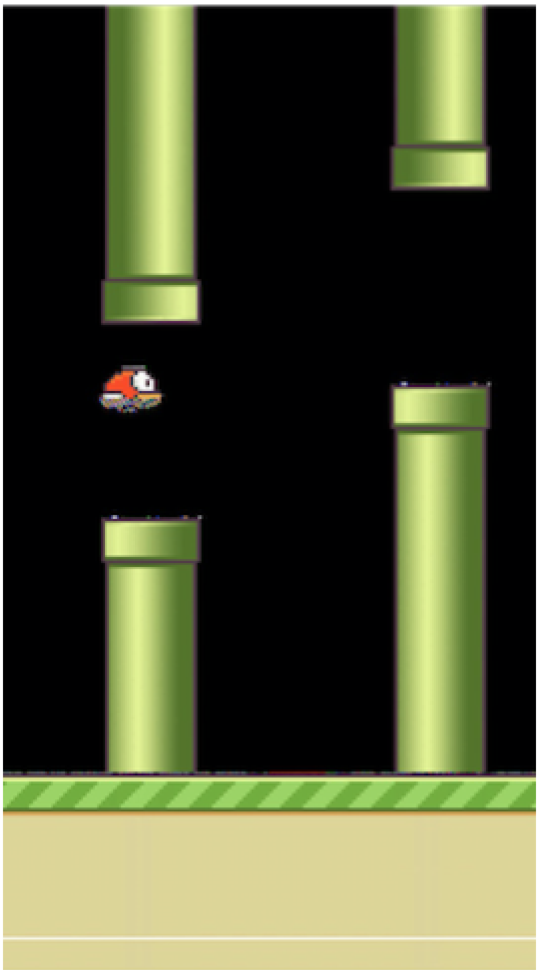
前言

2013年DeepMind 在NIPS上发表Playing Atari with Deep Reinforcement Learning 一文，提出了DQN（Dee Network）算法，实现端到端学习玩Atari游戏，即只有像素输入，看着屏幕玩游戏。Deep Mind就凭借 应用以6亿美元被Google收购。由于DQN的开源，在github上涌现了大量各种版本的DQN程序。但大多 复现Atari的游戏，代码量很大，也不好理解。

Flappy Bird是个极其简单又困难的游戏，风靡一时。在很早之前，就有人使用Q-Learning 算法来实现完 Flappy Bird。http://sarvagyavaish.github.io/FlappyBirdRL/
但是这个的实现是通过获取小鸟的具体位置信息来实现的。

能否使用DQN来实现通过屏幕学习玩Flappy Bird是一个有意思的挑战。（话说本人和朋友在去年年底 虑了这个idea，但当时由于不知道如何截取游戏屏幕只能使用具体位置来学习，不过其实也成功了）

最近，github上有人放出使用DQN玩Flappy Bird的代码，
https://github.com/yenchenlin1994/DeepLearningFlappyBird【1】
该repo通过结合之前的repo成功实现了这个想法。这个repo对整个实现过程进行了较详细的分析，但是 于其DQN算法的代码基本采用别人的repo，代码较为混乱，不易理解。



为此，本人改写了一个版本https://github.com/songrotek/DRL-FlappyBird

对DQN代码进行了重新改写。本质上对其做了类的封装，从而使代码更具通用性。可以方便移植到其他应用。

当然，本文的目的是借Flappy Bird DQN这个代码来详细分析一下DQN算法极其使用。

DQN 伪代码

这个是NIPS13版本的伪代码:

```
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 ϵ select random action a_t
```

```
        otherwise select a_t=max_a  Q($s_t$,a; $θ_i$)
        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+γ*max_(a^' )  Q(s_(j+1),a'; θ_i) for non-terminal s_(j+1)

        Perform a gradient step on (y_j-Q(s_j,a_j; θ_i))^2 with respect to θ
    end for
end for
```

基本的分析详见Paper Reading 1 - Playing Atari with Deep Reinforcement Learning
基础知识详见Deep Reinforcement Learning 基础知识（DQN方面）

本文主要从代码实现的角度来分析如何编写Flappy Bird DQN的代码

编写FlappyBirdDQN.py

首先，FlappyBird的游戏已经编写好，是现成的。提供了很简单的接口：

```
nextObservation,reward,terminal = game.frame_step(action)
```

即输入动作，输出执行完动作的屏幕截图，得到的反馈reward，以及游戏是否结束。

那么，现在先把DQN想象为一个大脑，这里我们也用BrainDQN类来表示，这个类只需获取感知信息也是上面说的观察（截图），反馈以及是否结束，然后输出动作即可。

完美的代码封装应该是这样。具体DQN里面如何存储。如何训练是外部不关心的。
因此，我们的FlappyBirdDQN代码只有如下这么短：

```
# -------------------------
# Project: Deep Q-Learning on Flappy Bird
# Author: Flood Sung
# Date: 2016.3.21
# -------------------------

import cv2
```

```python
import sys
sys.path.append("game/")
import wrapped_flappy_bird as game
from BrainDQN import BrainDQN
import numpy as np

# preprocess raw image to 80*80 gray image
def preprocess(observation):
    observation = cv2.cvtColor(cv2.resize(observation, (80, 80)), cv2.COLOR_BGR2GRAY)
    ret, observation = cv2.threshold(observation,1,255,cv2.THRESH_BINARY)
    return np.reshape(observation,(80,80,1))

def playFlappyBird():
    # Step 1: init BrainDQN
    brain = BrainDQN()
    # Step 2: init Flappy Bird Game
    flappyBird = game.GameState()
    # Step 3: play game
    # Step 3.1: obtain init state
    action0 = np.array([1,0])  # do nothing
    observation0, reward0, terminal = flappyBird.frame_step(action0)
    observation0 = cv2.cvtColor(cv2.resize(observation0, (80, 80)), cv2.COLOR_BGR2GRAY
    ret, observation0 = cv2.threshold(observation0,1,255,cv2.THRESH_BINARY)
    brain.setInitState(observation0)

    # Step 3.2: run the game
    while 1!= 0:
        action = brain.getAction()
        nextObservation,reward,terminal = flappyBird.frame_step(action)
        nextObservation = preprocess(nextObservation)
        brain.setPerception(nextObservation,action,reward,terminal)

def main():
    playFlappyBird()

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

核心部分就在while循环里面，由于要讲图像转换为80x80的灰度图，因此，加了一个preprocess预处理函数。

这里，显然只有有游戏引擎，换一个游戏是一样的写法，非常方便。

接下来就是编写BrainDQN.py 我们的游戏大脑

编写BrainDQN

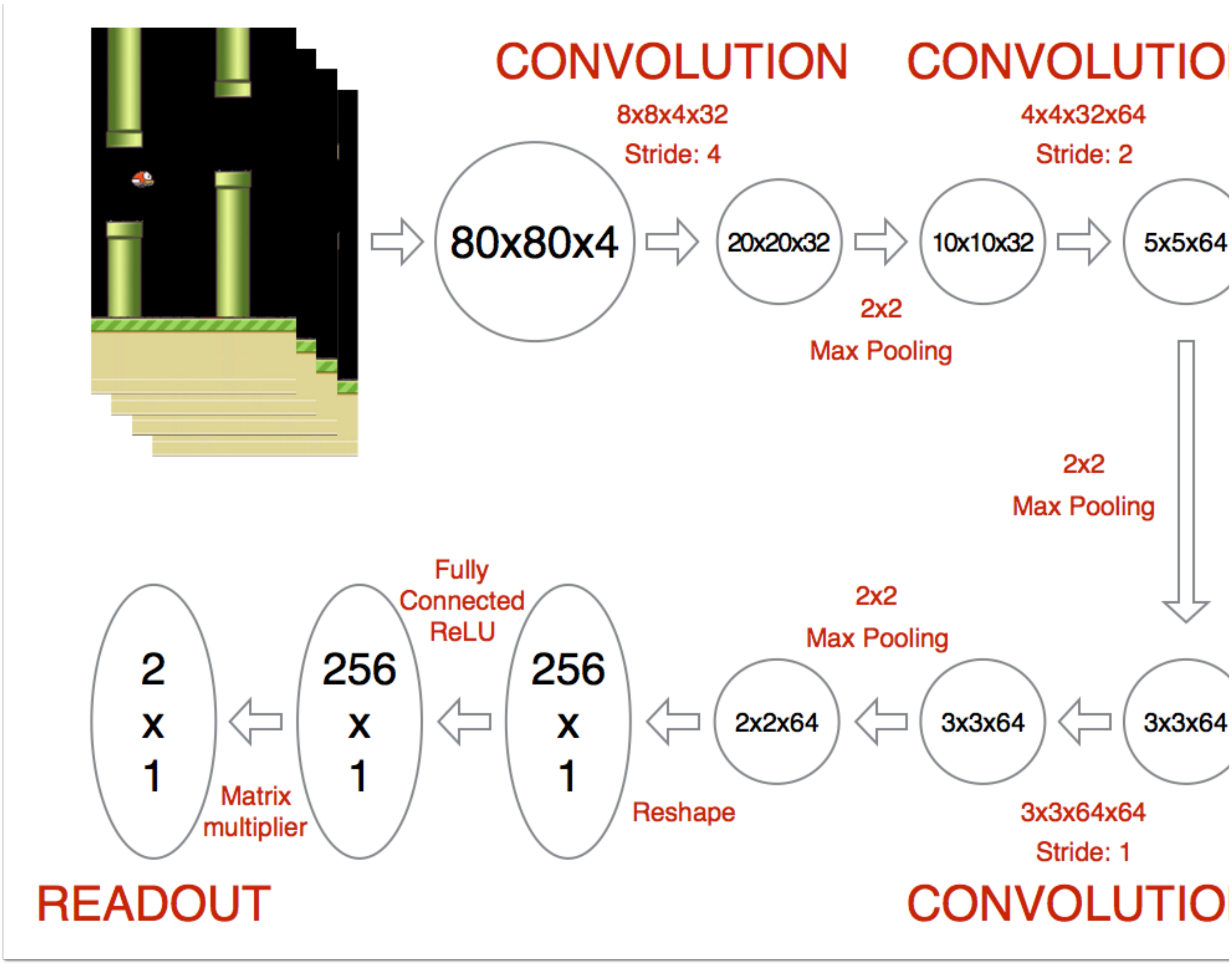
基本架构:

```
class BrainDQN:
    def __init__(self):
        # init replay memory
        self.replayMemory = deque()
        # init Q network
        self.createQNetwork()
    def createQNetwork(self):

    def trainQNetwork(self):

    def setPerception(self,nextObservation,action,reward,terminal):
    def getAction(self):
    def setInitState(self,observation):
```

基本的架构也就只需要上面这几个函数，其他的都是多余了，接下来就是编写每一部分的代码。

CNN代码

也就是createQNetwork部分，这里采用如下图的结构（转自【1】）：



这里就不讲解整个流程了。主要是针对具体的输入类型和输出设计卷积和全连接层。

代码如下：

```
    def createQNetwork(self):
        # network weights
        W_conv1 = self.weight_variable([8,8,4,32])
        b_conv1 = self.bias_variable([32])

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

        W_conv3 = self.weight_variable([3,3,64,64])
```

```
        b_conv3 = self.bias_variable([64])

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

        W_fc2 = self.weight_variable([512,self.ACTION])
        b_fc2 = self.bias_variable([self.ACTION])

        # input layer

        self.stateInput = tf.placeholder("float",[None,80,80,4])

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

        h_conv2 = tf.nn.relu(self.conv2d(h_pool1,W_conv2,2) + b_conv2)

        h_conv3 = tf.nn.relu(self.conv2d(h_conv2,W_conv3,1) + b_conv3)

        h_conv3_flat = tf.reshape(h_conv3,[-1,1600])
        h_fc1 = tf.nn.relu(tf.matmul(h_conv3_flat,W_fc1) + b_fc1)

        # Q Value layer
        self.QValue = tf.matmul(h_fc1,W_fc2) + b_fc2

        self.actionInput = tf.placeholder("float",[None,self.ACTION])
        self.yInput = tf.placeholder("float", [None])
        Q_action = tf.reduce_sum(tf.mul(self.QValue, self.actionInput), reduction_indi
        self.cost = tf.reduce_mean(tf.square(self.yInput - Q_action))
        self.trainStep = tf.train.AdamOptimizer(1e-6).minimize(self.cost)
```

记住输出是Q值，关键要计算出cost，里面关键是计算Q_action的值，即该state和action下的Q值。由于actionInput是one hot vector的形式，因此tf.mul(self.QValue, self.actionInput)正好就是该action下的Q值。

training 部分。

这部分是代码的关键部分，主要是要计算y值，也就是target Q值。

```
    def trainQNetwork(self):
        # Step 1: obtain random minibatch from replay memory
```

```
        # Step 1: obtain random minibatch from replay memory
        minibatch = random.sample(self.replayMemory,self.BATCH_SIZE)
        state_batch = [data[0] for data in minibatch]
        action_batch = [data[1] for data in minibatch]
        reward_batch = [data[2] for data in minibatch]
        nextState_batch = [data[3] for data in minibatch]

        # Step 2: calculate y
        y_batch = []
        QValue_batch = self.QValue.eval(feed_dict={self.stateInput:nextState_batch})
        for i in range(0,self.BATCH_SIZE):
            terminal = minibatch[i][4]
            if terminal:
                y_batch.append(reward_batch[i])
            else:
                y_batch.append(reward_batch[i] + GAMMA * np.max(QValue_batch[i]))

        self.trainStep.run(feed_dict={
            self.yInput : y_batch,
            self.actionInput : action_batch,
            self.stateInput : state_batch
            })
```

其他部分

其他部分就比较容易了，这里直接贴出完整的代码：

```
# -----------------------------
# File: Deep Q-Learning Algorithm
# Author: Flood Sung
# Date: 2016.3.21
# -----------------------------

import tensorflow as tf
import numpy as np
import random
from collections import deque

class BrainDQN:

    # Hyper Parameters:
```

```
    ACTION = 2
    FRAME_PER_ACTION = 1
    GAMMA = 0.99 # decay rate of past observations
    OBSERVE = 100000. # timesteps to observe before training
    EXPLORE = 150000. # frames over which to anneal epsilon
    FINAL_EPSILON = 0.0 # final value of epsilon
    INITIAL_EPSILON = 0.0 # starting value of epsilon
    REPLAY_MEMORY = 50000 # number of previous transitions to remember
    BATCH_SIZE = 32 # size of minibatch

    def __init__(self):
        # init replay memory
        self.replayMemory = deque()
        # init Q network
        self.createQNetwork()
        # init some parameters
        self.timeStep = 0
        self.epsilon = self.INITIAL_EPSILON

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

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

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

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

        W_fc2 = self.weight_variable([512,self.ACTION])
        b_fc2 = self.bias_variable([self.ACTION])

        # input layer

        self.stateInput = tf.placeholder("float",[None,80,80,4])

        # hidden layers
        h_conv1 = tf.nn.relu(self.conv2d(self.stateInput,W_conv1,4) + b_conv1)
```

```
        h_pool1 = self.max_pool_2x2(h_conv1)

        h_conv2 = tf.nn.relu(self.conv2d(h_pool1,W_conv2,2) + b_conv2)

        h_conv3 = tf.nn.relu(self.conv2d(h_conv2,W_conv3,1) + b_conv3)

        h_conv3_flat = tf.reshape(h_conv3,[-1,1600])
        h_fc1 = tf.nn.relu(tf.matmul(h_conv3_flat,W_fc1) + b_fc1)

        # Q Value layer
        self.QValue = tf.matmul(h_fc1,W_fc2) + b_fc2

        self.actionInput = tf.placeholder("float",[None,self.ACTION])
        self.yInput = tf.placeholder("float", [None])
        Q_action = tf.reduce_sum(tf.mul(self.QValue, self.actionInput), reduction_indi
        self.cost = tf.reduce_mean(tf.square(self.yInput - Q_action))
        self.trainStep = tf.train.AdamOptimizer(1e-6).minimize(self.cost)

        # saving and loading networks
        saver = tf.train.Saver()
        self.session = tf.InteractiveSession()
        self.session.run(tf.initialize_all_variables())
        checkpoint = tf.train.get_checkpoint_state("saved_networks")
        if checkpoint and checkpoint.model_checkpoint_path:
                saver.restore(self.session, checkpoint.model_checkpoint_path)
                print "Successfully loaded:", checkpoint.model_checkpoint_path
        else:
                print "Could not find old network weights"

    def trainQNetwork(self):
        # Step 1: obtain random minibatch from replay memory
        minibatch = random.sample(self.replayMemory,self.BATCH_SIZE)
        state_batch = [data[0] for data in minibatch]
        action_batch = [data[1] for data in minibatch]
        reward_batch = [data[2] for data in minibatch]
        nextState_batch = [data[3] for data in minibatch]

        # Step 2: calculate y
        y_batch = []
        QValue_batch = self.QValue.eval(feed_dict={self.stateInput:nextState_batch})
        for i in range(0,self.BATCH_SIZE):
            terminal = minibatch[i][4]
```

```
            if terminal:
                y_batch.append(reward_batch[i])
            else:
                y_batch.append(reward_batch[i] + GAMMA * np.max(QValue_batch[i]))

        self.trainStep.run(feed_dict={
            self.yInput : y_batch,
            self.actionInput : action_batch,
            self.stateInput : state_batch
            })

        # save network every 100000 iteration
        if self.timeStep % 10000 == 0:
            saver.save(self.session, 'saved_networks/' + 'network' + '-dqn', global_st


    def setPerception(self,nextObservation,action,reward,terminal):
        newState = np.append(nextObservation,self.currentState[:,:,1:],axis = 2)
        self.replayMemory.append((self.currentState,action,reward,newState,terminal))
        if len(self.replayMemory) > self.REPLAY_MEMORY:
            self.replayMemory.popleft()
        if self.timeStep > self.OBSERVE:
            # Train the network
            self.trainQNetwork()

        self.currentState = newState
        self.timeStep += 1

    def getAction(self):
        QValue = self.QValue.eval(feed_dict= {self.stateInput:[self.currentState]})[0]
        action = np.zeros(self.ACTION)
        action_index = 0
        if self.timeStep % self.FRAME_PER_ACTION == 0:
            if random.random() <= self.epsilon:
                action_index = random.randrange(self.ACTION)
                action[action_index] = 1
            else:
                action_index = np.argmax(QValue)
                action[action_index] = 1
        else:
            action[0] = 1 # do nothing

```

```python
        # change episilon
        if self.epsilon > self.FINAL_EPSILON and self.timeStep > self.OBSERVE:
            self.epsilon -= (self.INITIAL_EPSILON - self.FINAL_EPSILON)/self.EXPLORE

        return action

    def setInitState(self,observation):
        self.currentState = np.stack((observation, observation, observation, observati

    def weight_variable(self,shape):
        initial = tf.truncated_normal(shape, stddev = 0.01)
        return tf.Variable(initial)

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

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

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

```

一共也只有160代码。
如果这个任务不使用深度学习，而是人工的从图像中找到小鸟，然后计算小鸟的轨迹，然后计算出应该么按键，那么代码没有好几千行是不可能的。深度学习大大减少了代码工作。

小结

本文从代码角度对于DQN做了一定的分析，对于DQN的应用，大家可以在此基础上做各种尝试。