Artificial Intelligence for Games

Prof. Dr. Ullrich Köthe

Introduction to Reinforcement Learning

Report by

Denis Zavadski

July, 2019

Abstract

Since started in the 1950s, *reinforcement learning* made a huge leap forward in the last few years in terms of success and popularity. Nowadays agents trained via reinforcement learning are able to keep up and even surpass humans in real-time in games as complex go and starcraft, which was said to be to complex for AI to do. This report aims at giving a brief introduction into the topic of reinforcement learning and at providing the basic tools to deal with most of the typical reinforcement learning problems while explaining the general ideas behind them.

Contents

1 Introduction

Normally the standard practice in machine learning is training your algorithm supervised on a huge dataset with some input information and the according solutions to solve a problem with i.i.d. problem-data. But let's assume that the problem is now to teach a machine to play chess. First of all, the data is not any longer i.i.d., since e.g. if you move your pawn to a particular field, this field is not longer available to your other pieces. Furthermore there seems to be no unique right move, but a set of reasonable moves to decide from, the outcome of which is not always immediately clear, but might be delayed e.g. in case of sacrifices. This is where reinforcement learning comes in handy, since it works especially well in feedback-delayed, non i.i.d. environments where the agents behaviour directly affects the subsequent data.

1.1 Basics

The main changes with the reinforcement learning approach are that the agent has no needs for a supervisor to decide on an *action* a_t from a given set of available actions A_t which may change regarding the agent's *state* s_t for every time step t. In general there are two kinds of states. The *environmental state* is a complete description of the environment and contains all possible information about everything within the problem, but may be not or only partially visible to the agent. The other state is the *agent state*, which is the agent's personal representation of the environment. If the environmental state is equal to the agent state, the environment is called *observable*. In the following we will look only on such observable environments. In addition a state is called *Markov state* if the probability to get to the next state is only dependent on the current state the agent is in. In other words, all predictions about the future are only dependent on the presence and not the past.

To decide which action from the actionset may be the best, the agent needs to process *reward signals Rt*, which are scalar feedback signals obtained from the environment after each action. The agents goal is to maximize his overall reward. Therefore it must approximate all rewards it is going to get, which is equal to maximizing the *return* G_t

$$G_t = R_t + \gamma R_{t+1} \cdots = \sum_{k=1}^{\infty} \gamma^k R_{t+k} \tag{1}$$

in which $\gamma \in [0, 1]$ is the *discount factor*. Such a discount factor is necessary to prevent infinite returns in the case of cyclic rewards and to model uncertainty into the decision, since the further the agent tries to predict the future, the bigger the uncertainty and the less weight the expected reward for that future action has.

To maximize the overall return the agent follows and improves a certain policy π which defines the agent's behaviour by mapping from the set of states S to the set of actions A

$$\pi : S \longrightarrow A, \qquad s \in S, a \in A \tag{2}$$

$$\pi(s) = a \qquad \text{(deterministic)} \tag{3}$$
$$\pi(a|s) = \mathbb{P}[a|s] \qquad \text{(stachastic)} \tag{4}$$

The policy is called deterministic if for every certain state the result is one particular action, and stochastic if the result is a probability distribution over possible actions.

To be able to evaluate and improve the policy π the agent relies on *value functions*

$$v : S \longrightarrow \mathbb{R}, \qquad v_\pi(s) = \mathbb{E}_\pi[G_t|s] \tag{5}$$
$$q : S \times A \longrightarrow \mathbb{R}, \qquad q_\pi(s,a) = \mathbb{E}_\pi[G_t|s,a] \tag{6}$$

The *state-value-function* v_π which returns the expected overall return, starting in a state s and following policy π, and the *action-value-function* q_π which returns the expected return starting in a state s, performing a certain action a and from the next state following the policy π.

Given a policy the agent interacts with it's environment as shown in Figure 1. First the agent, represented by the brain, finds itself in a certain state in his environment, represented by the earth. The agent applies his policy to the state and picks an action, which has a direct impact on the environment. The environment then returns the agent a reward signal and it's new state before the circle restarts.

The tasks an agent may perform in such a manner are evaluating a given policy by predicting the return for every given state if the policy is followed, which is called the *prediction* task, and finding the best possible policy by finding the actions leading to the maximum return for every state, which is called the *control* task. Obviously the agent has to be able to solve the prediction task to be able to solve the control task.

2 Model-based Learning

2.1 Bellman Equation

Now let's take a closer look into the inner structure of the value functions. First we can write the definition (5) slightly different

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|s] \\ &= \mathbb{E}_\pi[R_t + \gamma G_{t+1}|s] \\ &= \mathbb{E}_\pi[R_t + \gamma v(s_{t+1})|s] \qquad \text{(Bellman Equation)} \end{aligned} \tag{7}$$

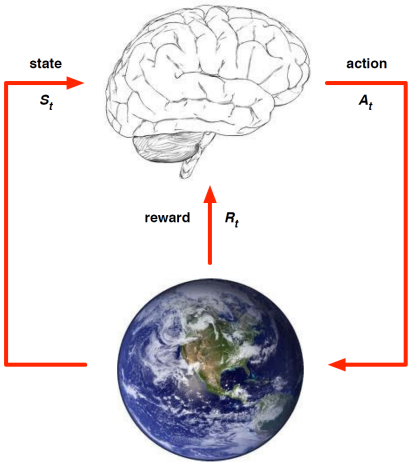

Figure 1: Agent-Environment-Interaction [1]

This iterative relationship of the value function of the current state with the value function of the successor state is called *Bellman Equation*. To solve it we can write it in matrix notation

$$v = \mathcal{R} + \gamma \mathcal{P} v \tag{8}$$

which can be easily rewritten into

$$\begin{aligned} (\mathbb{I} - \gamma \mathcal{P}) v &= \mathcal{R} \\ v &= (\mathbb{I} - \gamma \mathcal{P})^{-1} \mathcal{R} \end{aligned} \tag{9}$$

This can be done analogously for the state-value-function q.

While finding the value of an arbitrary value function can be done by just solving the Bellman equation there are still two major problems. The first problem is, that this works only for small matrices, since the operation is $\mathcal{O}(|S|)$. The second problem is that in general, we are not just interested in arbitrary value functions, but in the *optimal value functions* which are defined as the maximum value functions with respect to the policy.

$$v_*(s) = \max_\pi v_\pi(s) \tag{10}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{11}$$

Through the definitions of optimal value functions we can now define an ordering of policies as follows

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s), \quad \forall s$$

which already implies the definition of an *optimal policy* π_*

$$\pi_* \geq \pi, \quad \forall \pi \tag{12}$$

Consequently from (10),(11),(12) we get that the value functions of the optimal policy are always optimal.
With these new insights we can now formalize the goal of reinforcement learning algorithms, which is finding the optimal policy.

2.2 Policy Iteration

Through the maximum operation the Bellman Equation loses its linearity and now can not be solved the same way as before in just one go. Nevertheless we can utilize the matrix notation of the Bellman Equation (8) to evaluate and improve given policies iteratively. Let's assume we are given a certain policy π_0. For the evaluation task we just need to obtain all values of the according state-value-function v_{π_0}, which can be done by using (8)

$$v^{k+1} = \mathcal{R}^{\pi_n} + \gamma \mathcal{P}^{\pi_n} v^k \tag{13}$$

with the iteration number k. In our case the policy index is just $n = 0$. The initial values for v^0 may be chosen arbitrary as e.g. $v^0 = \vec{0}$. This iterative method works since with every iteration we are getting an additional piece of reality in terms of the actually received rewards. This gets our values every time slightly closer to the true state-value-function v_{π_n}. If we are only interested in solving the prediction task, we are done as soon as we are satisfied with the outcome, which could be measured e.g. with the euclidean distance

$$v_{\pi_n} \simeq v^k \quad \Longleftrightarrow \quad |v^k - v^{k-1}| < threshold$$

Although if we are interested in finding the optimal policy, then the next step is to improve our current policy π_n by acting greedy w.r.t. the newly obtained state-value-function

$$\pi_{n+1} = greedy(v_{\pi_n}) \tag{14}$$

This way we get our new policy which we will have to evaluate in order to improve our state-value-function to get another new, better policy and so on. This cyclic process is illustrated in Figure 2 where the left image shows the algorithm starting with an arbitrary v and a policy π, evaluating the policy until $v = v_\pi$ (arrow up), then improving the policy by acting greedy w.r.t. v (arrow down) and so on until the optimal policy and the optimal state-value-function are reached. The right graph shows the same process, but exhibits a stronger emphasis on the cyclic nature of the algorithm and underlines that once the optimal state-value-function and the optimal policy are reached, they are not changing any more.

2.3 Value Iteration

If we are not given any policy, we can utilize the following theorem

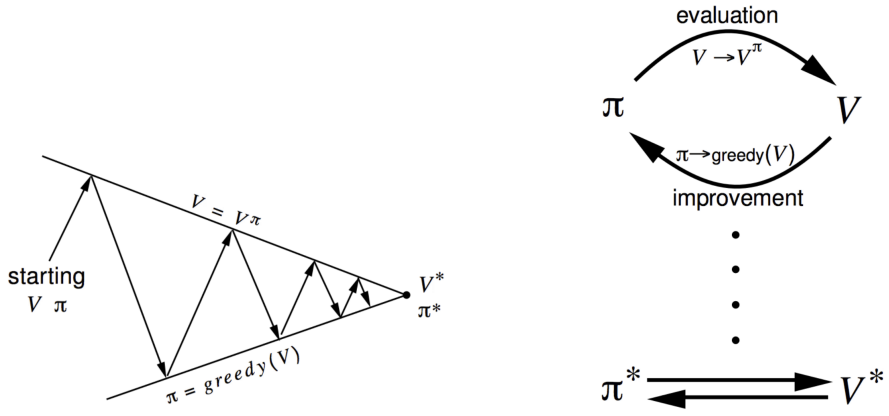

Figure 2: Policy Iteration [2]

Principle of Optimality
A policy $\pi(a|s)$ archieves the optimal value $v_\pi(s) = v_*(s)$ from state s if and only if for any state s' reachable from s, π archieves the optimal value $v_\pi(s') = v_*(s')$ from state s'.

Now we can subdivide any optimal policy into the optimal first action a_* followed by an optimal policy from the successor state s'. Now if we start with just arbitrary values for v^0, we make a one step look ahead by using the *Bellman Optimality Equation*

$$v^{k+1} = \max_{a \in A} \mathcal{R}^a + \gamma \mathcal{P}^a v^k \tag{15}$$

which also converges to the optimal state-value-function although intermediate value functions may not correspond to any policy.

2.4 Example

Now lets take a closer look on how such previously described interactions may be actually calculated. Consider the following toy example in Figure 3. A delivery company gets an order and has to deliver it to another city. They decide in $p_{\text{freeway}} = 0.8$ of all cases to take the motorway, which leads them with the probability of $p_{\text{jam}} = 0.4$ into a traffic jam $\rightarrow p_{\text{traffic jam}} = p_{\text{freeway}} + p_{\text{jam}} = 0.32$. While driving the driver may reconsider his decision, or drive through to his destination. If the motorway is free, he may try to drive faster to finish his job earlier, while risking an accident.

The ordered set of states in this problem is $S = \{$GET ORDER, FREE ROAD, TRAFFIC JAM, FREE SLOW ROAD, ACCIDENT, DELIVER ORDER FASTER, DELIVER ORDER, DELIVER ORDER SLOWLY$\}$ with the according action set $A = \{$FREEWAY, DRIVE THROUGH CITY, RACE LIKE A MADMAN, DRIVE THROUGH ESCAPE JAM$\}$

Furthermore in this particular case we are provided the *reward function* $\mathcal{R}$ and the *state-transition-probability-matrix* $\mathcal{P}$ which implicitly follows from the probabilities and the actions. These are defined as

$$s, s' \in S, a \in A$$

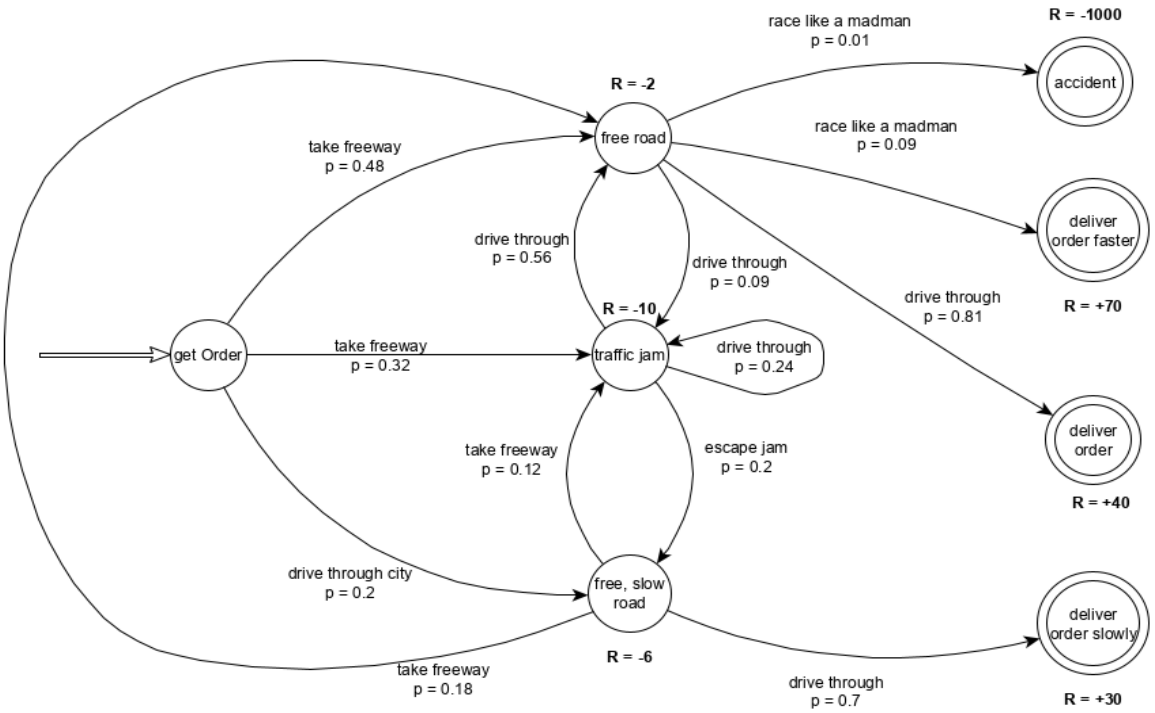

Figure 3: Model Based Delivery Problem

$$\mathcal{P}^a_{s's} = \mathbb{P}[s_{t+1} = s' | s_t = s, a_t = a] \tag{16}$$

$$\mathcal{R}^a_s = \mathbb{E}[R_t | s_t = s, a_t = a] \tag{17}$$

The tuple of the reward function and the state transition probability matrix $\langle \mathcal{R}, \mathcal{P} \rangle$ is called a *model*. Model based algorithms use exactly that to solve the prediction task.

In our particular example we get

$$\mathcal{R} = \begin{bmatrix} & 0 \\ - & 2 \\ - & 10 \\ - & 6 \\ - & 1000 \\ + & 70 \\ + & 40 \\ + & 30 \end{bmatrix} \quad \mathcal{P} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ ,48 & 0 & ,56 & ,18 & 0 & 0 & 0 & 0 \\ ,32 & ,09 & ,24 & ,12 & 0 & 0 & 0 & 0 \\ ,2 & 0 & ,2 & 0 & 0 & 0 & 0 & 0 \\ 0 & ,01 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & ,09 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & ,81 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0,7 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Now we can calculate the value functions for our states. Let's consider for example the state FREE ROAD and a discount factor $\gamma = 0$. Let the initial values of state-value-function to be equal to the rewards of the according state. With (5), (6) and (7) we get

$$
\begin{aligned}
v(\text{FREE ROAD}) &= -2 + 0.01 \cdot (-1000) + 0.09 \cdot 70 + 0.81 \cdot 40 + 0.09 \cdot (-10) \\
&= 25,8 \\
q(\text{FREE ROAD, RACE LIKE A MADMAN}) &= -2 + 0.1 \cdot (-1000) + 0.9 \cdot 70 \\
&= -39 \\
q(\text{FREE ROAD, DRIVE THROUGH}) &= -2 + 0,1 \cdot -10 + 0.9 \cdot 40 \\
&= 33
\end{aligned}
$$

So with the policy of the driver the overall value of the state FREE ROAD has after one step of policy iteration the expected return of $27,8$. The best action under consideration of $\gamma = 1$ would be DRIVE THROUGH with the expected return of 35 after one step look ahead.

3 Model free Learning

To know the state-transition-probabilities and the reward functions is not always possible, s.t. we are not always able to use the previously discussed policy and value iteration. Under the new circumstances we need to acquire the state-transition-probabilities and the reward functions by ourselves.

3.1 Monte Carlo Policy Evaluation

The solution to our lack of knowledge about the environment is going to be analysing the agent's direct experience. The goal is to approximate and modify our understanding of the values of the state-value-function of a particular state with the *Monte Carlo Algorithm (MC)* every time we actually visit that state. The idea behind the algorithm is to pick a terminated sequence (episode) and update the v for all visited states in the following way:

if the state s is visited for the first time in an episode,

- increment the counter $N(s)$, which counts the number of visits
- increment the total return over all already observed episodes for state s: $G_{tot}(s) = G_{tot}(s) + G_t(s)$ (remember, since we look on an already terminated sequence we know the true experienced G_t for every state of the sequence)
- update the approximation of the state-value-function $v(s) = G_{tot}(s)/N(s)$

What we are doing is basically updating our approximation towards the mean return, which by the law of large numbers converges to the optimal state-value-function.

By transforming the equation for the mean, we can compute it more elegantly

$$\begin{aligned}\mu_k &= \frac{1}{k}\sum_{j=1}^{k} = \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) \\ &= \frac{1}{k}(x_k + (k-1)\mu_{k-1}) \\ &= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}) \end{aligned} \tag{18}$$

With this modification our update rule becomes

$$\begin{aligned} N(s_t) &\longleftarrow N(s_t) + 1 \\ v(s_t) &\longleftarrow v(s_t) + \frac{1}{N(s_t)}(G_t - v(s_t)) \\ &\longleftarrow v(s_t) + \alpha(G_t - v(s_t)) \end{aligned} \tag{19}$$

with a custom learning rate $\alpha \in [0, 1]$ which can be used to replace the reciprocal visiting counter $N(s_t)^{-1}$

3.2 Temporal Difference Policy Evaluation

A slightly different approach is to also analyse sequences of the agent's experience, but to update the values $v(s)$ towards the experienced reward R_t and the approximated, discounted return of the successor state $\gamma v(s_{t+1})$. This is the idea behind the *Temporal Difference Algorithm* which has the update rule

$$v(s_t) \longleftarrow v(s_t) + \alpha(\underbrace{\underbrace{R_t + \gamma v(s_{t+1})}_{\text{TD target}} - v(s_t)}_{\text{TD error } \delta_t}) \tag{20}$$

with the custom learning rate α. This also converges to the optimal value function, since with every update we are updating towards reality through the actually experienced reward R_t

Unlike the MC algorithm, the TD algorithm does not have to wait until the sequence is terminated to update $v(s)$, which makes it more convenient to use since it can handle incomplete or cyclic experience on the fly whereas MC fails in such cases due to it's need to have a terminated sequence.

Even though both algorithms eventually converge to the optimal state-value-function, experience shows that the TD algorithm is usually more efficient than the MC algorithm.

3.3 Policy Iteration

The problem, why under the new circumstances we can not apply the previously discussed policy iteration, is that the iteration over $v(s)$ requires a model to be able to

update the agent's policy

$$\pi_{new} = \underset{a}{\operatorname{argmax}}\, \mathcal{R}_s^a + \gamma \mathcal{P}_{s's}^a v(s')$$

but since we do not have a model ($\hat{=}$ $\mathcal{R}$ and $\mathcal{P}$ are unknown) we need to modify the update rule. To do so, we can simply replace the state-value-function through the action-value-function which does not require the agent to know the probabilities any more. The new policy may then be obtained by

$$\pi_{new}(s) = \underset{a}{\operatorname{argmax}}\, q(s,a) \tag{21}$$

3.4 Exploration versus Exploitation

Before touching the control task with the MC and TD algorithms, we first need to know how to deal with the possibility of being stuck in local maxima. During the learning process the agent needs to solve two apparently contradicting tasks. The exploration task, the goal of which is to gain all the information possible to be able to chose the best actions, and the exploitation task, whose goal is to exploit the already gained information in order to actually maximize the return. The dilemma is that the agent wants to gain the information to make better decisions, without giving up on too much rewards while exploring.

A possible solution to this problem could be to perform such called ϵ-greedy exploration. With m actions available the agent picks a random action with a small probability of $\epsilon \in [0,1]$, which should decrease with the amount of gained information. The greedy action will then be chosen with the probability of $1-\epsilon$

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \operatorname{argmax}_a q(s,a) \\ \epsilon/m & \text{otherwise} \end{cases} \tag{22}$$

3.5 Monte Carlo and Temporal Difference Control

With the ϵ-greedy exploration we can now apply the same principle as in the model free policy iteration to solve the control task with the MC and TD algorithms. We simply take our previous update rules (19) and (20) and replace the state-value-function with the action-value-function s.t. the update rules become

$$v(s_t) \quad \longleftarrow v(s_t) + \alpha(G_t - v(s_t)) \tag{23}$$

$$\Rightarrow \quad q(s_t,a_t) \longleftarrow q(s_t,a_t) + \alpha(G_t - q(s_t,a_t)) \tag{24}$$

for the Monte Carlo control and

$$v(s_t) \quad \longleftarrow v(s_t) + \alpha(R_t + \gamma v(s_{t+1}) - v(s_t)) \tag{25}$$

$$\Rightarrow \quad q(s_t,a_t) \longleftarrow q(\underbrace{s_t}_{S}, \underbrace{a_t}_{A}) + \alpha(\underbrace{R_t}_{R} + \gamma q(\underbrace{s_{t+1}}_{S}, \underbrace{a_{t+1}}_{A}) - q(s_t,a_t)) \tag{26}$$

for Temporal Difference control, which is commonly called SARSA if used for the control task.

The policy improvements are done in both cases ϵ-greedily w.r.t. the action-value-function while reducing the probability to pick a random function with every sequence k

$$\epsilon \longleftarrow \frac{1}{k}, \qquad \pi \longleftarrow \epsilon - \text{greedy}(q) \tag{27}$$

4 Function Approximation

By applying the the methods discussed above, we need to store values of the value functions for every state in a look up table. This may work well for small problems, but becomes too slow and unfeasible very quickly due to memory reasons. E.g. for a game of go the agent would have to calculate and store values for 10^{170} states. Furthermore we have no approach to solve problems with a continuous state space, e.g. flying a helicopter or driving a car. With the methods discussed above we would need to try finding a clever way to reduce or generalize the state space, e.g. defining states as intervals of certain parameters when dealing with continuous state spaces. Unfortunately reducing the state space is not always possible.

In practice there is another approach which works pretty well on big problems. Instead of storing the values of the value functions for every state, we can try to approximate the value function, which would allow us to generalize from the experienced states we visit to every possible state we may encounter, without wasting memory.

4.1 State-Value-Function Approximation

To approximate the value function we first need a state representation which can be obtained by a *feature vector* $x(s)$

$$x(s) = (x_1(s), \ldots, x_n(s))^T \tag{28}$$

E.g. if the agent needs to navigate through space without hitting any obstacles, the entries of the feature vector could be the distances to the nearest obstacle for every direction and the distances to the target.
Next we can begin to get the representation of the state-value-function with any function approximation we like e.g. with a linear combination of features or a neural network. To illustrate the idea we will look at the linear combination of features, s.t. our approximated function becomes

$$\hat{v}(s, w) = x(s)^T w \tag{29}$$

with weights w. We can now minimize the mean squared error (MSE) between our predicted values and the true values we experienced

$$\mathcal{L}(w) = \mathbb{E}_\pi[(v_\pi(s) - \hat{v}_\pi(s,w))^2] \tag{30}$$

by gradient descent. Therefore we need to take the gradient of the MSE w.r.t. the weights w

$$\nabla_w \mathcal{L}(w) = -2\mathbb{E}_\pi[v_\pi(s) - \hat{v}_\pi(s,w)] \underbrace{\nabla_w \hat{v}(s,w)}_{=x(s)} \tag{31}$$

hence we get for the weight update

$$\Delta w = -\frac{1}{2}\alpha \nabla_w \mathcal{L}(w) \tag{32}$$

$$= \alpha \mathbb{E}_\pi[v_\pi(s) - \hat{v}_\pi(s,w)]x(s) \tag{33}$$

or if we use stochastic gradient descent to speed up the learning process

$$\Delta w = \alpha(v_\pi(s) - \hat{v}_\pi(s,w))x(s) \tag{34}$$

The last step is to adapt the weight update to the used algorithm by substituting the targeted true value $v_\pi(s)$ for the corresponding returns.
This means for Monte Carlo

$$v(s) \longrightarrow G_t \tag{35}$$

$$\Delta w = \alpha(G_t - \hat{v}_\pi(s,w))x(s) \tag{36}$$

and for Temporal Difference

$$v(s_t) \longrightarrow R_t + \gamma \hat{v}(s_{t+1}, w) \tag{37}$$

$$\alpha(R_t + \gamma \hat{v}(s_{t+1}, w) - \hat{v}_\pi(s_t, w))x(s_t) \tag{38}$$

4.2 Action-Value-Function-Approximation

To be able to solve the control task the same can be done for the action-value-function, with the slight difference that the feature vector in this case needs to depend not only on the state, but also on the action

$$x(s,a) = (x_1(s,a), \ldots, x_n(s,a))^T \tag{39}$$

the approximated action-value-function is then

$$\hat{q}(s,a,w) = x(s,a)^T w \tag{40}$$

Minimizing the MSE yields the weight updates with stochastic gradient descent

$$\Delta w = \alpha(q_\pi(s,a) - \hat{q}_\pi(s,a,w))x(s,a) \tag{41}$$

Analogue to the previous section we now need to substitute the targeted true value $q_\pi(s, a)$ for the corresponding returns which gives us the update rules for Monte Carlo

$$q(s, a) \longrightarrow G_t \tag{42}$$
$$\Delta w = \alpha(G_t - \hat{q}_\pi(s, a, w))x(s, a) \tag{43}$$

and Temporal Difference

$$q(s_t, a_t) \longrightarrow R_t + \gamma\hat{q}(s_{t+1}, a_{t+1}, w) \tag{44}$$
$$\alpha(R_t + \gamma\hat{q}(s_{t+1}, a_{t+1}, w) - \hat{q}_\pi(s_t, a_t, w))x(s_t, a_t) \tag{45}$$

5 Conclusion

In general reinforcement learning is a very handy addition to the commonly used machine learning techniques and has a strong stance in a certain kind of problems like controlling the movement of machines of beating humans in most kind of video games. As the title suggests, this report should have given a short introduction into the topic and does not claim all-encompassing, since there are still plenty interesting topics which ware not discussed e.g. experience replay or how to properly deal with non-stationary environments. Even though reinforcement learning methods might look promising they should not be applied to any kind of problem thoughtlessly, since in general they need much longer training as other machine learning techniques, if applicable. Reinforcement learning should rather be used as deus ex machina if all other methods fail.

References

[1] David Silver *Lecture 1: Introduction To Reinforcement Learning, p.17* `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`

[2] David Silver *Lecture 3: Planning by Dynamic Programming, p.13* `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`

[3] David Silver *UCL Course on RL, Lectures 1-7, slides and video lectures* `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`