FrozenLake-v0

https://gym.openai.com/envs/FrozenLake-v0/

The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.

Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend.

The surface is described using a grid like the following:

```
SFFF       (S: starting point, safe)
FHFH       (F: frozen surface, safe)
FFFH       (H: hole, fall to your doom)
HFFG       (G: goal, where the frisbee is located)
```

The episode ends when you reach the goal or fall in a hole. You receive a reward of 1 if you reach the goal, and zero otherwise.

Understanding an Environment

FrozenLake-v0

To understand the basics of importing Gym packages, loading an environment,
and other important functions associated with OpenAI Gym, here's an example
of a **Frozen Lake** environment.

Load the Frozen Lake environment in the following way:

```
import Gym
env = Gym.make('FrozenLake-v0') #make function of Gym loads the specified environment
```

Next, we come to resetting the environment. While performing a reinforcement
learning task, an agent undergoes learning through multiple episodes. As a
result, at the start of each episode, the environment needs to be reset so that it
comes to its initial situation and the agent begins from the start state. The
following code shows the process for resetting an environment:

```
import Gym
env = Gym.make('FrozenLake-v0')
s = env.reset()  # resets the environment and returns the start state as a value
print(s)
-----------
0  #initial state is 0
```

After taking each action, there might be a requirement to show the status of the
agent in the environment. Visualizing that status is done by:

```
env.render()
------------
SFFF
FHFH
FFFH
HFFG
```

The preceding output shows that this is an environment with 4×4
grids, that is,16 states arranged in the preceding manner where S, H, F, and G represents
different forms of a state where:

S: Start block
F: Frozen block

H: Block has hole
G: Goal block
In newer versions of the Gym, the environment features can't be modified
directly. This is done by unwrapping the environment parameters with:

```
env = env.unwrapped
```

Each environment is defined by the state spaces and action spaces for the agent
to perform. The type (discrete or continuous) and size of state spaces and action
spaces is very important to know in order to build a reinforcement learning
agent:

```
print(env.action_space)
print(env.action_space.n)
----------------
Discrete(4)
4
```

The `Discrete(4)` output means that the action space of the Frozen
Lake environment is a discrete set of values and has four distinct actions that can
be performed by the agent.

```
print(env.observation_space)
print(env.observation_space.n)
----------------
Discrete(16)
16
```

The `Discrete(16)` output means that the observation (state) space of the Frozen
Lake environment is a discrete set of values and has 16 different states to be
explored by the agent.

This environment consists of *4 x 4* grids representing a lake. Thus, we have 16
grid blocks, where each block can be a start block(S), frozen block(F), goal
block(G), or a hole block(H). Thus, the objective of the agent is to learn to
navigate from start to goal without falling in the hole:

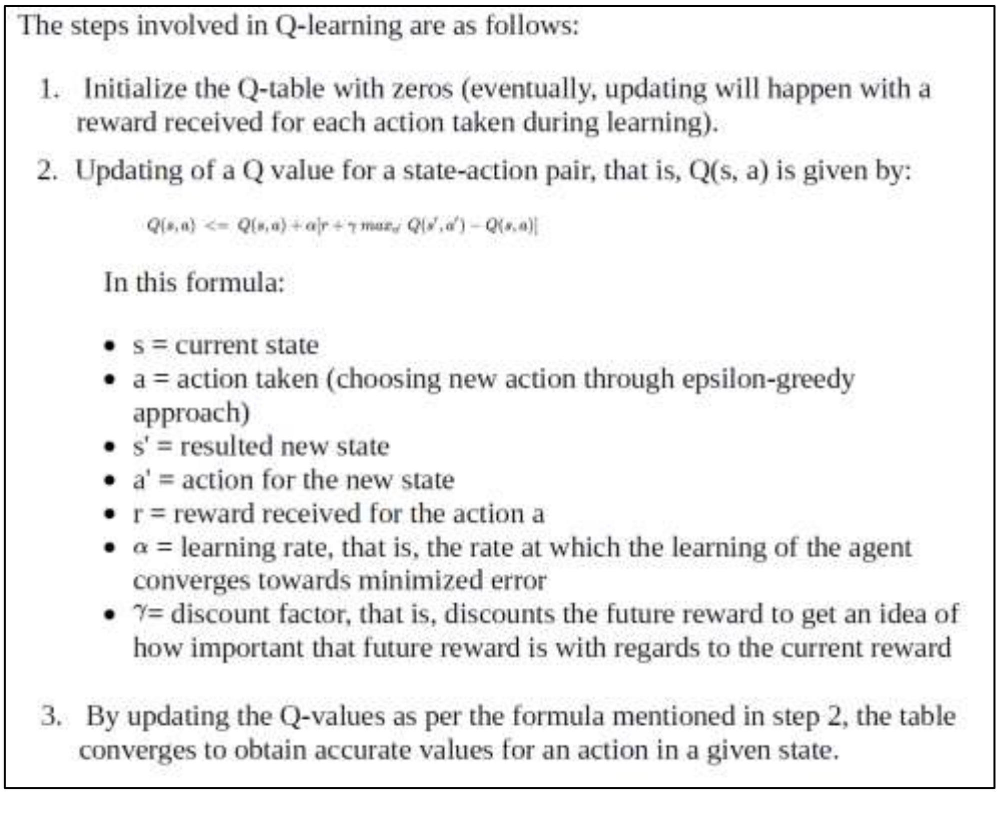
```
import Gym
env = Gym.make('FrozenLake-v0') #loads the environment FrozenLake-v0
env.render() # will output the environment and position of the agent
-------------------
SFFF
FHFH
FFFH
HFFG
```

At any given state, an agent has four actions to perform, which are up, down,
left, and right. The reward at each step is 0 except the one leading to the goal
state, then the reward would be 1. We start from the S state and our goal is to
reach the G state without landing up in the H state in the most optimized path
through the F states.

Implementing Q-Learning

Now, let's try to program a reinforcement learning agent using Q-learning. Q-learning consists of a Q-table that contains Q-values for each state-action pair.
The number of rows in the table is equal to the number of states in the environment and the number of columns equals the number of actions. Since the
number of states is 16 and the number of actions is 4, the Q-table for this environment consists of 16 rows and 4 columns. The code for it is given here:

```
print("Number of actions : ",env.action_space.n)
print("Number of states : ",env.observation_space.n)
----------------------
Number of actions : 4
Number of states : 16
```



The steps involved in Q-learning are as follows:

1. Initialize the Q-table with zeros (eventually, updating will happen with a reward received for each action taken during learning).
2. Updating of a Q value for a state-action pair, that is, Q(s, a) is given by:

   $$Q(s,a) <= Q(s,a) + \alpha[r + \gamma\, max_{a'}\, Q(s',a') - Q(s,a)]$$

   In this formula:

   - s = current state
   - a = action taken (choosing new action through epsilon-greedy approach)
   - s' = resulted new state
   - a' = action for the new state
   - r = reward received for the action a
   - α = learning rate, that is, the rate at which the learning of the agent converges towards minimized error
   - γ= discount factor, that is, discounts the future reward to get an idea of how important that future reward is with regards to the current reward

3. By updating the Q-values as per the formula mentioned in step 2, the table converges to obtain accurate values for an action in a given state.

The Epsilon-Greedy is a widely used solution to the explore-exploit dilemma.
Exploration is all about searching and exploring new options through experimentation and research to generate new values, while exploitation is all
about refining existing options by repeating those options and improving their
values.
The Epsilon-Greedy approach is very simple to understand and easy to implement:

```
epsilon( ) = 0.05 or 0.1 #any small value between 0 to 1
#epsilon( ) is the probability of exploration
p = random number between 0 and 1
if p epsilon( ) :
pull a random action
else:
pull current best action
```

Eventually, after several iterations, we discover the best actions among all at
each state because it gets the option to explore new random actions as well as
exploit the existing actions and refine them.

Let's try to implement a basic Q-learning algorithm to make an agent learn how
to navigate across this frozen lake of 16 grids, from the start to the goal without
falling into the hole:

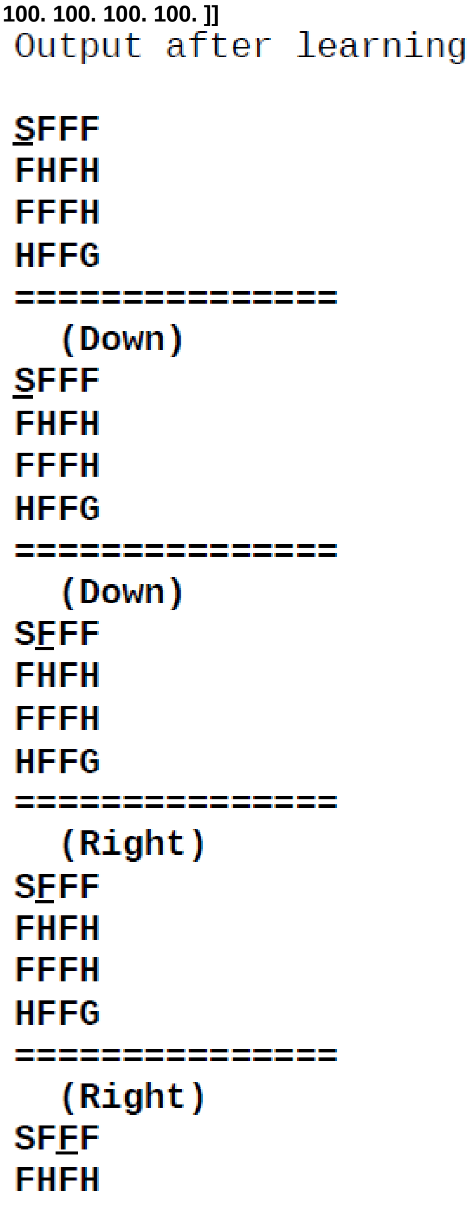
```
# importing dependency libraries
from __future__ import print_function
import Gym
import numpy as np
import time
#Load the environment
env = Gym.make('FrozenLake-v0')
s = env.reset()
print("initial state : ",s)
print()
env.render()
print()
print(env.action_space) #number of actions
print(env.observation_space) #number of states
print()
print("Number of actions : ",env.action_space.n)
print("Number of states : ",env.observation_space.n)
print()
#Epsilon-Greedy approach for Exploration and Exploitation of the state-action spaces
def epsilon_greedy(Q,s,na):
epsilon = 0.3
p = np.random.uniform(low=0,high=1)
#print(p)
if p > epsilon:
return np.argmax(Q[s,:])#say here,initial policy = for each state consider the action having highest else:
return env.action_space.sample()
# Q-Learning Implementation
```

```
#Initializing Q-table with zeros
Q = np.zeros([env.observation_space.n,env.action_space.n])
#set hyperparameters
lr = 0.5 #learning rate
y = 0.9 #discount factor lambda
eps = 100000 #total episodes being 100000
for i in range(eps):
s = env.reset()
t = False
while(True):
a = epsilon_greedy(Q,s,env.action_space.n)
s_,r,t,_ = env.step(a)
if (r==0):
if t==True:
r = -5 #to give negative rewards when holes turn up
Q[s_] = np.ones(env.action_space.n)*r #in terminal state Q value equals the reward
else:
r = -1 #to give negative rewards to avoid long routes
if (r==1):
r = 100
Q[s_] = np.ones(env.action_space.n)*r #in terminal state Q value equals the reward
Q[s,a] = Q[s,a] + lr * (r + y*np.max(Q[s_,a]) - Q[s,a])
s = s_
if (t == True) :
break
print("Q-table")
print(Q)
print()
print("Output after learning")
print()
#learning ends with the end of the above loop of several episodes above
#let's check how much our agent has learned
s = env.reset()
env.render()
while(True):
a = np.argmax(Q[s])
s_,r,t,_ = env.step(a)
print("===============")
env.render()
s = s_
if(t==True) :
break
```

```python
# importing dependency libraries
from __future__ import print_function
import Gym
import numpy as np
import time

#Load the environment

env = Gym.make('FrozenLake-v0')

s = env.reset()
print("initial state : ",s)
print()

env.render()
print()

print(env.action_space) #number of actions
print(env.observation_space) #number of states
print()

print("Number of actions : ",env.action_space.n)
print("Number of states : ",env.observation_space.n)
print()

#Epsilon-Greedy approach for Exploration and Exploitation of the state-action spaces
def epsilon_greedy(Q,s,na):
    epsilon = 0.3
    p = np.random.uniform(low=0,high=1)
    #print(p)
    if p > epsilon:
        return np.argmax(Q[s,:])#say here,initial policy = for each state consider the a
    else:
        return env.action_space.sample()

# Q-Learning Implementation

#Initializing Q-table with zeros
Q = np.zeros([env.observation_space.n,env.action_space.n])

#set hyperparameters
lr = 0.5 #learning rate
y = 0.9 #discount factor lambda
eps = 100000 #total episodes being 100000

for i in range(eps):
    s = env.reset()
    t = False
    while(True):
        a = epsilon_greedy(Q,s,env.action_space.n)
        s_,r,t,_ = env.step(a)
        if (r==0):
            if t==True:
                r = -5 #to give negative rewards when holes turn up
                Q[s_] = np.ones(env.action_space.n)*r #in terminal state Q value equals
            else:
                r = -1 #to give negative rewards to avoid long routes
        if (r==1):
                r = 100
                Q[s_] = np.ones(env.action_space.n)*r #in terminal state Q value equals
        Q[s,a] = Q[s,a] + lr * (r + y*np.max(Q[s_,a]) - Q[s,a])
        s = s_
        if (t == True) :
            break

print("Q-table")
print(Q)
print()
print("Output after learning")
print()
#learning ends with the end of the above loop of several episodes above
#let's check how much our agent has learned
s = env.reset()
env.render()
while(True):
    a = np.argmax(Q[s])
    s_,r,t,_ = env.step(a)
    print("===============")
    env.render()
    s = s_
    if(t==True) :
        break
```

<<OUTPUT>>

```
initial state : 0
SFFF
FHFH
FFFH
HFFG
Discrete(4)
Discrete(16)
Number of actions : 4
Number of states : 16
Q-table
[[ -9.85448046 -7.4657981 -9.59584501 -10. ]
[ -9.53200011 -9.54250775 -9.10115662 -10. ]
[ -9.65308982 -9.51359977 -7.52052469 -10. ]
[ -9.69762313 -9.5540111 -9.56571455 -10. ]
[ -9.82319854 -4.83823005 -9.56441915 -9.74234959]
[ -5. -5. -5. -5. ]
[ -9.6554905 -9.44717167 -7.35077759 -9.77885057]
[ -5. -5. -5. -5. ]
[ -9.66012445 -4.28223592 -9.48312882 -9.76812285]
[ -9.59664264 9.60799515 -4.48137699 -9.61956668]
[ -9.71057124 -5.6863911 -2.04563412 -9.75341962]
[ -5. -5. -5. -5. ]
[ -5. -5. -5. -5. ]
[ -9.54737964 22.84803205 18.17841481 -9.45516929]
[ -9.69494035 34.16859049 72.04055782 40.62254838]
[ 100. 100. 100. 100. ]]
```



```
Output after learning

SFFF
FHFH
FFFH
HFFG
===============
  (Down)
SFFF
FHFH
FFFH
HFFG
===============
  (Down)
SFFF
FHFH
FFFH
HFFG
===============
  (Right)
SFFF
FHFH
FFFH
HFFG
===============
  (Right)
SFFF
FHFH
```

```
FFFH
HFFG
===============
  (Right)
SFFF
FHFH
FFFH
HFFG
===============
  (Right)
SFFF
FHFH
FFFH
HFFG
===============
  (Right)
SFFF
FHFH
FFFH
HFFG
===============
  (Right)
SFFF
FHFH
FFFH
HFFG
===============
  (Right)
SFFF
FHFH
FFFH
HFFG
===============
  (Right)
SFFF
FHFH
FFFH
HFFG
===============
  (Right)
SFFF
FHFH
FFFH
HFFG
===============
  (Right)
SFFF
FHFH
FFFH
HFFG
```