Temporary Difference: Applications

Saturday, May 14, 2022 9:50 AM

We have now two codes that apply Q table learning using Temporary Difference method for two different GYM environments:

1) Mountain car
2) Cart Pole

1) Mountain car

The console output looks as follows:

```
e= 213 r_total= -200.0 r_MAX= -200.0 r_prom= -200.0 epsilon= 1
e= 214 r_total= -200.0 r_MAX= -200.0 r_prom= -200.0 epsilon= 1
e= 215 r_total= -200.0 r_MAX= -200.0 r_prom= -200.0 epsilon= 1
e= 216 r_total= -200.0 r_MAX= -200.0 r_prom= -200.0 epsilon= 1
e= 217 r_total= -200.0 r_MAX= -200.0 r_prom= -200.0 epsilon= 1
e= 218 r_total= -200.0 r_MAX= -200.0 r_prom= -200.0 epsilon= 1
e= 219 r_total= -200.0 r_MAX= -200.0 r_prom= -200.0 epsilon= 1
e= 220 r_total= -200.0 r_MAX= -200.0 r_prom= -200.0 epsilon= 1
e= 221 r_total= -200.0 r_MAX= -200.0 r_prom= -200.0 epsilon= 1
```

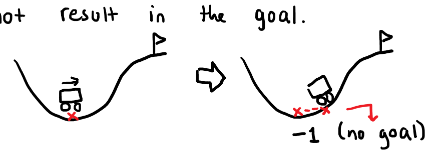
↳ current episode

episode: the car tries a maximum of 200 actions. If the car arrives the episode is over; if it can't arrive in 200 actions, the episode is over too.

→ The world punishes the agent with -1 for every action taken that does not result in reaching the goal.

r_total prints
↳ If the agent could not reach the goal in 200 steps = 200 punishes → reward = -200
↳ If the agent arrives to the goal (less than 200 steps in the episode) = less than 200 punishes = reward greater than -200 i.e. -137, etc.
↳ arriving to the goal means neither punish nor reward → r+=∅

r_max prints
↳ The maximum reward in all episodes so far: if there was a previous episode with greater r-prom (reward), it appears in following episodes as r_max
↳ a reward r-total = -200 in e = # means that in episode # the agent executed 200 actions that did not result in the goal.



-1 (no goal)

r_prom = -200 test show no learning
↳ r_prom: every 100 episodes, we test the agent in 20 episodes and compute the average of those 20 tests every 100th episode.

↳ we aim for greater reward prints every time.

→ How is the agent deciding the actions?

```
# table Q(s,a) initially with random numbers [0,1]
self.q  = np.random.rand(self.Nx+1, self.Nv+1, self.Na)
```

↳ The agent thus decides what to do with Q position (max)
Since initially with randoms in Q → random actions in the beginning
In agent:

```
def action(self, s, env):
    return self.q.accion(s)
```

→ the decision of action is Q which computes the max()

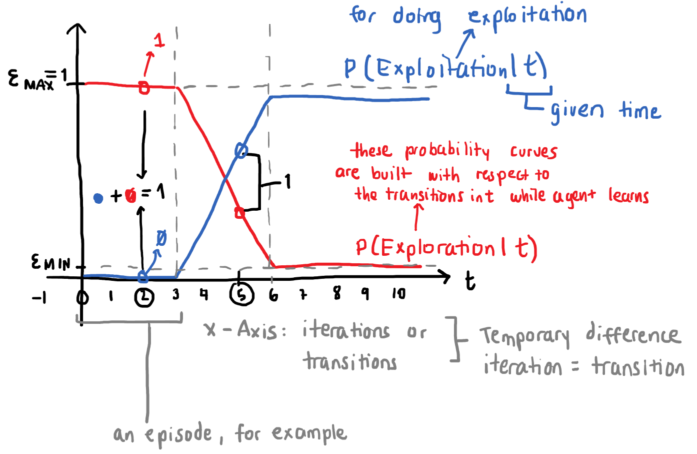
As learning progresses, the agent will start taking decisions with informed actions: the random Q values get adjusted better.

When the agent is LEARNING, the agent can take actions in TWO WAYS:

→ Exploration: When the agent does not know and takes random actions, the agent is exploring
↳ the agent, when taking random actions, is exploring the state space.
↳ Moves with no preference
→ We don't use Q really, we use random actions.

→ Exploitation: When the agent is taking actions based on the knowledge it has (Q table), we call it exploitation.
→ Once the agent knows the way to achieve max reward (Optimal Politic), it is time to take that path: exploitation.
→ We use Q which tells us what to do.

In the program, the agent always uses Q, but in the beginning is random → exploration. As learning goes on, exploration starts to shift to exploitation.
↳ We can thus define in functional form how much exploration and exploitation the agent does.
=
We define a probability distribution for those two concepts:



probability of exploration + probability of exploitation = 1

always: $\underbrace{P(\text{Exploration} \mid t) + P(\text{Exploitation} \mid t) = 1}$

all with respect to a particular transition (Conditioned to transition)
=
p(exploration) at point t + p(exploitation) at point t = 1

→ Exploration is how the agent takes an action

→ Say P(Expl|t) = 0.8 and P(Explr) = 0.2 that means
when agent wants a new action: 0.8 prob it is from Q
0.2 prob it is random

take a number: ← on that action taken [80% from Q
n > 0.8 → Rand 20% from random
n < 0.8 → Q

→ We don't know Q's real functional form ∄
a way to solve this is to define P(explr) and P(explt) curves.

→ In this example, 0 to 2 transition have P(explr) = 1

3 to 6 this prob changes

→ P(explr) does not reach 0, it reaches ε_{MIN}.
ε_{MAX} is maximum probability, or 1.

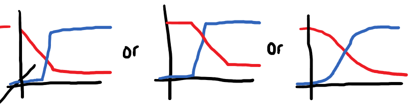
- P(explr) reaches ε_{MIN}
- P(explt) reaches ε_{MAX}

} because we want that when the agent learned, agent still takes random sometimes to adjust it: since agent very likely didn't explore all space and thus Q can still be $\underbrace{\text{adjusted}}$ a bit.

there may be other path that is better along the way.

→ What functional form should P(explt) and P(explr) have?

start in 1
put change it immediately



or

or

} could or could not be linear

we decide the slope

this is still the problem today

} This combination is experimental

But using these funcs guarantees convergence (learn)

→ if we $\underbrace{\text{always}}_{\text{P(explr)}}$ choose random action, the agent will never learn the goal. It might reach it, but never learn (Q-learning)

→ all this refers to Q but can also ask V.

When program finishes

```
e= 9999  r_total= -200.0  r_MAX= -115.0  r_prom= -199.65  epsilon= 1
```

-200.0: this episode never won

-115.0: Some episodes were won

-199.65: tests: learned a little → some will arrive some won't

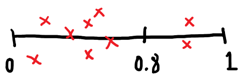
} You can increase the MAX-NUM-EP to 20,000 to improve this

After this we can test one episode: some will arrive some won't

→ One politic only, but initial conditions change (gym issue)

p(explr) = 0.2   p(explt) = 0.8   at current t

flip a coin (random num)



0   0.8   1

n < 0.8 → explt

n > 0.2 → explr

- If we never explore and always exploit, Q matrix will move it with a simple preference (not optimal) and with exploration, the Q gets updated with rewards and it spreads accross the matrix like a ripple = agent finds optimal

↳ when agent gets reward / punish, Q gets updated like a ripple