AU 332 Artificial Intelligence: Principles and Techniques

By: Xiaocheng Wang(517030910326)

HW#: 3

November 1, 2019

I. INTRODUCTION

In this period we have learnt about reinforcement learning, especially Q-learning. It's quite a powerful tool to train an agent to learn the optimal policy from reward in some particular environments. In this homework, we will implement RL in maze environments and on Atari Game.

II. IMPLEMENTATION

A. Reinforcement Learning in Maze Environment

In this assignment, I will implement a Dyna-Q learning agent to search for the treasure and exit in a grid-shaped maze. The agent will learn by trail and error from interactions with the environment and finally acquire a policy to get as high as possible scores in the game.

1. **Game sketch**

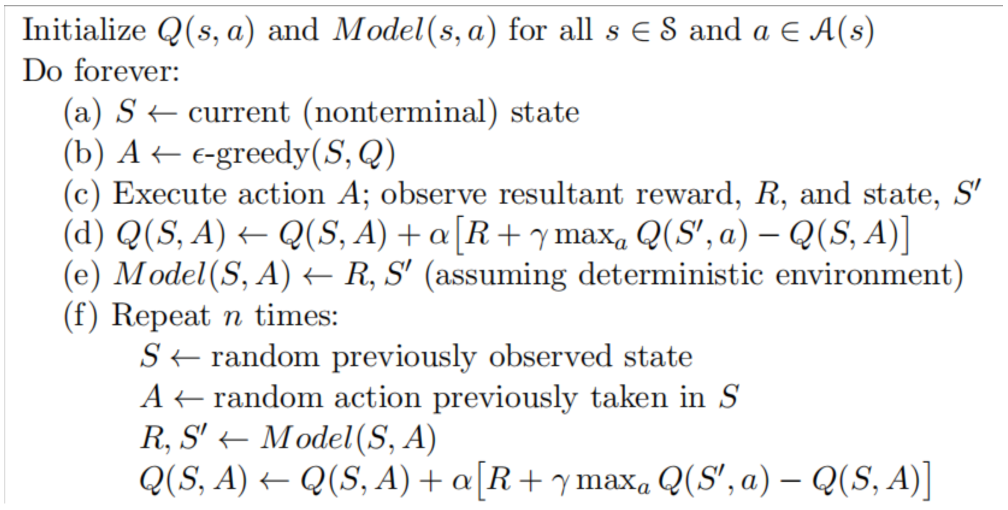
   The information used by the agent to interact with the environment is state, action and reward.

   (a) **state:** a state is a 5D vector, in which the first four elements are the pixel-coordinates of middle-points of the rectangle's edges, and the last is a boolean variable to indicate whether the treasure has been eaten.

   (b) **action:** an action is an integer among {0,1,2,3}, representing left, up, down and right respectively.

2. **Dyna-Q Learning**

   Dyna-Q learning differs from normal Q learning in each exploring step that it uses an experience model to generate sub-optimal policies, which accelerates the learning process. It's a combination of planning and learning. And here's Dyna-Q algorithm shown with pseudocodes.



```
Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
    (a) $S \leftarrow$ current (nonterminal) state
    (b) $A \leftarrow \epsilon$-greedy$(S,Q)$
    (c) Execute action $A$; observe resultant reward, $R$, and state, $S'$
    (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
    (f) Repeat $n$ times:
            $S \leftarrow$ random previously observed state
            $A \leftarrow$ random action previously taken in $S$
            $R, S' \leftarrow Model(S,A)$
            $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
```

(a)Dyna-Q Algorithm. $Model(s,a)$ denotes the contents of the model (predicted next state and reward) for state–action pair s, a. Direct reinforcement learning, model-learning, and planning are implemented by steps (d), (e), and (f), respectively. If (e) and (f) were omitted, the remaining algorithm would be one-step tabular Q-learning.

   Implementation just follows the preudocode. First I implement a q-learning algorithm with ϵ-greedy exploring strategy, and then use a model to backup (s,a,s,r) pairs and simulate n times after each real step.

3. **Exploration-Exploitation Strategy**

   Initially we use ϵ-greedy strategy, which is simple and effective but not efficient. There are some other random exploring strategies, and they are all called undirected exploration strategies. Undirected explorations take no learning experience into consideration, i.e. just make everything happen by accident. Though probability theory guarantees that possible things will happen, it's not clever.

   Therefore, using directed exploration is a must, It has been shown (Thrun, 1992b) that directed exploration strategies are inherently superior to undirected strategies.

   One basic idea to decide the direction is a counter to record how many times an action has been taken(or a state has been visited). In later episodes, we don't want to do the repeated things as in earlier episodes, and the unvisited states is more attractive.

   Besides, if two states have been visited equally often, but one of the two was late visited and the other very early in the exploration process, it makes intuitive sense to prefer to revisit that earlier state to see of what has since been learned improves the agent's performance from that state.

   There are other extended counter-based strategies like *Counter/Error-Based Exploration*, and other information based strategies as well. In my inplementation, I use *Counter-Based Exploration with Decay* idea and changed the original formula a little. My update-formula of Q values updating is

   $$Q(s,a) = Q(s,a) + \alpha \cdot (reward + \gamma \cdot \max_{a}^{\prime} \left(Q(s',a') - \frac{N(s',a')}{k} \right) - Q(s,a))$$

   Here we make the counter a direct penalty to a Q values instead of a reduction on a positive gain. $k > 0$ is a constant to control the influence of the number of visits. But this updating rule can lead to a deadly bug that when episode number is large, the increasing counting number may cause huge impact on the Q values and weaken the strength of reward, then the agent will want to terminate the process as soon as possible instead of searching for rewards.

   There are several ways to deal with it. A direct way is to set k as large as possible, which actually doesn't make sense, since in this case, the influence of counter still increases as the counting number grows. A more clever trick is to update $N(s,a)$ each time we update the Q values, i.e. a step is taken. The rule is shown below

   $$N(s,a) \leftarrow \lambda \cdot N(s,a)$$

   By setting a proper value of λ, we can control the influence of the counter in different learning period. For instance, assume $\lambda = 0.9$, then when $N < 10$, we have $N < \lambda(N+1)$, which means its influence on Q values is increasing; when $N > 10$, $N > \lambda(N+1)$, then the influence decrease. This idea comes up to the rule that at the beginning of the learning process, the agent should try more new possibilities, and at the end the result should converge.

   Notice that $N(s,a) \leftarrow \lambda \cdot N(s,a)$ also function as the decay strategy, how amazing!

4. **Results/Param Adjustment:**

   The params are not independent to each other. If learning rate α is too small, k has to be large enough to smooth the peek of $N(s,a)$ and make sure $N(s,a)$ works for a long period of time. Also λ should be more closer to 1, so that the turning point of the influence of $N(s,a)$ falls in proper time. Besides, the repeat time for dynamic q algorithm should also increase a little.

   However, α should not be very small in these maze problems for the problem size, but also not to big to miss some possible closer path. Every parameter make a different on the training result. By choosing proper parameters, my agent can find a optimal path within 100 episodes in most cases, sometimes a suboptimal result, and takes about 200 episodes to find a solution occationally.

B. Reinforcement Learning on Atari Game

1. Description

   In this part, we will implement a DQN agent to play atari game breakout. The given code is almost complete. We are asked to read the code and complete the remaining part. The performance of the agent may not be satisfactory and we have to tune it to get higher scores.

   Since the state and action space of such game is extremely large, we can not represent $Q(s, a)$s expicitly, instead we use neural networks to get q functions. After observing the given codes, we first finished the frame of a q learning. First initialize the game; for the first some amount of steps, we just randomly act to collect starting training data for the DQN and do not train the DQN; when after some number of actions, we begin to train the network every particular steps, and also collect more $(s, a, r, s'.done)$s for further training.

2. Tuning

   The original network params and structure performs badly after 5000 episodes, so we tune the agent.

   (a) We first try to set a larger ϵ so that the agent may find more strategies. Actually it doesn't work.

   (b) We notice that the input size is four times the size of state, since it's supposed to use more than one frame of the game process as a single training sample, say 4 frames. It does make sense, for a frame may not contain enough information about something like the moving direction. But a (512,) input seems to be to large for our humble network, and 5000 episodes may not be enough to converge. Therefore we try to use 2 frames per time. However it doesn't work neither.

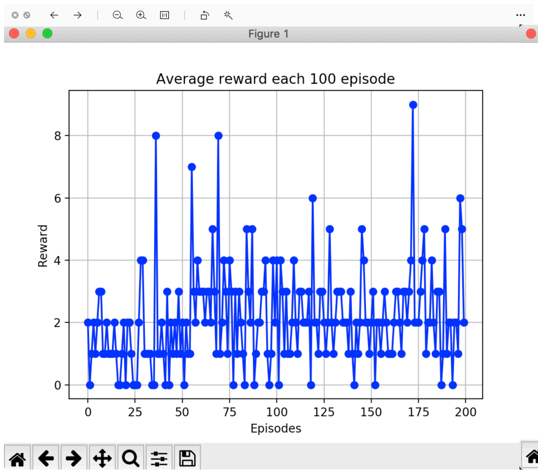
   (c) We also notice that the rewards for most states are very small, causing inbalance between input data. And for a neural network, this leads to a difficult training. So we try to use the some highest rewarded $(s, a, r, s'.done)$s in the memory to train the network rather than randomly choose. This still doesn't work.

   (d) Since training is really time consuming, we cannot adjust the params as much as we want.

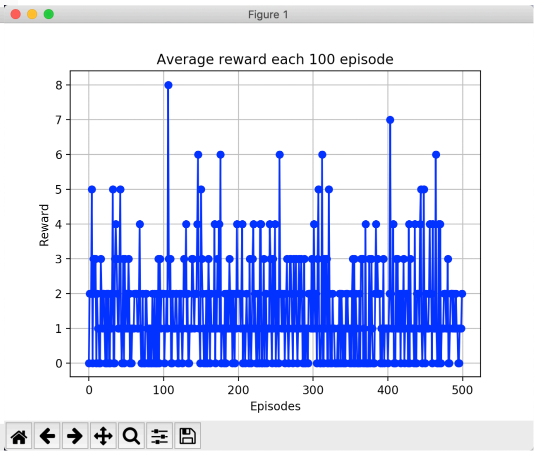
   (e) CODE The core code for DQN is displayed as follows:

```
dqn_agent.target_train()
for e in range(episodes):
    sum_rewards=0
    curr_state = dqn_agent.env.reset()
    curr_state = np.hstack((curr_state, curr_state, curr_state, curr_state)).
        reshape(1, dqn_agent.state_size)
    update_check=0
    for step in range(trial_len):
        action=dqn_agent.choose_action(curr_state,step)
        state,reward,done = dqn_agent.intercat_env(action)
        dqn_agent.remember(curr_state, action, reward, state, done)
        if done:
            break
        dqn_agent.replay()
        sum_rewards+=reward
        curr_state=state
        if update_check>=update:
            dqn_agent.target_train()
            update_check=0
        update_check+=1
```

   (f) RESULT Due to the time limitation, we have not stored the result for 5000 episodes, which have the average score over 7. Here we display the 200 episodes for result display. Note that the performance is getting polished in a rather fast pace.



(b)path



(c)policy

(g) DISCUSSION The network use above consists of full-connected layers. However, the paper written by deepmind tech actually indicates a network with 3 convolution layers and full-connected layers. The trial of such structure somehow comes with a rather unsatisfying result.

III. DISCUSSION & CONCLUSION

Reinforcement learning is interesting, and training an agent is full of fun!