studywolf

a blog for things I encounter while coding and researching neuroscience, motor control, and learning

NOV 25 2012

52 COMMENTS

BY TRAVISDEWOLF LEARNING, PROGRAMMING, PYTHON, REINFORCEMENT LEARNING

Reinforcement learning part 1: Q-learning and exploration

We've been running a reading group on Reinforcement Learning (RL) in my lab the last couple of months, and recently we've been looking at a very entertaining simulation for testing RL strategies, ye' old cat vs mouse paradigm. There are a number of different RL methods you can use / play with in that tutorial, but for this I'm only going to talk about Q-learning. The code implementation I'll be using is all in Python, and the original code comes from one of our resident post-docs, Terry Stewart, and can be garnered from his online RL tutorial (https://github.com/tcstewar/ccmsuite). The code I modify here is based off of Terry's code and modified by Eric Hunsberger, another PhD student in my lab. I'll talk more about how it's been modified in another post.

Q-learning review

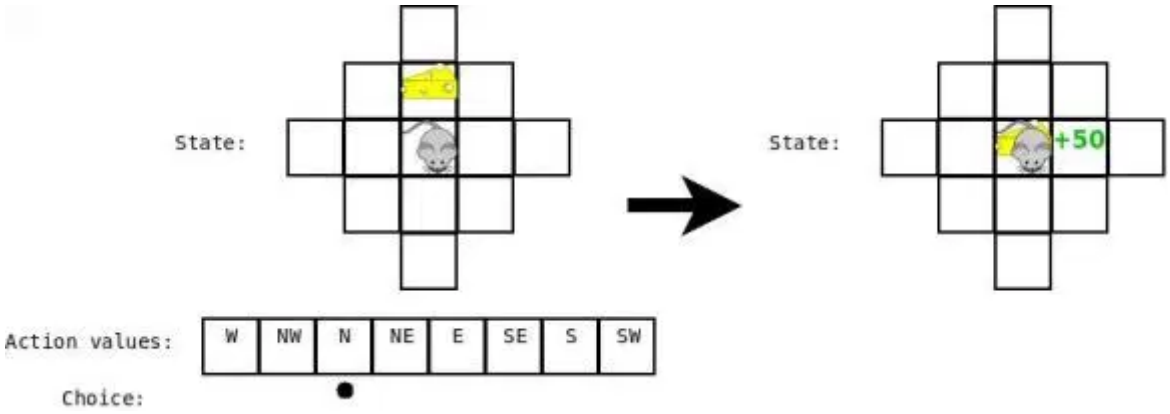
For those unfamiliar, the basic gist of Q-learning is that you have a representation of the environmental states `s`, and possible actions in those states `a`, and you learn the value of each of those actions in each of those states. Intuitively, this value, `Q`, is referred to as the state-action value. So in Q-learning you start by setting all your state-action values to 0 (this isn't always the case, but in this simple implementation it will be), and you go around and explore the state-action space. After you try an action in a state, you evaluate the state that it has lead to. If it has lead to an undesirable outcome you reduce the `Q` value (or weight) of that action from that state so that other actions will have a greater value and be chosen instead the next time you're in that state. Similarly, if you're rewarded for taking a particular action, the weight of that action for that state is increased, so you're more likely to choose it again the next time you're in that state. Importantly, when you update `Q`, you're updating it for the *previous* state-action combination. You can only update `Q` *after* you've seen what results.

Let's look at an example in the cat vs mouse case, where you are the mouse. You were in the state where the cat was in front of you, and you chose to go forward into the cat. This caused you to get eaten, so now reduce the weight of that action for that state, so that the next time the cat is in front of you you won't choose to go forward you might choose to go to the side or away from the cat instead (you are a mouse with respawning powers). Note that this doesn't reduce the value of moving forward when there is no cat in front of you, the value for 'move forward' is only reduced in the situation that there's a cat in front of you. In the opposite case, if there's cheese in front of you and

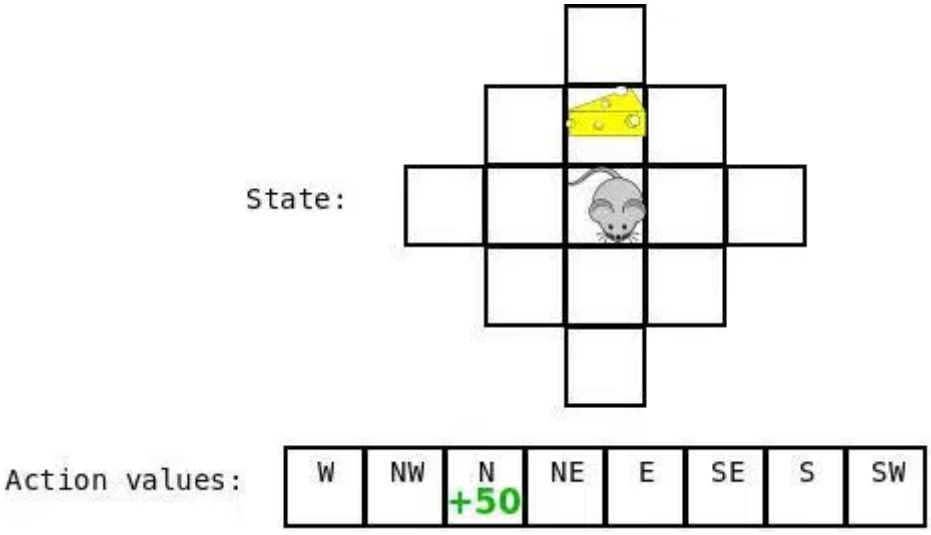
you choose 'move forward' you get rewarded. So now the next time you're in the situation (state) that there's cheese in front of you, the action 'move forward' is more likely to be chosen, because last time you chose it you were rewarded.

Now this system, as is, gives you no foresight further than one time step. Not incredibly useful and clearly too limited to be a real strategy employed in biological systems. Something that we can do to make this more useful is include a look-ahead value. The look-ahead works as follows. When we're updating a given `Q` value for the state-action combination we just experienced, we do a search over all the `Q` values for the state the we ended up in. We find the maximum state-action value in *this* state, and incorporate that into our update of the `Q` value representing the state-action combination we just experienced.

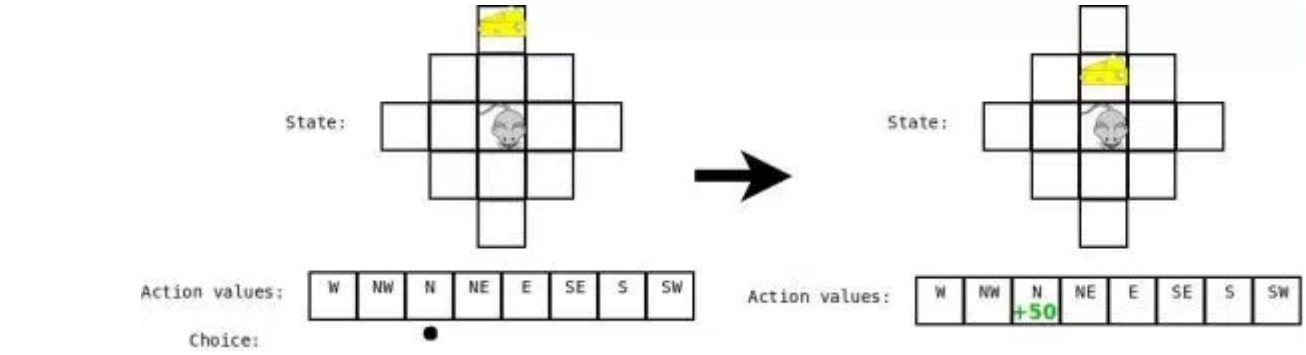
For example, we're a mouse again. Our state is that cheese is one step ahead, and we haven't learned anything yet (blank value in the action box represents 0). So we randomly choose an action and we happen to choose 'move forward'.



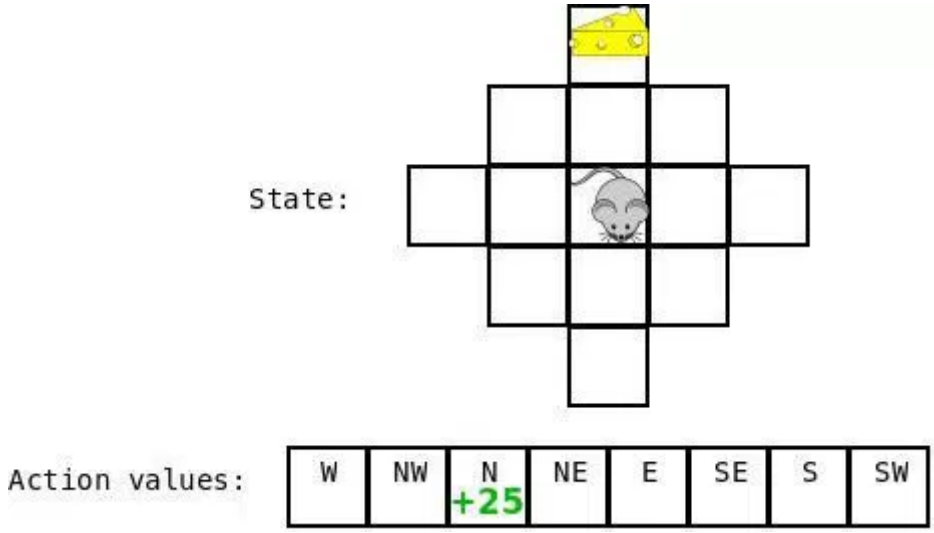
(https://studywolf.files.wordpress.com/2012/11/q-move-into-cheese.jpeg)
Now, in this state (we are on top of cheese) we are rewarded, and so we go back and update the `Q` value for the state 'cheese is one step ahead' and the action 'move forward' and increase the value of 'move forward' for that state.



(https://studywolf.files.wordpress.com/2012/11/q-cheese-ahead-updated.jpeg)
Now let's say the cheese is moved and we're moving around again, now we're in the state 'cheese is two steps ahead', and we make a move and end up in the state 'cheese is one step ahead'.



(https://studywolf.files.wordpress.com/2012/11/q-move-near-cheese.jpeg)
Now when we are updating the `Q` value for the previous state-action combination we look at all the `Q` values for the state 'cheese is one step ahead'. We see that one of these values is high (this being for the action 'move forward') and this value is incorporated in the update of the previous state-action combination.



(https://studywolf.files.wordpress.com/2012/11/q-cheese-near-updated.jpeg)
Specifically we're updating the previous state-action value using the equation: `Q(s, a) += alpha * (reward(s,a) + max(Q(s') - Q(s,a))` where `s` is the previous state, `a` is the previous action, `s'` is the current state, and `alpha` is the discount factor (set to .5 here).

Intuitively, the change in the Q-value for performing action `a` in state `s` is the difference between the actual reward (`reward(s,a) + max(Q(s'))`) and the expected reward (`Q(s,a)`) multiplied by a learning rate, `alpha`. You can think of this as a kind of PD control, driving your system to the target, which is in this case the correct Q-value.

Here, we evaluate the reward of moving ahead when the cheese is two steps ahead as the reward for moving into that state (0), plus the reward of the best action from that state (moving into the cheese +50), minus the expected value of that state (0), multiplied by our learning rate (.5) = +25.

Exploration
In the most straightforward implementation of Q-learning, state-action values are stored in a look-up table. So we have a giant table, which is size `N x M`, where `N` is the number of different possible states, and `M` is the number of different possible actions. So then at decision time we simply go to that table, look up the corresponding action values for that state, and choose the maximum, in equation form:

```
1  def choose_action(self, state):
2      q = [self.getQ(state, a) for a in self.actions]
3      maxQ = max(q)
4      action = self.actions[maxQ]
5      return action
```

Almost. There are a couple of additional things we need to add. First, we need to cover the case where there are several actions that all have the same value. So to do that, if there are several with the same value, randomly choose one of them.

```
def choose_action(self, state):
    q = [self.getQ(state, a) for a in self.actions]
    maxQ = max(q)
    count = q.count(maxQ)
    if count > 1:
        best = [i for i in range(len(self.actions)) if q[i] == maxQ]
        i = random.choice(best)
    else:
        i = q.index(maxQ)

    action = self.actions[i]
    return action
```

This helps us out of that situation, but now if we ever happen upon a decent option, we'll always choose that one in the future, even if there is a way better option available. To overcome this, we're going to need to introduce exploration. The standard way to get exploration is to introduce an additional term, `epsilon`. We then randomly generate a value, and if that value is less than `epsilon`, a random action is chosen, instead of following our normal tactic of choosing the max `Q` value.

```
def choose_action(self, state):
    if random.random() < self.epsilon: # exploration action = random.choice(
            best = [i for i in range(len(self.actions)) if q[i] == maxQ]
            i = random.choice(best)
        else:
            i = q.index(maxQ)

       action = self.actions[i]
    return action
```

The problem now is that we even after we've explored all the options and we know for sure the best option, we still sometimes choose a random action; the exploration doesn't turn off. There are a number of ways to overcome this, most involving manually adjusting `epsilon` as the mouse learns, so that it explores less and less as time passes and it has presumably learned the best actions for the majority of situations. I'm not a big fan of this, so instead I've implemented exploration the following way: If the randomly generated value is less than `epsilon`, then randomly add values to the `Q` values for this state, scaled by the maximum `Q` value of this state. In this way, exploration is added, but you're still using your learned `Q` values as a basis for choosing your action, rather than just randomly choosing an action completely at random.

```
def choose_action(self, state):
    q = [self.getQ(state, a) for a in self.actions]
    maxQ = max(q)

    if random.random()  1:
        best = [i for i in range(len(self.actions)) if q[i] == maxQ]
        i = random.choice(best)
    else:
        i = q.index(maxQ)

    action = self.actions[i]

    return action
```

Very pleasingly, you get results comparable to the case where you perform lots of learning and then set `epsilon = 0` to turn off random movements. Let's look at them!

Simulation results

So here, the simulation is set up such that there is a mouse, cheese, and a cat all inside a room. The mouse then learns over a number of trials to avoid the cat and get the cheese. Printing out the results after every 1,000 time steps, for the standard learning case where you reduce `epsilon` as you learn the results are:

```
 1  10000, e: 0.09, W: 44, L: 778
 2  20000, e: 0.08, W: 39, L: 617
 3  30000, e: 0.07, W: 47, L: 437
 4  40000, e: 0.06, W: 33, L: 376
 5  50000, e: 0.05, W: 35, L: 316
 6  60000, e: 0.04, W: 36, L: 285
 7  70000, e: 0.03, W: 33, L: 255
 8  80000, e: 0.02, W: 31, L: 179
 9  90000, e: 0.01, W: 35, L: 152
10  100000, e: 0.00, W: 44, L: 130
11  110000, e: 0.00, W: 28, L: 90
12  120000, e: 0.00, W: 17, L: 65
13  130000, e: 0.00, W: 40, L: 117
14  140000, e: 0.00, W: 56, L: 86
15  150000, e: 0.00, W: 37, L: 77
```

For comparison now, here are the results when `epsilon` is not reduced in the standard exploration case:

```
 1  10000, e: 0.10, W: 53, L: 836
 2  20000, e: 0.10, W: 69, L: 623
 3  30000, e: 0.10, W: 33, L: 452
 4  40000, e: 0.10, W: 32, L: 408
 5  50000, e: 0.10, W: 57, L: 397
 6  60000, e: 0.10, W: 41, L: 326
 7  70000, e: 0.10, W: 40, L: 317
 8  80000, e: 0.10, W: 47, L: 341
 9  90000, e: 0.10, W: 47, L: 309
10  100000, e: 0.10, W: 54, L: 251
11  110000, e: 0.10, W: 35, L: 278
12  120000, e: 0.10, W: 61, L: 278
13  130000, e: 0.10, W: 45, L: 295
14  140000, e: 0.10, W: 67, L: 283
15  150000, e: 0.10, W: 50, L: 304
```

As you can see, the performance now converges around 300, instead of 100. Not great.
Now here are the results from the alternative implementation of exploration, where `epsilon` is held constant:

```
10000, e: 0.10, W: 65, L: 805
20000, e: 0.10, W: 36, L: 516
30000, e: 0.10, W: 50, L: 417
40000, e: 0.10, W: 38, L: 310
50000, e: 0.10, W: 39, L: 247
60000, e: 0.10, W: 31, L: 225
70000, e: 0.10, W: 23, L: 181
80000, e: 0.10, W: 34, L: 159
90000, e: 0.10, W: 36, L: 137
100000, e: 0.10, W: 35, L: 138
110000, e: 0.10, W: 42, L: 136
120000, e: 0.10, W: 24, L: 99
130000, e: 0.10, W: 52, L: 106
140000, e: 0.10, W: 22, L: 88
150000, e: 0.10, W: 29, L: 101
```

And again we get that nice convergence to 100, but this time without having to manually modify `epsilon`. Which is pretty baller.

Code

And that's it! There is of course lots and lots of other facets of exploration to discuss, but this is just a brief discussion. If you'd like the code for the cat vs mouse and the alternative exploration you can access them at my github: Cat vs mouse – exploration (https://github.com/studywolf/blog/tree/master/RL/Cat%20vs%20Mouse%20exploration).
To alternate between the different types of exploration, change the `import` statement at the top of the `egoMouseLook.py` to be either `import qlearn` for the standard exploration method, or `import qlearn_mod_random as qlearn` to test the alternative method. To have `epsilon` reduce in value as time goes, you can uncomment the lines 142-145.

Tagged exploration, Python, Reinfocement learning

52 thoughts on "Reinforcement learning part 1: Q-learning and exploration"

1. *Reinforcement Learning: SARSA vs Q-learning | studywolf* says:
   July 1, 2013 at 9:50 pm
   […] my previous post about reinforcement learning I talked about Q-learning, and how that works in the context of a cat vs mouse game. I mentioned in […]

   Reply
   myasir68 says:

   September 9, 2013 at 2:46 am

Hi,
I am Phd student and I am looking to understand the concept of Qlearning in non-deterministic environment. Do you know any website or tutorial that could help in understanding by the putting reinforcement learning on an example. So that it would clear my understanding, Waiting for your positive and prompt reply

Reply

travisdewolf says:

September 11, 2013 at 12:13 pm

Hello,

I think I might have a Matlab tutorial that would be of interest to you, if you email me I can forward the file to you!

Reply

p says:

September 16, 2013 at 10:11 am

Hi travisdewolf,

i would be interested in the matlab tutorial for non-deterministic qlearning as well!. I would be grateful if you could send it at my email.
best wishes!

travisdewolf says:

September 16, 2013 at 10:21 am

Sent!

Nead says:

December 5, 2013 at 6:02 pm

Hi travisdewolf,
Could you send the tutorial to me, please?
Thanks

travisdewolf says:

December 6, 2013 at 2:48 pm

Sent!

dave says:

January 14, 2014 at 12:59 am

can i have it too? this is a great tutorial btw!

linkid says:

April 23, 2015 at 1:44 pm

Hi, could you send to my email too? nklinh91@gmail.com (matlabcode)

yashi21 says:

July 1, 2015 at 5:36 am

Hey can u plz send the tutorial in my id too?ASAP

jewa says:

December 16, 2013 at 12:43 pm

I have just stared to explore Q-learning, may this tutorial helpful to me, Thanks!

Reply

Yan King Yin says:

January 13, 2014 at 4:47 am

Thanks, this is nice for beginning to explore RL, without dealing with its full complexity at once. The GUI is neat. I'd like to fork a branch of it on Github, add some comments to the code, etc. But your repo contains other directories which makes forking a bit inconvenient. Anyway I have copied the code manually. Thanks again!

Reply

travisdewolf says:

January 13, 2014 at 11:25 am

Ah yeah, sorry about that! But glad you're finding the tutorial / code useful. Please keep me updated on your work with it! I'd be interested to see any extensions.
Thanks!

Reply

raju says:

February 2, 2014 at 11:10 pm

could you send me the tutorial .I am doing my research using RL

Reply

Fatemeh Jahedpari says:

February 10, 2014 at 5:08 pm

Many thanks, it was great.

Reply

Fatemeh Jahedpari says:

February 10, 2014 at 5:12 pm

Can I have that Matlab tutorial, as well?
Cheers

Reply

Fatemeh Jahedpari says:

February 11, 2014 at 8:04 am

BTW, it seems that the right side of figure labeled "Q moves near cheese" is not correct. I think both cheese and mouse should be placed one place upper than their current places.

Reply

travisdewolf says:

February 11, 2014 at 12:29 pm

Ah! That would be the case if the map was representing things in an allocentric (world or absolute coordinates) way, but here everything is displayed relative to the mouse (egocentricly). When the mouse moves up the cheese is now only one square away from him, and the figure reflects this. I'm actually working on another post about the difference between allocentric and egocentric, I'll let you know when it's up!

Reply

Fatemeh Jahedpari says:

February 12, 2014 at 4:31 am

Thank you

bandanashop says:

March 28, 2014 at 4:48 am

Excellent tutorial and introduction to a potentially complicated subject – well done!

Reply

Suparna says:

July 3, 2014 at 9:42 am

Could you please send me the Matlab tutorial, I am doing my thesis using RL. Thanks

Reply

AndyPark says:

July 9, 2014 at 8:25 am

Hi, it is excellent material! If it is not too late, can you also send me the matlab tutorial? Thanks!

Reply

travisdewolf says:

July 9, 2014 at 12:45 pm

Sure can, just send me your email address!

Reply

Stromaea says:

October 21, 2014 at 4:12 pm

Hi, I would be interested in the Matlab tutorial as well! I would be grateful if you could send it to my email. I sent you my request to your e-mail. Thanks!

Migel Tissera says:

August 13, 2014 at 11:19 pm

Hi Travis,
Great work! I'm a PhD student at University of South Australia, and have looked at SPAUN and your Python implementation of it too.
Is it possible to get your MATLAB code for this?
Thanks,

Reply

Maigha says:

December 10, 2014 at 12:09 pm

Hello!
Could I please get for the Matlab tutorial on non-deterministic Q-learning?
Thanks!

Reply

dz says:

March 17, 2015 at 12:27 pm

Hi travisdewolf,

Your explanation is really good. I am really novice in q-learning and was confused about how to set the exploration value (epsilon). If it does not inconvenience you, could you send me the MATLAB tutorial as well, please?

Thank you.

Reply

15. *1p – Reinforcement Learning – Q-Learning and Exploration | Exploding Ads* says:
May 14, 2015 at 12:34 am
[…] https://studywolf.wordpress.com/2012/11/25/reinforcement-learning-q-learning-and-exploration/ […]

Reply

Zeeshan says:

July 1, 2015 at 10:23 am

please send me the matlab tutorial for Qlearning in non-deterministic environment. Thanks

Reply

Taeyoung says:

July 8, 2015 at 6:29 pm

Reblogged this on Taeyoung Kim and commented:
Thing to learn more

Reply

Ria Candrawati says:

October 4, 2015 at 12:14 am

Sir, I am Phd student and I am also looking for the concept of Qlearning in non-deterministic environment. can you please sent the tutorial to me.. thanks

Reply

Erin says:

October 12, 2015 at 11:14 am

First of all, great tutorial, super useful explanations! Thank you very much for sharing.
Secondly, in q-move-into-cheese.jpeg, the mouse actually moves upward, so he lands where the cheese was… I think you moved the cheese into the mouse's tile. Or I am missing something?

Please continue posting, your skills are helping us a lot!
Erin

Reply

Erin says:

October 12, 2015 at 11:16 am

Nevermind! I guess that diamond shows the current position of the mouse and its surrounding tiles.

Reply

travisdewolf says:

October 12, 2015 at 9:08 pm

Hi Erin,

yup you're right! That's the world from the mouse's point of view🙂
Glad you found the tutorial useful!

Yucheng says:

November 19, 2015 at 9:40 pm

Hi!
Could you please send me the matlab tutorial please?
Thanks a lot!

Reply

haz says:

November 22, 2015 at 12:37 am

Is that possible to give dynamic rewards in Q-learning?

Reply

travisdewolf says:

November 22, 2015 at 11:51 am

Hi Haz,

yup. One of the most basic tasks you can do with RL is the bandit task (https://en.wikipedia.org/wiki/Multi-armed_bandit) and the reward changes over time. The speed of convergence depends on your learning rate. If you have a deterministic reward (i.e. not probabilistic) then the amount of noise in the system will determine how high you can crank this. The more noise in your reward the lower you need to keep the learning rate because you don't want to fully change your Q-value based on noise. Hope that helps!

Reply

haz says:

November 22, 2015 at 6:41 pm

Thank you very much

haz says:

November 22, 2015 at 7:21 pm

Hi Travis,
Thank you very much for your explanation on my earlier question.
I got another concern regarding Q-learning.
For an example,I have to navigate a robot to reach a specific point. But i have time constraints, where robot needs to reach the point within specific time period(e.g.10sec).
How and at which point that i need incorporate this constrain in my Q-learning algorithm?

Thanks in advance !

Reply

travisdewolf says:

November 24, 2015 at 12:06 pm

Hi Haz,

if you want to address this kind of problem with Q-learning, then you can do things like just restart the simulation after 10 seconds have gone by, or move the agent back to the start (but that can introduce some undesired artifacts). Q-learning normall assumes an infinite-horizon, but there are variations like average reward RL that remove time dependency (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.3423&rep=rep1&type=pdf).

If you're not tied to Q-learning, then in general for a finite horizon control problem I would use something like iterative linear quadratic Gaussian control (iLQG; http://maeresearch.ucsd.edu/skelton/publications/weiwei_ilqg_CDC43.pdf) or policy improvement with path integrals (PI2; http://www.jmlr.org/papers/volume11/theodorou10a/theodorou10a.pdf).
Hope that helps!

Reply

szerzp says:

December 12, 2015 at 10:56 am

Hi Travis,

Great tutorial here. I have a little confusion about the code tho: for printV function in qlearn.py, a single state can be split in form of x, y, a two element tuple. Whereas I looked into egoMouseLook.py, the state is encoded as a tuple of cellValues, which is in turn dependent on cells in lookcells. Number of elements in lookcells is definitely more than 2 according to your implementation, so the state should be encoded as a tuple of more than 2 elements. So why is the split x,y in qlearn.py possible? Wouldn't that lead to an error in unpacking of tuple?

Thanks

Reply

travisdewolf says:

December 14, 2015 at 1:07 pm

Hello!

Oh my that is a holdover print function from an allocentric based version of q-learning, sorry about that! You're the first to catch that one. Those functions aren't actually used anywhere, which is why no errors threw. I've taken them out now, thanks for the catch!

Reply

haz says:

February 8, 2016 at 5:56 pm

Hi Travis,

I got a very basic question about training and implementation of the Q-learning agent.
Initially, we need to train the agent in a simulation environment. So, what are the factors we need to consider at the training phase?
Do we normally train agents in single simulation with varying the environmental conditions (E.g. changing the arrival time of the agent to the system DURING SINGLE SIMULATION using few agents)

OR
Do we need to run separate simulations for a single agent, varying the arrival time to the system?

Thanks in advance !

Reply

travisdewolf says:

March 1, 2016 at 9:07 pm

Hi Haz,

sorry about the delay in replying! This got buried. There's definitely no hard and fast rules about best practices, it can vary greatly depending on the specific problem.

I'm not sure I understand what your context is for 'varying the arrival time to the system', could you elaborate on your set up? And how you would change the time an agent arrives in the system inside a single simulation? In general, depending on the problem, and what you want to train specifically, either running a single long simulation to train or a bunch of shorter ones can be appropriate.

Reply

shivaji says:

March 27, 2016 at 2:55 pm

HI ,Iam working on rehab robotics using machine learning techniques…can u please share matlab reinforcement learning example/tutorial

Reply

Muhammad Awais Jadoon says:

April 6, 2016 at 1:22 am

Hi, I've recently started exploring machine learning and then RL. I would much appreciate if you can help me with MATLAB Code/Tutorial for Q-Learning. I'm actually trying to use it in wireless communication area but right now I've no idea how to use it in practice.

Thank you

Reply

travisdewolf says:

April 22, 2016 at 7:51 am

Hello!
I'm sorry looks like I somehow missed this comment, if you send me an email tdewolf@uwaterloo.ca with some more details I can try to help you out.

Reply

Pogo says:

April 22, 2016 at 7:43 am

Hello! It seems like there's some code missing from the last code box (after random.random()). Can you update if possible?

Reply

travisdewolf says:

April 22, 2016 at 7:48 am

hmm, the last code box looks like this for me:

```
1  def choose_action(self, state):
2      q = [self.getQ(state, a) for a in self.actions]
3      maxQ = max(q)
4
5      if random.random()  1:
6          best = [i for i in range(len(self.actions)) if q[i] == maxQ]
7          i = random.choice(best)
8      else:
9          i = q.index(maxQ)
10
11     action = self.actions[i]
12
13     return action
```

I'm not sure why it wouldn't appear in full for you! But you could also check out the code on my github if it's not displaying for you here!

Reply

Pogo says:

April 22, 2016 at 7:59 am

Sorry! I wasn't clear. Yes, that's what I see, but there are several lines missing between random.random() and the next character 1:

Found it on github, thank you!

haz says:

August 17, 2016 at 12:04 am

Hi guys,
Does Reinforcement Learning(Q-learning) using lookup table (instead of function approximation) is EQUAL to Dynamic programming ?

Thanks in advance!

Reply

travisdewolf says:

August 26, 2016 at 3:13 pm

Hey Haz,

Dynamic programming is really more of an implementation detail, where you store the solution to sub-problems in a lookup table. So RL can be implemented using dynamic programming, but you wouldn't say RL with a lookup table is the same as dynamic programming, does that help?

Reply