9 Reinforcement Learning

9.1 Learning from reward and the credit assignment problem

9.1.1 The reinforcement learning problem

In supervised learning we assumed that a teacher supplies detailed information of the desired response of a learner. This was particularly suited to object recognition where we had a large number of labeled examples. However, there are different learning circumstances which seem to be much more common in real world applications and for human-type learning. For example, learning to play tennis is based on trying out moves and getting rewarded by points we score rather than a teacher who specifies every muscle movement we need to follow. It is indeed common that we only get some simple feedback after long periods of actions in form of reward or punishment without even detailing which of the actions has be crucial. So exploration is part of the game.

In this type of learning scenario, which we generically call **reinforcement learning**, there are several types of challenges. One is called the **credit assignment** problem. This includes which action (**spatial credit assignment**) and at which time (**temporal credit assignment**) should be given credit for the achievement of reward. So we must search for solutions by trying different actions. Another problem is when we find a series of actions that give us some reward. The question is, should we now only do these actions or should we search for better solution? This problem is commonly stated as **exploration versus exploitation**.

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. 9.1.1. Inthe 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. Similar settings can be applied to robots, for example robots that must learn to find a charging station. Even our original object recognition problems can be formulated in terms of reinforcement such the problem of naming the correct class to which an object belongs.

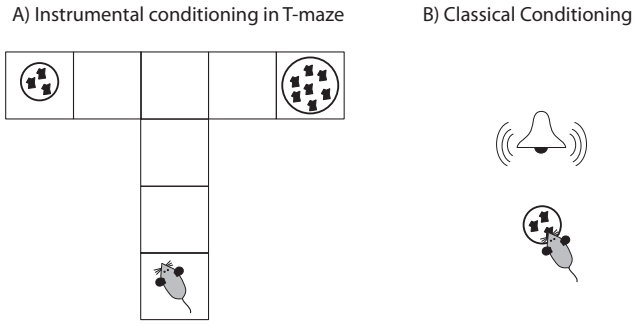

Fig. 9.1 Example of instrumental and classical conditioning. A) A rodent has to learn to transverse the maze and make a decision at the junction in which direction to go. Such as decision problem, which necessitates the action of an actor, is called instrumental conditioning in the animal learning literature. B) A slightly simpler setting is that of classical conditioning which does not require an action and thus concentrates on learning the reward associations. An typical example is when a subject is required to associate the ringing of a bell with reward.

9.1.2 Formalization of the problem setting: The Markov Decision Process

To discuss RL we need to formalize the reinforcement setting a bit more. We consider an agent that in each time interval t is in a specific state s_t from which it can take an action u_t. This action specifies a transition to a new state, and this transition is specified by the transition function τ as

$$\textbf{Transition function:} \quad s_{t+1} = \tau(s_t, u_t). \tag{9.1}$$

The transition function only depend on the previous state and the intended action from this state, which is called the **Markov condition**. 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 situation described by the Markov condition is quite natural as many decisions processes only depend on the current state, so it is a good place to start.

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

$$\textbf{Reward function:} \quad r_{t+1} = \rho(s_t, u_t). \tag{9.2}$$

This reward functions returns the value of reward when the agent is in state s_t and takes intended action u_t. In the deterministic case this is of course the reward at the next state $s_{t+1} = \tau(s_t, u_t)$.

Finally, an important function in reinforcement learning is the control **policy** that specifies which action u to take from each state,

$$\textbf{Policy:} \quad u_t = \pi(s_t) \tag{9.3}$$

In the context of a mobile agent, the action u_t is a motor command that the agent should take at time t, the time when the agent is in state s_t. For example, a muscle that should move the arm should be activated with a certain nerve pulse, or a robot

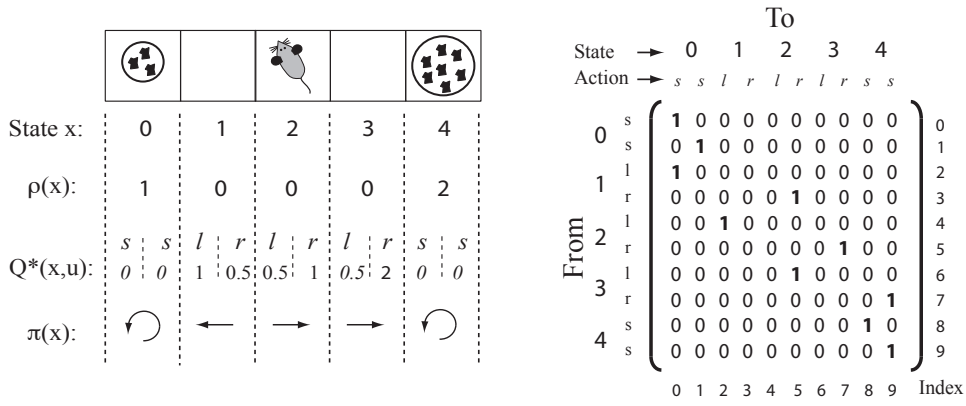

Fig. 9.2 Example experiment with the simplified T-maze where we concentrate on the more interesting horizontal portion of the maze (linear maze). The right hand side shows the corresponding transition matrix for the optimal policy.

that should turn is activated by a certain motor that should run for a certain time with a certain speed.

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. 9.2. We labeled the states x of the maze as 0 to 4. A reward of value 1 is provided in state 0 and a reward of 2 is provided in state 4. The discrete Q-function has 10 values corresponding to each possible action in each state. In states 1,2, and 3 these are the actions of move *left* or move *right*. The states with the reward, states 0 and 4, 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.

We provide here several python functions for later use. First there is the transition function $\tau(s,u)$ and the reward function $\rho(s,u)$. Next we provide a function to calculate the policy from a value function. This will be discussed further below. Finally, we provide a helper function $\mathrm{idx}(u)$ to transforms the action representation $u \in \{-1,1\}$ to the corresponding indices $\mathrm{idx} \in \{0,1\}$:

```
## Reinforcement learning in a maze
import numpy as np
import matplotlib.pyplot as plt

def tau(s,u):
    if s==0 or s==4: return(s)
    else: return(s+u)
def rho(s,u):
    return(s==1 and u==-1)+2*(s==3 and u==1)
def calc_policy(Q):
    policy=np.zeros(5)
    for s in range(0,5):
        uids=np.argmax(Q[s,:])
```

```
            policy[s]=2*uidx-1
    return policy
def idx(u):
    return((u+1)/2)
```

9.1.3 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 called **return** in economics. Of course, if we assume that this goes on forever than this return should be infinite, and we have hence to be a but more careful. One common choice is to define the return as the average reward in a finite time interval, also called the **finite horizon** case. Another common form to keep the return finite is to use a **discounted return** in which an agent values immediate reward more than reward far in the future. To capture this we define a discount factor $0 < \gamma < 1$. In the example program we will use a value of $gamma = 0.5$,

```
## discount factor
gamma=0.5;
```

although values much closer to one such as $\gamma = 0.99$ are common. For this case we now define a **state-action value function** which gives us a numerical value of the return (all future discounted reward) when the agent is in state x and takes action u and then follows the policy for the following actions,

$$\textbf{Value function (state-action):} \quad Q^{\pi}(s,u) = \rho(s,u) + \sum_{t=1}^{\infty} \gamma^t \rho(s_t, \pi(s_t)) \tag{9.4}$$

In other words, this functions tells us how good is action u in state s, and the knowledge of this value function should hence guide the actions of an agent. The aim of value-based reinforcement learning is to estimate this function.

Sometimes we are only interested in the value function when we follow the policy even for the first step in the action sequence from state x. This value function does then not depend explicitly on the action, only indirectly of course on the policy, and is defined as The total discounted return from state $s = s_0$ following policy π is,

$$\textbf{Value function (state):} \quad V^{\pi}(s) = Q^{\pi}(s, \pi(s)) \tag{9.5}$$

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

$$\textbf{Optimal Value function:} \quad V^{*}(s) = \max_{u} Q^{*}(s,u), \tag{9.6}$$

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

$$\textbf{Optimal policy:} \quad \pi^{*}(s) = \arg\max_{u} Q^{*}(s,u). \tag{9.7}$$

This function is implemented above with the python code for `calc_policy`.

The optimal value function and the optimal policy are closely related. We will discuss in the following several methods to calculate or estimate the value function

from which the policy can be derived. These methods can be put under the heading of **value-search**. Corresponding agents, or part of the corresponding RL algorithms, are commonly called a **critic**. However, there are also methods to learn the policy directly. Such methods are called **policy-search** is the corresponding agents are called an **actor**. At the end we will argue that combining these approaches in an **actor-critic scheme** has attractive creatures, and such schemes are increasingly used in practical applications.

9.2 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 $\rho(s,u)$ and the transfer functions $\tau(s,u)$. The knowledge of these functions, or a model thereof, is required for **model-based RL**, which is also called **dynamic programming**.

9.2.1 The basic Bellman equation

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

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

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

$$Q^\pi(\tau(s,u), \pi(\tau(s,u))) = V^\pi(\tau(s,u)). \tag{9.8}$$

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

$$\pi \textbf{ Bellman equation:} \quad Q^\pi(s,u) = \rho(s,u) + \gamma Q^\pi(\tau(s,u), \pi(\tau(s,u))). \tag{9.9}$$

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 $\rho(s,u)$ and the transition functions $\tau(s,u)$ are known. At this point the agent follows a specified policy $\pi(s)$. Let us further assume that we have X states and U possible actions in each state. We have thus $X \times U$ unknown quantities $Q^\pi(s,u)$ which are governed by the Bellman equation above. More precisely, the Bellman equations 9.9 are $X \times U$ coupled linear equations of the unknowns $Q^\pi(s,u)$. It is then convenient to write this equation system with vectors

$$\mathbf{Q}^\pi = \mathbf{R} + \gamma \mathbf{T}^\pi \mathbf{Q}^\pi \tag{9.10}$$

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

$$\mathbf{R} = (\mathbb{1} - \gamma \mathbf{T}^\pi)\mathbf{Q}^\pi, \tag{9.11}$$

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

$$\mathbf{Q}^\pi = (\mathbb{1} - \gamma \mathbf{T}^\pi)^{-1}\mathbf{R} \tag{9.12}$$

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.

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.9.12. We order therefore the quantities such as ρ and Q with ten indices. The first one correspond to $(s = 0, u = -1)$, the second to $(s = 0, u = 1)$, the third to $(s = 1, u = -1)$, etc. The reward vector can thus be coded as:

```
################################################
print('Analytic solution for optimal policy:')
# Defining reward vector R
i=0; R=np.zeros(10)
for s in range(0,5):
    for u in range(-1,2,2):
        R[i]=rho(s,u)
        i += 1
```

The transition matrix depends on the policy, so we need to choose one. We chose the one specified on the left in Fig. 9.2 where the agent would move to the left in state $s = 2$ and to the right in states $s = 3$ and $s = 4$. 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. 9.2. For example, if we are in state $s = 4$ and move to the left, $u = -1$, corresponding to the from-indes=7, then we end up in state $s = 3$, from which the policy say go right, $u = 1$. This correspond to the to-indes=6. Thus, the transition matrix should have an entry $T(7, 6) = 1$. Going through all the cases leaves us with

```
# Defining transition matrix
T=np.zeros([10,10]);
T[0,0]=1; T[1,1]=1; T[2,0]=1; T[3,5]=1; T[4,2]=1
T[5,7]=1; T[6,5]=1; T[7,9]=1; T[8,8]=1; T[9,9]=1
```

With this we can solve this linear matrix equations with the `inv()` function in the linear algebra package of numpy,

```
# Calculate Q-function
Q=np.linalg.inv(np.eye(10)-gamma*T) @ np.transpose(R)
Q=np.transpose(np.reshape(Q,[5,2]))
```

We reshaped the resulting Q-function so that the first row shows the values for left movements in each state and the second row shows the values for a right movement in each state. From this we can calculate which movement to take in each state, namely just the action corresponding to the maximum value in each column,

```
policy=np.zeros(5)
for s in range(0,5):
    uids=np.argmax(Q[s,:])
    policy[s]=2*uidx-1
```

Finally we print the results with

```
print(np.transpose(Q),'policy:',np.transpose(policy))
```

which gives

```
Analytic solution for optimal policy:
[[ 0.   1.   0.5  0.5  1. ]
 [ 0.   0.5  1.   2.   0. ]] policy: [-1. -1.  1.  1. -1.]
```

That is ignoring the end states it is moving left in state 1 as this would lead to an immediate reward of 1 and moving right in the other states as this would leave to a larger reward even when taking the discounting for more steps into account. Of course, at this our argument is circular as we have started already with the assumption that we use the optimal policy as specified in the transition matrix to start with. We will soon see how to start with an arbitrary policy an improve this to find the optimal strategy. Also note that the transition matrix was perfect in the sense that the intended move always leads to the intended end state. We will later see that a probabilistic extension of this transition matrix is quite useful in describing more realistic situations.

In the code we save the optimal Q-values for the optimal policy

```
Qana=Q
```

so that we can later plot the difference of other solution methods.

The Bellman equations are a set of n coupled linear equations for n unknown Q values, and we have solved these here with linear algebra function to find an inverse of a matrix. This is usually implemented with some form of Gauss elimination. Alternatively we can solve the Bellman equation with an iterative way. 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,

$$\textbf{Dynamic Programming:}\quad Q_{i+1}^\pi(s,u) \leftarrow \rho(s,u) + \gamma Q_i^\pi(\tau(s,u), \pi(\tau(s,u))). \tag{9.13}$$

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. The corresponding code is

```
################################################
print('Dynamic Programming:')
```

```
Q=np.zeros([5,2])
for iter in range(3):
    for s in range(0,5):
        for u in range(-1,2,2):
            act = np.int(policy[tau(s,u)])
            Q[s,idx(u)]=rho(s,u)+gamma*Q[tau(s,u),idx(act)]

for s in range(0,5):
    uids=np.argmax(Q[s,:])
    policy[s]=2*uidx-1
print(np.transpose(Q),'policy:',np.transpose(policy))
```

This is a much more common implementation and it does not require the explicit coding of the transition matrix. This is the approach we will take in all further methods. Note that we have only used three iterations to converge on the right solution. While we set here the number of iterations by hand, in practice we iterate until the changes in the values are sufficiently small.

9.2.2 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,

$$\textbf{Policy iteration:} \pi(s) \leftarrow \arg\max_{\mathrm{u}} \mathrm{Q}^{\pi}(\mathrm{s},\mathrm{u}). \tag{9.14}$$

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

$$\textbf{Optimal policy:} \quad \pi^*(s). \tag{9.15}$$

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

```
################################################
print('Policy iteration:')

Q=np.zeros([5,2])
policy=np.zeros(5)
for iter in range(3):
    for s in range(0,5):
        for u in range(-1,2,2):
            act = np.int(policy[tau(s,u)])
            Q[s,idx(u)]=rho(s,u)+gamma*Q[tau(s,u),idx(act)]
    for s in range(0,5):
```

```
        uids=np.argmax(Q[s,:])
        policy[s]=2*uidx-1
print(np.transpose(Q),'policy:',np.transpose(policy))
```

Note that we again only iterated three times over the policies. In principle we could and should iterate several times for each policy in order to converge to a stable estimate for this Q^π. However, the improvements will anyhow lead very quickly to a stable state, at least in this simple example.

9.2.3 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,

$$Q^*(s,u) = \rho(s,u) + \gamma Q^*(\tau(s,u), \pi^*(\tau(s,u))). \tag{9.16}$$

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

$$\textbf{Optimal Bellman equation:} \quad Q^*(s,u) = \rho(s,u) + \gamma \max_{u'} Q^*(\tau(s,u), u'). \tag{9.17}$$

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

$$\textbf{Q-iteration:} \quad Q^*_{i+1}(s,u) \leftarrow \rho(s,u) + \gamma \max_{u'} Q^*(\tau(s,u), u'). \tag{9.18}$$

The corresponding code for our maze example is

```
################################################
print('Q-iteration')

Q_new=np.zeros([5,2])
policy = np.zeros(5)
for iter in range(10):
    for s in range(0,5):
        for u in range(-1,2,2):
            maxValue = np.maximum(Q[tau(s,u),0],Q[tau(s,u),1])
            Q_new[s,idx(u)]=rho(s,u)+gamma*maxValue
    Q=np.copy(Q_new);

for s in range(0,5):
    uids=np.argmax(Q[s,:])
    policy[s]=2*uidx-1
print(np.transpose(Q),'policy:',np.transpose(policy))
```

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.

9.3 Model-free Reinforcement Learning

9.3.1 Temporal Difference Method for Value Iteration

Above we assumed a model of the environment by an explicit knowledge of the functions $\tau(s,u)$ and $\rho(s,u)$. 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 $u=\pi(s)$ and observe a reward r_{i+1} and the next state s_{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

$$\textbf{SARSA:}\quad Q_{i+1}(s_i,u_i) = Q_i(s_i,u_i) + \alpha\left[(r_{i+1} + \gamma Q_i(s_{i+1},u_{i+1}) - Q_i(s_i,u_i)\right]. \tag{9.19}$$

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

$$\textbf{ϵ-greedy policy:}\quad p\left(\arg\max_u Q(s,u)\right) = 1-\epsilon. \tag{9.20}$$

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

```
################################################
print('SARSA:')

Q=np.zeros([5,2])
policy = np.zeros(5)
error = []
alpha=1;
for trial in range(100):
    s=3
    for t in range(0,5):
        uids=np.argmax(Q[s,:])
        u=2*uidx-1
        # epsilon greedy
```

```
        if np.random.rand()<0.1:
            u=-u

        Q[s,idx(u)]=(1-alpha)*Q[s,idx(u)]+alpha*(rho(s,u)+gamma*Q[tau(s,u),idx(u)
        s=tau(s,u)
        error.append(np.sum(np.sum(np.abs(np.subtract(Q,Qana)))))

for s in range(0,5):
    uids=np.argmax(Q[s,:])
    policy[s]=2*uidx-1

print(np.transpose(Q),'policy:',np.transpose(policy))
plt.figure()
plt.plot(error)
```

Note that we have set here the learning rate $\alpha = 1$ for this example, which makes the update rule look similar to dynamic programming

$$Q_{i+1}(s_i, u_i) = (r_{i+1} + \gamma Q_i(s_{i+1}, u_{i+1}). \tag{9.21}$$

However, there is a major difference. In dynamic programming we iterate over all possible states with the knowledge of the transition function and the reward function. Thus an agent does not really has to explore the environment and can just "sit there" and calculate what the optimal action is. This is the benefit of model-based reinforcement learning. In contrast, here we discuss the case where we do not know the transition function and the reward function and hence have to explore the environment by acting in it. As there is usually associated with a physical movement, this takes times and hence limits the exploration we can do. Hence it is common that an agent can not explore all possible states. Also, a learning rate of $\alpha = 1$ is not always advisable since a more common setting is that reward itself is probabilistic. A smaller value of α then represents a form of taking a sliding average and hence estimating the expected value of the reward.

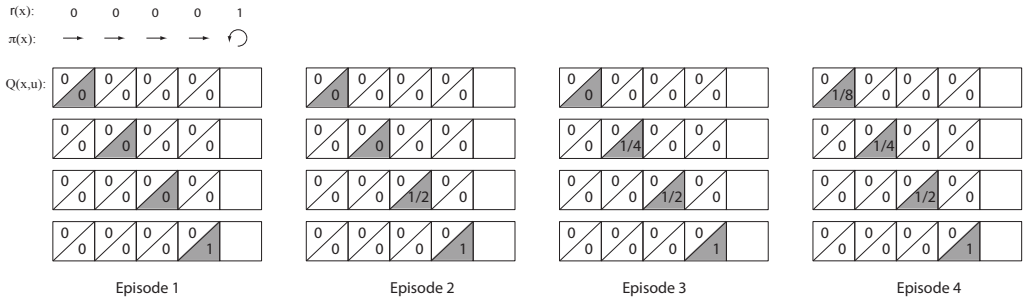

Fig. 9.3 Example of the "back-propagation" of the reward (not to be confused with the back propagation algorithm in supervised learning). In this example, an episode always starts in the leftmost state and the policy is to always go right. A reward is received in the rightmost state.

It is illustrative to go through the SARSA algorithm by hand for our linear-maze example. An example is shown in Fig. **??**.In this example we changed the situation to a linear maze in which the state always starts at the leftmost state and a reward of $r = 1$ is received in the rightmost state. The policy is to always go right, which is also the optimal policy in this situation. At the first time step of the first episode we are in the leftmost state and evaluate the value of going right. In the corresponding state to the right there is no reward given, and the value function is also zero. So the value function of this state-action is zero. The same is true for every step until we are in the state before the reward state. At this point the value is updated to the reward of the next state. In the second episode the value of the first and second state remains zero, but the third state is updated to $\gamma * 1$ since the value of the next state following the policy is given by one, and we discount this by γ. Going though more episodes it can be seen that the value "back-propagates" by one step in each episode. Note that this back propagation of the value is not to be confused with the back propagation algorithm in supervised learning. Also, notice that the values for the Q-function for going left are not updated as we only followed optimal policy deterministically. Some exploration steps will eventually update these values, although it might take a long time until these values propagate through the system.

Finally, a more common version of a temporal difference learning is to use an alternative way to estimate the value function using an **off-policy** approach for the estimation step from each visited state. That is we check all possible actions from the state that we evaluate state and update the value function with the maximal expected return,

$$\textbf{Q-learning:} \quad Q_{i+1}(s_i, u_i) = Q_i(s_i, u_i) + \alpha \left[(r_{i+1} + \gamma \max_{u'} Q_i(s_{i+1}, u') - Q_i(s_i, u_i) \right]. \tag{9.22}$$

We still have to explore the environment which is again usually following the optimal estimated policy with some allowance for exploration such as ϵ-greedy or a softmax exploration strategy.

The corresponding code for the Q learning is

```
################################################
print('Q-Learning:')

Q=np.zeros([5,2])
policy = np.zeros(5)
alpha=1
error = []
for trial in range(100):
    s=3
    for t in range(0,5):
        uids=np.argmax(Q[s,:])
        u=2*uidx-1
        if np.random.rand()<0.1:
            u=-u
        Q[s,idx(u)]=(1-alpha)*Q[s,idx(u)]+alpha*(rho(s,u)+gamma*np.maximum(Q[tau(
        Q[0]=0;Q[4]=0;
```

```
        s=tau(s,u)
        error.append(np.sum(np.sum(np.abs(np.subtract(Q,Qana)))))
for s in range(0,5):
    uids=np.argmax(Q[s,:])
    policy[s]=2*uidx-1

print(np.transpose(Q),'policy:',np.transpose(policy))
plt.plot(error,'r')
plt.show()
```

9.3.2 TD(λ)

The example of the linear maze in the previous section has shown that the expectation of reward propagates backwards in each episode which hence requires multiple repetition of the episodes in order to evaluate the value function. The reason for this is that we only give credit for making a step to a valuable state to the previous step and hence only update the corresponding value function. A different approach is to keep track of which states have led to the reward and assign the credit to each step that was visited. However, because we discount the reward proportional to the time it takes to get to the rewarded state, we need to also take this into account.

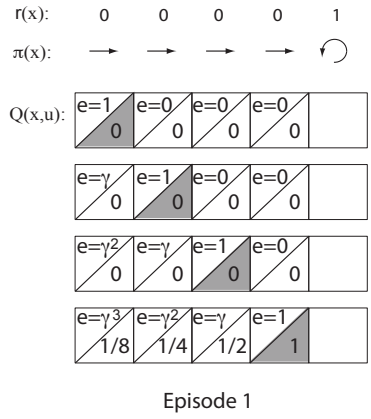

Fig. 9.4 Example of the "back-propagation" of the reward (not to be confused with the back propagation algorithm in supervised learning). In this example, an episode always starts in the leftmost state and the policy is to always go right. A reward is received in the rightmost state.

To realize this we introduce an eligibility trace that we call $e(s)$. At the beginning, this eligibility trace is set to zero for all the states. The for each visited state we set this eligibility state to one for this current state, and we discount the eligibility for all the other states by γ. This is demonstrated in Fig. 9.4. In the figure we reused the place for the $Q(s,-1)$ to indicate the eligibility trace at every time step. Note how the eligibility trace is building up during one episode until reaching this the rewarded state, at which time the values of all the states are updated in the right proportion. This algorithm does therefore only need one optimal episode to find the correct value function, at

least in this case with a learning rate of $\alpha = 1$. This algorithm is implemented for the Q-learning version of temporal difference learning in the code below:

```
################################################
print('TD(1) for Q-Learning:')

Q=np.zeros([5,2])
alpha=1

for trial in range(100):
    s=2; eligibility=np.zeros(5);
    for t in range(0,5):
        if s==0 or s==4: break
        eligibility*=gamma
        eligibility[s]=1
        uidx=np.argmax(Q[s,:])
        u=2*uidx-1
        if np.random.rand()<0.1: u=-u #epsilon greedy

        for x in range(1,4):
            Q[x,idx(u)]=Q[x,idx(u)]+alpha*(rho(x,u)+gamma*np.maximum(Q[tau(x,u),0]
        s=tau(s,u)
print(np.transpose(Q),'policy:',np.transpose(calc_policy(Q)))
```

While this algorithms does cut down the number of episodes to learn the value function, it does so on the expense of keeping memories of visited states and their respective times. A compromise is to allow some decaying memory in the algorithm. This can be implemented simply by a factor λ ind replacing the update of the eligibility trace from

```
eligibility*=gamma
```

to

```
eligibility*=gamma*lambda
```

With a term $\lambda < 1$, this corresponds to an exponential decay of the eligibility trace and hence the necessary memory. We will see later that this can be implemented efficiently in neural networks,

9.3.3 Actor and policy search

In all of the 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

$$\textbf{Actor objective function:} \quad J(\pi) = E\{\sum_{k=0}^{\infty} \gamma^k \rho(s_t, \pi(s_t))\}_{s_0}, \tag{9.23}$$

where the averaging is over all starting states s_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. We do not discuss implementations of an actor at this point as it is more common these days to implement them with neural networks that are discussed shortly.

9.4 Using function approximation with neural networks

At this point we have outlines the basic ideas behind 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 to tabulate the values for the value function. Thus, in these programs the functions were really lookup tables or array in programming terms that specifies the value function for each discrete state and action. Correspondingly this lead to table for the policy. Such algorithms are now commonly referred to as **tabular RL algorithms**. The problem with this approach is that these tables can be very large for large state and action dimensionalities. Indeed, the increased computational demand of calculating these quantities in many real world applications is often prohibitive.

For example, let us think how this would look like if we would implement learning to play a computer game with these tabular RL methods. Let us discuss the example Atari 2600 games that have been implemented in an Arcade Learning Environment at the University of Alberta. This environment simulates video input of 210×160 RGB video at 60Hz, which is equivalent to a state input vector of length 100800 every 1/60 of a second. Even if each pixel is only allowed to have say 8 bit representations, equivalent to $2^8 = 256$ possible values, and all the pixel values are independent, there would be 256^{100800} possible states. Clearly this is impossible to implement. Bellman himself already noticed this practical limitations and coined the phrase "Course of Dimensionality". The principle idea for overcoming the Course of Dimensionality 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.

9.4.1 Value-function approximation with ANN

The basic idea for generalizing the RL ideas to a high dimensional state-action space, and even to continuous spaces, is to use function approximators. Several type of func-

tion approximators have been used in the past. Linear function approximation (linear regression) is often discussed in engineering book as this provides some good baseline and are somewhat tractable analytically. However, many real world applications are highly non-linear, and it is the reason we have discussed neural networks to start with. Indeed, neural networks have been applied to RL for some time. 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."

To illustrate the basic idea of using neural networks with RL, we are going back to our maze example. The basic form of the implementation of the Q-function with a neural network is to make a ANN that receives as input a state and an action, and the output is the Q value as shown in Fig. 9.5 with network A. The question is then, how do we train this network? In supervised learning we would need examples of the value function, but the whole point of value based RL is that we need to estimate this from reward given only at some states typically at the end of long sequences. The answer follows closely our previous strategy in the the supervised desired state is estimated by the reward received in the next state plug the current estimate of the expected return from this state following the policy. For example, using the approach of SARSA with a mean square objective function, we can define the following error term (Loss-function)

$$E(s_t, u_t) = (r_{t+1} + \gamma Q(s_{t+1}, u_{t+1}; w) - Q(s_t, u_t; w))^2 \tag{9.24}$$

and use back propagation to train the weights of the network that represent the Q-function.

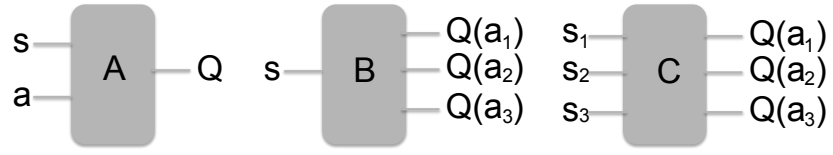

Fig. 9.5 Different ways to implement a function approximator for the value function.

While the neural network approach to SARSA can be applied immediately to a

continuous state and action space, many applications have a finite and relative small set of possible actions, and it is more common to use a Q-leaning (off-policy) strategy in this case. In this case we have to compare the Q values of all the possible actions from a specific state. While we could just use iterate over the previous network, it is now common to learn a network that provides the Q-values for all the possible actions. This approach has been suggested by Riedmiller (2005) and has been termed NFQ (Neurally Fitted Q-Iteration). The basic idea is shown in Fig. 9.5 with network B. In this case we can train the network with the following Loss-function

$$E(s_t, u_t) = (r_{t+1} + \gamma \max_u Q(s_{t+1}, u; w) - Q(s_t, u_t; w))^2 \tag{9.25}$$

which corresponds to training the connections to the winning node as well as the connections feeding to it through back propagation. To apply this strategy to the maze example, we can represent each state of the maze with a binary value that is one if the agent is in this state and zero otherwise. The network C in Fig. 9.5 shows this. The corresponding program is listed below,

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

gamma=0.5;
epsilon = 1

def tau(x,u):
    if (u == -1 and x[0] == 0):
        return(np.roll(x,-1))
    elif (u == 1 and x[4] == 0):
        return(np.roll(x, 1))
    else: return(x)
def rho(x):
    return(x[0]==1)+2*(x[4]==1)
def terminal_state(x):
    return(x[0]==1) or (x[4]==1)

def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev = 0.1)
    return tf.Variable(initial)
def bias_variable(shape):
    initial = tf.constant(0.1, shape = shape)
    return tf.Variable(initial)

tf.reset_default_graph()
sess = tf.InteractiveSession()
# the network
inpt = tf.placeholder(tf.float32, [None,5])
W1 = weight_variable([5, 10])
b1 = bias_variable([10])
```

```
W2 = weight_variable([10, 2])
b2 = bias_variable([2])
h1 = tf.nn.relu(tf.matmul(inpt, W1) + b1)
outpt = tf.matmul(h1, W2) + b2
# the loss function and trainer
Qtarget = tf.placeholder("float", [None,2])
cost = tf.reduce_sum(tf.square(Qtarget - outpt))
train_step = tf.train.GradientDescentOptimizer(learning_rate=0.1).minimize(cost)
sess.run(tf.initialize_all_variables())

print('training the network ....')
for trial in range(500):
    x= [0, 0, 1, 0, 0]
    for t in range(0,5):
        if terminal_state(x): break
        if trial > 3 and epsilon > 0.1: epsilon -= 0.001
        prediction = sess.run(outpt,feed_dict = {inpt : np.expand_dims(x, axis=0
        uidx=np.argmax(prediction)
        if (np.random.rand() < epsilon): uidx=np.random.randint(0,2)
        u=2*uidx-1
        next_x =  tau(x,u)
        if terminal_state(next_x):
            yy = rho(next_x)
        else:
            yy = rho(next_x) + gamma*np.max(sess.run(outpt,feed_dict = {inpt : np
        prediction[0,uidx] = yy
        sess.run([train_step,cost,h1],feed_dict={inpt:np.expand_dims(x, axis=0),
        x = np.copy(next_x)

policy = np.zeros(5)
xarray = [1,0,0,0,0]
Q=[]
for xx in range(0,5):
    Qxx=(sess.run(outpt,feed_dict = {inpt : np.expand_dims(xarray, axis=0)}))
    uidxx=np.argmax(Qxx)
    Q.append(Qxx)
    policy[xx]=2*uidxx-1
    xarray = np.roll(xarray,1)
print(np.transpose(Q),'policy:',np.transpose(policy))
```

9.4.2 Deep Q-learning

At the time of TD-Gammon, the MLP with one hidden layer has been the state of the art, more elaborate models with more hidden layers have been difficult to train. However, deep learning has now 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.

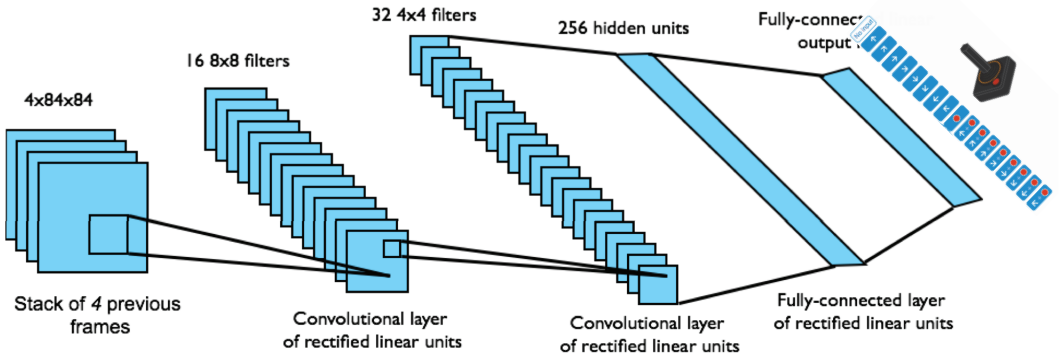

Fig. 9.6 Outline of the DQN network used to learn playing Atari games.

DQN (deep Q-learning network) is the basic network that has been used by Minh et al (2014/15) to learn to play Atari games from the Arcade Games Console benchmark environment. The network is basically a convolutional network as shown in Fig.9.6 which takes video frames as input and outputs Q-values for the possible joystick actions. While we outlines already the basic strategy of using the TD error with back propagation to train such networks in such a RL task, which basically represent the NFQ apporach, there are several important aspects in these solutions which seems important to make it possible to train such large networks. An important factor in the original DQN network was the use of **Experience Replay**. While the learning rate during learning has to be fairly small during learning to prevent single instances to dominate, this also means that specific episodes would have only a very small contribution and that one would need a large amount of episodes. In replay we memorize a chosen action and use mini-batches of random samples for training. A second important factor is the use of a **Target Q-network**. In this technique we freeze the parameters of the Q-network for the estimation of the future reward (lets call these weights w'), and calculate the temporal difference as

$$E = (r+\gamma maxQ(s',a',w')-Q(s,a,w))^2 \rightarrow \frac{\partial E}{\partial w} = (r+\gamma maxQ(s',a',w')-Q(s,a,w))\frac{\partial Q}{\partial w} \tag{9.26}$$

The weights of this target network are updated only periodically. Finally, **clipping rewards** or some form of **normalizing the network** are important to prevent an extreme buildup of Q values.

The reasons for all the above mention tricks is to keep the network somewhat stable since small changes in rewards can cause large fluctuations in policies. We come back to this point when discussing actor-critic networks below. Another important aspects

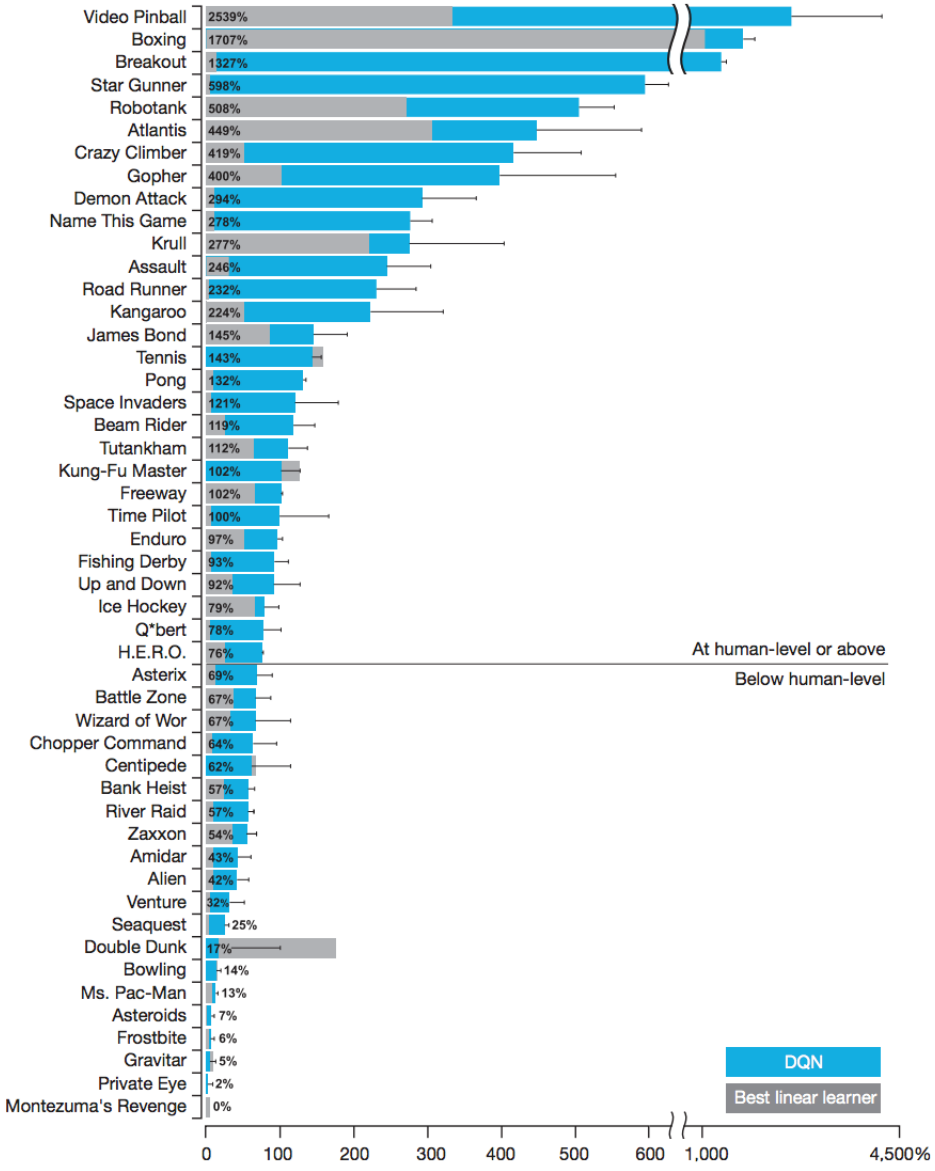

Fig. 9.7 Performance of the DQN network and a linear method on different Atari games (from Minh et al., 2015).

of even larger applications 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. Indeed, 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.

9.4.3 Actor-Critic schemes

As already suggested before, training a neural network while exploiting it to suggest actions is a dangerous situation and usually leads to oscillations and instabilities. One reason is that when Q-values are close to each other, then small changes in the Q-values can lead to drastic changes in response actions that can cause problems and inconsistencies in learning. With this respect, actor-based reinforcement learning seems to have some advantages, but building a appropriate parameterization of actions has it's own challenges. However, combining actors and critics has been shown to build much more robust systems. The basic scheme is illustrated in Fig.9.8 on the left, and in the context of using neural networks on the right. Another implementation is DDAC (Deterministic Deep Actor-Critic) as shown in Fig.9.9.

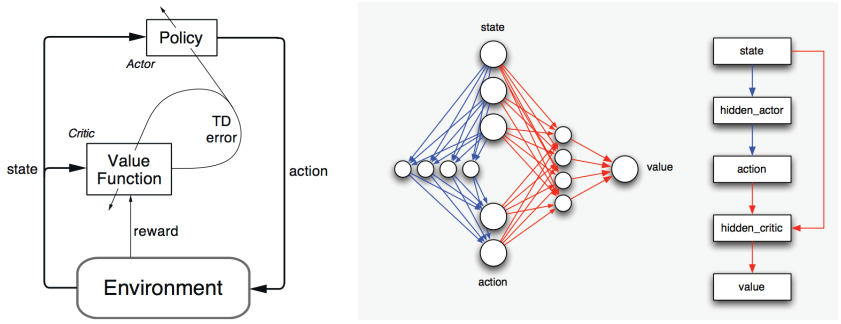

Fig. 9.8 Outline of the Actor-Critic approach (Sutton-Barto; 1998) and the neurally fitted Q-learning Actor-Critic (NFQAC) network (Rückstiess, 2010).

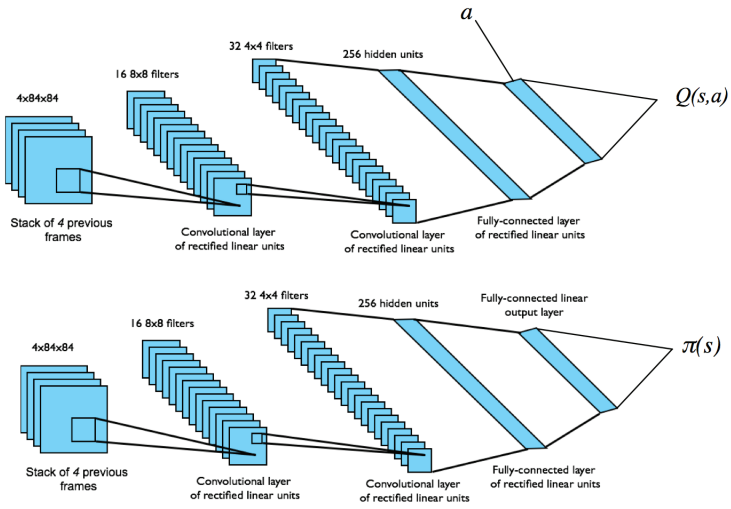

Fig. 9.9 Deterministic Deep Actor-Critic (DDAC) network (from Lillicrap, 2015).