Trading

From RL for Trading Thesis paper

At each time step *t* the agent has an observation vector *ot* = (*ht* ,*mt* , *et*) where *ht*

is the agent’s holding state, *mt* is a market observation and *et* is a binary variable

indicating whether *t* is the last time step in an an episode. This binary variable is

included because the agent is forced to close its position at the last time step in an

episode, and so the available actions are different when *et* = 1.

**the instructors' answer,**

*where instructors collectively construct a single answer*

Pick 3 indicators, use 10 discrete values for each = 1000 states.  I'd try bollinger, and two other factors.

 I would advise you to have a richer state with more resolution on the values of BB, momentum, and perhaps one or two other indicators.  These will help the learner infer when the escalator is going to go up.  I would also make sure that holding, and “cumu return since holding” are additional components of the state.

You should freeze your policy from the training period and use it for the testing period.

 just make calls to the querysetstate function and do no further Q Updates on the testing period

if you want to be able to learn the BB strategy you need to give your learner more information.  Imagine yourself as the learner, suppose I tell you your state is:

BB = 0

Holding = 0

You don't have enough information.  If you encoded the BB value with more resolution, and add momentum, the learner would be able to learn to BUY when the price is outside the lower band and headed back up.

You also need more info to help it know when to exit.  the exits are defined when the price moves back through the SMA and you don't have that encoded.

https://piazza.com/class/idadrtx18nie1?cid=1643

 proper discretization serves as a normalization.

[**mmueller36**](https://piazza.com/class/idadrtx18nie1?cid=1643) [10 hours ago](https://piazza.com/class/idadrtx18nie1?cid=1643)

Discretized indicators as states:  Makes sense and is easily computable.

https://d1b10bmlvqabco.cloudfront.net/photos/grkyh3conwv/1412358728_35.png

[**Tucker Balch**](https://piazza.com/class/idadrtx18nie1?cid=1643) [2 hours ago](https://piazza.com/class/idadrtx18nie1?cid=1643)

Meaning of going from s to s' depends on how you define state.

https://d1b10bmlvqabco.cloudfront.net/photos/i4n69zc1iFe/1440209905_35.png

[**mmueller36**](https://piazza.com/class/idadrtx18nie1?cid=1643) [1 hour ago](https://piazza.com/class/idadrtx18nie1?cid=1643)

Suppose a state is a discretized set of indicators.  For simplicity, let's we pick just ONE:  20 day bollinger bands, discretized to be -1 if the stock price exceeds "-2" (rolling\_mean - 2\*rolling\_std), +1 if stock price exceeds "+2", and zero in between.  (Discretized to only 3 factors, speaking R, or states in Q-learning).    
  
I'm following a B-Band strategy as discussed in the lecture, so I buy the stock after it crosses back over from the state s=-1 to the s'=0.  I've performed an ACTION a, but my action has not caused the state to make the transition.  The transition happens all by itself due to the market.  So now let's redefine an action as something that the market does to move my stock indicator.  around between the three BB states defined above.

Then how do I buy, sell and do nothing (within the context of a Q learner) if states and actions are already defined?

https://d1b10bmlvqabco.cloudfront.net/photos/i4n69zc1iFe/1440209905_35.png

[**mmueller36**](https://piazza.com/class/idadrtx18nie1?cid=1643) [43 minutes ago](https://piazza.com/class/idadrtx18nie1?cid=1643)

The other thread provides some hints:  The states can be position,indicators.  So for 3 indicator states (-1,0,1) and 3 types of position (buy,nothing,sell) there are 9 combinations:  (-1,buy),(-1,nothing),(-2,sell),(0,buy),...

The Q table is then a 9x9 matrix, with the market moving between some states, and a buy,sell moving between others. I think I'm getting there...

https://d1b10bmlvqabco.cloudfront.net/photos/grkyh3conwv/1412358728_35.png

[**Tucker Balch**](https://piazza.com/class/idadrtx18nie1?cid=1643) [6 minutes ago](https://piazza.com/class/idadrtx18nie1?cid=1643)

You should have as factors things like: Holding, not holding, cumulative return since holding.

**Trading Notes**

States:  Long, zero, short.

Actions:  Buy, Sell, none.

In addition to the position based states, you'll want states based on your technical indicators.You can discretize these states.

You need some indicators in there -- e.g. bollinger, momentum, etc.

rewards are daily return for each day of holding a position

s,a,s',r tuple is:  s (current position), a (possible action) s' (next state) and r (profit from closing position?)

You're going to be iterating over time-series data and making state transitions.  The "states" are based on the technical indicators you choose.

 You should have about 5 components of your state, so it's a 5 dimensional world.

**e instructors' answer,**

*where instructors collectively construct a single answer*

Pick 3 indicators, use 10 discrete values for each = 1000 states.  I'd try bollinger, and two other factors.

Meaning of going from s to s' depends on how you define state.

Suppose a state is a discretized set of indicators.  For simplicity, let's we pick just ONE:  20 day bollinger bands, discretized to be -1 if the stock price exceeds "-2" (rolling\_mean - 2\*rolling\_std), +1 if stock price exceeds "+2", and zero in between.  (Discretized to only 3 factors, speaking R, or states in Q-learning).    
  
I'm following a B-Band strategy as discussed in the lecture, so I buy the stock after it crosses back over from the state s=-1 to the s'=0.  I've performed an ACTION a, but my action has not caused the state to make the transition.  The transition happens all by itself due to the market.  So now let's redefine an action as something that the market does to move my stock indicator.  around between the three BB states defined above.

Then how do I buy, sell and do nothing (within the context of a Q learner) if states and actions are already defined?

 The states can be position,indicators.  So for 3 indicator states (-1,0,1) and 3 types of position (buy,nothing,sell) there are 9 combinations:  (-1,buy),(-1,nothing),(-2,sell),(0,buy),...

The Q table is then a 9x9 matrix, with the market moving between some states, and a buy,sell moving between others.

[**Tucker Balch**](https://piazza.com/class/idadrtx18nie1?cid=1643) [5 hours ago](https://piazza.com/class/idadrtx18nie1?cid=1643)

You should have as factors things like: Holding, not holding, cumulative return since holding.

**Update Q table**

**Q:**

Q[s', argmaxa'(Q[s', a'])]

argm = q[s,a].argmax()

q[s\_prime, q[s\_prime, argm].argmax()

A:

Note that for the future returns:

                          α · (r + γ · Q[s', argmaxa'(Q[s', a'])])

That argmax(Q[s', a']) is  is the action that maximizes the Q-value among all possible actions a' from s',

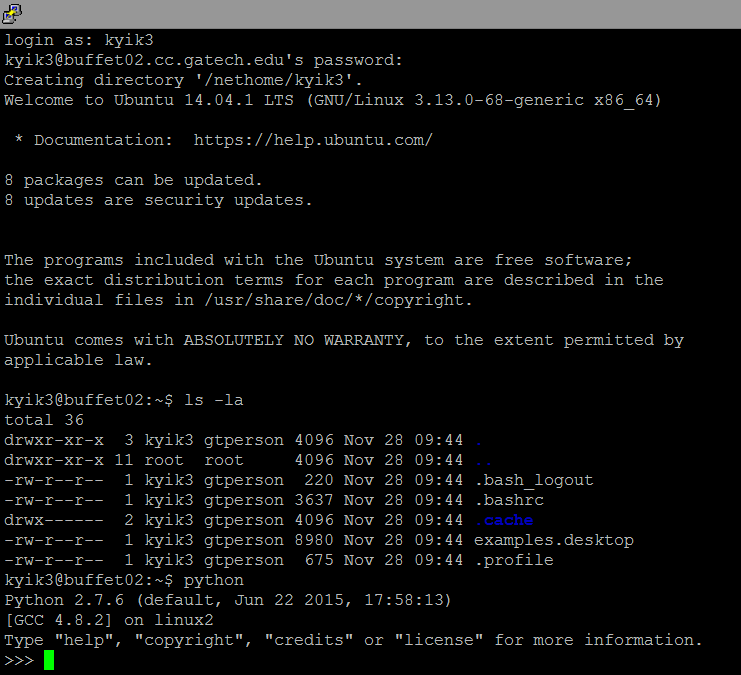
so use : instead of a' because we want to look at all values.

**Convergence**

After 500 iterations, your code should solve world01 in less than 23 steps.

How to check for platform requirements in MC3-Project-3

Just a quick note: this can be checked by logging into ssh of this server. It is the same server as mentioned is the project page.



N-S buffet03.cc.gatech.edu

If really no luck, try connecting on top of VPN.

**the instructors' answer,**

*where instructors collectively construct a single answer*

If you start at 9,9 (state 99) and take action 0, your new state will be 8,9 (state 89).

>> 'r' will be  '-1' throughout until convergence for each iteration of the Q table.

r is -1 until the robot reaches the goal.  It will take many trips to the goal until the policy converges.

MC3-P3 clarifications

 1) How will you be giving us the states (s) when you query the learner?

It will be an integer.

The learner should build a model of T and R to support Dyna updates.

But I don't think you need to worry too deeply about inference.  If you implement Q-Learning correctly this will happen naturally.

**Q table**

ndarray is probably the simplest and fastest container to use. It holds |S| x |A| entries, which is the number of states times the number of actions. It's a simple lookup table where

*Q*[*si*,*aj*]

**gives the expected value of taking action**

*aj*

 when in state

*si*

.

**~ An instructor (Tucker Balch) endorsed this answer  ~**

[**Tucker Balch**](https://piazza.com/class/idadrtx18nie1?cid=1488) [21 hours ago](https://piazza.com/class/idadrtx18nie1?cid=1488)

Here's how I instantiated Q in my solution

self.Q = rand.uniform(low=-1.0,high=1.0,size=(num\_states,num\_actions))

**querysetstate()**

querysetstate() should choose a random action with probability rar, otherwise it should follow the policy based on the Q-table.

First you throw the dice to see if there will be a random choice.  If not, you choose the action with the highest Q value.  If there is a random choice, then the probability of each is 0.25.

# Help Understanding the QLearner

These are my elementary thoughts. Can anyone confirm or redirect? Thank you!

1. The learner begins with a Q table that maps a particular state, and an action taken from that state, to the expected reward. Initially, all the expected rewards are random values. Our goal here is to update these expected rewards correctly. **Will every value in the table eventually be reset? That is to say, will the learner cover every state action pair, in the course of all its iterations?**

1. A: Obstacles are examples of states that will not be touched, in vanilla Q-learning at least, because testqlearner will not permit the robot to move to these states. With Dyna-Q, because you're making a bunch of hallucinations using random states and actions and you don't know about the concept of obstacles, it's very improbable that you'll leave any state untouched, unless you imagine absurdly large state spaces or something. Every action should be touched no matter what, though; rar is meant to ensure that.

. Technically, the convergence guarantees that Q-learning provides require that ALL states be visited infinitely often.  That doesn't happen in reality of course.  It is quite likely that some states will not be visited at all.

2. The learner maintains self.s and self.a. Self.s refers to the most recent state it was in; self.a refers to the most recent action taken.

Yes

3. The learner receives information from the world: rewards and next state

4. Initially, the learner's starting state is set by the method query set state. This sets self.s to an initial state. It does not, however, update the qTable.

5. querysetstate returns an action. This is a randomly generated action.

 querysetstate() should also choose an action according to the Q table.

6. So, from self.s, the learner will take action a. Now here is where I'm stuck: **how do we know what is the resultant state of taking action a from state s? In other words, how is s' derived?**

In the real world, we'd get s' from some sort of sensor on the robot. (Also, in the real world, there would usually be some kind of noise involved here, but thankfully that is left out of this assignment.) In testqlearner, we simulate that by moving our robot around on a map, and we enforce things like bounds constraints and obstacles ourselves.

You learn the next state and reward when query() is called again.

7. It seems that the query function takes s' and r. S' is the new state;

r is the reward for taking action a in state s.

- **what was the action that took us from self.s to s'?**

It is the action returned by query() or querysetstate() when it was last called.

**Was it the action derived from the previous call of the query function? In other words -- which field in the q table is being updated here?**

 If you implemented everything correctly, when you first call query, self.a is the action that took us from self.s to s\_prime. The action that we generate, either randomly or by looking at Q, is what's in the lectures as a'. That action will, before we leave query, become self.a, just as s\_prime will become self.s.

DYNA

the first thing we need to do is get T and we do so by T\_c. We set the T\_c array to a very low value in the constructor. There is then a formula on the Udacity videos to compute T, it is: T[s,a,s'] = T\_c[s,a,s']/sum(T\_c[s,a,:]). T\_c[s,a,s'] is how many times we have been in state s' and sum(T\_c[s,a,:]) is the sum of all the times we have been in state s and taken action a. This is how we calculate T. Dyna is only implemented in the query function. We also need R[s,a] = r. Now we loop through the number of times we specified Dyna as and choose a random number between 0 and 1 and sum over T until we get to a sum that is >= the random number we generated. We then update the Q table with the <s,a,s',r> that we have generated. This is done dyna times.

# hint regarding Dyna

If you record all real experience tuples <s, a, s', r> then randomly sample from them to update the Q-table, the result is statistically equivalent to building a model of T and R, and sampling from them.

[**Tucker Balch**](https://piazza.com/class/idadrtx18nie1?cid=1618) [8 hours ago](https://piazza.com/class/idadrtx18nie1?cid=1618)

Update the q table, run dyna, select an action.  Not sure what I said in the video, but it is a good idea to run dyna before you select the action so that you can get the benefit of the improved Qtable.

# Performance tip for random s\_prime from T

Thought I'd share a tip. numpy.random.choice with probabilities has a performance bottleneck in older versions of numpy due to the way they validate that the probabilities sum to 1.0. The following slows down testqlearner.py significantly:

np.random.choice(self.states, p=T[s, a])

Instead, use this:

np.random.multinomial(1, T[s, a]).argmax()