به نام خدا

دانشگاه صنعتی امیرکبیر
دانشکده مهندسی کامپیوتر

اصول علم ربات

پروژه پایانی

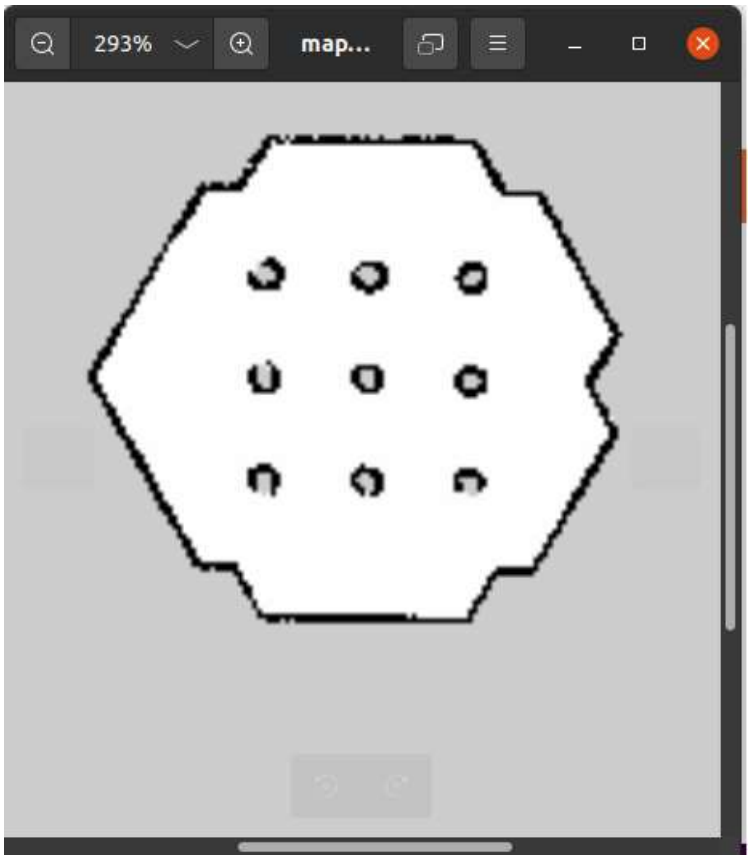
نام و نام خانوادگی	حمیدرضا همتی _ آرش اعلائی
شماره دانشجویی	۹۶۳۱۰۷۹ - ۹۷۳۱۰۷۲
تاریخ ارسال گزارش	

فهرست گزارش سوالات

سناریو اول

گام اول:

در شکل زیر تصویر نهایی از نقشه ایجاد شده را میتوانید مشاهده کنید. همچنین فایل ها در مسیر codes/senario1/step_one قرار دارند.

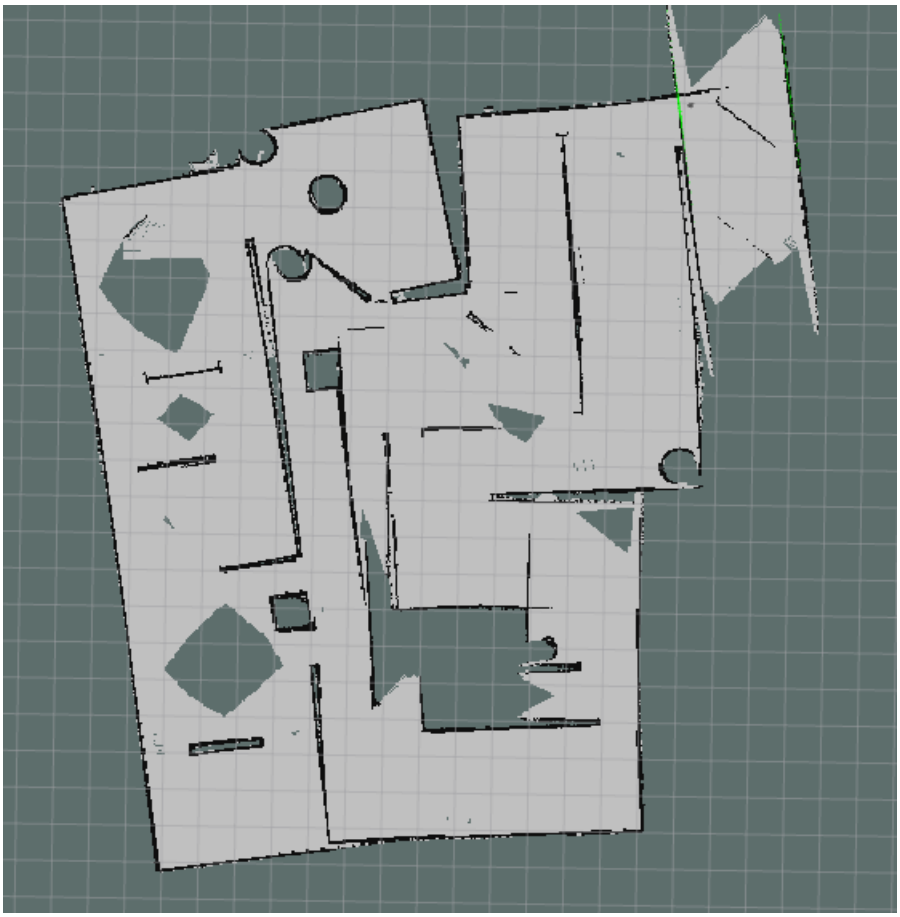


گام دوم:

کد این بخش با نام explorer.py و همچنین عکس و فایل های نقشه در مسیر codes/senario2/step_two قرار دارد.

کد این بخش مشابه کد حل ماز تمرین قبل است. در این کد ربات با استفاده از pd کنترلر دیوار را دنبال میکند.

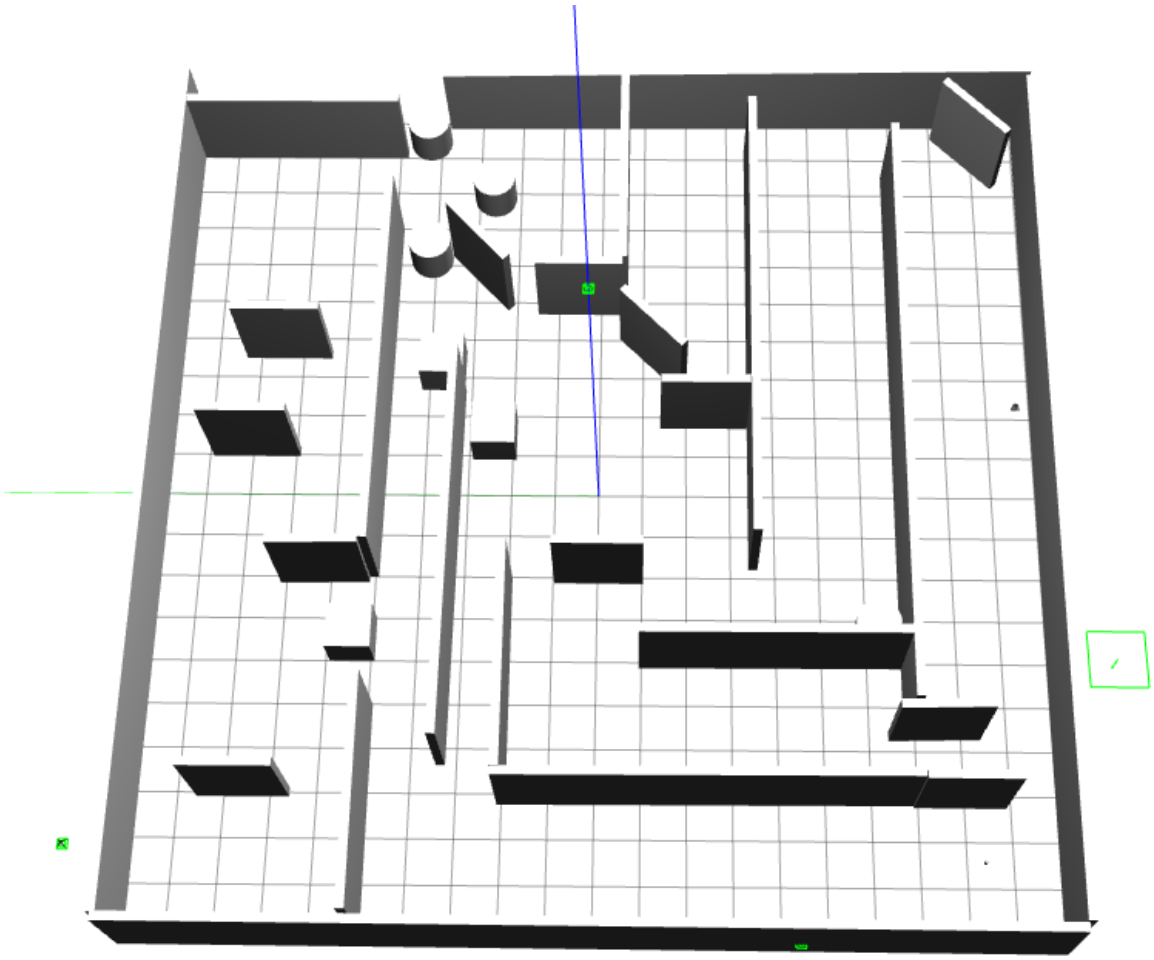
در شکل زیر میتوانید نقشه تهیه شده را مشاهده نمایید.



ربات بدون هیچ گونه برخورد یا گیر کردنی میتواند کل نقشه را پیمایش کند. ما با یک تست ۲۰ دقیقه ایی توانستیم کل نقشه را طی کنیم ولی میشکل تاین است که ربات وقتی در یک محسطی که قبلا کاوش کرده قرار میگیرد آن را برروی بخش جدیدی میکشد برای همین شکل خیلی دقیق نیست.

همانطور که میبینید بخش بالا سمت راست در شکل زیر بخش راست نقشه است که چون ۲ بار پیمایش شده است دوبار هم در rviz رسم شده.در شکل آمده در بالا دو بخش بالا سمت راست و پایین سمت راست در واقع یک جا هستند.

اما ربات تمام مسیر را پیمایش میکند.



سناریو دوم

فاز اول: محاسبه c,m,h,smothed_h

در این بخش ابتدا باید مقادیر c و m را محاسبه کنیم. توابع مشخص شده در شکل های زیر این وظیفه را دارند. همچنین به دلیل اینکه ماکسیمم رنج دید سنسور لیدار 3.5 متر است مقادیر a و b به ترتیب 0.875 و 0.25 میباشد.

```
335         while not rospy.is_shutdown():
336             ## calculating polar obstacle density (c,m,h,h')
337             c = self.vision()
338             m = self.magnitude_calculator(c)
339             h = self.pod(m)
340             smothed_h = self.smothed_pod(h)
```

کد بالا ترتیب صدا شدن متد های مربوط به محاسبه c تا smothed_h را در لوپ اصلی کد نشان میدهد.

```
67      def vision(self):
68          laser_data = rospy.wait_for_message("/scan", LaserScan)
69          laser_data = laser_data.ranges[0:359]
70          return laser_data
71
```

```
72      ## m = (ci,j)^2 * (a - b*di,j)
73      def magnitude_calculator(self, c):
74          m = [None] * len(c)
75          for degree, dist in enumerate(c):
76              value = 0
77              if dist != inf:
78                  value = self.a - (self.b * dist)
79              m[degree] = value
80          return m
81
```

دو تابع مشخص شده در شکل های بالا مقدار m را محاسبه میکنند. در صورتی که مقدار سنسور برای یک درجه inf بود برای آن مقدار 0 و در غیر این صورت طبق فرمول آمد در خط ۷۸ کد مقدار آن مشخص میشود.

```
    ## polar obstacle density --- h_k
    ## h_k = sigma(m)
    ## k is number of sectors and is equal to degrees/number of degrees in each sector --> 360/5 = 72
    def pod(self, m):
        h = [0] * self.k
        i = 0
        for degree, value in enumerate(m):
            if int(degree / self.alpha) != i:
                i += 1
            h[i] += value
        return h
```

تابع مشخص شده در شکل بالا مقدار polar obstacle density یا h را طبق فرمول زیر محاسبه میکند

$$h_k = \sum_{i,j} m_{i,j}$$

```
    ## smothed_h
    def smothed_pod(self, h):
        smothed_h = [None] * self.k
        for i in range(self.k):
            if i == 0:
                smothed_h[i] = (h[i - self.l] + 2 * h[i - self.l + 1] + 2 * h[i] + 2 * h[i + self.l - 1] + h[0]) / (
                        2 * self.l + 1)
            elif i == 71:
                smothed_h[i] = (h[i - self.l] + 2 * h[i - self.l + 1] + 2 * h[i] + 2 * h[0] + h[1]) / (2 * self.l + 1)
            else:
                smothed_h[i] = (h[i - self.l] + 2 * h[i - self.l + 1] + 2 * h[i] + 2 * h[i + self.l - 1] + h[
                    i + self.l]) / (2 * self.l + 1)
        return smothed_h
```

در نهایت پس از محاسبه h یا pod مقادیر آن را طبق فرمول زیر نرمال میکنیم. تابع شکل بالا این کار را انجام میدهد.

$$h'_k = \frac{h_{k-l}+2h_{k-l+1}+\ldots+lh_k+\ldots+2h_{k+l-1}+h_{k+l}}{2l+1}$$

پس از محاسبات گفته شده در بالا به پلات کردن هیستوگرام های smothed_h و h در شرایط مختلف میپردازیم. از این کار برای بهتر مشخص کردن مقدار threshold برای پیدا کردن قعر و قله ها استفاده میکنیم.

توابع زیر برای پلات کردن هیستوگرام دایره ایی و خطی استفاده شدند.

```
    def bar_plot(self, data):
        data = np.array(data)
        index2 = np.array([i for i in range(72)])

        df = pd.DataFrame(list(zip(data, index2)), columns=['data', 'index'])

        mask1 = data < self.threshold
        mask2 = data >= self.threshold

        plt.bar(index2[mask1], data[mask1], color='green')
        plt.bar(index2[mask2], data[mask2], color='red')
        plt.show()
```

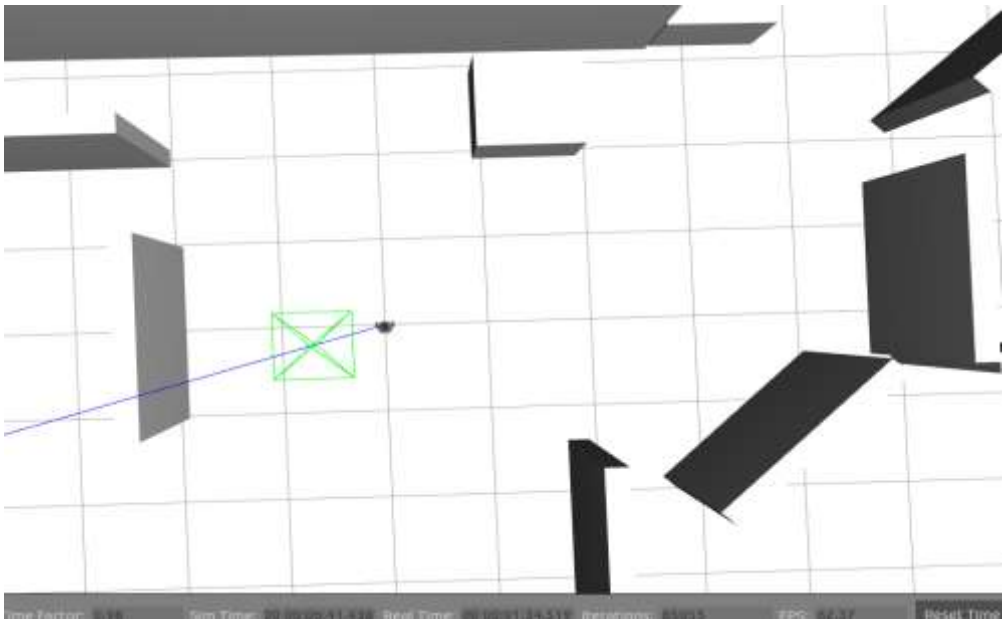
```
    def bar_plot_circular(self, data):
        data = np.array(data)
        index = np.array([i for i in range(72)])

        df = pd.DataFrame(
            {
                'index': index,
                'data': data
            })

        plt.figure(figsize=(20, 10))
        # plot polar axis
        ax = plt.subplot(111, polar=True)
        # remove grid
        plt.axis('off')
        upperLimit = max(data)
        lowerLimit = 0
        _max = df['data'].max()
        slope = (_max - lowerLimit) / _max
        heights = slope * df.data + lowerLimit
        # Compute the width of each bar. In total we have 2*Pi = 360°
        width = 2 * np.pi / len(df.index)
        # Compute the angle each bar is centered on:
        indexes = list(range(1, len(df.index) + 1))
        angles = [element * width for element in indexes]

        bars = ax.bar(
            x=angles,
            height=heights,
```

```
        width=width,
        bottom=lowerLimit,
        linewidth=2,
        edgecolor="white")
    # Add labels
    for bar, angle, height, label in zip(bars, angles, heights, df["index"]):

        # Labels are rotated. Rotation must be specified in degrees :(
        rotation = np.rad2deg(angle)

        # Flip some labels upside down
        alignment = ""
        if angle >= np.pi / 2 and angle < 3 * np.pi / 2:
            alignment = "right"
            rotation = rotation + 180
        else:
            alignment = "left"

        # Finally add the labels
        ax.text(
            x=angle,
            y=lowerLimit + bar.get_height(),
            s=label,
            ha=alignment,
            va='center',
            rotation=rotation,
            rotation_mode="anchor")
    plt.show()
```

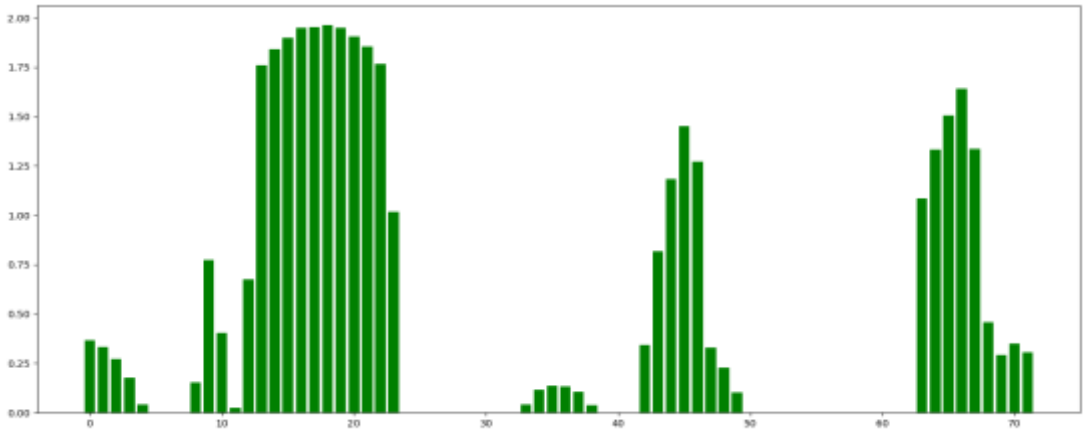
خروجی پلات های توابع بالا را در شکل های زیر میتوانید مشاهده نمایید.

۱. مقدار threshold = 2

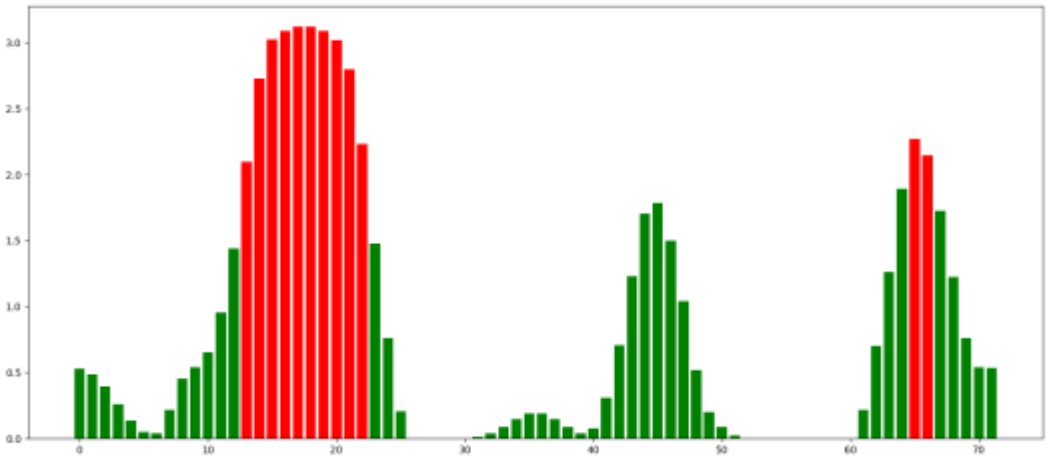
- ربات در نقطه (0,0)



موقعیت ربات در شبیه ساز

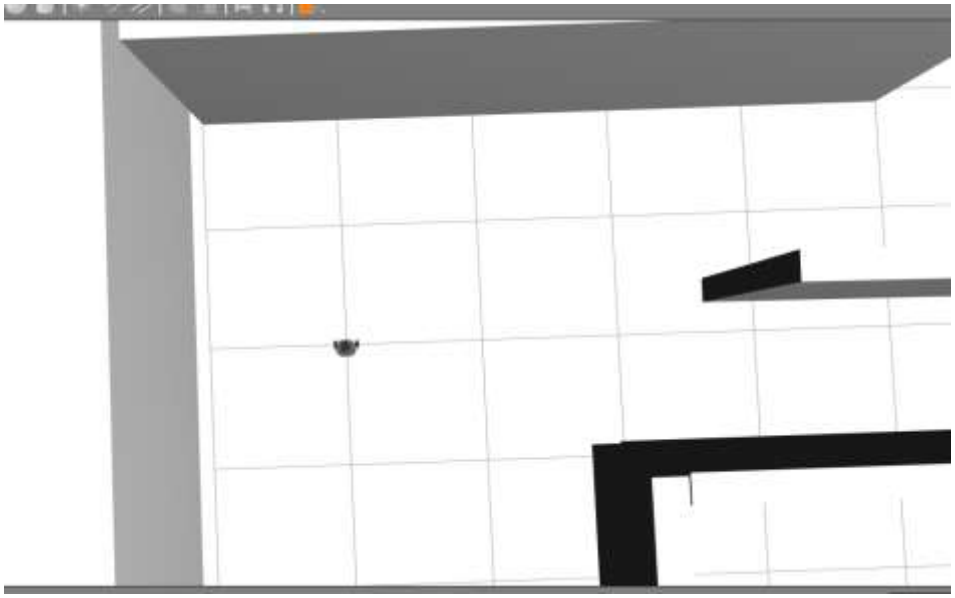


پلات **h**

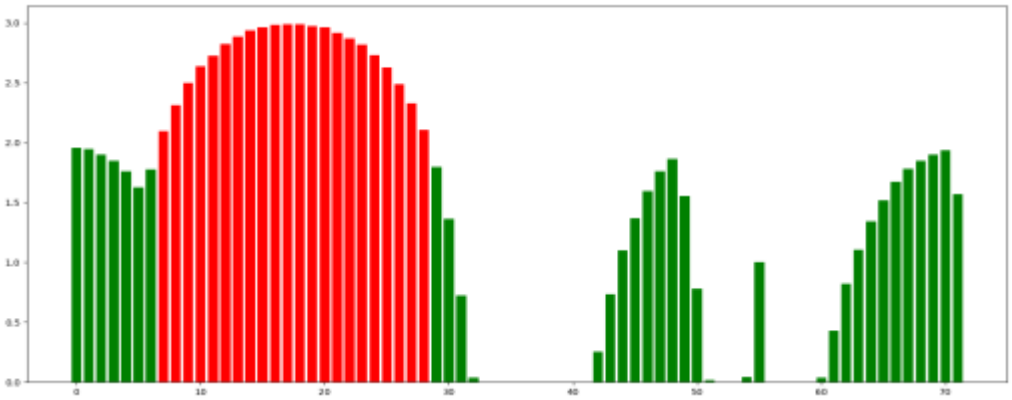


پلات **smothed_h**

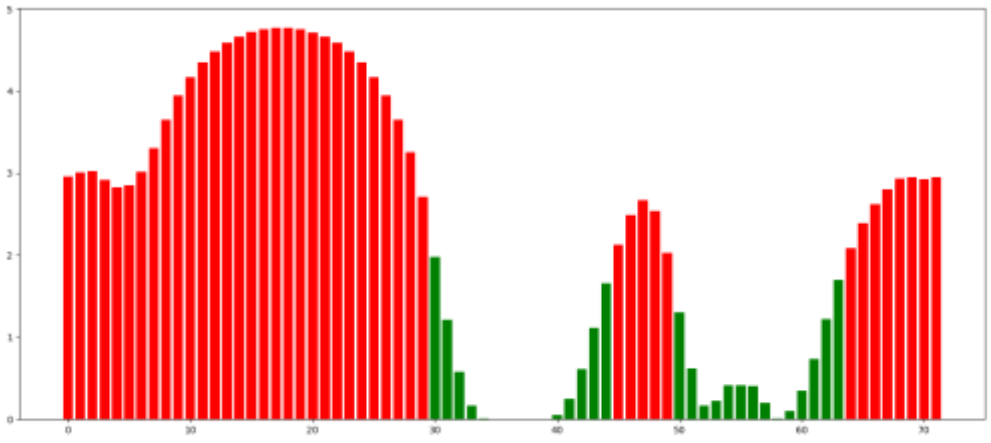
- ربات در نقطه (-9,3)



موقعیت ربات در شبیه ساز

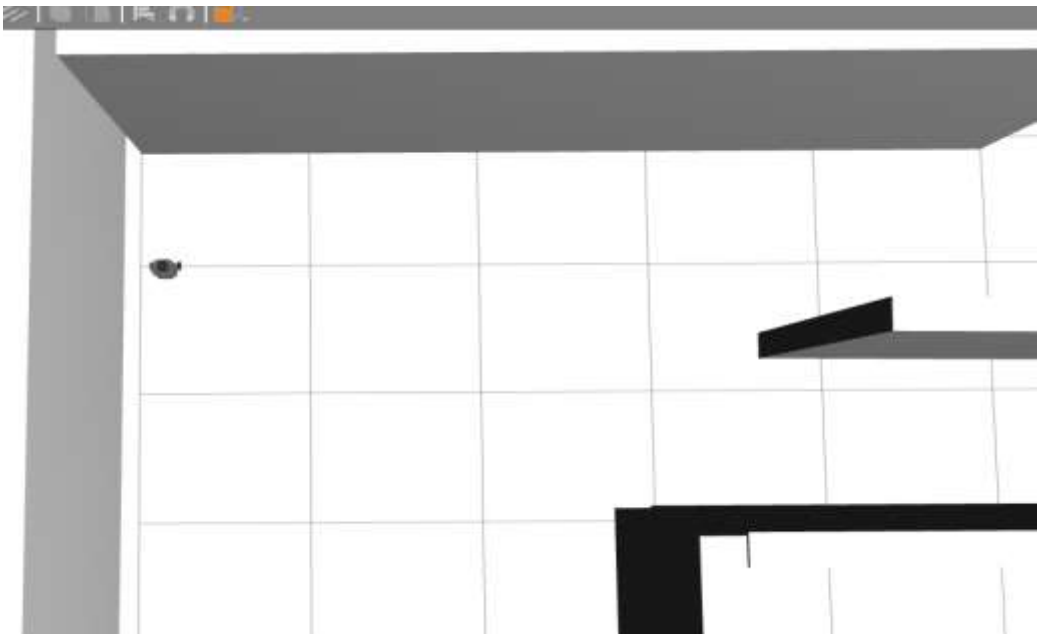


پلات **h**

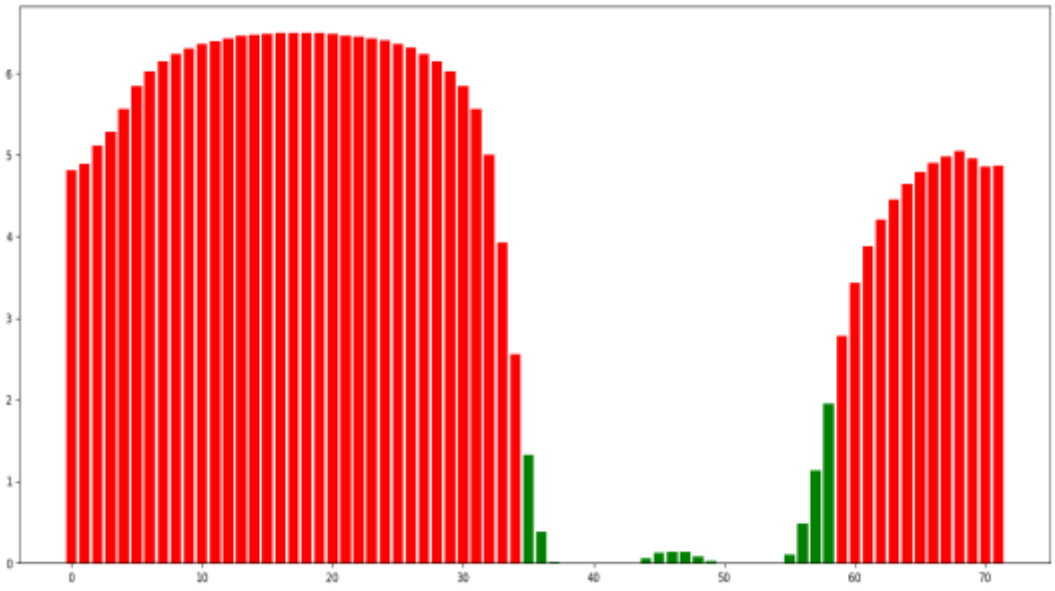


پلات **smothed h**

- ربات در نقطه (-9.85,4)



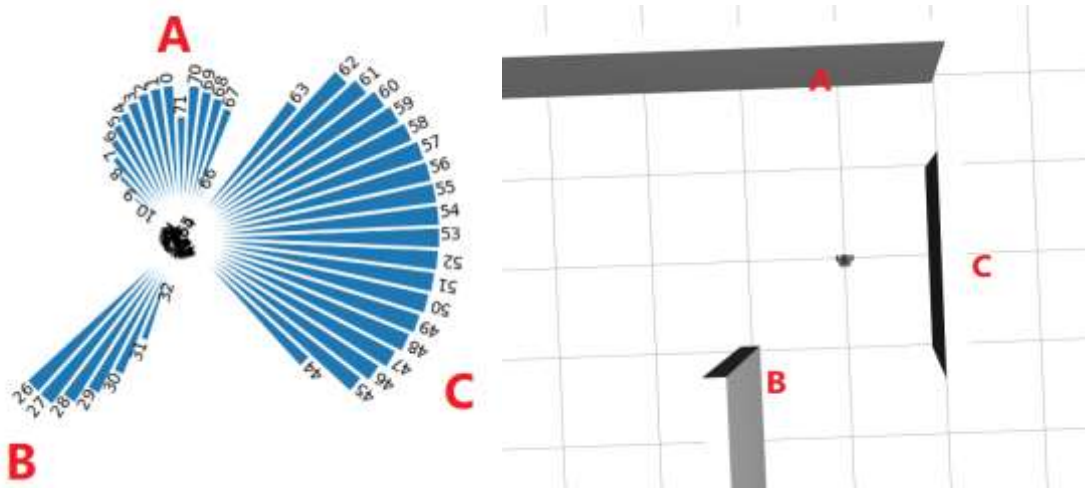
موقعیت ربات درون شبیه ساز

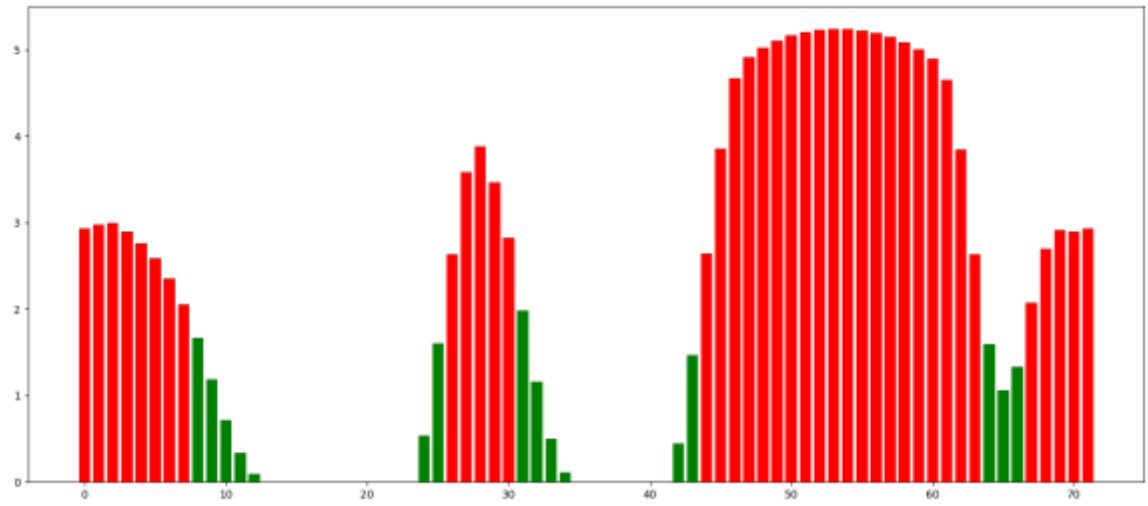


پلات **smothed pid**

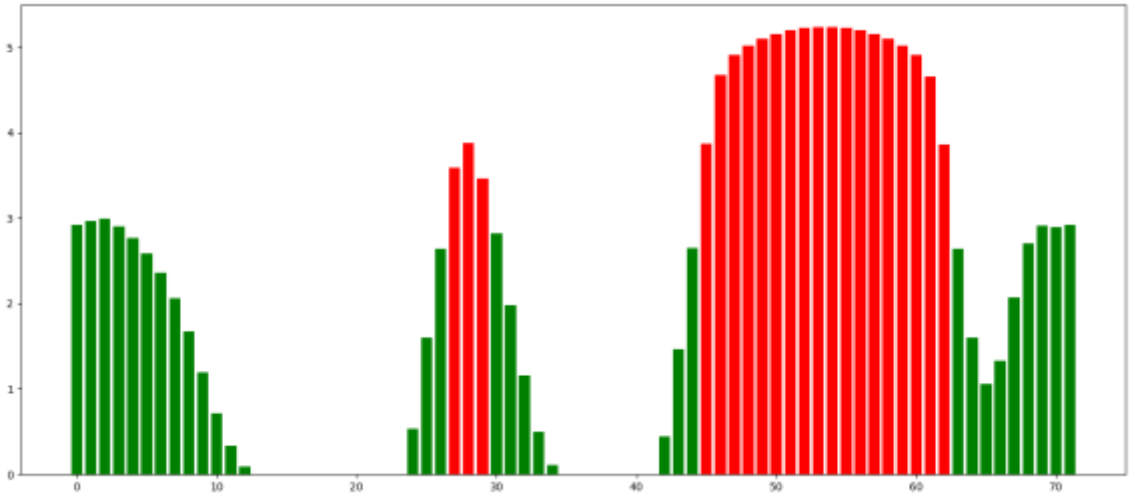
- ربات در نقطه (-3,0)

برای این نقطه پلات دایره ایی رسم شده در شکل زیر را مشاهده کنید. همانطور که مشخص است در اطراف ربات سه مانع A,B و C وجود دارد. پلات های دایره ایی و صاف از smothed_h آمده در تصاویر زیر نشان میدهد که توابعی که نوشته شده و در بالا توضیح داده شده است به درستی کار میکنند و میتوانند موانع را تشخیص دهند.





پس از پلات کردن مقادیر در موقعیت های متفاوت نقشه نتیجه گرفته شد که مقدار threshold برابر ۳ از باقی مقادیر مناسب تر عمل میکند. برای موقعیت بالا پلات smothed h را میتوانید در شکل زیر با threshold =3 مشاهده کنید.



فاز دوم: محاسبه زاویه حرکت یا theta

در این فاز از نتایج به دست آمده در فاز قبلی استفاده میکنیم و مقادیر قعر و قله را تشخیص میدهیم. سپس مقدار theta که بهترین زاویه برای حرکت کردن ربات است را به دست میآوریم.

برای به دست آوردن theta دو حالت وجود دارد. یا زاویه ایی که هدف با ربات دارد درون یکی از قعر ها که در پلات های بالا با رنگ سبز نشان داده شده است قرار گرفته که در این صورت theta برابر با اختلاف زاویه ربات با خود هدف میشود.

حالت دوم این است که زاویه ایی که هدف با ربات دارد درون یکی از قعر ها قرار ندارد. در این صورت باید نزدیک ترین قعر به هدف را پیدا کرده و سپس طبق فرمول kn+kf / 2 مقدار theta را محاسبه کنیم.

```
344         ## finding robot heading(in lidar sensor order) towards the target
345         angular_deffrences_with_target = self.target_angular_difference()
346         target_heading = self.heading_to_target(angular_deffrences_with_target)
347         target_sector = self.get_target_sector(target_heading)
348
349         ## finding theta using heading and smothed_h
350         valleys = self.valley_finder(smothed_h)
351         print(valleys)
352         theta_sector = self.find_theta(valleys, target_sector)
353         print("heading sector ", theta_sector)
354         theta = self.get_trigonometric_circle_theta(theta_sector)
355         print("heading theta ", theta)
```

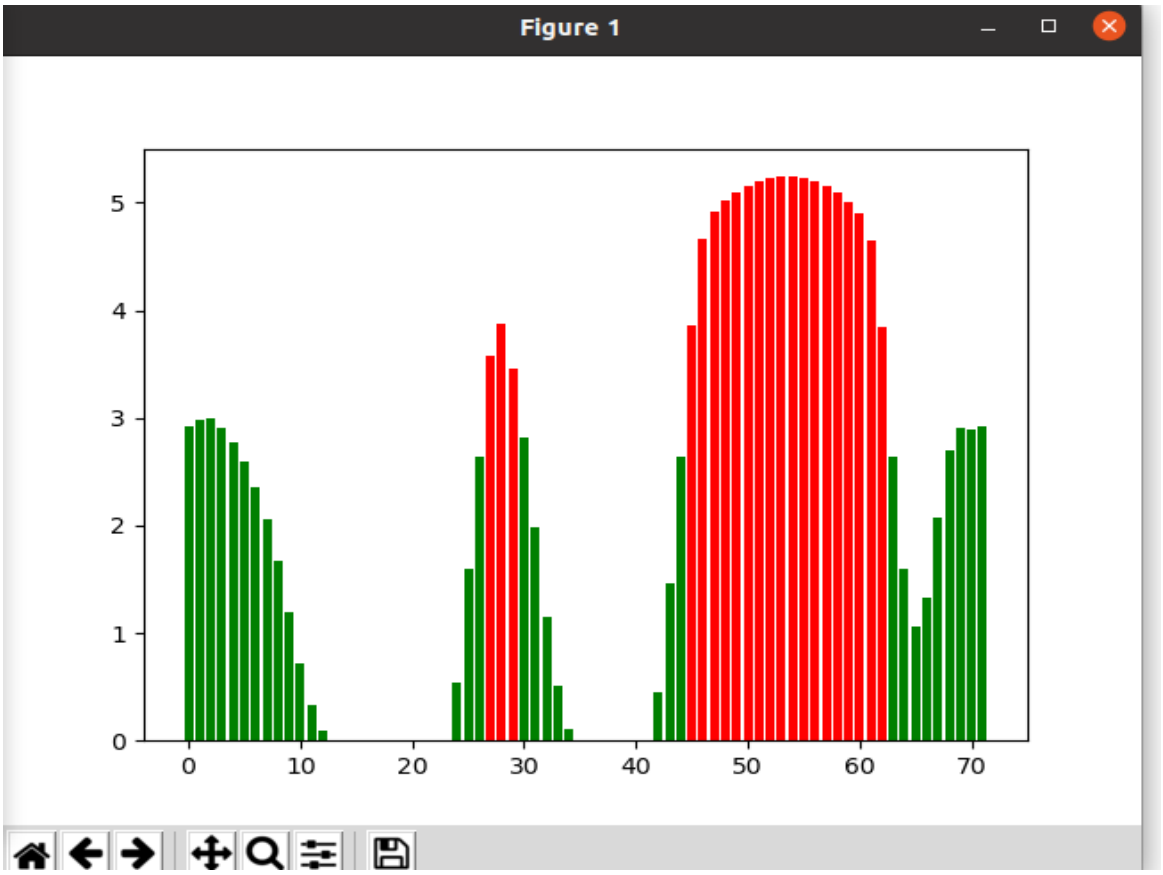
شکل بالا ترتیب صدا شدن توابع مربوط به فاز دوم در لوپ اصلی کد را نشان میدهد. در سه تابع اول ابتدا سکتوری که هدف درون آن قرار گرفته شده است محاسبه میشود. سپس با استفاده از تابع valley_finder که در شکل زیر آمده است دره هارا با استفاده از نتایج فاز اول محاسبه میکنیم.

```
    ## this method finds valleys according to VFH algorithm
    ## returns a list of vallyes with their sector numbers
    def valley_finder(self, smothed_h):
        rows, cols = (72, 2)
        arr = [[0 for i in range(cols)] for j in range(rows)]
        for i, v in enumerate(smothed_h):
            arr[i][0] = i
            arr[i][1] = v

        valleys = []
        flag = False
        counter = 0
        while True:
            new_valley = []
            if arr[counter][1] < self.threshold:
                new_valley.append(arr[counter])
                for j in range(counter + 1, len(arr)):
                    if arr[j][1] < self.threshold:
                        new_valley.append(arr[j])
                    else:
                        flag = True
                        break
                    flag = True
            if flag is True:
                counter = new_valley[-1][0] + 1
                valleys.append(new_valley)
                flag = False
```

```
            else:
                counter += 1
            if counter == len(arr):
                break

        valleys_sector_numbers = []
        for valley in valleys:
            new = []
            for v in valley:
                new.append(v[0])
            valleys_sector_numbers.append(new)

        return valleys_sector_numbers

    def circular_dist(self, lenght_of_list, idx_1, idx_2):
        i = (idx_1 - idx_2) % lenght_of_list
        j = (idx_2 - idx_1) % lenght_of_list
        return min(i, j)
```

خروجی این تابع لیستی از دره ها با شماره سکتور هایشان است. نمونه خروجی در شکل زیر آمده است.

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]
[30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44]
[63, 64, 65, 66, 67, 68, 69, 70, 71]
```



دو تابع بعدی که در خط های ۳۵۲ و ۳۵۴ صدا شده اند با اسفاده از شماره سکتور هدف و لیست دره ها مقدار theta را بر حسب زاویه دایره مثلثاتی محاسبه میکنند.

فاز سوم: حرکت دادن ربات

در این فاز با استفاده از نتیجه فاز دوم یا همان theta ربات را به سمت هدف با استفاده از کد زیر حرکت میدهیم. ربات ابدا به سمت هدف پرخیده و سپس به سمت آن حرکت میکند.

```
def move_to_target(self, theta):
    speed = Twist()
    distance_error = self.euclidean_distance((self.x, self.y), (self.target.x, self.target.y))
    if distance_error > 0.08:
        if abs(theta - self.yaw) > 0.15:
            print("rotating")
            speed.linear.x = 0
            speed.angular.z = -0.25
        else:
            print("moving to point")
            speed.linear.x = 0.2
            speed.angular.z = 0
    else:
        speed.linear.x = 0
        speed.angular.z = 0

    return speed

def run(self):
    rospy.sleep(2)
    twist = Twist()
```

نتایج، تصاویر و فیلم حرکت ربات را در مسی codes/scenario 2 میتوانید مشاهده نمایید.

سناریو سوم - بخش امتیازی: دنبال کردن دیوار با یادگیری تقویتی

مقدمه

یادگیری تقویتی یکی از سه عملکرد اصلی در مواجه با مسائل یادگیری ماشین میباشد. بر خلاف دو گونه دیگر یعنی یادگیری با نظارت و بدون نظارت داده‌ی از پیش تعیین شده‌ای در اختیار مدل که در اینجا به آن عامل گفته میشود قرار داده نمیشود. بلکه عامل باید در تعامل با محیط نحوه‌ی عملکرد آنرا بیاموزد. عامل به ازای هر عمل مجازی که انجام میدهد از محیط خود پاداشی را دریافت میکند. هدف عامل بیشینه کردن امید ریاضی پاداش‌های تجمعی در هر لحظه در زمان است و طبق ویژگی مارکوف برای پیشبینی آینده تنها حالت فعلی محیط برای عامل کافی میباشد.

چند تعریف: سیاست ، تابع ارزش حالت و تابع ارزش عمل

سیاست نشان‌دهنده‌ی توزیع اعمال در حالات مختلف است که عامل باید بر اساس آن عمل خود را انتخاب کند. تابع ارزش حالت بیانگر ارزش هر یک از حالت‌های محیط براساس امید ریاضی پاداش تجمعی دریافتی در آن حالت است ، و اگر به ازای اعمال گوناگون در حالت مختلف این مقدار میانگین را حساب کنیم به تابع ارزش عمل که آنرا با Q نمایش میدهیم، میرسیم.

ترکیب با شبکه‌های عصبی

همانطور که در قسمت قبل گفته شد باید برای ذخیره ساختن ارزش اعمال جدولی به اندازه تعداد حالات در تعداد اعمال نیاز بسازیم. این رویکرد برای محیط گسسته با اعمال گسسته کافی است اما برای محیط پیوسته اعمال پیوسته چطور؟ راه حل استفاده از شبکه‌های عصبی به عنوان ابزاری برای تخمین زدن در حالات پیوسته میباشد. ترکیب مفاهیم یادگیری تقویت با شبکه‌های عصبی ، یادگیری تقویتی عمیق را تولید میکند.

انواع روش‌های یادگیری تقویتی عمیق

چهار روش مختلف برای یادگیری تقویتی عمیق پیشنهاد شده است:

۱) یادگیری توابع ارزش

۲) یادگیری سیاست با استفاده از گرادیان سیاست

۳) روش عامل- نقاد (ترکیبی از دو روش قبل)

۴) روش مبتنی بر مدل

روش دوم یعنی گرادیان سیاست بهترین روش برای محیط پیوسته با اعمال پیوسته(کنترل بهینه) میباشد. اما ما در این پروژه که قصد داریم وظیفه دنبال کردن دیوار را پیاده سازی کنیم برای ساده سازی از روش اول استفاده کرده‌ایم.

یادگیری-کیو

رایج‌ترین روش یادگیری توابع ارزش است که بر خلاف روش‌های مونتی-کارلو(یادگیری آفلاین) سعی در یادگیری در حین انجام عمل دارد(یادگیری آنلاین). روش کار به این صورت است که برای انتخاب عمل با استفاده از یک سیاست دلخواه مثلا اپسیلون-حریصانه بهینه ترین عمل را انتخاب کرده و بر اساس سیاستی دیگر آنرا بهبود میبخشیم(انتخاب بیشترین ارزش عمل در حالت بعدی) به چنین روش هایی که انتخاب عمل بر اساس یک سیاست و ارزیابی بر اساس سیاستی دیگر است off-policy گفته میشود.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

یادگیری کیو این تضمین را میکند که در تعداد تکرار بالا سیاست بهینه را بدست آورد.

یادگیری-کیو عمیق

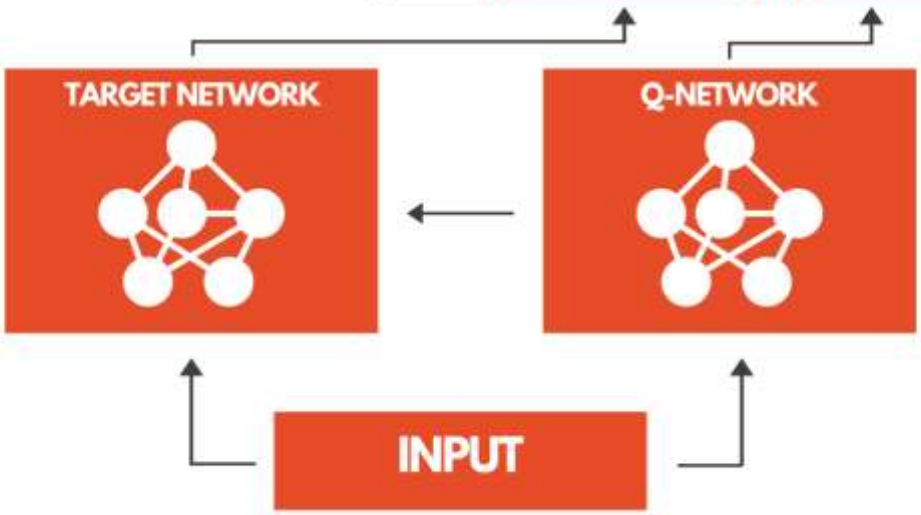
حال اگر ما مقادیر کیو را با استفاده از شبکه‌های عصبی تخمین بزنیم شبکه‌ای به نام شبکه‌ی کیو-عمق را ساخته ایم.

الگوریتم آن به شکل زیر است

initialize replay memory D
initialize action-value function Q with random weight $\boldsymbol{\theta}$
initialize target action-value function $\hat{Q}$ with weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
for *episode = 1 to M* **do**
  initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for *t = 1 to T* **do**
    following ϵ-greedy policy, select $a_t = \begin{cases} \text{a random action} & \text{with probability } \epsilon \\ \arg\max_a Q(\phi(s_t), a; \boldsymbol{\theta}) & \text{otherwise} \end{cases}$
    execute action a_i 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
    // experience replay
    sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
    perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \boldsymbol{\theta}))^2$ w.r.t. the network parameter $\boldsymbol{\theta}$
    // periodic update of target network
    in every C steps, reset $\hat{Q} = Q$, i.e., set $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
  end
end

توضیح: عامل باید برای آموزش شبکه از محیط داده جمع کند. در ابتدای کار که داده‌ای وجود ندارد حرکات شانسی و رندوم هستند و داده ها در بافری به اسم Experience Replay ذخیره و بازیابی میشوند و تخمین مقادیر ارزش کیو مانند یادگیری بانظارت خواهد بود اما نکته‌ی مهمی که وجود دارد این است که در یادگیری بانظارت توزیع داده‌ها یکنواخت و مستقل از هم میباشند(به جز سری های زمانی) این در حالی است که در یادگیری تقویتی ما دنباله‌ای از اعمال و حالات مرتبط به هم را ذخیره کردیم پس برای مستقل کردن آنها باید به صورت تصادفی نمونه برداری شوند که راه پیشنهادی مینی-بچ میباشد. نکته‌ی دیگر این است این الگوریتم از ۲ شبکه عصبی استفاده میکند که یکی c قدم از دیگری جلوتر و راهنمای آن برای رسیدن به نقطه‌ی بهینه است و براساس آن آموزش میبینند مانند آنچه در شکل زیر مشخص است.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$



که در آن آلفا اندازه قدم(ضریب یادگیری) و گاما ضریب کاهش است.

صحت الگوریتم پیاده‌سازی شده در این پروژه

از آنجایی که آموزش مدل در Gazebo بسیار زمان بر است ما ابتدا الگوریتم‌های خود را در محیط OpenAI GYM بر روی وظیفه Cart-Pole که وظیفه کنترلی مشابه با دنبال کردن دیوار است(منظور کنترل مقادیر و نه کلیت وظیفه) امتحان کرده و فیلم آن در پیوست قرار دارد.

دنبال کردن دیوار با یادگیری تقویتی

لازم به ذکر است برای نیل به هدف این آزمایش تعداد دفعات فرسایشی عامل در محیط بوده است و یادگیری انجام داده است چیزی بالغ بر بیش از ۱۵۰۰ تکرار!!

حال الگوریتم‌های تایید شده‌ی خود را به محیط gazebo میبریم. کلیت ماجرا هیچ فرقی ندارد و تنها باید حالات و اعمال و جوایز(تنبیه) ها مشخص شود.

ورودی شبکه‌ی عصبی که همان حالت محیط است در این مسئله کمترین فاصله روبات از دیوار است.

خروجی شبکه که به تعداد اعمال است نشان‌دهندی ارزش هر عمل است. همچنین اعمال به شکل زیر تعریف میشوند که سرعت زاویه‌ای ربات است.

```
self.action  =  [-1.,  -0.7,  -0.4,  -0.25,  -0.1  ,  -0.06,  -0.02,  0.0,
                                    0.02,  0.06,  0.1, 0.25, 0.4, 0.7, 1.]
```

سیستم پاداش دهی:

فاصله‌ی مورد نظر ما از دیوار مقدار ۱,۳ است حال اگر به دیوار نزدیک یا دور شود به همین میزان اختلاف تنبیه میشود. یعنی اختلاف فاصله از دیوار و ۱,۳ (برای اطلاعات بیشتر تابع get_reward را بررسی کنید)

لازم به ذکر است که ربات در هر مرحله اگر بیش از یک مقدار مجاز (در اینجا 4.) خطا داشته باشد باید به حالت اولیه‌ی خود برگرد و یادگیری را از ابتدا شروع کند. این کار با تابع check_done و reset انجام میشود

```
self.start_pose = (2, 7.0, -1.45)
```

تابع فراخوان یادگیری تابع run میباشد که درآن الگوریتم dqn اجرا میشود.

همچنین ویدیو های مربوط به این قسمت پیوست شده است