Reinforcement Learning

We're looking at table-driven implementations of reactive agents, in which the action function is stored as a big lookup table. Each entry in the table maps from a percept to the action that should be taken. And we've considered two ways of creating this table: human designers can work out all the entries, or we can use a genetic algorithm to try to evolve a table of high fitness. In this lecture, we look instead at how the agent can *learn* an action function stored in a table.

We've studied learning previously, when we were looking at classifiers. But, what we studied before is a different *type* of learning from what we will study in this lecture. What we studied before is called *supervised learning*; what we will study here is *reinforcement learning*. What distinguishes them is the nature of the information the learner receives from the 'teacher' or from the environment:

Supervised learning: The learner receives examples of inputs and target outputs. E.g. each example might be an email and a human-supplied classification (ham or spam); or each example might be a description of a person's food and drink consumption along with a classification (under- or over-the-limit) based on a breathalyser reading. Do not, however, assume that supervised learning is used only when we are building a classifier. It is much broader. It can be used to learn a system that carries out regression. Remember, the job of a regression system is to predict a real number e.g. a system that predicts tomorrow's rainfall. In this case, the learner will receive descriptions of one day's weather along with the actual rainfall recorded on the following day. Equally, we can use supervised learning to learn an agent's action function. Suppose an expert chess player creates a set of examples, each comprising a board configuration plus the best move that can be made in that configuration. Then, we can use supervised learning to learn an agent function.

Reinforcement learning: In reinforcement learning, the learner receives only rewards (or punishments) after executing certain actions and thereby on entering certain states. The rewards (or punishments) act as positive (or negative) reinforcement on actions, i.e. they make it more (or less) likely that the agent will execute those actions if it finds itself in the same or similar situations in the future. For example, a character in a game might observe the effect its actions have on its score or on some measurement of its health; if it is capable of reinforcement learning, it can use these observations to improve its action function. A complication in many tasks is that reward is *delayed*, giving rise to a *credit assignment problem*, i.e. it must work out which of its earlier actions contributed to the reward and by how much to reinforce them. For example, in chess reward comes at the end of the game (when the agent wins or loses); in waiting on tables in a restaurant reward comes at the end of the meal (on receipt or non-receipt of a tip).

In this context, machine learning researchers often distinguish a third type of learning, namely *unsupervised learning*. We won't be covering unsupervised learning in this module.

Reinforcement learning is a huge subject, with many different algorithms for different settings. But in this lecture we gain a flavour of the topic by looking at only one such algorithm.

1 Reward

At each time step t, the agent uses its sensors to obtain a percept $\mathbf{s}_t$. We will assume that the environment is fully observable and that $\mathbf{s}_t$ completely determines the state that the agent is in. On the basis of $\mathbf{s}_t$, the agent chooses and executes an action a_t. The agent obtains a reward r_t (a real number) based on $(\mathbf{s}_t, a_t)$. This reward might come from a teacher who is observing the agent, or it might come from the external environment, or it might even come from within the agent itself (e.g. a measure of improved health or decreased hunger). Following execution of the action, the world will now be in a new state. The agent can sense this new state by using it sensors to obtain percept $\mathbf{s}_{t+1}$.

For simplicity, we will assume a deterministic environment. This means that if on more than one occasion the agent senses a particular percept $\mathbf{s}$ and chooses to execute a particular action a, then the reward and the next percept will be

the same on each occasion. In a non-deterministic environment, by contrast, after sensing $\mathbf{s}$ and executing a on one occasion the reward might be r' and the next percept might be $\mathbf{s}'$ but on another occasion after sensing $\mathbf{s}$ and executing a the reward might be r'' ($r' \neq r''$) and the next percept might be $\mathbf{s}''$ ($\mathbf{s}' \neq \mathbf{s}''$).

The agent tries to maximise the reward it receives in the long run. To make this more precise, we need to distinguish two cases. On the one hand, some agents execute actions until they reach a particular terminal state or one of a set of terminal states. When their sensors tell them they have reached a terminal state, they stop. For example, an agent that walks a maze stops when it reaches a particular point (e.g. the exit); an agent that plays a board game stops when the game is over (when it wins, loses or draws). Of course, the agent can then reinitialise itself and 'go again', e.g. walk the maze again, perhaps from a different starting point; or walk a different maze; or play another round of the game.

In the other case, the agent is not trying to reach any particular terminal state. Instead it acts continuously, in principle without limit. An example is an agent whose job is forever to walk the walls of a room. A more realistic example is an agent that monitors a factory process. To keep things simple, in this lecture we will focus on this second case, i.e. on agents that act continuously.

In the case of an agent that acts continuously, the cumulative reward that the agent receives over time is calculated as follows:

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$$

or, equivalently,

$$\sum_{t=0}^{t=\infty} \gamma^t r_t$$

You can see that it is the sum of the rewards, but the formula includes what is called *discounting*. γ is the *discount rate*, $0 \leq \gamma \leq 1$. If we set $\gamma = 0$, only r_0 is considered. If we set $\gamma > 0$, then the later a reward is received, the less it counts. As we set γ closer to 1, future rewards are taken into account more strongly.

The task of the agent is to learn an action function that produces the greatest possible cumulative reward for the agent over time.

2 Action-Value Functions

We're going to start by adding an extra column and extra rows to our table.

In previous lectures, our table had one row per percept. So, for example, if the agent has n touch-sensors (which return 0 or 1), there are 2^n different percepts and hence also 2^n rows in its table; each row pairs a percept with the action to be taken when that percept has been sensed.

But in this lecture, we will pair each percept with *each* action. For example, suppose the agent has 2 touch-sensors (which return 0 or 1) and is capable of three actions, `Move`, `Turn(RIGHT, 2)` and `Turn(LEFT, 2)`. Then, the table will have the following entries:

Percept	Action	Q
00	`Move`	
00	`Turn(RIGHT, 2)`	
00	`Turn(LEFT, 2)`	
01	`Move`	
01	`Turn(RIGHT, 2)`	
01	`Turn(LEFT, 2)`	
10	`Move`	
10	`Turn(RIGHT, 2)`	
10	`Turn(LEFT, 2)`	
11	`Move`	
11	`Turn(RIGHT, 2)`	
11	`Turn(LEFT, 2)`	

Class Exercise. Suppose the agent has n touch sensors (which return 1 or 0) and m different actions. How many rows will the table contain?

What is the extra column for? In a given row of the table in which the percept is $\mathbf{s}$ and the action is a, the Q-value, $Q(\mathbf{s}, a)$, will be an estimate of the cumulative reward that the agent will receive if it chooses action a on sensing percept $\mathbf{s}$. Roughly then, it says how good the action is in that situation.

If the Q-values are 'correct', then in situation $\mathbf{s}$, the agent should choose the action a for which $Q(\mathbf{s}, a)$ is highest. Mathematically, this is written

$$\arg\max_a Q(\mathbf{s}, a)$$

Class Exercise. Suppose the current percept, $\mathbf{s}$, is 01. And suppose that the Q-values are as follows:

Percept	Action	Q
⋮	⋮	⋮
01	`Move`	2
01	`Turn(RIGHT, 2)`	1
01	`Turn(LEFT, 2)`	7
⋮	⋮	⋮

What is $arg \max_a Q(\mathbf{s}, a)$, i.e. what action will the agent choose?

But this assumes that the Q-values are 'correct'. What we have to do now is consider how the agent can learn these values.

3 Q-Learning

We are going to start with randomly-chosen Q-values (or perhaps all zeroes). The agent will improve these values by a process of trial-and-error: it will choose and execute actions, and it will use the rewards it receives to update the Q-values. Over time, we hope that the values will converge to the 'correct' values.

Here is the basic algorithm. It continuous to assume an agent that acts continuously.

```
$\mathbf{s}$ := SENSE();
do forever
{   rand := a randomly-generated number between 0 and 1;
    if rand < $\epsilon$
    {   Choose action $a$ randomly;
    }
    else
    {   $a := \arg\max_a Q(\mathbf{s}, a)$;
    }
    EXECUTE($a$);
    $r$ := SENSEREWARD();
    $\mathbf{s}'$ := SENSE();
    $Q(\mathbf{s}, a) := r + \gamma \max_{a'} Q(\mathbf{s}', a')$;
    $\mathbf{s} := \mathbf{s}'$;
}
```

In essence, the agent repeatedly chooses an action, obtains a reward, and updates the values in the table.

Note that the algorithm uses both the following notation:

$$\arg\max_a Q(\mathbf{s}, a)$$

which, as we discussed previously, means the action with the highest Q-value for $\mathbf{s}$. But it also uses the following notation:

$$\max_{a'} Q(\mathbf{s}', a')$$

which means the highest Q-value for $\mathbf{s}'$ (rather than *the action* with the highest Q-value).

3.1 Exploration versus Exploitation

As you can see, sometimes the agent chooses a random action; other times it chooses the action that has the highest Q-value. It is choosing between *exploration* and *exploitation*:

Exploration: The agent chooses an action which, according to its current estimates, may not necessarily be the best action to take but, by choosing this action, it has the opportunity of gaining new experiences and improving its estimates of the Q-values.

Exploitation: The agent chooses the action which, according to its current estimates, is the best action to take (the one with the highest Q-value). This gives it the opportunity of gaining reward.

Pure exploration is of no use: it never puts into practice the information it has learned. Pure exploitation is no good: it leaves the agent stuck in a rut. The agent must strike a balance between the two. The algorithm takes a simple view of how to strike this balance: exploration is chosen with probability ϵ; exploitation is chosen with probability $1 - \epsilon$.

There are more sophisticated ways of striking this balance. For example, by keeping track of which actions it has chosen, it can favour actions that it has not tried very often. It also might make sense for the probability of exploration to decrease over time.

3.2 Updating $Q(\mathbf{s}, a)$

Let's see whether we can make sense of the way the algorithm updates the Q-values:

$$Q(\mathbf{s}, a) := r + \gamma \max_{a'} Q(\mathbf{s}', a');$$

We see that the new value is the reward the agent has just received for its latest action added to an estimate of the cumulative reward it can receive. Do we know the cumulative reward? We don't. But we can use the Q-value of the best action that we might take next. For this we use $\max_{a'} Q(\mathbf{s}', a')$. In other words, we find the action a' that, when executed in the new situation $\mathbf{s}'$, gives the highest Q-value, and we use that Q-value as our estimate of the cumulative reward we will receive in the future.

Class Exercise. Suppose the table currently contains the following entries:

Percept	Action	Q
⋮	⋮	⋮
10	`Move`	5
10	`Turn(RIGHT, 2)`	4
10	`Turn(LEFT, 2)`	1
11	`Move`	0
11	`Turn(RIGHT, 2)`	4
11	`Turn(LEFT, 2)`	6

1. Suppose the current percept, $\mathbf{s}$ is 10. Assuming exploitation rather than exploration, which action will the agent choose?
2. Suppose that, after executing the chosen action, the agent receives a reward of 3 and its next percept is 11. (Assume γ is 1.) Update the table accordingly.

Why does this formula for updating the table work? The first time the agent consults the table to choose an action for a given percept, the Q-values are arbitrary, and so they are not really giving any meaningful information on which to base the decision. However, once an action has been executed, the agent receives information (the reward), which it uses to update the table, hence improving the information the agent will obtain the next time it consults the table to choose an action for that percept.

Over the course of repeated updates, the Q-values will get better and better. When one estimated Q-value improves, then the estimated Q-values of its immediate predecessors will also improve next time they get updated.

Eventually, the tabulated Q-values should converge to give values that perfectly 'estimate' the actual cumulative rewards. In fact, this only happens under certain conditions, one of which is that every percept-action pair gets tried infinitely often.

The conditions of convergence are rather stringent. But, in practice, even when the Q-values are not perfectly correct, the agent may still end up with values that enable it to perform well in its environment.

4 Concluding Remarks

Reinforcement learning is an extremely promising approach. One of its notable successes is TD-Gammon, which, after training on 1.5 million backgammon games, came to rival the performance of expert human players. There's a lot more that could be said about it. If it interests you, you can look it up in a book. But here's a synopsis of some of the issues:

- In non-deterministic environments, the way we are updating Q-values is not guaranteed to converge, even if all percept-action pairs get tried infinitely often. This is why in some books you might see a different way of updating Q:
$$Q(\mathbf{s}, a) := Q(\mathbf{s}, a) + \alpha(r + \gamma \max_{a'} Q(\mathbf{s}', a') - Q(\mathbf{s}, a));$$
or equivalently
$$Q(\mathbf{s}, a) := (1 - \alpha)Q(\mathbf{s}, a) + \alpha(r + \gamma \max_{a'} Q(\mathbf{s}', a'));$$
This takes a weighted average of the current Q-value and the revised estimate. The parameter α ($0 \leq \alpha \leq 1$) weights the parts of this average and should decay over time.

- Books will discuss ways of improving convergence (for both deterministic and non-deterministic environments). In essence, these involve updating more than one Q-value each time round the loop. But this requires the agent to remember sequences of actions that it has taken, which requires a memory; hence the agent would not strictly be a reactive agent.

- As our agent tries out actions, it will be able to observe the consequences of actions, e.g. if it senses $\mathbf{s}$ and executes a, the agent obtains a reward of r and its next percept will be $\mathbf{s}'$. From this, it can build what is sometimes called a 'model' of the world. Books may discuss reinforcement learners that learn and use the 'model'; they may also discuss approaches in which it is assumed that the learner has been given a 'model' in advance. Again these agents might not qualify as being purely reactive.

- The use of an explicit lookup table, with an entry for every percept-action pair, is a major constraint. First, storage may be a problem: tables may be too large (or, for continuous environments, they may be infinite). Second, no attempt is made to generalise, i.e. no attempt is made to infer the Q-values of unseen percept-action pairs from seen percept-action pairs.

  Active research is looking at how to represent and learn the Q-function in some more compact form, in which generalisations are captured. One of the most successful approaches is to use kNN. Only some rows of the table are stored. Then, when we want a Q-value, we find the k-nearest neighbours and predict the Q-values from the neighbours. (This is regression, rather than classification.) Of course, updates to Q-values must be handled in a cleverer way too.

  Books are more likely to discuss training a neural net to store the Q-function. This has, in general, met with less success than the kNN approach.

- Books might discuss research that addresses concerns about scalability and convergence properties. Hierarchically organised tasks are, for example, an active area of research.