شرح فاز سوم پروژه ی هوش مصنوعی. علی نوروزی 9831067

بخش اول: Value Iteration

در این بخش چندین تابع را پیاده سازی کرده ایم که در ادامه به توضیح پیاده سازی آنها می‌پردازیم.

computeActionFromValues: این تابع بر روی Action های ممکن در استیت فعلی پیمایش میکند و با استفاده از تابعی که مقدار Q value را محاسبه میکند، اکشنی که ماکزیمم Q value را دارد، برمیگرداند.

computeQValueFromValues: این تابع اکشن و استیت را دریافت میکند و بر اساس آن مقدار Q value را بدست می‌آورد. در این تابع با استفاده از تابع getTransitionStatesAndProbs که یک آرایه ای از تاپل (s_prime, transition) میباشد و s_prime استیت بعدی و transition مقدار احتمال برای ورود به استیت بعدی میباشد، مقدار Q value را بدست می آوریم.

runValueIteration: این تابع یک سری تکرار[1] را انجام میدهد. در ابتدای هر تکرار یک دیکشنری خالی را میسازیم و مقادیر value ها را در تکرار آپدیت میکنیم. پس از تمام شدن هر تکرار، مقادیر value که در این تکرار بدست آوردیم را برابر با مقادیر اصلی value قرار میدهیم. در هر تکرار بر روی استیت ها پیمایش میکنیم و ماکزیمم Q value را بدست می آوریم و مقادیر iteration_values را آپدیت میکنیم.

maxQvalue: یک تابع کمکی نیز تعریف کرده ام که به وسیله ی آن میتوان مقدار value هر استیت که برابر با ماکزیمم مقدار Q valueهای آن استیت می‌باشد را بدست آورد.

بخش دوم: عامل نویزی

با قرار دادن مقدار نویز برابر با 0.01 سیاست بهینه موجب عبور عامل از پل میشود. در این حالت تقریبا از هر صد اکشن یک اکشن نویزی وجود دارد که از استیت فعلی به یک استیت ناخواسته ی جدید برود.

[1] Iteration

بخش سوم: سیاست ها

در ابتدا نقش این سه مورد را در اینکه عامل خروجی نزدیک را ترجیح میدهد یا دور، از صخره اجتناب میکند یا خیر بررسی میکنیم.

هر چه مقدار تخفیف یا Discount بیشتر باشد، میزان آینده نگری عامل ما بیشتر میشود. چرا که طبق معادله ی Q value هر چه مقدار تخفیف به عدد یک نزدیکتر شود، تاثیر Q value استیت های بعدی در مقدار Value فعلی بیشتر میشود. از طرف دیگر هرچه مقدار نویز بیشتر باشد، عامل برای اینکه ریسک کمتری کند به صخره ها نزدیک نمیشود. و برعکس. و در نهایت هرچه living reward بیشتر باشد، امکان اینکه عامل به ترمینال نرسد بیشتر است چرا که در حال گشتن در grid میباشد و پاداش نیز دریافت میکند😊

در قسمت اول برای اینکه خروجی نزدیک را ترجیح دهد مقدار تخفیف را 0.5 قرار دادیم و همچنین مقدار living reward منفی 2 قرار دادیم تا حالت ترمینال سریعتر را انتخاب کند. و از طرف دیگر مقدار نویز را عدد کوچکی قرار دادیم تا خطر صخره ها را ریسک کند.

در قسمت دوم بر خلاف قسمت الف مقدار نویز را بیشتر گذاشتیم تا از صخره ها دوری کند.

در قسمت سوم برای اینکه خطر صخره را ریسک کند مقدار نویز را کم گذاشتیم(0.1) و همچنین برای اینکه خروجی دور تر را ترجیح دهد مقدار تخفیف را برابر با عدد بزرگ قرار دادیم(0.9)

در قسمت چهارم برای اینکه از صخره دور شود مقدار نویز را بزرگ تر گذاشتیم. مقدار تخفیف همانند قسمت سوم میباشد. همچنین مقدار living reward را برابر با صفر قرار دادیم.

در قسمت پنجم تنها چیزی که اهمیت دارد living reward میباشد. مقدار living reward برابر با عدد بزرگ مثلا 10 قرار داده شده. و دو مقدار دیگر صفر هستند. این living reward بزرگ موجب میشود که عامل در حال گشتن در grid باشد چرا که در حال گشتن مقدار پاداش بزرگی دریافت میکند و احتمال رسیدن به خانه ی ترمینال کاهش می‌یابد.

بخش چهارم: تکرار ارزش ناهمزمان

در این بخش تابع runValueIteration را پیاده سازی کرده ایم. در این تابع بر روی iteration پیمایش میشود و با استفاده از indexing، هر سری یک استیت را انتخاب میکنیم و در صورتی که آن استیت ترمینال نبود، مقدار Q value آنرا آپدیت میکنیم.

بخش پنجم: تکرار ارزش اولویت بندی شده

در این بخش طبق الگوریتم گفته شده در دستور کار برای پیاده سازی تابع runValueIteration عمل کرده ایم.

در ابتدا یک دیکشنری خالی تعریف میکنیم. از این دیکشنری برا نگهداری حالت های پسین هر استیت استفاده میکنیم.

سپس یک حلقه بر روی تمام استیت ها میزنیم و مقادیر مربوط به هر استیت را برابر با یک set خالی قرار میدهیم. دلیل استفاده از set، جلوگیری از تکراری بودن حالت های پسین هر استیت میباشد.

حال مجدد بر روی استیت ها پیمایش انجام میگیرد و حالت های پسین استیت ها در دیکشنری تعریف شده قرار میگیرد. بدین صورت که در هر استیت اکشن های موجود به دست آورده میشود و سپس با استفاده از تابع getTransitionStatesAndProbs حالت های پسین هر استیت بدست آورده میشود و states_predecessors آپدیت میشود.

حال یک صف اولویت خالی را میسازیم. بر روی استیت ها پیمایش میکنیم و در صورتی که استیت ترمینال نبود، طبق دستور کار پروژه مقدار diff را بدست می آوریم و استیت فعلی را با اولویت منفی diff وارد صف میکنیم.

حلقه ی بعدی که مربوط به پیمایش بر روی iterations میباشد دقیقا همان کار گفته شده در دستورکار را انجام میدهد.

بخش ششم:

در این قسمت که مربوط به یادگیری Q میباشد چند متد را پیاده سازی کرده ایم:

Update: در این تابع مقدار Q را با استفاده از فرمول sample و Q بدست آورده ایم.

Sample = reward + discount * Value

Q_value = old_Q_value + alpha * (sample – old_Q_value)

بدین صورت مقدار Q_value بدست آمده و مقدار آن آپدیت میشود.

computeValueFromQValues: در این تابع ابتدا اکشن های مجاز بدست آورده میشود و سپس ماکزیمم Q valueهای مربوط به اکشن های مجاز برگردانده میشود.

getQValue: تنها مقدار Q برگردانده میشود.

computeActionFromQValues: در این تابع بر روی اکشن های مجاز پیمایش میشود و هر سری مقدار Q value بدست اورده میشود. با توجه به اینکه در اینجا ممکن است استیت هایی را داشته باشیم که هنوز کشف نشده باشند، در نتیجه مقدار Q آنها صفر است و ممکن است در یک استیت، چندین مقدار Q value صفر باشند. در این حالت بهتر است که یک اکشن تصادفی مربوط به این Q value ها انجام شود. برای همین در صورتی که مقدار q_val با مقدار max_q_val برابر شد یک انتخاب تصادفی برای انتخاب اکشن صورت میگیرد. در حالتی که تنها یک q valueماکزیمم داشته باشیم، اکشن مربوط به همین مقدار Q value را برمیگردانیم.

بخش هفتم: epsilon حریصانه

در این بخش تابع getAction را پیاده سازی کرده ایم.

پیاده سازی بدین صورت میباشد که یک عدد رندوم بین 0و 1 تولید میشود و در صورتی که مقدار epsilon از عدد رندوم بزرگتر بود، یک انتخاب تصادفی برای اکشن صورت میپذیرد. در غیر این صورت بهترین اکشن یا همان policy برگردانده میشود.

با فرض اینکه تولید کننده ی عدد رندوم، با یک توزیع نرمال عددهای بین 0 و 1 تولید کند، هر چه مقدار epsilon بزرگتر باشد، امکان این که epsilon بزرگتر از عدد رندوم باشد افزایش می‌یابد. در نتیجه احتمال انتخاب اکشن های تصادفی بیشتر میشود. به عبارتی دیگر با افزایش مقدار epsilon میزان انتخاب اکشن های تصادفی بیشتر شده و میزان exploration بیشتر میشود.

بخش هشتم:

جوابی ندارد و NOT POSSIBLE برگردانده میشود.

بخش نهم:

در این بخش پک‌من به درستی اجرا میشود!

بخش دهم:

در این قسمت برای پیاده سازی getQValue مقدار weights * featuresVector برگردانده میشود.

همچنین برای پیاده سازی متد آپدیت نیز طبق این دو فرمول عمل نمودیم:

Difference = reward + discout * max_Q_value(s’, a’) – Q(s,a)

Wi <- Wi + alpha *diffence * Fi