## Reinforcement Learning problems

As the task as hand was not only to provide a reinforcement learning agent, but also to convert a game itself into something the agent can successfully play, I will in this chapter go into detail about Reinforcement Learning in general, to give insights into why I did what I did. Also, I will try to keep this stuff as general as possible, getting into detail when speaking about the used algorithms.

[The sense of this chapter is to give an intro of MDPs and RL. It shall also go into enough details on how to specify an MDP such that an RL agent can learn on it, because a big part of the work was exactly that. It’s supposed to end with SARSA and Q-learning as the two Ideas on how to perform RL]

Machine Learning can mainly be subdivided into three main categories: Supervised Learning, Unsupervised Learning, and Semi-supervised learning. The first deals with direct classification or regression using labelled data (i.e. it uses pairs of datapoints with their corresponding category or value). In unsupervised learning, no such label exists, and the data must be clustered into meaningful parts without any knowledge, by for example grouping objects by similarity of their properties.

What will be mainly considered in this thesis will be a certain kind of semi-supervised learning: Reinforcement learning. In Reinforcement Learning (RL), instead of labels for the data, there is a *weak teacher,* which provides feedback to the actions the agent took.

The metaphor behind RL is that of an agent and an environment. The agent makes observations in the environment, taking actions and receiving rewards. In contrast to the classical ML approaches, in RL the agent is also responsible for exploration, as he has to acquire his knowledge actively. As the environment does not explicitly provide actions the agent has to perform, its goal is to maximize its cumulative reward until a training episode ends. In basically all modern Reinforcement Learning approaches, the environment is discretized into a so-called **Markov Decision Process** (MDP). Formally, an MDP is a 5-tuple <S, A, P, R, γ> consisting of the following:

S - set of states

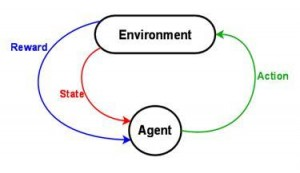
A - set of actions

Pa(s, s’) - transition probability function from state s to state s’ under action a  
Ra(s, s’) - reward function for the agent in state s if the environment moves to s’

γ - the discount factor for future rewards.

In the following, I will refer to the transition Probability function for a specific state s’ as s’ = P(s,a) and to the reward function for a specific reward as r = R(s,a). Note, that both of these functions may be indeterministic.

The markov property holds in this case, if the future state of the environment depends only on the current state as well as the action of the agent.

In the process of reinforcement learning, the agent will encounter states s of the environment, performing actions a. The future state st+1 of the environment may be indeterministic, but depends on the current state st as well as the action of the agent at. Additionally, the agent receives a reward r, depending on his action as well as the environment. In many RL problems, the full state of the environment is not known to the agent, and it only perceives an observation depending on the environment: o(s). (“partial observability”) Additionally the agent knows when a final state of the environment is reached, terminating the current training episode.   
(An episode then consists of a sequence of states, actions and rewards: ((s0,a0,r0),(s1,a1,r1),(s2,a2,r2),...)

A training example for the agent thus consists of the tuple <o(s), a, r, o(s), t>. (From now on, when I mean the state of the environment, I will refer to it as se, while reserving s for the agent’s observation of the environment o(se).

In the process of reinforcement learning, the agent now tries to perform as well as possible in this environment, which is unknown to him.



The agent itself tries to learn the action-policy pi: S→A, which tells him what actions to take in what state. As the agent does not have any supervised data for the actions, it must learn some kind of measure for the value of of a state or an action.

A measure of the value of a state can be calculated by the immediate reward plus the discounted future reward: V(st)= [simga t = gamma rt plus gamma2rt+1 blabla]. (discounted is useful because the agent is less sure the more into the future we go) V(s) can be interpreted as the maximally archievable reward starting in state s. [das schon nach max pi machen, LATEX!!!]

If the value of every state is known, then the optimal policy pi\* is easily calculated as the one, maximizing the the discounted future reward V(st) for every state s. However, for an agent, there are two practical problems, namely that neither the successor function Pa(s,s’) nor the reward function Ra(s) are known to the agent. While so-called **model-based** reinforcement learning tries to learn both of those explicitly to reconstruct the entire MDP (to afterwards simply solve it), **model-free** agents use a different measure of quality: the Q-value. Q\*(s,a) is defined as R(s,a)+ yV\*(P(s,a)), [dadrunter noch ne formel haben wie V von Q berechnet wird). When an agent knows the Q-value for each action of a state, the optimal action to take in each state s is simply maxaQ(s,a). Model-free reinforcement learning agents don’t bother to explicitly learn the reward- and transition function, but instead try to estimate the quantity Q\* instead. For that, they have an estimate of the current action-value function, Qpi In very small settings with a highly limited amount of discrete states and actions, the respective values can be stored in a table, whereas for areas of interest, the estimate of the Action-value-function Qpi is calculated using a kind of nonlinear function approximator. Throughout exploration of the environment, the agent collects more information about the environment and continually updates its estimate Qpi,

In the following segment, I will explain explain the Q-learning algorithm.

Note for that, that I will refer to the underlying, optimal Q-function as Q\*(s,a), defined as maxpiQpi(s,a). A policy, that at every step t takes the maxaQ\*(st,a) guarantees maximum cumulative reward, and is referred to as an optimal policy. The current estimate of an agent, using its current policy pi, is referred to as Qpi(s,a), its current policy is pi

[ The goal of the agent is now to find the optimal policy V\*, yielding the maximal reward

for every state. Because the agent cannot learn the policy on its own (as no supervised data is available), the best way to]

## Reinforcement Learning

The goal of a reinforcement-learning agent is to continually update its Q\*-estimate, Qpi, such that it can follow the policy that always takes the action giving the highest cumulative reward. To update its Q-function, the agent can either use full pairs of State,action,reward,state,action-tuples (SARSA, source), or Q-learning (source).

Throughout the process of RL, the agent continually improves its estimates Qpi of Q\*. An optimal solution would be to minimize the squared difference between Qpi and Q\*, to take the derivative and perform gradient descent in order to minimize that difference. However, as the “real” Q-function Q\* is unknown to the agent, that is not possible. Instead, the agent performs iterative approximation, where it uses the knowledge it gains to update its estimates of Q.

For that, it is important to keep in mit the recursiveness of the Q-function:  
Qpi = r(s,a) + yVpi (vorher schon vpi formally defined as cumulative rewards?)  
And because as shown in (2), Vpi = max…, it holds that  
Qpi = r(s,a) + ymaxa’s’a’  
Thus, when using the update rule <rule>, the agent gains continually better updates of Q\* (as proven in source).

The update rule in an online-learning, model-free, active RL agent is Temporal difference learning: the agent compares what he thought would happen with what actually happened. For that,

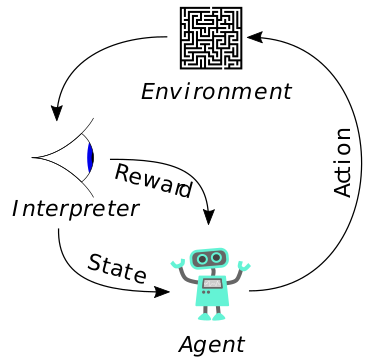
Left to do:

* Credit assignment problem
* Exploration vs exploitation
* Am anfang tables, wenn zu big → DeepNet (function approximators!)
* Medium post!
* Offline vs online, offpolicy vs onpolicy, model-free vs model-based, policy gradient methods

Sources for this part:

* Heidemanns slides
* <https://en.wikipedia.org/wiki/Markov_decision_process>
* <https://en.wikipedia.org/wiki/Reinforcement_learning>
* <https://github.com/ahoereth/ddpg/blob/master/exploration/FrozenLake.ipynb>
* Medium post
* Heidis slides
* Russel, norvig
* Die originalen RL und Q-learn paper
* Valentins und melisas präsi
* Meine und bennis präsi
* Report von valentin melisa und mir und benni
* Nature paper
* Bellmann equation!!!!!!!!!!!!!!!!!!!!!!!!!11111111111111111einseinseinseinself

Converting the game into an RL problem

* Dass interestingly die environment indeterministic ist, as can be seen when looking at individual runs of a determinstic agent. (even though performing the action a fixed Unity-time after the last)
* 
* As already mentioned, an RL problem looks like this: This, paired with technical stuff, leads to the use of a server, which stores the environment-state, whereas the agent only percieves a subset of that o(s), defined in a function, ...
* Dass das model, part-of my agent, eben seiner approximation der policy function pi: S→A entspricht [The agent will have an estimate of V\*, and for that it can use any function approximator]
* Nochmal genau auf alles eingehen: state ist X, actions sind y,...