Ludwigs-Maximilians-Universität München
Institut für Statistik

Reinforcement Learning in R

Markus Dumke

München, 02. November 2017

Supervision: Prof. Bernd Bischl, Xudong Sun

Abstract

Reinforcement learning is a class of algorithms to solve sequential decision making problems. While there are many implementations available in Python, there are nearly no algorithms available in R. This thesis introduces the `reinforcelearn` package, which aims to make a range of important reinforcement learning algorithms available for R users.

The first section describes the reinforcement learning problem and the corresponding notation, the second section explains algorithms to solve reinforcement learning problems, the third section then introduces the R package `reinforcelearn` and how to use it.

Furthermore we introduce Double Q(σ) and Q(σ, λ), two new reinforcement learning algorithms which subsume Q-Learning, Sarsa and Q(σ).

Contents

1 Introduction to Reinforcement Learning

Machine learning is a field in the intersection of statistics and computer science addressing the problem of automated data-driven learning and prediction. Machine learning techniques can be categorized into three different subfields: supervised learning, unsupervised learning and reinforcement learning. In *supervised learning* the goal is to learn the relationship between a set of input variables (also called features or covariates) and an output variable (also called label or target variable) to predict the output variable with high accuracy using the features. These classification or regression models are usually trained on a fixed predefined data set, which provides known values of inputs and outputs, often under the assumption of independent and identical distributed (i.i.d.) training examples. In *unsupervised learning* there is no output variable, instead the goal is to detect hidden structure between input variables, e.g. by clustering or density estimation.

1.1 What is Reinforcement Learning?

Reinforcement learning is different from both supervised and unsupervised learning. It is about sequential decision making and can be formulated as an interaction of an *agent* and an *environment* over a number of discrete time steps. The agent is a goal-driven learning algorithm. Everything outside the agent's control is considered as part of the environment, so the border between agent and environment is not necessarily a physical border, e.g. between a robot and its surroundings, but can lie inside the robot itself, i.e. the agent can be a part of a larger control system. Usually the agent has incomplete knowledge about the environment and tries to learn the best way to behave in this environment. The approach to learn from direct interaction with the environment, is very similar to learning of humans and animals (Sutton and Barto 2017).

At each time step t the agent chooses an *action* $A_t \in \mathcal{A}(S_t)$ and the environment subsequently returns a *state* observation $S_{t+1} \in \mathcal{S}$ and *reward* $R_{t+1} \in \mathbb{R}$ (Figure 1.1).[1]

The stream of data consists of a sequence of states, actions and rewards. The state observation provides some information about the problem, e.g. information about the

[1]Some authors denote the next reward as R_t instead of R_{t+1} and the next state S_t instead of S_{t+1}. This is just a difference in terms of notation. The notation in this thesis will be the same as the notation in Sutton and Barto (2017).

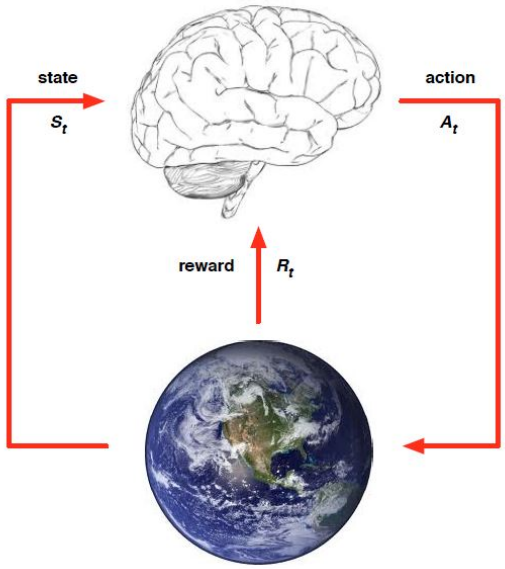


Figure 1.1: Environment agent interaction: The agent is symbolized as a brain, the environment as a globe. Silver (2015)

position of all pieces on a chess board. The agent can then build its decision making on this state context. The actions are the possible decisions an agent can make depending on the current state of the environment. The reward is a scalar number, which quantifies, how good an action is in an immediate sense. Low or negative rewards correspond to bad actions and can be interpreted as a punishment, high positive rewards to good actions. Reinforcement learning is completely based on rewards, this way an agent may learn to choose actions yielding higher rewards over actions leading to low rewards. Rewards are often sparse or delayed, for example in a game like chess the only reward is obtained at the end of the game, whether the game has been won or not and the reinforcement learning agent needs to figure out which of the moves were important for the outcome of the game. So the reward of an action might be returned long time after the action was taken and often underlies stochasticity. States and rewards are considered as part of the environment, so the agent can therefore influence these solely by its actions.

In comparison to supervised learning there are no labels telling the agent the correct action at each time step, instead it has to learn from the sequence of past transitions (states, actions, rewards), how to receive higher rewards. There is no fixed data set in the beginning, instead data is gathered over time and the agent can influence the

subsequent data it receives with its actions. For example a robot wandering around with a camera would receive a different image of its surrounding, whether it turns left or right. The camera image is in this case the state observation returned from the environment and turning left or right could be two possible actions. The data received at each point in time is usually highly correlated with previous data, e.g. an image obtained by a camera is probably very similar to the image a second before. Therefore methods working only with i.i.d. data cannot be applied. Reinforcement learning is also different from unsupervised learning because it is not used to detect hidden structure in the data, but to solve a clearly defined optimization problem.

Policy

The agent acts at each time step according to a *policy* π, a mapping from state to actions,

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]. \tag{1}$$

Policies can be deterministic or stochastic and describe a probability distribution over actions. So for each state the policy describes the probabilities of taking each action. The agent tries to adapt its policy over time, so to obtain higher rewards. The agent's actions influence not only the next reward and state observation, but also subsequent states and rewards. Therefore the goal is to maximize not only the immediate reward, but the cumulative reward over the life-time of the agent, which is called the *return*.

Return

The return is defined by

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \qquad \gamma \in [0, 1]. \tag{2}$$

where γ is called the discount factor. It is usually considered as part of the problem and not a tuning parameter of the algorithm. A reward k steps in the future is worth only $\gamma^{k-1} R_{t+k}$, so discounting controls how far-sighted the agent is. For a value of 0 the

return is equal to the immediate reward, so the agent only cares about the immediate reward, when γ approaches 1, the agent learns to act more far-sighted. Often future rewards are discounted with $\gamma < 1$, so that rewards in the far future contribute less than rewards now. This is analogous to human and animal behavior, which shows preferences for immediate reward compared to delayed reward. Mathematically discounting prevents infinite returns if the time horizon is infinite as later rewards get a smaller and smaller weight in the sum.

Exploration-Exploitation trade-off

One important aspect in reinforcement learning is the trade-off between exploration and exploitation, which does not appear this way in supervised or unsupervised learning. Because the agent has incomplete knowledge about the environment it needs to explore which actions are good, e.g. by taking a random action. On the other hand it needs to exploit the current knowledge to obtain as much reward as possible. Both goals cannot be achieved at the same time, so the agent faces a dilemma. When exploring, e.g. by taking a random action, it sacrifices immediate reward, but learns more about the environment, so it can obtain higher rewards later. When exploiting, i.e. taking the action, which currently looks best, it probably acts suboptimal, because there might be an unexplored action, which yields higher rewards. As an analogy to human life you can think of exploration as trying out a new restaurant and exploitation as visiting your favorite restaurant (Silver 2015).

Higher exploration in the beginning is often useful because the agent will gain more reward in the long run while it sacrifices short-term reward. After it has learned the action values it then should exploit the knowledge and always take the best action. The optimal trade-off when to explore and when to exploit of course depends on the problem. In non-stationary problems continued exploration is especially important as otherwise the algorithm sticks with a solution, which might not be optimal anymore, when the reward dynamics of the problem have changed.

Finding a good trade-off between exploration and exploitation is therefore one of the main challenges in reinforcement learning.

Model

A *model* is a mathematical representation of the environment's dynamics. It is used to predict state transitions and rewards given the agent's current action. A model might be known in advance or learned from interaction, e.g. by estimating the transition probabilities from sampled transitions.

When a model is known, the agent can directly improve the policy using the equations describing the model and no interaction with the environment is needed. This is known as *planning* or model-based reinforcement learning and a class of solution methods are dynamic programming algorithms presented in Section 2.3. This stands in contrast to *model-free* reinforcement learning, which learns from direct interaction with an environment without constructing a model of the environment's dynamics. Instead the policy is improved using samples of states, actions and rewards observed from interaction with the environment.

A model also allows to sample experiences according to the transition dynamics and learn from sampled experiences rather than from actual real world experience, which could be cheaper in many cases. Even if all the dynamics of the environment are known, it might be easier to treat this as a model-free reinforcement learning task and learn from samples instead of solving the very complex equations describing the problem (Sutton and Barto 2017).

In most cases the reinforcement learning model is formulated as a Markov Decision Process.

1.2 Markov-Decision Process

A Markov Decision Process (MDP) is a time-discrete stochastic process. State transitions and rewards are random and can be modeled by a state transition array and a reward matrix. A (finite) Markov Decision Process is then a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where $\mathcal{S}$ is the (finite) state space, $\mathcal{A}$ the (finite) action space, $\mathcal{P}$ the state transition array, $\mathcal{R}$ the reward matrix and γ is the discount factor with $\gamma \in [0, 1]$ (Sutton and Barto 2017).

The most important assumption is the Markov Property. It states that the next state and reward only depend on the current state and action and not on the history of all the states, actions and rewards before, so the current state already summarizes all

relevant information about the environment. In other words the future is independent of the past given the present.

The Markov Property is defined by

$$\mathbb{P}[S_{t+1} = s_{t+1}|S_t = s_t, A_t = a_t] = \mathbb{P}[S_{t+1} = s_{t+1}|S_1 = s_1, A_1 = a_1, ..., S_t = s_t, A_t = a_t]. \tag{3}$$

The state transition array $\mathcal{P}$ defines the transition probabilities between all states. $\mathcal{P}^a_{ss'}$ is the probability to transition from state s to state s' when taking action a,

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]. \tag{4}$$

The reward matrix specifies the expected reward when taking action a in state s. Rewards can be deterministic or stochastic and depend on the previous state and action.

$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]. \tag{5}$$

MDP's, especially finite MDP's, build the mathematical fundament of reinforcement learning. When formulated as a Markov Decision Process the state is fully observable by the agent. Sometimes not all relevant information is observable by the agent, e.g. in a card game a player only knows his cards but not the cards of his opponents. In this case we speak of a partially observable problem, which can be formulated as a Partially Observable Markov Decision Process (POMDP).

Episodic and continuing tasks

A reinforcement learning task is called an *episodic* task, if the interaction between agent and environment breaks down into a sequence of episodes, each consisting of a finite number of time steps. Then a terminal state exists, for example the end of a game. When this terminal state is reached, the episode is over and the environment is reset to a start state, from which a new episode is started. E.g. a game of chess is finished, when the king is checkmated and a new game always starts with the same position of

all pieces on the chess board. The start state could also be sampled from a distribution over possible start states.

Some tasks do not fall into this category, for example a trading agent might be concerned to maximize financial return in his life-time. This is a *continuing* task which does not naturally fall into episodes. Discounting rewards helps in continuing tasks to prevent infinite returns.

We will see algorithms in Section 2, which can only be applied to episodic problems, and algorithms, which can also be applied to continuing problems.

Value functions

While rewards are an immediate feedback, values represent the long-term consequences of actions. The state value function $v_\pi(s)$ is defined as the expected return following policy π from state s:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{6}$$

The action value function $q_\pi(s, a)$ is defined analogous as the expected return of taking action a in state s and then following policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{7}$$

A value quantifies how good a certain state or state-action pair is. Value functions play a central role in many reinforcement learning algorithms. A good policy can be found by optimizing a value function and then deriving the policy from the value function. Policies based on MDPs are always stationary, i.e. they only depend on the current state and not on states further back in time.

The action value function is especially useful because it contains the values for each action in a state. Therefore an agent may choose the action with the highest q value, because this is the best action in this state.

Bellman Equations

The Bellman equations define a recursive relationship between values.

A value function can be decomposed into the immediate reward plus the value of the successor state. This is called the Bellman expectation equation (Sutton and Barto 2017).

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...)|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \end{aligned} \tag{8}$$

Similarly the action value function can be written as

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]. \tag{9}$$

The expectation can then be computed using the transition dynamics of the MDP

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')\right]. \tag{10}$$

Equation 10 can be thought of as a one-step look-ahead. Starting in a state s each action is weighted by its probability under policy π. The value of the next state s' is weighted by the environment's transition probabilities. Summing this together we obtain the value of state s.

Similarly for the action value function

$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a'\in\mathcal{A}} \pi(a'|s')\, q_\pi(s', a'). \tag{11}$$

When taking action a in state s the environment returns a state according to the transition probabilities, then the agent chooses a new action a' from policy π.

State and action value function can also be expressed in terms of each other. The state value function is an average over all action values weighted by the probability of taking that action $\pi(a|s)$:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s)\, q_\pi(s,a). \tag{12}$$

Similarly the action value function can be expressed in terms of the state value function:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a\, v_\pi(s'). \tag{13}$$

The value of state s and action a is then the expected immediate reward plus the discounted sum of the values of all successor states s' weighted by their transition probabilities.

Optimality

In reinforcement learning the goal is to find the best (=optimal) policy, which achieves the highest cumulative reward. In finite MDPs there is connection between the optimal policy and its value function. Therefore it is enough to find the optimal value function, from which the optimal policy can then be easily derived.

Policies can be sorted according to their value functions. A policy π' is better than a policy π (denoted $\pi' \geq \pi$) if

$$v_{\pi'}(s) \geq v_\pi(s) \quad \text{and} \quad q_{\pi'}(s,a) \geq q_\pi(s,a) \qquad \forall s \in \mathcal{S} \quad \text{and} \quad a \in \mathcal{A}. \tag{14}$$

The optimal value function is the maximal value function over all policies:

$$v_*(s) = \max_\pi v_\pi(s) \quad \text{and} \quad q_*(s,a) = \max_\pi q_\pi(s,a) \qquad \forall s \in \mathcal{S} \quad \text{and} \quad a \in \mathcal{A}. \tag{15}$$

If the optimal action value function $q_*(s,a)$ is known, the problem is solved because the optimal policy can be obtained by acting greedily with respect to the optimal action

value function (Sutton and Barto 2017):

$$\pi_*(a|s) = \begin{cases} 1, & \text{if } a = \text{argmax}_{a \in \mathcal{A}}\, q_*(s,a) \\ 0, & \text{else.} \end{cases} \tag{16}$$

All suboptimal actions must have zero probability. If a perfect model of the environment is known, it is sufficient to know the optimal state value function $v_*(s)$ because the policy can then be derived from Equation 10.

The Bellman optimality equations define a relationship between the optimal value functions:

$$v_*(s) = \max_{a \in \mathcal{A}} q_*(s,a) \tag{17}$$

$$v_*(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a\, v_*(s') \right) \tag{18}$$

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a' \in \mathcal{A}} q_*(s',a') \tag{19}$$

With this knowledge of Markov Decision Processes we can understand the algorithms presented in Section 2. We will see that the Bellman equations can be turned into iterative updates to improve the value function estimates.

2 Algorithms

Reinforcement learning algorithms can be categorized into different subfields depending on their task and update procedure. Generally two tasks can be differentiated, the *prediction* and the *control* task. In the prediction problem the policy is fixed and the goal is to estimate the value function of this policy. In the control task the goal is to find the optimal policy and the policy evolves over time.

Another distinction can be made between *model-based* and *model-free* algorithms. Model-based algorithms use a model of the environment to update the value function, i.e. a representation of state transition probabilities and reward dynamics. Model-based dynamic programming algorithms will be introduced in Section 2.3. Model-free reinforcement learning in contrast learns from sampled transitions without any knowledge of the underlying model. *Temporal-difference* (TD) learning is a class of model-free algorithms which update the value function after each step using the estimated value of the successor state and will be explained in Section 2.4.

Model-free reinforcement learning can also be grouped into on-policy and off-policy algorithms. The policy used to generate samples by interacting with the environment is called *behavior policy*, while the policy which is optimized for is called *target policy*. For on-policy algorithms behavior and target policy are identical, in the off-policy case both policies are different, e.g. the behavior policy could be an exploratory policy like ϵ-greedy, while the target policy is the greedy policy (this is the well-known Q-Learning algorithm). Of course on-policy can then be seen as a special case of off-policy learning, when target and behavior policy are identical.

First we will cover *tabular* solution methods, where value function and policy are represented as a table. Each entry of the table is then the value of a state or state-action pair. In contrast to that *approximate* solution methods approximate the value function or policy using a function approximator, e.g. a linear combination of features or a neural network and will be introduced in Section 2.5.

Value-based methods try to learn a value function, e.g. the state value function V or the action value function Q. The optimal policy can then be derived implicitly by finding the maximal values. *Policy-based* methods directly parametrize the policy and try to find the optimal policy without constructing a value function. Algorithms that use both value function and policy are called *actor critic* and will be treated in Section 2.6.

2.1 Multi-armed Bandit

A multi-armed bandit is a simplified reinforcement learning problem, where each episode consists of only one time step. After taking an action the agent receives a numerical reward from the environment. The goal is to maximize the expected total return over a number of action selections (= episodes), which usually involves some trade-off between exploration and exploitation. Formally bandits can be formulated as Markov Decision Processes with one state.

The expected return of each action, which is the value of the action, can be expressed as

$$q_t(a) = \mathbb{E}[R_t | A_t = a]. \tag{20}$$

The true action values $q_t(a)$ are usually unknown in the beginning and must be estimated by trying out actions and observing the corresponding rewards. In the following we will use capital letters V and Q to denote estimated value functions in contrast to the true value functions q and v.Note that in the simplest form the action values do not depend on states, respectively there is only one state, so it can be neglected. The time index t now corresponds to the episodes. So the return of each episode is equal to the immediate reward and there is no discounting.

The action values can then be estimated by taking the mean of all rewards observed in the t episodes,

$$Q_t(a) = \frac{1}{t} \sum_{i=1}^{t} R_i \, I_{\{A_i = a\}}, \tag{21}$$

and the computationally advantageous incremental update equation is

$$Q_t(a) = Q_{t-1} + \frac{1}{t} \left[R_t - Q_{t-1} \right]. \tag{22}$$

Greedy Action Selection

An agent which acts according to a greedy policy always chooses the action with the highest action value.

$$A_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q_t(a) \tag{23}$$

A problem with greedy action selection is that it does not explore, which is problematic because the action value estimates are uncertain and an action with a low estimate might have a high true value but is never chosen.

Optimistic Initial Values

A simple trick to get better results using a greedy action selection is to initialize the Q values to high values which might be overly optimistic. Then the greedy strategy will choose an action and is "disappointed" from the result, hence the action value of this action will be decreased and next time one of the other actions is chosen. This way all actions are chosen many times until the value of the best action is found. Higher initial values lead to a slower decrease of the action values and therefore to more exploration as the actual observed rewards have a lower weight.

ϵ-greedy Action Selection

ϵ-greedy is a very simple but often effective algorithm trading off exploration and exploitation. With probability $1 - \epsilon$ the greedy action is chosen, with probability ϵ a random action:

$$A_t = \begin{cases} \operatorname{argmax}_{a \in \mathcal{A}}\, Q_t(a) & \text{with probability } 1 - \epsilon \\ \text{a random action } a \in \mathcal{A} & \text{with probability } \epsilon \end{cases} \tag{24}$$

Therefore higher values of ϵ lead to more exploration. Usually the value of ϵ is decreased over time to ensure more exploration in the beginning and then start to act more and more greedily. ϵ-greedy action selection is also useful if the environment is non-stationary, i.e. the reward distribution changes over time. By taking a random action once in a time the algorithm can discover a change in the reward distribution and adapt its Q values accordingly.

Upper-Confidence-Bounds (UCB)

While the simple ϵ-greedy action selection works well for many problems, sometimes a more sophisticated approach to balance exploration and exploitation is useful. One disadvantage of the ϵ-greedy approach is that all non-greedy actions will be chosen with the same probabilities, invariant to the fact that some are close in action value to the greedy action and are therefore more likely to be better, while others are far off. Secondly ϵ-greedy does not keep track of the number of times an action has been chosen, though it would be preferable to give actions which have been chosen scarcely a higher weight while sampling, because the action values are more uncertain.

Using an Upper-Confidence-Bound Action Selection actions are selected greedily with respect to a term consisting of the action value $Q(a)$, therefore giving higher weight to actions with high action value, and a second term reflecting the uncertainty of the estimated value making it more likely to choose an action with uncertain value,

$$A_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left[Q_t(a) + \sqrt{\frac{C\, log(t)}{N_t(a)}} \right], \tag{25}$$

where $N_t(a)$ denotes the number of times action a has been chosen. Initially all $N_t(a) = 0$, in this case an action with $N_t(a) = 0$ is given highest priority, so first all actions are taken at least once, before actions are selected according to Equation 25. The term under the root is the uncertainty term and is increased at each time step for all actions which have not been selected and decreased for the action which has been selected due to an increased number of selections of this action $N_t(a)$.

Gradient Bandit Algorithm

A different approach to solve a multi-armed bandit problem is to use a gradient-based method, which does not use an action value function. Instead of an action value a numerical preference $H_t(a)$ will be stored for each action. Probabilities can then be computed using the softmax formula $P_t(a) = \frac{exp(H_t)}{\sum_a exp(H_t(a))}$. Actions are sampled according to these probabilities. The average reward is computed and compared to the observed reward of a chosen action, so the average reward is used as a baseline. If the reward exceeds the average reward, the preference and therefore the probability of

choosing this action is increased and for all other actions decreased. (Sutton and Barto 2017)

2.2 Generalized Policy Iteration

Generalized policy iteration is a framework to find the optimal policy in a reinforcement learning environment based on value functions. This is used by nearly all algorithms presented in the following. It iterates between two steps: policy evaluation and policy improvement. In the policy evaluation step the goal is to estimate the value functions v_π or q_π for a policy π. In the policy improvement step the goal is to improve upon the current policy π to find a better policy π'. Ultimately the goal is to find the optimal value functions v_{π_*} and q_{π_*} and the optimal policy π_*.

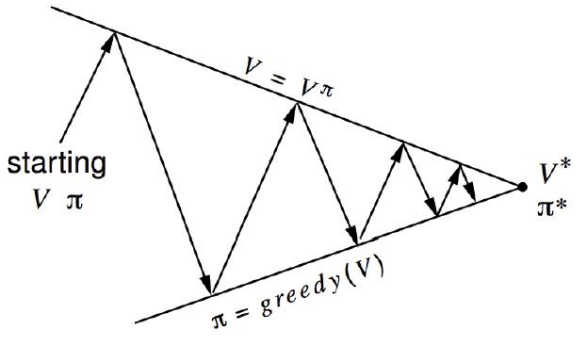
Policy Evaluation

We will see different approaches to evaluate a policy used in the different algorithms. The value-based methods presented in the following mainly differ by their policy evaluation procedure. Policy evaluation can be done until the true values q_π and v_π of policy π are approximated closely, but often only one step of policy evaluation is done, so the value function is updated only once and then the policy is immediately updated using a policy improvement step. This is called value iteration.

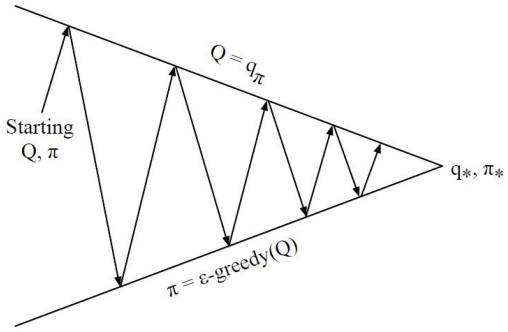
Policy Improvement

A policy improvement step improves the current policy π towards a better policy π' as defined in Equation 14. But how do we obtain a better policy?

For each state we consider the action values $q_\pi(s, a) \, \forall a \in \mathcal{A}$. If $q_\pi(s, a) > v_\pi(s)$ then it is better to select a in s instead of following policy π. Then the new policy is better than the old policy. For every state we can select the action with the highest action value q_π to obtain a better policy, in other words to act greedily with respect to the action value function. The resulting new policy π' will then be at least as good as the previous policy π according to the policy improvement theorem (Sutton and Barto 2017).



(a) Generalized Policy Iteration using the state value function V and a greedy policy improvement. This iteration procedure can be used when the model of the MDP is known.



(b) Generalized Policy Iteration using the action value function Q and an ϵ-greedy policy improvement. This is used in model-free reinforcement learning.

Figure 2.1: Generalized Policy Iteration. Sutton & Barto (2017)

When a model of the MDP is known a better policy can be obtained using the state value function V with a one-step look-ahead at the values of the successor states using the transition probabilities, so the policy improvement becomes

$$\pi'(a|s) = \begin{cases} 1, & \text{if } a = \text{argmax}_{a \in \mathcal{A}}(\mathcal{R}_s^a + \gamma P_{ss'}^a V(s')) \\ 0, & \text{else.} \end{cases} \tag{26}$$

But when the model is unknown a greedy policy is not useful, because it does not explore. Instead the idea is to use an ϵ-greedy policy, which acts greedily with probability $1 - \epsilon$, but takes a random action with probability ϵ. Furthermore because the transition dynamics are unknown in a model-free setting, we need to use the action value function Q instead of the state value function V. The policy improvement update is then

$$\pi'(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{m}, & \text{if } a = \text{argmax}_{a \in \mathcal{A}} Q(s, a) \\ \frac{\epsilon}{m}, & \text{else.} \end{cases} \tag{27}$$

The update cycle in generalized policy iteration is then $\pi_0 \to Q_{\pi_0} \to \pi_1 \to Q_{\pi_1} \to \ldots \to \pi_* \to Q_{\pi_*}$ using the action value function. Figure 2.1 visualizes the principle of generalized policy iteration for model-based and model-free reinforcement learning.

2.3 Dynamic Programming

Dynamic programming is a class of solution methods to find the optimal policy π_* using a model of the environment. A Markov Decision Process with a finite state and action space is assumed, which is fully determined by the state transition probabilities and reward dynamics. When a perfect model of the environment is known no actual or simulated experience sampled from interaction with the environment is needed. Instead the Bellman equations can be turned into iterative update procedures of the value function. In each iteration all state values are updated using the values of all successor states. This is called a *full-backup*. This stands in contrast to *sample backups* used in model-free reinforcement learning, which update a state value using the value of only one sampled successor state.

Iterative Policy Evaluation

Using dynamic programming a policy can be evaluated iteratively by turning the Bellman expectation equation (Equation 10) into an iterative update.

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right] \tag{28}$$

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s)\ q_k(s,a). \tag{29}$$

A new state value estimate $v_{k+1}(s)$ is obtained by weighting the action values $q_k(s,a)$ by the probability of taking that action under policy π. The action values itself are computed as the sum of the expected immediate reward plus the values of all successor states weighted by the probability of a transition to them.

At each iteration all state values are updated towards the value of their successor states. Of course these values are at first random, but by using the knowledge about the true reward and transition dynamics, the estimates will get better over time and are proofed to converge to the true state value function v_π. (Sutton and Barto 2017)

Usually the iteration process (Equation 29) will be performed until a stop criterion is met, e.g. until a predefined number of steps is exhausted or the difference between two

subsequent value functions is less than a given threshold.

Policy Iteration

With the knowledge of how to evaluate a policy the next step is how to improve the policy and find the optimal policy π_*. To find the optimal policy we will iterate between iterative policy evaluation and a greedy policy improvement step as in Equation 26.

The update cycle is then $\pi_1 \rightarrow v_{\pi_1} \rightarrow \pi_2 \rightarrow v_{\pi_2} \rightarrow \ ... \ \rightarrow \pi_* \rightarrow v_{\pi_*}$.

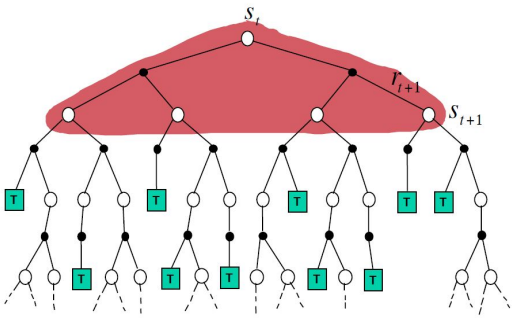
Each policy will be evaluated until one of the stop criteria is met. Then a greedy update in all states improves the policy. Then the new policy will be evaluated and so forth. At each iteration the policy evaluation will start with the value function of the previous policy and because the value function is relatively stable from one policy to the next, this will increase the speed of convergence. This iteration process is proven to converge to the optimal policy π_* (Sutton and Barto 2017).

Value Iteration

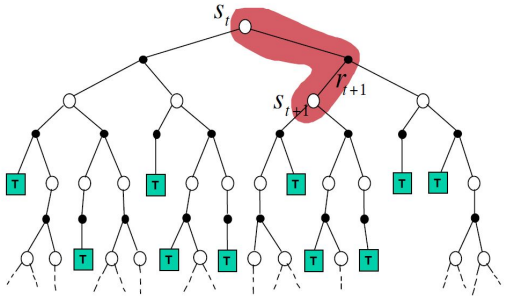
Value iteration in contrast to policy iteration evaluates each policy only once and then immediately improves the current policy by acting greedily. Therefore the intermediate value functions may not correspond to any policy. Often the greedy policy with respect to a value function does not change anymore after a few steps of policy evaluation. This is why value iteration will typically converge much faster than policy iteration, because it does not need to wait until a policy is fully evaluated, but can use new information after just one evaluation step.

The update rule can be directly derived from the Bellman optimality equation (Equation 18),

$$V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_k(s') \right]. \tag{30}$$



(a) Dynamic programming updates the value of a state using the values of all successor states weighted by the transition probabilities ("full backup"). It can therefore be only applied if a model of the environment is known.



(b) Model-free reinforcement learning (here temporal-difference learning) updates state values using the value of a sampled successor state ("sample backup"). It can therefore be applied without knowledge of the underlying model.

Figure 2.2: Silver (2015)

2.4 Temporal-difference learning

Dynamic programming can only be applied if the model of the environment is fully known. In most interesting problems this is not the case. The solution is to learn from sampled experience from interaction with the environment, either from real-world interaction or from simulated experience sampled from a model. Often it is much easier to obtain samples from some probability distribution than to obtain the equations describing the probability distribution (Sutton and Barto 2017).

In the following agent and environment interact over a number of discrete time steps, so that a sequence of states, rewards and actions is obtained. The policy used to interact with the environment is also called *behavior policy*. It is usually an exploratory policy, e.g. ϵ-greedy, to ensure that all states and actions are visited to obtain a good value function estimate. ϵ-greedy action selection is a common choice to trade off exploration and exploitation.

When the goal is to estimate a fixed policy we can evaluate the policy's state value function V_π or action value function Q_π. When optimizing the policy to find the optimal way to behave, the action value function Q must be used, because the best action in a state can be easily found by taking the maximum over the Q values. The idea is to estimate the expectations in

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \quad \text{or} \quad q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a], \tag{31}$$

instead of directly computing them as in dynamic programming.

Temporal-difference (TD) learning is a widely used class of model-free reinforcement learning algorithms. It can be applied to continuing problems, because the value function is updated after each step (on-line learning).

Similar to dynamic programming TD learning corrects the estimate of a state towards the estimate of a successor state by a one-step look-ahead. But in contrast to dynamic programming a state value is not updated towards the value of **all** successor states but only towards the value of a **sampled** successor state (Figure 2.2).

In temporal-difference learning the policy evaluation step usually contains an update of the form

$$V(S_t) \leftarrow V(S_t) + \alpha\, \delta_t \tag{32}$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\, \delta_t, \tag{33}$$

where the current estimate of the value function is shifted by the step size α (also called learning rate) in the direction of the so called TD error δ_t. The TD error is the difference between a newly computed target value and the old estimate of the value function.

TD(0)-Algorithm

The simplest TD algorithm for policy evaluation is TD(0), which uses a TD error of the form

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{34}$$

This is again the Bellman expectation equation, which decomposes the return G_t into the immediate reward plus the discounted value of the next state. In comparison to

iterative policy evaluation in dynamic programming (Equation 29) the expectation is now approximated using the value of a sampled successor state S_{t+1}, instead of looking ahead at all possible successor states.

Sarsa

Sarsa (Rummery and Niranjan 1994) is a temporal-difference algorithm for control. At each time step an action is sampled according to an ϵ-greedy policy with respect to Q as defined in Equation 27. After taking this action the following reward and next state are observed. Then a successor action A_{t+1} is sampled from the same policy.

Using Equation 33 the action value function is then updated using the following TD error,

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t). \tag{35}$$

Using the bellman expectation equation for Q (Equation 11) the old estimate $Q(S_t, A_t)$ is backed up by the reward plus the discounted value of the value of the successor state $Q(S_{t+1}, A_{t+1})$.

Sarsa is an on-policy algorithm, i.e. the TD target consists of $Q(S_{t+1}, A_{t+1})$, where A_{t+1} is sampled using the same policy as A_t. In general the behavior policy used to sample state and actions can be different from the target policy π, which is used to compute the TD target. If behavior and target policy are different this is called off-policy learning.

Q-Learning

Q-Learning (Watkins 1989) is an off-policy control algorithm, i.e. behavior and target policy are different. Like Sarsa experiences are generated using an ϵ-greedy behavior policy. But the value estimates for the sampled state-action pairs are then updated by taking the maximum value of all possible next actions. The TD error can be derived from the Bellman optimality equation,

$$\delta_t = R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t). \tag{36}$$

Expected Sarsa

Expected Sarsa (Seijen et al. 2009) is an extension of the Sarsa algorithm. Unlike Sarsa, which samples the next action and uses its value $Q(S_{t+1}, A_{t+1})$ as an update, Expected Sarsa does not sample the next action, instead it takes an expectation over all possible successor actions. All actions are weighted by their probabilities $\pi(a|S_{t+1})$.

The TD error of Expected Sarsa is therefore

$$\delta_t = R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1})|S_t] - Q(S_t, A_t) \tag{37}$$

$$\delta_t = R_{t+1} + \gamma \sum_{a \in \mathcal{A}} \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \tag{38}$$

The updates in Expected Sarsa are computationally more complex than the updates in Sarsa, but they eliminate the variance due to the second action selection in Sarsa, the sampling of A_{t+1}, and can therefore perform better in stochastic environments (Seijen et al. 2009). Expected Sarsa can be used both as an on-policy or off-policy algorithm. Expected Sarsa is an off-policy algorithm when the target policy π used in Equation 38 is different than the policy used to sample A_t. If the target policy π is the greedy policy with respect to Q and the behavior policy is an exploratory policy, e.g. the ϵ-greedy policy, then Expected Sarsa is exactly Q-Learning.

Q(σ)-Algorithm

The $Q(\sigma)$ algorithm (Asis et al. 2017) generalizes the temporal-difference control methods presented so far. It subsumes Sarsa as well as Expected Sarsa and therefore also Q-Learning. Instead of full sampling as in Sarsa and pure expectation as in Expected Sarsa the $Q(\sigma)$ algorithm has a parameter σ, which controls the sampling ratio. $Q(1)$ is equivalent to Sarsa, $Q(0)$ is equivalent to Expected Sarsa. For intermediate values of σ new algorithms are obtained. The TD target is therefore a weighted average of the sarsa and expected sarsa targets.

The TD error of $Q(\sigma)$ is

$$\delta_t = R_{t+1} + \gamma \left(\sigma Q(S_{t+1}, A_{t+1}) + (1-\sigma) \sum_{a \in \mathcal{A}} \pi(a|S_{t+1}) Q(S_{t+1}, a) \right) - Q(S_t, A_t). \quad (39)$$

Double Learning

Double Q-Learning (H. V. Hasselt 2010) is a strategy to avoid the overestimation of action values, which can be problematic when using Q-Learning in stochastic environments. Q-Learning uses the maximum value as an estimate of the maximum expected value, which can lead to a high positive bias in the action values and therefore poor performance on some tasks. This is called maximization bias. The idea behind this is that due to the stochasticity of reward or state transitions the estimates of action values are uncertain and some estimates might be higher than their true value. In some sense Q-Learning chooses the action, which is most wrong in an upward direction, because the estimate is to high.

Double Q-Learning is a method to avoid this bias by decoupling action selection (which action is the best one?) and evaluating this best action (what is the value of this action?). The implementation is simple, instead of using only one value function Q it stores two different action value functions Q_A and Q_B. States and actions are sampled using an ϵ-greedy policy with respect to $Q_A + Q_B$. To update the values at each step it is randomly determined, which of the two value functions is updated. If Q_A is updated the TD error is

$$\delta_t = R_{t+1} + \gamma Q_B(S_{t+1}, \operatorname*{argmax}_{a \in \mathcal{A}} Q_A(S_{t+1}, a)) - Q_A(S_t, A_t), \quad (40)$$

else if Q_B is updated it is

$$\delta_t = R_{t+1} + \gamma Q_A(S_{t+1}, \operatorname*{argmax}_{a \in \mathcal{A}} Q_B(S_{t+1}, a)) - Q_B(S_t, A_t). \quad (41)$$

Double Learning can also be used with Sarsa and Expected Sarsa. Using Double Learning these algorithms can be more robust and perform better in stochastic environments

(Ganger, Duryea, and Hu 2016). The decoupling of action selection and action evaluation is weaker than in Double Q-Learning because the next action A_{t+1} is selected according to an ϵ-greedy behavior policy using $Q_A + Q_B$ and evaluated either with Q_A or Q_B. For Expected Sarsa the policy used in Equation 38 could be the ϵ-greedy behavior policy as proposed by Ganger, Duryea, and Hu (2016), but it is probably better to use a policy according to Q_A, then it can also be used off-policy with Double Q-Learning as a special case, if π is the greedy policy with respect to Q_A.

To extend the Q(σ)-Algorithm to Double Learning we propose a new algorithm called Double Q(σ), which would have the following TD error when Q_A is selected,

$$\delta_t = R_{t+1} + \gamma \left(\sigma Q_B(S_{t+1}, A_{t+1}) + (1-\sigma) \sum_a \pi(a|S_{t+1}) Q_B(S_{t+1}, a) \right) - Q_A(S_t, A_t) \quad (42)$$

and analogous with interchanged roles for Q_B. Because Q(σ) subsumes Q-Learning and Expected Sarsa as a special case, the new algorithm Double Q(σ) subsumes Double Q-Learning and Double Expected Sarsa as special cases. While the behavior policy is the ϵ-greedy policy according to $Q_A + Q_B$, the target policy π in Equation 42 is computed with respect to Q_A and evaluated using Q_B. More details on this new algorithm can be found in the Appendix.

Eligibility Traces

The algorithms presented so far in this section are one-step methods, i.e. they update only one state-action pair at each step. This is not data-efficient. All the previous states and actions influenced the current TD error and therefore the TD error should be assigned backwards to all previously visited states and action, which can result in much faster learning (Sutton and Barto 2017). This can be achieved using eligibility traces. A motivating example is visualized in Figure 2.3.

An eligibility trace $E_t(s)$ (or $E_t(s,a)$) is a scalar numeric value for each state (or state-action pair). Whenever a state is visited its eligibility is increased, if not, the eligibility fades away over time with an exponential decrease. Therefore states visited more often will have a higher eligibility trace than those visited less frequently and states visited recently will have a higher eligibility than those visited a long time ago.

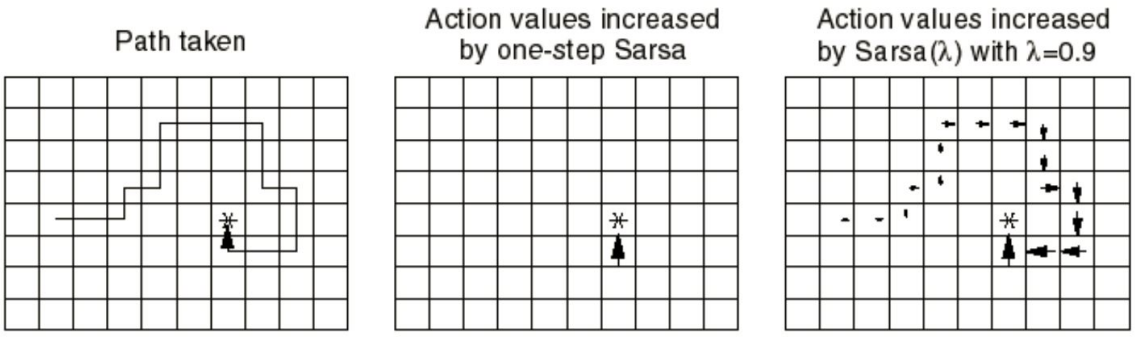


Figure 2.3: Action values increased by one-step Sarsa and Sarsa(λ) in a gridworld task. Using Sarsa(λ) all state-action pairs are updated according to their share on the current TD error, which is assigned backwards to all predecessor states and actions. Sutton & Barto (2017)

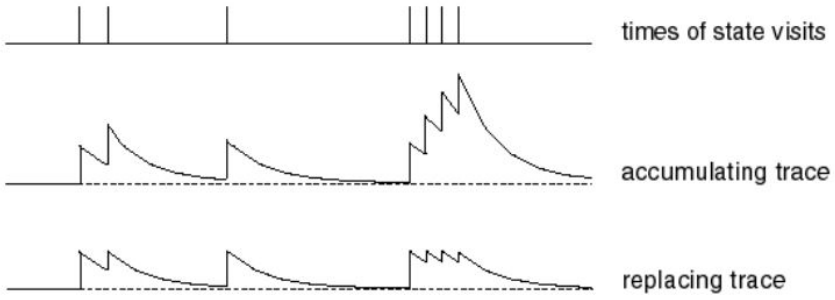


Figure 2.4: Comparison between accumulating and replacing eligibility trace for state values. Values are increased, whenever a state is visited, then they fade away over time. Singh and Sutton (1996)

In the following action value eligibility traces $E_t(s, a)$ are used, but the equations for state value eligibility traces $E_t(s)$ can be obtained analogously.

Types of Eligibility Traces

Different types of eligibility traces have been proposed in the literature. Two kinds are especially commonly used: the accumulating trace and the replacing trace (Singh and Sutton 1996), which are visualized in Figure 2.4.

At the beginning of each episode all eligibility traces are reset to 0. Then at each time step the eligibility of the current state-action pair (S_t, A_t) is increased. For accumulating

traces the update is

$$E_t(S_t, A_t) = E_{t-1}(S_t, A_t) + 1 \tag{43}$$

and for replacing traces

$$E_t(S_t, A_t) = 1. \tag{44}$$

Recently a different eligibility trace has been introduced, the so called Dutch trace, which is a weighted average between accumulating and replacing traces (Seijen et al. 2015):

$$E_t(S_t, A_t) = (1 - \psi)\, E_{t-1}(S_t, A_t) + 1 \tag{45}$$

The factor ψ controls if accumulate or replace traces are used with $\psi = 0$ being the regular accumulate trace update and $\psi = 1$ being the regular replace trace update. For intermediate values a mixture of both traces is used.

Comparing the different types of eligibility traces the following can be noted: Replacing traces are bounded with an upper bound of 1, while accumulating traces can become larger than 1. The latter can be a problem, when a state-action pair is revisited often during a long episode and the discount factor and λ value are high. Then the eligibility trace and value updates can become very large resulting in instable learning and possibly divergence (Singh and Sutton 1996).

After the eligibility trace of the current state-action pair has been increased, the values of all state-action pairs are then updated towards the TD error weighted by their eligibility

$$Q(s, a) \leftarrow Q(s, a) + \alpha\, \delta_t\, E_t(s, a) \qquad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}. \tag{46}$$

Afterwards the eligibility traces of all states and actions are decreased exponentially

$$E_{t+1}(s,a) = \gamma\lambda E_t(s,a) \qquad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}, \tag{47}$$

where the trace decay parameter λ controls the speed of the decay. When $\lambda = 0$ the eligibility traces of all state-action pairs except the current state-action pair are 0, so only the value of this state-action pair is updated and we obtain again a one-step algorithm. For higher λ values the error is passed to all previously visited state-action pairs.

The algorithms using eligibility traces are called TD(λ), Sarsa(λ) and so on.

Eligibility Traces for off-policy Learning

There are different approaches to use eligibility traces with off-policy algorithms like Q-Learning. The so called "naive" approach is to use Equations 43 - 47 without changes, ignoring the fact, that actions are sampled due to an ϵ-greedy behavior policy and not due to the target policy (e.g. the greedy policy). Another approach to Q-Learning with eligibility traces called Watkin's Q(λ) uses the same Equations as long as the greedy action is chosen by the behavior policy, but sets the Q values to 0, whenever a non-greedy action is chosen. It assigns credit only to state-action pairs the agent would actually have visited if following the target policy π and not the behavior policy μ. This can be generalized for other target policies by weighting the eligibility by the target policy's probability of the next action.

The eligibility is then decreased by

$$E_{t+1}(s,a) = \gamma\lambda E_t(s,a)\, \pi(A_{t+1}|S_{t+1}) \qquad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}. \tag{48}$$

Whenever an action occurs, which is unlikely in the target policy, the eligibility traces are decreased sharply. If the target policy is the greedy policy, the eligibility traces will be set to 0 for the complete history if a non-greedy action is chosen.

We propose a new algorithm called Q(σ, λ) which extends Q(σ) to an on-line multi-step algorithm using eligibility traces. As Q(σ) is a combination between on-policy Sarsa and off-policy Expected Sarsa, the natural idea is to combine the eligibility trace updates in Equation 47 and 48 in the same way.

The new update equation is then

$$E_{t+1}(s,a) = \gamma\lambda(\sigma + (1-\sigma)\pi(A_{t+1}|S_{t+1}))\, E_t(s,a) \qquad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}. \tag{49}$$

When $Q(1,\lambda)$ is equal to Sarsa(λ) and the eligibility update is equal to Equation 47. For $Q(0,\lambda)$ is equal to Expected Sarsa and the eligibility update is equal to Equation 48. More details on the new Q(σ,λ) algorithm can be found in the Appendix.

2.5 Value Function Approximation

Until now tabular solution methods have been presented, which use a table to store the value function. For the state value function the table needs n entries, for the action value function n x m entries, when n is the number of states and m is the number of actions. This quickly becomes difficult if there are many states and / or actions. It can be impractical to store the table in memory and it is inefficient and slow to update states individually. But most importantly tables cannot be used if the state or action space is continuous. In the following we will introduce the idea of function approximation to deal with continuous state spaces. Continuous action spaces are especially difficult to handle, because algorithms using an ϵ-greedy policy improvement, e.g. Q-Learning, need to do a maximization over the action space to find the best action. With a continuous action space this becomes a difficult optimization problem (Sutton and Barto 2017). When the problem has a continuous action space it is therefore often better to use a policy gradient algorithm (Section 2.6).

State Aggregation

A first approach when dealing with a continuous state space could be to discretize the state space and then apply a tabular solution method. A simple approach is grid tiling, which aggregates the state space with a grid. Each state observation falls into one of the grid cells, so the tabular value function would then have one entry for each grid cell. Generalization occurs across all states, which fall into the same grid cell. An extension of this idea is to overlay the state space with multiple grid tilings all offset from each other. Then a state observation falls into one tile per tiling, so the value function has then a number of rows equal to the number of tilings multiplied with the number of

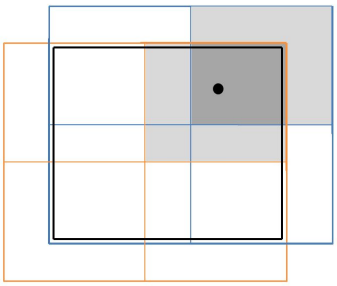


Figure 2.5: Grid tiling in a two-dimensional state space (black) with two tilings (orange and blue), each with 4 tiles. A state (black point) generalizes over all points which fall into the same tiles.

tiles per tiling. The value of a state is then simply the sum of all values of active tiles. Figure 2.5 visualizes a possible grid tiling for a two-dimensional state space.

Value Function Approximation

A different approach is to approximate the value function

$$\hat{v}(s, w) \approx v_\pi(s) \tag{50}$$

$$\hat{q}(s, a, w) \approx q_\pi(s, a), \tag{51}$$

where w are the real-valued parameters (also called weights) of the function approximator, e.g. the weights of a neural network. Usually the number of weights is much smaller than the number of states, so the function approximator has to generalize across the state space. When a state is visited the parameters of the model are updated which usually effect the values of many states.

For the action value function there are two slightly different implementations. Either the function approximator takes the state as an input and returns the values for each action, or the function approximator receives a state-action pair and returns the value of this state-action pair.

The idea is to train the function approximator on examples (input-output pairs) from the function we would like to approximate. This is similar to the approach in supervised learning, e.g. regression. The challenge in reinforcement learning is that no fixed data

set exists, over which multiple passes can be made, but the model must be able to learn on-line while interacting with an environment. So it needs to learn from incrementally acquired data.

To measure the performance we have to define a loss function. A natural choice for the loss function is the Mean Squared Value Error (MSVE),

$$\text{MSVE}(w) = \mathbb{E}_\pi \left[(v_\pi(S_t) - \hat{v}(S_t, w))^2 \right] \tag{52}$$

$$\text{MSVE}(w) = \mathbb{E}_\pi \left[(q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, w))^2 \right] \tag{53}$$

The MSVE expresses how much the estimated values differ from the true values under policy π. Usually it is not possible to achieve a global optimum, i.e. $\text{MSVE}(w^*) \leq \text{MSVE}(w)\, \forall w$, only a local optimum and often there are no convergence guarantees at all (Sutton and Barto 2017).

Stochastic Gradient Descent

Gradient descent is a widely used optimization method and well-suited for the on-line nature of reinforcement learning. For a differentiable function $J(w)$ the gradient is defined by

$$\nabla J(w) = \begin{pmatrix} \frac{\partial J(w)}{\partial w_1} \\ \vdots \\ \frac{\partial J(w)}{\partial w_n} \end{pmatrix}. \tag{54}$$

The idea is to iteratively update the weights using the gradient of the MSVE. At each time step at $t = 0, 1, 2, ...$ the weights are adjusted by a small step in the negative direction of the gradient to minimize the MSVE. Stochastic gradient descent is a variant of gradient descent which trains only on one training example at each time step,

$$w_{t+1} = w_t - \frac{1}{2}\alpha\nabla_w \left[(v_\pi(S_t) - \hat{v}(S_t, w))^2\right] \tag{55}$$

$$= w_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, w)\right] \nabla_w \hat{v}(S_t, w). \tag{56}$$

$$\tag{57}$$

Equations for $\hat{q}(S_t, A_t, w)$ can be obtained similarly.

The negative gradient defines the direction in which the error falls most rapidly. The step size α controls how large the step in this direction is. Using a high learning rate we can reduce the error on the current training example, but this will make other training examples probably more incorrect, so a small learning rate may be advantageous.

In practice the true values $v_\pi(S_t)$ are unknown, so we must substitute a target for them. We can use the targets of any of the algorithms presented so far. For example the update for TD(0) is

$$w_{t+1} = w_t + \alpha[R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)] \nabla_w \hat{v}(S_t, w)$$

and the update for Sarsa(0) is

$$w_{t+1} = w_t + \alpha[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w) - \hat{q}(S_t, A_t, w)] \nabla_w \hat{q}(S_t, A_t, w).$$

More general the update rule becomes

$$w_{t+1} = w_t + \alpha\, \delta_t\, \nabla_w \hat{q}(S_t, A_t, w),$$

where δ_t is the TD error at time step t.

Though the target value also depends on the current weight vector w we ignore its effect in the gradient, therefore these are no true gradient descent, but only semi-gradient descent methods (Sutton and Barto 2017).

Eligibility Traces for Function Approximation

The idea of eligibility traces can be easily extended to function approximation. Instead of storing one eligibility trace per entry of a table, one eligibility trace per weight is used.

The Equations 43 - 47 can be modified for function approximation. For accumulating traces the update of the weights of the action value function would then be

$$E_t(w) = E_t(w) + \nabla_w \hat{q}(S_t, A_t, w). \tag{58}$$

The weights are then updated by multiplying the TD error by their eligibility trace

$$w_{t+1} = w_t + \alpha\, \delta_t\, E_t(w). \tag{59}$$

A great variety of regression models can be used for function approximation in reinforcement learning. In the following two of the most popular ones will be presented.

Linear Function Approximation

A common choice for approximate reinforcement learning is linear function approximation. The state observation is represented by a feature vector $x(S_t)$. The state value can then be represented as a linear combination of these features:

$$\hat{v}(S_t, w) = x(S_t)^T w = \sum_{j=1}^{n} x_j(S_t)\, w_j \tag{60}$$

The loss function is then quadratic in w:

$$\text{MSVE}(w) = \mathbb{E}_\pi \left[(v_\pi(S_t) - x(S_t)^T w)^2 \right] \tag{61}$$

The gradient $\nabla_w \hat{v}(S_t, w)$ is then simply the feature vector $x(S_t)$. The update rule is therefore

$$w_{t+1} = w_t + \alpha\, \delta_t\, x(S_t). \tag{62}$$

The weights are adjusted in the direction of the TD error times the value of the feature. When observing a large positive TD error, e.g. the action is better than expected, the weights corresponding to features with a high value will be updated more, so these features will get most of the credit for the error.

Table-look-up is a special case of linear function approximation using a one-hot representation of the state as a feature vector:

$$\hat{v}(S_t, w) = \begin{pmatrix} I_{\{S_t=s_1\}} & \dots & I_{\{S_t=s_n\}} \end{pmatrix} \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} \tag{63}$$

For each state (state-action pair) there is one entry in the feature vector and at each time step only one of those is active, so the feature vector consists of a 1 for this feature and 0 elsewhere. There are as many weights as there were entries in the table and at each step only the weight corresponding to the active feature is updated.

The equations for $\hat{q}(s, a, w)$ can be derived analogous.

Deep Reinforcement Learning

Deep reinforcement learning is a field of reinforcement learning, which uses deep neural networks as function approximators. A neural network is a statistical model used for regression or classification tasks which can approximate nonlinear functions. The simplest neural network is the feed-forward neural network. It consists of multiple layers, each consisting of multiple units. Inputs are processed layer-wise and passed through the network, so that a more advanced representation of the input is created. The value of each unit is computed as a sum of all connected units from the preceding layer and then transformed by some activation function. The last layer is called the output layer, layers between input and output layer are called hidden layers. A neural network with more than one hidden layer is typically called a deep neural network. There are other neural network types used for different tasks, e.g. convolutional neural networks for image data and recurrent neural networks for sequential data (e.g. language processing).

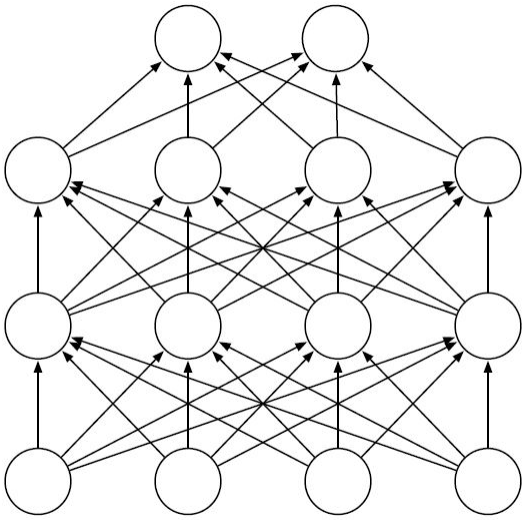


Figure 2.6: A feed-forward neural network with two hidden layers and two output units. Sutton and Barto (2017)

With the use of neural networks the handcrafting of meaningful features is not so important anymore. The combination of reinforcement learning algorithms with deep neural networks as value function approximators achieved great results on different tasks, e.g. learning to play different Atari games on human-level directly from raw pixels and rewards (Mnih et al. 2015) or solving the game of Go (Silver et al. 2016). These results were achieved using Q-Learning with a deep neural network, a so called Deep Q-Network (DQN).

Instability of Learning with Function Approximation

Reinforcement learning can be instable whenever the following three components are combined: function approximation, temporal-difference learning and off-policy training (Sutton and Barto 2017). This is called the "deadly triad". When only two of these three components are present, this is usually not a problem, but the presence of all three can cause instability and even divergence.

There are a number of challenges arising with reinforcement learning which are not present in supervised learning. In supervised learning models are usually trained on lots of labeled training examples. Reinforcement learning in comparison learns from a scalar reward, which can be sparse, noisy and delayed (Mnih et al. 2013). The consequences

of actions can be observed sometimes thousands of time steps later. Furthermore data is received consecutively over time and therefore often highly correlated, which is problematic as most models assume independent training examples. Finally the value function, which is approximated, changes over time as the behavior of the agent, the policy, changes. Therefore the distribution generating the observed transitions changes over time and is not stationary, so the transitions are not identically distributed.

Experience Replay

When training a reinforcement learning agent data is received consecutively over time. This data is usually highly correlated and makes training difficult. Experience replay is a technique which solves this problem (Mnih et al. 2013).

Experience Replay consists of the following steps:

- add transition to replay memory
- sample mini-batch from replay memory
- train model on mini-batch

You can think of the replay memory as a list storing every transition the agent experiences. Each entry consists of state, action, reward and the following state. At each step the agent trains on a number of examples (called mini-batch) randomly sampled from the replay memory. Because the agent does not train on the transitions the way it experiences them, but in a random sequence, this breaks the correlation, which we face when training on consecutive experiences. Experience replay makes reinforcement learning more like supervised learning. Training the neural network on a mini-batch of data points is usually more data efficient than training on-line on a single data point. When training on-line the data is used only once at the current time step and then disregarded. Using experience replay one data point can be replayed more than once.

Schaul et al. (2015) propose to sample experiences not uniformly from the replay memory but to prioritize experiences with a high TD error because the model could not predict these experiences well. They propose a proportional prioritization as well as a rank-based prioritization, where the latter is shown to be more robust. Using a proportional prioritization each entry in the replay memory has a transition priority p_i. The current transition $(S_t, A_t, R_{t+1}, S_{t+1})$ is stored in replay memory replacing the

oldest entry and is set to maximal priority $p_t = \max_i p_i$. Then a mini-batch is sampled according to their transition probabilities

$$j \sim P(j) = \frac{p_j^\alpha}{\sum_i p_i^\alpha}, \qquad \alpha \in [0, 1].$$

The model is then trained on the sampled transitions and the transition probabilities are updated using the observed TD errors. To prevent that experiences with a TD error of 0 are never replayed, a small positive constant will be added to all priorities p_i.

2.6 Policy Gradient Methods

The algorithms presented so far in this section have been using a value function, from which a policy was derived by acting greedily or ϵ-greedily. Now we present policy gradient methods, which directly parametrize the policy. Actor-critic algorithms combine value-based and policy gradient algorithms.

The idea of a policy-based algorithm is to directly parametrize the policy

$$\pi(a|s, \theta) = \mathbb{P}(A_t|S_t, \theta). \tag{64}$$

Two common choices for policies are the softmax policy, which is used with discrete action spaces, and the Gaussian policy used for continuous action spaces.

The softmax policy is

$$\pi(a|s, \theta) = \frac{\exp(h(s, a, \theta))}{\sum_{b \in \mathcal{A}} \exp(h(s, b, \theta))}. \tag{65}$$

The $h(s, a, \theta)$ are numerical preferences for each state-action pair and using the softmax formula these are then converted into probabilities. The preferences itself could be represented using e.g. a table or a linear combination of features

$$h(s, a, \theta) = \theta^T x(s, a), \tag{66}$$

using a feature vector $x(s, a)$.

A Gaussian policy is well suited for continuous actions, the policy is then the density of the normal distribution

$$\pi(a|s,\theta) = \frac{1}{\sigma(s,\theta)\sqrt{2\pi}} \exp(-\frac{(a-\mu(s,\theta))^2}{2\sigma(s,\theta)^2}), \tag{67}$$

where the mean could be parametrized as a linear function and the standard deviation as the exponential of a linear function (to ensure that $\sigma > 0$)

$$\mu(s,\theta) = \theta_\mu^T x(s) \qquad \text{and} \qquad \sigma(s,\theta) = \exp(\theta_\sigma^T x(s)). \tag{68}$$

To make the policy better a performance measure needs to be defined. A typical choice is to use the value of the starting state as a performance measure

$$J(\theta) = v_{\pi_\theta}(S_0).$$

The policy parameters are then updated using gradient ascent

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t), \tag{69}$$

Then, according to the policy gradient theorem (Sutton and Barto 2017), the gradient of $J(\theta)$ is equal to

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\gamma^t q_\pi(S_t, A_t) \nabla_\theta log\pi(A_t|S_t,\theta) \right]. \tag{70}$$

Often it is advantageous to include a baseline term in the gradient, which tells the agent how much better the action is in comparison to the average action value, e.g.

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\gamma^t (q_\pi(S_t, A_t) - v_\pi(S_t)) \nabla_\theta log\pi(A_t|S_t,\theta) \right]. \tag{71}$$

The term $q_\pi(S_t, A_t) - v_\pi(S_t)$ is often called the advantage function $A_\pi(S_t, A_t)$. $A_\pi(S_t, A_t)$ can be approximated by the error of TD(0),

$$A_\pi(S_t, A_t) \approx R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w).$$

This algorithm is called an actor critic. It combines value-based and policy-based reinforcement learning using a parametrized policy π_θ and a parametrized value function v_w or q_w. The policy is the *actor*, which defines which actions to take. The value function is the *critic*, which evaluates the actions.

The agent takes an action A_t according to its policy $\pi(a|S_t, \theta)$. Then the TD error δ_t is computed

$$\delta_t = R_{t+1} + \gamma\, \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)$$

The parameters of the critic are then updated by

$$w_{t+1} = w_t + \beta \delta_t\, \nabla_w \hat{v}(S_t, w)$$

and the policy's parameter by

$$\theta_{t+1} = \theta_t + \alpha \gamma^t \delta_t \nabla_\theta \log \pi(A_t|S_t, \theta).$$

α is here the learning rate for the policy and β the learning rate for the value function. When the TD error is positive the parameters of the value are adjusted, so it predicts a higher value for that state. The policy's parameters are then adjusted as to make the action leading to this positive error more likely.

Combination with eligibility traces is straightforward, just use one eligibility trace per weight of the actor and critic

$$E_t(w) = \gamma \lambda E_{t-1}(w) + \gamma^t \nabla_w \hat{v}(S_t, w) \tag{72}$$

$$E_t(\theta) = \gamma \lambda E_{t-1}(\theta) + \gamma^t \nabla_\theta log \pi(A_t|S_t, \theta) \tag{73}$$

and update the parameters accordingly

$$w_{t+1} = w_t + \beta\gamma^t \delta_t E_t(w) \tag{74}$$

$$\theta_{t+1} = \theta_t + \alpha\gamma^t \delta_t E_t(\theta). \tag{75}$$

A comparison of value-based and policy-based methods

Depending on the problem, it can be easier to approximate the value function or the policy. Policy-based methods are effective in high-dimensional and continuous action spaces, e.g. using a Gaussian policy, whereas value-based methods are not so effective there, because they need to do a maximization operation, which can be an expensive optimization problem itself. Policy gradient algorithms adjust the policy parameters smoothly, while an ϵ-greedy policy can change dramatically for a small change in the value function, i.e. if this changes which action has the highest action value. Policy-based methods can also find a stochastic optimal policy and are therefore useful in partially observable MDPs. Value-based methods can only output a (nearly) deterministic policy, e.g. the ϵ-greedy policy with respect to Q. (Sutton and Barto 2017)

3 R Package reinforcelearn

3.1 Installation

R is a language for statistical computations (*R: A Language and Environment for Statistical Computing* 2016). Functionality in R lives in packages, which needs to be installed and loaded. The goal of the `reinforcelearn` package is to make reinforcement learning available to R users.

The package is located at Github, from where it can be installed in R using the package `devtools` (Wickham and Chang 2017).

```
install.packages("devtools")
devtools::install_github("markdumke/reinforcelearn")
```

Then the package can be loaded.

```
library(reinforcelearn)
# Set a seed for reproducibility
set.seed(1)
```

3.2 How to create an environment?

What is an environment in reinforcelearn?

Environments in `reinforcelearn` are implemented as `R6` classes with certain methods and attributes. The environment can then be passed on to the algorithms using the `envir` argument.

There are some attributes of the `R6` class, which are essential for the interaction between environment and agent:

- `state`: The current state observation of the environment. Depending on the problem this can be anything, e.g. a scalar integer, a matrix or a list.
- `reward`: The current reward of the environment. It is always a scalar numeric value.
- `done`: A logical flag specifying whether an episode is finished.

- `n.steps`: Number of steps in the current episode. Will be reset to 0 when `reset` is called. Each time `step` is called it is increased by 1.

The interaction between agent and environment is done via the `reset` and `step` methods:

- `reset()`: Resets the environment, i.e. it sets the `state` attribute to a starting state and sets the `done` flag to `FALSE`. It is usually called at the beginning of an episode.
- `step(action)`: The basic interaction function between agent and environment. `step` is called with an action as an argument. It then takes the action and alters the `state` and `reward` attributes of the `R6` class. If the episode is done, e.g. a terminal state reached, the `done` flag is set to `TRUE`.

Note: All states and actions are numerated starting with 0!

The `makeEnvironment` function provides different ways to create an environment. It takes care of the creation of an `R6` class with the above mentioned attributes and methods.

Markov Decision Process

A Markov Decision Process (MDP) is a stochastic process, which is commonly used for reinforcement learning environments. When the problem can be formulated as a MDP, all you need to pass to `makeEnvironment` is the state transition array $P^a_{ss'}$ and reward matrix R^a_s of the MDP. The state transition array describes the probability of a transition from state s to state s' when taking action a. It is a 3-dimensional array with dimensions [number of states x number of states x number of actions], so for each action there is one state transition matrix. The reward matrix has the dimensions [number of states x number of actions], each entry is the expected reward obtained from taking action a in a state s.

We can create a simple MDP with 2 states and 2 actions with the following code.

```
# State transition array
P = array(0, c(2, 2, 2))
P[, , 1] = matrix(c(0.5, 0.5, 0.8, 0.2), 2, 2, byrow = TRUE)
P[, , 2] = matrix(c(0, 1, 0.1, 0.9), 2, 2, byrow = TRUE)
```

```
print(P)
#> , , 1
#>
#>      [,1] [,2]
#> [1,]  0.5  0.5
#> [2,]  0.8  0.2
#>
#> , , 2
#>
#>      [,1] [,2]
#> [1,]  0.0  1.0
#> [2,]  0.1  0.9
# Reward matrix
R = matrix(c(5, 10, -1, 2), 2, 2, byrow = TRUE)
print(R)
#>      [,1] [,2]
#> [1,]    5   10
#> [2,]   -1    2
env = makeEnvironment(transitions = P, rewards = R)
#> Warning in self$initializeMDP(transitions, rewards, initial.state, reset, :
#> There are no terminal states in the MDP!
```

We will get a warning that there are no terminal states in the MDP, i.e. an episode never ends in this MDP. Some algorithms assume that there is a terminal state, so we have to be careful, when we want to solve this. A terminal state has a probability of 1 remaining in this state. Here is an example.

```
P = array(0, c(2, 2, 2))
P[, , 1] = matrix(c(0.5, 0.5, 0, 1), 2, 2, byrow = TRUE)
P[, , 2] = matrix(c(0.1, 0.9, 0, 1), 2, 2, byrow = TRUE)
print(P)
#> , , 1
#>
#>      [,1] [,2]
#> [1,]  0.5  0.5
#> [2,]  0.0  1.0
```

```
#>
#> , , 2
#>
#>      [,1] [,2]
#> [1,]  0.1  0.9
#> [2,]  0.0  1.0
```

```
env = makeEnvironment(transitions = P, rewards = R)
print(env$terminal.states)
#> [1] 1
```

Every episode starts in some starting state. There are different ways to pass on the starting state in `makeEnvironment`. The simplest is to specify the `initial.state` argument with a scalar integer or an integer vector. When `initial.state` is a scalar every episode will start in this state. If `initial.state` is a vector then the starting state will be uniformly sampled from all elements of the vector. As a default the initial state will be sampled randomly from all non-terminal states.

```
env = makeEnvironment(transitions = P, rewards = R, initial.state = 0)
env$reset()
print(env)
#> Number of steps: 0
#> State: 0
#> Reward:
#> Done: FALSE
```

A different possibility is to specify a custom `reset` function, which takes no arguments and returns the starting state. This is a way to specify a custom probability distribution over starting states. If the starting state is a terminal state you will get a warning!

```
# Specify a custom probability distribution for the starting state.
reset = function() {
  p = c(0.2, 0.8)
  sample(0:1, prob = p, size = 1)
}
env = makeEnvironment(transitions = P, rewards = R, reset = reset)
env$reset()
```

```
#> Warning in env$reset(): The starting state is a terminal state!
print(env)
#> Number of steps: 0
#> State: 1
#> Reward:
#> Done: TRUE
```

The reward argument can also be a three-dimensional array, i.e. the reward can also depend on the next state.

```
R = array(0, c(2, 2, 2))
R[, 1, ] = 1
R[2, 2, 2] = 10
print(R)
#> , , 1
#>
#>      [,1] [,2]
#> [1,]    1    0
#> [2,]    1    0
#>
#> , , 2
#>
#>      [,1] [,2]
#> [1,]    1    0
#> [2,]    1   10

env = makeEnvironment(transitions = P, rewards = R)

env$reset()
env$step(1)
print(env)
#> Number of steps: 1
#> State: 1
#> Reward: 0
#> Done: TRUE
```

Instead of specifying a reward array you can also pass on a function `sampleReward`, which takes three arguments, the current state, action and next state and returns a scalar numeric reward. This way the reward of taking an action can be stochastic. Here is a simple example, where the reward is either 0 or sampled from a normal distribution depending on the next state and action.

```
sampleReward = function(state, action, n.state) {
  if (n.state == 0 & action == 1L) {
    0
  } else {
    rnorm(1)
  }
}
env = makeEnvironment(transitions = P, sampleReward = sampleReward)
env$reset()
env$step(0)
print(env)
#> Number of steps: 1
#> State: 0
#> Reward: 1.3297992629225
#> Done: FALSE
```

Gridworlds

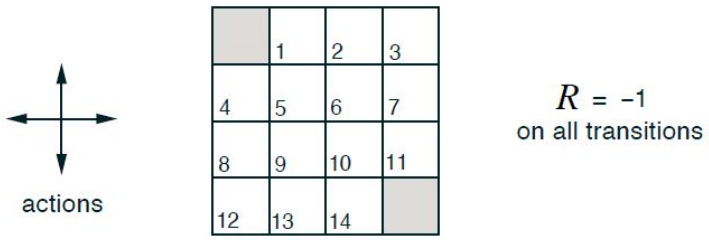
A gridworld is a simple navigation task with a discrete state and action space. The agent has to move through a grid from a start state to a goal state. Each episode starts in the start state and terminates if the agent reaches a goal state. States are always numerated row-wise starting with 0. Possible actions are the standard moves (left, right, up, down) or could also include the diagonal moves (leftup, leftdown, rightup, rightdown).

If an action would take the agent off the grid, the next state will be the nearest cell inside the grid. For each step the agent gets a reward, e.g. - 1, until it reaches a goal state, then the episode is done.

Gridworlds with different shapes, rewards and transition dynamics can be created with

the function `makeGridworld`. It computes the state transition array and reward matrix of the specified gridworld (because a gridworld is a MDP) and then internally calls `makeEnvironment`. Arguments from `makeEnvironment` can be passed on via the `...` argument, e.g. `initial.state`.

Here is an example of a 4x4 gridworld (Sutton and Barto 2017, Example 4.1) with the 4 standard actions and two terminal states in the lower right and upper left of the grid. Rewards are - 1 for every transition until reaching a terminal state.



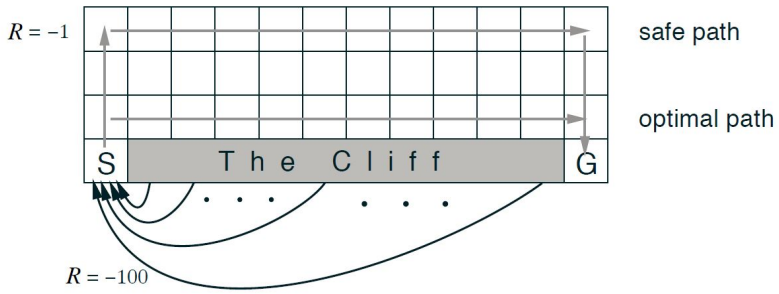
The following code creates this gridworld. This example gridworld is already included in the package and can be created with `gridworld()`.

```
# Gridworld Environment (Sutton & Barto (2017) Example 4.1)
env = makeGridworld(shape = c(4, 4), goal.states = c(0, 15))
print(env$states)
#>  [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
print(env$actions)
#> [1] 0 1 2 3

# Identical to the above call
env = gridworld()

# Same gridworld, but with diagonal moves
env = makeGridworld(shape = c(4, 4), goal.states = c(0, 15),
  diagonal.moves = TRUE)
print(env$actions)
#> [1] 0 1 2 3 4 5 6 7
```

In this gridworld actions will deterministically change the state, e.g. when going left from state 5 the new state will always be 4. A stochastic gridworld can be specified via

the `stochasticity` argument. Then the next state will be randomly sampled from all eight successor state with a probability `stochasticity`.

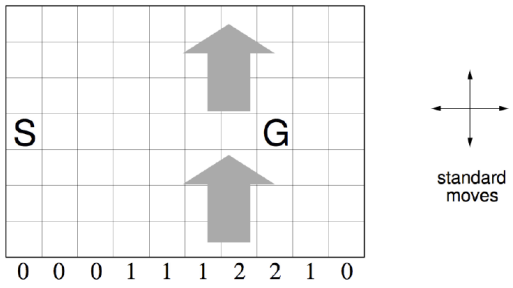
```
# Gridworld with 10% random transitions
env = makeGridworld(shape = c(4, 4), goal.states = c(0, 15), stochasticity = 0.1)
```



The cliff walking gridworld (Sutton and Barto 2017, Example 6.6) has a cliff in the lower part of the grid. Stepping into this cliff will result in a high negative reward of -100 and a transition back to the starting state in the lower left part of the grid. So the agent has to learn to avoid stepping into this cliff. Other transitions have the usual reward of -1. The optimal path is directly above the cliff, while the safe path runs at the top of the gridworld far away from the dangerous cliff.

In `makeGridworld` we can specify a cliff via the `cliff.states` argument and the reward when stepping into the cliff via `reward.cliff`. The states to which the agent transitions, when stepping into the cliff can be specified via `cliff.transition.states`.

```
# Cliff Walking (Sutton & Barto (2017) Example 6.6)
env = makeGridworld(shape = c(4, 12), goal.states = 47,
  cliff.states = 37:46, reward.step = - 1, reward.cliff = - 100,
  cliff.transition.states = 36, initial.state = 36)

# Identical to the above call
env = cliff()
```



The windy gridworld (Sutton and Barto 2017, Example 6.5) is a gridworld with shape 7x10. The agent will be pushed up a number of cells when transitioning into a column with an upward wind. The `wind` argument specifies the strength of this wind. It is an integer vector with the same size as the number of columns in the grid. E.g. going right from the state directly left to the goal, will push the agent to a state two cells above the goal. The reward for each step is -1.

```
# Windy Gridworld (Sutton & Barto (2017) Example 6.5)
env = makeGridworld(shape = c(7, 10), goal.states = 37,
  reward.step = - 1, wind = c(0, 0, 0, 1, 1, 1, 2, 2, 1, 0), initial.state = 30)

# Identical to the above call
env = windyGridworld()
```

OpenAI Gym Environments

OpenAI Gym (Brockman et al. 2016) is a toolkit for developing and comparing reinforcement learning algorithms. It provides a set of environments, which can be used as benchmark problems. The environments are implemented in python and can be acessed via the OpenAI Gym API. Have a look at https://gym.openai.com/envs for possible environments. To use this in R you need to install the dependencies listed here. You also need to install the R package `gym` (Hendricks 2016).

Then it is simple to use one of the existing OpenAI Gym environments. First you need to start the python server. Open a terminal and manually start the file `gym_http_server.py` inside the gym_http_api folder. You can also start the python server from R, here using a copy of the file included in the `reinforcelearn` package.

```
# Create an OpenAI Gym environment.
```

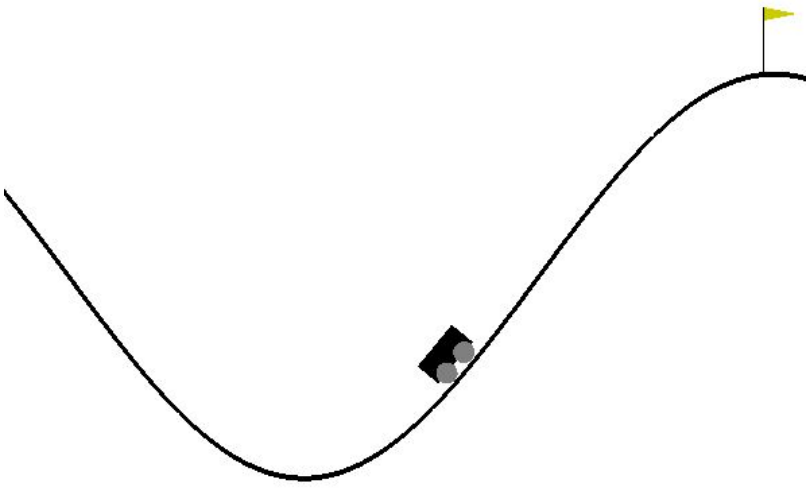
```
# Make sure you have Python and Gym installed.
# Start server from within R.
package.path = system.file(package = "reinforcelearn")
path2pythonfile = paste0(package.path, "/gym_http_server.py")
system2("python", args = path2pythonfile, stdout = NULL,
  wait = FALSE, invisible = FALSE)

env = makeEnvironment("MountainCar-v0")
```

The `render` argument specifies whether to render the environment. If `render = TRUE` a python window will open showing a graphical interface of the environment when calling the `step` method.

The `reset`, `step` and `close` method can then be used to sample experience. Here is an example running a random agent for 200 steps on the mountain car task.

```
env$reset()
for (i in 1:200) {
  action = sample(env$actions, 1)
  env$step(action)
}
env$close()
```

You should see a window opening showing the graphical interface.



Create your own environment

Some reinforcement learning problems cannot be formulated in the above way. Then it is necessary to create the environment yourself and pass it on to the algorithms. Make sure, that the environment is an `R6 class` with the necessary attributes and methods.

Here is a full list describing all attributes of the `R6` class created by `makeEnvironment`. Depending on the algorithm different of these attributes and methods may be necessary, e.g. `terminal.states`, `rewards` and `transitions` for model-based dynamic programming or `step`, `reset`, `state`, `previous.state`, `n.steps`, `reward` and `done`, if using a model-free algorithm.

Attributes:

- `state`: The current state observation of the environment. Depending on the problem this can be anything, e.g. a scalar integer, a matrix or a list.
- `reward`: The current reward of the environment. It is always a scalar numeric value.
- `done`: A logical flag specifying whether an episode is finished.
- `n.steps`: Number of steps in the current episode. Will be reset to 0 when `reset` is called. Each time `step` is called it is increased by 1.
- `previous.state`: The previous state of the environment. This is often the state which is updated in a reinforcement learning algorithm.
- `state.space`: One of `Discrete` or `Box`.
- `state.shape`: Number of state variables in a continuous state space.
- `state.space.bounds`: The bounds of a boxed state space, a list with lower and upper bound for each state variable as one list element.
- `states`: States (in a discrete state space). Numerated starting with 0.
- `terminal.states`: Terminal states in a Markov Decision Process. Will be derived from the `transition` argument in `makeEnvironment`.
- `n.states`: Number of states (for a discrete state space).
- `action.space`: One of `Discrete` or `Box`.

- `action.shape`: Number of action variables in a continuous action space.
- `action.space.bounds`: The bounds of a boxed action space, a list with lower and upper bound for each action variable as one list element.
- `actions`: Actions (in a discrete action space). Numerated starting with 0.
- `n.actions`: Number of actions (for a discrete action space).
- `transitions`: A state transition array (n.states x n.states x n.actions).
- `rewards`: A state reward matrix (n.states x n.actions).

Methods:

- `reset()`: Resets the environment, i.e. it sets the `state` attribute to a starting state and sets the `done` flag to `FALSE`. It is usually called at the beginning of an episode.
- `step(action)`: The basic interaction function between agent and environment. `step` is called with an action as an argument. It then takes the action and alters the `state` and `reward` attributes of the `R6` class. If the episode is done, e.g. a terminal state reached, the `done` flag is set to `TRUE`.
- `done()`: When using an OpenAI Gym environment this method closes the python window. Else it returns the `R6` class object unchanged, i.e. `self$close = function() {invisible(self)}`.

Mountain Car

The Mountain Car problem (Sutton and Barto 2017) is a simple episodic reinforcement learning task with a continuous state space and discrete action space. The goal is to drive an underpowered car up a steep slope. Because the car cannot accelerate fast enough, the optimal strategy consists of first going backwards to build enough momentum and then drive up the slope. The two-dimensional state space is characterized by the position and velocity of the car. These are updated due to the following equations:

$$\text{position}_{t+1} \leftarrow \text{bound}[\text{position}_t + \text{velocity}_{t+1}] \tag{76}$$

$$\text{velocity}_{t+1} \leftarrow \text{bound}[\text{velocity}_t + 0.001(A_t - 1) - 0.0025\cos(3 * \text{ position}_t)]. \tag{77}$$

The position is bounded in $[-1.2, 0.5]$, the velocity in $[-0.07, 0.07]$. When reaching the left position bound, the velocity will be set to 0. Each episode starts from a random position in the valley (position $\in [-0.6, -0.4]$) and a velocity of 0.

The original formulation of the problem has three actions: "push left", "no push" and "push right", which are encoded as 0, 1 and 2. The reward for each step is - 1 until the terminal state at the right mountain summit is reached.

Here is an example implementation of the mountain car environment.

```
mountainCar = R6::R6Class("MountainCar",
  public = list(
    action.space = "Discrete",
    actions = 0:2,
    n.actions = 3,
    state.space = "Box",
    state.space.bounds = list(c(-1.2, 0.5), c(-0.07, 0.07)),
    done = FALSE,
    n.steps = 0,
    state = NULL,
    previous.state = NULL,
    reward = NULL,
    velocity = NULL,
    position = NULL,

    reset = function() {
      self$n.steps = 0
      self$previous.state = NULL
      self$done = FALSE
      self$position = runif(1, - 0.6, - 0.4)
      self$velocity = 0
```

```
      self$state = matrix(c(self$position, self$velocity), ncol = 2)
      invisible(self)
    },

    step = function(action) {
      self$previous.state = self$state
      self$n.steps = self$n.steps + 1

      self$velocity = self$velocity + 0.001 * (action - 1) -
        0.0025 * cos(3 * self$position)
      self$velocity = min(max(self$velocity, self$state.space.bounds[[2]][1]),
        self$state.space.bounds[[2]][2])
      self$position = self$position + self$velocity
      if (self$position < self$state.space.bounds[[1]][1]) {
        self$position = self$state.space.bounds[[1]][1]
        self$velocity = 0
      }

      self$state = matrix(c(self$position, self$velocity), ncol = 2)
      self$reward = - 1
      if (self$position >= 0.5) {
        self$done = TRUE
        self$reward = 0
      }
      invisible(self)
    }
  )
)
```

We can then create a new instance of the mountain car class and pass this on the an algorithm. Here we will sample random actions and interact with the environment.

```
m = mountainCar$new()
set.seed(123456)
m$reset()
while (!m$done) {
```

```
  action = sample(m$actions, 1)
  m$step(action)
}
print(paste("Episode finished after", m$n.steps, "steps."))
#> [1] "Episode finished after 787 steps."
```

Note: The mountain car implementation above is already included in the package and can be called with the `mountainCar` function.

```
# The classical mountain car problem.
m = mountainCar()
m$reset()
m$step(1)
print(m)
#> Number of steps: 1
#> State: -0.454482559330265 -0.000518469774599052
#> Reward: -1
#> Done: FALSE
```

There is also a variant with a continuous action space (bounded in $[-1, 1]$)

```
# Mountain car with a continuous action space
m = mountainCar(action.space = "Continuous")
m$reset()
print(m)
#> Number of steps: 0
#> State: -0.454710142640397 0
#> Reward:
#> Done: FALSE
m$step(0.27541)
print(m)
#> Number of steps: 1
#> State: -0.454810022373572 -9.98797331752701e-05
#> Reward: -1
#> Done: FALSE
```

3.3 How to solve an environment?

Q-Learning, Sarsa, Expected Sarsa and Q(sigma)

Q(sigma), Q-Learning, Expected Sarsa and Sarsa build a family of reinforcement learning algorithms, which can be used to find the optimal action value function using the principle of generalized policy iteration.

In `reinforcelearn` you can use the `qlearning`, `sarsa`, `expectedSarsa` and `qSigma` functions. In the following we will train on a simple gridworld navigation task. The first argument of these algorithms is called `envir`, where the environment can be specified. The number of episodes can be specified via `n.episodes`.

```
# Gridworld environment
env = makeGridworld(shape = c(4, 4), goal.states = 15, initial.state = 0)

res = qlearning(env, n.episodes = 20)
# Note: to find a good policy we need to run more episodes.
```

These functions return the action value function Q (here a matrix).

```
print(round(res$Q1, 2))
#>        [,1]  [,2]  [,3]  [,4]
#>  [1,] -1.81 -1.82 -1.92 -1.75
#>  [2,] -1.52 -1.47 -1.41 -1.44
#>  [3,] -1.22 -1.20 -1.18 -1.22
#>  [4,] -1.06 -1.18 -1.19 -1.09
#>  [5,] -1.41 -1.44 -1.46 -1.38
#>  [6,] -1.30 -1.24 -1.32 -1.26
#>  [7,] -1.08 -1.12 -1.06 -1.08
#>  [8,] -0.95 -1.01 -0.95 -0.99
#>  [9,] -1.20 -1.18 -1.14 -1.21
#> [10,] -1.05 -1.05 -1.10 -1.10
#> [11,] -0.86 -0.87 -0.89 -0.89
#> [12,] -0.67 -0.78 -0.69 -0.65
#> [13,] -1.10 -1.09 -1.08 -1.02
#> [14,] -0.93 -0.98 -0.98 -1.02
```

```
#> [15,] -0.69 -0.65 -0.68 -0.80
#> [16,]  0.00  0.00  0.00  0.00
```

We can then find a policy by acting greedily with respect to the action value function.

```
# Values of each grid cell
state.values = matrix(apply(res$Q1, 1, max), ncol = 4, byrow = TRUE)
print(round(state.values, 1))
#>      [,1] [,2] [,3] [,4]
#> [1,] -1.8 -1.4 -1.2 -1.1
#> [2,] -1.4 -1.2 -1.1 -1.0
#> [3,] -1.1 -1.0 -0.9 -0.7
#> [4,] -1.0 -0.9 -0.7  0.0

# Policy: Subtract 1 to be consistent with action numeration in env
policy = max.col(res$Q1) - 1
print(matrix(policy, ncol = 4, byrow = TRUE))
#>      [,1] [,2] [,3] [,4]
#> [1,]    3    2    2    0
#> [2,]    3    1    2    0
#> [3,]    2    1    0    3
#> [4,]    3    0    1    1
```

They also return some statistics about learning behavior, e.g. the number of steps and returns per episode.

```
print(res$steps)
#>  [1] 60 57 13 43 59 11 56 31 80 66 11  8 48  9 37 17 63  9 10 14
```

Expected Sarsa can be used as an on-policy or an off-policy algorithm depending on the `target.policy` argument, which can be `"greedy"` or `"egreedy"` for ϵ-greedy.

```
# This is equivalent to qlearning(env):
res = expectedSarsa(env, target.policy = "greedy", n.episodes = 20)

# Expected Sarsa with an epsilon-greedy target policy:
res = expectedSarsa(env, target.policy = "egreedy", n.episodes = 20)
```

The Q(σ) algorithm (Asis et al. 2017) generalizes Sarsa, Expected Sarsa and Q-Learning.

Its parameter σ controls a weighting between Sarsa and Expected Sarsa. Q(1) is equal to Sarsa and Q(0) to Expected Sarsa.

```
res = qSigma(env, sigma = 0.5, n.episodes = 20)

# This is equivalent to Sarsa:
res = qSigma(env, sigma = 1, n.episodes = 20)

# This is equivalent to Q-Learning:
res = qSigma(env, sigma = 0, target.policy = "greedy", n.episodes = 20)
```

The hyperparameters of the algorithms can be specified as arguments, e.g. the discount factor γ via `discount`, the learning rate α via `learning.rate` and the exploration factor ϵ via the `epsilon` argument. These parameters can also be adjusted over time by specifying an `update` function, e.g. `updateEpsilon`. The `update` function takes two arguments, the old value of the parameter and the number of episodes finished. It returns the updated parameter, e.g. a decreased learning rate. The update functions are called after each episode is finished.

```
res = qlearning(env, epsilon = 0.2, learning.rate = 0.5,
  discount = 0.99, n.episodes = 20)

# Decay epsilon over time. Every 10 episodes epsilon will be halfed.
decayEpsilon = function(epsilon, i) {
  if (i %% 10 == 0) {
    epsilon = epsilon * 0.5
  }
  epsilon
}

res = qlearning(env, epsilon = 0.5, n.episodes = 20,
  updateEpsilon = decayEpsilon)
```

The action value function will be initialized to 0. But you can also pass on an initial value function via the `initial.value` argument.

```
Q = matrix(100, nrow = env$n.states, ncol = env$n.actions)
```

```
res = qlearning(env, n.episodes = 5, initial.value = Q)

# After 5 episodes the Q values will still be similar to 100.
print(matrix(round(apply(res$Q1, 1, max), 1), ncol = 4, byrow = TRUE))
#>      [,1] [,2] [,3]  [,4]
#> [1,] 99.4 99.5 99.6  99.6
#> [2,] 99.5 99.6 99.7  99.8
#> [3,] 99.6 99.7 99.8  99.9
#> [4,] 99.7 99.7 99.8 100.0
```

Function Approximation

So far the value function has been represented as a table (number of states x number of actions). In many interesting problems there are lots of states and actions or the space is continuous. Then it is impractical to store a tabular value function and to slow to update state-action pairs individually. The solution is to approximate the value function with a function approximator, e.g. a linear combination of features.

In the following we will have a look at the mountain car problem, where the goal is to drive an underpowered car up a steep hill. The state space is continuous in two dimensions, the position and velocity of the car, each bounded in some interval. There are three different actions: push back (0), do nothing (1) and push forward (2).

```
env = mountainCar()
print(env$state.space)
#> [1] "Box"
print(env$state.space.bounds)
#> [[1]]
#> [1] -1.2  0.5
#>
#> [[2]]
#> [1] -0.07  0.07

env$reset()
print(env$state)
```

```
#>             [,1] [,2]
#> [1,] -0.5105886    0
```

We will solve this environment using linear function approximation. With linear function approximation the action value function is represented as

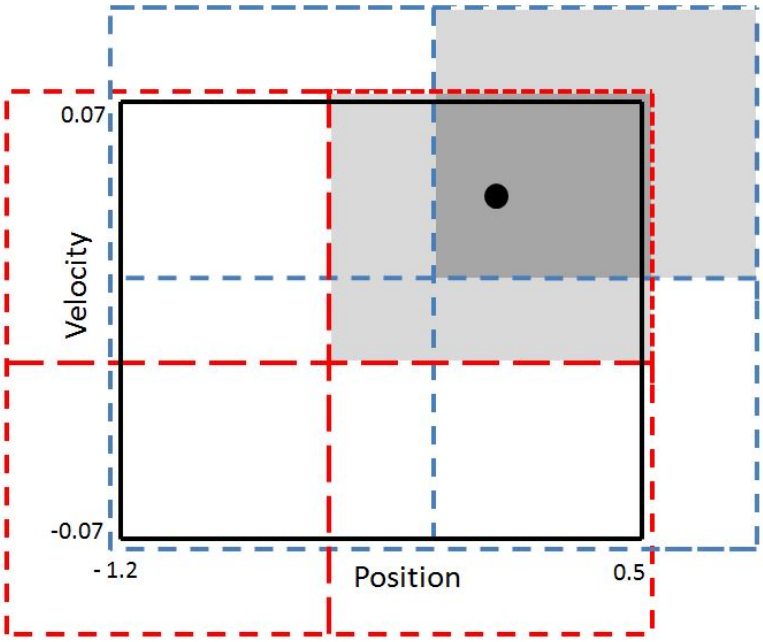
$$\hat{q}(S_t, A_t, w) = x(S_t)^T w = \sum_{j=1}^{n} x_j(S_t)\, w_j$$

and updated by gradient descent. There is a distinct weight vector per action.

Preprocessing the state

The raw state observation returned from the environment must be preprocessed using the `preprocessState` argument. This function takes the state observation as input and returns a preprocessed state which can be directly used by the function approximator. To use a tabular value function `preprocessState` must return an integer value between [0, number of states - 1]. For linear function approximation the output of `preprocessState` must be a matrix with one row. For a neural network you have to make sure that the dimensions of the preprocessed state and the neural network match, so that `model$predict(preprocessState(envir$state))` works.

But how to get a good feature vector from the state observation? One idea is to use grid tiling (Sutton and Barto 2017) to aggregate the state space. Each tiling is a grid which overlays the state space. A state observation then falls into one tile per tiling and we will use as many weights as there are tiles. The feature vector is then just a one-hot vector of all active tiles.



We can define a function, which takes the original state observation as an input and returns a preprocessed state observation.

```
# Define preprocessing function (we use grid tiling)
n.tilings = 8
max.size = 4096
iht = IHT(max.size)

position.max = env$state.space.bounds[[1]][2]
position.min = env$state.space.bounds[[1]][1]
velocity.max = env$state.space.bounds[[2]][2]
velocity.min = env$state.space.bounds[[2]][1]
position.scale = n.tilings / (position.max - position.min)
velocity.scale = n.tilings / (velocity.max - velocity.min)

# Scale state first, then get active tiles and return n hot vector
gridTiling = function(state) {
  state = c(position.scale * state[1], velocity.scale * state[2])
  active.tiles = tiles(iht, 8, state)
  makeNHot(active.tiles, max.size, out = "vector")
}
```

We can then pass this function to the `preprocessState` argument in `qlearning`. Via the `fun.approx` argument we tell the algorithm to use a linear combination of features to approximate the value function. Currently `fun.approx` supports `table`, `linear` and `neural.network`.

```
res = qlearning(env, fun.approx = "linear",
  preprocessState = gridTiling, n.episodes = 20)
print(res$steps)
#>  [1]  658 1117  538  610  326  347  218  299  269  400  222  295  234  228
#> [15]  328  155  270  252  223  185
```

Neural Network

To use a neural network you have to specify a `keras` model. Here is an example:

```
env = makeGridworld(c(4, 4), goal.states = 15, initial.state = 0)

# A one-hot feature vector
makeOneHot = function(state) {
  one.hot = matrix(rep(0, env$n.states), nrow = 1)
  one.hot[1, state + 1] = 1
  one.hot
}

# Define keras model
library(keras)
model = keras_model_sequential()
model %>% layer_dense(units = env$n.actions, activation = 'linear',
  input_shape = c(env$n.states))

res = qSigma(env, fun.approx = "neural.network", model = model,
  preprocessState = makeOneHot, n.episodes = 20)
```

Note that neural network training can be slow because at each step the Keras API is called.

Extensions

There are several extensions of the algorithms, which can improve learning behavior.

Eligibility Traces

Eligibility traces assign credit for the error back to all previously visited states and actions (Sutton and Barto 2017). The trace decay parameter λ and the type of the eligibility trace can be specified as arguments.

If `eligibility.type = 1` a replacing trace is used, if `eligibility.type = 0` an accumulating trace (Singh and Sutton 1996). Intermediate values are also possible.

```
env = makeGridworld(c(4, 4), goal.states = 15, initial.state = 0)

# Sarsa with replacing traces
res = sarsa(env, lambda = 0.9, eligibility.type = 1, n.episodes = 20)
print(res$steps)
#>  [1] 60 73 17 26 10 28 16 15 32 18 24 38 23 16  8 11 20  6  8 12
```

Double Learning

The idea of double learning (H. V. Hasselt 2010) is to decouple action selection and action evaluation.

To use Double Learning with `qSigma`, `qlearning`, `sarsa` and `expectedSarsa` just pass `double.learning = TRUE` to the algorithm.

```
res = expectedSarsa(env, double.learning = TRUE, n.episodes = 20)
print(res$steps)
#>  [1]  61  14  31  58  60  58  40  28 106  12  39  32  30  12 104  14  42
#> [18]  11  30  55
```

Experience Replay

When using function approximation in reinforcement learning training can be instable because subsequent state observations are often highly correlated and we train on these states the order they are experienced. Experience replay (Mnih et al. 2013) is a simple

idea to break these correlations and stabilize learning. Instead of training on a simple observation at each time step the algorithm trains now on more than one observation sampled randomly from a replay memory, which stores all previously visited states and actions. Because the observations are trained on in a random order correlations are much smaller .

In `reinforcelearn` experience replay can be used by passing on a list of experiences to the `replay.memory` argument. Each list entry is itself a list with the entries `state`, `action`, `reward` and `next.state`. `state` and `next.state` should have been preprocessed, e.g. by calling `preprocessState(state)`. A different possibility is to specify the `replay.memory.size` argument, which will then be initialized with experiences generated by a random policy. The number of experiences trained on at each step is controled via the `batch.size` argument.

When experiencing a new transition the algorithm replaces the oldest entry in the replay memory by the new transition.

```
# Fill a replay memory of size 100 on the gridworld task.
memory = vector("list", length = 100)
env$reset()
for (i in 1:100) {
  if (env$done) {
    env$reset()
  }
  action = sample(env$actions, size = 1)
  env$step(action)
  memory[[i]] = list(state = env$previous.state, action = action,
    reward = env$reward, next.state = env$state)
}
print(memory[[1]])
#> $state
#> [1] 0
#>
#> $action
#> [1] 2
#>
```

```
#> $reward
#> [1] -1
#>
#> $next.state
#> [1] 0

# Pass on replay memory.
res = sarsa(env, replay.memory = memory, batch.size = 32, n.episodes = 20)

# Specify replay memory size, replay memory will be filled internally.
res = sarsa(env, replay.memory.size = 100, batch.size = 32, n.episodes = 20)
print(res$steps)
#>  [1] 55 24 20 17 10  8  9  6  6  6 12 11 14 13 23 11  8 13  6 14
```

As a default experiences will be randomly sampled from the replay memory. A prioritized experience replay prioritizes experiences with a high error (Schaul et al. 2015).

When $\alpha = 0$ (the default) experiences are sampled with equal probability else experiences with a high error have a higher probability of being sampled. A small positive constant θ is added to each priority to prevent that experiences with an error of 0 are never replayed.

```
# Prioritized experience replay
res = sarsa(env, replay.memory.size = 100, batch.size = 32,
  n.episodes = 20, alpha = 0.5, theta = 0.01)
print(res$steps)
#>  [1] 16 40 64  6  7  6 10 18 12  6  6  8  7  6  9 14  6  6  7  7
```

There are other algorithms implemented in the package which will be described in the following.

TD(lambda)

The TD(lambda) algorithm is used to evaluate a fixed policy. In `reinforcelearn` `td` can be used with a tabular or linear function approximation and with eligibility traces.

`td` takes a `policy` argument, which is the policy to evaluate. In the tabular case this

is just a matrix (number of states x number of actions) with the probabilities of each action given a state. For the linear function approximation `policy` must be a function, which returns an action given a preprocessed state observation. You can specify a maximal number of steps or episodes, so `td` can be used with both continuing and episodic environments.

Here we will solve a random walk task (Sutton and Barto 2017, Example 6.2).

```
# Random Walk Task (Sutton & Barto Example 6.2)
P = array(dim = c(7, 7, 2))
P[, , 1] = matrix(c(rep(c(1, rep(0, 6)), 2), c(0, 1, rep(0, 5)),
  c(0, 0, 1, rep(0, 4)), c(rep(0, 3), 1, rep(0, 3)), c(rep(0, 4), 1, rep(0, 2)),
  c(rep(0, 6), 1)), ncol = 7, byrow = TRUE)
P[, , 2] = matrix(c(c(1, rep(0, 6)), c(0, 0, 1, rep(0, 4)),
  c(rep(0, 3), 1, rep(0, 3)), c(rep(0, 4), 1, rep(0, 2)),
  c(rep(0, 5), 1, 0), c(rep(0, 6), 1), c(rep(0, 6), 1)), ncol = 7, byrow = TRUE)
R = matrix(c(rep(0, 12), 1, 0), ncol = 2)
env = makeEnvironment(transitions = P, rewards = R, initial.state = 3)

# Uniform random policy
random.policy = matrix(1 / env$n.actions, nrow = env$n.states,
  ncol = env$n.actions)

# Estimate state value function with TD(0)
res = td(env, random.policy, n.episodes = 20, lambda = 0.5)
print(res$V)
#> [1] 0.0000000 0.0334874 0.1075444 0.1840309 0.3581764 0.5554538 0.0000000
```

Dynamic Programming

Dynamic programming (Sutton and Barto 2017) is a class of solution methods solving a MDP not by interaction but by iterative computations using the state transition array and reward matrix. It can therefore only be applied when the model of the MDP is known.

In R we can evaluate a policy with dynamic programming with the following code:

```
# Set up gridworld problem
env = gridworld()

# Define uniform random policy, take each action with equal probability
random.policy = matrix(1 / env$n.actions, nrow = env$n.states,
  ncol = env$n.actions)

# Evaluate this policy
res = evaluatePolicy(env, random.policy, precision = 0.01)
print(round(matrix(res$v, ncol = 4, byrow = TRUE)))
#>      [,1] [,2] [,3] [,4]
#> [1,]    0  -14  -20  -22
#> [2,]  -14  -18  -20  -20
#> [3,]  -20  -20  -18  -14
#> [4,]  -22  -20  -14    0
```

In theory it converges to the true values, but in practise we have to stop iteration before that. You can either specify a maximal number of iterations via the `n.iter` argument or a `precision` term, then the evaluation stops if the change in two subsequent values is less than `precision` for every state. You can specify an initial value function via the `v` argument. Note that the values of all terminal states must be 0 else the algorithm does not work.

Policy iteration tries to find the best policy in the MDP by iterating between evaluating and improving a policy.

```
# Find optimal policy using Policy Iteration
res = iteratePolicy(env)
print(round(matrix(res$v, ncol = 4, byrow = TRUE)))
#>      [,1] [,2] [,3] [,4]
#> [1,]    0   -1   -2   -3
#> [2,]   -1   -2   -3   -2
#> [3,]   -2   -3   -2   -1
#> [4,]   -3   -2   -1    0
```

You can specify an initial policy else the initial policy will be the uniform random policy.

`iteratePolicy` stops if the policy does not change in two subsequent iterations or if the specified number of iterations is exhausted. For the policy evaluation step in policy iteration the same stop criteria as in `evaluatePolicy` are applied via the `precision.eval` and `n.iter.eval` can be passed on.

Value iteration evaluates each policy only once and then immediately improves the policy by acting greedily.

```
# Find optimal policy using Value Iteration
res = iterateValue(env, n.iter = 100)
print(round(matrix(res$v, ncol = 4, byrow = TRUE)))
#>      [,1] [,2] [,3] [,4]
#> [1,]    0   -1   -2   -3
#> [2,]   -1   -2   -3   -2
#> [3,]   -2   -3   -2   -1
#> [4,]   -3   -2   -1    0
```

`iterateValue` runs until the improvement in the value function in two subsequent steps is smaller than the given precision in all states or if the specified number of iterations is exhausted.

`evaluatePolicy`, `iteratePolicy` and `iterateValue` return a list with state value function, action value function and policy.

Actor Critic

An actor critic is a policy-based reinforcement learning algorithm, which parametrizes value function and policy (Sutton and Barto 2017). In `reinforcelearn` a simple advantage actor critic is implemented, which uses the td error of the state value function as a critic.

The policy can be a softmax policy for discrete actions or a gaussian policy for a continuous action space.

There are now two learning rates α and β, one for the critic and one for the actor.

```
env = mountainCar()
```

```
# Linear function approximation and softmax policy
res = actorCritic(env, fun.approx = "linear",
  preprocessState = gridTiling, n.episodes = 20)
print(res$steps)
#>  [1] 1166  532  378  227  314  236  244  245  199  155  164  155  230  185
#> [15]  195  156  153  154  127  222
```

With a gaussian policy we can also solve problems with a continuous action space.

Here we will solve a continuous version of the mountain car problem, where the action is a real number.

```
# Mountain Car with continuous action space
env = mountainCar(action.space = "Continuous")


# Linear function approximation and gaussian policy
set.seed(123)
res = actorCritic(env, fun.approx = "linear", policy = "gaussian",
  preprocessState = gridTiling, n.episodes = 20)
print(res$steps)
#>  [1] 780 319 317 157 149 194 153 132 106 151 106 136 110  98  96 101 144
#> [18]  96 103 165
```

The actor critic can be used with eligibility traces, then there are separate eligibility traces for the policy parameters and the value function parameters, which can be decayed by a different factor λ.

```
# Cliff walking environment
rewardFun = function(state, action, n.state) {
  if (n.state %in% 37:46) {
    return(- 100)
  } else {
    return(- 1)
  }
}
env = makeGridworld(shape = c(4, 12), goal.states = 47,
  cliff.states = 37:46, reward.step = - 1, reward.cliff = - 100,
```

```
  cliff.transition.done = TRUE, initial.state = 36, sampleReward = rewardFun)

res = actorCritic(env, n.episodes = 20, lambda.actor = 0.5, lambda.critic = 0.8)
```

Multi-armed Bandit

There are also solution methods for simple multi-armed bandit problems.

In the following we will consider an example bandit with four different actions. For each action the reward will be sampled from a probability distribution. The reward of the first action is sampled from a normal distribution with mean 1 and standard deviation 1, the second action from a normal distribution with mean 2 and standard deviation 4, the third action from a uniform distribution with minimum 0 and maximum 5 and the fourth action from an exponential distribution with rate parameter 0.25. Therefore the fourth action is the best with an expected reward of 4.

To solve this bandit problem we need to specify the reward function,

```
# Define reward function
rewardFun = function(action) {
  if (action == 0) {
    reward = rnorm(1, mean = 1, sd = 1)
  }
  if (action == 1) {
    reward = rnorm(1, mean = 2, sd = 4)
  }
  if (action == 2) {
    reward = runif(1, min = 0, max = 5)
  }
  if (action == 3) {
    reward = rexp(1, rate = 0.25)
  }
  reward
}
```

To solve the bandit, i.e. to find out, which action returns the highest reward, we can use the `bandit` function. There are several different action selection methods implemented,

e.g. `greedy`, `epsilon-greedy`, `UCB` and `gradient-bandit`.

```
# Greedy action selection.
bandit(rewardFun, n.actions = 4, n.episodes = 1000,
  action.selection = "greedy")
#> [1] -0.1976585 -1.3866575  2.1102407  3.9465064

# Epsilon-greedy action selection.
bandit(rewardFun, n.actions = 4, n.episodes = 1000,
  action.selection = "egreedy", epsilon = 0.2)
#> [1] 1.032849 1.929346 2.711898 4.041045

# Upper-confidence bound action selection.
bandit(rewardFun, n.actions = 4, n.episodes = 1000,
  action.selection = "UCB", C = 2)
#> [1] 0.4598108 1.1766058 2.4985695 1.4298745

# Gradient-bandit algorithm.
bandit(rewardFun, n.actions = 4, n.episodes = 10000,
  action.selection = "gradientbandit", alpha = 0.1)
#> [1] 5.379818e-05 3.348153e-05 2.775565e-04 9.996352e-01
```

In the `bandit` function we can specify the argument `initial.value` which sets all Q values initially to this number. Additionally we can assign a confidence to this initial value via the `initial.visits` argument. A value of 10 for example means that the algorithm has already seen 10 rewards for each action with an average value of `initial.value`.

```
# Greedy action selection with optimistic initial values.
bandit(rewardFun, n.actions = 4, n.episodes = 1000,
  action.selection = "greedy",
  initial.value = 5, initial.visits = 100)
#> [1] 3.987659 3.984982 3.988113 4.005252
```

3.4 Comparison with other packages

In this section we compare the `reinforcelearn` package with other packages available on the web.

MDPtoolbox

From the description file of the `MDPtoolbox` package (Chades et al. 2017):

> The Markov Decision Processes (MDP) toolbox proposes functions related to the resolution of discrete-time Markov Decision Processes: finite horizon, value iteration, policy iteration, linear programming algorithms with some variants and also proposes some functions related to Reinforcement Learning.

The following algorithms can be found in both packages and will be compared in the following:

- Dynamic Programming: Iterative Policy Evaluation, Policy Iteration, Value Iteration
- Reinforcement Learning: TD(0), Q-Learning

In the `MDPtoolbox` the transition array and reward array must be passed on to the algorithms. Here is a comparison of functions in both packages on a simple example MDP.

```
library(MDPtoolbox)
library(reinforcelearn)

# MDP
P = array(0, c(2,2,2))
P[, , 1] = matrix(c(0.5, 0.5, 0.8, 0.2), 2, 2, byrow = TRUE)
P[, , 2] = matrix(c(0, 1, 0.1, 0.9), 2, 2, byrow = TRUE)
R = matrix(c(5, 10, -1, 2), 2, 2, byrow = TRUE)

env = makeEnvironment(transitions = P, rewards = R)

# Iterative Policy Evaluation
```

```
mdp_eval_policy_iterative(P, R, 0.8, policy = c(2, 1),
  epsilon = 0, max_iter = 1000, V0 = c(0, 0))
evaluatePolicy(env, policy = matrix(c(0, 1, 1, 0), ncol = 2),
  discount = 0.8, n.iter = 1000)

# Policy Iteration
mdp_policy_iteration(P, R, discount = 0.9,
  max_iter = 1000, policy0 = c(1, 1), eval_type = 1)
iteratePolicy(env, n.iter = 1000, discount = 0.9,
  policy = matrix(c(1, 1, 0, 0), ncol = 2))

# Value Iteration
mdp_value_iteration(P, R, discount = 0.9, max_iter = 1000, epsilon = 0)
iterateValue(env, discount = 0.9, n.iter = 1000)

# TD(0)
mdp_eval_policy_TD_0(P, R, discount = 0.9, policy = c(1, 2), N = 10000)
td(env, discount = 0.9, policy = matrix(c(1, 0, 0, 1), ncol = 2),
  n.steps = 10000)
```

Policies in `MDPtoolbox` are always deterministic, while policies in `reinforcelearn` can be stochastic. Therefore policies in `MDPtoolbox` are represented as an integer vector (which action to take in a state), while a policy in `reinforcelearn` is always a matrix (n.states x n.actions), where each entry is the probability of this action in this state.

TD(0) and Q-Learning are implemented in `MDPtoolbox` in their basic form without eligibility traces, double learning and other extensions.

ReinforcementLearning

From the description file of the `ReinforcementLearning` package (Proellochs and Feuerriegel 2017):

> Performs model-free reinforcement learning in R. This implementation enables the learning of an optimal policy based on sample sequences consisting of states, actions and rewards. In addition, it supplies multiple predefined

reinforcement learning algorithms, such as experience replay.

```
data = sampleGridSequence(1000)

# Setting reinforcement learning parameters
control = list(alpha = 0.1, gamma = 0.1, epsilon = 0.1)

# Performing reinforcement learning
model = ReinforcementLearning(data, s = "State", a = "Action",
  r = "Reward", s_new = "NextState", control = control)
```

The package `ReinforcementLearning` implements a reinforcement learning algorithm based on experience replay. The data must be a sequence of states, actions and rewards and must be specified as an argument to the function. It is not a full reinforcement learning agent, but just one step of experience replay, which might be used with an online performing agent. It is unclear from the documentation of the package `ReinforcementLearning`, which algorithmic procedure is used (Q-Learning?). This is the main difference to the `reinforcelearn` package, where a full reinforcement learning agent can be trained sequentially, while interacting with an environment, e.g. using an experience replay update at each time step.

Declaration

This is to certify that

i. The thesis comprises only my original work except where indicated,
ii. Due acknowledgement has been made in the text to all other material used.

List of Figures

4 References

Asis, Kristopher De, J. Fernando Hernandez-Garcia, G. Zacharias Holland, and Richard S. Sutton. 2017. "Multi-Step Reinforcement Learning: A Unifying Algorithm." *CoRR* abs/1703.01327. http://arxiv.org/abs/1703.01327.

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. "OpenAI Gym." *CoRR* abs/1606.01540. http://arxiv.org/abs/1606.01540.

Chades, Iadine, Guillaume Chapron, Marie-Josee Cros, Frederick Garcia, and Regis Sabbadin. 2017. *MDPtoolbox: Markov Decision Processes Toolbox*. https://CRAN.R-project.org/package=MDPtoolbox.

Ganger, Michael, Ethan Duryea, and Wei Hu. 2016. "Double Sarsa and Double Expected Sarsa with Shallow and Deep Learning." *Journal of Data Analysis and Information Processing* 4: 159–76. doi:10.4236/jdaip.2016.44014.

Hasselt, Hado V. 2010. "Double Q-Learning." In *Advances in Neural Information Processing Systems 23*, edited by J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, 2613–21. Curran Associates, Inc. http://papers.nips.cc/paper/3964-double-q-learning.pdf.

Hendricks, Paul. 2016. *Gym: Provides Access to the Openai Gym Api*. https://github.com/paulhendricks/gym-R.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. "Playing Atari with Deep Reinforcement Learning." *CoRR* abs/1312.5602. http://arxiv.org/abs/1312.5602.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, et al. 2015. "Human-Level Control Through Deep Reinforcement Learning." *Nature* 518.

Proellochs, Nicolas, and Stefan Feuerriegel. 2017. *ReinforcementLearning: Model-Free Reinforcement Learning*. https://CRAN.R-project.org/package=ReinforcementLearning.

R: A Language and Environment for Statistical Computing. 2016. Vienna, Austria: R

Foundation for Statistical Computing. https://www.R-project.org/.

Rummery, G. A., and M. Niranjan. 1994. "On-Line Q-Learning Using Connectionist Systems."

Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver. 2015. "Prioritized Experience Replay." *CoRR* abs/1511.05952. http://arxiv.org/abs/1511.05952.

Seijen, Harm van, Hado van Hasselt, Shimon Whiteson, and Marco Wiering. 2009. "A Theoretical and Empirical Analysis of Expected Sarsa." In *ADPRL 2009: Proceedings of the Ieee Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, 177–84.

Seijen, Harm van, Ashique Rupam Mahmood, Patrick M. Pilarski, Marlos C. Machado, and Richard S. Sutton. 2015. "True Online Temporal-Difference Learning." *CoRR* abs/1512.04087. http://arxiv.org/abs/1512.04087.

Silver, David. 2015. "University Lecture." University College London; http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html.

Silver, David, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, et al. 2016. "Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature* 529 (7587): 484–89. doi:10.1038/nature16961.

Singh, Satinder P., and Richard S. Sutton. 1996. "Reinforcement Learning with Replacing Eligibility Traces." *Machine Learning* 22 (1): 123–58. doi:10.1007/BF00114726.

Sutton, Richard S., and Andrew G. Barto. 2017. "Reinforcement Learning : An Introduction." Cambridge, MA, USA: http://incompleteideas.net/sutton/book/the-book-2nd.html; MIT Press.

Watkins, Christopher John Cornish Hellaby. 1989. "Learning from Delayed Rewards." PhD thesis, King's College, Cambridge.

Wickham, Hadley, and Winston Chang. 2017. *Devtools: Tools to Make Developing R Packages Easier*. https://CRAN.R-project.org/package=devtools.

A Appendix

Double Q(σ) and Q(σ, λ) Unifying reinforcement learning control algorithms

Markus Dumke
Department of Statistics
Ludwig-Maximilians-Universität München
markus.dumke@campus.lmu.de

Abstract

Temporal-difference (TD) learning is an important field in reinforcement learning. The most used TD control algorithms are probably Sarsa and Q-Learning. While Sarsa is an on-policy algorithm, Q-Learning learns off-policy. The Q(σ) algorithm (Sutton and Barto (2017)) unifies both. This paper extends the Q(σ) algorithm to an on-line multi-step algorithm Q(σ, λ) using eligibility traces and introduces Double Q(σ) as the extension of Q(σ) to double learning.

1 Introduction

Reinforcement Learning is a field of machine learning addressing the problem of sequential decision making. It is formulated as an interaction of an agent and an environment over a number of discrete time steps t. At each time step the agent chooses an action A_t based on the environment's state S_t. The environment takes A_t as an input and returns the next state observation S_{t+1} and reward R_{t+1}, a scalar numeric feedback signal.

The agent is thereby following a policy π, which is the behavior function mapping a state to action probabilities

$$\pi(a|s) = P(A_t = a|S_t = s). \tag{1}$$

The agent's goal is to maximize the return G_t which is the sum of discounted rewards,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{T-1} \gamma^k R_{t+1+k}, \tag{2}$$

where $\gamma \in [0, 1]$ is the discount factor and T is the length of the episode or infinity for a continuing task.

While rewards are short-term signals about the goodness of an action, values represent the long-term value of a state or state-action pair. The action value function $q_\pi(s, a)$ is defined as the expected return taking action a from state s and thereafter following policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]. \tag{3}$$

Value-based reinforcement learning is concerned with finding the optimal action value function $q_* = \max_\pi q_\pi$. Temporal-difference learning is a class of model-free methods which estimates q_π from sample transitions and iteratively updates the estimated values using observed rewards and estimated values of successor actions. At each step an update of the following form is applied:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\, \delta_t, \tag{4}$$

where Q is an estimate of q_π, α is the learning rate (also called step size) and δ_t is the TD error, the difference between our current estimate and a newly computed target value. The following TD control algorithms can all be characterized by their different form of TD error.

When the action values Q are represented as a table we call this tabular reinforcement learning, else we speak of approximate reinforcement learning, e.g. when using a neural network to compute the action values. For sake of simplicity the following analysis is done for tabular reinforcement learning but can be easily extended to function approximation.

2 TD control algorithms: From Sarsa to Q(σ)

Sarsa (Rummery and Niranjan (1994)) is a temporal-difference learning algorithm which samples states and actions using an epsilon-greedy policy and then updates the Q values using Equation 4 with the following TD error

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t). \tag{5}$$

The term $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ is called the TD target and consists of the reward plus the discounted value of the next state and next action.

Sarsa is an on-policy method, i.e. the TD target consists of $Q(S_{t+1}, A_{t+1})$, where A_{t+1} is sampled using the current policy. In general the policy used to sample the state and actions - the so called behaviour-policy μ - can be different from the target policy π, which is used to compute the TD target. If behaviour and target policy are different we call this off-policy learning. An example for an off-policy TD control algorithm is the well known Q-Learning algorithm proposed by Watkins (1989). As in Sarsa states and actions are sampled using an exploratory behaviour policy, e.g. an ϵ-greedy policy, but the TD target is computed using the greedy policy with respect to the current Q values. The TD error of Q-Learning is

$$\delta_t = R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t). \tag{6}$$

Expected Sarsa generalizes Q-Learning to arbitrary target policies. The TD error is

$$\delta_t = R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t). \tag{7}$$

The current state-action pair is updated using the expectation of all subsequent action values with respect to the action value. You can easily see that Q-Learning is just a special case of Expected Sarsa if π is the greedy policy with respect to Q:

$$\pi(a|s) = \begin{cases} 1 & \text{if } a = \text{argmax}_a\, Q(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Then $\sum_{a'} \pi(a'|S_{t+1})Q(S_{t+1}, a')$ is equal to $\max_{a'} Q(S_{t+1}, a')$ because all non-greedy actions will have a probability of 0 and the sum reduces to the Q value of the greedy action, which is the maximum Q value.

Of course Expected Sarsa could also be used as an on-policy algorithm if the target policy is chosen to be the same as the behaviour policy (Van Seijen et al. (2009)).

Sutton and Barto (2017) propose a new TD control algorithm called Q(σ) which unifies Sarsa and Expected Sarsa. The TD target of this new algorithm is a weighted mean of the Sarsa and Expected Sarsa TD targets, where the parameter σ controls the weighting. When $\sigma = 1$ Q(σ) is equal to Sarsa, when $\sigma = 0$ Q(σ) is equal to Expected Sarsa and when using $\sigma = 0$ and a greedy target policy Q(σ) is equal to Q-Learning. For intermediate values of σ new algorithms are obtained, which can have better performance (Asis et al. (2017)).

The TD error of Q(σ) is

$$\delta_t = R_{t+1} + \gamma(\sigma Q(S_{t+1}, A_{t+1}) + (1-\sigma)\sum_{a'} \pi(a'|S_{t+1})Q(S_{t+1}, a')) - Q(S_t, A_t). \tag{9}$$

3 Q(σ, λ): An on-line multi-step algorithm

The TD methods presented so far are one-step methods, which use only rewards and values from the next step $t+1$. These can be extended to use eligibility traces to incorporate data of multiple time steps.

An eligibility trace is a scalar numeric value for each state-action pair. Whenever a state-action pair is visited its eligibility is increased, if not, the eligibility fades away over time. State-action pairs visited often will have a higher eligibility than those visited less frequently and state-action pairs visited recently will have a higher eligibility than those visited long time ago.

The accumulating eligibility trace (Singh and Sutton (1996)) uses an update of the form

$$E_{t+1}(s,a) = \begin{cases} \gamma\lambda E_t(s,a) + 1, & \text{if } A_t = a, S_t = s \\ \gamma\lambda E_t(s,a), & \text{otherwise.} \end{cases} \tag{10}$$

Whenever taking action A_t in state S_t the eligibility of this pair is increased by 1 and for all states and actions decreased by a factor $\gamma\lambda$, where λ is the trace decay parameter.

Then all state-action pairs are updated according to their eligibility trace

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta_t\, E_t(s,a) \tag{11}$$

The corresponding algorithm using the one-step Sarsa TD error and an update using eligibility traces is called Sarsa(λ). Though it looks like a one-step algorithm, it is in fact a multi-step algorithm, because the current TD error is assigned back to all previously visited states and actions weighted by their eligibility.

For off-policy algorithms like Q-Learning different eligibility updates have been proposed. Watkin's Q(λ) uses the same updates as long as the greedy action is chosen by the behaviour policy, but sets the Q values to 0, whenever a non-greedy action is chosen assigning credit only to state-action pairs we would actually have visited if following the target policy π and not the behaviour policy μ. More generally the eligibility is weighted by the target policy's probability of the next action. The update rule is then

$$E_{t+1}(s,a) = \begin{cases} \gamma\lambda E_t(s,a)\pi(A_{t+1}|S_{t+1}) + 1, & \text{if } A_t = a, S_t = s \\ \gamma\lambda E_t(s,a)\pi(A_{t+1}|S_{t+1}), & \text{otherwise.} \end{cases} \tag{12}$$

Whenever an action occurs, which is unlikely in the target policy, the eligibility of all previous states is decreased sharply. If the target policy is the greedy policy, the eligibility will be set to 0 for the complete history.

In this paper we introduce a new kind of eligibility trace update to extend the Q(σ) algorithm to an on-line multi-step algorithm, which we will call Q(σ, λ). Recall that the one-step target of Q(σ) is a weighted average between the on-policy Sarsa and off-policy Expected Sarsa targets weighted by the factor σ:

$$\delta_t = R_{t+1} + \gamma(\sigma Q(S_{t+1}, A_{t+1}) + (1-\sigma)\sum_{a'} \pi(a'|S_{t+1})Q(S_{t+1}, a')) - Q(S_t, A_t) \tag{13}$$

In this paper we propose to weight the eligibility accordingly with the same factor σ. The eligibility is then a weighted average between the on-policy eligibility used in Sarsa(λ) and the off-policy eligibility used in $Q(\lambda)$. The eligibility trace is updated at each step by

$$E_{t+1}(s,a) = \begin{cases} \gamma\lambda E_t(s,a)(\sigma + (1-\sigma)\pi(A_{t+1}|S_{t+1})) + 1, & \text{if } A_t = a, S_t = s \\ \gamma\lambda E_t(s,a)(\sigma + (1-\sigma)\pi(A_{t+1}|S_{t+1})), & \text{otherwise.} \end{cases} \tag{14}$$

When $\sigma = 0$ the one-step target of Q(σ) is equal to the Sarsa one-step target and therefore the eligibility update reduces to the standard accumulate eligibility trace update. When $\sigma = 1$ the one-step target of Q(σ) is equal to the Expected Sarsa target and accordingly the eligibility is weighted by the target policy's probability of the current action. For intermediate values of σ the eligibility is weighted in the same way as the TD target. Asis et al. (2017) showed that a dynamic value of σ can outperform the classical TD control algorithms Q-Learning and Sarsa. By extending this algorithm to an on-line multi-step algorithm we can make use of the good initial performance of Sarsa(λ) combined with the good asymptotic performance of Q(λ). In comparison to the n-step Q(σ) algorithm (Asis et al. (2017)) the new Q(σ, λ) algorithm can learn on-line and is therefore likely to learn faster.

Pseudocode for tabular episodic Q(σ, λ) is given in Algorithm 2. This can be easily extended to continuing tasks and to function approximation using one eligibility per weight of the function approximator.

4 Double Q(σ) Algorithm

Double learning is another extension of the basic algorithms. It has been mostly studied with Q-Learning Hasselt (2010) and prevents the overestimation of action values when using Q-Learning in stochastic environments. The idea is to use decouple action selection (which action is the best one?) and action evaluation (what is the value of this action?). The implementation is simple, instead of using only one value function we will use two value functions Q_A and Q_B. Actions are sampled due to an ϵ-greedy policy with respect to $Q_A + Q_B$. Then at each step either Q_A or Q_B is updated, e.g. if Q_A is selected by

Algorithm 1 Q(σ, λ)

Initialize $Q(s,a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$

Repeat for each episode:

$E(s,a) \leftarrow 0 \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$

Initialize $S_0 \neq$ terminal

Choose A_0, e.g. ϵ-greedy from $Q(S_0, .)$

Loop for each step of episode:

Take action A_t, observe reward R_{t+1} and next state S_{t+1}

Choose next action A_{t+1}, e.g. ϵ-greedy from $Q(S_{t+1}, .)$

$\delta = R_{t+1} + \gamma \left(\sigma \, Q(S_{t+1}, A_{t+1}) + (1-\sigma) \sum_{a'} \pi(a'|S_{t+1}) \, Q(S_{t+1}, a')\right) - Q(S_t, A_t)$

$E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

$Q(s,a) \leftarrow Q(s,a) + \alpha \, \delta \, E(s,a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$

$E(s,a) \leftarrow \gamma \lambda E(s,a)(\sigma + (1-\sigma)\pi(A_{t+1}|S_{t+1})) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$

$A_t \leftarrow A_{t+1}, S_t \leftarrow S_{t+1}$

If S_t is terminal: Break

$$Q_A(S_t, A_t) \leftarrow Q_A(S_t, A_t) + \alpha(R_{t+1} + \gamma Q_B(\underset{a \in \mathcal{A}}{\operatorname{argmax}} \, Q_A(S_{t+1}, a)) - Q_A(S_t, A_t)) \tag{15}$$

$$Q_B(S_t, A_t) \leftarrow Q_B(S_t, A_t) + \alpha(R_{t+1} + \gamma Q_A(\underset{a \in \mathcal{A}}{\operatorname{argmax}} \, Q_B(S_{t+1}, a)) - Q_B(S_t, A_t)) \tag{16}$$

Double learning can also be used with Sarsa and Expected Sarsa as proposed by Michael Ganger and Hu (2016). Using double learning these algorithms can be more robust and perform better in stochastic environments. The decoupling of action selection and action evaluation is weaker than in Double Q-Learning because the next action A_{t+1} is selected according to an ϵ-greedy behavior policy using $Q_A + Q_B$ and evaluated either with Q_A or Q_B. For Expected Sarsa the policy used for the target in Equation 7 could be the ϵ-greedy behavior policy as proposed by Michael Ganger and Hu (2016), but it is probably better to use a policy according to Q_A (if updating Q_A), because then it can also be used off-policy with Double Q-Learning as a special case, if π is the greedy policy with respect to Q_A.

In this paper we propose the extension of double learning to Q(σ) - Double Q(σ) - to obtain a new algorithm with the good learning properties of double learning, which generalizes (Double) Q-Learning, (Double) Expected Sarsa and (Double) Sarsa. Of course double Q(σ) can also be used with eligibility traces.

Double Q(σ) has the following TD error when Q_A is selected,

$$\delta_t = R_{t+1} + \gamma \left(\sigma Q_B(S_{t+1}, A_{t+1}) + (1-\sigma) \sum_a \pi(a|S_{t+1}) Q_B(S_{t+1}, a) \right) - Q_A(S_t, A_t) \tag{17}$$

and

$$\delta_t = R_{t+1} + \gamma \left(\sigma Q_A(S_{t+1}, A_{t+1}) + (1-\sigma) \sum_a \pi(a|S_{t+1}) Q_A(S_{t+1}, a) \right) - Q_B(S_t, A_t) \tag{18}$$

if Q_B is selected. The target policy π is computed with respect to the value function which is updated, i.e. with respect to Q_A in Equation 17 and with respect to Q_B in Equation 18.

Pseudocode for Double Q(σ) is given in Algorithm 2.

Algorithm 2 Double Q(σ)

Initialize $Q_A(s,a)$ and $Q_B(s,a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$

Repeat for each episode:

Initialize $S_0 \neq$ terminal

Choose A_0, e.g. ϵ-greedy from $Q_A(S_0,.) + Q_B(S_0,.)$

Loop for each step of episode:

Take action A_t, observe reward R_{t+1} and next state S_{t+1}

Choose next action A_{t+1}, e.g. ϵ-greedy from $Q_A(S_{t+1},.) + Q_B(S_{t+1},.)$

Randomly update either Q_A:

$$\delta = R_{t+1} + \gamma(\sigma Q_B(S_{t+1}, A_{t+1}) + (1-\sigma)\sum_a \pi(a|S_{t+1}) Q_B(S_{t+1}, a)) - Q_A(S_t, A_t)$$

$$Q_A(S_t, A_t) \leftarrow Q_A(S_t, A_t) + \alpha\,\delta$$

or update Q_B:

$$\delta = R_{t+1} + \gamma(\sigma Q_A(S_{t+1}, A_{t+1}) + (1-\sigma)\sum_a \pi(a|S_{t+1}) Q_A(S_{t+1}, a)) - Q_B(S_t, A_t)$$

$$Q_B(S_t, A_t) \leftarrow Q_B(S_t, A_t) + \alpha\,\delta$$

$A_t \leftarrow A_{t+1}$, $S_t \leftarrow S_{t+1}$

If S_t is terminal: Break

5 Conclusions

This paper has presented two extensions to the Q(σ) algorithm, which unifies Q-Learning, Expected Sarsa and Sarsa. Q(σ, λ) extends the algorithm to an on-line multi-step algorithm using eligibility traces and Double Q(σ) extends the algorithm to double learning.

Dynamically varying the value of σ allows to combine the good initial performance of Sarsa with the good asymptotic performance of Expected Sarsa to obtain new state of the art results.

References

Asis, K. D., Hernandez-Garcia, J. F., Holland, G. Z. and Sutton, R. S. (2017). Multi-step reinforcement learning: A unifying algorithm, *CoRR* **abs/1703.01327**.
URL: *http://arxiv.org/abs/1703.01327*

Hasselt, H. V. (2010). Double q-learning, *in* J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel and A. Culotta (eds), *Advances in Neural Information Processing Systems 23*, Curran Associates, Inc., pp. 2613–2621.
URL: *http://papers.nips.cc/paper/3964-double-q-learning.pdf*

Michael Ganger, E. D. and Hu, W. (2016). Double sarsa and double expected sarsa with shallow and deep learning, *Journal of Data Analysis and Information Processing* **4**: 159–176.

Rummery, G. A. and Niranjan, M. (1994). On-line q-learning using connectionist systems, *Technical report*.

Singh, S. P. and Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces, *Machine Learning* **22**(1): 123–158.

Sutton, R. S. and Barto, A. G. (2017). Reinforcement learning : An introduction. Accessed: 2017-08-01.

Van Seijen, H., Van Hasselt, H., Whiteson, S. and Wiering, M. (2009). A theoretical and empirical analysis of expected sarsa, *Adaptive Dynamic Programming and Reinforcement Learning, 2009. ADPRL'09. IEEE Symposium on*, IEEE, pp. 177–184.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*, PhD thesis, King's College, Cambridge.