4. Reinforcement Learning

This section describes Reinforcement Learning referring to [7][16]. The idea of **R**einforcement **L**earning (RL) is simply copied from nature. For instance a child learns walking by trying. At first not much successful but after some time when it has learned, how to balance, how to use legs then unsuccessful attempts are reduced and walking becomes better and better. This process can be described as a search of choosing the right action in an appropriate situation, simply trial-and-error-search. If an action was successfully matched to a situation, it may be promising but to reach the goal this process must be kept on, i.e. the first step of a child may cause a „WOW", a reward. But already the next step could cause it to cry. Thus every action influences not only the current state but also the following states, hence the following success, i.e. the delayed reward. In conclusion this means a learning object achieves a goal by interacting with its environment. Learning means on one hand side, to remember that a match of an action to a certain state caused a benefit but on the other hand, to try new actions in order to get a bigger reward. This is meant as the trade-off between exploitation and exploration. Not only exploiting already as successful marked actions lead to a terminal state in an optimal way but also exploring new ones. To do so it needs an active object, an agent that is able to collect information from an uncertain environment. This information is used to identify the current state and to match an action to this state. Depending on the reward, received in a state, it will meet an assumption about the next step to do. A policy is a casting for the behaviour of an agent. Executing a certain policy results in the more or less success of the agent. A policy can either be a simple lookup table to associate an action with a state or a complex search strategy. The policy is adapted to an optimal policy while learning. The agent tries to increase the overall amount of rewards. To decide about reward or penalty when reaching a certain state, a mechanism is used to match a state with an appropriate reward, in other words a reward function. But a reward expresses only the immediate feedback from a certain state. Thus it might be possible that after passing a high rewarded state a series of penalty states follows. To take this long-term view in consideration it needs another map, which is called value-function. The value of a state marks the overall reward that is expected to get when choosing that state as starting point. The values are significant for planing future decisions. Thus the strategy to move forward and choose appropriate actions bases on the values because high values promise the optimal way to reach the aim. This sounds easy but actually it is a difficult challenge to plan the sequence of steps to reach final goal. An alternative to this state-value-function can be the overall return for choosing a certain action in a state and after that following an optimal policy. This kind of feedback is provided by action-value-functions. That means summarised, the state-value-function returns the value of achieving a certain state and the action-value-function returns the value for choosing an action in a state, whereas a value means the total amount of rewards until reaching terminal state. Also helpful in order to plan behaviour is a model of the environment. This model supplies an agent with information, about what action might be a promising choice in a certain state. In a stationary environment the values for choosing a certain action do not change, i.e. once an optimal action is matched with a state, exploiting is advisable. But if the values change over time in a non-stationary world exploring other actions rates better because non-greedy actions might have become better.

In a Reinforcement Learning framework the environment consists of a set of possible states S. Over the time the agent gets into certain states $s_{t \in} S$ per time step t. The agent as a learner and decision maker selects an action a_t from the set of possible actions $A(s_t)$ and executes it. Then the environment reacts to the action of the agent, i.e. the agent gets a reward r_{t+1} in the next time step also known as the reinforcement value. Rewards are numerical values. At the same time the agent passes to a new state s_{t+1} of the environment. In this scenario the agent's policy π_t is the process of mapping state representations to probabilities of selecting each possible action. The process of updating a policy to maximise the expected overall reinforcement is the general characteristic of a Reinforcement Learning Problem. $\pi(s, a)$ means the probability that $a_t = a$ if $s_t = s$. As the probabilities are updated in each step to approximate an optimal value, the policy adapts as well.

4.1. Reward functions

An agent tries to maximise its total reward in long term view. The kind of reward function depends on the task. Some tasks are limited somehow, e.g. by time. These limited runs are called episodes and consist of a set of non-terminal states S and a terminal state. The complete set of states is denoted as S^+. In this case a simple way to note the expected return of future rewards R_t after time step t can be simply the sum

$$R_t = r_{t+1} + r_{t+2} \ldots + r_T = \sum_{x=t+1}^{T} r_x$$

where

- r_T is the reward that leads to the terminal state.

Example for an episodic task:

An agent negotiates in order to buy a good. The terminal state can be a price at which the agent decides to buy. Thus it is tries to bargain somehow to persuade the opponent to decrease the price. There can also be a deadline for the agent. The agent may try to buy the good as cheap as possible until the date of expiration.

Continual tasks are such ones that never reach a final state. The agent is a kind of workaholic, always endeavoured to get its wages. This objective is found in continuous control processes like in reactors that are observed about for instance the temperature to keep it in the right range. The approach of summing up expected rewards does not properly work in this case because $T \to \infty$. Thus there is a need for a discount factor γ, a positive fraction, applied over time.

$$R_t = r_{t+1} + \gamma * r_{t+2} + \gamma^2 * r_{t+3} + \ldots = \sum_{k=t+1}^{\infty} \gamma^k r_{t+k+1}$$

where

- $0 \leq \gamma \leq 1$.

Choosing $\gamma = 0$ causes the agent to base its decisions only on selecting actions by taking care of an immediate reward r_{t+1}. This is also known as myopic-case. This behaviour can badly influence further future rewards.

Choosing $\gamma = 1$ lets the agent weight future rewards stronger and thus this is called the for-sight-case. In general continuous and episodic task can be described as

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$$

Episodic task: $\gamma = 0 \wedge T < \infty$

Continuous task: $0 \leq \gamma \leq 1 \wedge T \to \infty$

4.2. Markov property

An agent receives information in order to determine the current state. If this always contains all relevant information, it is called Markov. Speaking about relevant information means not the complete history of a state but everything that matters for future events. This is the Markov property and expresses that environment's next state s_{t+1} at time $t+1$ only depends on the representation of the current state and action. This property has the advantage that it is possible to foresight the next state and reward due to an action. As this is a good base for future decisions even when the Markov property is not completely satisfied it should be worked with an approximation of the Markov state. It can also increase the speed of learning if details are omitted from calculation in order to esteem Markov case.

Markov **D**ecision **P**rocess (MDP) describes a Reinforcement Learning task that states can be specified by Markov properties. A finite MDP means that state and action spaces are finite.

4.3. Reinforcement comparison methods

Assume an agent that earns a reward for executing an action and has to decide how to go on. It would be helpful to know if the reward is a high one or a low one to classify the action. To do so a reference is needed, e.g. comparing the reward with the average of all received rewards. If it is lower it is rated bad. If it is higher it is rated good. To rate an action after using it a preference factor can be applied. This factor is updated by comparing the reward with a reference reward.

4.4. Associative search

In non-associative tasks an agent just chooses the action depending on its rating. Thus a good action is reputed as good in every state. But this is just a simple model. Many times it is necessary to select the action depending on the current state, i.e. associating actions with states in order to learn a successful policy.

4.5. Value Functions in recursive notions

4.5.1. State-Value-Function

Following a policy π the state-value-function returns the value, i.e. the expected return for selecting a certain state *s*. Return means the overall reward.

$$V^{\pi}(s) = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\}$$

where

- $E\pi$ is the expected value by following policy π .

4.5.2. Action-value-function

Following a policy π the action-value-function returns the value, i.e. the expected return for using action *a* in a certain state *s*. Return means the overall reward.

$$Q^{\pi}(s,a) = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\}$$

where

- $E\pi$ is the expected value by following policy π .

4.6. Bellman-Equation

The Bellman-Equation expresses the relationship between a value of a state and the value of a successor state.

$$V^{\pi}(s) = \sum_{a \in A} \pi(s,a) * \sum_{s \in S'} P^{a}_{ss'} * (R^{a}_{ss'} + \gamma * V^{\pi}(s'))$$

where

- $R^{a}_{ss'}$ is the reward for passing from state s to successor state s'.
- $V\pi(s')$ returns the state-value of successor state s'.
- $P^{a}_{ss'}$ is the probability to access state s' when choosing action a in state s.
- $\pi(s,a)$ is the probability to choose action a in state s.
- A is the action space.
- S is the state space.

The Bellman-Equation calculates the value of a state by considering all available options and assessing each by its likelihood of appearance.

Example:

Set of actions available in each state: $A_s = \{a, b\}$

Set of states: $S = \{s_1, s_2, s\}$

Set of rewards: $R = \{r_1, r_2, r_3, r_4\}$

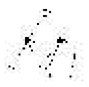

Figure 1: Backup diagram

The diagram shows that when initially in state s action a is selected, the successor state is s_1 and reward r_1 is expected but also r_2 is expected passing to s_2. If in state s action b is chosen, reward r_3 is expected and leads to s_1 but also reward r_4 is expected and leads to state s_2.

This diagram can be described by the following Bellman-Equation:

$$V(s) = \pi(s,a) * (P^{a}_{ss_1} * [r_1 + \gamma * V(s_1)] + P^{a}_{ss_2} * [r_2 + \gamma * V(s_2)]) + \pi(s,b) * (P^{b}_{ss_1} * [r_3 + \gamma * V(s_1)] + P^{b}_{ss_2} * [r_4 + \gamma * V(s_2)])$$

The aim of an agent is to find the optimal policy. There is at least always one policy that is better or equal than all other policies. This policy is called the optimal policy. There is only one optimal-value-function, which is the base for an optimal policy. The optimal value-function chooses an action a in a state s not by following a policy but by choosing the maximum.

The optimal-value-function can be noted as follows:

$$V^{*}(s) = \max_{a \in A(s)} \left(\sum P^{a}_{ss'} \cdot [R^{a}_{ss'} + \gamma * V^{*}(s')]\right)$$

As this equation represents a system of equations the "only" thing that has to be done is to solve it. Because of the complexity of many problems this is almost impossible with today's computer capacities. Reinforcement Learning provides a way of approximation in order to find a solution. For the action-value-functions there is a Bellman-Equation available as well. Among all policies there is also one that is at least equal or better than the others using the optimal action-value-function. This best action-value-function in notion of Bellman optimality equation is

$$Q^{*}(s,a) = \sum_{s'} P_{ss'}^{a}[R_{ss'}^{a} + \gamma * \max_{a'}(Q^{*}(s',a')]$$

The difficulties of solving this equation system are the same as for *V**.

4.7. Temporal-Difference learning

There are several different approaches to solve the Reinforcement Learning problem. Dynamic programming algorithms require a complete perfect model of the environment and thus allow to achieve an optimal policy by approximating the optimal V^{*} or Q^{*}. But they need a high calculation cost. Unlike Dynamic programming Monte Carlo methods do not need a model of the environment but learn from samples. Another kind of Reinforcement Learning is **T**emporal-**D**ifference (TD) learning. In this section it is briefly summarised, as it is important for the further work. In Temporal-Difference learning an earlier value is updated by calculating a difference between that value and a later value. Let S be the set of all states and $S \neq \{\}$. May V be the function that returns the value of a state $s \in S$. Then

$$V(s) \leftarrow V(s) + \alpha\,(V(s') - V(s))$$

where

- $0 \leq \alpha \leq 1$

α is the learning rate. It is a step-size parameter of a positive fraction. It is used to progressively approximate the optimal policy and therefor decreased over time in stationary environments. If the optimal policy is reached the learning rate should be zero as the system learns from its experience. It is necessary to generalise knowledge in order to learn from the past. How well this is done is decisive for the learning process.

Usually either the value of a state and the value for using the available actions are unknown. Thus the values are estimated by using a function. For instance let a be an action and $Q^{*}(a)$ be the function that returns the actual value for choosing a in a certain state. Then $Q_t(a)$ is the action-value function that estimates the value for choosing a in a certain state and at step t. A simple method could be used to calculate the average rewards.

$$Q_t(a) = \begin{cases} \dfrac{r_1 + r_2 + \ldots + r_{k_a}}{k_a}, & k_a \geq 1 \\ Q^{*}(a), & k_a \rightarrow \infty \\ 0, & k_a = 0 \end{cases}$$

where

- k_a is the quantity of a reward that was given for the action a

If in a certain state the action-value for each possible action is calculated, then in the next step an action will be chosen. Here the balance between exploitation and exploration has to be considered. Always choosing the action with the highest action-value, i.e. the greedy action only exploits. But selecting a non-greedy action has the risk of exploration. The balance between both can be adjusted by randomly, uniformly, independently exploring once in a while with a probability ε **.** Selecting greedy or ε -greedy depends on the problem. If the variance of a reward that is distributed with normal probability is small, then choosing always the greedy action is a good way. But if the rewards are noisier then exploring other actions makes it more likely to find an optimal action. To combine both ways, ε may be decreased over time. Just choosing one of the non-greedy actions with equal probability without taking care, if it is the worst or the best choice, is a disadvantage of this strategy. To avoid this a Boltzmann distribution can be used. It ensures that actions that promise a high reward are chosen more likely. The probability *$P(a_t, s)$* of choosing an action *a* at time step *t* in state *s* is then as follows

$$P(a_t) = \frac{e^{Q_{t-1}(a,s)/\tau}}{\sum_{b \in A} e^{Q_{t-1}(a,b)/\tau}}$$

where

- *A* is the set of all actions
- $\tau$is the temperature, a positive factor.

Is the temperature τ = *0* then the function behaves as a greedy method. But a high temperature effects action probabilities to be nearly equal. This kind of action selection rule is known as *Softmax* method.

In non-stationary environments, e.g. in multi-agent systems where the conditions in an already visited state are different because of the behaviour of the other agents, recent rewards can be rated higher than old ones.

A very popular TD learning algorithm is *1-step Q-learning*. It is defined as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma * \max_{\alpha \in A}(Q(s_{t+1}, a) - Q(s_t, a_t))]$$

where

- $\gamma$is the discounting factor to influence myopic or foresight behaviour
- $\alpha$is the learning rate.

As easy to guess the Q-learning algorithm approximates the action-values in order to approach the optimal action-value-function Q^*. The current policy effects only the state-action pairs that are selected and updated. Q-learning can be characterised by simultaneously learning the dynamics of the environment and adapting the policy in an iterative way to an optimal policy. It is guaranteed that the action-values converge to the optimal values in an infinite run by selecting each action in the appropriate state infinitely often. In *1-step-Q-learning* values from one step in the past are updated, i.e. update an earlier estimate according to the difference of a later estimate considering appropriate discounts. If more than one step in the past is updated, e.g. *2,3…n* it is appropriately called an n-step method.

4.8. Eligibility traces

Eligibility Traces are a basic concept of Reinforcement Learning. This technique can be applied to any Reinforcement Learning algorithm in order to increase performance. Eligibility traces handle delayed rewards and thus events like a certain state or choosing a certain action are temporary recorded for learning changes. When updating last events (e.g. the value of the last states) there is also an additional variable associated with each state. This is called an eligibility trace and the notation for state *s* at time *t* is $e_t(s) \in \Re^+$.

4.8.1. Accumulating traces

$$e_t(s) = \begin{cases} \gamma * \lambda * e_{t-1}(s), s \neq s_t \\ \gamma * \lambda * e_{t-1}(s) + 1, s = s \end{cases}$$

where

- γ is the discount factor
- λ is the trace-decay factor

This is called an *accumulating trace* because recently visited states are accumulated at each time. The trace fades to zero when it is not visited any more.

Learning changes due to Reinforcement Learning events that are applied to the states. The trace supplies the information of the degree. A Reinforcement Learning event comes as

$$\delta_t = r_{t+1} + \gamma * V_t(s_{t+1}) - V_t(s_t)$$
$$\Delta V_t(s) = \alpha * \delta_t * e_t(s), \forall s \in S$$

- For $\lambda = 0$ only the last visited state has a trace $\neq 0$ and thus only this is updated.
- For $0 < \lambda < 1$ past states are updated according to their trace value. The update is smaller if the state has been visited a long time ago. They are given less credit for temporal difference error.
- For $\lambda = 1$ only γ has influence on the trace.
- For $\gamma = 1$ there is no decay in the trace.

For state-action pairs the notation is like following

$e_t(s, a)$: Trace for state-action pair s, a

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha * \delta t * e_t(s,a), \forall s, a$$
$$\delta_t = r_{t+1} + \gamma * Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$$
$$e_t(s,a) = \begin{cases} \gamma * \lambda * e_{t-1}(s,a) + 1, s = s_t, a = a_t \\ \gamma * \lambda * e_{t-1}(s,a), otherwise \end{cases} \forall s, a$$

4.8.2. Replacing traces

A trace of a state could exceed 1 when it is visited before the trace value decayed to zero. In replacing traces therefor the value is set to one.

$$e_t(s,a) = \begin{cases} \gamma * \lambda * e_{t-1}(s,a), s = s_t, a = a_t \\ 1, otherwise \end{cases}$$

or

$$e_t(s,a) = \begin{cases} 1 + \gamma * \lambda * e_{t-1}(s,a), s = s_t, a = a_t \\ 0, s = s_t, a \neq a_t \\ \gamma * \lambda * e_{t-1}(s,a), s \neq s_t \end{cases}$$

Thus multiple visiting of a bad state or choosing a bad action prevents from increasing their traces disproportionately.