Reinforcement Learning

Jesse Hoey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, CANADA, N2L3G1
jhoey@cs.uwaterloo.ca

1 Q-Learning

$Q^*(s,a)$ is **expected value** of being in state s and taking action a,

$$Q^*(s,a) = R(s) + \gamma \sum_{s'} Pr(s'|s,a) \max_{a'} Q^*(s',a')$$

Note that

$$V^*(s) = \max_a [Q^*(s,a)]$$

However, we can estimate Q *online* using temporal differences, by using a *learning rate*, $\alpha < 1$, in the following *Q-learning algorithm*:

initialise $Q[s,a]$ arbitrarily for all s and a
observe s
repeat forever:
  select a based on Q
  do a, observe reward r and state s'
  update Q-function:

$$Q[s,a] \leftarrow (1-\alpha)Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'])$$

  $s \leftarrow s'$
end repeat

Q-learning converges to Q^* (and hence V^*) if $\alpha = 1/N[s,a]$ where $N[s,a]$ is the number of times a is taken in state s, and if

1. $\sum_{t=0}^{\infty} \alpha_t = \infty$
2. $\sum_{t=0}^{\infty} \alpha_t^2 < \infty$

The only thing left to figure out is how to select a based on Q. We can either

- **Greedy**: Exploit our current Q by selecting $a^* = \arg\max_a Q([s,a])$
- **Exploratory**: Explore to find out more about Q, for example by taking a random a.

Usually, a Q-learning agent will do something in-betwen, by sometimes randomizing. One example is an ϵ-greedy approach ($0 \leq \epsilon \leq 1$)

1. ϵ of the time: exploit
2. rest of the time: take random action

Note: this is the opposite of the convention used in the book. It should be the other way around (so ϵ of the time it takes a random action).

The problem with the ϵ-greedy approach is that it treats all non-optimal actions the same. Instead, we can use **soft-max** action selection, where the agent chooses action a with probability

$$p(a) \propto e^{Q[s,a]/\tau}$$

and take $\tau \to 0$ slowly over time. When $\tau = 0$, then this **soft-max** action selection is the same as a purely greedy approach. To see why, consider a state and two possible actions, a_1 and a_2, and write $Q(s, a_i) = Q_i$, we have that

$$p(a_1) = \frac{e^{Q_1/\tau}}{e^{Q_1/\tau} + e^{Q_2/\tau}}$$

If $\tau \to \infty$, then this is just $\frac{1}{2}$, and we have an agent that acts randomly. If, on the other hand $\tau \to 0$, consider $\frac{1}{p(a_1)}$

$$\frac{1}{p(a_1)} = \frac{e^{Q_1/\tau} + e^{Q_2/\tau}}{e^{Q_1/\tau}} = 1 + e^{(Q_2 - Q_1)/\tau}$$

which, if $Q_2 > Q_1$, is ∞ (so $p(a_1) = 0$) and if $Q_1 \geq Q_2$ is 1 (so $p(a_1) = 1$). This is a greedy agent.

Another method is **optimism in face of uncertainty**, in which we initialise all Q values to be very high, then follow greedy (exploit-only) policy. This works because exploratory actions will always be chosen if available.

The final method to mention is the **Upper Confidence Bound (UCB)** formula, in which we also store $N[s, a]$ (number of times that state-action pair has been tried) and use

$$\arg\max_a \left[Q(s, a) + k\sqrt{\frac{N[s]}{N[s, a]}} \right]$$

where $N[s] = \sum_a N[s, a]$. The idea is that, as $N[s, a]$ gets small with respect to $N[s]$ (this is an action that has not been tried very much at s), the second term will eventually overcome any differences in $Q(s, a)$: our confidence in the value of $Q(s, a)$ for this un-tried action drops with respect to the other actions, and the UCB formula will eventually give the un-tried action another try. This method works well in theory and practice.

2 SARSA

In SARSA the only thing that changes is the term $\max_{a'} Q(s', a')$. In Q-learning, you take the Q-value of the best next action a' to take (the one that maximizes $Q(s', a')$), whereas in SARSA you use the Q-value of the actual next action you will take, $a^\dagger$, as determined by your exploration policy (which may include a random choice for example), and use $Q(s', a^\dagger)$.

So the update rule works like this:

SARSA:

- take action a, get reward r and end up in state s'

- select the action ($a^\dagger$) that will be taken in s' (e.g. use $\text{argmax}_{a'}\, Q(s', a')$ 90
- update: $Q(s, a) = (1 - \alpha)Q(s, a) + \alpha \left(r + \gamma Q(s', a^\dagger)\right)$
- take action $a^\dagger$ in state s'

Q-LEARNING:

- take action a, get reward r and end up in state s'
- find the best action to take in s' according to the current Q-function, $a^* = \text{argmax}_{a'}\, Q(s', a')$
- update: $Q(s, a) = (1 - \alpha)Q(s, a) + \alpha \left(r + \gamma Q(s', a^*)\right)$
- select the action ($a^\dagger$) that will be taken in s' (e.g. use $\text{argmax}_{a'}\, Q(s', a')$ 90
- take action $a^\dagger$ in state s'

Note that in Q-learning, the Q function is not necessarily being updated with the action that will be taken, whereas in SARSA, it is.

Here is pseudocode for SARSA

initialise $Q[s, a]$ arbitrarily for all s and a
observe s
repeat forever:
  select a based on $Q(s, a)$ (using an exploration policy)
  do a, observe reward r and state s'
  select a' based on $Q(s', a')$ (using an exploration policy)
  update Q-function:

$$Q[s, a] \leftarrow (1 - \alpha)Q[s, a] + \alpha(r + \gamma Q[s', a'])$$

  $s \leftarrow s'$
end repeat

3 Deep Q Network

Deep Reinforcement Learning uses an error function which is based upon the temporal difference formula:

$$Q[s, a] \leftarrow (1 - \alpha)Q[s, a] + \alpha(r + \gamma \max_{a'} Q[s', a'])$$

This formula says that, for an observed state transition s, a, s', r, we expect $Q[s, a]$ to converge to $r + \gamma \max_{a'} Q([s', a'])$. Thus, we will use an error function for a deep network which is the squared difference between the current estimate and the value estimated from the currently gathered reward and the next best Q value. Thus, the network will take as input the state s (represented as a set of features, one input per feature), and produce at the last (L^{th}) layer as output the Q function for each action a (written as $output^L[a](s)$), and the error will be:

$$error((s, a, s', r), w) = (output^L[a](s) - (r + \gamma \max_j output^L[j](s')))^2$$

The derivative can be taken as usual by implementing backpropagation through the network. In practice, it is not sufficient to simply throw a deep network at the problem, because Q-learning may diverge

for this case, as Q is defined as a non-linear function of the input features. To mitigate this problem, two techniques are used. First, two networks are used, one that produces $output^L[a](s)$ and the other that produces the target $\widehat{output}^L[a](s)$, and the error function is:

$$error((s, a, s', r), w) = (output^L[a](s) - (r + \gamma \max_j \widehat{output}^L[j](s')))^2$$

The backpropagation updates are applied to the $output$ network, while the $\widehat{output}$ network stays the same. Every now and again, the $\widehat{output}$ network is replaced with the $output$ one. Second, a buffer of (s, a, s', r) experiences is kept and after each experience, backpropagation is applied across a mini-batch of randomly sampled experiences from the buffer.

4 Questions

1. Design a reinforcement learning robot that will make your breakfast for you. You would like this robot to make sure the breakfast is still hot by the time you get there, and since it usually takes you about 10 minutes to get up and to the breakfast room, the robot must plan when to wake you. Describe the following elements of this agent. Note that this problem involves reasoning about time explicitly. You can do this by modeling time explicitly, or by modeling the effects of time (e.g. the coffee cooling down). Usually, the second option leads to an simpler model.

   (a) the states (assume full observability).
   (b) the actions the system can take
   (c) the rewards the system will get
   (d) how the agent will trade-off exploration and exploitation

   SOLUTION: this is meant as an open-ended question to get you thinking about how to define a RL agent for a real situation. You can make this as detailed as you like (e.g. should your robot be able to deal with flies in the kitchen? Does your robot have a fly-swatter for such situations? What if it discovers that the eggs have gone off? etc.)

2. A Q-learning agent is in a state, s, with the following Q function Q(s,a) and reward function R(s,a) for each of the four actions available to it (a1,a2,a3 and a4):

action	$Q(s, a)$	$R(s, a)$
a1	4	-1
a2	-3	-2
a3	3	0
a4	3	1

   (a) What is the optimal action, a^*, to take using a Greedy Q-learning policy for this agent?

   SOLUTION: a1

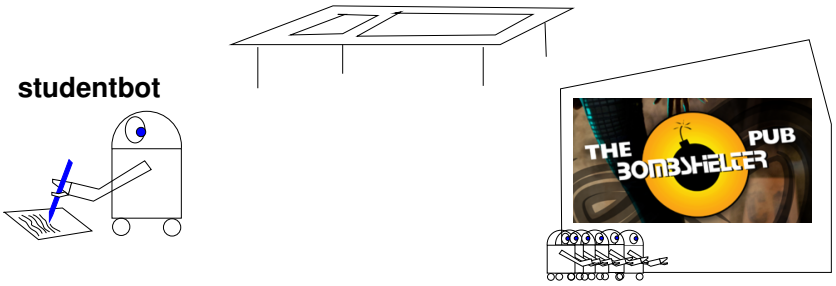
   (b) Suppose that the agent takes action a^*, and winds up in state s' in which

   $$max_{a'} Q(s', a') = 1.$$

   If the agent is using a learning rate of 0.1, what is the new value computed for $Q(s, a^*)$?

SOLUTION: $Q(s, a1) \rightarrow (1-\alpha)Q(s, a1) + \alpha\left(r + \gamma \max_{a'} Q(s', a')\right)$ which, since $Q(s, a1) = 4$ and $r = -1$ is 3.59

3. Studentbot is a robot whose goals make it exactly like a typical Waterloo student.



Studentbot has 4 actions as follows:

- **study**: studentbot's knowledge increases, studentbot gets tired
- **sleep**: studentbot gets less tired
- **party**: studentbot has a good time if he's not tired, but gets tired and loses knowledge
- **take test**: studentbot takes a test (can take test anytime)

Student bot's state is described with the following four variables (3x2x4x2=48 states):

- **tired**: studentbot is tired (no/a bit/very)
- **passtest**: studentbot passes test (no/yes)
- **knows**: studentbot's state of knowledge (nothing/a bit/a lot/everything)
- **goodtime**: studentbot has a good time (no/yes)

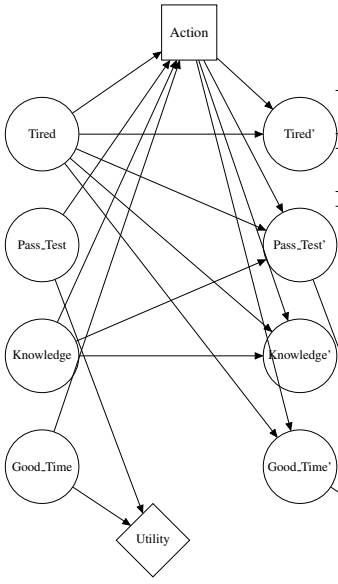
Studentbot's utility function is as follows:

- **+20** if studentbot passes the test
- **+2** if studentbot has a good time

The Decision network for studentbot, including all CPTs is as shown here:

			Knowledge′			
Action	*Tired*	*Knowledge*	*nothing*	*a_bit*	*a_lot*	*everything*
study	*no*	*nothing*	0.5	0.5	0	0
	no	*a_bit*	0	0.5	0.5	0
	no	*a_lot*	0.0	0.0	0.5	0.5
	no	*everything*	0		0	1.0
	a_bit	*nothing*	0.5	0.5	0	0
	a_bit	*a_bit*	0	0.5	0.5	0.0
	a_bit	*a_lot*	0	0	0.5	0.5
	a_bit	*everything*	0	0	0	1.0
	very	*nothing*	1.0	0	0	0
	very	*a_bit*	0	1.0	0.0	0.0
	very	*a_lot*	0	0	1.0	0.0
	very	*everything*	0	0	0	1.0
party	–	*nothing*	1.0	0.0	0	0
	no	*a_bit*	0.5	0.5	0.0	0.0
	no	*a_lot*	0.0	0.5	0.5	0.0
	no	*everything*	0	0	0.5	0.5
sleep	–	*nothing*	1.0	0.0	0	0
	–	*a_bit*	0.0	1.0	0.0	0.0
	–	*a_lot*	0.0	0.0	1.0	0.0
	–	*everything*	0	0.0	0.0	1.0
take_test	–	–	1.0	0.0	0	0

		Good_Time′	
Action	*Tired*	*no*	*yes*
party	*no*	0.0	1.0
party	*a_bit*	0.2	0.8
party	*very*	1.0	0.0
other	–	1.0	0.0



		Tired′		
Action	*Tired*	*no*	*a_bit*	*very*
study	*no*	0.8	0.2	0.0
study	*abit*	0.0	0.8	0.2
study	*very*	0.0	0.0	1.0
sleep	–	1	0	0
party	*no*	0	1	0
party	*a_bit*	0	0	1
party	*a_lot*	0	0	1
take_test	–	0	0	1

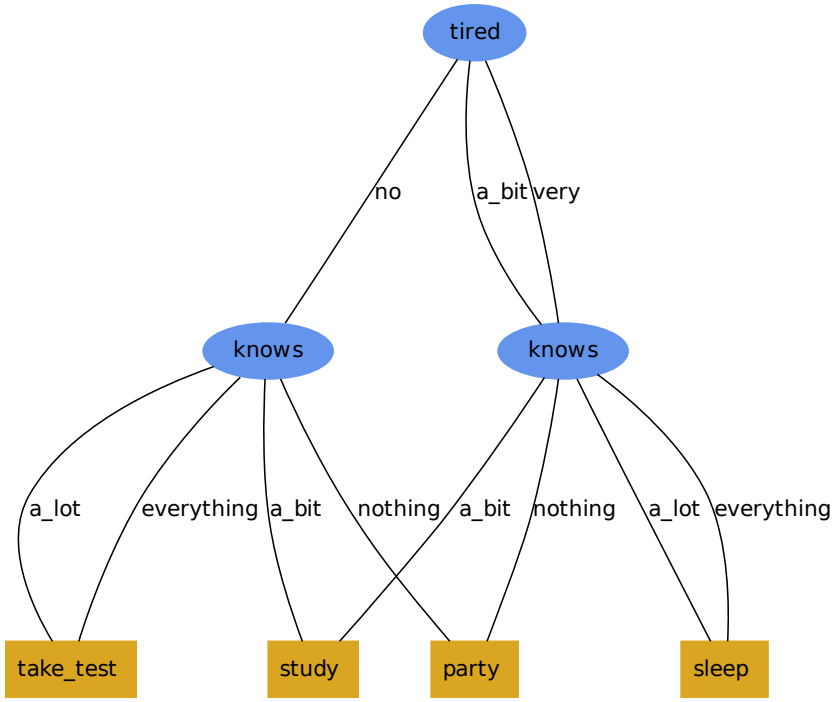
			PassTest′	
Action	*Tired*	*Knowledge*	*no*	*yes*
take_test	*no*	*nothing*	1	0
	no	*a_bit*	0.9	0.1
	no	*a_lot*	0.7	0.3
	no	*everything*	0.1	0.9
	a_bit	*nothing*	1	0
	a_bit	*a_bit*	0.9	0.1
	a_bit	*a_lot*	0.7	0.3
	a_bit	*everything*	0.1	0.9
	very	–	1	0
else	–	–	1	0

Good_Time	*Pass_Test*	*U(Good_Time, Pass_Test)*
yes	*yes*	22
yes	*no*	2
no	*yes*	20
no	*no*	0

Studentbot has a good time when she parties, and gains knowledge when she studies, but these effects are weakened if Studentbot is tired. Studentbot gets tired when she studies or parties, but gets less tired if she sleeps. Finally, Studentbot passes the test if she takes the test when she has knowledge, and the probability of her success on the test depends both on how tired she is and how much knowledge she has. Studentbot's basic tradeoff is between short term rewards for partying vs. long-term rewards for studying, leading to success on the test. Use $\alpha = 0.1$ and $\gamma = 0.9$ and work out a few iterations for actions of your choice. Start with $Q(s, a) = 0$ for all states and actions.
The order of actions in this table is $[study, sleep, party, test]$
and the order of states is $[tired, passtest, knows, goodtime]$:

state s	$Q(s, A)$	action a	new state s'	reward	$Q(s', A')$	updated $Q(s, A)$
$[no, no, nothing, no]$	$[0, 0, 0, 0]$	party	$[abit, no, nothing, yes]$	2	$[0, 0, 0, 0]$	$[0, 0, 0.2, 0]$
$[abit, no, nothing, yes]$	$[0, 0, 0, 0]$	party	$[very, no, nothing, yes]$	2	$[0, 0, 0, 0]$	$[0, 0, 0.2, 0]$
$[very, no, nothing, yes]$	$[0, 0, 0, 0]$	party	$[very, no, nothing, no]$	0	$[0, 0, 0, 0]$	$[0, 0, 0, 0]$
$[very, no, nothing, no]$	$[0, 0, 0, 0]$	sleep	$[no, no, nothing, no]$	0	$[0, 0, 0.2, 0]$	$[0, 0.018, 0, 0]$
$[no, no, nothing, no]$	$[0, 0, 0.2, 0]$	party	$[abit, no, nothing, yes]$	2	$[0, 0, 0.2, 0]$	$[0, 0, 0.398, 0]$
$[abit, no, nothing, yes]$	$[0, 0, 0.2, 0]$	study	$[abit, no, abit, no]$	0	$[0, 0, 0, 0]$	$[0, 0, 0.18, 0]$
$[abit, no, abit, no]$	$[0, 0, 0, 0]$	take test	$[very, yes, nothing, no]$	10	$[0, 0, 0, 0]$	$[0, 0, 0, 1]$
$[very, yes, nothing, no]$	$[0, 0, 0, 0]$	sleep	$[no, no, nothing, no]$	0	$[0, 0, 0.398, 0]$	$[0, 0.03582, 0, 0]$
$[no, no, nothing, no]$	$[0, 0, 0.398, 0]$	study	$[abit, no, alot, no]$	0	$[0, 0, 0, 1]$	$[0, 0, 0.4482, 0]$
$[abit, no, alot, no]$	$[0, 0, 0, 1]$	take test	$[very, yes, nothing, no]$	10	$[0, 0.03582, 0, 0]$	$[0, 0, 0, 1.903]$

This is the final policy (as a decision tree) for Studentbot after convergence:



5 Further Reading

The standard text on reinforcement learning is Sutton's book [SB98]. A good introduction to RL can be found in Kaelbling's survey [KLM96]. Deep Reinforcement Learning is described in [MKS+15], in which Atari games were tackled with the Deep Q-network algorithm. The network consisted of two convolutional network layers and two fully connected layers with rectified linear layers in between each one (a squashing function $\max(0, x)$), and achieved or surpassed human-level play for many of them.

References

[KLM96] Leslie Pack Kaelbling, Michael Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, February 2015.

[SB98] Rich Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.