Exercise 9

Machine Learning for Robotics SS 2017

Instructor: Georg Martius georg.martius@tuebingen.mpg.de (mailto:georg.martius@tuebingen.mpg.de)

Due-date: 06.07.2017 (hand in at beginning of recitation session)

In [ ]:

```
##############################################################
```

This time we will start getting familiar with Open AI gym, see https://gym.openai.com/ (https://gym.openai.com/) For the code and installation instructions see https://github.com/openai/gym (https://github.com/openai/gym)

This time we don't need additional environments, but you can also install all of them.

The main task for this exercise is to implement Q-learning with linear function approximation using radial basis functions. More instructions follow below

Initialization

In [ ]:

```
import gym
from gym import spaces
import numpy as np
import itertools
import time
```

In [ ]:

```
env = gym.make('CartPole-v0')
ac_space = env.action_space
o_space = env.observation_space
print(ac_space)
print(o_space)
print np.round(zip(env.observation_space.low, env.observation_space.high),2)
```

Here are our limits that we use. (in particular for the velocities we need to give proper limits, because they are infinity)

In [ ]:

```
my_limits = [[  -2 , 2], [-2 , 2 ], [-0.4, 0.4], [ -3, 3]]
```

Feature Mapping

The feature map is taking and observation and tranforming it into activations of features. Here we use radial basis functions. You have to distribute them in the space in a regular grid.

In [ ]:

```
# radial basis function with mean "mu" and width "sigma"
def rbf(x,mu,sigma):
    return np.exp(-np.inner(x-mu,x-mu)/(2*sigma*sigma))

# this function gets the observations space and returns a function that will calcul
#  feature activations for a given observation
def rbf_featuremap(o_spaceob, features_per_dim=10, limits=None):
    dim = env.observation_space.shape[0]
    if limits is None:
        limits = zip(env.observation_space.low, env.observation_space.high)
    ## the limits give for each dimension of the input the range of values
    raise ValueError("Implement the rest")
    # hints: use np.linspace and meshgrid to get a grid of points in the dim-dimens
    # (you need some transpositions and reshapes)
    # you probably want to generate grid points in the unit cube and rescale the in
    # you have to return a function: observation -> featurevector
```

example of a 3x3x3x3 grid

In [ ]:

```
f=rbf_featuremap(o_space, features_per_dim=3, limits=my_limits)
# and get the feature vector for the starting point
ob = env.reset()
ob, f(ob)
```

Test your features

You want to sweep through the first of the coordinates and see how your first 3 features react (depending how you where creating your grid points)

In [ ]:

```
ValueError("Implement")
```

Q-learning with linear function approximation

Complete the following code.

In [ ]:

```
class LinearQAgent(object):
    """
    Agent implementing Q-learning with linear function approximation.
    """
    def __init__(self, observation_space, action_space, featuremap, **userconfig):
        if not isinstance(observation_space, spaces.box.Box):
            raise UnsupportedSpace('Observation space {} incompatible with {}. (Req
        if not isinstance(action_space, spaces.discrete.Discrete):
            raise UnsupportedSpace('Action space {} incompatible with {}. (Reqire D
        self.observation_space = observation_space
        self.action_space = action_space
        self.action_n = action_space.n
        self.featuremap = featuremap
        self.config = {
            "init_mean" : 0.0,      # Initialize weights with this mean
            "init_std" : 0.0,       # Initialize weights with this standard deviati
            "learning_rate" : 0.05,
            "eps": 0.05,            # Epsilon in epsilon greedy policies
            "discount": 0.95}
        self.config.update(userconfig)
        ob = observation_space.sample()
        features = featuremap(ob)

        raise ValueError("Implement")

    def getQ(self,observation,action):
        raise ValueError("Implement")

    def act(self, observation, eps=None):
        if eps is None:
            eps = self.config["eps"]
        # epsilon greedy.
        if np.random.random() > eps:
            action = np.argmax([self.getQ(observation, a) for a in range(self.actio
        else:
            action = self.action_space.sample()
        return action

    def learn(self, ob, action, reward, ob_next):
        raise ValueError("Implement")
```

In [ ]:

```
q_agent = LinearQAgent(o_space, ac_space, featuremap=rbf_featuremap(o_space, 20, my
                      learning_rate=0.05, init_mean= 0.0, init_std= 0, discount=0.9
```

check the q values for the initial state

In [ ]:

```
q_agent.getQ(ob, ac_space.sample())
```

Training

Initially it is set to 2 episodes with random a policy and rendering switched on, to get an idea.

For training use "Q" mode and a few 100 episodes to play with the parameters and then use 1000, without rendering of course

In [ ]:

```
max_episodes=2
fps=100
#mode="random"
show=True # Render the simulation or not
#mode="Q"
mode="random"
max_steps = env.spec.tags['wrapper_config.TimeLimit.max_episode_steps']

for i in range(max_episodes):
    total_reward = 0
    ob = env.reset()
    if show: env.render(mode='human')
    for t in range(max_steps):
        done = False
        if mode == "random":
            a = ac_space.sample()
        elif mode == "Q":
            a = q_agent.act(ob)
        else:
            raise ValueError("no implemented")
        (ob_new, reward, done, _info) = env.step(a)
        total_reward+= reward
        if mode == "Q":
            q_agent.learn(ob, a, reward, ob_new)
        ob=ob_new
        if show:
            time.sleep(1.0/fps)
            env.render()
        if done: break
    # print("Done after {} steps. Reward: {}".format(t+1, total_reward))
```

Plot learning curve

In [ ]:

```
ValueError("Implement")
```

Investigate performance for different parameters and number of features

In [ ]:

```
ValueError("Implement")
```

Test run

with greedy policy

In [ ]:

```
total_reward = 0
fps = env.metadata.get('video.frames_per_second') or 100
ob = env.reset()
env.render(mode='human')
observations = []
for t in range(max_steps):
    done = False
    a = q_agent.act(ob, 0.0) # greedy
    (ob_new, reward, done, _info) = env.step(a)
    observations.append(ob_new)
    total_reward+= reward
    ob=ob_new
    time.sleep(1.0/fps)
    env.render()
    if done: break
print("Done after {} steps. Reward: {}".format(t+1, total_reward))
```

look at the observations

In [ ]:

```
```

In [ ]:

```
ValueError("Implement")
```

In [ ]:

```
```

Bonus

Try another environment, such as Acrobot-v1 or so