Reinforcement Learning

1

1.1 Learning from reward and the credit assignment problem

In supervised learning as outlined in Chapter 6, we assumed that a teacher supplies the exact desired state of each output node in the network. Such a teacher has to supply very detailed information, which seems unrealistic in common learning situations. More realistic are teaching procedures with limited feedback such as an answer being 'good' or 'wrong'. When training animals, a correct response is often acknowledged with a food reward. An absence of a food reward can indicate a 'false response' to the animal. Another challenge for a learner is that feedback might only be provided at the end of a series of actions, and we need to solve how to reward components of actions that have contributed to successful outcomes. This section discusses such learning systems.

1.1.1 Classical conditioning and the reinforcement learning problem

Learning with reward signals has been studied by psychologists for many years under the term *conditioning*. An example of classical conditioning in animal learning is shown in Fig. 1.1. In the illustrated experiment, we place a rodent in a T-maze and supply food of different sizes when the rodent goes to the end of each horizontal arm of the T-maze.

The rodent might wander around, and let us assume that it finds the smaller food reward at the end of the left arm of the T-maze. It is then likely that the rodent will turn left in subsequent trials to receive food reward. Thus the animal learned that the action of taking a left turn and going to the end of the arm is associated with food reward. The learning challenge here is a form of a *credit assignment* problem. We can think of this in terms of a *temporal credit assignment* problem in which the animal has to associate the food at a later state with the action at a previous state. If we have a distributed system implementing possible action such as a neural network, then we can also view this as a *spatial credit assignment*, which means in this context the question of which part of the action system to credit for the receivership of the food reward.

We are discussing here reinforcement learning directly in the context of an action system, which relates to *instrumental conditioning* in the animal learning literature. We will discuss below that this usually requires a action system as well as a value system. A slightly simpler form of reinforcement learning

classical conditioning that formally mainly evaluates a value system. For example, we could ring a bell and then present food to the rodent. In our terms this is another interesting example of a temporal assignment problem, that of associating the sound of the bell with food reward, and a situation where we can concentrate on the value system. We will see later that this distinction is not entirely necessary when using the language developed here.

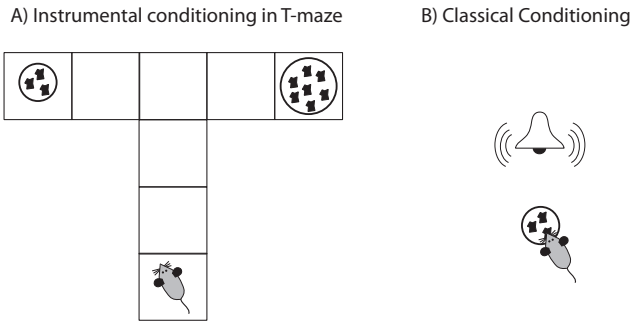

Fig. 1.1 Example of instrumental and classical conditioning. A) A rodent has to learn to transverse the maze and make a decision at the junction in which direction to go. Such as decision problem, which necessitates the action of an actor, is called instrumental conditioning in the animal learning literature. B) A slightly simpler setting is that of classical conditioning where a subject is required to associate the ringing of a bell with reward. Such a setting is focusing on a value system in our reinforcement learning setting.

1.1.2 Formalization of the problem setting: The Markov Decision Process

We consider an agent that in each time interval k is in a specific state x_k from which it can take an action u_k. This action specifies a transition to a new state, and this transition is specified by a function τ as

$$\textbf{Transition function:} \quad x_{k+1} = \tau(x_k, u_k). \tag{1.1}$$

The transition function only depend on the previous state and the intended action from this state, and this is called the *Markov condition*. A Markov decision process is a good place to start as it often applies with the right definition of state and also illustrates best the principle ideas. In contrast, a non-Markov condition would be the case in which the next state depends on a series of previous states and actions, and our agent would then need a memory to make optimal decisions.

The environment or a teacher provides reward according to a function ρ,

$$\textbf{Reward function:} \quad r_{k+1} = \rho(x_k, u_k). \tag{1.2}$$

And finally, the agent has a control *policy* that specifies which action u to take from each state,

$$\textbf{Policy:} \quad u_k = \pi(x_k) \tag{1.3}$$

In the context of a mobile agent, the action u_k that the agent should take at time k, the time when the agent is in state x_k, is a motor command. For example, a mussel that should move the arm should be activated with a certain nerve pulse, or a robot that should turn is activated by a certain motor that should run for a certain time with a certain speed.

1.1.3 Return and Value functions

The goal of the agent is to maximize the *total expected reward* in the future from every initial state. This quantity is also called *return*, especially in economics. Here we discuss the exemplary case in which the agent values immediate reward more than reward far in the future. To capture this we define the infinite discounted return from state $x = x_0$ following policy π,

$$\textbf{V-function:} \quad V^\pi(x) := \sum_{k=0}^{\infty} \gamma^k \rho(x_k, \pi(x_k)) \tag{1.4}$$

Another common choice is the average reward in a finite horizon case, but we focus on the former to keep the number of cases under control. A more detailed value function that evaluates the value of each action taken from a state is the infinite discounted return, which is taking action $u = u_0$ from state $x = x_0$ and then follows policy π, that is, $u_1 = \pi(x_1) \to x_2 = \tau(x_1, u_1) \to u_2 = \pi(x_2)$, etc,

$$\begin{aligned} \textbf{Q-function:} \quad Q^\pi(x,u) &:= \sum_{k=0}^{\infty} \gamma^k \rho(x_k, u_k) \\ &= \rho(x,u) + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} \rho(x_k, \pi(x_k)) \\ &= \rho(x,u) + \gamma V^\pi(\tau(x,u)). \end{aligned} \tag{1.5}$$

The goal of RL is to find the policy which maximized the return. If the agents knows the

$$\textbf{Optimal Value function:} \quad V^*(x) = \max_u Q^*(x,u), \tag{1.6}$$

then the optimal policy is simply given by taking the action that leads to the biggest expected return, namely

$$\textbf{Optimal policy:} \quad u^*(x) = \arg\max_u Q^*(x,u). \tag{1.7}$$

Thus, the optimal value function and the optimal policy are closely related. Nevertheless, the methods below can be distinguish somewhat in terms of how the algorithms chose actions during learning. We will start with methods that focus on finding the value function, which are several algorithms that can be put under the heading *value-search*. Corresponding agents, or part of the corresponding algorithm, are commonly called a *critic*. A agent that learns by focusing on *policy-search* is called an *actor*. Later we argue that combining these in an *actor-critic scheme* is a viable method in practice and likely used in the brain.

1.2 Model-based Reinforcement learning

In this section we assume that the agent has a *model* of the environment and its behaviour by knowing the reward function $\rho(x,u)$ and the transfer functions $\tau(x,u)$. The knowledge of these functions, or a model thereof, is required for *model-based RL*, which is also called *dynamic programming*.

1.2.1 The basic Bellman equation

The key to learning the value functions is the realization that the right hand side of eq.1.5 can be written in terms of the Q-function itself, namely

$$\begin{aligned} Q^\pi(x,u) &= \rho(x,u) + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} \rho(x_k, \pi(x_k)) \\ &= \rho(x,u) + \gamma \left[\rho(\tau(x,u), \pi(\tau(x,u))) + \gamma \sum_{k=2}^{\infty} \gamma^{k-2} \rho(x_k, \pi(x_k)) \right]. \end{aligned}$$

The term in the square bracket is equal to the value function of the state that is reached after the transition $\tau(x,u)$

$$Q^\pi(\tau(x,u), \pi(\tau(x,u))) = V^\pi(\tau(x,u)). \tag{1.8}$$

The Q-function and the V-function are here equivalent since we are following the policy in these steps. Using this fact in the equation above we get the

$$\pi \textbf{ Bellman equation: } \quad Q^\pi(x,u) = \rho(x,u) + \gamma Q^\pi(\tau(x,u), \pi(\tau(x,u))). \tag{1.9}$$

If we combine this with known dynamic equations in the continuous time domain, then this becomes the Hamilton-Jacobi-Bellman equation. often encountered in engineering.

As stated above, we assume here the reward function $\rho(x,u)$ and the transition functions $\tau(x,u)$ are known. At this point the agent follows a specified policy $\pi(x)$. Let us further assume that we have X states and U possible actions in each state. We have thus $X \times U$ unknown quantities $Q^\pi(x,u)$ which are governed by the Bellman equation above. More precisely, the Bellman equations 1.9 are $X \times U$ coupled linear equations of the unknowns $Q^\pi(x,u)$. It is then convenient to write this equation system with vectors

$$\mathbf{Q}^\pi = \mathbf{R} + \gamma \mathbf{T}^\pi \mathbf{Q}^\pi \tag{1.10}$$

Where T^π is an appropriate transition matrix which depends on the policy. This equation can also be written as

$$\mathbf{R} = (\mathbb{1} - \gamma \mathbf{T}^\pi) \mathbf{Q}^\pi, \tag{1.11}$$

where $\mathbb{1}$ is the identity matrix. This equation has the solution

$$\mathbf{Q}^\pi = (\mathbb{1} - \gamma \mathbf{T}^\pi)^{-1} \mathbf{R}(?) \tag{1.12}$$

if the inverse exists. In other words, as long as the agent knows the reward function and the transition function, it can calculate the value function for a specific policy without even taking a single step.

Alternatively we can solve the Bellman equation with an iterative algorithm. We thereby starts with a guess of the Q-function, let's call this Q_i^π, and improve it by calculating the right hand side of the Bellman equation,

$$\textbf{Dynamic Programming: } \quad Q_{i+1}^\pi(x,u) \leftarrow \rho(x,u) + \gamma Q_i^\pi(\tau(x,u), \pi(\tau(x,u))). \tag{1.13}$$

The fixed-point of this equation, that is, the values that does not change with these iterations, are the desired values of Q^π. Another way of thinking about this algorithm is that the Bellman equality is only true for the correct Q^π values. For our guess, the difference between the left- and right-hand side is not zero, but we are minimizing this with the iterative procedure above.

1.2.2 Programming example

To illustrate the different reinforcement schemes discussed in this chapter, we will apply these to the example of the T-maze outlined in the introduction. To keep the programs minimal and clean, we will concentrate on the upper linear part for the maze as illustrated in Fig. 1.2. We labeled the states x of the maze as 1 to 5. A reward of value 1 is provided in state 1 and a reward of 2 is provided in state 5. The discrete Q-function has 10 values corresponding to each possible action in each state. In states 2,3, and 4 these are the actions of move *left* or move *right*. The states with the reward, states 1 and 5, are terminal states and the agent would stay in these states. We coded this with two *stay* actions to keep some consistency in the representation.

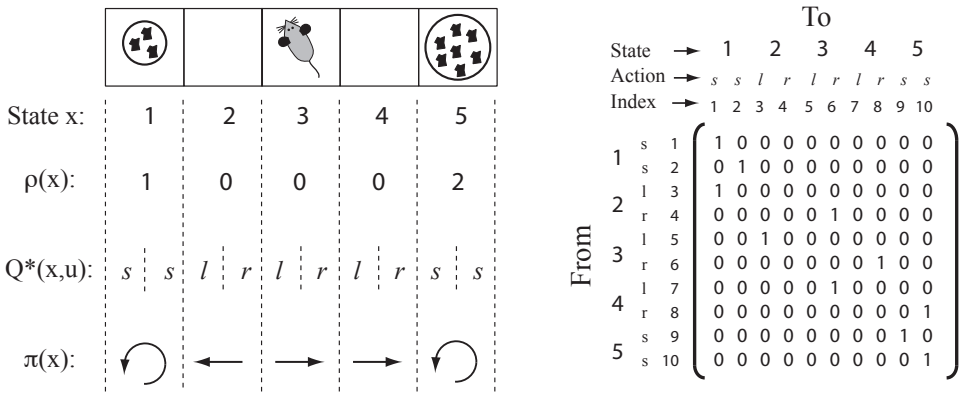

Fig. 1.2 Example experiment with the simplified T-maze where we concentrate on the more interesting horizontal portion of the maze. The right hand side shows the corresponding transition matrix for the optimal policy.

We first set up some code that we need in all examples:

```
%% Reinforcement learning in a maze
clear; close all;

tau = @(x,u) x+heaviside(x-1.5).*heaviside(4.5-x)*u;
rho = @(x,u) (x==2&&u==-1)+2*(x==4&&u==1);
idx = @(u)   (u+3)/2';
gamma=0.5;
```

We provide here three inline functions for $\tau(x,u)$, $\rho(x,u)$, and a helper function that transforms the action representation $u \in \{-1,1\}$ to the corresponding indices idx $\in \{1,2\}$. We also define here the discount factor $\gamma = 0.5$.

To demonstrate how to solve the Bellman equation with linear algebra tools, we need to define the corresponding vectors and matrices as used in eq.**??**. We order therefore the quantities such as ρ and Q with ten indices. The first one correspond to $(x = 1, u = -1)$, the second to $(x = 1, u = 1)$, the third to $(x = 2, u = -1)$, etc. The reward vector can thus be coded as:

```
%% Analytic solution for optimal policy
i=0;
for x=1:5; for u=-1:2:1;
  i=i+1; R(i)=rho(x,u);
end; end;
```

The transition matrix depends on the policy, so we need to choose one. We chose the one specified on the left in Fig. 1.2 where the agent would move to the left in state $x = 2$ and to the right in states $x = 3$ and $x = 4$. This happens to be the optimal solution as we will show later so that this will also give us a solution for the optimal value function. We use this policy to construct the transition matrix by hand as shown on the right in Fig. 1.2. For example, if we are in state $x = 4$ and move to the left, $u = -1$, corresponding to the from-index=7, then we end up in state $x = 3$, from which the policy say go right, $u = 1$. This correspond to the to-index=6. Thus, the transition matrix should have an entry $T(7, 6) = 1$. Going through all the cases leaves us with

```
T=zeros(10,10);
T(1,1)=1; T(2,2)=1; T(3,1)=1; T(4,6)=1; T(5,3)=1;
T(6,8)=1; T(7,6)=1; T(8,10)=1; T(9,9)=1; T(10,10)=1;
```

With this we can solve the equations in Matlab simply by the `inv()` function.

```
Q=inv(eye(10)-gamma*T)*R';
Q=reshape(Q,2,5)
```

In the last line we have reshaped the Q-vector into a matrix with two lines, the first correspond to the values for moving left in the corresponding state, and the second correspond to moving right in the corresponding state. The result is

```
0    1.0000    0.5000    0.5000         0
0    0.5000    1.0000    2.0000         0
```

1.2.3 Policy iteration

The goal of RL is of course to find the policy which maximized the return. So far we have only a method to calculate the value for a given policy. However, we can start with an arbitrary policy and can use the corresponding value function to improve the policy by defining a new policy which is given by taken the actions from each state that gives us the best next return value,

$$\textbf{Policy iteration:} \pi(x) \leftarrow \arg\max_{\mathrm{u}} \mathrm{Q}^{\pi}(\mathrm{x}, \mathrm{u}). \tag{1.14}$$

For the new policy we can then calculate the corresponding Q-function and then use this Q-function to improve the policy again. Iterating over the policy gives us the

$$\textbf{Optimal policy:} \quad \pi^*(x). \tag{1.15}$$

The corresponding value function is Q^*. In the maze example we can see that the maximum in each column of the Q-matrix is the policy we started with. This is hence the optimal policy as we stated before.

The corresponding code for our maze example is

```
%% policy iteration
Q=zeros(5,2); pi=-ones(5,1);
for iter=1:3;
    for x=1:5; for u=-1:2:1
        Q(x,idx(u))=rho(x,u)+gamma*Q(tau(x,u),idx(pi(tau(x,u))));
    end; end
    for x=1:5;
        [~,uidx]=max([Q(x,1),Q(x,2)]); pi(x)=2*uidx-3;
    end
end
disp('Policy iteration'); disp(Q')
```

In this example we used only 3 iterations over the policies, which is the minimal for this example. In practice, we would check if the values actually change and would stop if not.

1.2.4 Bellman function for optimal policy and value iteration

Since we are foremost interested in the optimal policy, we could try to solve the Bellman equation right away for the optimal policy,

$$Q^*(x,u) = \rho(x,u) + \gamma Q^*(\tau(x,u), \pi^*(\tau(x,u))). \tag{1.16}$$

The problem is that this equation does now depend on the unknown π^*. However, we can check in each state all the actions and take the one which gives us the best return. This should be equivalent to the equation above in the optimal case. Hence we propose the

$$\textbf{Optimal Bellman equation:} \quad Q^*(x,u) = \rho(x,u) + \gamma \max_{u'} Q^*(\tau(x,u), u'). \tag{1.17}$$

We can solve this with dynamic programming when the transfer function and the reward functions are known,

$$\textbf{Q-iteration:} \quad Q^*_{i+1}(x,u) \leftarrow \rho(x,u) + \gamma \max_{u'} Q^*(\tau(x,u), u'). \tag{1.18}$$

The corresponding code for our maze example is

```
%% Q-iteration
Q=zeros(5,2);
for iter=1:10;
    for x=1:5; for u=-1:2:1
```

```
        Qnew(x,idx(u))=rho(x,u)+gamma*max(Q(tau(x,u),1),Q(tau(x,u),2));
    end; end
Q=Qnew;
end
disp('Q-iteration'); disp(Q')
```

In this example we again used 3 iterations which are sufficient to reach the correct values. In practice we would terminate the program if the changes are sufficiently small.

1.3 Model-free RL: Temporal difference method

Above we assumed a model of the environment by an explicit knowledge of the functions $\tau(x,u)$ and $\rho(x,u)$. We could use modelling techniques and some sampling strategies to learn these functions explicitly and then use model-based RL as described above to find the optimal policy. Instead we will here combine here the sampling by exploring the environment directly with reinforcement learning.

We will start again with a version for a specific policy by choosing a policy and estimate the Q-function for this policy. As in dynamic programming we want to minimize the difference between the left- and right-hand side of the Bellman equation. But we can not calculate the right hand side since we do not know the transition function and the reward function. However, we can just take a step according to our policy $u = \pi(x)$ and observe a reward r_{i+1} and the next state x_{i+1}. Now, since this is only one sample we should take this only with a small learning rate α into account to update the value function

$$\textbf{SARSA:}\quad Q_{i+1}(x_i,u_i) \leftarrow Q_i(x_i,u_i) + \alpha\left[(r_{i+1} + \gamma Q_i(x_{i+1},u_{i+1}) - Q_i(x_i,u_i)\right]. \tag{1.19}$$

The term in the square brackets on the right hand-side is called the *temporal difference*. Note that we are following here the policy, and the methods is therefore often labeled as *on-policy*. The next step is to use the estimate of the Q-function to improve policy. However, since we are anyhow mainly interested in the optimal policy we should improve the policy by taking the the steps that maximizes the payoff. However, one problem in this scheme is that we have to estimate the Q values by sampling so that we have to make sure to trade off *exploitation* with *exploration*. A common way to choose the policy in this scheme is the

$$\epsilon\textbf{-greedy policy:}\quad p\left(\arg\max_u Q(x,u)\right) = 1-\epsilon. \tag{1.20}$$

So, we are really evaluating the optimal policy that requires to make the exploration zero at the end, $\epsilon \to 0$.

The corresponding code for the SARSA is

```
%% SARSA
Q=zeros(5,2); alpha=0.1;  %starting state
for trial=1:1000
```

```
    x=3;
    for t=1:5;
        [~,u]=max([Q(x,1),Q(x,2)]);u=2*u-3;
        if rand()<0.1; u=-u; end %epsilon exploration
        Q(x,idx(u))=(1-alpha)*Q(x,idx(u))+alpha*(rho(x,u)+gamma*Q(tau(x,u),idx(u)));
        x=tau(x,u);
    end;
    error(trial)=sum(sum(abs(Q-Qnew)));
end
disp('SARSA'); disp(Q')
plot(error)
```

Alternatively we can evaluate the samples *off-policy*, that is, we can check again all possible actions from a state and update the value function with the maximal expected return,

$$\textbf{Q-learning:}\quad Q_{i+1}(x_i,u_i) \leftarrow Q_i(x_i,u_i)+\alpha\left[(r_{i+1}+\gamma \max_{u'} Q_i(x_{i+1},u') - Q_i(x_i,u_i)\right]. \tag{1.21}$$

This is also the right choice if we take the policy

$$\textbf{Optimal policy:}\quad u^*(x) = \max_{u'} Q(x,u') \tag{1.22}$$

The corresponding code for the Q learning is

```
%% Q-learning
Q=zeros(5,2); alpha=0.1;  %starting state
for trial=1:1000
    x=3;
    for t=1:5;
        [~,u]=max([Q(x,1),Q(x,2)]);u=2*u-3;
        if rand()<0.1; u=-u; end %epsilon exploration
        Q(x,idx(u))=(1-alpha)*Q(x,idx(u))+alpha*(rho(x,u)+gamma*max(Q(tau(x,u),1),Q(tau(x,u),2)));
        x=tau(x,u);
    end;
    error(trial)=sum(sum(abs(Q-Qnew)));
end
disp('Q-learning'); disp(Q')
plot(error)
```

1.3.1 Actor and policy search

In the previous algorithms we have basically focussed on finding a value function and the policy was determined by the value function as being the action that lead to the state with the largest return. The value function is then often described as a *critic*. Interestingly, this way of reinforcement learning is not always the best approach during a learning phase. For example, while learning the value function it can happen that the values of two states are similar, and a slight shift in the estimation can shift the policy dramatically. Such sudden shifts can destabilize a system which itself might depend on the actions of agents that make up the environment. We therefore now allow the agent to consider its own policy and study the gradual change of the policy. Such a component of the system is called the *actor*.

In the previous settings for the critic we considered maximizing the return for each possible starting state. In policy search it is more common to consider

policies that would work for every starting state and to maximize the slightly relaxed condition of maximizing the average return – or the expected return – over all possible starting position when following the policy. This objective function is given by

$$\textbf{Actor objective function:} \quad J(\pi) = E\{\sum_{k=0}^{\infty} \gamma^k \rho(x_k, \pi(x_k))\}_{x_0}, \qquad (1.23)$$

where the averaging is over all starting states x_0.

Minimizing this objective function can be done in various was. For example, we could use guided random search such as evolutionary algorithms to search for suitable policies. It is also common to parameterize possible policies and use a gradient-based optimization in the in the parameter space that maximizes the objective function.

1.4 Using function approximation with neural networks

At this point we have outlines the basic ideas behind the basic reinforcement learning algorithm, and we will no move to an important topic to scale these ideas to real world applications. In the previous method we used functions of the state and action variables such as the transition function $\tau(x, u)$. The general, the state and action variables could be continuous variables, though we used only small discrete sets in the maze example. For discrete values we can think of representing the functions as a table, and such methods are correspondingly called tabular methods. Many problems in the literature use a small number of states and actions, in particular in simplified psychophysical experiments. The general problem in reinforcement learning is that the necessary computations scale up quickly with the number of states, a difficulty that has been termed the "course of dimensionality" by Bellman himself. The principle idea in this section is to use function approximators to represent the functions, and it should be no surprise that we will specifically use neural networks for these function approximations.

1.4.1 Temporal delta rule

In order to introduce implementations of systems which learn from reward within neural architectures, we simplify the situation further with an example of classical conditioning in which the action of the agent is fixed. The purpose of the system introduced now is to predict future reward with this fixed policy, or in other words, to evaluate the state value function $V^\pi(x)$, from episodes in which some states are rewarded, possibly at a later time. Let's assume that this state value function, relating states to reward, is simple enough to be learned by a linear population node as discussed in Chapter 6 and illustrated in Fig. 1.3A. The input of the node represents a state at time t, $x(t)$. This state is represented by an input vector $r_i^{\text{in}}(x)$, which we can think of as reprinting cortical activity which encodes the internal representation of the state of an

agent. The output of the linear reward-prediction node is

$$V(x) = \sum_i w_i(t) r_i^{\text{in}}(x). \tag{1.24}$$

The rate values of the perceptron are time dependent in the sense that the agent changes states in each time step so that the input states x change with time. We also indicated explicitly the time dependence of the weight values, $w_i(t)$, since they will change over time as a function of reinforcement signals. Note that time here refers to behaviourally relevant time slots.

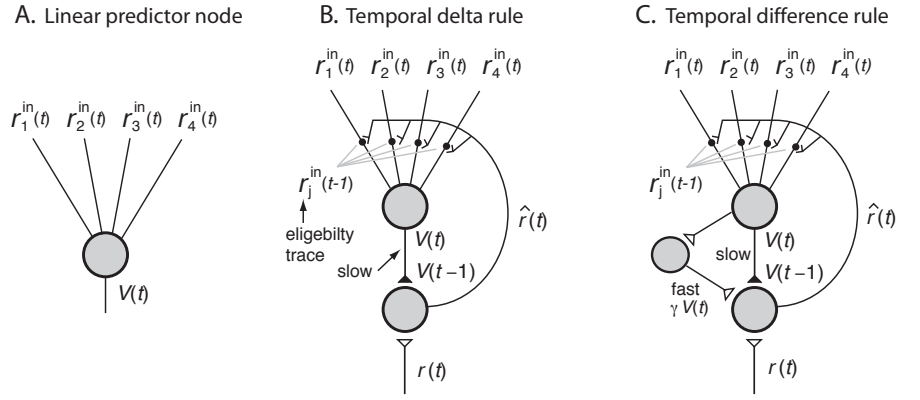

Fig. 1.3 Neural implementation of reward learning algorithms. (A) Linear predictor node. (B) Temporal delta rule with primary reward r. The output of the predictor node is indicated at time $t-1$ since the connection is considered to be slow. (C) Temporal difference rule with a fast side loop.

The agent is taking an action in response to the state according to the fixed policy, π, and this action results in a reward at the next time step, $r(t+1)$. We want to update the weights at this time of reward to reflect the new knowledge of reward at this time. We first consider the case in which the system should predict the reward for each state, which corresponds to a value function that treats the reward following each state independently. In this case, the actual reward can be used as the desired output, and we can use the delta rule discussed in Chapter 6to train the linear perceptron,

$$w_i^{t+1} = w_i^t + \alpha * \hat{r}(t) r_i^{\text{in}}(x). \tag{1.25}$$

The parameter α is a learning rate, and we introduce in this equation the *effective reinforcement signal*

$$\hat{r}(t) = r(t-1) - V(t-1). \tag{1.26}$$

The learning rule, eqn 1.25, is sometimes called a temporal delta rule because of the temporal difference between actual reward and predicted time return. Of course, since this is a one step prediction task, the predicted return is equal to the predicted reward. In essence this is just the regular perceptron learning rule where we considered different time slots for the prediction of the perceptron and the supervision signal to arrive. In this case, the system needs to remember the value $V(x^t)$ for one time step, and we indicate this memory in the model shown in Fig. 1.3B as slow output channel, or axional delay. In essence this could be any short term memory mechanisms. In addition, the system must

remember the input values ($r_i^{\text{in}}(x)$) for one time step because we attribute the cause for the reward to taking the fixed action from the previous state. This memory is often called an *eligibility trace* in the corresponding literature, in particular if we consider longer sequences until a reward is given, as discussed later. In the simplified case here with a one-step-ahead prediction tasks, the eligibility trace for the example discussed here is simply a short-term memory which can easily be implemented in neural tissue. The eligibility trace is shown in Fig. 1.3B as short-term memory at the synapses from the node that supplies the training signal.

The eligibility trace and the axional delay solves here the temporal credit assignment problem. The weights of the input channels to the node that learns the state value function depend on its delayed output and the primary reward (reinforcement) signal r. The primary reinforcement signal is assumed to have an excitatory effect on a node that mediates learning, whereas the prediction node has an inhibitory effect on the node which mediates learning. This *training node* therefore calculates the effective reinforcement signal of equation 1.26. This effective reinforcement signal has to be conveyed to all the synapses of the prediction node, where it has to be correlated with an eligibility trace to produce the appropriate weight change proposed in eqn 1.25. This model of reward learning is equivalent to the *Rescorla–Wagner theory* in classical conditioning. The model can produce one-step ahead predictions of a reward signal and corresponds to a particular choice of the state value function. The effective reinforcement signal is zero when the activity of the prediction node matches the primary reinforcement signal.

1.4.2 Temporal difference learning with a perceptron

Since we discussed so far only the one-step ahead prediction tasks, which is the generally setting to which the Rescorla–Wagner model is applied, this corresponds to a value function definition with $\gamma = 0$. The more general case with longer horizons and a finite discount factor follows the above discussion of the Bellman equations. That is, the optimal state value function V^* at time $t-1$ is defined as

$$\begin{aligned} V^*(t-1) &= r(t) + \gamma r(t+1) + \gamma^2 r(t+2) + ... \\ &= r(t) + \gamma[r(t+1) + \gamma r(t+2) + ...], \end{aligned} \tag{1.27}$$

where the right hand side can be expressed again as a state value function. This leads to the version of the Bellman exquation for the state-value function

$$r(t) + \gamma V^*(t) - V*(t-1) = 0. \tag{1.28}$$

Of course, this is only true if we have a perfect prediction. If we do not have a perfect prediction, then the equation does not hold. In this case we can again minimize the *temporal difference error*

$$\hat{r}(t) = r(t) + \gamma V^{\pi}(t) - V^{\pi}(t-1), \tag{1.29}$$

which is analog to the SARSA algorithm in this classical conditioning setting. This temporal difference should be used in the learning rule eqn 1.25 as effective reinforcement signal to update the weights. Compared with the temporal

delta rule ($\gamma = 0$), we only need the additional value $V^\pi(t)$. Fig. 1.3C shows a possible neuronal implementation of a temporal difference learning, which includes a fast side-loop with a decay factor that conveys the value $\gamma V^\pi(t)$ to the node that calculates the primary reinforcement signal.

1.4.3 TD(λ)

TD learning is solving credit assignment problem without long-term memory at the expense of repeated exploration. This is likely most clear in the maze example as the agent has to traverse the maze several times in order to 'back propagate' the reward signal to previous states. The other extreme is to keep a memory of all intermediate predictions of the reward predictor and then adjust all of them when reward is received. To do this we consider a perceptron that predicts values at a specific times in a sequence, $V_t(\mathbf{x}_t)$, from input $\mathbf{x}_t$ at time t,

$$V_t(\mathbf{x}_t; \mathbf{w}) \approx V_t(\mathbf{x}_t). \tag{1.30}$$

We can train this MLP with a gradient-descent methods on an objective function E. 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. The error at each time step is given by the square difference between predicted and actual reward. This correspond to the objective function

$$E = \sum_{t=1}^{m} (r - V_t)^2, \tag{1.31}$$

for which the gradient-descent rule is given by

$$\Delta \mathbf{w} = \alpha \sum_{t=1}^{m} (r - V_t) \frac{\partial V_t}{\partial \mathbf{w}}. \tag{1.32}$$

One specific difference of this situation to the supervised-learning examples before is that the reward is typically 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 time step t as

$$r - V_t = \sum_{k=t}^{m} (V_{k+1} - V_k). \tag{1.33}$$

Using this in equation 1.32 gives

$$\Delta \mathbf{w} = \alpha \sum_{t}^{m} \sum_{k=t}^{m} (V_{k+1} - V_k) \frac{\partial V_t}{\partial \mathbf{w}} \tag{1.34}$$

$$= \alpha \sum_{t=1}^{m} (V_{t+1} - V_t) \sum_{k=1}^{t} \frac{\partial V_k}{\partial \mathbf{w}}, \tag{1.35}$$

Which can be verified by writing out the sums and reordering the terms. Of course, this is just rewriting the original equation 1.32. 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 1.32 and 1.35 are equivalent, we also introduce here some modified rules suggested by Richard Sutton. In particular, we can weight recent gradients more 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

$$\Delta_t \mathbf{w} = \alpha(V_{t+1} - V_t) \sum_{k=1}^{t} \lambda^{t-k} \frac{\partial V_k}{\partial w}. \tag{1.36}$$

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

$$\Delta_t \mathbf{w} = \alpha(V_{t+1} - V_t) \frac{\partial V_t}{\partial w}. \tag{1.37}$$

This rule gives different results with respect to the original supervised learning problem described by TD(1), but this rule is local in time and does not require any memory. This TD(0) rule is actually equivalent to the original TD rule when no reward is given in intermediate states. The TD(λ) algorithm can be implementing with a multilayer perceptron when back-propagating the error term to hidden layers.

1.4.4 Deep Q-learning

A nice example of the success of TD(λ) was made by Gerald Tesauro from the IBM research labs and published in the *Communications of the ACM* March 1995 / Vol. 38, No. 3 with the title *Temporal Difference Learning and TD-Gammon*. His program learned to play the game at an expert level. The following is an excerpt from this article (see http://www.research.ibm.com/massive/tdl.html):

"Programming a computer to play high-level backgammon has been found to be a rather difficult undertaking. In certain simplified endgame situations, it is possible to design a program that plays perfectly via table look-up. However, such an approach is not feasible for the full game, due to the enormous number of possible states (estimated at over 10 to the power of 20). Furthermore, the brute-force methodology of deep searches, which has worked so well in games such as chess, checkers and Othello, is not feasible due to the high branching ratio resulting from the probabilistic dice rolls. At each ply there are 21 dice combinations possible, with an average of about 20 legal moves per dice combination, resulting in a branching ratio of several hundred per ply. This is much larger than in checkers and chess (typical branching ratios quoted for these games are 8-10 for checkers and 30-40 for chess), and too large to reach significant depth even on the fastest available supercomputers."

At the time of TD-Gammon, the MLP with one hidden layer has been a the state of the art, more elaborate models with more hidden layers have been difficult to train. However, deep learning has no made major progress based on several factors, including faster computers in form go GPUs, larger databases

with lots of training example, the rediscovery of convolutional networks, and better regularization techniques. The combination of deep learning with reinforcement learning has recently made mayor breakthroughs in AI sushi as learning to play ATARI games and most recently that a deep learning system called alphaGo won against a grandmaster of the Chinese board game of Go, which has been considered previously one of the most challenging examples for AI.

There are several aspects in these solutions which seems important to make it possible to train such large networks. One is to find a good starting position to generate valuable responses. That is, if one starts playing the games with random weights it is unlikely to even get to a point were sensible learning can take place. This, the alphaGo used supervised learning on expert data to train the system initially, while the RL procedures could then go on and advance the system to the point where it could outperform human players. A second well known factor is that it is important to use frequent replay of episodes during learning.

1.5 Actor-Critic schemes and the basal ganglia

We have contrasted above a critic, which can learn a value function and then dictates the policy, and an actor which tries to search for optimal policies by searching in the action space. In practice it has been shown that a combination of these systems works quite well as it combines the strength of both worlds. That is, the learning of the actor should be guided by a critic, and only making smaller changes in the actor instead of following entirely possibly large fluctuations in the critic is also desirable.

Sutton and Barto have incorporated temporal difference learning into a powerful control method called the *actor–critic scheme*, illustrated in Fig. 1.4. Note the similarities of this control scheme to the inverse model controller outlined in Fig. **??**B. The critic is designed to predict the correct motor command for accurate future actions and can thus supervise the motor command generator. The motor command generator is called *actor* within this framework. The *adaptive critic* estimates value functions and uses them to guide the actions of the agent. This scheme has proven to be very useful in engineering applications such as controlling elevators or adjusting the parameters of a petroleum refinery's operation. Neural implementations have been suggested, and it has been noted that there are some similarities with specific architectural features in the *basal ganglia*.

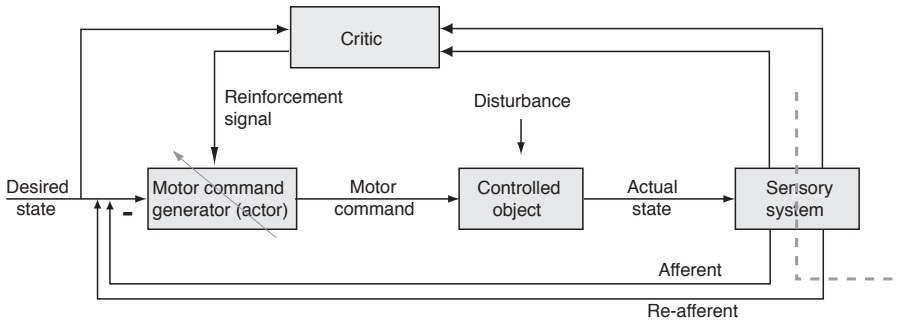

Fig. 1.4 Adaptive critic controller.

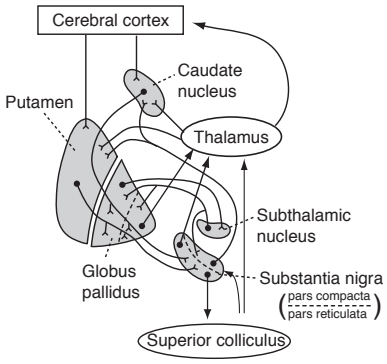

Fig. 1.5 Anatomical overview of the connections within the basal ganglia and the major projections comprising the input and output of the basal ganglia. [Adapted from Kandel, Schwarz, and Jessell, *Principles of neural science,* McGraw-Hill, 4th edition (2000).]

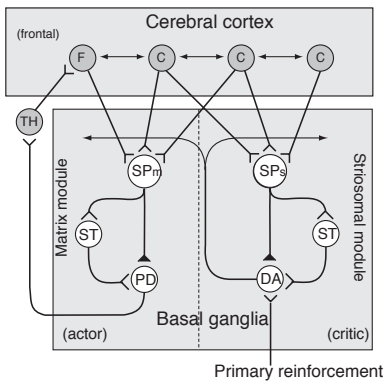

Fig. 1.6 Actor–critic model of the basal ganglia with cerebral cortex (C), frontal lobe (F), thalamus (TH), subthalamic nucleus (ST), pallidus (PD), spiny neurons in the matrix module (SPm), spiny neurons in the striosomal module (SPs), substantia nigra pars compacta (SNc) dopaminergic neurons (DA).

The basal ganglia, which are known to be instrumental in the initiation of motor commands, have many of the structural components required to implement an adaptive critic and to supervise actors that can control the initiation of motor movements. The basal ganglia are a collection of five subcortical nuclei as illustrated in Fig. 1.5. They receive cortical and thalamic input mainly through the *putamen* and the *caudate nucleus*, together called the *striatum* which comprises the input layer of the basal ganglia. The information stream then runs through the *globus pallidus* (with an internal and external segment) to the major output layer, the *substantia nigra pars reticulata*. The internal side-loop from the globus pallidus via the *subthalamic nucleus* is also important for our next discussion. In addition, note that the *substantia nigra pars compacta* projects back to the striatum with neurons that use *dopamine* as neurotransmitter.

The information streams within the basal ganglia are thought to be segregated (to a certain extent) within interleaved modules that are called matrix modules and striosomal modules, respectively. A suggested implementation of the actor–critic scheme in the basal ganglia is shown in Fig. 1.6. The input layer of the basal ganglia is rich in *spiny neurons* (SP), which receive massive cortical (C) connections. The spiny neurons in the striosomal module (SPs) also receive projections from dopaminergic neurons (DA) in the substantia nigra pars compacta (SNc) which synapse on to spines of spiny neurons in the caudate and putamen. It is possible that dopaminergic input is able to alter the efficiencies of specific cortical inputs that are marked with an eligibility trace. The neurons shown in the basal ganglia are also inhibitory so that the dopaminergic neurons in the SNc are inhibited by SPs activity. The subthalamic side-loop, in contrast, disinhibits the DA, which can result in some excitation of DA neurons proportional to the inputs from this side-loop and a primary reinforcement signal. If, in addition, the side-loop is faster than the direct SPs–DA influence, then it is possible that the striosomal module implements the critic that minimizes the temporal difference, as discussed above (compare Fig. 1.3C).

Several experiments show signals of neural activities in the basal ganglia that can be related to reinforcement learning. For example, work by *Wolfram Schultz* has shown how activities of dopaminergic neurons in SNc relate well to the temporal difference error in reinforcement learning. An example is shown in Fig. 1.7. These neurons respond to an unexpected reward, but do not re-

spond to a reward after the agent has learned to associate the reward with a stimulus. The activity of the dopaminergic neurons in SNc is also transferred when introducing an earlier predictive stimulus. Furthermore, these neurons respond with decreased activity when an expected reward is not given. This behaviour corresponds well to the effective reinforcement signal in the schemes discussed above and simulated below. An example is also shown in Fig. 1.7 where reward was only given at the time of presenting Pattern 3. Before learning, the neuron which conveys the effective reinforcement signal responds only at the time following the reward (the time Pattern 4 is presented), but the response is transferred to the time of presenting Pattern 3 after learning. At episode 50, a new pattern (Pattern 2) is introduced at the time step preceding Pattern 3, and the neurons transfers the response to this pattern. The removal of an expected reward leads to a negative value of the effective reinforcement learning signal, not shown in the simulation.

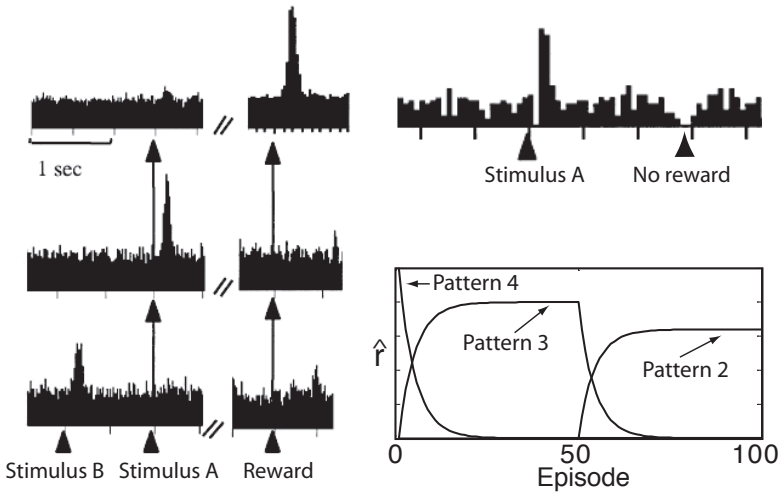

Fig. 1.7 Activities of dopaminergic neurons from Schultz and colleagues [adapted Suri, *Neural Networks* 15: 523–533 (2002)] and results from simulations discussed below.

The similarities between reward responses of dopaminergic neurons in the SNc and the effective reinforcement signal in temporal difference learning have contributed largely to the hypothesis of reinforcement learning theories of the basal ganglia. This part of the reinforcement system corresponds largely to the critic in the control system. Specific implementations of actors in the basal ganglia have been discussed much less in the literature. One of the earliest suggestions was that the matrix module could implement the actor. Dopaminergic neurons project to these spiny neurons (SPm) and could therefore alter the C–SPm weights in a fashion similar to that in the C–SPs connections. The only difference is that these neurons project to the internal segment of the pallidus and the substantia nigra pars reticulata (SNr), which are thought to be the major output layers of the basal ganglia. The output of the basal ganglia can, in such a way, control the initiation of specific motor actions that are associated with reward.

The classic proposal of an actor–critic implementation in the basal ganglia discussed above leaves many open questions and some questionable assump-

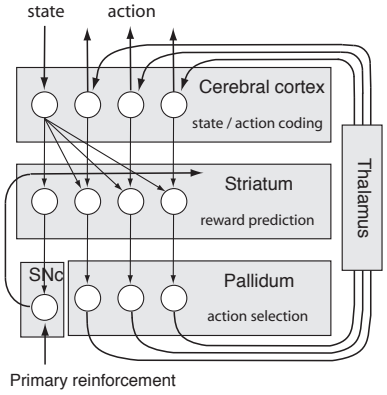

Fig. 1.8 Q-learning scheme of the basal ganglia [adapted from Doya, *HFSP Journal* 1: 30–40 (2007)].

tions. While there are still many open questions regarding the specific implementation of reinforcement learning in the basal ganglia and the more refined role of the basal ganglia in brain processing, there is a wide consensus that the basal ganglia do contribute to some form of reinforcement learning. More recently, Q-learning has also been mapped on to basal ganglion functions in the work of leading researchers such as *Kenji Doya* and *Peter Dayan*. An example is illustrated in Fig. 1.8 where the striatum is hypothesized to calculate a state-action value function, and the dopaminergic neurons in SNc calculate the TD error that is used to update cortico-striatal connections. The state-action value function is then used in the pallidum and thalamus for stochastic action selection. Other researchers, such as *Peter Redgrave* and *Kevin Gurney* also pointed out that the basal ganglia might have a large role in discovering novelty actions, which are also essential in establishing task-relevant behaviour. This area is a nice example where theoretical and experimental research strongly benefit from each other.

While we concentrated our discussion of reinforcement learning in the brain on the basal ganglia, it should not be forgotten that there are other areas in the brain that have been associated with reinforcement learning, or at least association of reward contingencies with specific motor actions. Some brain areas that have been implicated with making reward associations are, for example, the prefrontal cortex and the subcortical area called *amygdala*. Both areas receive projections from many senses and are therefore placed strategically to form associations between different modalities and reward contingencies. Bilateral damage of the amygdala is known to considerably impair such associations. Furthermore, it has been shown that neurons in the *orbitofrontal cortex* adapt their response to stimuli after changing their reward associations. Dopaminergic neurons also project into the frontal cortex so that reward mechanisms could originate predominantly in the frontal cortex as opposed to the hypothesis of the basal ganglia discussed above. Finally, there are also other neurotransmitters and specific reward systems that might be important in reinforcement learning. For example, serotonin is a neurotransmitter that has been associated with the modulation of such things as mood, appetite, sexuality, and aversive signals, which are all factors that can influence reward values and action selection. Also, human planning, as well as abilities to evaluate long-term goals, are not yet covered with the simple reward associations used in the experiments discussed here.