强化学习：第七次作业

阅读《RL》第九章

本章的主要思想在于利用以权重作为参数的函数来估计值函数，以取代之前表格形式的值函数表示方法。

> The novelty in this chapter is that the approximate value function is represented not as a table but as a parameterized functional form with weight vector.

9.1 Value-function Approximation

不同方法的值函数估计方法存在相同的范式：$s \mapsto u$，即把估计值`estimated value`应用到更新目标`backed-up value`上。对于表格式的值函数表示，某个状态的值函数更新不会影响到其他状态的值函数。而对于函数方式的值函数估计，状态值的更新通过调整函数中权重参数来实现，因此一旦某个权重改变将会改变所有状态的值函数。

9.2 The Prediction Objective

表格式的值函数估计：每个状态的值函数是解耦的，且可以认为学习到的值函数等价于真实的值函数。

函数式的值函数估计：定义了一个通过均方根来进行误差估计的目标函数$\overline{VE}$。

9.3 Stochastic-gradient and Semi-gradient Methods

> SGD methods are among the most widely used of all function approximation methods and are particularly well suited to online reinforcement learning.

SGD：对目标函数$\overline{VE}$求关于权重w的微分，就能得知权重的修改方法（公式9.7）。

Semi-gradient：对于公式$U_t = R_{t+1} + \gamma v(S_{t+1}, w)$，半梯度的方法只关心$v(S_{t+1}, w)$而不关心权重的调整对目标函数$U_t$的影响。

9.4 Linear Methods

假定$v(S,w)$函数是线性的，可以寻找一个特征函数$x(s)$来对$v(S,w)$函数进行逼近，这样问题就转换成求这个特征函数的各项分量的系数。

9.5 Feature Construction for Linear Methods

- 多项式：有点类似与正定二次型？
- 傅里叶：感觉是由于任何信号都可以用傅里叶展开去逼近，所以就可以将其应用在这种估计值函数的场景中。
- Coarse Coding/Tile Coding/Radial Basis Functions

9.6 Selecting Step-Size Parameters Manually

随机梯度下降方法需要设置学习率（step-size parameter α），确定学习率的方法是：$\alpha = (\tau E[x^T x])^{-1}$。

9.7 Nonlinear Function Approximation: ANN

正向传播：计算损失函数

反向传播： 计算神经网络各层的梯度，并通过多种优化算法（比如SGD、ADAM）来修改梯度

9.8 LSTD

突破迭代计算权重w和偏差b的局限，但是需要大量的空间来存储权重矩阵（复杂度是指数级的）。

9.9 & 9.10 Memory-based / Kernel-based Function Approximation

- Memory-based：在需要查询某个状态值估计的时候，才利用存储在内存中的数据进行权重更新，是一种lazy learning。
- Kernel-based：Kernel regression is the memory-based method that computes a kernel weighted average of the targets of all examples stored in memory, assigning the result to the query state.

阅读Github的Flappy Bird源码，体会DQN算法

DQN算法

前身是Q-learning，下图是DQN算法的伪代码：

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

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

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

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

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    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 D

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

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Flappy Bird源码

除去游戏的本体部分，关于DQN的代码均在`deep_q_network.py`文件中，因此接下来我仅分析该文件的代码。

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

表明该程序使用的Python版本是Python2。

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

程序使用`tensorflow`、`opencv`、`numpy`等第三方工具包。

```
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
```

设置超参，比如动作空间、折扣因子、贪心算法的ϵ、

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

以均值为0，标准差为0.01，权重的维度三个参数来初始化权重。

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

以常量值0.01，偏差的维度两个参数来初始化偏差。

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

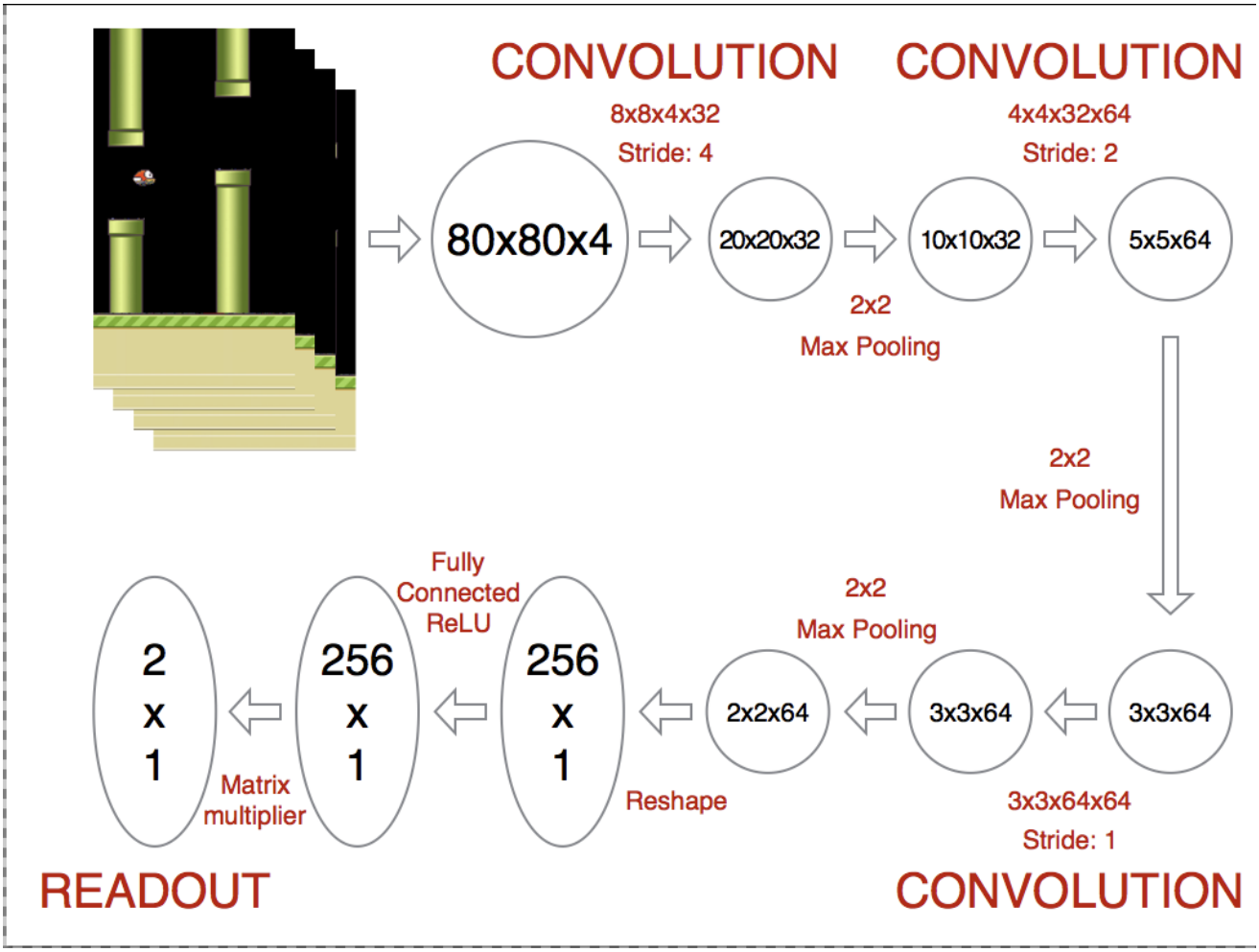
定义卷积层，x是输入图像，其维度是[batch, in_height, in_width, in_channels]；W是卷积核，其维度是[filter_height, filter_width, in_channels, out_channels]；strides是卷积在每个维度的步长；padding决定了卷积方式，‘SAME’表示卷积核可以停留在图像的边缘。

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

定义池化层，程序使用的是最大值池化。x是池化的输入，其维度是[batch, height, width, channels]；ksize表示池化窗口的大小，其维度是[batch, height, width, channels]；stride和padding同卷积层的定义。

```
def createNetwork():
    ...
```

构建卷积网络，经过分析，网络的结构如下：




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

定义网络的输入，维度是[batch, 80：高, 80：宽, 4：通道（4张图片）]。

```
def createNetwork():
    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])
    ...
```

定义3个卷积层的权重和偏差以及2个全连接层的权重和偏差。

```
def createNetwork():
    # 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)
    ...
```

卷积层1+池化层1+卷积层2+卷积层3

```
def createNetwork():
    #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)
    ...
```

卷积结果传递给全连接层之前，需要将卷积结果展平。在全连接层1之后使用ReLU非线性激活函数来防止过拟合。

```
def createNetwork():
    # readout layer
    readout = tf.matmul(h_fc1, W_fc2) + b_fc2

    return s, readout, h_fc1
```

通过矩阵乘法构建全连接层2。`createNetwork`函数最终返回图像输入s，全连接层2的输出，全连接层1的输出。

```
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)
    ...
```

定义损失函数。网络给出的动作`readout_action`由网络的输出`readout`与动作空间`a`进行对应元素相乘并按行求和产生。损失函数通过计算真实值`y`与动作`readout_action`之间的均方差定义。优化器选用的是Adam优化器，学习率为1e-6。

```
def trainNetwork(s, readout, h_fc1, sess):
    # 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')
    ...
```

与游戏交互，获取游戏图像。用队列存储观察结果。动作选择和权重文件存放在`logs_`文件夹中。

```
def trainNetwork(s, readout, h_fc1, sess):
    # 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)
    ...
```

初始动作选择，利用opencv修改游戏帧的大小并进行二值化`cvtColor(),threshold()`，4张图片通过`stack`函数实现堆叠。

```
def trainNetwork(s, readout, h_fc1, sess):
    # 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")
    ...
```

保存模型，模型的checkpoint存放在`saved_networks`文件夹中。

```
def trainNetwork(s, readout, h_fc1, sess):
    # start training
    epsilon = INITIAL_EPSILON
    t = 0
    while "flappy bird" != "angry bird":
        ...
```

开始训练，初始化贪心策略的ϵ，训练的终止条件是`"flappy bird" != "angry bird"`，因此就是True。

```
def trainNetwork(s, readout, h_fc1, sess):
    # 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
    ...
```

执行计算图（计算图的输入s的值是s_t，即通过np.stack堆叠的4张图像），将全连接层2的输出`readout`的第0维数据赋值给`readout_t`。

对于起始时刻，动作选择总是选择第一个动作；对于其他时刻，按照$\epsilon-$贪心策略进行动作选择。

```
def trainNetwork(s, readout, h_fc1, sess):
    # scale down epsilon
    if epsilon > FINAL_EPSILON and t > OBSERVE:
        epsilon -= (INITIAL_EPSILON - FINAL_EPSILON) / EXPLORE
        ...
```

如果当前的时刻已经超过了可以接受的最大观察次数`OBSERVE`，那么就减小贪心策略的ϵ。

```
def trainNetwork(s, readout, h_fc1, sess):
    # 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)
    ...
```

根据动作选择来获取奖赏以及接下来的状态。

```
def trainNetwork(s, readout, h_fc1, sess):
    # store the transition in D
    D.append((s_t, a_t, r_t, s_t1, terminal))
    if len(D) > REPLAY_MEMORY:
        D.popleft()
        ...
```

在队列D中存储当前的状态、动作、奖赏、下一时刻的状态，通过先进先出来控制队列的长度。

```
def trainNetwork(s, readout, h_fc1, sess):
    # 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}
            )
```

生成一条Eposide后才开始训练。从一条Eposide中，随机采样`minibatch`的数据。

`s_j_batch,a_batch,r_batch,s_j1_batch`分别是minibatch中的当前状态、动作、奖赏和下一时刻的状态。

`y_batch`：存放真实的值函数，如果遇到终止状态，则只保存奖赏。

`readout_j1_batch`：通过将下一个时刻的状态作为输入，计算整个计算图，得到动作选择

根据`y_batch,a_batch,s_j_batch`来计算损失函数并进行梯度下降。

```
def trainNetwork(s, readout, h_fc1, sess):
# 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
...
```

何时保存模型以及控制程序当前进行训练还是进行生成数据。

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

def main():
    playGame()

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

程序入口。

总结

阅读源码后，我对使用tensorflow进行DQN学习有了大致的了解。相较于tensorflow，我更习惯用PyTorch进行深度学习，我不太习惯tensorflow对张量的声明方式（placeholder）以及训练的模式（session），在以后的作业中我都采用PyTorch进行网络的训练和学习。