Marco Wiering
Department of Artificial Intelligence
University of Groningen
m.a.wiering@rug.nl

Reinforcement learning: from methods to applications

Reinforcement learning (RL) algorithms enable computer programs to learn from interacting with an environment. The goal is to learn the optimal policy that maximizes the long-term intake of a reward signal, where rewards are given to the agent for reaching particular environmental situations. The field of RL has developed a lot since the past decade. In this paper, Marco Wiering will first examine different decision making problems and RL algorithms. Then the combination of RL with different function approximators that can be used to solve large-scale problems will be described. After explaining several other ingredients of an RL system, a wide variety of different applications that can be fruitfully solved by RL systems will be covered.

Machine learning is a very hot research area within the larger field Artificial Intelligence (AI). The main idea of machine learning algorithms is to enable a computer program to optimize a particular performance metric using data or experiences to learn from. The field can be divided in three main areas: supervised learning, unsupervised learning and reinforcement learning. In supervised learning, a dataset with inputs and target outputs is given to a machine learning algorithm. The algorithm (with all of its tunable hyperparameters) then generates a model (or function) that can map inputs to their corresponding target outputs. An important issue in supervised learning is to generate a trained model that also performs well for new, unseen, inputs, which means the model should generalize over the whole input space.

In unsupervised learning, an algorithm receives input data without target outputs. The algorithm can use the input patterns to compute clusters of inputs that seem to belong to each other, or it can derive particular features or frequently occurring sub-patterns from the data.

Whereas the aforementioned algorithms work with a dataset of examples to generate a model, reinforcement learning (RL) algorithms train an agent, which is a software program that can select and execute actions based on its inputs. Furthermore, such RL agents interact with an environment through which it receives experiences. Therefore, RL methods do not need a dataset, but are connected to a simulator or the real world. The experiences obtained by an RL agent are generally its observations of the environment, the actions which it executes, and rewards which it receives from selecting actions in particular environmental situations. The goal of the RL agent is to maximize the sum of rewards that it obtains over time.

In particular problems, such as learning to optimally play a game, the reward is given at the end of the game, and is simply 1 for winning, -1 for losing, and 0 for a draw. Even though the agent only receives such *sparse* rewards, it can still learn to play the game very well in many cases. Currently, the most impressive demonstration of the power of RL is Google DeepMind's AlphaGo Zero system [47] that learned to play the complex game of Go extremely well by only playing games against itself. AlphaGo Zero was able to beat its predecessor AlphaGo [45], which first learned from games played by human players and won a match against the human Grandmaster Lee Sedol in 2016 with $4{-}1$.

In this paper, first different kinds of sequential decision making problems for RL are examined. Then we will study several RL algorithms that learn from the experiences obtained by the agent in order to maximize its performance over time. We then examine the issue of exploration, which enables an agent to explore the results of selecting different actions, and which is necessary to learn to perform optimally. To deal with very large state spaces, function approximation techniques are necessary and the most often used techniques will be explained. Then some other topics in RL will be covered, which increase the speed or applicability of RL systems. After having explained the ingredients of RL systems, a number of different applications will be finally described for which RL can be used effectively.

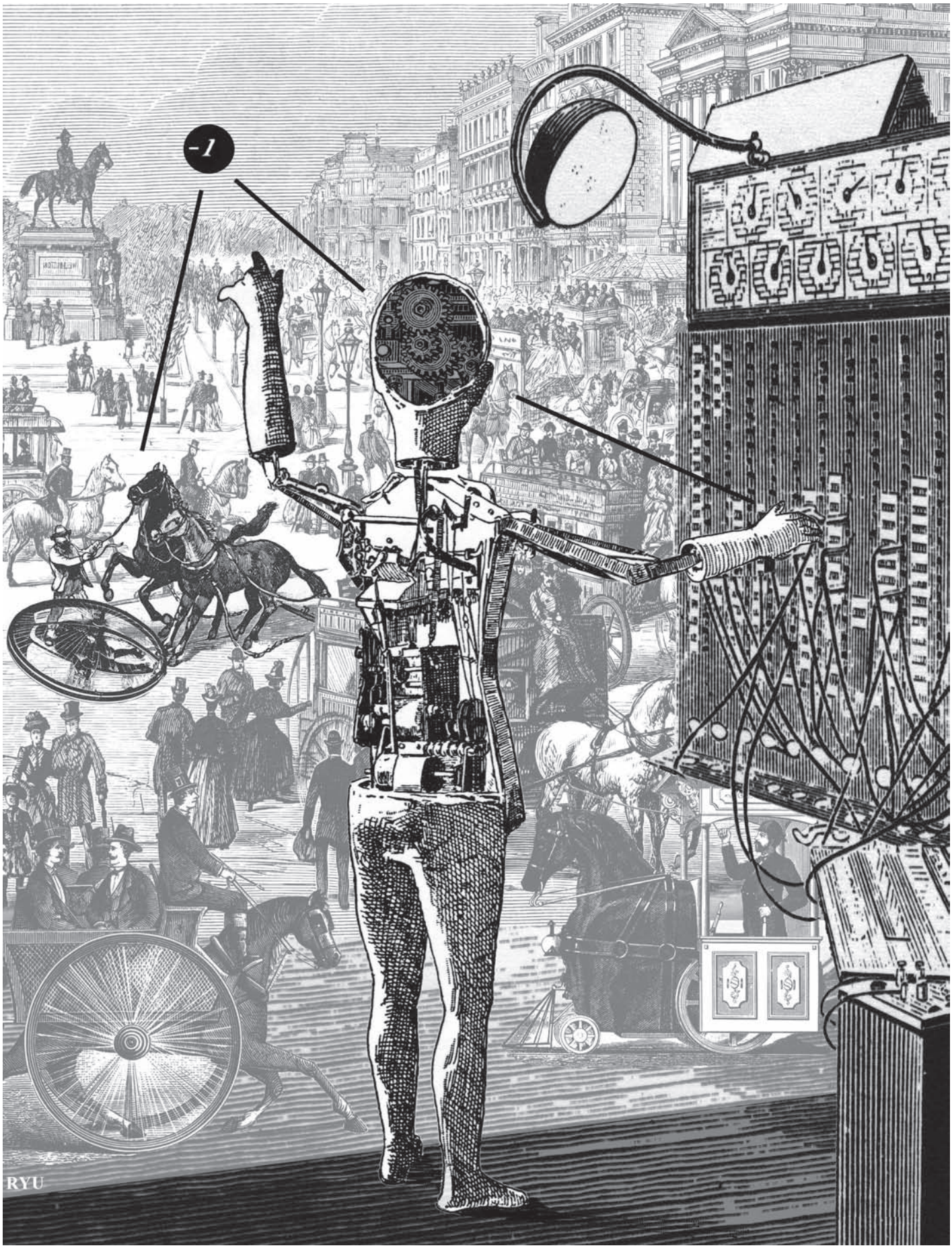

Illustration: Ryu Tajiri

Sequential decision making

Reinforcement learning algorithms enable an agent to optimize its behavior by learning from the interaction with the environment. At each time step t, the agent perceives the state s_t, and uses the state information to select and execute action a_t. Based on this executed action, the agent transits to a new state s_{t+1} and receives a scalar reward r_t. The interaction history of the agent with its environment is $h_t = s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_t$. The interaction between the agent and its environment is illustrated in Figure 1. For some problems, the agent cannot perceive the full state information, and only receives a partial observation o_t instead of s_t. An important property of a sequential decision making problem is whether it has the Markov property, which holds if the current state s_t and action a_t contain sufficient information to predict the next state and the expected received reward: $P(s_{t+1}, r_t | s_t, a_t) = P(s_{t+1}, r_t | h_t, a_t)$. This means that instead of needing to use the whole interaction history h_t, only the current state s_t can be used to select the optimal action. This simplifies matters a lot, since the history could be a very long sequence of previous states and previously executed actions.

There are different kinds of sequential decision making problems. The simplest one is a finite Markov Decision Process (MDP) [58], which has the Markov property and is defined by:

- A set of states S, where $s_t \in S$ denotes the state at time t.
- A set of actions A, where $a_t \in A$ denotes the action selected at time t.
- A transition function $T(s,a,s')$, which specifies the probability of moving to state s' after selecting action a in state s.
- A reward function $R(s,a)$, which sends a reward signal to the agent for executing action a in state s. r_t denotes the reward obtained at time step t. For simplicity we denote the average reward obtained by executing action a in state s also by $R(s,a)$.
- A discount factor γ that makes rewards received further in the future less important than immediate rewards, where $0 \leq \gamma \leq 1$.

The goal in an MDP is to learn a policy, $\pi(s) \rightarrow a$, mapping states to actions in such a way that the agent receives the highest cumulative discounted reward sum in its future. This cumulative discounted reward sum is called the *return* and is defined as:

$$R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i} = r_t + \gamma R_{t+1}.$$

Note that an infinite horizon (future) is used here. By having a discount factor $\gamma < 1$, it is guaranteed that the sum is bounded. Instead of directly optimizing the action-selection policy, value-function based RL algorithms use state-value functions or state-action value functions. The state-action value function $Q^\pi(s,a)$ denotes the expected sum of discounted rewards obtained when the agent selects action a in state s and follows policy π afterwards:

$$Q^\pi(s,a) = E\left(\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right).$$

Where E denotes the expectancy operator. For small-sized finite MDPs, the Q-function can be stored in a lookup table of size $|S||A|$. The goal is then to learn the Q-function Q^* of the optimal policy, for which the following holds:

$$Q^*(s,a) \geq Q^\pi(s,a) \quad \forall s,a,\pi.$$

When the optimal Q-function is known (either computed or learned), the agent can always select the optimal action with the highest Q-value in some state s:

$$\pi^*(s) = \arg\max_a Q^*(s,a).$$

When the MDP is known a priori, it is possible to compute the optimal Q-function with dynamic programming techniques such as value interaction [7]. Bellman noted that the following must hold for the optimal Q-function:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_b Q^*(s',b).$$

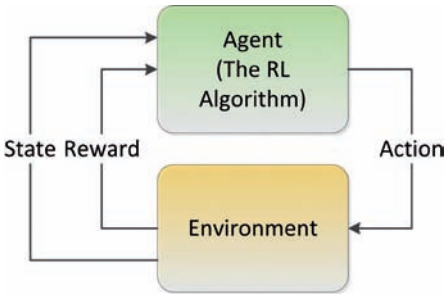

Figure 1 The interaction process between an RL agent and its environment. The agent receives as perception of the environment the state information and uses this to select an action. After this, the agent receives a reward and goes to a next state.

This equation is known as Bellman's optimality equation and uses an efficient recursive formulation. The right-hand side of this equation is basically using a one-step lookahead in the future and could be used to update the Q-value from the left-hand side. The dynamic programming algorithm called value iteration makes use of this to efficiently compute the optimal policy for a known MDP. The algorithm starts with an arbitrary policy π and an arbitrary value function V and repeats the following steps for all $s \in S$. First, the Q-values for all actions in state s are computed:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') V(s').$$

Then, the new value function is calculated:

$$V(s) = \max_a Q(s,a).$$

Then, $\pi(s)$ is changed so that the action a which maximizes the Q-function in a state s will be chosen:

$$\pi(s) = \arg\max_a Q(s,a).$$

The algorithm halts when the largest difference between two successive value functions is smaller than ϵ, some positive parameter with a value close to 0. During each iteration, an additional lookahead step in the future is performed. Finally, with a discount factor $\gamma < 1$, the far future is discounted so much, that more iterations do not change the value function anymore.

Although in a lot of research about reinforcement learning, the MDP framework is used, there are also more complex sequential decision making problems. One of these is the Partially Observable Markov Decision Process (POMDP) framework [28]. In a POMDP, the Markov property does not hold, because the agent only receives an observation o_t in each state using an observation function $o_t = O(s_t)$. One problem of POMDPs is that the same observation may be received when the agent is in different states, which is called *perceptual aliasing*. Since the observation does not fully describe the current state, the Markov property does not hold. For this reason, it is necessary to make use of the history h_t instead of only the current observation o_t in order to enable the agent to choose optimal actions.

There exist different algorithms to compute solutions to a priori known POMDPs. The most commonly used approach is to

use the history h_t to create a belief state $b(s_t)$, which is a probability distribution over the entire state space [13]. The probability that the agent is in each state given the history can be efficiently computed using recursive formulas. Then dynamic programming can be used with the belief states in order to compute the optimal policy. The complexity of solving POMDPs is however much larger than that of solving MDPs. Whereas MDPs can be solved in polynomial time, solving POMDPs soon becomes intractable with more states, actions and observations.

Other sequential decision making problems exist when multiple agents learn at the same time. In this case, the Markov property also does not hold anymore, because the actions of other agents change over time and influence the goodness of selecting a particular action in some state for a specific agent. Furthermore, in multi-agent sequential decision making problems, all agents may have a cooperative reward function, but there may also be competitive individual reward functions.

Reinforcement learning algorithms

For most real-world applications a prior model of an MDP (or POMDP) is not known beforehand. Furthermore, the state-action space may be high-dimensional or continuous, so that dynamic programming cannot be effectively used. In that case, RL algorithms can be used to learn the optimal policy by interacting with the environment. The main idea of an RL algorithm is to use each experience of the agent in the form (s_t, a_t, r_t, s_{t+1}) in a way to improve the policy or Q-function. The Q-function can be stored in a lookup table for small state-action spaces, but can also be approximated with function approximators. The most often used algorithm is Q-learning [61,62]. In online Q-learning the Q-value of a state-action pair is updated using the experience by:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha(r_t + \gamma \max_b Q(s_{t+1},b) - Q(s_t,a_t)).$$

Where α is the learning rate that determines how much the agent learns from the current experience. In case the last state s_{t+1} is a terminal state, the update rule becomes:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha(r_t - Q(s_t,a_t)).$$

Because most environments are stochastic, it is not always a good idea to use a large learning rate to speed up the learning process. This is because learning on one experience also means forgetting about previous experiences. With a finite number of states and actions, Q-learning converges to the optimal policy when all state-action pairs are visited for an infinite number of times and the learning rate is properly annealed [22]. Note that this means all actions should be tried out infinitely often in each state. Therefore always selecting the action with the highest current Q-value in a state is not good for optimizing the policy. There is also the need to *explore* alternative actions, which currently do not seem optimal, but which could lead to better future states and higher rewards. Exploration in RL will be described in the following section.

Q-learning is called an *off-policy* RL algorithm. Due to the necessity to use an exploration strategy, the behavioral policy that selects actions, is not the same as the greedy policy that is the result of the learning dynamics. Because Q-learning uses the $\max$ operator in its update rule, Q-learning is able to learn the optimal Q-function (and therefore policy) even if always random actions are selected during the learning process. This has several advantages, since it usually allows higher exploration rates than *on-policy* RL algorithms. A well known on-policy RL algorithm is SARSA [39,50] which stands for state-action-reward-state-action. SARSA updates the Q-value of action a_t in state s_t in the following way:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha(r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)).$$

As can be seen from this equation, instead of the best thought action in state s_{t+1}, the actual selected action in the next state is used to update the Q-value. Because SARSA is an on-policy RL algorithm, it is not able to learn an optimal policy as long as there is exploration. Therefore, SARSA only converges to the optimal policy and Q-function if the same conditions hold as for Q-learning and if the exploration policy becomes greedy in the limit of infinite exploration (GLIE) [48].

The difference between the learned behavior when off-policy Q-learning or on-policy SARSA is used can be best illustrated through an example [51]. Given the grid shown in Figure 2, the agent can choose to move North, West, South and East. If an action is not possible (the agent goes outside the grid), the agent remains in the same state. For every action a negative reward of -1 is given except when the agent goes to a state where it falls from the cliff (in yellow), where the agent dies and receives a reward of -100 after which a new epoch is started. When the agent is in the goal state, an epoch terminates as well and the agent does not receive negative rewards anymore. When an epoch ends, the agent starts again in

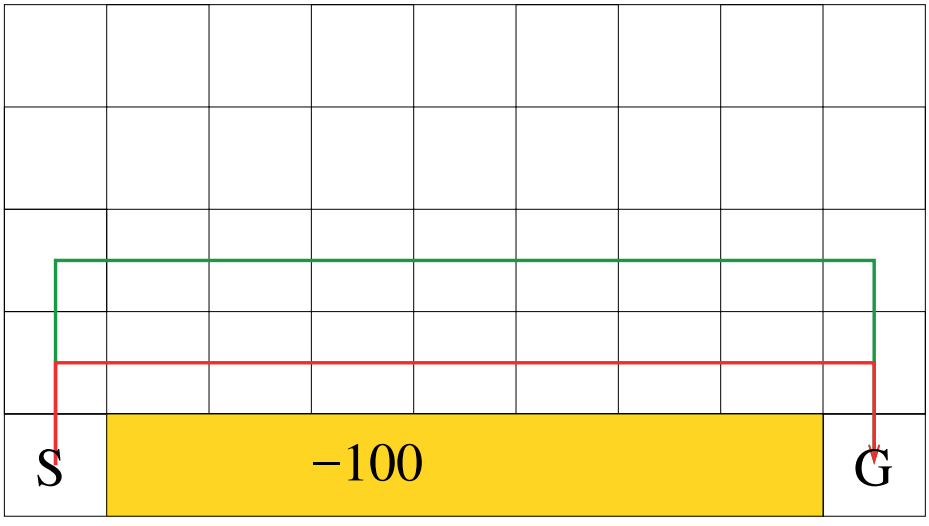

Figure 2 The cliff environment illustrating the difference in learning dynamics between Q-learning and SARSA. The goal is to find the shortest path from the start state S to the goal state G, however a large penalty is given if the agent falls off the cliff. Q-learning will learn the optimal red path, whereas SARSA will converge to a slightly longer path (in green). On the other hand, while learning, the Q-learning agent will fall off the cliff much more often due to exploration actions than the SARSA agent, which takes the possible exploration actions into account when learning the policy.

the start state and this is repeated for a large number of times.

The problem is deterministic, but the agent sometimes chooses an exploration action with some probability. Q-learning will converge to the optimal path shown in red in the figure. SARSA will converge to the green path. It can be seen that the final path of SARSA is longer. This is because the SARSA agent takes into account its exploration actions which often bring it to fall off the cliff when learning. Therefore, the SARSA agent will learn to avoid coming close to the cliff. The result is that the SARSA agent obtains a higher average reward while learning, but after the learning process has converged to a stable policy, the agent receives a slightly lower sum of rewards. Although this is a very simple problem, it clearly shows the differences between off-policy and on-policy RL algorithms.

There are a number of other reinforcement learning algorithms. QV-learning [66] uses two value functions: the state value function $V(s)$ and the state-action or Q-function $Q(s,a)$. The Q-function is trained based on the value function and the value function is trained by using the immediate reward r_t and the value of the next state.

In actor-critic algorithms, the critic uses temporal difference learning [49] to train the value function $V(s)$, and the actor is used to select actions. In the actor-critic framework, the actor is trained using the critic's evaluations. A main advantage of actor-critic algorithms is that it becomes easier to deal with continuous action spaces. If a Q-function $Q(s,a)$ is used in a critic-only system, it is complex to select the continuous action that maximizes the Q-value. With actor-critic approaches, the actor can immediately output the action that should be selected.

A related class of algorithms are policy-gradient algorithms. In their basic form, these algorithms only use an actor with a policy and no critic. One of the earliest approaches, REINFORCE [72], trains the policy using the full (Monte-Carlo) return. If this return is high, then the policy is adjusted so that the taken actions will be strengthened (reinforced). Otherwise, if bad things happen, then the policy is changed to circumvent selecting the same actions.

Another class of algorithms is called evolutionary reinforcement learning [64]. Evolutionary algorithms can be used for many different kinds of problems such as combinatorial optimization problems. They maintain a population of individuals that encode a policy and each individual is tested to obtain its fitness value for the task at hand. Then individuals are selected based on their fitness value and the best individuals are allowed to reproduce most often. New offspring can then be created by mixing the information of the two selected parent individuals using crossover operators and some randomness is added by using a mutation operator. These evolutionary RL methods have been successfully applied to different problems such as playing Othello [33] and were also successfully used to learn policies for Atari games [19]. Value-function based RL and evolutionary RL have also been combined by a number of researchers, see [15] for a review of such combinations.

RL algorithms sometimes need very many experiences to learn a good policy. One of the reasons is that algorithms such as Q-learning and SARSA only update a single Q-value given one new experience. There exist several methods to improve on this. Eligibility traces [49] enable multiple states to be eligible to change their values based on a new experience. Experience replay [27] is a technique in which many experiences are stored in a replay memory and the replay memory is then used to make multiple updates. Experience replay is nowadays often used in many deep RL algorithms [30].

Finally, model-based RL method compute an approximation to the transition and reward functions based on the obtained experiences. When these are modelled, dynamic programming techniques can be used to efficiently compute the Q-function [32, 70]. Although model-based RL techniques are very efficient for finite MDPs with not too many states and actions, it is very complex to use them for high-dimensional and continuous state-action spaces.

Exploration

As mentioned before, an agent needs to make exploration actions in order to be able to learn an optimal policy. However, to get most rewards while interacting with the environment, the agent should also exploit its current knowledge in the form of the Q-values it learned. This leads to the exploration-exploitation dilemma [53]. This dilemma has often been studied using multi-armed bandits (MABs), which are a special kind of MDP with only a single state. In a MAB, the agent can select among k actions, each having its own stochastic reward function $R(a)$. By trying an arm a_t at time step t, the agent receives a reward r_t, and uses this to update its estimate $Q(a_t)$ of the average reward defined by function $R(\cdot)$. When the agent would try out all actions an infinite number of times, then the Q-function $Q(a)$ converges to the average reward $R(a)$ due to the law of large numbers. After this, the agent can simply select the action with the highest Q-value. However, if the agent tries out all actions many times, it also selects suboptimal actions many times, and this is of course not desired.

The MAB-problem has often been used to create proofs for exploration algorithms, as it is simpler than solving a full MDP. Such proofs can for example give guarantees about the maximal amount of experiences needed to learn the optimal action most of the times. Furthermore, exploration algorithms designed for the MAB-problem can also be used as exploration strategy in an MDP. We will now describe a number of often used exploration strategies for solving MDPs [55].

ϵ-greedy exploration is one of the most used exploration strategies. It uses $0 \leq \epsilon \leq 1$ as parameter of exploration to decide which action to perform using $Q(s_t,a)$. The agent chooses the action with the highest Q-value in the current state with probability $1-\epsilon$, and a random action otherwise. A larger value for ϵ means more exploration actions are selected by the agent. Although ϵ-greedy is very simple, it is sometimes hard to beat.

One drawback of ϵ-greedy exploration is that the exploration action is selected uniform randomly from the set of possible actions. Therefore, it is as likely to choose the worst action as it is to choose the second-best action if an exploration action is selected. That is why Boltzmann or softmax exploration [51] uses a Boltzmann distribution to assign a probability $\pi(s_t,a)$ to the actions in order to choose actions with higher Q-values with higher probability:

$$\pi(s_t,a) = \frac{e^{Q(s_t,a)/T}}{\sum_{b=1}^{m} e^{Q(s_t,b)/T}}.$$

$\pi(s_t,a)$ is then the probability with which the

agent selects action a in state s_t. $T \geq 0$ is the temperature parameter used in the Boltzmann distribution. When $T = 0$ the agent does not explore at all, and when $T \rightarrow \infty$ the agent always selects random actions.

Another simple way to allow the agent to explore, is to initialize all Q-values to large values, which is called being optimistic in the face of uncertainty. This makes the agent explore a lot in the beginning during which the values of explored actions will drop. After some time, the initial Q-values have washed out, but the agent has learned which state-action pairs lead to more reward intake than others. A problem of this approach is that the Q-values can drop quite fast when the learning rate is large. Another problem is that this method is difficult to combine with function approximation techniques, which will be discussed in the next section.

The above techniques are all undirected exploration strategies: they only use the Q-values in order to select (exploration) actions. Directed exploration techniques use other information as well. For example, count-based methods [70] keep track of the number of times each action has been selected in each state. This allows the agent to go to areas of the state space, in which it has not (often) been before. Another directed technique is to learn how large the Q-value updates and errors were for particular state-action values. The idea is that large errors correspond to more uncertainty in the actual values and therefore state-action pairs with large errors should be explored more often.

A difficulty of the error-based exploration strategy is that in some states the environment may be very stochastic. In such problems the error will stay high, but these state-action pairs may not be important to visit over and over for learning the optimal policy. Therefore Schmidhuber [41] proposed the idea of curiosity and novelty-based exploration. Here, the agent is attracted to new parts of the state space, but at the same time the agent records if it can really learn new knowledge in these regions of the environment. Think for example about a television. Although black-white noise images are always different, they cannot be predicted and therefore watching these will not be useful when considered by this framework. Curiosity-driven exploration allows the agent to learn to increase its knowledge about the environment, and can be combined with a reward function for solving a task.

Function approximation

In the previous sections, we examined tabular representations for storing the state-action value function. Although these allow to optimally represent the true value function, they cannot be used with very large state spaces or with continuous state spaces. Often decision making problems involve multiple variables representing the state. When this number of variables increases the state spaces explodes, which is called the *curse of dimensionality*. For example, with 20 binary variables representing the state, the number of states is already 2^{20}, or around 1 million. Although current computing power is large enough to cope with 1 million states, things get worse when representing the state of for example all possible chess positions. There are in total around 10^{30} different chess positions, so storing them in a lookup table would be infeasible. To handle this, function approximation techniques can be used to learn to approximate the state-action value function. Next to the decrease in necessary storage space, another advantage of function approximators is that they generalize over the entire state space. If lookup tables are used, states that have not been visited do not have any updated Q-values yet and therefore selecting an action can only occur randomly. With function approximation techniques, similar states will get assigned similar action values and therefore not all states have to be visited in order to select meaningful actions.

In supervised learning, there are many algorithms that can learn mappings from input vectors to real-valued outputs. Basically, all such methods can be used also in reinforcement learning, but in RL research has focused mostly on a specific subset of such methods. The most commonly used function approximators used in RL are: linear function approximation techniques such as tile-coding and radial-basis function networks, multi-layer perceptrons and deep neural networks.

Linear function approximation techniques use a set of basis functions to map the state s_t to an internal representation $\phi_1(s_t), \ldots, \phi_k(s_t)$. To approximate a Q-function these basis function activations are linearly combined:

$$Q(s,a|\theta) = \sum_{i=1}^{k} \phi_i(s)\theta_{i,a}.$$

The advantages of these linear function approximation techniques are that the algorithms are fast, have good convergence properties and are easy to implement. A disadvantage is that the performance of these methods depends heavily on the overall design of the basis functions. One way of designing the basis functions is to use tile coding or CMACS [2, 50]. In tile coding a number of tiles are used with cells inside. Given a state, one cell in each tile is activated, and the activated cells form the new representation of the state. The use of multiple overlapping tilings helps to generalize better for continuous state spaces, as multiple cells which are activated in slightly different input regions can be updated at the same time. A problem of tile coding is that it is difficult to design the tiles and cells for high-dimensional state spaces. In this case, tiles need to combine many different projections of a few state variables, because otherwise there are far too many cells in each tile.

Another linear function approximation technique is the use of radial-basis functions (RBFs). Radial basis functions resemble Gaussian functions without the necessity to be a true probabilistic function that integrates to one. The idea is to place a number of RBFs at specific places in the input space with a specific width, and then given an input state, several of these RBFs are activated based on their distance to the input. The closest RBFs get the highest activations and most of the other RBFs receive an activation of zero. Then the activated RBFs form the new representation of the state. Similarly to the use of tile coding, the use of RBFs has the problem that it is very difficult to fill high dimensional state spaces with these local basis functions. Therefore also this method is best used with not very many state variables.

A more powerful technique to approximate the state-action value function is to use multi-layer perceptrons (MLPs) [38, 63]. Multi-layer perceptrons map an input vector to an output vector using multiple layers of neurons. The neurons are connected with weighted connections, and these weights can be adapted by a learning algorithm to learn the right mapping of input vectors to desired output vectors. One of the first times MLPs were com-

bined with RL, was in Tesauro's backgammon learning program [52]. This program, called TD-Gammon, was able to learn to play backgammon purely by playing games against itself. After around 1.5 million training games, the program was able to attain the level of the strongest human players. MLPs are very powerful in their way to represent non-linear functions, but often need more time and experiences to train and they have much worse convergence guarantees than linear function approximation techniques. In some cases, MLPs combined with Q-learning can even diverge and lead to continuously increasing weight and output values. Still, in many applications MLPs have been fruitfully combined with RL [8, 27, 57].

Inspired by the recent successes in deep learning [25, 42], convolutional neural networks have also been introduced in reinforcement learning, a field that is called *deep reinforcement learning*. In [30], the authors used a convolutional neural network combined with Q-learning to learn to play different Atari games using pixel information as input. The difficulty of using pixels as input is that the input space is incredibly large. However, by using convolutional neural networks, the authors showed that it was possible to play different Atari games at a very good level using only pixel input. In a later paper [31], the authors used the same methodology to learn to play 49 different Atari games at a level comparable to a professional human games tester. Deep RL has also been used with a number of other algorithms such as Monte-Carlo Tree Search [24] to learn to play Go at a level better than any human player [47]. Almost the same system has also been used to learn to play Chess and Shogi at a level much better than any human or previous computer program by letting the system learn from playing games against itself [46]. Nowadays, the field of deep RL attracts a lot of interest from the RL community, and different algorithms have been introduced to make it more efficient [20]. Figure 3 shows how a Deep RL system interacts with the classical Arcade game Breakout to learn to optimize its score.

For solving POMDPs with RL systems, recurrent neural networks can be effectively used. One of the best known and performing recurrent neural networks is the long short term memory (LSTM) network

Figure 3 A Deep RL system learning to play the game Breakout from pixel information as input

[21]. An LSTM can learn to overcome long-time relationships between previous inputs and future predictions and has been successfully used for many time-series applications. Bakker [4] was the first who used the LSTM for solving a POMDP and showed that the LSTM was able to learn to overcome long time lags.

Other topics in RL

We have examined how an agent can use an RL algorithm to learn to optimize its behavior by interacting with an environment. In most research in RL, an agent learns a single state-action value function or policy and uses that to select actions. In the previous decades, a number of new topics in RL have emerged, which are described in detail in [67]. Here, we will have a look at several topics: hierarchical RL, transfer learning, multi-agent RL, and multi-objective RL.

In hierarchical RL a divide and conquer strategy is used to split the overall task in a number of subtasks [5]. In one of the earliest hierarchical RL systems, HQ-learning [69], a higher-level policy is used to learn to select subgoals which the lower-level policy has to attain. This allowed HQ-learning to solve a number of difficult tasks modelled as POMDPs. The options framework [36] introduced the notion of options, which are macro-actions that select multiple actions before finishing. Such options make it easier to learn to solve problems where many actions are needed to arrive at a goal state, since shorter sequences of options can arrive in the goal state. This makes it easier to quicker find the goal and solve the temporal credit assignment problem, which relates to how important a previously executed action was for attaining a goal. MAXQ-learning [14] is another hierarchical RL system in which the overall task is broken up in smaller tasks until at the bottom of the tree primitive actions can be chosen. In general, hierarchical RL systems help to make learning faster when compared to flat learning algorithms.

Humans can learn to solve new tasks faster by using their previous experience with related tasks. This is the field of transfer learning that aims to reuse solutions to previously solved tasks to solve new sequential decision making problems. Transfer learning has been used in different ways such as instance transfer in which experiences are reused, representation transfer in which an abstraction process is used to allow transferring parts of the representation, and parametric transfer to use previous Q-functions for learning to solve new problems [26].

Multi-agent RL is used when multiple agents can learn and interact in the same environment [11]. The reward function can be shared among the agents, leading to cooperative multi-agent systems, or can be individual functions for each agent, possibly leading to competitive systems. One of the oldest multi-agent RL algorithms was a system that used Q-learning on multiple nodes to route packages over a network [9]. The system learned that under busy situations, the shortest route between different parts of the network will become saturated, leading to long waiting times. Therefore the RL system learned to select alternative longer routes if necessary. Multi-agent RL has also been used to learn to find solutions to game theoretical problems such as matrix games. In a matrix game, each agent selects an action and based on all selected actions, each agent receives its own reward. Many matrix games have a competitive nature, and therefore the aim is to learn to converge to a Nash equilibrium, in which no agent is better off when it deviates from the joint action strategy [35].

Although most RL research has focused on scalar reward functions, there also exist RL systems that learn to optimize reward vectors where multiple objectives are assigned rewards at the same time. This is the field of multi-objective RL and its aim is not so much to learn a single policy, but to learn all polices that are not dominated [16, 37, 68, 71]. Policies are non-dominated if they do not obtain a lower cumulative reward on all objectives than another policy. In general it is much harder to learn the set

of non-dominated policies than a single optimal policy, and that is why multi-objective RL has not been used yet for solving very complex problems.

Applications

There are many applications to which reinforcement learning algorithms have been efficiently applied. We will describe a wide range of different applications, starting with game playing, multi-agent problems, and robotics and then discuss applications for medicine and health, electric power grids, and end with personal education systems.

Games provide researchers with a large number of interesting problems having different complexities, but still allow for controlled and repeatable settings. Therefore, they have always played an important part of AI research [73]. The oldest self-learning program that learned to play a game is Samuel's checkers playing program [40]. It combined several machine learning methods and reached a decent amateur level in playing checkers. A very successful attempt to using reinforcement learning to play games is TD-Gammon [52], that learned to play the game of Backgammon at human expert level using temporal difference learning [49] and multi-layer perceptrons. Although in the nineties, learning from 1.5 million games took multiple months, with the current computing power this can be done within several hours. Another board-game that has received a lot of attention from RL researchers is Othello [10, 29, 57, 56]. In [56], structured multi-layer perceptrons were used in which not all board-fields were connected to hidden units, but specific lines or regions were connected to them. This led to much better results and a faster learning process. As mentioned before, AlphaZero obtained an excellent level of playing Chess, but Chess has also been used a long time before to let RL systems to learn to play the game, although the final performance of these RL systems was much worse [6, 54].

Next to using RL for learning to play board-games, RL has also been used for many other types of games. In [8], the authors used Q-learning with a multi-layer perceptron to learn to play the game Ms. Pac-Man. The novel thing was that the system only used seven input units, which extracted higher-level information from the game state. This allowed the system to learn to perform well with only $10{,}000$ training games. Deep RL has also been used for learning to optimize the behavior of Ms. Pac-Man, but using only pixel-information, this took much more training games and led to worse results [31]. Some researchers have decomposed the overall Q-function into a set of single objective reward functions, which are then combined to solve the overall problem. This technique has been successfully used to obtain the highest score with an RL system for the game Ms. Pac-Man in [59].

Many other games such as Starcraft [43], Tron [23], and others have also been used to develop different RL systems that can learn to play them well.

Although games provide researchers with an interesting test problem for their algorithm, the societal relevance is a bit lower than for other problems. In the field of multi-agent reinforcement learning (MARL), different systems have been constructed to learn to optimize the behavior of multiple agents. The network routing problem mentioned earlier is one example. Another successful example was the use of MARL for elevator control [12]. In this system, four elevators have to collaborate to minimize the total waiting time of people wanting to take the elevator from one floor to another. The RL system was able to learn to control the elevators such that the overall waiting time was shorter than with many commonly used techniques. Another MARL problem that has received a lot of attention is traffic light control. Many intersections in cities have traffic lights, but the commonly used controllers can in many cases be improved using RL. In [65], one of the first RL systems is presented that learned to control traffic lights on many intersections. This resulted in much lower waiting times compared to non-adaptive controllers. RL has also been used in a realistic traffic network modelled after the city Tehran where multiple traffic disruptions can take place [3]. Figure 4 shows the used traffic light simulator in [65].

Although RL techniques usually need many interactions with an environment to optimize an agent or controller, RL has also been effectively used for robot control. In [1], first a model was learned from the interaction between an unmanned aerial vehicle (helicopter) and the sky, and then RL was used to learn very complex behaviors such as looping etc. In [60], RL was combined with motor primitives, which take care of particular action sequences, to train a robot to play table tennis. In a recent paper [18], a number of robots was used at the same time to train robots to open a door. There is a growing amount of research using RL for different robot control problems, and of course we cannot cover them all in this short survey.

Therapy planning for patients is another sequential decision making problem where for example different therapies can be giv-

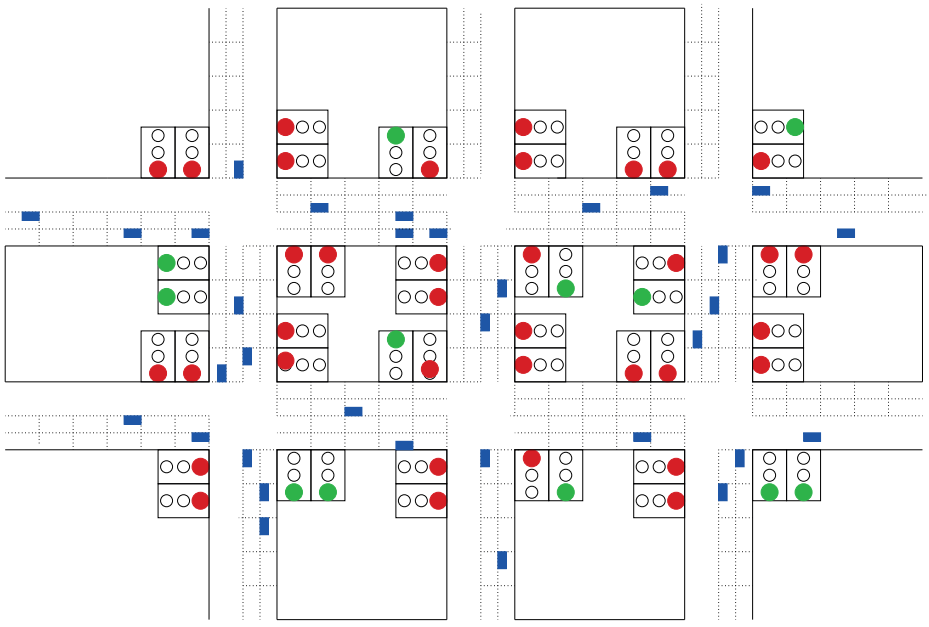

Figure 4 The traffic light simulator consisting of 6 intersections. There are traffic lights for going straight ahead/right or for going left (no collissions).

en to a patient and the goal is to make the patient healthy as soon as possible. Therefore this is also a very good problem for RL, but some things have to be adapted, for example exploration is more complex in this scenario. In [44], an RL system is developed to plan therapies for patients suffering from schizophrenia. In [34], a deep RL system is used for medication dosing. It should be clear that there are many more possible applications of RL to health and medicine.

RL systems have also been used to solve electric power system decision and control problems [17] and make it easier to cope with additional variable power sources such as sun and wind energy. In the finance sector, RL systems can be used to learn from a historical huge amount of data to create automatic trading bots. Also in advertising RL systems have been used to estimate on which advertisement a specific user is most likely to click on. Chatbots have recently been constructed using deep reinforcement learning and because these systems can become better with more interactions and feedback, the end of this is not in sight. Finally, there is some recent work on intelligent tutoring systems that use RL to educate people. These systems can offer much more personalized education than currently used in the classroom.

To conclude, RL systems have been applied to a wide range of different problems. RL systems can continuously learn to improve themselves and do not need a dataset or humans to provide labels to the system. Therefore, reinforcement learning is a very promising direction for Artificial Intelligence to evolve further and to help human kind to cope with many different challenges in life.

References

1 P. Abbeel, A. Coates, M. Quigley and A.Y. Ng, An application of reinforcement learning to aerobatic helicopter flight, in B. Schölkopf, J.C. Platt and T. Hoffman, eds., *Advances in Neural Information Processing Systems* 19, MIT Press, 2007, pp. 1–8.

2 J.S. Albus, A new approach to manipulator control: The cerebellar model articulation controller (CMAC), *Journal of Dynamic Systems, Measurement and Control* 97 (1975), 220–227.

3 M. Aslani, M. Mesgari and M. Wiering, Adaptive traffic signal control with actor-critic methods in a real-world traffic network with different traffic disruption events, *Transportation Research Part C: Emerging Technologies* 85 (2017), 732–751.

4 B. Bakker, Reinforcement learning with long short-term memory, in T.G. Dietterich, S. Becker and Z. Ghahramani, eds., *Advances in Neural Information Processing Systems* 14, MIT Press, 2002, pp. 1475–1482.

5 A. Barto and S. Mahadevan, Recent advances in hierarchical reinforcement learning, *Discrete Event Dynamic Systems* 13 (2003), 341–379.

6 J. Baxter, A. Tridgell and L. Weaver, Learning to play chess using temporal differences, *Machine Learning* 40(3) (2000), 243–263.

7 R. Bellman, A markovian decision process, *Indiana Univ. Math. J.* 6(4) (1957), 679–684.

8 L. Bom, R. Henken and M. Wiering, Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs, in *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, 2013, pp. 156–163.

9 J.A. Boyan and M.L. Littman, Packet routing in dynamically changing networks: A reinforcement learning approach, in *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS'93, Morgan Kaufmann Publishers, 1993, pp. 671–678.

10 M. Buro, The evolution of strong Othello programs, *Entertainment Computing*, Springer, 2003, pp. 81–88.

11 L. Busoniu, R. Babuska and B.D. Schutter, Multi-agent reinforcement learning: A survey, *2006 9th International Conference on Control, Automation, Robotics and Vision*. 2006.

12 R. Crites and A. Barto, Improving elevator performance using reinforcement learning, in D. Touretzky, M. Mozer and M. Hasselmo, eds., *Advances in Neural Information Processing Systems* 8, MIT Press, 1996, pp. 1017–1023.

13 B. D'Ambrosio, POMDP learning using qualitative belief spaces, Technical report, Oregon State University, Corvallis, 1989.

14 T. Dietterich, Hierarchical reinforcement learning with the MAXQ value function decomposition, Technical report, Oregon State University, 1997.

15 M.M. Drugan, Synergies between evolutionary algorithms and reinforcement learning, in *Genetic and Evolutionary Computation Conference, GECCO 2015*, pp. 723–740.

16 M.M. Drugan and A. Nowé, Designing multi-objective multi-armed bandits algorithms: A study, in *The 2013 International Joint Conference on Neural Networks, IJCNN*, 2013.

17 M. Glavic, R. Fonteneau and D. Ernst, Reinforcement learning for electric power system decision and control: Past considerations and perspectives, 20th IFAC World Congress, *IFAC-PapersOnLine* 50(1) (2017), 6918–6927.

18 S. Gu, E. Holly, T. Lillicrap and S. Levine, Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates, in *Proceedings 2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017.

19 M. Hausknecht, J. Lehman, R. Miikkulainen and P. Stone, A neuroevolution approach to general Atari game playing, *IEEE Transactions on Computational Intelligence and AI in Games*, 2013.

20 M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M.G. Azar and D. Silver, Rainbow: Combining improvements in deep reinforcement learning, in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

21 S. Hochreiter and J. Schmidhuber, Long short-term memory, *Neural Computation*, 9(8) (1997), 1735–1780.

22 T. Jaakkola, M.I. Jordan and S.P. Singh, On the convergence of stochastic iterative dynamic programming algorithms, *Neural Computation*, 6 (1994), 1185–1201.

23 S. Knegt, M. Drugan and M. Wiering, Opponent modelling in the game of Tron using reinforcement learning, in *ICAART 2018: 10th International Conference on Agents and Artificial Intelligence*, 2018, pp. 29–40.

24 L. Kocsis and C. Szepesvári, Bandit based Monte-Carlo planning, in J. Fürnkranz, T. Scheffer and M. Spiliopoulou, eds., *Machine Learning: ECML 2006*, Springer, 2006, pp. 282–293.

25 A. Krizhevsky, I. Sutskever and G.E. Hinton, Imagenet classification with deep convolutional neural networks, in F. Pereira, C. Burges, L. Bottou and K. Weinberger, eds., *Advances in Neural Information Processing Systems* 25, 2012, pp. 1097–1105.

26 A. Lazaric, Transfer in reinforcement learning: A framework and a survey, in M. Wiering and M. van Otterlo, eds., *Reinforcement*

Learning: State-of-the-Art, Springer, 2012, pp. 143–173.

27 L.J. Lin, *Reinforcement Learning for Robots Using Neural Networks*, PhD thesis, Carnegie Mellon University, Pittsburgh, 1993.

28 M.L. Littman, *Algorithms for Sequential Decision Making*, PhD thesis, Brown University, 1996.

29 S. Lucas and T. Runarsson, Temporal difference learning versus co-evolution for acquiring Othello position evaluation, in *Computational Intelligence and Games, 2006 IEEE Symposium*, 2006, pp. 52–59.

30 V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, Playing atari with deep reinforcement learning, arXiv:1312.5602, 2013.

31 V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, et al., Human-level control through deep reinforcement learning, *Nature* 518(7540) (2015), 529–533.

32 A.W. Moore and C.G. Atkeson, Prioritized sweeping: Reinforcement learning with less data and less time, *Machine Learning* 13 (1993), 103–130.

33 D.E. Moriarty and R. Miikkulainen, Discovering complex Othello strategies through evolutionary neural networks, *Connection Science* 7(3) (1995), 195–209.

34 S. Nemati, M.M. Ghassemi and G.D. Clifford, Optimal medication dosing from suboptimal clinical examples: A deep reinforcement learning approach, In *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2016, pp. 2978–2981.

35 A. Nowé, P. Vrancz and Y.M. De Hauwere, Game theory and multi-agent reinforcement learning, in M. Wiering and M. van Otterlo, eds., *Reinforcement Learning: State-of-the-Art*, Springer, 2012, pp. 441–470.

36 D. Precup and R. Sutton, Theoretical results on reinforcement learning with temporally abstract options, in *Proceedings of the Tenth European Conference on Machine Learning (ECML'98)*, 1998.

37 D.M. Roijers, P. Vamplew, S. Whiteson and R. Dazeley, A survey of multi-objective sequential decision-making, *Journal of Artificial Intelligence Research* 48 (2013), 67–113.

38 D.E. Rumelhart, G.E. Hinton and R.J. Williams, Learning internal representations by error propagation, in *Parallel Distributed Processing*, Vol. 1, MIT Press, 1986, pp. 318–362.

39 G.A. Rummery and M. Niranjan, On-line Q-learning using connectionist systems, CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

40 A.L. Samuel, Some studies in machine learning using the game of checkers, *IBM Journal on Research and Development* 3 (1959), 210–229.

41 J. Schmidhuber, Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts, *Connection Science* 18(2) (2006), 173–187.

42 J. Schmidhuber, Deep learning in neural networks: An overview, *Neural Networks* 61 (2015), 85–117.

43 A. Shantia, E. Begue and M. Wiering, Connectionist reinforcement learning for intelligent unit micro management in Starcraft, in *The 2011 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2011, pp. 1794–1801.

44 S.M. Shortreed, E.B. Laber, D.J. Lizotte, T.S. Stroup, J. Pineau and S.A. Murphy, Informing sequential clinical decision-making through reinforcement learning: an empirical study, *Machine Learning* 84(1-2) (2011), 109–136.

45 D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis, Mastering the game of Go with deep neural networks and tree search, *Nature* 529(7587) (2016), 484–489.

46 D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan and D. Hassabis, Mastering chess and shogi by self-play with a general reinforcement learning algorithm, arXiv:1712.01815, 2017.

47 D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel and D. Hassabis, Mastering the game of go without human knowledge, *Nature* 550 (2017), 354.

48 S. Singh, T. Jaakkola, M. Littman and C. Szepesvari, Convergence results for single-step on-policy reinforcement-learning algorithms, *Machine Learning* 38(3) (2000), 287–308.

49 R.S. Sutton, Learning to predict by the methods of temporal differences, *Machine Learning* 3(1) (1988), 9–44.

50 R.S. Sutton, Generalization in reinforcement learning: Successful examples using sparse coarse coding, in D.S. Touretzky, M.C. Mozer and M.E. Hasselmo, eds., *Advances in Neural Information Processing Systems* 8, MIT Press, 1996, pp. 1038–1045.

51 R.S. Sutton and A.G. Barto, *Introduction to Reinforcement Learning*, MIT Press, 1998, 1st edition.

52 G. Tesauro, Temporal difference learning and TD-gammon, *Commun. ACM* 38(3) (1995), 58–68.

53 S. Thrun, Efficient exploration in reinforcement learning, Technical Report CMU-CS-92-102, Carnegie-Mellon University, 1992.

54 S. Thrun, Learning to play the game of chess, *Advances in Neural Information Processing Systems* 7 (1995), 1069–1076.

55 A.D. Tijsma, M.M. Drugan and M.A. Wiering, Comparing exploration strategies for Q-learning in random stochastic mazes, in *2016 IEEE Symposium Series on Computational Intelligence*, *SSCI*, 2016.

56 S. van den Dries and M.A. Wiering, Neural-fitted TD-leaf learning for playing Othello with structured neural networks, *IEEE Transactions on Neural Networks and Learning Systems* 23(11) (2012), 1701–1713.

57 M. van der Ree and M.A. Wiering, Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play, in *2013 IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL)*, 2013, pp. 108–115.

58 M. van Otterlo and M. Wiering, Reinforcement learning and Markov decision processes, in M. Wiering and M. van Otterlo, eds., *Reinforcement Learning: State-of-the-Art*, Springer, 2012, pp. 3–42.

59 H. van Seijen, M. Fatemi, J. Romoff, R. Laroche, T. Barnes and J. Tsang, Hybrid reward architecture for reinforcement learning, arXiv: 1706.04208, 2017.

60 Z. Wang, A. Boularias, K. Muelling, B. Schoelkopf and J. Peters, Anticipatory action selection for human-robot table tennis, *Artificial Intelligence* 247 (2017), pp. 399–414.

61 C.J. Watkins and P. Dayan, Q-learning, *Machine Learning* 8(3) (1992), 279–292.

62 C.J.C.H. Watkins, *Learning from Delayed Rewards*, PhD thesis, King's College, Cambridge, UK, 1989.

63 P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, PhD thesis, Harvard University, 1974.

64 S. Whiteson, Evolutionary computation for reinforcement learning, in M. Wiering and M. van Otterlo, eds., Reinforcement Learning: State-of-the-Art, Springer, 2012, pp. 325–355.

65 M. Wiering, Multi-agent reinforcement learning for traffic light control, in *17th International Conference on Machine Learning*, 2000, pp. 1151–1158.

66 M. Wiering, QV(lambda)-learning: A new on-policy reinforcement learning algorithm, in D. Leone, ed., *Proceedings of the 7th European Workshop on Reinforcement Learning*, 2005, pp. 29–30.

67 M. Wiering and M. van Otterlo, *Reinforcement Learning: State of the Art*, Springer, 2012.

68 M.A. Wiering and E.D.D. Jong, Computing optimal stationary policies for multi-objective Markov decision processes, in *IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning, 2007 (ADPRL 2007)*, IEEE, 2007, pp. 158–165.

69 M.A. Wiering and J.H. Schmidhuber, HQ-learning, *Adaptive Behavior* 6(2) (1997), 219–246.

70 M.A. Wiering and J.H. Schmidhuber, Efficient model-based exploration, in J.A. Meyer and S.W. Wilson, eds., *Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior: From Animals to Animats* 6, MIT Press/Bradford Books, 1998, pp. 223–228.

71 M.A. Wiering, M. Withagen and M.M. Drugan, Model-based multi-objective reinforcement learning, in *2014 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning, ADPRL*, 2014.

72 R.J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, *Machine Learning* 8 (1992), 229–256.

73 G.N. Yannakakis and J. Togelius, *Artificial Intelligence and Games*, Springer, 2018.