Starting up RL on OpenAI Gym

DAVIDE BACCIU – BACCIU@DI.UNIPI.IT

IN SUPREMÆ DIGNITATIS · 1343 ·
UNIVERSITÀ DI PISA

Introduction

OpenAI Gym

A toolkit for developing and comparing reinforcement learning algorithms

- ✓ Implementation of the interaction environment
- ✓ Plug-in your agent with integration of main DL frameworks

```
import gym

# create the environment
env = gym.make("FrozenLake-v0")
# reset the environment before starting
env.reset()

# loop 10 times
for i in range(10):
    # take a random action
    env.step(env.action_space.sample())
    # render the game
    env.render()

# close the environment
env.close()
```

FrozenLake-V0 – A modified gridworld

Gridworld/FrozenLake-v0

- Start (S)

- Frozen/Safe (F)

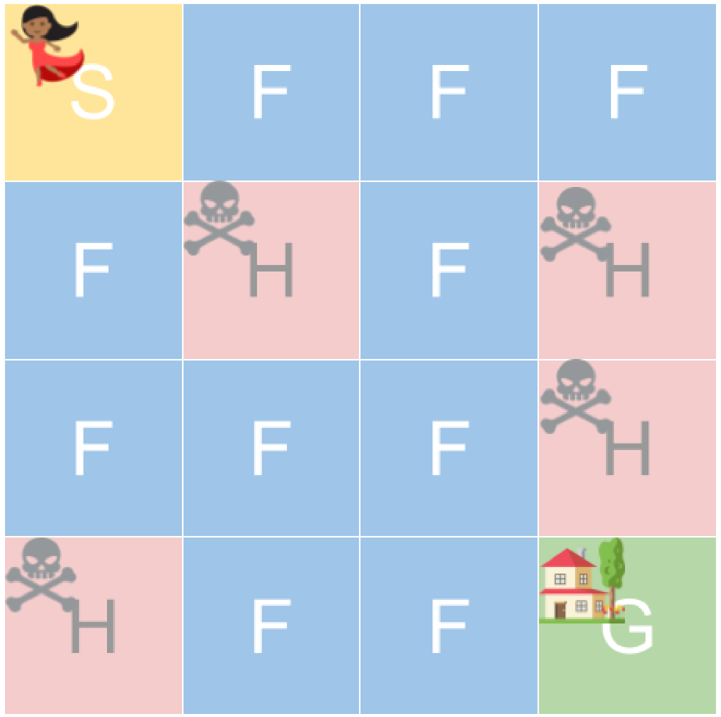
Hole/Danger (H)

- Goal (G)

The game is so; you go from Start point to Goal without falling into Hole.

You get reward of +1 for reaching the Goal; otherwise you receive nothing.

Every time you can go left, right, up and down. But you can't leave the grid.



FrozenLake-V0 – Running Episodes

```
import gym

# create the environment
env = gym.make("FrozenLake-v0")
# reset the environment before starting
env.reset()

## run in total episodes
for i_episode in range(20):
    ## restart and reset the game state.
    ## save the observation, observation == state
    state = env.reset()
     for t in range(100):
```

```
for t in range(100):
    ## render the environment
    env.render()
    print("State:", state)
    ## select a random action from available actions
    action = env.action_space.sample()
    ## apply the selected actions, and get next state
    next_state, reward, done, info = env.step(action)

    if done:
        print("Episode finished after {} timesteps".format(t+1))
        break

    state = next_state
```

```
SFFF
FHFH
FFFH
HFFG
State:  0
   (Left)
SFFF
FHFH
FFFH
HFFG
State:  0
   (Up)
…
```

Dynamic Programming

Step 1 – Prepare a main learning loop

```
# spaces dimension
nA = env.action_space.n
nS = env.observation_space.n

# initializing value function and policy
V = np.zeros(nS)
policy = np.zeros(nS)
# some useful variable
policy_stable = False
it = 0
  while not policy_stable:
      policy_evaluation(V, policy)
      policy_stable = policy_improvement(V, policy)
      it += 1
#Learning converged
run_episodes(env, policy)
```

Value function evaluation

Policy improvemnt on value function

Full code here

Value Function Evaluation

```
def policy_evaluation(V, policy, eps=0.0001):
    '''
    Policy evaluation. Update the value function until it
reach a steady state
    '''
    while True:
        delta = 0
        # loop over all states
        for s in range(nS):
            old_v = V[s]
            # update V[s] using the Bellman equation
            V[s] = eval_state_action(V, s, policy[s])
            delta = max(delta, np.abs(old_v - V[s]))

        if delta < eps:
            break
```

```
def eval_state_action(V, s, a, gamma=0.99):
    return np.sum([p * (rew + gamma*V[next_s]) for p,
                                          next_s, rew, _ in env.P[s][a]])
```

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_k(s') \right)$$

Policy Update

```
def policy_improvement(V, policy):
    '''
    Policy improvement. Update the policy based on the value function
    '''
    policy_stable = True
    for s in range(nS):
        old_a = policy[s]

        # update the policy with the action that bring to the highest state value
        policy[s] = np.argmax([eval_state_action(V, s, a)

         for a in range(nA)])
            if old_a != policy[s]:
                policy_stable = False

    return policy_stable
```

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} q_\pi(s, a)$$

Value Iteration ([Full Code Here])

$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_k(s') \right)$$

```
def value_iteration(eps=0.0001):
    ## Value iteration algorithm
    V = np.zeros(nS)
    it = 0

    while True:
        delta = 0
        # update the value of each state using as "policy" the max operator

        for s in range(nS):
            old_v = V[s]
            V[s] = np.max([eval_state_action(V, s, a) for a in range(nA)])
            delta = max(delta, np.abs(old_v - V[s]))

        if delta < eps:
            break
        else:
            print('Iter:', it, ' delta:', np.round(delta, 5))
        it += 1

    return V
```

Model Free

Q Learning (I) (Full Code Here)

```
def Q_learning(env, lr=0.01, num_episodes=10000, eps=0.3, gamma=0.95,
                                                                  eps_decay=0.00005):
    nA = env.action_space.n
    nS = env.observation_space.n

    # Initialize the Q matrix
    # Q: matrix nS*nA where each row represent a state and each colums represent a
different action
    Q = np.zeros((nS, nA))
    games_reward = []
    test_rewards = []

    for ep in range(num_episodes):
        state = env.reset()
        done = False
        tot_rew = 0

        # decay the epsilon value until it reaches the threshold of 0.01
        if eps > 0.01:
            eps -= eps_decay

[…]
```

Q Learning (II) (Full Code Here)

```
  for ep in range(num_episodes):
      [...]

    # loop the main body until the environment stops
    while not done:
      # select an action following the eps-greedy policy
      action = eps_greedy(Q, state, eps)

      next_state, rew, done, _ = env.step(action) # Take one step in the environment

      # Q-learning update the state-action value (get the max Q value for the next state)
      Q[state][action] = Q[state][action] + lr*(rew + gamma*np.max(Q[next_state]) - 
Q[state][action])

      state = next_state
      tot_rew += rew
      if done:
        games_reward.append(tot_rew)

    # Test the policy every 300 episodes and print the results
    if (ep % 300) == 0:
      test_rew = run_episodes(env, Q, 1000)
      print("Episode:{:5d}  Eps:{:2.4f}  Rew:{:2.4f}".format(ep, eps, test_rew))
      test_rewards.append(test_rew)

  return Q
```

Intermezzo - ϵ-greedy

```
def eps_greedy(Q, s, eps=0.1):
    '''
    Epsilon greedy policy
    '''
    if np.random.uniform(0,1) < eps:
        # Choose a random action
        return np.random.randint(Q.shape[1])
    else:
        # Choose the action of a greedy policy
        return np.argmax(Q[s])
```

$$\pi(a|s) = \begin{cases} {}^{\epsilon}/_{m} + (1-\epsilon) & \text{if } a^* = \arg\max_{a \in \mathcal{A}} Q(s,a) \\ {}^{\epsilon}/_{m} & \text{otherwise} \end{cases}$$

Q Learning (II) (Full Code Here)

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \max_{a'} \gamma Q(S',a') - Q(S,A)\right)$$

```
for ep in range(num_episodes):
        […]

    # loop the main body until the environment stops
    while not done:
        # select an action following the eps-greedy policy
        action = eps_greedy(Q, state, eps)

        next_state, rew, done, _ = env.step(action) # Take one step in the environment

        # Q-learning update the state-action value (get the max Q value for the next state)
        Q[state][action] = Q[state][action] + lr*(rew + gamma*np.max(Q[next_state]) - Q[state][action])

        state = next_state
        tot_rew += rew
        if done:
            games_reward.append(tot_rew)

    # Test the policy every 300 episodes and print the results
    if (ep % 300) == 0:
        test_rew = run_episodes(env, Q, 1000)
        print("Episode:{:5d}  Eps:{:2.4f}  Rew:{:2.4f}".format(ep, eps, test_rew))
        test_rewards.append(test_rew)

return Q
```

SARSA
(Full Code Here)

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$$

```
def SARSA(env, lr=0.01, num_episodes=10000, eps=0.3, gamma=0.95,
                                                          eps_decay=0.00005):
[…]

  for ep in range(num_episodes):
    […]
    action = eps_greedy(Q, state, eps)

    # loop the main body until the environment stops
    while not done:
      next_state, rew, done, _ = env.step(action) # Take one step in the environment

      # choose the next action (needed for the SARSA update)
      next_action = eps_greedy(Q, next_state, eps)
      # SARSA update
      Q[state][action] = Q[state][action] + lr*(rew + gamma*Q[next_state][next_action] -
                                                              Q[state][action])

      state = next_state
      action = next_action
      tot_rew += rew
      if done:
        games_reward.append(tot_rew)

    # Test the policy every 300 episodes and print the results
      […]

  return Q
```