6 Reinforcement Learning

As discussed above, a basic form of supervised learning is function approximation, relating input vectors to output vectors, or, more generally, finding density functions $p(\mathbf{y}, \mathbf{x})$ from examples $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$. However, in many applications we do not have this kind of teacher that tells us exactly at any time the appropriate response for a specific input. Rather, feedback from a teacher is often delayed in time and also often given just in the form of general feedback such as 'good' or 'bad' instead of detailed explanations what the learner should have done. That is, we are often interested in learning a sequence of appropriate actions and from this to predicting an expectation of future situations from the history of past events. In this section we will therefore talk about this general from of learning which is somewhat in-between supervised and unsupervised learning. More formally, let us now consider learning a temporal density function

$$p(\mathbf{y}(t+1)|\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, ...\mathbf{x}^{(1)}). \tag{6.1}$$

We encountered such models already in the form of specific temporal Bayesian networks such as a Hidden Markov Process. We will now discuss this issue further within the realm of **reinforcement learning** or **learning from reward**. While we consider here mainly the prediction of a scalar utility function, most of this discussion can be applied directly to **generalized reinforcement learning** (**?**).

6.1 Markov Decision Process

Reinforcement learning can be best illustrated in a Markovian world. As discussed before, such a world[3] is characterized by transition probabilities between states, $T(s'|s, a)$, that only depend on the current state $s \in S$ and the action $a \in A$ taken from this state. We now consider feedback from the environment in the form of reward $r(s)$ and ask what actions should be taken in each state to maximize future reward. More formally, we define the **value function** or **utility function**,

$$Q^{\pi}(s, a) = E\{r(s) + \gamma r(s_1) + \gamma^2 r(s_2) + \gamma^3 r(s_3) + ...\}_{\pi}, \tag{6.2}$$

as the expected future payoff (reward) when being in state s and then at s_1, s_2, etc. We introduced here the discount factor $0 \leq \gamma < 1$ so that we value immediate reward more than later reward. This is a common treatment to keep the expected value finite. An alternative scheme would be to consider only finite action sequences. The policy $\pi(a|s)$ describes what action to take in each state. In accordance with our general

[3]Most often we talk about Markov models where the model is a simplification or abstraction of a real world. However, here we discuss a specific toy world in which state transitions fulfill the Markov condition.

probabilistic world view, we can consider most generally probabilistic policies, that is, we want to know with which probability an action should be chosen. If the policies are deterministic, then taking a specific action is determined by the policy, and the value function is then often written as $V^{\pi}(s)$.[4] Our goal is to find the optimal policy, which is the policy that maximizes the expected future payoff,

$$\pi^*(a|s) = \arg\max_{\pi} Q^{\pi}(s,a), \tag{6.3}$$

where the argmax function picks the policy for which Q is maximal. Such a setting is called a **Markov Decision Process (MDP)**.

MDPs have been studied since the mid 1950s, and Richard Bellman noted that it is possible to estimate the value function for each policy π from a self-consistent equation now called **Bellman equation**. He called the corresponding algorithm **dynamic programming**. Specifically, we can separate the expected value of the immediate reward from the expected value of the reward from visiting subsequent states,

$$Q^{\pi}(s,a) = E\{r(s)\}_{\pi} + \gamma E\{r(s_1) + \gamma r(s_2) + \gamma^2 r(s_3) + ...\}_{\pi}. \tag{6.4}$$

The second expected value on the right hand side is that of the value function for state s_1. However, state s_1 is related to state s since state s_1 is the state that can be reached with a certain probability from s when taking action a_1 according to policy π (e.g., $s_1 = s + a_1$, or more generally $s_n = s_{n-1} + a_n$). This transition probability is encapsulated in the matrix $T(s'|s,a)$, and we can incorporate this knowledge into the equation by writing

$$Q^{\pi}(s,a) = r(s) + \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi(a'|s') E\{r(s') + \gamma R(s'_1) + \gamma^2 R(s'_2) + ...\}_{\pi}, \tag{6.5}$$

where s'_1 is the next state after state s', etc. Thus, the expression on the right is the state-value-function of state s'. If we substitute the corresponding expression of equation (6.2) into the above formula, we get the **Bellman equation** for a specific policy, namely

$$Q^{\pi}(s,a) = r(s) + \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi(a'|s') Q^{\pi}(s',a'). \tag{6.6}$$

We will discuss below an example with deterministic policies. Since in this cases an action a is uniquely specified by the policy, the value function $Q^{\pi}(s,a)$ reduces to $V^{\pi}(s)$. In this case the equation simplifies to[5]

$$V^{\pi}(s) = r(s) + \gamma \sum_{s'} T(s'|s,a) V^{\pi}(s'). \tag{6.7}$$

The Bellman equation is a set of N linear equations in an environment with N states, one equation for each unknown value function of each state. Given that the environment

[4] $V^{\pi}(s)$ is usually called the state value function and $Q^{\pi}(s,a)$ the state-action value function. However, note that the value depends in both cases on the states and the actions taken.

[5] This formulation of the Bellman equation for the MDP (Alpaydin 2010; Thrun et al. 2006; Russel & Norvig 2010) is slightly different to the formulation of Sutton and Barto in (Sutton & Barto 1988) since the latter used a slightly different definition of the value function as the cumulative reward from the next state only and not the current state. The corresponding Bellman equation is then $V^{\pi}(s) = \sum_{s'} T(s'|s,a)(r(s') + \gamma V^{\pi}(s'))$. This is just a matter of when we consider the prediction, just before getting the current reward of after taking the next step.

is known, that is, knowing functions r and T, we can use well known methods from linear algebra to solve for $V^\pi(s)$. This can be formulated compactly with Matrix notation,

$$\mathbf{r} = (1 - \gamma \mathbf{T})\mathbf{V}^\pi, \tag{6.8}$$

where $\mathbf{r}$ is the reward vector, 1 is the unit diagonal matrix, and $\mathbf{T}$ is the transition matrix. To solve this equation we have to invert a matrix and multiply this with the reward values,

$$\mathbf{V}^\pi = (1 - \gamma \mathbf{T})^{-1}\mathbf{r}^t, \tag{6.9}$$

where $\mathbf{r}^t$ is the transpose of $\mathbf{r}$. It is also common to use the Bellman equation directly and calculate a state-value-function iteratively for each policy. We can start with a guess $\mathbf{V}$ for the value of each state, and calculating from this a better estimate

$$\mathbf{V} \leftarrow \mathbf{r} + \gamma \mathbf{T}\mathbf{V} \tag{6.10}$$

until this process converges. This then gives us a value function for a specific policy. To find the best policy, the one that maximizes the expected payoff, we have to loop through different policies to find the maximal value function. This can be done in different ways. Most commonly it is to use **Policy iteration**, which starts with a guess policy, iterates a few times the value function for this policy, and then chooses a new policy that maximizes this approximate value function. This process is repeated until convergence.

It is also possible to derive a version of **Bellman's equation for the optimal value function** itself,

$$V^*(s) = r(s) + \max_a \gamma \sum_{s'} T(s'|s,a)V^*(s'). \tag{6.11}$$

The max function is a bit more difficult to implement in the analytic solution, but we can again easily use an iterative method to solve for this optimal value function. This algorithm is called **Value Iteration**. The **optimal policy** can always be calculated from the optimal value function,

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s'|s,a)V^*(s'). \tag{6.12}$$

A policy tells an agents what action should be chosen, and the optimal policy is hence related to optimal control as long as the reward reflects the desired performance.

The previously discussed policy iteration has some advantages over value iterations. In value iteration we have to try out all possible actions when evaluating the value function, and this can be time consuming when there are many possible actions. In policy iteration, we choose a specific policy, although we have then to iterate over consecutive policies. In practice it turns out that policy iteration often converges fairly rapidly so that it becomes a practical method. However, value iteration is a little bit easier and has more similarities to the algorithms discussed below that are also applicable to situations where we do not know the environment a priori.

6.2 Temporal Difference learning

In dynamic programming, the agent goes repeatedly to every possible state in the system. This can be time consuming and only works if we have complete knowledge of the system. In this case we we do not even have to perform the actions physically and can instead sit and calculate the solution in a planning phase. However, we might not know the rewards given in different states, nor the transition probabilities, etc. One approach would be to estimate these quantities from interacting with the environment before using dynamic programming. The following methods are more direct estimations of the state value function that determines optimal actions. These online methods assume that we still know exactly in which state the agent is. We will consider partially observable situations later in which the agent might not even be sure in which state it is in.

A general strategy for estimating the value of states is to act in the environment and thereby to sample reward. Such a sampling should be done with some stochasticity to ensure sufficient exploration of the states, and such methods are generally called **Monte Carlo methods**. Monte Carlo methods can be combined with the bootstrapping ideas of dynamic programming. The resulting algorithms are called **temporal difference (TD) learning** since they rely on the difference between expected reward and actual reward.

We start again by estimating the value function for a specific policy before moving to schemas for estimating the optimal policy. Bellman's equations require the estimation of future reward

$$\sum_{s'} T(s'|s,a)V^{\pi}(s') \approx V^{\pi}(s'). \tag{6.13}$$

In this equation we introduced an approximation of this sum by the value of the state that is reached by one Monte Carlo step. In other words, we replace the total sum that we could build when knowing the environment with a sampling step. While this approach is only an estimation, the idea is that this will still result in an improvement of the estimation of the value function, and that other trials have the possibility to evaluate other states that have not been reached in this trial. The value function should then be updated carefully, by considering the new estimate only incrementally,

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha\{r(s) + \gamma V^{\pi}(s') - V^{\pi}(s)\}. \tag{6.14}$$

This is called **temporal difference** or **TD learning**. The constant α is a learning rate and should be fairly small. This policy evaluation can then be combined with policy iteration as discussed already in the section on dynamic programming.

An example program is given in Table 6.1. The state space in this example consists of a chain of 10 states as shown on the left in Fig.6.1. The 10th state is rewarded with $r = 1$, while the first state receives a large negative reward, $r = -1$. The intermediate states receive a small negative reward to account for movement costs. The estimated value function after 10 policy iteration is shown on the right. This curve represents the mean of 100 such simulations, and the standard deviation is shown as errorbar.

One of the greatest challenges in this approach is the potential conflict between taking a step that provides a fairly large expected reward and exploring an unknown

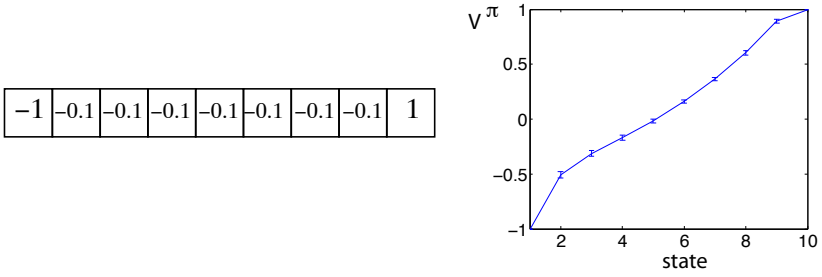

Fig. 6.1 Example of using TD learning on a chain of rewarded states. The right graph shows the estimated value function over 100 runs. The errorbars depict the standard deviations.

territory to potentially find larger rewards. This **exploration-exploitation dilemma** will now be addressed with stochastic policies and will thus return to the notation of the state-action value function.[6] To include exploration in the algorithm, we include some randomness in the policy. For example, we could follow most of the time the **greedy policy** and only choose another possible action in a small number ϵ of times. This probabilistic policy is called the ϵ**-greedy policy**,

$$\pi(a = \arg\max_a Q(s,a)) = 1 - \epsilon. \tag{6.15}$$

This policy is choosing the policy with the highest expected payoff most of the time while treating all other actions the same. A more graded approach employs the **softmax policy** which choses each action proportional to a Boltzmann distribution

$$\pi(a|s) = \frac{e^{\frac{1}{T}Q(s,a)}}{\sum_{a'} e^{\frac{1}{T}Q(s,a')}}. \tag{6.16}$$

This policy choses most often the one with highest expected reward, followed by the second highest, etc, and the temperature parameter T sets the relative probability of these choices. Temporal difference learning for the optimal value function with stochastic policies is

$$Q(s,a) \leftarrow Q(s,a) + \alpha\{r(s) + \gamma Q(s',a') - Q(s,a)\}, \tag{6.17}$$

where the actions a' is the action chosen according to the policy. This **on-policy TD algorithm** is called **Sarsa** for state-action-reward-state-action (Sutton & Barto 1988). A slightly different approach is using only the action to the maximal valued state for the value function update while still exploring the state space through the stochastic policy,

$$Q(s,a) \leftarrow Q(s,a) + \alpha\{r(s) + \max_{a'} \gamma Q(s',a') - Q(s,a)\}. \tag{6.18}$$

Such an **off-policy TD algorithm** is called **Q-leaning**. These algorithms have been instrumental in the success of reinforcement learning in many engineering applications.

[6] We will drop the star in the notation for the optimal value function since we will now only consider optimal value functions.

Table 6.1 TD learning with a chain state space

```
% Chain example: Policy iteration with TD learning
clear; N=10; P=0.8; gamma=0.9; % parameters
r=zeros(1,N)-0.1; r(1)=-1; r(N)=1; % reward function

% transition probabilities; 1 (going left) and 2 (going right)
T=zeros(N,N,2); %T(1,1,:)=1; T(N,N,:)=1; %Absorbing end states
for i=2:N-1;
    T(i,i-1,1)=P; T(i,i+1,1)=1-P; T(i,i-1,2)=1-P; T(i,i+1,2)=P;
end

% initially random start policy and value function
policy=floor(2*rand(1,N))+1; Vpi=rand(N,1);

for iter=1:10 %policy iteration
    Vpi(1)=0; Vpi(N)=0; %absorbing end states
    %Transition matrix of choosen action
    for s=2:N-1;
        Tpi(s,s-1)=T(s,s-1,policy(s));
        Tpi(s,s+1)=T(s,s+1,policy(s));
    end
    % Estimate V for this policy using TD learning
    for i=1:100 %loop over episodes
        s=floor(9*rand)+1; %initial random position
        while s>1 && s<N
            a=policy(s); %choose action
            s1=s+2*(a-1)-1; %calculate next state
            if rand>Tpi(s,s1); s1=s-2*(a-1)+1; end %random execution
            Vpi(s)=Vpi(s)+0.01*(r(s1)+gamma*Vpi(s1)-Vpi(s)); %update
            s=s1;
        end
    end
    Vpi(1)=r(1); Vpi(N)=r(N);
    % Updating policy
    policy(1)=0; policy(N)=0; %absorbing states
    for s=2:N-1
        [tmp,policy(s)] = max([Vpi(s-1),Vpi(s+1)]);
    end
end
```

6.3 Function approximation and TD(λ)

There are several challenges in reinforcement learning remaining. Specifically, the large number of states in real world applications makes these algorithms unpractical. This was already noted by Richard Bellman who coined the phrase **curse of dimensionally**. Indeed, we have only considered so far discrete state spaces, while many applications are in a continuous state space. While discretizing a continuous state space is a common approach, increasing the resolution of the discretization will increase the number of states exponentially. Another major problem in practice is that the environment is not fully or reliably observable. Thus, we might not even know exactly in which state the agent is when considering the value update. A common approach to the partially observable Markov decision process (POMDP) is the introduction of a probability map. In the update of the Bellman equation we need then to consider all possible states that can be reached from the current state, which will typically increase

the number of calculations even further. We will thus not follow this approach here and consider instead the use of function approximators to overcome these problems.

The idea behind the following method is to make a hypothesis of the relation between sensor data and expected values in form of a parameterized function as in supervised learning,[7]

$$V_t = V(\mathbf{x}_t) \approx V(\mathbf{x}_t; \theta), \tag{6.19}$$

and estimate the parameters by maximum likelihood as before. We used thereby a time index to distinguish the sequences of states. In principle, one could build very specific temporal Bayesian models for specific problems as discussed above, but I will outline here again the use of general learning machines in this circumstance. In particular, let us consider adjusting the weights of a neural network using gradient-descent methods on a mean square error (MSE) function

$$\Delta\theta = \alpha \sum_{t=1}^{m} (r - V_t) \frac{\partial V_t}{\partial \theta}. \tag{6.20}$$

We considered here the total change of the weights for a whole episode of m time steps by summing the errors for each time step. One specific difference of this situation to the supervised learning examples before is that the reward is only received after several time steps in the future at the end of an episode. One possible approach for this situation is to keep a history of our predictions and make the changes for the whole episode only after the reward is received at the end of the episode. Another approach is to make incremental (online) updates by following the approach of temporal difference learning and replacing the supervision signal for a particular time step by the prediction of the value of the next time step. Specifically, we can write the difference between the reward and the prediction at the end of the sequence at time t when reward is received, $V_{t+1} = r$, as

$$r - V_t = \sum_{k=t}^{m} (V_{k+1} - V_k) \tag{6.21}$$

since the intermediate terms chancel out. Using this in equation (6.20) gives

$$\Delta\theta = \alpha \sum_{t}^{m} \sum_{k=t}^{m} (V_{k+1} - V_k) \frac{\partial V_t}{\partial \theta} \tag{6.22}$$

$$= \alpha \sum_{t=1}^{m} (V_{t+1} - V_t) \sum_{k=1}^{t} \frac{\partial V_k}{\partial \theta}, \tag{6.23}$$

which can be verified by writing out the sums and reordering the terms. Of course, this is just rewriting the original equation 6.20. We still have to keep a memory of all the gradients from the previous time steps, or at least a running sum of these gradients.

While the rules (6.20) and (6.23) are equivalent, we also introduce here some modified rules suggested by Richard Sutton (Sutton 1988). In particular, we can weight

[7] The same function name is used on both sides of the equation but these are distinguished by inclusion of parameters. The value functions below refer all to the parametric model, which should be clear from the context.

recent gradients more heavily than gradients in the more remote past by introducing a decay factor $0 \leq \lambda \leq 1$. The rule above correspond to $\lambda = 1$ and is thus called the **TD(1) rule**. The more general **TD(λ) rule** is given by

$$\Delta_t \theta = \alpha(V_{t+1} - V_t) \sum_{k=1}^{t} \lambda^{t-k} \frac{\partial V_k}{\partial \theta}. \tag{6.24}$$

It is interesting to look at the extreme of $\lambda = 0$. The **TD(0) rule** is given by

$$\Delta_t \theta = \alpha(V_{t+1} - V_t) \frac{\partial V_t}{\partial \theta}. \tag{6.25}$$

While this rule gives gives in principle different results with respect to the original supervised learning problem described by TD(1), it has the advantage that it is local in time, does not require any memory, and often still works very well. TD(λ) algorithm can be implemented with a multilayer perceptron when back-propagating the error term to hidden layers. A generalization to stochastic networks has been made in the free-energy formalism (Sallans & Hinton 2004).