School of Data and Computer Science, Sun Yat-sen University

E16 Deep Q-Learning (C++/Python)

18340215 张天祎

2021 年 1 月 4 日

Contents

1 Deep Q-Network (DQN)

We consider tasks in which an agent interacts with an environment $\mathcal{E}$, in this case the Atari emulator, in a sequence of actions, observations and rewards. At each time-step the agent selects an action a_t from the set of legal game actions, $\mathcal{A} = \{1, ..., K\}$. The action is passed to the emulator and modifies its internal state and the game score. In general $\mathcal{E}$ may be stochastic. The emulator's internal state is not observed by the agent, instead it observes an image $x_t \in \mathbb{R}^d$ from the emulator, which is a vector of raw pixel values representing the current screen. In addition it receives a reward r_t representing the change in game score. Note that in general the game score may depend on the whole prior sequence of actions and observations; feedback about an action may only be received after many thousands of time-steps have elapsed.

Since the agent only observes images of the current screen, the task is partially observed and many emulator states are perceptually aliased, i.e. it is impossible to fully understand the current situation from only the current screen x_t. We therefore consider sequences of actions and observations, $s_t = x_1, a_1, x_2, ..., a_{t-1}, x_t$, and learn game strategies that depend upon these sequences. All sequences in the emulator are assumed to terminate in a finite number of time-steps. This formalism gives rise to a large but finite Markov decision process (MDP) in which each sequence is a distinct state. As a result, we can apply standard reinforcement learning methods for MDPs, simply by using the complete sequence s_t as the state representation at time t.

The goal of the agent is to interact with the emulator by selecting actions in a way that maximises future rewards. We make the standard assumption that future rewards are discounted by a factor of γ per time-step, and define the future discounted *return* at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the time-step at which the game terminates. We define the optimal action-value function $Q^*(s, a)$ as the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action a, $Q^*(s, a) = \max_\pi \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$, where π is a policy mapping sequences to actions (or distributions over actions).

The optimal action-value function obeys an important identity known as the *Bellman equation*. This is based on the following intuition: if the optimal value $Q^*(s', a')$ of the sequence s' at the next time-step was known for all possible actions a', then the optimal strategy is to select the action a' maximising the expected value of $r + \gamma Q^*(s', a')$,

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') \Big| s, a\right] \tag{1}$$

The basic idea behind many reinforcement learning algorithms is to estimate the action-value function, by using the Bellman equation as an iterative update, $Q_{i+1}(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s', a') | s, a\right]$.

Such *value iteration* algorithms converge to the optimal action-value function, $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$. In practice, this basic approach is totally impractical, because the action-value function is estimated separately for each sequence, without any generalisation. Instead, it is common to use a function approximator to estimate the action-value function, $Q(s, a; \theta) \approx Q^*(s, a)$. In the reinforcement learning community this is typically a linear function approximator, but sometimes a non-linear function approximator is used instead, such as a neural network. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q\left(s, a; \theta_i\right)\right)^2\right], \tag{2}$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a that we refer to as the *behaviour distribution*. The parameters from the previous iteration θ_{i-1} are held fixed when optimising the loss function $L_i\left(\theta_i\right)$. Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before learning begins. Differentiating the loss function with respect to the weights we arrive at the following gradient,

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)\nabla_{\theta_i} Q(s, a; \theta_i)\right]. \tag{3}$$

Rather than computing the full expectations in the above gradient, it is often computationally expedient to optimise the loss function by stochastic gradient descent. If the weights are updated after every time-step, and the expectations are replaced by single samples from the behaviour distribution ρ and the emulator $\mathcal{E}$ respectively, then we arrive at the familiar *Q-learning* algorithm.

Note that this algorithm is *model-free*: it solves the reinforcement learning task directly using samples from the emulator $\mathcal{E}$, without explicitly constructing an estimate of $\mathcal{E}$. It is also *off-policy*: it learns about the greedy strategy $a = \max_a Q(s, a; \theta)$, while following a behaviour distribution that ensures adequate exploration of the state space. In practice, the behaviour distribution is often selected by an ϵ-greedy strategy that follows the greedy strategy with probability $1 - \epsilon$ and selects a random action with probability ϵ.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N

Initialize action-value function Q with random weights

for episode $= 1, M$ **do**

  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

  for $t = 1, T$ **do**

    With probability ϵ select a random action a_t

    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$

    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

  end for

end for

2 Deep Learning Flappy Bird

Overview

This project (`https://github.com/yenchenlin/DeepLearningFlappyBird`) follows the description of the Deep Q Learning algorithm described in *Playing Atari with Deep Reinforcement Learning* and shows that this learning algorithm can be further generalized to the notorious Flappy Bird.

Installation Dependencies:

- Python 2.7 or 3
- TensorFlow 0.7
- pygame
- OpenCV-Python

How to Run?

```
git clone https://github.com/yenchenlin1994/DeepLearningFlappyBird.git
```

```
cd DeepLearningFlappyBird
python deep_q_network.py
```

What is Deep Q-Network?

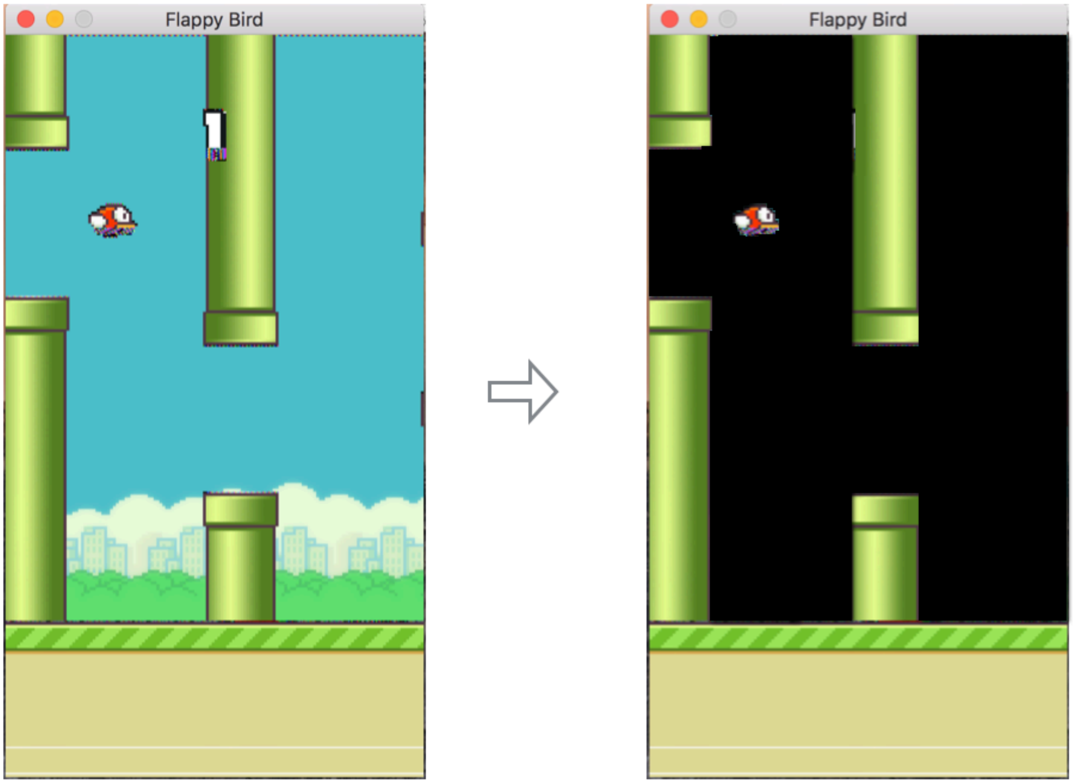
It is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards.

For those who are interested in deep reinforcement learning, I highly recommend to read the following post: Demystifying Deep Reinforcement Learning

Deep Q-Network Algorithm

The pseudo-code for the Deep Q Learning algorithm can be found below:

```
Initialize replay memory D to size N
Initialize action-value function Q with random weights
for episode = 1, M do
    Initialize state s_1
    for t = 1, T do
        With probability  select random action a_t
        otherwise select a_t=max_a  Q(s_t,a;  _i)
        Execute action a_t in emulator and observe r_t and s_(t+1)
        Store transition (s_t,a_t,r_t,s_(t+1)) in D
        Sample a minibatch of transitions (s_j,a_j,r_j,s_(j+1)) from D
        Set y_j:=
            r_j for terminal s_(j+1)
            r_j+ *max_(a^' )  Q(s_(j+1),a';  _i) for non-terminal s_(j+1)
        Perform a gradient step on (y_j-Q(s_j,a_j;  _i))^2 with respect to
    end for
end for
```



Experiments

Environment

Since deep Q-network is trained on the raw pixel values observed from the game screen at each time step, so removing the background appeared in the original game can make it converge faster.

This process can be visualized as the following figure:

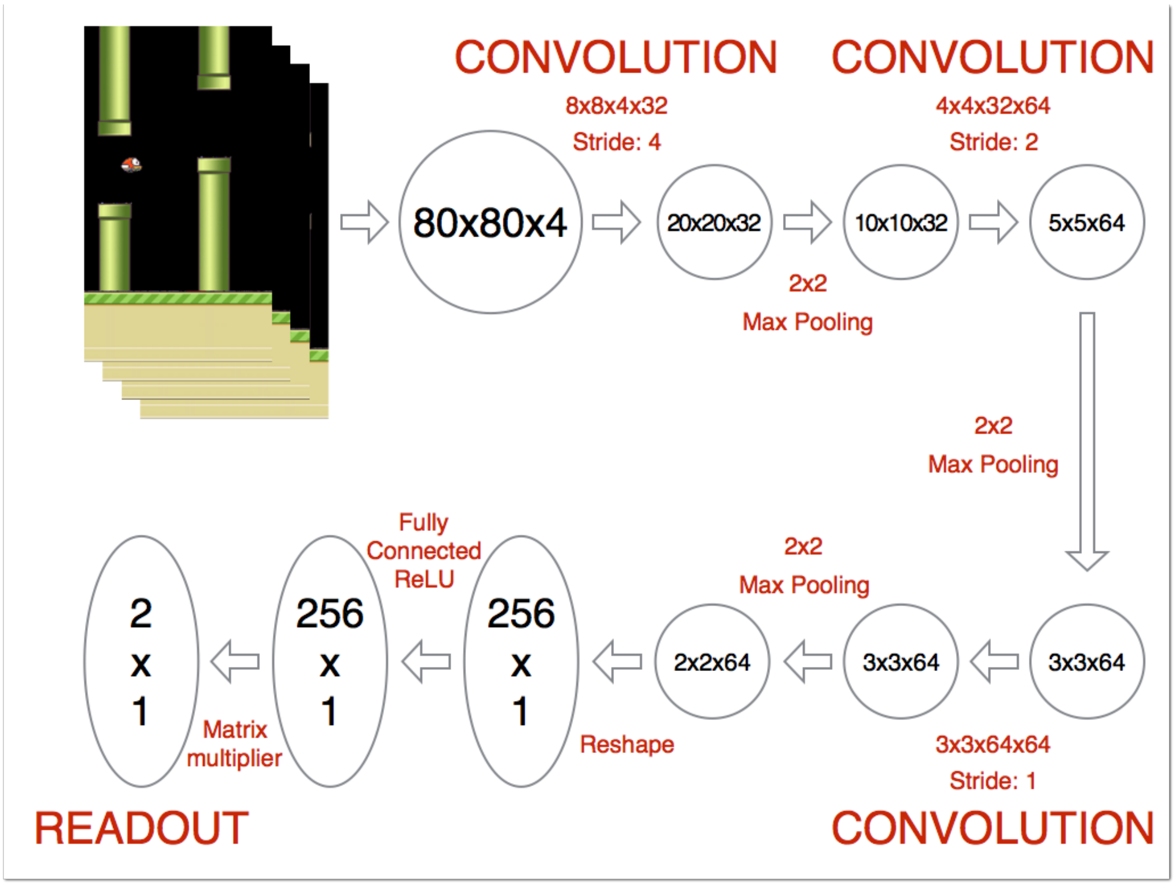
Network Architecture

I first preprocessed the game screens with following steps:

1. Convert image to grayscale
2. Resize image to 80x80
3. Stack last 4 frames to produce an $80 \times 80 \times 4$ input array for network

The architecture of the network is shown in the figure below. The first layer convolves the input image with an $8 \times 8 \times 4 \times 32$ kernel at a stride size of 4. The output is then put through a 2×2 max pooling layer. The second layer convolves with a $4 \times 4 \times 32 \times 64$ kernel at a stride of 2. We then max pool again. The third layer convolves with a $3 \times 3 \times 64 \times 64$ kernel at a stride of 1. We then max pool one more time. The last hidden layer consists of 256 fully connected ReLU nodes.

The final output layer has the same dimensionality as the number of valid actions which can be



performed in the game, where the 0th index always corresponds to doing nothing. The values at this output layer represent the Q function given the input state for each valid action. At each time step, the network performs whichever action corresponds to the highest Q value using a ϵ greedy policy.

Training

At first, I initialize all weight matrices randomly using a normal distribution with a standard deviation of 0.01, then set the replay memory with a max size of 500,00 experiences.

I start training by choosing actions uniformly at random for the first 10,000 time steps, without updating the network weights. This allows the system to populate the replay memory before training begins.

I linearly anneal $\epsilon$from 0.1 to 0.0001 over the course of the next 3000,000 frames. The reason why I set it this way is that agent can choose an action every 0.03s (FPS=30) in our game, high ϵ will make it **flap** too much and thus keeps itself at the top of the game screen and finally bump the pipe in a clumsy way. This condition will make Q function converge relatively slow since it only start to look other conditions when ϵ is low. However, in other games, initialize ϵ to 1 is more reasonable.

During training time, at each time step, the network samples minibatches of size 32 from the replay memory to train on, and performs a gradient step on the loss function described above using the Adam optimization algorithm with a learning rate of 0.000001. After annealing finishes, the network continues to train indefinitely, with ϵ fixed at 0.001.

3 Tasks

1. Please implement a DQN to play the Flappy Bird game.
2. You can refer to the codes in https://github.com/yenchenlin/DeepLearningFlappyBird
3. Please submit a file named `E18_YourNumber.zip`, which should includes the code files and the result pictures, and send it to `ai_2020@foxmail.com`

4 Codes and Results

```
#!/usr/bin/env python
from __future__ import print_function

# import tensorflow as tf
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import cv2
import sys
sys.path.append("game/")
import wrapped_flappy_bird as game
import random
import numpy as np
from collections import deque

GAME = 'bird' # the name of the game being played for log files
ACTIONS = 2 # number of valid actions
GAMMA = 0.99 # decay rate of past observations
OBSERVE = 100000. # timesteps to observe before training
EXPLORE = 2000000. # frames over which to anneal epsilon
FINAL_EPSILON = 0.0001 # final value of epsilon
INITIAL_EPSILON = 0.0001 # starting value of epsilon
REPLAY_MEMORY = 50000 # number of previous transitions to remember
BATCH = 32 # size of minibatch
FRAME_PER_ACTION = 1

def weight_variable(shape):
```

```
    initial = tf.truncated_normal(shape, stddev = 0.01)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.01, shape = shape)
    return tf.Variable(initial)

def conv2d(x, W, stride):
    return tf.nn.conv2d(x, W, strides = [1, stride, stride, 1], padding = "SAME")

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize = [1, 2, 2, 1], strides = [1, 2, 2, 1], padding = "
        SAME")

def createNetwork():
    # network weights
    W_conv1 = weight_variable([8, 8, 4, 32])
    b_conv1 = bias_variable([32])

    W_conv2 = weight_variable([4, 4, 32, 64])
    b_conv2 = bias_variable([64])

    W_conv3 = weight_variable([3, 3, 64, 64])
    b_conv3 = bias_variable([64])

    W_fc1 = weight_variable([1600, 512])
    b_fc1 = bias_variable([512])

    W_fc2 = weight_variable([512, ACTIONS])
    b_fc2 = bias_variable([ACTIONS])

    # input layer
    s = tf.placeholder("float", [None, 80, 80, 4])

    # hidden layers
    h_conv1 = tf.nn.relu(conv2d(s, W_conv1, 4) + b_conv1)
    h_pool1 = max_pool_2x2(h_conv1)

    h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2, 2) + b_conv2)
    #h_pool2 = max_pool_2x2(h_conv2)

```

```
    h_conv3 = tf.nn.relu(conv2d(h_conv2, W_conv3, 1) + b_conv3)
    #h_pool3 = max_pool_2x2(h_conv3)

    #h_pool3_flat = tf.reshape(h_pool3, [-1, 256])
    h_conv3_flat = tf.reshape(h_conv3, [-1, 1600])

    h_fc1 = tf.nn.relu(tf.matmul(h_conv3_flat, W_fc1) + b_fc1)

    # readout layer
    readout = tf.matmul(h_fc1, W_fc2) + b_fc2

    return s, readout, h_fc1

def trainNetwork(s, readout, h_fc1, sess):
    # define the cost function
    a = tf.placeholder("float", [None, ACTIONS])
    y = tf.placeholder("float", [None])
    readout_action = tf.reduce_sum(tf.multiply(readout, a), reduction_indices=1)
    cost = tf.reduce_mean(tf.square(y - readout_action))
    train_step = tf.train.AdamOptimizer(1e-6).minimize(cost)

    # open up a game state to communicate with emulator
    game_state = game.GameState()

    # store the previous observations in replay memory
    D = deque()

    # printing
    a_file = open("logs_" + GAME + "/readout.txt", 'w')
    h_file = open("logs_" + GAME + "/hidden.txt", 'w')

    # get the first state by doing nothing and preprocess the image to 80x80x4
    do_nothing = np.zeros(ACTIONS)
    do_nothing[0] = 1
    x_t, r_0, terminal = game_state.frame_step(do_nothing)
    x_t = cv2.cvtColor(cv2.resize(x_t, (80, 80)), cv2.COLOR_BGR2GRAY)
    ret, x_t = cv2.threshold(x_t,1,255,cv2.THRESH_BINARY)
    s_t = np.stack((x_t, x_t, x_t, x_t), axis=2)

    # saving and loading networks
    saver = tf.train.Saver()
```

```
    sess.run(tf.initialize_all_variables())
    checkpoint = tf.train.get_checkpoint_state("saved_networks")
    if checkpoint and checkpoint.model_checkpoint_path:
        saver.restore(sess, checkpoint.model_checkpoint_path)
        print("Successfully loaded:", checkpoint.model_checkpoint_path)
    else:
        print("Could not find old network weights")

    # start training
    epsilon = INITIAL_EPSILON
    t = 0
    while "flappy bird" != "angry bird":
        # choose an action epsilon greedily
        readout_t = readout.eval(feed_dict={s : [s_t]})[0]
        a_t = np.zeros([ACTIONS])
        action_index = 0
        if t % FRAME_PER_ACTION == 0:
            if random.random() <= epsilon:
                print("----------------Random Action----------------")
                action_index = random.randrange(ACTIONS)
                a_t[random.randrange(ACTIONS)] = 1
            else:
                action_index = np.argmax(readout_t)
                a_t[action_index] = 1
        else:
            a_t[0] = 1 # do nothing

        # scale down epsilon
        if epsilon > FINAL_EPSILON and t > OBSERVE:
            epsilon -= (INITIAL_EPSILON - FINAL_EPSILON) / EXPLORE

        # run the selected action and observe next state and reward
        x_t1_colored, r_t, terminal = game_state.frame_step(a_t)
        x_t1 = cv2.cvtColor(cv2.resize(x_t1_colored, (80, 80)), cv2.COLOR_BGR2GRAY)
        ret, x_t1 = cv2.threshold(x_t1, 1, 255, cv2.THRESH_BINARY)
        x_t1 = np.reshape(x_t1, (80, 80, 1))
        #s_t1 = np.append(x_t1, s_t[:,:,1:], axis = 2)
        s_t1 = np.append(x_t1, s_t[:, :, :3], axis=2)

        # store the transition in D
        D.append((s_t, a_t, r_t, s_t1, terminal))
```

```
        if len(D) > REPLAY_MEMORY:
            D.popleft()

        # only train if done observing
        if t > OBSERVE:
            # sample a minibatch to train on
            minibatch = random.sample(D, BATCH)

            # get the batch variables
            s_j_batch = [d[0] for d in minibatch]
            a_batch = [d[1] for d in minibatch]
            r_batch = [d[2] for d in minibatch]
            s_j1_batch = [d[3] for d in minibatch]

            y_batch = []
            readout_j1_batch = readout.eval(feed_dict = {s : s_j1_batch})
            for i in range(0, len(minibatch)):
                terminal = minibatch[i][4]
                # if terminal, only equals reward
                if terminal:
                    y_batch.append(r_batch[i])
                else:
                    y_batch.append(r_batch[i] + GAMMA * np.max(readout_j1_batch[i]))

            # perform gradient step
            train_step.run(feed_dict = {
                y : y_batch,
                a : a_batch,
                s : s_j_batch}
            )

        # update the old values
        s_t = s_t1
        t += 1

        # save progress every 10000 iterations
        if t % 10000 == 0:
            saver.save(sess, 'saved_networks/' + GAME + '-dqn', global_step = t)

        # print info
        state = ""
```

```
        if t <= OBSERVE:
            state = "observe"
        elif t > OBSERVE and t <= OBSERVE + EXPLORE:
            state = "explore"
        else:
            state = "train"

        print("TIMESTEP", t, "/ STATE", state, \
            "/ EPSILON", epsilon, "/ ACTION", action_index, "/ REWARD", r_t, \
            "/ Q_MAX %e" % np.max(readout_t))
        # write info to files
        '''
        if t % 10000 <= 100:
            a_file.write(",".join([str(x) for x in readout_t]) + '\n')
            h_file.write(",".join([str(x) for x in h_fc1.eval(feed_dict={s:[s_t]})[0]]) + '\n')
            cv2.imwrite("logs_tetris/frame" + str(t) + ".png", x_t1)
        '''

def playGame():
    sess = tf.InteractiveSession()
    s, readout, h_fc1 = createNetwork()
    trainNetwork(s, readout, h_fc1, sess)

def main():
    playGame()

if __name__ == "__main__":
    main()
```

本地因为安装了 tensorflow2.*，而原代码为 tensorflow0.7，所以仅需要在 import 库时做一下额外处理，把库的版本降低即可运行。原代码的结果已经足够好了，没有进行调参。训练次数特别多，结果如下：

```
TIMESTEP 11361 / STATE observe / EPSILON 0.0001 / ACTION 0 / REWARD 0.1 / Q_MAX 1.275663e+01
TIMESTEP 11362 / STATE observe / EPSILON 0.0001 / ACTION 0 / REWARD 0.1 / Q_MAX 1.277851e+01
TIMESTEP 11363 / STATE observe / EPSILON 0.0001 / ACTION 0 / REWARD 0.1 / Q_MAX 1.272854e+01
TIMESTEP 11364 / STATE observe / EPSILON 0.0001 / ACTION 0 / REWARD 0.1 / Q_MAX 1.275863e+01
TIMESTEP 11365 / STATE observe / EPSILON 0.0001 / ACTION 0 / REWARD 0.1 / Q_MAX 1.287276e+01
TIMESTEP 11366 / STATE observe / EPSILON 0.0001 / ACTION 0 / REWARD 0.1 / Q_MAX 1.296755e+01
TIMESTEP 11367 / STATE observe / EPSILON 0.0001 / ACTION 0 / REWARD 0.1 / Q_MAX 1.284153e+01
TIMESTEP 11368 / STATE observe / EPSILON 0.0001 / ACTION 0 / REWARD 0.1 / Q_MAX 1.273492e+01
TIMESTEP 11369 / STATE observe / EPSILON 0.0001 / ACTION 1 / REWARD 0.1 / Q_MAX 1.292669e+01
TIMESTEP 11370 / STATE observe / EPSILON 0.0001 / ACTION 0 / REWARD 0.1 / Q_MAX 1.300509e+01
TIMESTEP 11371 / STATE observe / EPSILON 0.0001 / ACTION 0 / REWARD 0.1 / Q_MAX 1.303499e+01
TIMESTEP 11372 / STATE observe / EPSILON 0.0001 / ACTION 0 / REWARD 0.1 / Q_MAX 1.297799e+01

Process finished with exit code -1
```

训练时可以看到小鸟飞了近 100 次左右才碰壁，效果拔群。