Reinforcement learning

So far we have considered networks that learn in a supervised manner. In practice agents often learn from rewards that are provided by the environment. This is the subject matter of reinforcement learning (RL). This is an active area of research. In this assignment you will learn to implement an agent which can learn from its mistakes using RL.

We provide an environment (EvidenceEnv in my_env.py) that generates observations (evidence) about an unknown state (0 or 1). The state determines if an observation is equal to the state (with probability p) or different from the state (with probability 1-p). Consult the code. The agent receives a reward of +1 if it chooses the correct state and a reward of -1 if it chooses the wrong state. This state can only be indirectly observed through the observations.

Below we define a RandomAgent which takes random actions based on its observations:

```
class RandomAgent(object):

    def __init__(self, env):
        """

        Args:
            env: an environment
        """

        self.env = env

    def act(self, observation):
        """
        Act based on observation and train agent on cumulated reward (return)

        :param observation: new observation
        :param reward: reward gained from previous action; None indicates no reward because of initial state
        :return: action (Variable)
        """

        return np.random.choice(self.env.n_action)

    def train(self, a, old_obs, r, new_obs):
        """

        :param a: action
        :param old_obs: old observation
        :param r: reward
        :param new_obs: new observation
        :return:
        """

        pass
```

We provide you with the code to run an agent on the environment using the RandomAgent:

```
# Number of iterations
n_iter = 1000

# environment specs
env = EvidenceEnv(n=2, p=0.95)

# define agent
agent = RandomAgent(env)

# reset environment and agent
obs = env.reset()
```

```
reward = None
done = False

R = []
for step in range(n_iter):

    env.render()

    action = agent.act(obs)

    _obs, reward, done, _ = env.step(action)

    # no training involved for random agent
    agent.train(action, obs, reward, _obs)

    obs = _obs

    R.append(reward)
```

1. Run the code and plot the cumulative reward over time.

We want to improve on the random agent. There exist **many** different RL algorithms. For detailed reading consult the following slides:

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

We will implement an RL algorithm called Q-learning. The goal of Q-learning is to estimate the value of taking an action a in a state s. The agent selects that action which maximizes the value given an input state (observations). Read the following blog up to and including the section Q-learning:

http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/

2. Implement the Tabular Q-learning algorithm and show that your TabularQAgent learns to accumulate reward over time. Plot the cumulative rewards for this agent. Also plot the values for Q(s,a) before and after learning. Note that the state can be represented by converting the binary observations into an integer number. You may use the function EvidenceEnv.asint(obs) for this. Your task is to implement the act and train functions.

Ideally we want to get rid of the tabular representation (why?). Read the full blog previously mentioned. We won't implement deep Q-learning. Instead we will implement an agent which uses an MLP that takes observations and learns to compute the Q value for all possible actions. This is done by backpropagation on the loss defined by the sum squared difference between the predicted and desired Q values. For simplicity we backpropagate per example.

3. Implement a NeuralQAgent in chainer which uses an MLP that takes observations and learns to compute the Q value for all possible actions. Use backpropagation to train your network. Plot the cumulative rewards for this agent. Also plot the values for Q(s,a) before and after learning. You may use the function EvidenceEnv.asbinary (I, b_len) for this.

Note that this is a first step in RL. Our environment adheres to openAI gym's definitions: https://openai.com. If you wish to dive into RL then check out their website. Note further that we did not implement various tricks to improve convergence. Also, we are dealing with an environment in which the reward does not depend on the past. More general setups require that we deal with past influences as well. How would you tackle this? Also, how to model continuous rather than discrete actions? These are all important questions.