress into continuous spaces as well, but first we introduce a simple technique to discretize continuous spaces: *binning*.

Binning refers to diving up a space into cells, that represent all the continuous values that fall into that cell. So we might discretize the space $[0,1]$ into $D := \{[0,0,1], [0.1,0.2), ..., [0.9,1]\}$.

After binning the state-space, we define a discrete Q-function on those bins. To evaluate a value we would lookup the value of the appropriate bin, e.g. to evaluate 0.23, we take the value of the third bin $[0.2,0.3)$. Discrete Q-functions are often called *Q-table* to clearly differentiate the discrete and continuous approach.

In the same way, we can discretize a higher dimensional state space $[0,1]^2$ into 10 times 10 bins $D \times D$ and so forth. Each additional dimension increases the number of bins exponentially, so while this concept is powerful for simple problems, it is not viable for higher dimensional problems. This is one instance of the *curse of dimensionality*, incidentally a phrase coined by Bellman. [1]

5.2.6 *Tasks*

Task 2. *Experiment with the classic reinforcement learning algorithms introduced in the chapter. Refer to the* Jupyter notebook*!*

(a) Implement Sarsa or Q-Learning to solve the FrozenLake-v1 environment with `is\_slippery=True` *with* $\gamma = 0.9$*. Use an* ε*-greedy policy with* $\varepsilon = 0.1$*! Experimentally estimate the expected value of this policy.*

(b) *Create the CartPole-v1 environment. Get familiar with it by implementing a random policy for it and rendering a rollout of the random policy. Render might not work in Jupyter. If so, use a python script!*

(c) (***Master students only***) *Adapt the algorithm from (a) to solve the CartPole-v1. Use binning to discretize the state-space. Experimentally estimate the expected return for the learned policy and render one rollout.*

5.3 DEEP REINFORCEMENT LEARNING

5.3.1 *Introduction*

Deep neural networks and the powerful hardware to train them is one of the main driver of the 21st centuries hype around artificial intelligence. Neural networks can potentially estimate any function and we have the means to find those representations. It is not surprising, that the currently rising interest in reinforcement learning is highly inspired by deep neural networks.

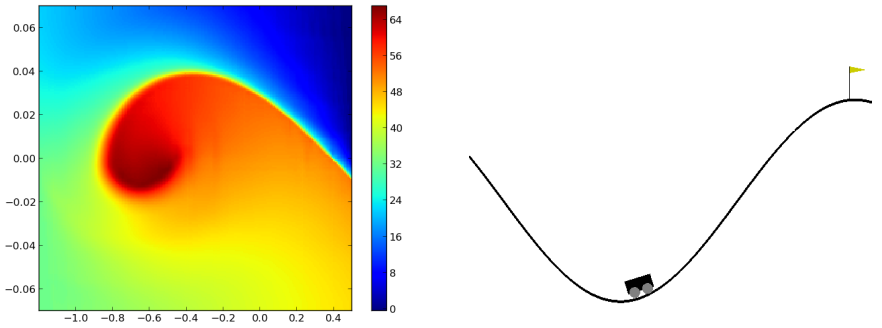
The term *deep reinforcement learning* is vague, but its minimal building blocks are a deep neural network that get to work on a Markov decision process. The first, really impressive result, has been the effective use of *deep Q-learning* to learn to play Atari games on a level of (super-)human capability. [6]

Bots that play games are not a new invention, so let us recap, why this is remarkable: The only feedback the learner gets, is the score. The learner has the same inputs and output options as a human, "looking" at just the screen. The same architecture is used for any game. This learning process resembles a real, independent learning process on a complex, big, and uncertain problem. Usually, bots would be programmed to do specific actions in specifics situations and use internal data to understand the state. We did the same in the CartPole-v1 environment where we use the "internal states" position and velocity, not the screen as input.

In "Human-level control through deep reinforcement learning"[6], the authors managed to find a single deep neural network architecture and a reinforcement learning algorithm which could achieve superhuman performance on 49 Atari games. They fixed hyperparameters, like the step size, the ε used for the ε-policy, the batch size, training steps, or prepossessing, while the weights of the network were trained new for each game.

As a true reinforcement learning problem, no rules of the game (system dynamics) were available to the agent. In fact, a state consisted only of the last four frames[1] of the game, so the agent sees a visual representation of the game. Besides replacing the $Q - table$ with a deep

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



(a) Value function (b) Initial state

Figure 5.6: The value function of the mountain car environment (computed with sparse grids [4]) 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 $[-0.07, 0.07]$.

neural network and including all the bookkeeping that is required to do that, some technical modifications were made to make the algorithm work. We will describe the basic ideas of this *DQN (Deep Q-network)*, the resulting algorithm as an introduction to the modern approach to deep reinforcement learning.

5.3.2 *Continuous Q-Function*

The straight-forward way to define a Q-function with a continuous state-space is of course to work with a continuous function. We would then approximate Q by a function Q_θ which is parameterized by a finite set of real values $\theta \in \mathbb{R}^m$ and chosen to be easy to store and to evaluate. This has the inherent advantage, that binning is not necessary anymore. In some situations it but it still may prove to be advantageous, but in general it is very beneficial to represent a continuous state(-action) space through a continuous input.

Take for example the linear approximation

$$Q_\theta(\cdot, \cdot) := \sum_{k=1}^{m} \theta_k f_k(\cdot, \cdot)$$

for a set of (feature) maps f_k. Feature maps can be projections, polynomials, Fourier basis functions or piece-wise constant functions. Optimally, they are easy to evaluate and represent relevant information for the computation of the value

If we try to adjust an algorithm like Sarsa to fit this paradigm, the update of Q needs to be adapted for Q_θ, because we can only change the Q_θ by changing θ. While we could try to find θ such that Q_θ is equal to the new value at (s, a), this might significantly worsen the approximation at other points. Instead, we try to find $\arg\min_\theta \|Q - Q_\theta\|$ for some norm. With every experience from the MDP, our idea of

what this arg min is, changes. An update of θ based on a gradient is natural: We could update θ by

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

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

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

Notice that we only need to store the vector θ compared to $|S||A|$ values before. Also infinite state spaces S are now allowed.

For linear approximations convergence and error bounds on $\|Q - Q_\theta\|$ in suitable norms can be shown under certain assumptions, while nonlinear approximations are tricky ([10]).

Another approach is to represent Q_θ as a function $Q_\theta : S \to \mathbb{R}^{|A|}$. There, each state is mapped to a vector of all its Q-values.

Algorithm 5 Episodic Semi-gradient Sarsa (see [10])

function GRADSARSA($(S, A, T, \gamma, R), N > 0, \alpha$)
  Initialize $\theta \in \mathbb{R}^m$ arbitrarily.
  for $i \in \{0, ..., N\}$ **do**
    Initialize $s \in S$
    a is determined for s from Q_θ
    repeat
      Take action a, observe reward r and next state s'
      if s' is terminal **then**
        $\theta \leftarrow \theta + \alpha\,(r - Q_\theta(s, a))\,\nabla Q_\theta(s, a)$
      else
        a' is determined for s' from Q_θ
        $\theta \leftarrow \theta + \alpha\,(r + \gamma Q_\theta(s', a') - Q_\theta(s, a))\,\nabla Q_\theta(s, a)$
        $s \leftarrow s'; a \leftarrow a'$
      end if
    until s' is terminal
  end for
  $Q^* \leftarrow Q$
  return Q^*
end function

5.3.3 *Deep Q Networks*

The basic idea of Deep Q Networks (DQN) is to represent the Q-function with a deep neural network instead of a table or a sum of basis functions.

For a theoretical discussion of this deep learning approach, we fix the architecture of the neural network, i.e. the nodes, edges, activation

functions, and only think about the weights θ that are optimized during training and the neural network Q_θ as an approximation of the optimal Q-function.

We can see, that the algorithm 6 uses the same basic building blocks as for the linear functions. We do need to make at least two basic additions to make it work though: The gradient descent step is only done every n_{replay} steps and the neural net is trained from a database of past steps, called *replay memory*.

Algorithm 6 Deep Q-Learning

function TRAINDQN((S, A, T, γ, R), N, n_{replay}, n_{batch}, ε)
  Initialize $\theta \in \mathbb{R}^m$ arbitrarily.
  for $i \in \{0, ..., N\}$ **do**
    Initialize $s \in S$
    a is determined for s from Q_θ
    Apply a, observe reward r and the new state s'.
    Store the transition (s, a, r, s') in the replay memory.
    if $i \equiv 0 \mod n_{\text{replay}}$ **then** ▷ Train with batch from memory:
      Sample a batch $(s_j, a_j, r_j, s'_j)_{j=1}^{n_{\text{batch}}}$ from the replay memory.
      $$y_j = \begin{cases} r_j & \text{if } s'_j \text{ is terminal,} \\ r_j + \gamma \max_{a'} Q_\theta(s'_j, a') & \text{else.} \end{cases}.$$
      Perform a gradient descent step on
        $||y_j - Q_\theta(s_j, a_j)||$.
    end if
  end for
  return θ.
end function

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*. Instead of training with the loss $\frac{1}{2}(y - Q_\theta(s, a))^2$, we draw n_{batch} uniformly distributed samples $(s_i, a_i, r_i, s'_i)_{i=1}^{n_{\text{batch}}}$ from the replay memory. The batch size $n_{\text{batch}} \in N$ is fixed previously. Then the targets y_i are computed for each transition (s_i, a_i, r_i, s'_i). As above, we could use $y_i := r_i + \max_{a' \in A_s} Q_\theta(s', a')$. Finally, we perform a gradient descent step using the loss

$$\frac{1}{n_{\text{batch}}} \sum_{i=1}^{n_{\text{batch}}} (y_i - Q_\theta(s_i, a_i))^2. \tag{5.9}$$

Nitpicking

Instead of drawing the samples uniformly, a subsequent paper [8] proposes to draw the samples using, for example, $|y_i - Q_\theta(s_i, a_i)|$ as weights. With a

weighted sampling, a batch is more likely to contain transitions where Q_θ has a high error. This method is called *prioritized experience replay*.

DECOUPLED TARGETS A useful addition not mentioned in 6 is decoupling targets to counter oscillations. The targets y are not computed using Q_θ, but instead a second independent instance of *target weights* $\hat{\theta}$ is used:

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

So we have one network Q_θ used for selecting the action to apply in order to obtain a new transition, and a second *target* network of identical shape but with weights $\hat{\theta}$ which is used to compute the targets.

Each time after a previously fixed number of steps n_{update} the weights are copied $\hat{\theta} \leftarrow \theta$. The weights $\hat{\theta}$ are not affected by gradient descent steps done on θ.

5.3.4 *Tasks*

Task 3. *Experiment with the advanced reinforcement learning algorithms introduced in the chapter. Refer to the* Jupyter notebook*!*

(a) Adapt the Q-learning or Sarsa algorithm from before to solve the CartPole-v1 environment. Use linear functions to represent $Q_\theta(., a) = \sum_i \theta_i s_i$ to represent the value of each action a. Experimentally estimate the expected return for the learned policy and render one rollout.

(b) Adapt the Q-learning or Sarsa algorithm from before to solve the CartPole-v1 environment. Use a deep neural network that maps to a vector for action-values. Experimentally estimate the expected return for the learned policy and render one rollout.

*(c) (**Master students only**) Pick another environment and try to solve it using the algorithm developed in (b). You may need to change the hyperparameters or implement additional tweaks like a decaying ε-strategy or decoupled targets.*

5.3.5 *Further Developments*

DQN was a milestone for reinforcement learning but it does not reflect the state-of-the-art for the Arcade Learning Environment. Similar to 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.

There are reinforcement learning algorithms for continuous action spaces, for example the *policy gradient* methods [11], one recent example is PPO [9]. The basic idea of policy gradient methods is to approximate π using a, say, neural network π_θ. Then, a measure for the reward is maximized w.r.t. to θ.

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 number of training steps does not seem justified, there is hope to lower the samples needed, which would make reinforcement learning also more applicable in non-simulated environments like robotics, where 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.

We can get an interesting feel for the behaviour of reinforcement learning agents by watching them play, especially through the eyes of excellent commentators Daniel King [2] or Artosis. [3]

On the theoretic side, the books by Bertsekas are interesting [2, 3] (see also all his books), as well as the book by Szepesvári [12][4]. More generally, the already mentioned book by Sutton and Barto is a classic [10] [5] and recommended for the beginning.

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)
- The mean-squared error is optimal. (not clear, in fact, DQN is using a clipped variant)

2 https://www.youtube.com/watch?v=Sv-PJhwHehg
3 https://www.youtube.com/watch?v=nbiVbd_CEIA
4 A regularly updated draft is available online at https://sites.ualberta.ca/~szepesva/papers/RLAlgsInMDPs.pdf.
5 Second edition from 2018 available at http://incompleteideas.net/book/the-book-2nd.html.

- 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)

REFERENCES

[1] R. Bellman and Rand Corporation. *Dynamic Programming*. Rand Corporation Research Study. Princeton University Press, 1959. URL: https://books.google.de/books?id=Pfgtj_klGsoC.

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

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

[4] 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.

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

[8] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. "Prioritized Experience Replay." In: *CoRR* abs/1511.05952 (2016).

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

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

[11] 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.

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