Tsinghua-Berkeley Shenzhen Institute
Learning from Data
Fall 2019

Programming Homework 4

TIAN Chenyu December 5, 2019

- **Acknowledgments:** This template takes some materials from course CSE 547/Stat 548 of Washington University: `https://courses.cs.washington.edu/courses/cse547/17sp/index.html`.
- **Collaborators:** I finish this homework by myself.

4.1. The code is in the *Q_ learning.py*.

4.2. The matrix of learned reward is shown in 1. The MSE is 0.08.

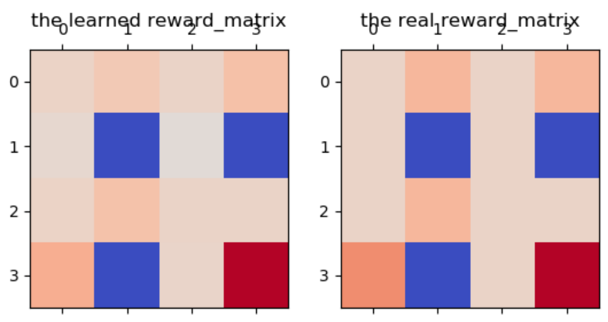
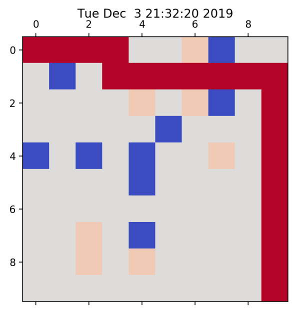
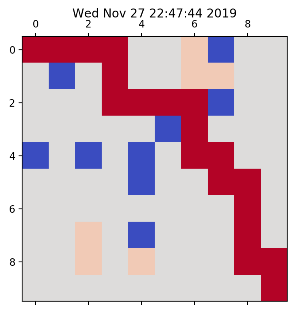


Figure 1: Matrix Distribution

The most different part is (3, 0), where the reward is 2 but the learned reward is 1.23.
Explanation: It is obvious that (3, 0) is near the trap. In the beginning, because of random action, the mouse is easy to get a negative reward if starts from here. As the ϵ getting smaller afterwards, then (3,0) is rarely visited for a low Q_value. Therefore, small samples of Q_value in (3, 0) leads to some estimation bias.



(a) The Learned route by DQN



(b) The Learned route by Q-table

4.3. I used Keras 2.3.0 to build the neural network by adding methods in the origin *Agent* class. At each iteration of DQN, a mini-batch of states, actions, rewards, and next states are sampled from the replay memory as observations to train the Q-network, which approximates the Q function.

```
    def _build_model(self):
        # Neural Net for Deep-Q learning Model
        model = Sequential()
        model.add(Dense(128, input_dim=2, activation='relu'))
        model.add(Dense(128, activation='relu'))
        model.add(Dense(128, activation='relu'))
        model.add(Dense(self.action_size, activation='linear')
            )
        model.compile(loss='mse',
                      optimizer=Adam(lr=self.alpha))
        return model

    def update_target_network(self):
        weights = self.model.get_weights()
        self.target_model.set_weights(weights)
```

DQN works and finds some different routes from the Q_table method as shown in the 2a.
There are some comparisons between 2 methods:

(a) In this 10*10 senario, DQN takes much more time to train the agent than q-table.

(b) It is more difficult to tune the DQN hyperparameters such as netword architecture and batch size. But Q_table is only have to tune learning rate.

(c) The solution of Q_table is stable. But the learned solution of DQN can be different everytime.

(d) The DQN learned actions sometimes can fall into dead loop. For example, its action can be go left then go right, repeat this pair of actions forever. So the maximum steps need to be set to jump out of the loop.

(e) In this discrete senario, I think Q learning is more efficient. But DQN is more flexible to complex tasks with continous states and actions.

The source code could be found in *dqn_ 10_ 10.py*. Here are the core codes:

```
    def replay(self):
        batch_size = min(self.batch_size, len(self.memory))
        minibatch = random.sample(self.memory, self.batch_size
            )
        X = minibatch[0][0]
        Y = self.model.predict(X)
        for state, action, reward, next_state in minibatch:
            X = np.append(X, state, axis=0)
            target = (reward + self.gamma *
                      np.amax(self.target_model.predict(
                          next_state)))
            target_f = self.model.predict(state)
            target_f[0][action] = target # action is a
                action_list index
            if state[0][0] == 0:
                target_f[0][2] = 0
            if state[0][0] == self.max_row:
                target_f[0][3] = 0
            if state[0][1] == 0:
                target_f[0][0] = 0
            if state[0][1] == self.max_col:
                target_f[0][1] = 0
            Y = np.append(Y, target_f, axis=0)
        self.model.fit(X[1:,],Y[1:,], batch_size=self.
            batch_size, epochs=20, verbose=False)
        print(the error
            value, self.model.evaluate(X[1:,],Y[1:,]))
```