The Q Function

The expected value of a state if we perform a certain action, a, and then follow policy π, is called $Q^\pi(s,a)$.

It satisfies the following consistency condition:

$$Q^\pi(s,a) = \sum_{s'} \sum_{r} \sum_{a'} P(s_{t+1}=s',\, r_{t+1}=r \mid s_t=s,\, a_t=a)\, P(a_{t+1}=a' \mid s_{t+1}=s')\, (r + \gamma Q^\pi(s',a'))$$

Here, $P(a_{t+1}=a' \mid s_{t+1}=s')$ is determined by the policy π.

If the optimal policy, π, is deterministic, always taking action $\pi(s)$ in state s, then clearly $\pi(s)$ will be the action, a, that maximizes $Q^\pi(s,a)$.

So knowing Q^π is enough to define the optimal policy. Learning Q^π is one way of learning the optimal policy.

Exploration While Learning an Optimal Policy

When we don't know how the world works, we need to trade off between exploiting what we do know and exploring for useful new knowledge.

A simple (not necessarily optimal) scheme is to take what seems to be the optimal action with probability $1-\epsilon$, and take a random action (chosen uniformly) with probability ϵ. A larger value for ϵ will increase "exploration".

I'll use this scheme below. In practice, though, it might be better to never take actions that you're pretty sure will have disastrous consequences.

Q Learning

We can merge together learning the optimal policy (modified to allow for exploration) with estimating the Q function for this policy. I'll refer to this procedure as "Q learning", though strictly speaking, that term may not apply when exploration is incorporated, as below.

We turn the consistency equation for Q into an update rule, except we update only a bit, so that over time the updates will be based on expected values. Based on our current Q function, in state s we take the action that maximizes $Q(s,a)$ with probability $1-\epsilon$, and take a random action with probability ϵ.

If we take action a in state s, and obtain immediate reward r and end in the new state s', then we update $Q(s,a)$ as follows:

$$Q(s,a) \;\leftarrow\; (1-\alpha)\, Q(s,a) \;+\; \alpha\, (r + \gamma\, (\,\epsilon\, \text{mean}\, [Q(s',a')] \;+\; (1-\epsilon)\, \max[Q(s',a')]))$$

The mean and max are over actions a'. The learning rate α controls how fast we allow Q to change. It needs to change slowly enough that the average change over many updates estimates the expected change.

A Demonstration Problem

A robot travels on a circular track, with ten positions, labelled 1, 2, ..., 10.

At each time step, the robot can take any of three actions:

1 Move to the next-lower-numbered position, wrapping from 1 to 10
2 Don't move
3 Move to the next-higher-numbered position, wrapping from 10 to 1

The movement indicated above happens with probability 0.95. With probability 0.05, the next state is instead chosen from among all ten, with equal probabilities.

If the robot chooses action 1 or action 3, it receives a reward of +1 with probability $s'/10$, where s' is the new state. Otherwise, it receives a reward of 0.

So, for example,

$$P(s_{t+1}=8,\, r_{t+1}=1 \mid s_t=7,\, a=3) \;=\; (0.95 + 0.05/10) \times 8/10$$

$$P(s_{t+1}=3,\, r_{t+1}=0 \mid s_t=9,\, a=2) \;=\; (0.05/10) \times 1$$

R Code for Q Learning

R code for doing Q learning, condensed from a program on the course web page:

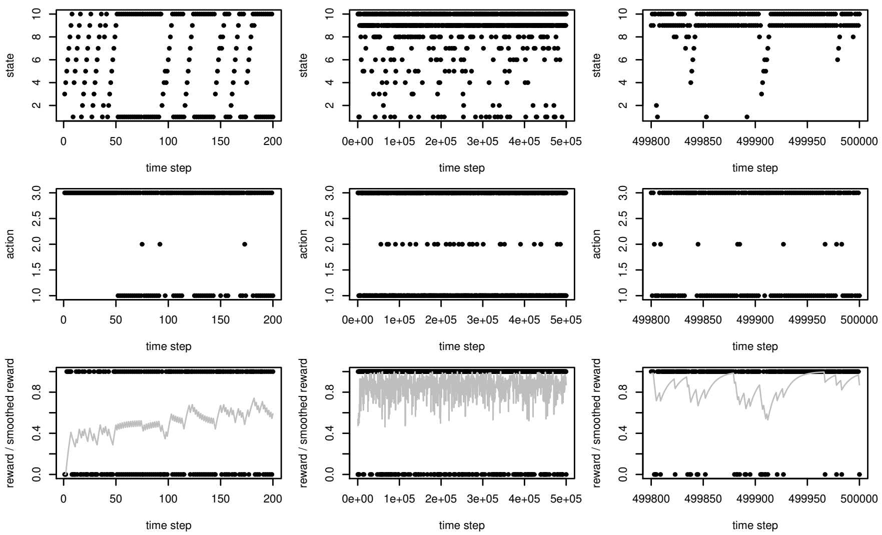
```
simulate = function (init, world, gamma, alpha, epsilon, steps)
{ ...
  s = init()
  for (t in 1:steps)
  { if (runif(1)<epsilon)
    { a = sample(n.actions,1)
    }
    else
    { a = order(Q[s,])[n.actions]
    }
    w = world(s,a)
    r = w$r
    sn = w$s
    Q[s,a] = (1-alpha) * Q[s,a] +
       alpha * (r + gamma * (epsilon*mean(Q[sn,]) + (1-epsilon)*max(Q[sn,])))
    s = sn
  }
... }
```


R Code for the Demonstration

```
init1 = function () { sample(n.states,1) }
world1 = function (s, a)
{ if (runif(1)<0.05)
  { s = sample(n.states,1)
  }
  else
  { s = s + (a-2)
    if (s<1) s = n.states
    if (s>n.states) s = 1
  }
  if (a==2)
  { r = 0
  }
  else
  { r = as.numeric (runif(1) < s/n.states)
  }
  list (s=s, r=r)
}
result1 = simulate (init1, world1, gamma, alpha, epsilon, n.steps)
```


Plots From a Simulation Run

500,000 times steps. First and last 200 shown at closer scale.



The grey line is an exponential smoothing of rewards up a given time.

The Q Function Learned From the Simulation Run

Here is the table of the Q function after 500,000 time steps:

		Action		
		1	2	3
State	1	17.29	16.27	15.68
	2	16.53	15.43	15.17
	3	15.53	14.83	15.88
	4	15.29	15.31	16.24
	5	15.66	15.64	16.50
	6	16.02	15.94	16.83
	7	16.61	16.18	17.00
	8	16.93	16.32	17.25
	9	17.03	16.30	17.29
	10	17.25	16.30	16.40

Note how the optimal direction to move changes between state 2 and 3, and how action 2 (don't move) always has value about one less than the optimal action.

How Well Does Q Learning Work When the State is Wrong?

We've so far assumed that Q learning is done with the correct state, that actually is all that's needed to predict future states and rewards. What if this is false? Does it still do something reasonable?

I modified the problem with the circular track so that each of the ten positions is either "marked" or not — this is part of the real state, but not part of the state seen by Q learning.

At each time step, a position that is not marked becomes marked with probability 0.3. When the robot's position changes, it gets +1 reward if its new position is marked. The position the robot is at becomes unmarked, and remains so until it moves.

Finally, the robot gets reward of −10 when it is at position 1.

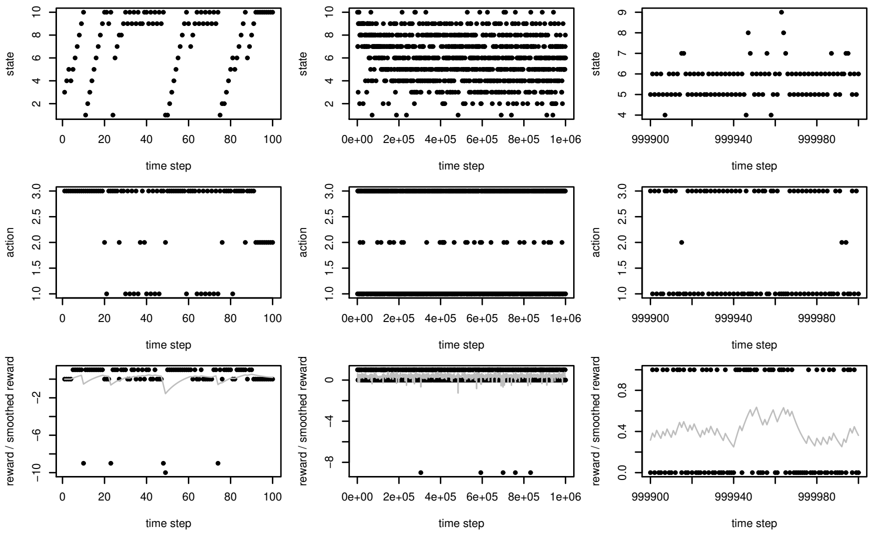
Without this last modification, the simple strategy of constantly moving in one direction is pretty much optimal. The need to avoid position 1 makes things more interesting.

R Code for the Modified Problem

```
init2 = function ()
{ marks <<- rep(0,n.states)  # Global variable storing extra part of state
  sample(n.states,1)
}

world2 = function (s, a)
{ if (runif(1)<0.05)
  { s = sample(n.states,1)
  }
  else
  { s = s + (a-2)
    if (s<1) s = n.states
    if (s>n.states) s = 1
  }
  r = marks[s] - 10*as.numeric(s==1)
  marks <<- as.numeric (marks>0 | (runif(n.states)<0.3))
  marks[s] <<- 0
  list (s=s, r=r)
}
```


Plot of Learning & Performance



The average reward over the one million time steps was 0.363.

Table of Q Function Learned

	1	2	3	best action
1	7.27	-2.55	7.85	3
2	-1.88	6.80	7.95	3
3	7.19	6.94	7.75	3
4	7.29	6.90	7.65	3
5	7.33	6.94	7.33	1
6	7.33	6.95	7.13	1
7	7.25	6.75	7.20	1
8	7.24	6.80	7.09	1
9	7.22	6.70	7.15	1
10	7.22	6.53	-1.64	1

The robot moves to higher positions in positions 1, 2, 3, 4 and to lower positions in positions 5, 6, 7, 8, 9, 10. (The action to do at position 5 is almost a toss-up.) This avoids position 1, usually going back forth and between two other positions.

Adding Some Memory to the State

Going back and forth between two positions (eg, 4 and 5) isn't optimal. Marks will get set at other positions, but won't be collected as reward. Average reward would be only 0.3 per time step, if it weren't for the occasional random moves to other positions.

Solution: Go back and forth over a larger set of positions, while avoiding position 1.

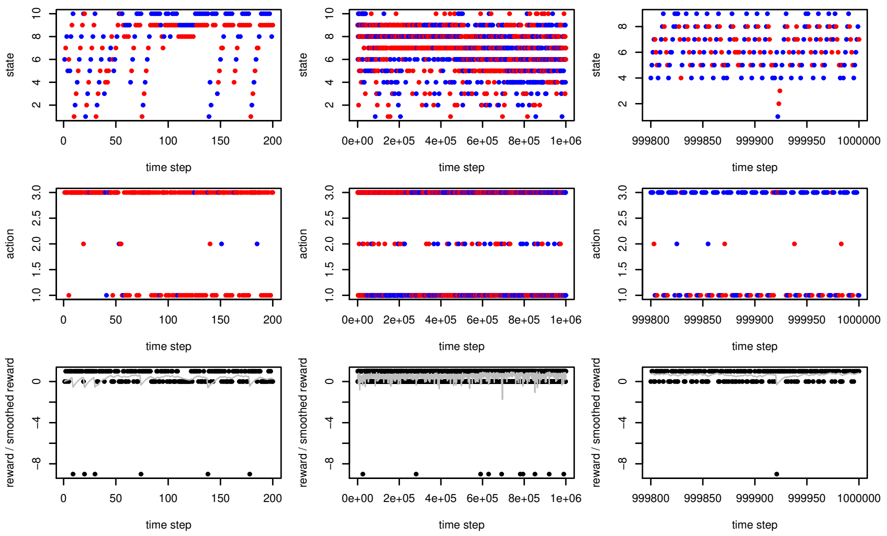
For this, the robot needs to remember which way it's going. So we provide it with one bit of memory, which is part of the state (so there are now 20 states).

The robot now has six possible actions — the three movements with no change in the memoery bit, and the three movements with the memory bit being toggled.

R Code for the Modified Problem with Memory

```
world2m = function (s, a)
{ bit = as.numeric(s>n.states/2)  # Obtain memory bit from state
  s = s - bit*n.states/2          # Get rid of memory from state for now
  if (runif(1)<0.05)
  { s = sample(n.states/2,1)      # Move to random position
  }
  else
  { if (a>3)                      # Action that toggles memory bit
    { bit = 1-bit
      a = a-3
    }
    s = s + (a-2)                 # Move according to action
    if (s<1) s = n.states/2
    if (s>n.states/2) s = 1
  }
  r = marks[s] - 10*as.numeric(s==1)  # Look at marks and update
  marks <<- as.numeric (marks>0 | (runif(n.states)<0.3))
  marks[s] <<- 0
  s = s + bit*n.states/2          # Add memory bit back into state
  list (s=s, r=r)
}
```


Plot of Learning & Performance



The average reward over the one million time steps was 0.528.

Table of Q Function Learned

	1	2	3	4	5	6	move	toggle
1	5.54	-0.20	4.80	4.43	-0.76	11.27	3	1
2	-0.72	7.02	8.34	0.04	6.84	11.61	3	1
3	8.89	8.74	9.45	8.77	8.74	11.51	3	1
4	11.08	10.74	11.65	11.06	10.72	11.19	3	0
5	11.19	10.75	12.17	11.22	10.94	11.44	3	0
6	11.30	11.00	12.06	11.29	11.16	11.52	3	0
7	11.46	11.10	12.02	11.49	11.15	11.54	3	0
8	11.39	10.84	11.98	11.39	11.06	11.46	3	0
9	11.15	10.74	10.93	11.76	10.76	10.94	1	1
10	7.31	6.26	-2.42	11.42	6.69	-1.61	1	1
11	4.88	-0.79	6.19	4.17	-0.75	11.36	3	1
12	-0.33	6.31	8.04	-0.32	6.77	11.52	3	1
13	8.89	8.78	9.61	8.92	8.65	11.63	3	1
14	11.04	10.52	11.03	11.09	10.53	11.73	3	1
15	11.46	10.99	11.43	11.76	10.93	11.09	1	1
16	12.01	11.13	11.12	11.47	11.05	11.32	1	0
17	12.13	11.08	11.25	11.49	11.00	11.26	1	0
18	12.05	11.05	11.14	11.37	10.90	11.20	1	0
19	11.69	10.73	10.96	11.05	10.64	10.94	1	0
20	7.09	6.71	-0.14	11.47	5.75	-0.46	1	1