Q-Learning and DDQN for Flappy Bird

HE Tianlang*, XU Haoran†, Pei Yulin‡and Dong Ziwei§
*Email: siriushe2019@gmail.com
†Email: hxubh@connect.ust.hk
‡Email: ypeiae@connect.ust.hk
§Email: zdongaj@connect.ust.hk

***Abstract*—Reinforcement learning is trying to learn from the interaction with the environment, which is very similar to the process of human learning knowledge. Deep reinforcement learning is a combination of reinforcement learning and deep learning. In this project, we try to use reinforcement learning method and deep reinforcement learning method to train an agent to play the game Flappy Bird. We use Q-learning and Double Deep Q-Networks to train two agents separately. The agent gets the information about the location of the bird and the pipes. According to the information, agents evaluate the Q-functions. Both agents show excellent performance. Furthermore, we discuss the performance of the two agents.**

1. Introduction

Reinforcement learning is a branch of machine learning. Compared with classical supervised learning and unsupervised learning problems, the biggest characteristic of reinforcement learning is learning from interaction. Agent constantly learns knowledge according to the reward or punishment obtained in the interaction with the environment. The pattern of reinforcement learning is similar to the way of human learning knowledge. Therefore, reinforcement learning is an important way to implement artificial intelligence.

In Reinforcement learning, if the number of the states is large, it really difficult to represent the Q-function and the agent can not determine the actions for the states never visit. deep reinforcement learning can help to solve the problem.

The goal of the project is to learn a policy to get an agent do an excellent job on playing the game Flappy Bird. Flappy Bird is a very simple game but hard to play. The player needs to keep the bird alive. The bird automatically falls towards the ground. Every time the player clicks the bird, the bird will have the power to fly higher for a while. The bird will keep flying towards and there are pipes in front of the bird. The bird needs to fly through the area between the two pipes. If the bird hit the pipe or the ground, the bird will die. Therefore the player needs the control the bird to fly through the obstacles. Once the bird flies through a pair of pipes, the player will get one score.

We use the Flappy Bird game from the pygame package and reference the clone of Flappy Bird game of sourabhv in Github. The location information about the bird and pipes are given to the agent. It is very difficult to program code in the way of logical judgement to solve the problem because even the programmer itself cannot find when to click the bird to fly higher.

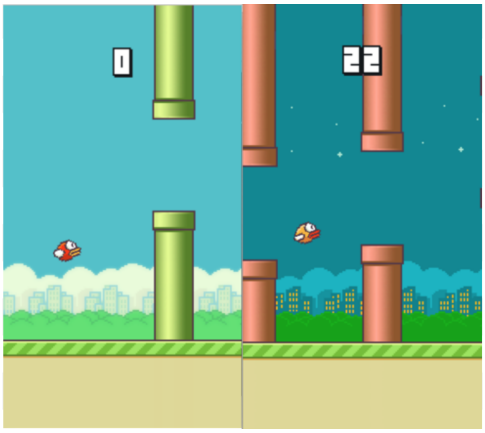

Figure 1. Two screenshots of the game Flappy bird.

2. Related Work

Q-learning algorithm, originally proposed by Watkins in his doctoral dissertation [1], is a very effective learning algorithm in reinforcement learning. Every step, the agent will get the reward or the punishment. The algorithm tries to find the policy the maximize the reward and avoid the punishment according to the historical experience. It's traditional but work very well to the problems which have no single correct way to solve.

There are many people try to combine deep learning and reinforcement learning, but the first one to do it is DeepMind [2]. They try to use CNN to extract features from high-dimensional sensory inputs and then uses a reinforcement learning algorithm to learn control strategies. The trained agents have been tested in 7 Atari 2600 games and performed very well in 6 games. Agents played better than human experts in 3 games. The learning model is the combination of the CNN and Q-learning, so it's called deep Q-network(DQN).

In 2016, DeepMind came up with the Double DQN to

TABLE 1. THE CONTENT OF THE STATES

Element	Description
isDead	False
upperPipes	'x': 488, 'y': -219, 'x': 632.0, 'y': -154
upperPipes	'x': 488, 'y': -219, 'x': 632.0, 'y': -154
lowerPipes	'x': 488, 'y': 211, 'x': 632.0, 'y': 276
score	0
playerVelY	-9
playerRot	45
pipeVelX	-4
playerAccY	1
basex	0
playerx	57
playery	244

improve the original deep Q-network 's performance [3]. Double DQN reduces the overestimation of the DQN and performs better.

3. Markov Decision Process Formulation

In Markov Decision Process [4], actions are indeterminate, and the cost function depends on the starting and ending states of the action. A MDP consists of states, actions, reward function, goal condition and a discount factor. In this section, we describe how the model is parameterized. Because we use two different methods to solve the problem, we model the game differently.

3.1. In Q-Learning

The state of the Flappy Bird game is represented by the bird's coordination and pipes' coordination. The original states we get from pygame are in JSON format and the content is shown in Table 1.

In the original state, there is much useless information. Before training, we preprocess the states into the ideal format. After we analyze the game, we find that we only need the distances between the bird and the next lower pipe in x-direction and y-direction to train the agent. Therefore, we transform the original state into $gridX\_gridY\_v$ format. $gridX$ represents the distance between the bird and the next first lower pipe in the x-direction. The $gridY$ represents the distance between the bird and the next first lower pipe in the y-direction. v represents the velocity of the bird falling down. We define the initial state is 420_240_0.

There are only two kinds of actions we can choose: click the bird to make it fly higher or do nothing. According to the game, every time the bird passes a pair of pipes, the score will be increased by 1. If the bird died, the game will be over. We should give a large punishment for letting the bird died. Therefore, we define that in the current state, if the bird is still alive, the agent can get 1 unit as a reward, otherwise, the agent gets -1000 unit as punishment. If the bird died because of crashing at the upper pipe and there is a click action in the last 3 states, the agent gets -1000 unit as punishment once. When the number of running times reaches the condition we set, the training is over. To simplify the process, we define the discount is 1.

Figure 2. The pictorial description of the state information.

TABLE 2. MODEL DEFINITION FOR Q-LEARING

Element	Description
state(s)	gridX_gridY_v
Initial state(s_0)	420_240_0
Action(a)	Click or do nothing
Reward function(r)	Alive: 1;Died: -1000 Alive but take action click leading died: -1000
Goal condition (End(s))	the number of running times reaches the condition
Discount factor(γ)	1

3.2. In Double Deep Q-Network

In DDQN we also need to preprocess the original states and we decide to use [gridX, gridY, v, playerRot] to represent the states. Compared with the states in Q-learning, the rotation of the bird is contained in the states. The starting state is automatically calculated for the bird and the random initial pipes. The actions are the same as the actions in Q-learning.

In our analysis, if the bird already gets high scores, the agent should get a higher reward for maintaining the scores. So we define the reward is the scores the bird maintained in the current states. In case the bird is flying too high, every time the gridY is larger than 250, the reward will minus 200. If the bird died, the agent will get -1000 for punishment.

When the number of running times reaches the condition we set, the training is over. For the performance of the training, the discount factor is 0.99.

TABLE 3. MODEL DEFINITION FOR Q-LEARING

Element	Description
state(s)	[gridX, gridY, v, playerRot]
Initial state(s_0)	Random
Action(a)	Click or do nothing
Reward function (r)	Alive and $gridY <= 250$: reward = score ; Alive and $gridY > 250$: reward = score - 200; Died: reward = score -1000;
Goal condition (End(s))	the number of running times reaches the condition
Discount factor(γ)	0.99

4. Method

4.1. Q-Learning

In Q-learning algorithm, Q-function $Q(s, a)$ represents the utility of doing action a in state s. We need to maintain every action's Q-function in every state s. The representation is a kind of table. We always want to maximize the utility. When we need to choose an action at a specific state, we can look up the table to find the best action that gives us the highest utility. According to the Q-learning algorithm, we will update the Q-function as equation (1).

$$Q(s,a) = (1-\mu) * Q(s,a) + \mu(r + \gamma * \max_{a'} Q(s',a')) \quad (1)$$

In equation (1), μ represent the learning rate. Learning rate control the balance between the new knowledge and the old experience.$Q(s', a')$ represents the Q-function of next state s' after taking action a'. Before we stop the training, the Q-function keeps on updating. The algorithm 1 shows the whole procedure for training the agent.

Algorithm 1: Q-Learning for Flappy Bird

```
Initialize Q(s, a)
for the number of running times smaller than the
 condition do
    for state s, pick action a which has maximal
     Q(s, a)
    put [s,a,s'] into the actions list
    bird gets a new state s'
    if the bird is dead then
        foreach state record [s,a,s'] in the actions
         list do
            NewQvalue = r + γ * max_a' Q(s', a')
            Q(s, a) = (1 − μ) * Q(s, a) + μ * NewQvalue
        end
        set actions list to empty
    end
end
```

4.2. Double Deep Q-Network

Q-learning algorithm gives a good way to train the agent, but we need to store all the Q(s, a) for every possible state. The agent can only choose a random action for the states never visited. So, we try to use Double Deep Q-Network to train the agent.

4.2.1. Deep Q-Network with target network.

DDQN is the improve version of Deep Q-Network(DQN). DQN is a value based algorithm. DQN algorithm tries to use a neural network to find an optimal estimation $Q(s, a; \theta)$ of Q-function for every state. For training the neural network we need to find an objective function, so that $Q(s, a; \theta)$ can get closer to the true target $Q(s, a)$. Because we don't know the target, so the algorithm use TD target in equation (2).

$$y = r(s,a) + \gamma * \max_{a'} Q(s', a'; \theta) \quad (2)$$

According to equation (2), we can get the loss function equation (3).

$$\begin{aligned} L(\theta) &= (y - Q(s,a;\theta))^2 \\ &= (r(s,a) + \gamma * \max_{a'} Q(s',a';\theta) - Q(s,a;\theta))^2 \end{aligned} \quad (3)$$

The upating rule is equation (4).

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta) \quad (4)$$

Because the target is always changing, the algorithm defines a target network that has the same structure as DQN. let θ^- be the parameters. We need to set $\theta^- = \theta$ once in a while.So the TD target is now defined as equation (5).

$$y = r(s,a) + \gamma * \max_{a'} Q(s', a'; \theta^-) \quad (5)$$

4.2.2. Experience Replay.

There is a problem with the training process. In reinforcement learning, the experiences we got from the same episode is step by step. They are very correlated. This kind of data will lead to inefficient training. So the idea of experience replay is used to update the parameters. Every time we get a new state, we put the state into a fixed size buffer. If the buffer is full, we randomly get experiences from the buffer to train the neural network. As a result, our experiences are no longer likely to be correlated.

4.2.3. DDQN.

In DQN, the algorithm use equation (5) to estimate the TD target which causes the problem of overestimation of the ideal value. In order to maximize the Q-function of the state s, we always want to choose the best action. In equation (5), the action a is chosen when it can maximize the Q-function of the target network. In DDQN, the action a is chosen when it can maximize the Q-function of the training network of

the DQN, which makes the TD target is much closer to the ideal value. The TD target is shown in equation (6)

$$\begin{aligned} y &= r(s,a) + \gamma * Q(s', a^*; \theta^-) \\ a^* &= \arg\max_{a'} Q(s', a'; \theta) \end{aligned} \tag{6}$$

The algorithm 2 shows the procedure of using DDQN in the flappy bird.

Algorithm 2: DDQN for Flappy Bird

```
Initialize: ReplayBuffer is empty
Initialize: gamma = 0.99
Initialize: define trainNetwork with weight θ
Initialize: define targetNetwork which is the same
  as trainNetwork with weight θ⁻
foreach episode e ∈ [1, 2, ...400] do
    when state changes, get bird's old state s,
      action a, new state s', score and exploreTime.
    reward = score
    if girdY > 250 then
        reward = reward − 200
    end
    if the bird is dead then
        θ⁻ = θ
        put[s, a, reward − 1000,s',Alive or Dead]
          into ReplayBuffer
    else
        put[s, a, reward,s',Alive or Dead] into
          ReplayBuffer
    end
    if exploreTime > 1000 then
        get random samples from ReplayBuffer
        foreach sample sam ∈ samples do
            if the bird is dead then
                y[sam][a] = reward
            else
                get the action a* that maximize the
                  trainQvalue of s' in trainNetwork
                get the targetQvalue in
                  targetNetwork with s' and a*
                y[sam][a*] = reward + γ * targetQvalue
            end
        end
        train the trainNetwork with y
    end
    if score >= 10 and score mod 10 == 0 then
        save the model
    end
end
```

4.3. Exploration and Exploitation

When we training, we always want to explore new states. In the same time, we want to use our experience to get better performance. So we define the probability ϵ. In training, we select a random action with probability ϵ and choose the optimal action from our training model with probability $1 - \epsilon$.

Besides, we find that if we choose action click, the bird will fly higher in the next several states. So we give less probability in selecting click among the random choices.

TABLE 4. THE DEFINITION OF THE QAGENT

Attribute and function	Description
self.dumpNumber = 30	the time to save model
self.gameCount = 0	the number of the game iterations
self.discount = 1.0	discount factor
self.lState = 420_240_0	last state
self.lAction = 0	last action
self.actions = []	actions list
self.istrain = istrain	whether it is a training
load_qvalues(self)	load value from qvalues.json
dump_qvalues(self, exit = False)	save model
act(self, cs)	get action from state
update(self, dump_qvalues = True)	updating model
stateToGrid(self,x, y, x1, y1, x2, y2, v)	preprocess the states

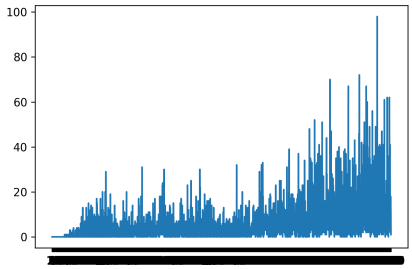

Figure 3. The scores of QAgent with ϵ after QAgent played 3300 times.

5. Result

A video can be found at the following link: http://youtube.com. Our metric for evaluating the performance of the two algorithm is the game scores. Both of the two agents give excellent performance.

5.1. Result of Q-learning

In Q-learning, we define a QAgent to play the game. Table 4 shows the specific definition of QAgent.

We first let the QAgent with ϵ play the game 3300 times. When the QAgent chooses random actions, the QAgent may die. So we set the ϵ is 0.008. Because of the randomness, the performance while training is not so good. Figure 3 shows the performance while training. We can see that the scores slowly increases while playing. The maximal score is 98 and the average score is 5.

After we train the QAgent with ϵ , we use QAgent which totally depend on the experience to play the game. The result

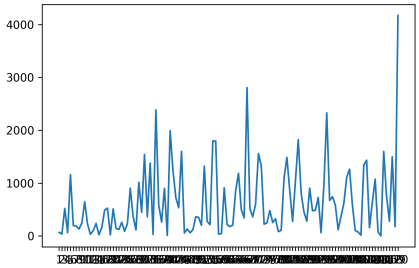

Figure 4. The scores of QAgent without ϵ after QAgent played 120 times.

TABLE 5. THE DEFINITION OF THE DEEPAGENT

Attribute and function	Description
self.model=DeepTrain()	define the training model
self.syncCount=0	the count of the states' change
self.tryTime=0	the number of the game iterations
self.exploreTime=0	the times of the random actions the agent choosee
onStateChange(self, oldState, action, newState)	some process needs to do when state changes
getNextAction(self, state)	return the action
processState(self,rawState)	preprocess State
onDone(self,reward)	some process needs to do when bird died

is shown in Figure 4. After 120 games, the average score is 635 which is much higher than 98. The maximal score is 4171 which shows excellent performance of QAgent.

5.2. Result of DDQN

In DDQN, we define a DeepAgent to play the game and a DeepTrain model to train the DeepAgent. Table 5 shows the specific definition of DeepAgent and Table 6 shows the specific definition of DeepTrain model.

In DDQN, we set the epsilon is 0.2 at first and the epsilon will minus 0.001 when the DeepAgent died with a score higher than 0. Eventually, the epsilon will decay to 0.001. We let the DeepAgent play the game 1292 times. Figure 5 shows the performance while training.

Due to the effectiveness of epsilon, the DeepAgent shows a really bad performance at the beginning. After playing 800 times, the DeepAgent begin to get more scores. In the last 200 iterations, the average score is 72. In DDQN, the input feature dimension is 4 and neural network layers have at most 64 hidden units which mean we have plenty of parameters to train. So the neural network needs much more time to train.

6. Conclusion

We can successfully play the game Flappy Bird by using the two agent we trained. Both of them give better performance than human. Q-learning and DDQN are used to train agents. The two algorithms have different parameters and different training process. Because the neural network needs much more time to train, we can not compare them according to the number of the game iterations. According to Figure 3 and Figure 5, we can roughly find that QAgent is improved steadily while DeepAgent gets higher scores during training. We can make a wild guess that the DeepAgent will show better performance if it get more time to train.

The preprocess with the game Flappy Bird and the training of DDQN is done by Pei Yulin. The training of Q-learning is done by HE Tianlang and XU Haoran. The method analysis and report writing are done by DONG Ziwei. The video is made by XU Haoran.

TABLE 6. THE DEFINITION OF THE DEEPTRAIN MODEL

Attribute and function	Description
self.stateShape=4	the size of the state
self.actionSpaceNum=2	the size of action space
self.gamma = 0.99	the discount factor
self.epsilon = 0.2	epsilon(ϵ)
self.epsilon_decay = 0.001	the decay amount of epsilon
self.epsilon_min = 0.0001	the minimum of epsilon
self.learingRate = 0.0001	learning rate
self.replayBuffer = deque(maxlen=60000)	replay Buffer
self.trainNetwork = self.createNetwork()	the network used to train the Qvalue
self.numPickFromBuffer = 32*2	the size of the samples picked from the replay Buffer
self.targetNetwork = self.createNetwork()	the target network
self.targetNetwork.set_weights (self.trainNetwork.get_weights())	make the parameters of the target network same as the trainNetwork
createNetwork(self)	create a neural network
getBestAction(self, state)	return the action
trainFromBuffer(self)	train the model

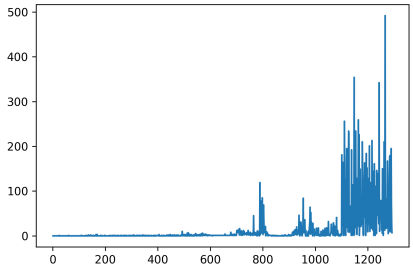

Figure 5. The scores of DeepAgent after DeepAgent played 1292 times.

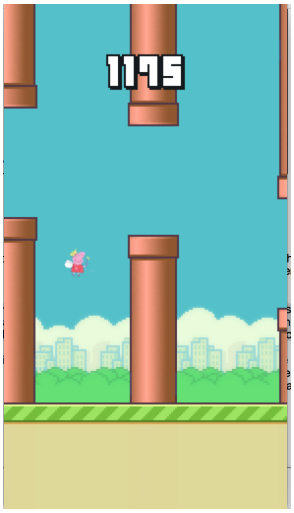

Figure 6. The screen shot of Flappy Bird

Acknowledgments

The authors wish to thank Professor Nevin L. Zhang from the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, for providing the necessary advice. The authors also like to thank the sourabhv for providing the clone of the Flappy Bird game on Github. The authors also would like to thank SarvagyaVaish for providing the Q-learning analysis on game Flappy Bird.

References

[1] Watkins C J C H, Dayan P. Q-learning[J]. Machine learning, 1992, 8(3-4): 279-292.

[2] Mnih V, Kavukcuoglu K, Silver D, et al. Playing atari with deep reinforcement learning[J]. arXiv preprint arXiv:1312.5602, 2013.

[3] Van Hasselt H, Guez A, Silver D. Deep reinforcement learning with double q-learning[C]//Thirtieth AAAI Conference on Artificial Intelligence. 2016.

[4] Russell S J, Norvig P. Artificial intelligence: a modern approach[M]. Malaysia; Pearson Education Limited,, 2016.