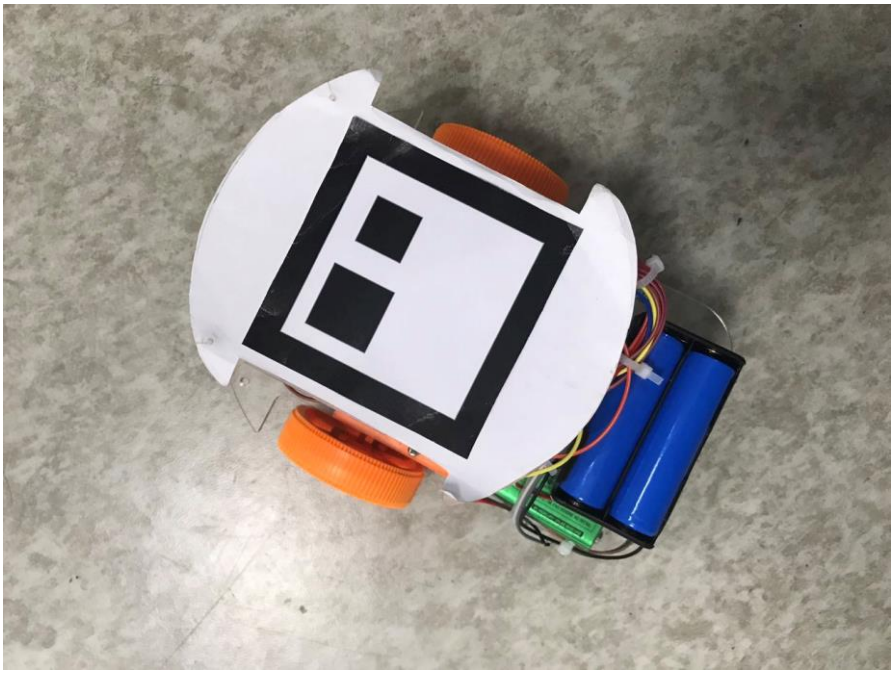
Robot-Car-DRL
Construction Documentation

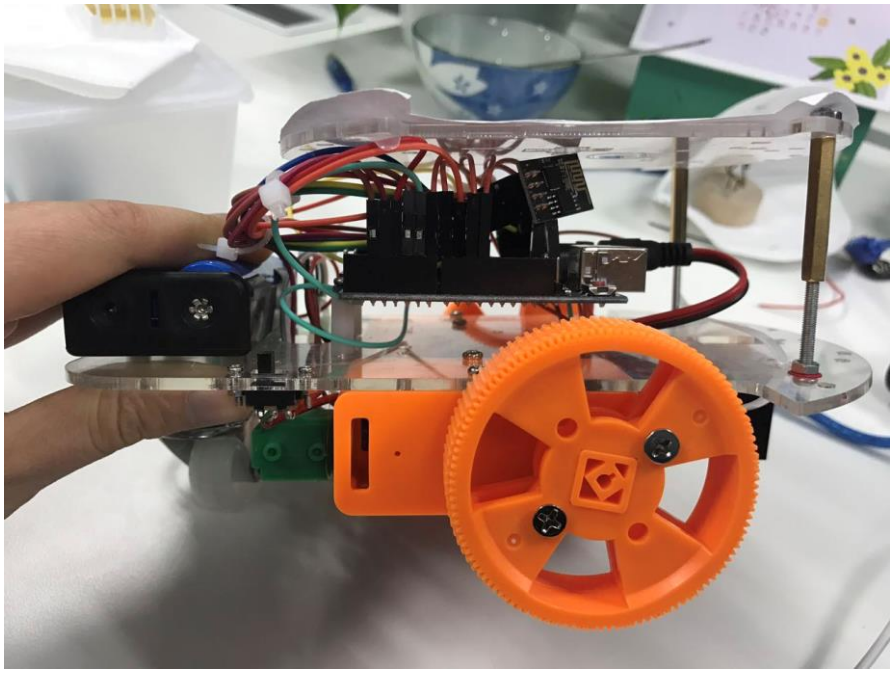
Xia, Dongqing
2017.01.04 ~ Now
Horizon Robotics @ Beijing

Context

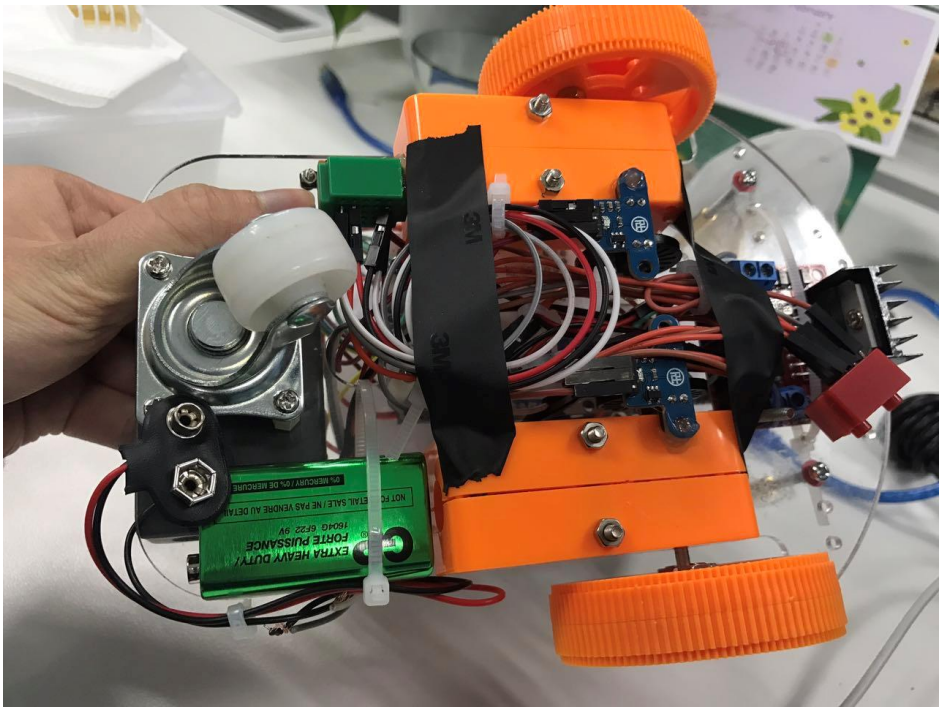
1. Hardware Connection



[Fig 1: 小车 Version 0.0.1 鸟瞰图]



[Fig 2: 小车 Version 0.0.1 侧视图]



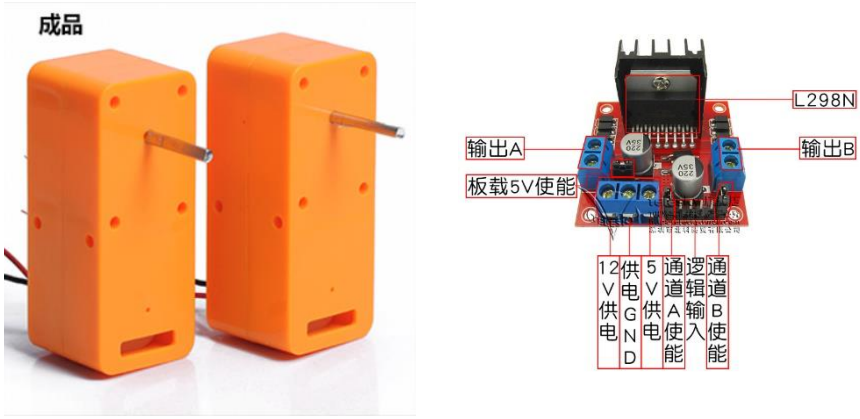
[Fig 3: 小车 Version 0.0.1 底部图]

1.1 Motor & Driving Board

1.1.1 参数明细：

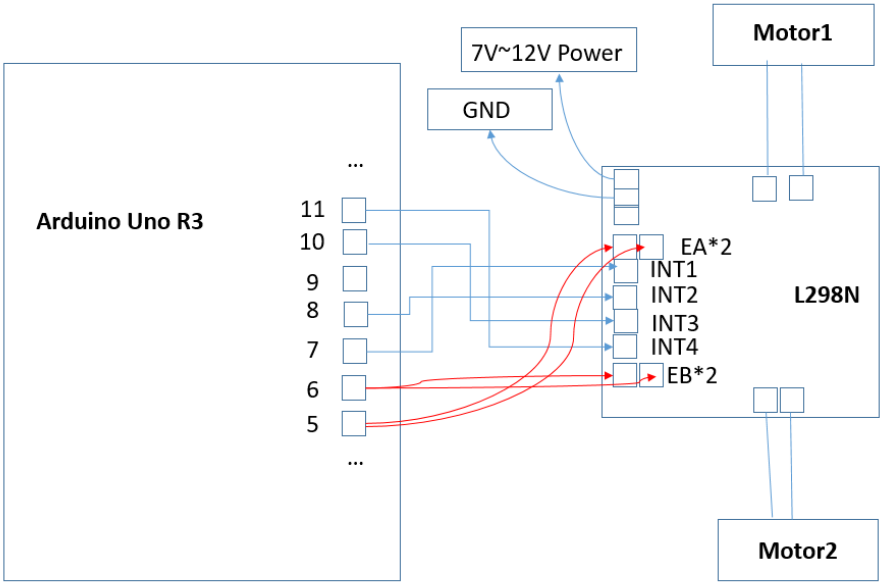
1. N20 电机+橙色减速箱
https://item.taobao.com/item.htm?spm=a230r.1.14.33.B4IU2Y&id=520198150050&ns=1&abbucket=7#detail

2. L298N 电机驱动板模块
https://detail.tmall.com/item.htm?spm=a230r.1.14.9.3WbsKy&id=41248562401&cm_id=140105335569ed55e27b&abbucket=7



[Fig 4: 左：减速箱；右：驱动板示意图]

1.1.2 实际连线说明：




```
// Pin Constants
const int EApin = 5; //motor driver enable: analong Write to control speed
const int EBpin = 6; //motor driver enable: analong Write to control speed
const int rightmotorpin1 = 7; // 直流电机信号输出
const int rightmotorpin2 = 8;
const byte leftmotorpin1 = 10;
const byte leftmotorpin2 = 11;
```

[Fig 5: 连线示意图 以及 Arduino 代码]

1.1.3 Wiki

(1) EA 和 EB 一共 4 个针脚都要连上。如果 motor 不能转动，
第一步，确认一下相应的 output pin 有没有电压，然后看一下 VCC 的电压(应为 7~12V)；
第二步，确认 EA，EB 的电压够高(>= 2.8)

(2) power 一共有 3 个接口，除了标出的 7V~12V(VCC)输入和 GND，还有一个 5V 输出。
那边可能会有跳帽决定板内电源还是板外电源，请根据说明来决定要不要去掉。

(3) EA pin 和 EB pin 需要用 pwm 调速，所以需要有 analogWrite 功能的 pin。
如果以后不用 Arduino Uno R3, 请确认一下连接的 pin 是不是输出占空比不同的方波。

(4) INT1, 2, 3, 4 只需要能写高写低就可以了。
所以在 Arduino Uno R3 的代码里面，使用了 digitalWrite(pin, HIGH/LOW)就可以了。

(5) L298N 的 OUT 分别于 motor1，motor2 的红黑线连接好。具体连线没什么具体要求。

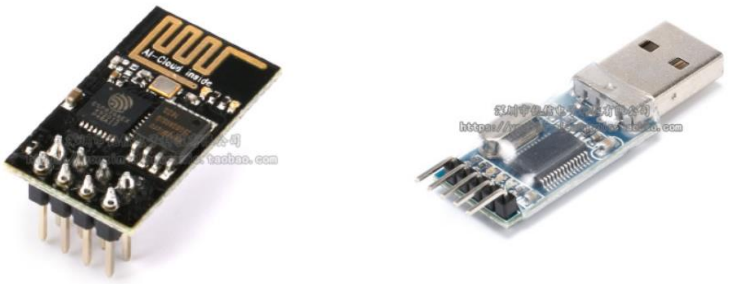
如果发现 motor 与想象的方向转反了，可以检查一下 INT1，2，3，4 的信号有没有写对;或者，直接把 motor 红黑线与 output pin 反接。

1.2 Esp8266 Wifi Module

1.2.1 参数明细：

1. ESP-01 ESP8266 串口 WIFI 模块
https://item.taobao.com/item.htm?spm=a1z09.2.0.0.CNXZhl&id=522564863938&_u=pqn4uc71def

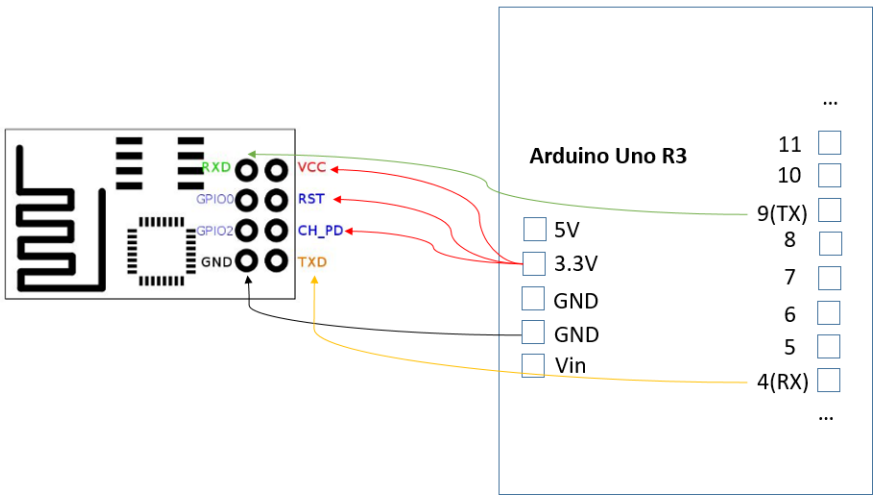
2. USB 转 TTL PL2303HX STC 单片机下载线刷机线(必需)
https://item.taobao.com/item.htm?spm=a1z09.2.0.0.CNXZhl&id=522575308496&_u=pqn4uc7a33b



[Fig 6: 左：ESP8266; 右：USB 转 TTL PL2303]

1.2.2 实际连线说明：

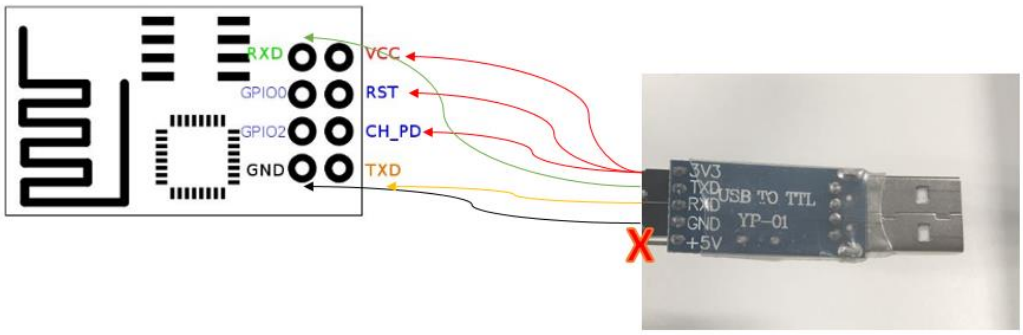
Esp8266 + Arduino Uno R3 (Working Mode)：




```
#include <SoftwareSerial.h>
SoftwareSerial BT(4, 9); // 接收，传送，程序中2为RX需要接esp8266的TXD,9为TX，需要接esp8266的RX。
```

[Fig 7: 上：连线示意图；下：arduino 代码]

Esp8266 + USB 转 TTL (Debug Mode)：



[Fig 8: 连线示意图]

1.2.3 Wiki:

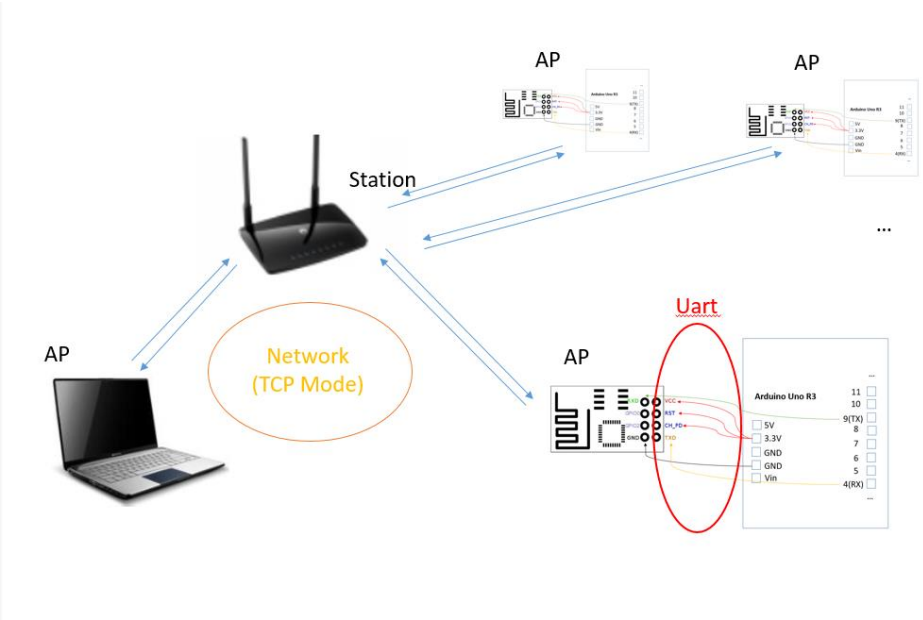
(1) 因为 esp8266 模块的默认波特率为 115200bps，所以刚开始的时候，应该使用 USB 转 TTL 模块在 PC 端手动将 esp8266 的波特率调至 38400bps (Arduino Uno R3 只能提供最大的波特率为 57600bps)

(2) 每次 arduino setup 时都向 esp8266 重写一次 AT+CWJAP 的命令，能够保证你加进了你想要的内网里。然后 PC 有线连接路由，可以在 192.168.1.1 查看加进的设备的 IP 地址。确认 arduino 烧写的代码里面连接的 server ip 与网页中显示的 PC ip 一致。

(3) 在 window 下调试的时候，“网络助手”软件可以设置端口号为 8080；但是在 linux 下 8080 是已占的默认端口号，所以应该设置一个区别于 8080 的监听端口号。PC server thread 的 port 这里设置的是 5678。请确认 arduino 连接 port 与 server 设置的一样。（不要盲拉网上代码）

(4) Esp8266 模块的 AT 命令，其反馈是有延时的。建议 initialization 的时候每个指令留 2~3 秒的 delay。

(5) 通信模型示意图如下：



[Fig 9: 通信模型。如果 esp8266 收发的频率太高，arduino uart buffer 就会有乱码的问题。]

1.3 Encoding Disk

1.3.1 参数明细：

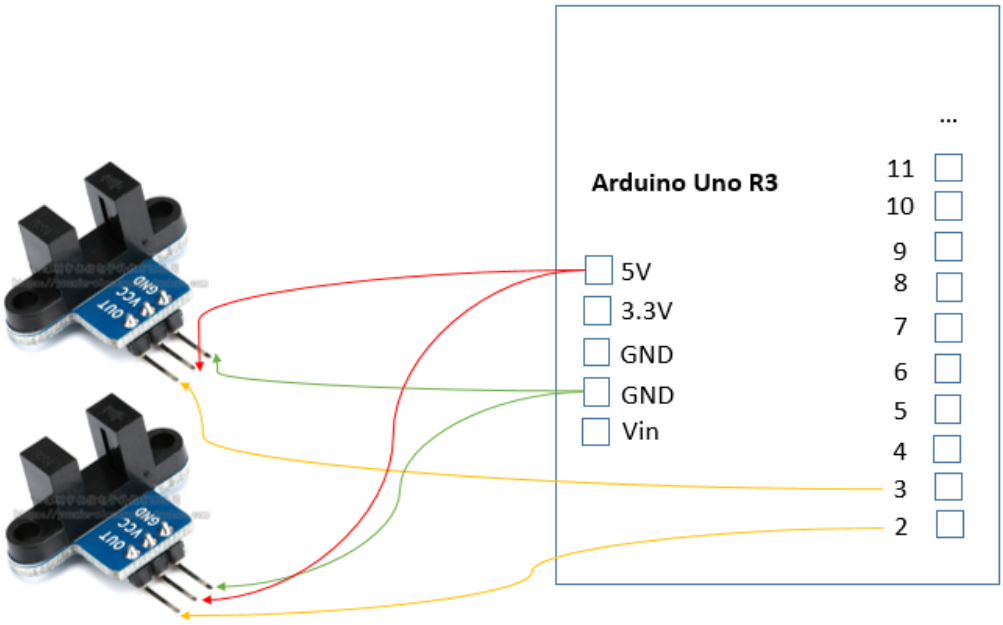
1. 测速传感器

https://item.taobao.com/item.htm?spm=a1z09.2.0.0.LyLRF1&id=528277382354&_u=mqn4uc74107

2. 计数码盘

https://item.taobao.com/item.htm?spm=a1z09.2.0.0.q5kEug&id=526497045791&_u=mqn4uc748b4

1.3.2 实际连线说明：



[Fig 10: 光电测速计和 arduino 的连线。此处省略码盘和轮子接合的连线图。
上电后转动码盘看看指示灯会不会亮来检测传感器是否正常工作。导通时指示灯会亮。]

1.3.3 Wiki:

(1) 测距离：测速传感器输出为脉冲信号，一个脉冲中断一次；红外射线导通的时候是低电平，所以我们设置中断为低 电平触法模式。一般码盘上有整数格子，无论是多少格其实原理一样，例如此处为 20 格码盘，也就是有 20 个空格子，电机转一圈后便是射线导通 20 次，外部低电平触法 20 次； 于是我们计算中断次数，得到的总次数除于 20 也就是电机转动次数了，然后按照轮子的周长，计算轮子一圈是多长，就可以推算出小车已经跑多远了。

(2) 测速度： 按照测距离的思路，我们用一个 MCU 定时器计算，1 秒内接收多少个外部中断，例如一秒内接收了 20 个外部中断，我们就可以判断小车速度为 1 秒小车轮子转一圈，然后再计算出小车轮子的周长，就可得知小车 1 秒行驶的速度。

(3) 中断命令语法介绍

attachInterrupt(interrupt, function, mode)
-- interrupt：中断引脚数
-- function：中断发生时调用的函数，此函数必须不带参数和不返回任何值。该函数有时被称为中断服务程序。
-- mode：定义何时发生中断以下四个 contstants 预定有效值：

LOW 当引脚为低电平时，触发中断
CHANGE 当引脚电平发生改变时，触发中断
RISING 当引脚由低电平变为高电平时，触发中断
FALLING 当引脚由高电平变为低电平时，触发中断.

当发生外部中断时，调用一个指定函数。当中断发生时，该函数会取代正在执行的程序。大多数的 Arduino 板有两个外部中断：0（数字引脚 2）和 1（数字引脚 3）。

(4) 外部设置一个类似定时计的功能计算 speed:
调用 millis()函数返回以毫秒表示的时间,而 micros()函数返回以微秒表示的时间。此处使用 millis(). 每间隔一段时间去取 rising edge 上升次数，计算 speed. 注意如果间隔时间太短，就会出现计算 speed = (此时刻 rising edge - 上一时刻 rising edge) / 间隔时间 = 0 的情况。以及，speed 要定义为 float，而不是 int…

根据网上帖子的经验是，不要使用内部的定时器，否则 timer interrupt 部分的读取会错乱的。

1.4 Power

1.4.1 参数明细：

(1) Motor power：GP 超霸 6F22 9V 环保碳性电池 层叠电池 方块电池 用于测试仪等
https://item.taobao.com/item.htm?spm=a1z09.2.0.0.q5kEug&id=522582253855&_u=mqn4uc72fb6
comment：掉电极快…… 不推荐继续使用。

(2) Board Power：模型电子配件大容量平头 18650 电池 3.7V 锂电池 2500mAh
https://item.taobao.com/item.htm?spm=a1z09.2.0.0.q5kEug&id=534589414746&_u=mqn4uc72e13
comment：两节供电，因为 board 上面有个稳压器，所以理论能够给板提供正常电压。但曾经出现过稳压模块崩掉的 case。所以还是，每用一段时间之后就看看各供电/输出电压接口之类的对不对。

1.5 New Version (under developing)

1.5.1 Changes：

(1) 更换 power supply：使用一组共 6 节 5 号电池给 board 和 motor 上电。
(2) 整理连线。

2. Software Support

2.1 Image processing

(1) Helper_Feb16.py

```python
# find the largest contour, it should be the board
def board_pos_deter(contours):
    maxarea = 0
    for contour in contours:
      area = cv2.contourArea(contour)
      if area > maxarea:
    maxcontour = contour
    maxarea = area

    min_rect = cv2.minAreaRect(contour)
    (x,y),(w,h), angle = min_rect

    return (x, y), (w, h), angle, contour
```

找到最大的轮廓就是背景

```python
# find the contours of cars
def car_pos_deter(contours):...

def axis_convert2_normal(point_xy_in_video):
    return (point_xy_in_video[0], -point_xy_in_video[1])

def vector_direction(start_pt, end_pt):...


def car_pos_is_table(car_center, car_center_pre, mark1_center, mark1_center_pre,

def command_calculator(target_direction, current_direction, angle_param):...

def find_corner(img, suspect_point, range_num, direction_case, corner_value):...

def two_point_distance(start_pt, end_pt):
    return math.sqrt((start_pt[0]-end_pt[0])**2 + (start_pt[1]-end_pt[1])**2)
```

其余函数:

a. Car_pos_deter 是找到小车上的方框，然后再去检测方框内的两个大小黑框，然后算的小车的中心和方向。主要用到 area 大小以及 $\frac{\text{area}}{\text{bounding\_rect area}}$ 的比例对 contour 进行筛选。
b. Vector_direction 就是用来算指定起始点到指定终点的矢量的方向。
c. Car_pos_is_stable 是用来算上一帧和下一帧的小车中心和方向差是否在设定的阈值之内，如果是，则使用新测量值；如果否，则使用旧的测量值。
d. Command_calculator()是用以计算控制命令的，可以设置一个正负范围作为前进方向，比如目标方向在前进方向的[-15°，15°]间时，可以前进。
e. Find_corner()，是对上个找 board 的函数得到的 x,y,w,h,找边上的 4 个角的坐标。
f. Two_point_distance()计算指定亮点的距离。

(2) Image_process_helper.py

```python
def cut_restore(frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    ret, bin2 = cv2.threshold(gray, 128, 255, 8)
    # cv2.imshow("bin2", bin2)

    # bin2_tmp = cv2.copyMakeBorder(bin2,0,0,0,0,cv2.BORDER_REPLICATE)
    contours, hierarchy = cv2.findContours(bin2,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
    (x,y), (w,h), board_angle, board_cnt = board_pos_deter(contours) # return position and angle of

    left_top = (int(x-w/2), int(y-h/2))
    right_top = (int(x+w/2), int(y-h/2))
    left_bottom = (int(x-w/2), int(y+h/2))
    right_bottom = (int(x+w/2), int(y+h/2))

    pts1 = np.float32([[left_top,right_top,left_bottom,right_bottom]])
    pts2 = np.float32([[[0,0],[int(round(w)),0],[0,int(round(h))],[int(round(w)),int(round(h))]]])

    M = cv2.getPerspectiveTransform(pts1,pts2)

    # print(int(round(h)),int(round(w)))
    gray_dst = cv2.warpPerspective(gray,M,(int(round(w)),int(round(h))))
    #cv2.imshow("gray_dst", gray_dst)

    ret, bin2_dst = cv2.threshold(gray_dst, 100, 255, 0)
    # find the cars' contour on the board
    contours_dst, hierarchy=cv2.findContours(bin2_dst,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)

    return gray_dst, contours_dst
```

将找到的板子的 x,y,w,h 转换为四个边角点之后，要对板子进行旋转切割，最终使得一个 window 就是我们的板子。

2.2 Server Setup

```python
#!/usr/bin/python
#-*-coding:utf-8-*-
import socket   #socket模块


HOST = ''               # Symbolic name meaning all available interfaces
PORT = 5678             # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
    print data
conn.close()
```

简单版本的 test. (Note: Linux 下监听的 port 不能设置 8080.)
参考 INTERN_codes/Robot-carDRL/wifi_server_2.py

复杂版本的 server setup 就直接参考 INTERN_codes/Robot-carDRL/async_threads_ver/try_multhreads_scene.py 里面 socketThread 的 init()

```
int wifi_init(){
  for (int i = 0; i < 8; i++){
      BT.write(wifi_command_set[i]);
      delay(3000);
      while(BT.available()){
      tmp_ch = BT.read();
      Serial.write(tmp_ch);
      }
   }

   BT.write(wifi_command_set[8]);
   for(int i=0; i<4; i++){
    wifi_answer[i] = BT.read();
   }
   if(wifi_answer[0] == 'C' && wifi_answer[1] == 'a' && wifi_answer[2] == 'r'&& wifi_answer[3] == '3'){
    init_stage = 1; // init success
   }
   return init_stage;
}
```

而在 arduino 端需要向 esp8266 发送一系列指令集，建立连接。

```
char* wifi_command_set[] = {"AT\r\n",
                            "AT+CWMODE=3\r\n",
                            "AT+CWJAP=\"Hobot_AutonomouDriving\",\"czs918170\"\r\n",
                            "AT+CIFSR\r\n",
                            "AT+CIPMUX=1\r\n",
                            "AT+CIPSERVER=0\r\n",
                            "AT+CIPSTART=1,\"TCP\",\"192.168.1.100\",5678\r\n",
                            "AT+CIPSEND=1,4\r\n",
                            "c3c\n"};
// if want to change baudrate, use command: "AT+UART=38400,8,1,0,0\r\n"
```

附 esp8266 指令集，为了能够完备地建立连接，建议每次都把整个指令集都发一遍。

2.3 Arduino Car Agent

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(38400); // 开始串行监测
  Serial.println("BT is ready!");
  // esp8266默认, 115200, but arduino R3 only provide 57600bps at muximum
  // details: https://www.arduino.cc/en/Reference/SoftwareSerial
  BT.begin(38400);

  aaUp = millis( );
  incomeUp = 0;

  //信号输出接口
  for (int pinindex = 5; pinindex < 12; pinindex++) {
    pinMode(pinindex, OUTPUT); // set pins 5 to 11 as outputs
  }

  pinMode(3, INPUT_PULLUP);
  pinMode(2, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(2), left_motor, RISING);
  attachInterrupt(digitalPinToInterrupt(3), right_motor,RISING);

  int wifi_init_success = wifi_init();
  Serial.println("Initialization success.");
}
```

设置波特率，设置 pin 口的输入输出模式，设置中断，然后尝试连接 server。

Wifi_init()的函数在这里省略细节，就是把该走的命令全部显式地向 esp8266 发一遍(注

意信息发送间隔要稍微设置得长一些，这里是 3 秒一个命令)，然后如果连上了 pc 设置的 server，就应该会收到自己在 server 那边设置的一个信息，然后就应该得到 initialization 成功的信息。

Loop:

(1) 先从 buffer 里面读最多 32 字节。企图能包含一整个命令。
(2) 然后从这个 32 字节的 string 中恢复出 8 字节的控制命令信息。(有时候中间会丢 1 个字节，可以尝试地恢复)
(3) 从完整的命令中得到控制的速度和方向信息。
(4) 执行命令，如果命令无效则 try_ask_server(),意即想 pc 发送一些执行失败的信号。
(5) Clear_buffer(),把 esp8266 module 返回的形如 send success 的信息读掉，如果在 clear 过程中读到了疑似 command 的信息，立刻停止 clear 而进行下一个 loop
(6) 用 millis()监测，每一段时间计算 speed

2.4 Async-threads Communication

(1) Simple demo: INTERN_codes/Robot_car_DRL/async_threads_ver/signaltest.py

```
exitFlag = 0

class myThread (threading.Thread):   #继承父类threading.Thread
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):                   #把要执行的代码写到run函数里面 线程在创建后会直接运行run函数
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

def signal_handler(signal, frame):
        print('You pressed Ctrl+C!')
        exitFlag = 1
        time.sleep(3)
        sys.exit(0)
```

Key points:

1. 建立多个线程（类）然后分别定义其特定类别的 run 函数，使之一直在 while true 的情况下运行。
2. 有一个外部的 main thread 在接收到 Ctrl+C 信号时，将每个子线程的 exitFalg 置 1，使其退出 while loop 然后 main thread 再退出。

(2) complex: INTERN_codes/Robot_car_DRL/try_multhreads_scene.py

```
class cameraThread (threading.Thread):
        def __init__(self, threadID, name, command_q, feedback_q):
                threading.Thread.__init__(self)
                self.threadID = threadID
                self.name = name
                self.command_q = command_q
                self.feedback_q = feedback_q
                self.loopcounter = 0
                self._running = True
                self._knowServerAlive = False


                self.instruction_backup = self.instruction_to_give = self.command_last = 'c3+w010q'
                '''backup command for last frame;
                   instruction-to-give is updated in current frame;
                   command_last related to last command put into command queue
                   In this way, backup rate should be faster than recording command_last'''

                self.routeList = [(300, 200)]
                self.testmode_arr = {"car_center_test_sum":0,   "car_center_valid_time":0,  "mark1_center_test_sum":0,
                self.agent =  RBcar((0,0), (0,0), (0,0), routeList[0])
                self.agent.clear()

                return
```

定义 cameraThread 的初始化，testmode_arr 是用于 server 连接上之前，单纯地使用前 200 帧平均值检测板子的位置以及小车的中心和角度。再去等待 server 与小车的连接。

```
def terminate(self):
        print "trying to terminate camera thread"
        self._running = False
        #sys.exit(0)
        return

def run(self):
        print "Starting " + self.name
        cap= self.cam_init()
        while self._running:
                self.loopcounter = self.loopcounter+1
                print "loop:"+str(self.loopcounter)+", cam:self running is " + str(self._running)
                self.cam_mainloop(cap, self.loopcounter, testmode_arr)
                self.img2command(self.name, self.command_q, self.feedback_q, self.instruction_to_give)
                time.sleep(0.025)
        print "Exiting " + self.name
        return
```

run 函数内包括了 cam_init()读取出一个有效的 cap reference.
然后再 while loop 里面:

(1) 运行 cam_mainloop()不停地对小车进行定位和计算控制命令;
(2) 将控制命令放到 queue 里面供 socketThread 读取
(3) 检查_running 参数。(这个参数由外部 mainThread 接收到 Ctrl+C 之后调用 thread.terminate()置 0，从而停止该子 Thread)

注意 thread 之间的运行时平行的；然而每个 thread 的 run()里面的函数运行时时序性的！小心一个 thread 在等另一个 thread 的信号的时候，陷在了自己的 loop 里面走不下去的情况，然后循环等待。

逻辑：
if frame_counter <= 200:
    循环测试小车定位；
else:
    算当前的控制命令；
    While(knowServerAlive == 0): //等待 server 启动，用一个 loop 同步一下
        循环去取 queue 看 server feedback == InitOK → knowServerAlive = 1

```
cv2.waitKey(25*((frame_counter <= 200) or self._knowServerAlive))
```

```
def img2command(self, threadName, command_q, feedback_q, instruction_to_give):

        queueLock.acquire()
        if(self._knowServerAlive == 1 and feedback_q.empty() != 1):
                feedback = feedback_q.get()
                print "camera knows feedback:", feedback
                command_q.put(instruction_to_give)
                self.command_last = instruction_to_give
                queueLock.release()
                return

        else:
                queueLock.release()
                return
```

此处本来是用以获得 server 正常连接的信号，或者是了解 agent 的速度反馈。然后当正常连接的时候，camThread 再向 socketThread 中放入下一个时刻的控制命令。

```
class socketThread (threading.Thread):
        def __init__(self, threadID, name, command_q, feedback_q):
                threading.Thread.__init__(self)
                self.threadID = threadID
                self.name = name
                self.command_q = command_q
                self.feedback_q = feedback_q
                self._running = True
                self._conn = None
                self._initSuccess = False
                return

        def terminate(self):
                print "trying to terminate sockect thread"
                self._running = False
                if(self._initSuccess == True):
                        self._conn.shutdown(socket.SHUT_RD)
                        self._conn.close()
                #sys.exit(0)
                return

        def run(self):
                print "Starting " + self.name
                self._conn = self.server_init(self.command_q, self.feedback_q)
                while self._running:
                        self.server_process(self._conn)
                        print "network : self running is " + str(self._running)
                        #command2server(self.name, self.q,self._running)
                print "Exiting " + self.name
                return
```

run 函数内包括了 server_init()读取出一个有效的 connection.
然后再 while loop 里面:
(1) 不停地进行 server_process()的 function
(2) Check running 参数，置 1 的方式与 cameraThread 的方式相同。

server_init()是要接受到小车传回的信号，确认 wifi 已连接，获得一个 connection 并通过 queue 告知 cameraThread 已连接。server_process()就是接收 cameraThread 放入的 command 信号，然后传给小车。具体可以看代码。

Key points:
(1) 图像处理和通信进程是异步的，但是还是有部分阶段性的同步；
(2) queueLock 的使用

3.Learning Algorithms

3.1 Basic for RL

(1) Bellman 方程

Bellman 期望方程将值函数 $Q\pi$ 展开:

$$Q\pi(s,a)=\mathrm{E}\,[rt+1+\gamma rt+1+\gamma 2rt+1+\ldots|s,a]=\mathrm{E}s',a'[r+\gamma Q\pi(s',a')|s,a]$$

Bellman 最优方程将最优值函数 $Q*$展开：

$$Q*(s,a)=\mathrm{E}s'[r+\gamma \max a' Q*(s',a')|s,a]$$

策略迭代算法求解了 Bellman 期望方程:

$$Q_{i+1}(s,a)=\mathrm{E}s'[r+\gamma Qi(s',a')|s,a]$$

值迭代算法求解了 Bellman 最优方程

$$Q_{i+1}(s,a)=\mathrm{E}s',a'[r+\gamma \max a' Qi(s',a')|s,a]$$

(2) 非线性 Sarsa 策略迭代

使用参数为 w 的 Q-network 来表示值函数

$$Q(s,a,w)\approx Q\pi(s,a)$$

通过 Q-值的均方误差来定义目标函数

$$L(w)=E[(r+\gamma Q(s',a',w)-Q(s,a,w))^2]$$

我们就有了下面的 Sarsa 梯度

$$\frac{\partial L(w)}{\partial w}=E[(r+\gamma Q(s',a',w)-Q(s,a,w))\frac{\partial Q(s,a,w)}{\partial w}]$$

借助 SGD 使用 $\frac{\partial L(w)}{\partial w}$ end-to-end 优化目标函数

(3) 深度强化学习的稳定性问题

基本的 Q-学习采用神经网络会出现振荡或者发散的情况

数据是序列化的，相邻的样本是相关的，非独立同分布的

Q-值微小变化会导致策略迅速变化，从而策略可能会振荡

数据的分布会从一个极端摇摆到另一个极端

奖励和 Q-值的范围未知

基本的 Q-学习梯度会在反向传播时变得很大从而不能稳定

(4) 深度 Q-网络（DQN）

Sample Code: DQN in openAI gym (cart-pole)

a：经验回放

为了打破关联，从 agent 自己的经验中构建数据集
根据 ϵ 贪婪策略采取行动 a_t
将 transition $(s_t, a_t, r_{t+1}, s_{t+1})$存放在 replay buffer 中
从 replay buffer 中随机采样一个小批量样本 (s, a, r, s′)
优化 Q-网络和 Q-学习目标之间的均方误差 MSE，如
$L(w)=E_{s,a,r,s'\sim D}[(r+\gamma \max_{a'} Q(s',a',w)-Q(s,a,w))^2]$

b: 固定目标 Q-网络
为了避免振荡，固定在 Q-学习目标网络的参数
根据旧的、固定的参数 w^-计算 Q-学习目标函数
$r+\gamma \max_{a'} Q(s',a',w^-)$
优化 Q-网络和 Q-学习目标之间的均方误差 MSE
$L(w)=E_{s,a,r,s'\sim D}[(r+\gamma \max_{a'} Q(s',a',w^-)-Q(s,a,w))^2]$
周期化地更新固定参数 $w^-\leftarrow w$

c: 奖励/值范围
DQN 截断奖励在$[-1,+1]$
让 Q-值不会过大
确保梯度完好（well-conditioned）
不能够区分奖励的大小
更好的方式：正规化网络输出
比如，通过 batch 正规化

(5) 连续行动上的策略梯度 (policy gradient)
使用一个权重为 u 的深度神经网络 $a=\pi(s,u)$来表示策略
定义目标函数为总折扣奖励
$J(u)=E[r1+\gamma r2+\gamma 2r3+\ldots]$
使用 SGD 来端对端优化目标函数
即调整策略参数 u 来达到更大的奖励

(6) 确定型策略梯度
策略的梯度由下式给出

$$\frac{\partial J(u)}{\partial u} = E_s\left[\frac{\partial Q_\pi(s,a)}{\partial u}\right] = E_s\left[\frac{\partial Q_\pi(s,a)}{\partial a}\frac{\partial \pi(s,u)}{\partial u}\right]$$

策略梯度是最大化提升 Q 的方向

(7) 确定型 Actor-Critic
使用两个网络
Actor 是参数为 u 的策略 $a = \pi(s,u)$
Critic 是参数为 w 的值函数 $Q(s,a,w)$
Critic 为 Actor 提供损失函数，梯度从 Critic 到 Actor 反向传播
确定型 Actor-Critic：学习规则
Critic 通过 Q-学习估计当前策略的值

$$\frac{\partial L(w)}{\partial w} = E[(r + \gamma Q(s', \pi(s'), w) - Q(s, a, w)) \frac{\partial Q(s, a, w)}{\partial w}]$$

Actor 按照提升 Q 的方向更新策略

$$\frac{\partial J(u)}{\partial u} = Es[\frac{\partial Q(s, a, w)}{\partial a} \frac{\partial \pi(s, u)}{\partial u}]$$

(8) 确定型深度策略梯度（DDPG）

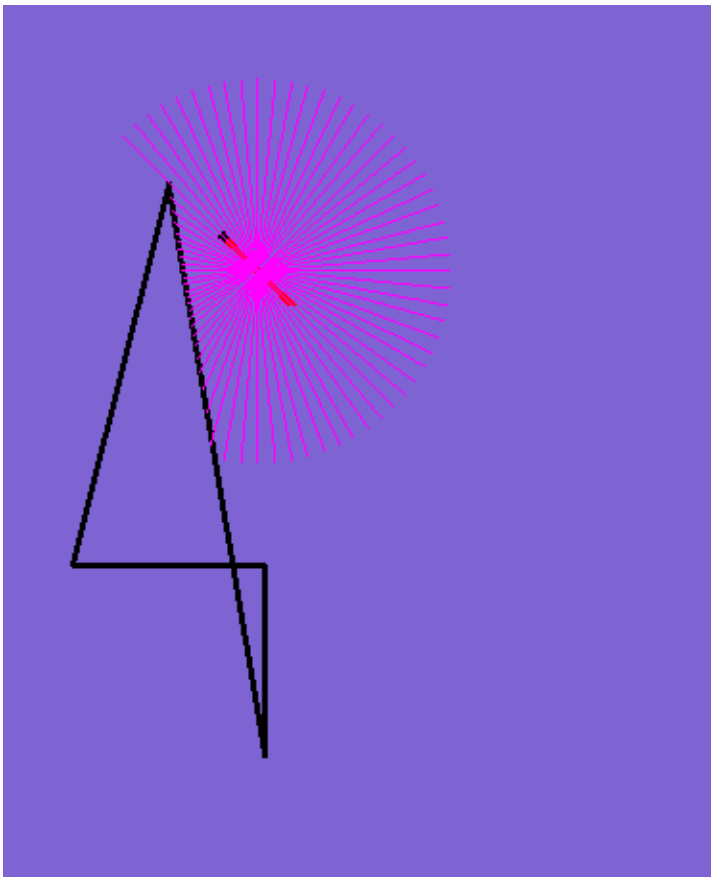
基本的 actor-critic 使用神经网络会振荡或者发散，DDPG 给出了稳定 solution:

对 actor 和 critic 均使用 experience replay buffer &冻结目标网络来避免振荡

$$\frac{\partial L(w)}{\partial w} = E_{s,a,r,s' \sim D}[(r + \gamma Q(s', \pi(s', u^-), w^-) - Q(s, a, w)) \frac{\partial Q(s, a, w)}{\partial w}]$$

$$\frac{\partial J(u)}{\partial u} = E_{s,a,r,s' \sim D}[\frac{\partial Q(s, a, w)}{\partial a} \frac{\partial \pi(s, u)}{\partial u}]$$

3.2 Virtual Env.



Virtual env 是基于 pygame module 开发的一个简单 2d 环境，有一个或多个智能体，能够获得自身周围 360 度方向的 lidar 数据。即，如果该方向上没有障碍，则能获得 lidar 最大长度；否则获得到障碍的距离。

Demo: python2 INTERN_codes/ai-gym/game.py
awsd 是 4 个命令，在画面内按 Esc 可以退出 demo

3.3 DQN – Snake

```
class DQN():
    #DQN Agent
    def __init__(self, env):

        # "env" for car agent should be virtual sensor playground
        # env.observation should be 360 degree vec
        # env.action_space should be discrete action like 'wasd' x '0123' + 'q3' = 17

        # init experience replay
        self.replay_buffer = deque()
        # init some parameters
        self.time_step = 0
        self.epsilon = INITIAL_EPSILON
        self.state_dim = 362
        self.action_dim = 3

        self.create_Q_network()
        self.create_training_method()

        self.session = tf.InteractiveSession()
        self.session.run(tf.initialize_all_variables())
        self.time_t = 0
        self.train_time = 1
        self.loss = 0
```

离散动作空间 DQN 贪吃蛇初始化 codes

```
    def create_Q_network(self): ...

    def create_training_method(self): ...

    def perceive(self, state, action, reward, next_state, done): ...

    def train_Q_network(self): ...

    def egreedy_action(self, state): ...

    def action(self, state): ...
```

```
def create_Q_network(self):

    if USE_FC_ONLY == 1:
        # network weights
        W1 = self.weight_variable([self.state_dim, 64])
        b1 = self.bias_variable([64])
        W2 = self.weight_variable([64, 20])
        b2 = self.bias_variable([20])

        W3 =  self.weight_variable([20, self.action_dim])
        b3 = self.bias_variable([self.action_dim])

        if DEBUG_MODE:
            print "All weights and bias:"
            print "W1: ", W1
            print "b1: ", b1
            print "W2: ", W2
            print "b2: ", b2
            print "W3:", W3
            print "b3: ", b3
            time.sleep(0.5)

        #input layer
        self.state_input = tf.placeholder("float", [None, self.state_dim])
        # hidden layers
        h1_layer = tf.nn.relu(tf.matmul(self.state_input, W1) + b1)
        h2_layer = tf.nn.relu(tf.matmul(h1_layer, W2) + b2)
        # Q Value layer
        self.Q_value = tf.matmul(h2_layer, W3)+b3
```

在这里定义网络结构，该图代码定义了一个 362->64->20->3 的全连接网络。

```
def create_training_method(self):
    self.action_input = tf.placeholder("float", [None, self.action_dim]) # one hot presentation
    self.y_input = tf.placeholder("float", [None])
    Q_action = tf.reduce_sum(tf.mul(self.Q_value, self.action_input), reduction_indices = 1)
    self.cost = tf.reduce_mean(tf.square(self.y_input - Q_action))
    self.optimizer = tf.train.AdamOptimizer(0.0001).minimize(self.cost)
```

定义 placeholder, loss, optimizer.

```python
def train_Q_network(self):

    self.time_step += 1
    # Step 1: obtain random minibatch from replay memory
    minibatch = random.sample(self.replay_buffer, BATCH_SIZE)
    state_batch = [data[0] for data in minibatch] # refer to append order
    action_batch = [data[1] for data in minibatch]
    reward_batch = [data[2] for data in minibatch]
    next_state_batch = [data[3] for data in minibatch]

    # Step 2: calculate y
    y_batch = []
    Q_value_batch = self.Q_value.eval(feed_dict={self.state_input: next_state_batch})
    for i in range(0, BATCH_SIZE):
        done = minibatch[i][4]
        if done:
            y_batch.append(reward_batch[i])
        else:
            y_batch.append(reward_batch[i]+GAMMA*np.max(Q_value_batch[i]))

    self.optimizer.run(feed_dict={
            self.y_input: y_batch,
            self.action_input: action_batch,
            self.state_input: state_batch
      })
```

在这里计算 y_i, Q，然后进行训练。

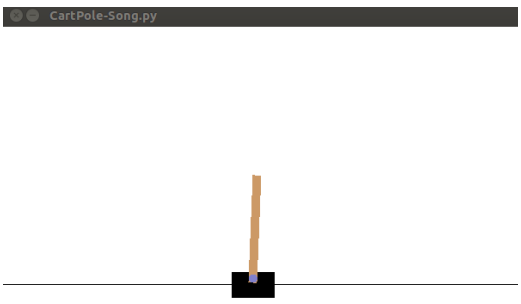
Demo: python2 INTERN_codes/ai-gym/car_DQN/myRL.py

展示了 一个训练完成的 DQN 贪吃蛇。

State space 由 360° virtual sensor 数据以及小车自身极坐标下到目标点的距离和前进方向夹角差。Action space 是 awd 没有后退。

只要把 myRL.py 中 restore request 设为 0 即可不调用 saver，重新 train 一个 dqn agent.

Demo: python2 INTERN_codes/ai-gym/other/**.py



openAI gym 内置的游戏环境，可以套用自己写的 RL 网络结构来测试算法

3.4 DDPG – Snake & Car

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

实现细节：

(1) Replay buffer (to minimize correlations between samples)
(2) 2 separated network for both critic/actor ($\tau << 1$, “soft update”)
(3) Batch normalization
(4) Exploration (add a noise process)

Actor network：

```
def __init__(self, sess, state_dim, action_dim, action_bound, learning_rate, tau):
    self.sess = sess
    self.s_dim = state_dim
    self.a_dim = action_dim
    self.action_bound = action_bound
    self.learning_rate = learning_rate
    self.tau = tau

    # Actor Network
    self.inputs, self.out, self.scaled_out = self.create_actor_network()

    self.network_params = tf.trainable_variables()

    # Target Network
    self.target_inputs, self.target_out, self.target_scaled_out = self.create_actor_network()

    self.target_network_params = tf.trainable_variables()[len(self.network_params):]

    # Op for periodically updating target network with online network weights
    self.update_target_network_params = \
        [self.target_network_params[i].assign(tf.mul(self.network_params[i], self.tau) + \
            tf.mul(self.target_network_params[i], 1. - self.tau))
            for i in range(len(self.target_network_params))]

    # This gradient will be provided by the critic network
    self.action_gradient = tf.placeholder(tf.float32, [None, self.a_dim])

    # Combine the gradients here
    self.actor_gradients = tf.gradients(self.scaled_out, self.network_params, -self.action_gradient)

    # Optimization Op
    self.optimize = tf.train.AdamOptimizer(self.learning_rate).\
        apply_gradients(zip(self.actor_gradients, self.network_params))

    self.num_trainable_vars = len(self.network_params) + len(self.target_network_params)
```

Key points:

1. 2 个 network，network_params[]和 target_network_params[]是 2 个网络的 trainable_tensor 的 list. Update 的时候是 assign 的方式乘以 tau 让 target_network_params 缓慢变化。目的是让 y_i 的计算较为稳定，避免 training process 震荡
2. 由于 actor 是接收 critic 的 gradients 来 train 的所以有一个 action_gradient 的 placeholder 用于显式地接收 critic 对 action 的求导。self.actor_gradients = tf.gradients(self.scaled_out, self.network_params, -self.action_gradient)是在求近似 policy gradient。
3. tf.gradient(x, y, z) =$\frac{\partial x}{\partial y} * z$

```
def create_actor_network(self):
    inputs = tf.placeholder("float", [None, self.s_dim])
    # actor network u(s)
    # weight for policy output layer
    s_W_fc1 =  weight_variable([self.s_dim, 64])
    s_b_fc1 = bias_variable([64])
    s_W_fc2 =  weight_variable([64, 20])
    s_b_fc2 = bias_variable([20])
    a_h1_layer = tf.nn.relu(tf.matmul(inputs, s_W_fc1) + s_b_fc1)
    a_h2_layer = tf.nn.relu(tf.matmul(a_h1_layer, s_W_fc2) + s_b_fc2)

    # weight for value output layer
    W_fc3 = weight_variable([20, self.a_dim])
    b_fc3 = bias_variable([self.a_dim])

    out = tf.nn.tanh(tf.matmul(a_h2_layer, W_fc3)+ b_fc3)
    scaled_out = tf.mul(out, self.action_bound) # Scale output to -action_bound to action_bound
    return inputs, out, scaled_out
```

图为一个简单的全连接 fc 网络作为 actor 网络

```python
def train(self, inputs, a_gradient):
    self.sess.run(self.optimize, feed_dict={
        self.inputs: inputs,
        self.action_gradient: a_gradient
    })

def predict(self, inputs):
    return self.sess.run(self.scaled_out, feed_dict={
        self.inputs: inputs
    })

def predict_target(self, inputs):
    return self.sess.run(self.target_scaled_out, feed_dict={
        self.target_inputs: inputs
    })

def update_target_network(self):
    self.sess.run(self.update_target_network_params)

def get_num_trainable_vars(self):
    return self.num_trainable_vars
```

图为 actor network 其余函数定义方式

Critic network：

```python
# Create the critic network
self.inputs, self.action, self.out = self.create_critic_network()

self.network_params = tf.trainable_variables()[num_actor_vars:]

# Target Network
self.target_inputs, self.target_action, self.target_out = self.create_critic_network()

self.target_network_params = tf.trainable_variables()[(len(self.network_params) + num_actor_vars):]

# Op for periodically updating target network with online network weights with regularization
self.update_target_network_params = \
    [self.target_network_params[i].assign(tf.mul(self.network_params[i], self.tau) + tf.mul(self.target_network_params[i], 1. - self.tau))
        for i in range(len(self.target_network_params))]

# Network target (y_i)
self.predicted_q_value = tf.placeholder(tf.float32, [None, 1])

# Define loss and optimization Op
self.loss =tf.reduce_mean(tf.square(self.predicted_q_value - self.out))
self.optimize = tf.train.AdamOptimizer(self.learning_rate).minimize(self.loss)
```

Key points:

1. num_actor_vars 是输入 critic 的一个外部数据。还有注意，network_params[]和 target_network_params[]的提取方式。
2. 定义 loss 和 optimize
3. Critic network 的网络结构和其余函数的构造方式与 actor network 类似。
4. 最后 critic 和 actor 在外部的一个 object 里面综合起来即可。Initialization，perceive，train 等等与 dqn 网络结构比较相似。

Dqn for continuous time network：(ddpg)

```
class DQN_CT():
        #DQN Continuous output for Agent
        def __init__(self, env):

                # ''env'' for car agent should be virtual sensor playground
                # env.observation should be 360 degree vec + distance + angle-diff
                # init experience replay
                self.replay_buffer = deque()
                # init some parameters
                self.time_step = 0
                self.state_dim = 362    # 360 vec + distance + angle diff
                self.action_dim = 2      # speed + steer
                self.speed_range = 5
                self.steer_range = 22.5

                self.session = tf.InteractiveSession()
                self.create_Q_network()
                self.session.run(tf.initialize_all_variables())
                self.actor.update_target_network()
                self.critic.update_target_network()
```

1. 可以在 InteractiveSession()里面定义 config 的参数，然后即可使用 gpu，或者是 allow_growth 的设置，这样可以在一台机器上同时跑多于 10 个这种小实验，否则内存一下子就会被占满。
2. 先 initialize_all_variables，然后再 restore from saver.(如果需要恢复 checkpoint)

```
def create_Q_network(self):
        self.actor = ActorNetwork(self.session, self.state_dim, self.action_dim, [self.speed_range, self.steer_range], \
                ACTOR_LEARNING_RATE, TAU)

        self.critic = CriticNetwork(self.session, self.state_dim, self.action_dim, \
                CRITIC_LEARNING_RATE, TAU, self.actor.get_num_trainable_vars())


def perceive(self, state, action, reward, next_state, done):
        self.time_t += 1

        self.replay_buffer.append((state, action, reward, next_state, done))

        if len(self.replay_buffer) > REPLAY_SIZE:
                self.replay_buffer.popleft()

        if len(self.replay_buffer) > BATCH_SIZE*2 and self.time_t%self.train_time == 0:
                self.train_Q_network()
                self.time_t = 0
```

升级版 perceive 应该是操作两个 buffer,一个控制正样本，一个控制负样本。

```python
def train_Q_network(self):

        self.time_step += 1
        # Step 1: obtain random minibatch from replay memory
        minibatch = random.sample(self.replay_buffer, BATCH_SIZE)
        state_batch = [data[0] for data in minibatch] # refer to append order
        action_batch = [data[1] for data in minibatch]
        reward_batch = [data[2] for data in minibatch]
        next_state_batch = [data[3] for data in minibatch]

        # Step 2: calculate y
        y_batch = []
        target_q = self.critic.predict_target(next_state_batch, self.actor.predict_target(next_state_batch))

        for i in range(BATCH_SIZE):
            done = minibatch[i][4]
            if done:
                y_batch.append(reward_batch[i])
            else:
                y_batch.append(reward_batch[i] + GAMMA * target_q[i])

        # Update the critic given the targets
        predicted_q_value, _ = self.critic.train(state_batch, action_batch, np.reshape(y_batch, (BATCH_SIZE, 1)))


        a_outs = self.actor.predict(state_batch)
        grads = self.critic.action_gradients(state_batch, a_outs)
        self.actor.train(state_batch, grads[0])

        # Update target networks
        self.actor.update_target_network()
        self.critic.update_target_network()
```

Key points:

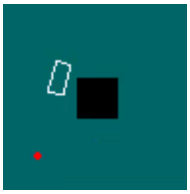
1. 用 target_network 算 y_i, 而 predict Q 的时候还是用 network
2. 每次 train 完需要 update_target_network * tau

```python
def egreedy_action(self, state):

        # a_explore = self.actor.predict(np.reshape(state+(random.random()*5), (-1, self.actor.s_dim)))
        a_deploy = self.actor.predict(np.reshape(state, (-1, self.actor.s_dim)))
        Q_value = self.critic.predict(np.reshape(state, (-1, self.critic.s_dim)), np.reshape(a_deploy, (-1, self.critic.a_dim)))


        if random.random() <= self.epsilon:
                self.epsilon -= (self.epsilon - FINAL_EPSILON)/10000
                ## Note!!!!
                ## Initially, self.epsilon -= (INITIAL_EPSILON - FIANAL_EPSILON)/10000
                ## But actually, result is after 10000/300 ~= 33 episode, exploration rate is 0.01
                action = [0., 0.]
                action = [random.random()*self.speed_range, random.random()*self.steer_range]

        else:
                self.epsilon -= (self.epsilon - FINAL_EPSILON)/10000
                action = [0., 0.]
                action = a_deploy[0]

        if DEBUG_MODE:
                print "Q_value:", Q_value
                print "agent speed", action[0]
                print "agent steer", action[1]
                time.sleep(0.05)

        return action[0], action[1], Q_value
```

1. 还是使用一个 epsilon 来控制探索率。

Demo: python2 INTERN_codes/ai-gym/ddpg/myRL_CT.py
一个连续动作的贪吃蛇，区域内没有障碍。如果需要障碍，则在 virtual_Env.py 里面将 various_border 参数设为非 0.

Demo: python2 INTERN_codes/ai-gym/demo_playground/demo_exp/myRL_CT_simu.py
自动避障的驾驶场景。区域内有一个简单障碍。
(在 ddpg network 文件中增加了一个 simulator 的网络，输入是此 state，输出是下一时刻 state 数据。然而，simulator 的预测效果很差)



Actor-critic network：自适应地跑向目标点和以及避障

3.5 DRL 一些心得

(1) 对不同维度的输入，必须要进行 bn
(2) RL 的问题中，正负样本分布极度不平衡，需要控制学习时的正负样本率。
(3) 对于激活函数的使用，感觉使用 relu 时，train 的速度比较快，可能是因为 gradient 回传效率高，不容易出现 vanish 的情况。相反，sigmoid 可能能够拟合得更好(因为它的非线性特征)，但是 sigmoid 的导数在 0 时最大也只有 0.25，所以很容易出现 gradient vanish 的情况。tanh 比较适合使用在输出有正负的最后一层。
(4) 对于 actor-critic 这个网络，Q(s,a),由于 action 与 state 的维数差别太大，所以 action 输入应该在网络第 2 或 3 层之后再 concatenate 进去。然后加入了不同 dimension 的数据之后，一定一定一定要记得 bn 或者规范化一下。

4 Searching and Path-planing

4.1 A-star & D-star

http://blog.csdn.net/chinaliping/article/details/8525411

Demo: python2 INTERN_codes/A-star.py

4.2 Monte Carlo Tree Search in Go

AlphaGo 总体上包含离线学习(上图上半部分)和在线对弈(上图下半部分)两个过程。
离线学习过程分为三个训练阶段。

第一阶段：利用 3 万多幅专业棋手对局的棋谱来训练两个网络。
一个是基于全局特征和深度卷积网络(CNN)训练出来的**策略网络(Policy Network)**，其主要作用是给定当前盘面状态作为输入，输出下一步棋在棋盘其它空地上的落子概率。
另一个是利用局部特征和线性模型训练出来的**快速走棋策略(Rollout Policy)**。
策略网络速度较慢，但精度较高；快速走棋策略反之。

第二阶段：利用第 t 轮的策略网络与先前训练好的策略网络互相对弈，利用增强式学习来修正第 t 轮的策略网络的参数，最终得到增强的策略网络。

第三阶段：先利用普通的策略网络来生成棋局的前 U-1 步(U 是一个属于[1, 450]的随机变量)，然后利用随机采样来决定第 U 步的位置(这是为了增加棋的多样性，防止过拟合)。
随后，利用增强的策略网络来完成后面的自我对弈过程，直至棋局结束分出胜负。此后，第 U 步的盘面作为特征输入，胜负作为 label，学习一个价值网络(Value Network)，用于判断结果的输赢概率。
价值网络其实是 AlphaGo 的一大创新，围棋最为困难的就是很难根据当前的局势来判断最后的结果，这点职业棋手也很难掌握。
通过大量的自我对弈，AlphaGo 产生了 3000 万盘棋局，用作训练学习价值网络。但由于为其的搜索空间太大，3000 万盘棋局也不能帮 AlphaGo 完全攻克这个问题。

在线对弈过程包括以下 5 个关键步骤：其核心思想是**在蒙特卡洛搜索树(MCTS)中嵌入深度神经网络来减少搜索空间。AlphaGo 并没有具备真正的思维能力。**
1、根据当前盘面已经落子的情况提取相应特征；
2、利用 SL 策略网络估计出棋盘其他空地的落子概率；
3、根据落子概率来计算此处往下发展的权重，初始值为落子概率本身(如 0.18)。实际情况可能是一个以概率值为输入的函数，此处为了理解简便。
4、利用价值网络和快速走棋网络分别判断局势，两个局势得分相加为此处最后走棋获胜的得分。
这里使用快速走棋策略是一个用速度来换取量的方法，从被判断的位置出发，快速行棋至最

后，每一次行棋结束后都会有个输赢结果，然后综合统计这个节点对应的胜率。
价值网络只要根据当前的状态便可直接评估出最后的结果。两者各有优缺点、互补。
5、利用第四步计算的得分来更新之前那个走棋位置的权重(如从 0.18 变成了 0.12)；此后，从权重最大的 0.15 那条边开始继续搜索和更新。
这些权重的更新过程应该是可以并行的。当某个节点的被访问次数超过了一定的门限值，则在蒙特卡罗树上进一步展开下一级别的搜索。

4.3 KR-UCT

Algorithm 1 Kernel Regression UCT

1: **procedure** KR-UCT(*state*)
2: **if** *state* is terminal **then**
3: **return** utility(*state*), *false*
4: *expanded* $\leftarrow$ *false*
5: $A \leftarrow$ actions considered in *state*
6: *action* $\leftarrow \text{argmax}_{a \in A} \mathbb{E}(v|a) + C\sqrt{\frac{\log \sum_{b \in A} W(b)}{W(a)}}$
7: **if** $\sqrt{\sum_{a \in A} n_a} < |A|$ **then**
8: *newState* $\leftarrow$ child of *state* by taking *action*
9: rv, *expanded* $\leftarrow$ KR-UCT(*newState*)
10: **if not** *expanded* **then**
11: *newAction* $\approx \text{argmin}_{K(action,a)>\tau} W(a)$
12: add $newAction$ to $state$
13: *newState* $\leftarrow$ child of *state* by taking *newAction*
14: add initial actions to *newState*
15: $rv \leftarrow$ ROLLOUT(*newState*)
16: Update $\bar{v}_{action}, n_{action}$, and KR with rv
17: **return** rv, *true*

$$\begin{aligned} \mathbb{E}(v|a) &= \frac{\sum_{b \in A} K(a,b)\bar{v}_b n_b}{\sum_{b \in A} K(a,b) n_b} \\ W(a) &= \sum_{b \in A} K(a,b) n_b. \end{aligned}$$

Problem:
(1)Alpha Go 里面的蒙特卡洛树限制于离散动作空间的搜索。
(2)真实世界中行动带有不确定性的情况，如何搜索。

Innovation in KR-UCT:
(1)选择(selection)：用 line 6 的公式选取一个 mean+bonus 最大的动作作为探索点。
(2)扩展(expansion)：用 line 7 的 function 做 decision bound 决定变深还是变广。会发现，tree 先是往深长，访问次数达到一定以后超过了 decision bound，tree 会变宽。这个变宽是通过在原先选择的动作上，基于预设的 kernel sample 出 k 个动作。这 K 个动作还是在原先动作附近（用一个σ bound 下限），然后选择这 k 个动作中离原动作最远的一个进

行探索。Idea 是基于 sample 出来的目前最优解附近进行探索，但是希望找到一个未被 well-represented 的动作（即 k 个中最远的那个）探索可能性。
(3)评估(evaluation)： 变深的时候，即再次 call UCT function。变广的时候，就使用 rollout policy 进行快速的 simulation. 然后把这个 simulation 值作为该 new action node 的 state value. 最最最最关键的是，使用这个 new action node value 去更新原来选择出来的“目前最优解”，这样就间接地把 uncertianty 信息利用相关度大小回归到别的 node 上。
(4)回溯(backup)：基于子节点的 value 和访问次数更新父节点的 value 和访问次数
(5)终选(final selection)：使用负数的 C.这样间接地选择了 mean+访问次数较多的 node，idea 是选择更加确定的 node。可能出现的问题是，探索时的深度可能不一定能在终选时被利用上。（因为探索和终选的路径是不一样的）

4.4 V-KRUCT

Algorithm 1 Variation of Kernel Regression UCT

function V-KRUCT(*simulator*, *thisNode*)
  if simulator(thisNode.info) is terminal **then**
    return thisNode.reward, thisNode.visitNum, *false*
  end if
  if thisNode.depth is maxDepth **then**
    return thisNode.value, thisNode.visitNum, *false*
  end if
  expanded ← *false*
  if thisNode.children ≤ len(action considered at this node) **then**
    add resulting states as children to thisNode for $a \in$ action set **A**
  end if
  totalVisitNum = $\sum$ child.visitNum
  selectedNode = $\arg\max_{a \in \mathbf{A}} \mathbb{E}(\mathrm{v}|\mathrm{a}) + C\sqrt{\frac{\log(\text{totalVisitNum})}{\text{a.visitNum}}}$
  if $\sqrt{\text{totalVisitnum}} \leq |\mathbf{A}|$ **then**
    value, *visitNum*, *expanded* ← V-KRUCT(*simulator*, *selectedNode*)
  end if
  if not *expanded* **then**
    newAction ≈ selectedNode.action + random process $\mathbb{P}$
    newNode ← resulting info by taking *newAction* to thisNode.children
    add initial actions to *newNode*
    for $i = 0 \rightarrow rolloutTimes$ **do**
      rv += ROLLOUT(simulator, newNode, steps)
    end for
    newNode.value ← newNode.reward + $\gamma \frac{rv}{rolloutTimes}$
    newNode.visitNum ← 1
    selectedNode.value ← ω·newNode.value + (1-ω)·selectedNode.value
    selectedNode.visitNum += 1
  end if
  thisNode.value = thisNode.reward + $\gamma \cdot \frac{\sum_{i \in \text{child}} v_i \cdot n_i}{\sum_{j \in \text{child}} n_j}$
  thisNode.visitNum += 1
  return thisNode.value, thisNode.visitNum, *true*
end function

Problem & Statement:

(1) KRUCT 里面的蒙特卡洛树搜索是在冰壶模拟器里面完成的。这个场景值需要针对一个 state 产生一个击打动作的数据，并不需要产生一个 decision sequence(比如在驾驶场景中，绕过障碍物的决策可以由左偏，前进，右偏回到原轨道的决策序列来表示)

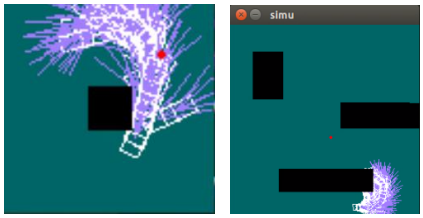
(2) 区别于 A 星和 D 星的基于状态空间的搜索，V-KRUCT 是直接基于动作空间的搜索。

(3) A 星和 D 星是需要将地图栅格化的做法。V-KRUCT 不需要进行这样的一步。

Innovation in V-KRUCT:

(1)Terminal state 的定义：在 KRUCT 中由于冰壶游戏单方最长不超过 8 手，而且会有提前结束的情况，所以不太需要考虑 tree 无限长深的情况。与驾驶的模拟场景并不相同，所以 terminal state 的判断中加入了 tree maximum depth 的考量。(line 2-7)
(2)选择(selection)：用 line 13 的公式选取一个 mean+bonus 最大的动作作为探索点。
(3)扩展(expansion)： 用 line 14 的 function 做 decision bound 决定变深还是变广。会发现，tree 先是往深长，访问次数达到一定以后超过了 decision bound，tree 会变宽。这个变宽是通过在原先选择的动作上，基于某个随机过程产生一个新的动作来探索最优解可能性。
(3)评估(evaluation)： 变深的时候，即再次 call function。变广的时候，就使用 rollout policy 进行快速的 simulation. 然后把这个 simulation 值作为该 new action node 的 state value. 最最最最关键的是，使用这个 new action node value 去更新原来选择出来的“目前最优解”，这样就间接地把 uncertianty 信息利用相关度大小回归到原来选择的 node 上。
(4)回溯(backup)： 基于子节点的 value 和访问次数更新父节点的 value 和访问次数。这里使用了 bellman equation 的 value iteration，将一个带有长远目光（通过 γ 来实现， line 29）的 decision sequence 选择出来。
(5)终选(final selection)： 使用负数的 C.这样间接地选择了 mean+访问次数较多的 node，idea 是选择更加确定的 node。可能出现的问题是，探索时的深度可能不一定能在终选时被利用上。（因为探索和终选的路径是不一样的）

Demo: python2 INTERN_codes/ai-gym/demo_playground/UCT/search.py



Simple border & Complex border

5 Debug Tool – Tensorboard

为什么我们要使用 tensorboard：

私以为，在 train 的过程中，一定需要：

1. 系统的评价标准(比如 acc，以及 RL 中不断在提升的 Q 值)
2. 说明模型的收敛性的数据(比如 loss 的稳定下降)
3. 对训练过程进行监测的数据(比如 gradient 在合理的范围内)

而 Tensorboard 是 TensorFlow 自带的一个强大的可视化工具

(1) 功能

Event：展示训练过程中的统计数据（最值，均值等）变化情况

Image：展示训练过程中记录的图像

Audio：展示训练过程中记录的音频

Histogram：展示训练过程中记录的数据的分布图

一般只使用 event 和 histogram

(2) 原理

在运行过程中，记录结构化的数据

运行一个本地服务器，监听 6006 端口

请求时，分析记录的数据，绘制

(3) 实现

在构建 graph 的过程中，记录你想要追踪的 Tensor

```
with tf.name_scope('output_act'):
    hidden = tf.nn.relu6(tf.matmul(reshape, output_weights[0]) + output_biases)
tf.histogram_summary('output_act', hidden)
```

其中，

histogram_summary 用于生成分布图，也可以用 scalar_summary 记录存数值

使用 scalar_summary 的时候，tag 和 tensor 的 shape 要一致

name_scope 可以不写，但是当你需要在 Graph 中体现 tensor 之间的包含关系时

(4) 封装

官网封装了一个函数，可以调用来记录很多跟某个 Tensor 相关的数据：

```
def variable_summaries(var, name):
    """Attach a lot of summaries to a Tensor."""
    with tf.name_scope('summaries'):
        mean = tf.reduce_mean(var)
        tf.scalar_summary('mean/' + name, mean)
        with tf.name_scope('stddev'):
            stddev = tf.sqrt(tf.reduce_sum(tf.square(var - mean)))
        tf.scalar_summary('sttdev/' + name, stddev)
        tf.scalar_summary('max/' + name, tf.reduce_max(var))
        tf.scalar_summary('min/' + name, tf.reduce_min(var))
        tf.histogram_summary(name, var)
```

只有这样记录过 max 和 min 的 Tensor 才会出现在 Event 里面

Graph 的最后要写一句这个，给 session 回调

```
merged = tf.merge_all_summaries()
```

(5) Session 中调用

构造一个 writer，在 train（或 valid）的时候写数据：

```
train_writer = tf.train.SummaryWriter(summary_dir + '/train', session.graph)
```

这里的 summary_dir 存放了运行过程中记录的数据，等下启动服务器要用到

构造 run_option 和 run_meta，在每个 step 运行 session 时进行设置形如：

```
summary, _, l, predictions =
    session.run([merged, optimizer, loss, train_prediction], options=run_options, feed_dict=feed_dict)
```

注意要把 merged 拿回来，并且设置 options

在每次训练时，记一次：

```
train_writer.add_summary(summary, step)
```

达到一定训练次数后，记一次 meta 做一下标记

```
train_writer.add_run_metadata(run_metadata, 'step%03d' % step)
```

(6) 查看可视化结果

启动 TensorBoard 服务器：

```
tensorboard --logdir=Your/log/DIR…
```

然后在浏览器输入给出的端口地址（形如 http://127.0.0.1:6006）就可以访问到 tensorboard 的结果

Wiki:

1. 可以使用 tf.gradient 计算 grad 然后用这些工具进行可视化。对 gradient 大小进行一个监督，避免 gradient vanish 的情况
2. 看 weights，bias，以及其他数据的 mean，std 分析数据分布情况。
3. summary_dir + '/train'中，summary_dir 就可以是命令行中的 logdir。如果 summary_dir 下有许多子文件夹表示做过的多组实验，那么运行

   ```
   tensorboard --logdir=Your/log/DIR…
   ```

   的时候，就可以让多组实验数据同时进行可视化上的比较。