به نام خدا

موضوع پروژه:

Car Racing

منتورهای پروژه:

محمدجواد میرشکاری حقیقی

امین چینی‌فروشان

اعضای گروه:

سینا علی‌نژاد (۹۹۵۲۱۴۶۹)

نوید ابراهیمی (۹۹۵۲۱۰۰۱)

بهمن ماه ۱۴۰۱

ورژن صفر:

توضیح کلی:

ورژن صفر به این صورت عمل میکند که برای تمام فریم‌های بازی باید تجربه کسب کند. در واقع stateهای ما در این حالت ترکیب رنگ‌های پیکسل‌های هر فریم میباشند و از آنجا که این تعداد بسیار زیاد است، یادگیری آن به راحتی صورت نمیگیرد و در عمل خوب کار نمیکند زیرا به هر stateیی که میرود یک state جدید است و به stateهای قبلی تعمیم نمیدهد و از generalization استفاده نمیکند.

در ادامه جزئیات ورژن صفر را بررسی میکنیم:

* برای محدود کردن و گسسته سازی legal action ۳ متغیر زیر را تعریف کردیم.

۱- Steer_action_num: تعداد حالات جهت فرمان را نشان میدهد. در سوال ما این مقدار را ۱۰ عدد قرار دادیم. یعنی فرمان میتواند ۱۰ مقدار از مقادیر بین -۱ تا ۱ را کسب کند.

۲- GAS_ACTION_NUM: تعداد حالات مجاز برای گاز را نشان میدهد. چون بازه مجاز بین ۰ تا ۱ است، با دیفالت ۵ که ما گذاشتیم صرفا ۵ مقدار را میتواند بگیرد.

۳- BRAKE_ACTION_NUM: تعداد حالات مجاز برای ترمز را نشان میدهد. چون بازه مجاز بین ۰ تا ۱ است، با دیفالت ۵ که ما گذاشتیم صرفا ۵ مقدار را میتواند بگیرد.

* init: مقادیر نام مدل، epsilon، gamma، learning rate، k و memory را برای استفاده مشترک توابع ذخیره میکند.

* f: برای اینکه actionهایی که نتایج مناسبی نداشته دوباره انتخاب نشوند این تابع را تعریف کرده‌ایم. فرمولش هم بر اساس u + k/n است که به صورت مشابه در کد پیاده سازی شده است.

```
def f(self, action, state):
    return self.memory[state][0][action][0] + self.k / self.memory[state][0][action][1]
```

تصویر ۱ – تابع f در ورژن صفر

* تابع explore: وقتی میخواهیم actionیی انجام دهیم، با احتمال epsilon یک اکشن random انجام میدهیم. این تابع همین وظیفه را برعهده دارد. به این صورت که در ابتدا ۳مقدار جهت فرمان، مقدار گاز و ترمز را بین مقدارهای مجاز تعیین، و سپس آن را برای جهت فرمان بین -۱ تا ۱ و برای دو مورد دیگر بین ۰ تا ۱ تبدیل میکند. در ادامه اگر state که از ورودی آمده در memory نباشد، آرایه آن را ایجاد میکند. حال اگر این action در آن state نباشند، مقادیر را مطابق با کد در مموری آپدیت میکند.

```
def explore(self, state):
    random_steer_action = random.randint(-self.STEER_ACTIONS_NUM/2, self.STEER_ACTIONS_NUM/2 + 1)
    random_gas_action = random.randint(0, self.GAS_ACTIONS_NUM + 1)
    random_brake_action = random.randint(0, self.BRAKE_ACTIONS_NUM + 1)
    random_steer_action = (180 / self.STEER_ACTIONS_NUM) * random_steer_action / 90
    random_gas_action = 1 / self.GAS_ACTIONS_NUM * random_gas_action
    random_brake_action = 1 / self.BRAKE_ACTIONS_NUM * random_brake_action
    if state not in self.memory:
        self.memory[state] = [None, None]
        self.memory[state][0] = {}
    if (random_steer_action, random_gas_action, random_brake_action) not in self.memory[state][0]:
        self.memory[state][0][(random_steer_action, random_gas_action, random_brake_action)] = [0, 0]
        self.memory[state][1] = (random_steer_action, random_gas_action, random_brake_action)
    return random_steer_action, random_gas_action, random_brake_action
```

تصویر۲ – تابع explore در ورژن صفر

* تابع exploid: اگر مقدار random ما از epsilon در تابع get_action بیشتر باشد، این تابع صدا زده میشود. نحوه کار به این صورت است که اگر state در مموری موجود باشد، action مربوطه را برمیگرداند و در غیر این صورت یک random action ساخته و برگردانده میشود. در واقع قسمت موجود بودن action همان حالتی است که ما قبلا این state را یادگرفتهایم و در این لحظه فقط نتیجه را برمیگردانیم.

```
def exploid(self, state):
    if state in self.memory:
        return self.memory[state][1]
    else:
        return self.explore(state)
```

تصویر۳ – تابع exploid در ورژن صفر

* تابع update_best_action: این تابع برای آپدیت کردن مقادیر بعد از انجام action صدا زده میشود. به این صورت که ابتدا بررسی میکند state بعدی که در آن رفتهایم در queue table یا همان مموری باشد. اگر

نبود آن را اضافه میکند. سپس با استفاده از فرمول آپدیت کردن Q، مقدار Q را برای current state تغییر میدهد و اگر این مقدار از best action بهتر باشد، best action را آپدیت میکنیم که این مقایسه توسط تابع f انجام میشود.

```
def update_best_action(self, current_state, action, next_state, reward):
    if next_state not in self.memory:
        self.memory[next_state] = [None, None]
        self.memory[next_state][0] = {}

    next_state_best_q = 0
    if self.memory[next_state][1] is not None:
        next_state_best_action = self.memory[next_state][1]
        next_state_best_q = self.memory[next_state][0][next_state_best_action]
    new_q = reward + self.gamma * next_state_best_q
    self.memory[current_state][0][action][0] = new_q
    self.memory[current_state][0][action][1] += 1
    if self.f(action, current_state) > self.f(self.memory[current_state][1], current_state):
        self.memory[current_state][1] = action

    self.normalize()
```

تصویر۴ – تابع update_best_action در ورژن صفر

* تابع get_Action: به این صورت پیاده سازی شده که اگر مقدار random ما کمتر از epsilon باشد، explore صدا زده میشود و در غیر این صورت exploid. منطقش هم بر این اساس است که هم از تجربیات قبلی و هم از action های رندم و جدید در یادگیری استفاده شود.

```
def get_action(self, state):
    if random.random() < self.epsilon:
        return self.explore(state)
    else:
        return self.exploid(state)
```

تصویر۵ – تابع get_action در ورژن صفر

* تابع normalize: به منظور آپدیت کردن مقادیر epsilon و learning rate استفاده میشود. به این صورت که این آپدیت کردن کاملا تجربی است و تا حد مشخصی انجام میشود.

```
def normalize(self):
    if(self.epsilon > 0.05):
        self.epsilon -= 0.001

    if(self.learning_rate > 0.05):
        self.learning_rate -= 0.001
```

تصویر۶ – تابع normalize در ورژن صفر

ورژن یک:

توضیح کلی:

در ورژن صفر خانه‌ها را تعمیم نمیدادیم و همین مورد باعث طولانی و سخت شدن learning مدل ما میشد و اطلاعاتی که از یک state به دست می‌آمد به نوعی هدر میرفت. پس در ورژن ۱ تصمیم گرفتیم stateهای مشابه روی یکدیگر تاثیر داشته باشند یعنی وقتی در پیچ‌های مختلف قرار داریم، از نظر state یکسان باشند و در واقع stateهای ما از روی ویژگی‌هایی مثل مکان ماشین، خطوط جاده و موقعیت پیچ‌ها به دست بیاید. در این حالت تعداد stateها به مراتب کمتر از ورژن صفر خواهد بود. در این روش ما از template matching، Line detection و الگوریتم canny استفاده کرده‌ایم.

در توضیح توابع زیر، توابعی شرح داده میشوند که با ورژن صفر مشترک نیستند.

* تابع color_difference: این تابع مجموع مقدار تفاضل ۲ rgb را برمیگرداند.

```
def color_difference(self, color1, color2):
    return abs(color1[0] - color2[0]) + abs(color1[1] - color2[1]) + abs(color1[2] - color2[2])
```

تصویر۷ – تابع color_difference در ورژن یک

* تابع get_line_equation: وظیفه این تابع محاسبه ضرایب خط یک خط با استفاده از فرمول‌هایی ریاضی‌ست.

```
def get_line_equation(self, line):
    x1, y1, x2, y2 = line[0]
    b = 1
    a = (y2 - y1) / (x1 - x2)
    c = -y1 - a * x1
    return a, b, c
```

تصویر۸ – تابع get_line_equation در ورژن یک

* تابع explore: وقتی میخواهیم actionیی انجام دهیم، با احتمال epsilon یک اکشن random انجام میدهیم. این تابع همین وظیفه را برعهده دارد. به این صورت که در ابتدا ۳مقدار جهت فرمان، مقدار گاز و ترمز را بین مقدارهای مجاز تعیین، و سپس آن را برای جهت فرمان بین -۱ تا ۱ و برای دو مورد دیگر بین ۰ تا ۱ تبدیل میکند. حال اگر این action در آن state نباشند، مقادیر را مطابق با کد در ممووری آپدیت میکند.

```
def explore(self, state, data):
    random_steer_action = random.randint(-self.STEER_ACTIONS_NUM/2, self.STEER_ACTIONS_NUM/2 + 1)
    random_gas_action = random.randint(0, self.GAS_ACTIONS_NUM + 1)
    random_brake_action = random.randint(0, self.BRAKE_ACTIONS_NUM + 1)
    random_steer_action1 = (180 / self.STEER_ACTIONS_NUM) * random_steer_action / 90
    random_gas_action1 = 1 / self.GAS_ACTIONS_NUM * random_gas_action
    random_brake_action1 = 1 / self.BRAKE_ACTIONS_NUM * random_brake_action
    feature_state = self.specify_state(data)
    if (random_steer_action, random_gas_action, random_brake_action) not in self.memory[feature_state][0]:
        self.memory[feature_state][0][(random_steer_action, random_gas_action, random_brake_action)] = [0, 0]
        self.memory[feature_state][1] = (random_steer_action, random_gas_action, random_brake_action)
    return (random_steer_action1, random_gas_action1, random_brake_action1), (random_steer_action, random_gas_action, random_brake_action)
```

تصویر۹ – تابع explore در ورژن یک

* تابع exploid: در این تابع میخواهیم طبق چیزهایی که از قبل یادگرفته‌ایم عمل کنیم. ابتدا مشخص میکنیم stateیی که در آن قرار داریم از نظر شاخصه‌هایی که در نظر گرفتیم چه stateیی است. سپس بررسی میکنیم که آیا best action برای این state تاکنون set شده است یا نه. اگر set نشده بود تابع explore را صدا میزنیم. در غیر این صورت همان best action را برمیگردانیم.

```python
def exploid(self, state, data):
    feature_state = self.specify_state(data)
    if self.memory[feature_state][1] is None:
        return self.explore(state, data)
    best = self.memory[feature_state][1]
    steer_action = (180 / self.STEER_ACTIONS_NUM) * best[0] / 90
    gas_action = 1 / self.GAS_ACTIONS_NUM * best[1]
    brake_action = 1 / self.BRAKE_ACTIONS_NUM * best[2]
    return (steer_action, gas_action, brake_action), best
```

تصویر ۱۰ – تابع exploid در ورژن یک

* تابع update_best_action: در این تابع current state و next state را توسط تابع specify_state و با استفاده از شاخص‌های خودمان در یک دسته‌ای قرار میدهیم. حال از فرمول آپدیت کردن Q برای current state استفاده میکنیم و این مقدار را برای current state موجود در queue table آپدیت میکنیم. سپس اگر این new action بهتر از best action برای current state باشد، مقدار best action را برای current state آپدیت میکنیم و این مقایسه actionها با استفاده از تابع f انجام میشود.

```python
def update_best_action(self, current_state, action, next_state, reward, curr_data, next_data):
    next_state_best_q = 0
    feature_next_state = self.specify_state(next_data)
    feature_current_state = self.specify_state(curr_data)
    if self.memory[feature_next_state][1] is not None:
        next_state_best_action = self.memory[feature_next_state][1]
        next_state_best_q = self.memory[feature_next_state][0][next_state_best_action][0]
    new_q = reward + self.gamma * next_state_best_q
    self.memory[feature_current_state][0][action][0] = new_q
    self.memory[feature_current_state][0][action][1] += 1
    if self.f(action, feature_current_state) > self.f(self.memory[feature_current_state][1], feature_current_state):
        self.memory[feature_current_state][1] = action

    self.normalize()
```

تصویر ۱۱ – تابع update_best_action در ورژن یک

* تابع specify_state: در این تابع میخواهیم stateهایی که از ترکیب رنگ پیکسل‌ها تشکیل شده است را به stateیی براساس ویژگی‌های خودمان تبدیل کنیم. این ویژگی‌ها عبارتند از:

۱- موقعیت ماشین نسبت به خطوط کناری جاده: عددی بین -۱ و ۱ است که ۱ به معنای وسط جاده بودن و هرچه به صفر نزدیک شود، به خطوط کناری جاده نزدیک‌تر می‌شود. عدد منفی به معنای بیرون جاده بودن است.

۲- موقعیت نسبت به پیچ: اگر درون پیچ باشیم، این مقدار برابر است با فاصله ماشین از پیچ که اگر ماشین درون جاده باشد، عددی مثبت و اگر بیرون از جاده باشد، عددی منفی می‌شود. در حالتی که ماشین در وسط جاده باشد مقدارش برابر با ۱ میشود.

۳- سرعت متوسط ماشین: این سرعت متوسط را بر حسب فاصله طی شده تقسیم بر تعداد فریم به دست می‌آوریم. به دست آوردن مقدار فاصله با استفاده از مختصات دونقطه‌ای که ماشین در آن قرار دارد با الگوریتم template matching انجام داده‌ایم.

سپس با استفاده از مقادیر به دست آمده برای متغیرهای ذکر شده، بررسی میکنیم که این ۳متغیر در کدام محدوده قرار دارند و دسته مربوطه را برمیگردانیم.

```
def specify_state(self, data):
    car_loc, lines_loc, left_loc, right_loc, avg_v = data
    a1,b1,c1,a2,b2,c2 = 0,0,0,0,0,0
    if lines_loc is not None:
      if len(lines_loc) > 2:
        a1,b1,c1 = self.get_line_equation(lines_loc[0])
        a2,b2,c2 = self.get_line_equation(lines_loc[1])
    try:
      dis1 = abs(a1*car_loc[0]+b1*car_loc[1]+c1)/math.sqrt(a1**2+b1**2)
      dis2 = abs(a2*car_loc[0]+b2*car_loc[1]+c2)/math.sqrt(a2**2+b2**2)
    except:
      dis1, dis2 = 0,0
    if dis1 + dis2 < self.ROAD_WIDTH:
      pos_f = 1 - abs(dis1-dis2)/self.ROAD_WIDTH
    else:
      dis_out = min(dis1,dis2)
      pos_f = -(dis_out)/self.ROAD_WIDTH
      if pos_f < -1:
        pos_f = -1
    curve_f = 1
    if self.is_turn_left(car_loc, left_loc) or self.is_turn_right(car_loc, right_loc):
      curve_f = pos_f
    velocity_f = avg_v/self.MAX_VELOCITY
    for cat in self.states:
      if pos_f >= cat[0][0] and pos_f < cat[0][1] and curve_f >= cat[1][0] and curve_f < cat[1][1] and velocity_f >= cat[2][0] and velocity_f < cat[2][1]:
        return cat
    return self.states[0]
```

تصویر۱۲ – تابع specify_state در ورژن یک

* تابع is_turn_left: با استفاده از الگوریتم template matching مختصات پیچ به سمت چپ را شناسایی کردیم. حال با مقایسه آن با مختصات ماشین، میبینم که ماشین در پیچ هست یا نه.

```
def is_turn_left(self, car_loc, left_loc):
  x1, y1 = car_loc
  x2, y2 = left_loc
  if abs(x1 - x2) < 0.1 and abs(y1 - y2) < 0.1:
    return True
  return False
```

تصویر۱۳ – تابع is_turn_left در ورژن یک

* تابع is_turn_right: با استفاده از الگوریتم template matching مختصات پیچ به سمت راست را شناسایی کردیم. حال با مقایسه آن با مختصات ماشین، میبینم که ماشین در پیچ هست یا نه.

```python
def is_turn_right(self, car_loc, right_loc):
    x1, y1 = car_loc
    x2, y2 = right_loc
    if abs(x1 - x2) < 0.1 and abs(y1 - y2) < 0.1:
        return True
    return False
```

تصویر۱۴ – تابع is_turn_right در ورژن یک

فایل LineDetection: وظیفه این فایل شناسایی خطوط در محیط بازی است. ورودی تابع موجود در این فایل محیط بازی (stateیی که در آن هستیم) است که در ابتدا آن را خاکستری کرده و سپس با استفاده از canny و HoughLinesP خطوط را شناسایی میکند.

فایل TemplateMatching: به این صورت است که برای شناسایی template در یک محیط استفاده میشود. در این فایل سه‌حالت ماشین، پیچ چپ و پیچ راست را تعریف کردیم و برای هرکدام از الگوریتم خاصی بهره برده‌ایم. دلیل اینکار به طور مثال در پیچ راست و چپ این است که تعدای از الگوریتم‌ها پیچ‌ها را با template خاص به طور یکسان تشخیص میدادند ولی این دو الگوریتم بین این دو حالت تفاوت قائل بودند.