RL_Navie_Q_Week_2

October 20, 2020

1 Setup Environment

[1]:
```
import gym
import numpy as np
from gym import spaces
env = gym.make("CartPole-v0")
```

2 Warpper

Most gym environment have space in multi-dimension (Box format). To save memory, we apply a wrapper to discretize the observation.

env.unwrapped will give back the internal original environment object.

[2]:
```
class DiscretizedObservationWrapper(gym.ObservationWrapper):
    """This wrapper converts a Box observation into a single integer.
    """
    def __init__(self, env, n_bins=10, low=None, high=None):
        super().__init__(env)
        assert isinstance(env.observation_space, gym.spaces.Box)

        low = self.observation_space.low if low is None else low
        high = self.observation_space.high if high is None else high

        self.n_bins = n_bins
        self.val_bins = [np.linspace(l, h, n_bins + 1) for l, h in
                         zip(low, high)]
        self.observation_space = gym.spaces.Discrete(n_bins ** len(low))

    def _convert_to_one_number(self, digits):
        return sum([d * ((self.n_bins + 1) ** i) for i, d in enumerate(digits)])

    def observation(self, observation):
        digits = [np.digitize([x], bins)[0]
                  for x, bins in zip(observation, self.val_bins)]
        return self._convert_to_one_number(digits)
```

```
env = DiscretizedObservationWrapper(
    env,
    n_bins=8,
    low=[-2.4, -2.0, -0.42, -3.5],
    high=[2.4, 2.0, 0.42, 3.5]
)
```

3 Naive Q-Learning

The key point of Naive Q-learning is while estimating what is the next action, it does not follow the current policy but rather adopt the best Q value independently. Here is the Bellman equation we will use.

$Q(s,a) \leftarrow (1-)Q(s,a) + (r + * maxQ(s\ ,a\ ))$

3.1 Calculate and Update Q value

The Q value for all (s, a) pairs can be simply tracked in a dict.

[3]:
```
from collections import defaultdict

Q = defaultdict(float)

gamma = 0.99  # Discounting factor
alpha = 0.5  # soft update param
actions = range(env.action_space.n) # all possible actions

# Calculate Q Value
def update_Q(s, r, a, s_next, done):
    # s: observations
    # r: reward
    # a: actions
    # s_next: same as s
    # done: bool

    max_q_next = max([Q[s_next, a] for a in actions])  # get max Q value of␣
↪(s', a')

    # Do not include the next state's value if currently at the terminal state.
    Q[s, a] += alpha * (r + gamma * max_q_next * (1.0 - done) - Q[s, a]) # the␣
↪Bellman equation to update Q value.
    # If done is True, Q(s,a) = (1-alpha)*Q(s,a)
```

3.2 Pick Action

Action is decided based on max Q value and we use -greedy to force exploration.

[4]:
```
epsilon = 0.1  # 10% chances to apply a random action

def act(ob):
    if np.random.random() < epsilon:
        # action_space.sample() is a convenient function to get a random action
        # that is compatible with this given action space.
        return env.action_space.sample()

    # Pick the action with highest q value.
    qvals = {a: Q[ob, a] for a in actions}
    max_q = max(qvals.values())  # get best Q value
    # In case multiple actions have the same maximum q value.
    actions_with_max_q = [a for a, q in qvals.items() if q == max_q] # range(2);
 ↪ max_q
    return np.random.choice(actions_with_max_q)
```

3.3 Result

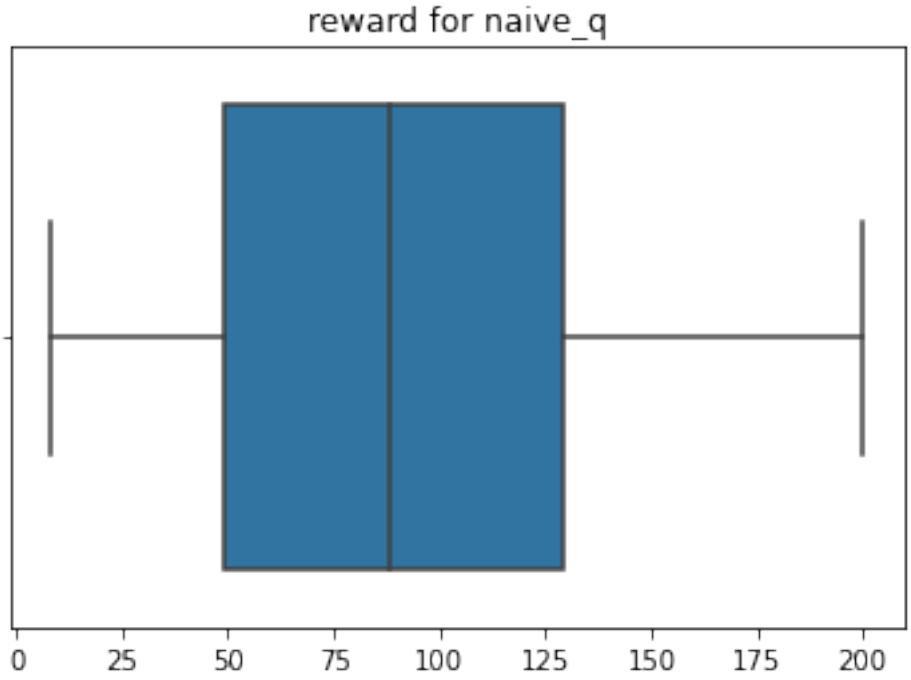
Compare the total reward using Naive Q-learning and random play.

[10]:
```
n_steps = 100000

ob = env.reset()
rewards = [] # reward for each round
reward = 0.0 # total reward in one round

for step in range(n_steps):
    #env.render()
    a = act(ob)
    ob_next, r, done, _ = env.step(a)
    update_Q(ob, r, a, ob_next, done)
    reward += r
    if done:
        rewards.append(reward)
        reward = 0.0
        ob = env.reset()
    else:
        ob = ob_next

rewards_r = [] # reward for each round
reward = 0.0 # total reward in one round
for step in range(n_steps):
    #env.render()

    ob_next, r, done, _ = env.step(env.action_space.sample())
    reward += r
    if done:
```

```
        rewards_r.append(reward)
        reward = 0.0
        ob = env.reset()
    else:
        ob = ob_next
```

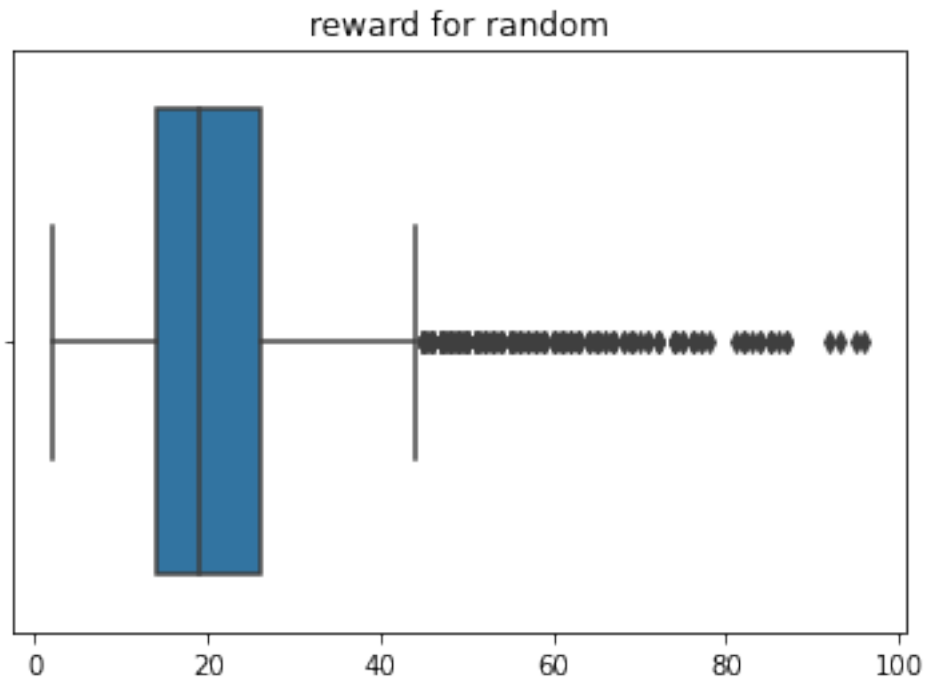
[24]:
```
import seaborn as sns
sns.boxplot(x=rewards).set_title('reward for naive_q')
```

[24]: Text(0.5, 1.0, 'reward for naive_q')



[25]:
```
sns.boxplot(x=rewards_r).set_title('reward for random')
```

[25]: Text(0.5, 1.0, 'reward for random')



As we can see from the plot, there is a significant increase on reward after applying Naive Q_learning method.