به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

سیستم های هوشمند

تمرین سری 5

نام و نام خانوادگی	شایان واصف
شماره دانشجویی	810197603
تاریخ ارسال گزارش	23 دی ماه

فهرست گزارش سوالات

سوال 1 – عنوان سوال

در ابتدا دیتاست داده شده را load می‌کنیم و سپس دقت را روی مجموعه Train و Test به ترتیب اعلام می‌کنیم :

Train data

```
b = []
for i in mush_train.index:
    #print(classify(mush.loc[i,features]),mush.loc[i,target])
    b.append(classify(mush_train.loc[i,features]) == mush_train.loc[i,target])
print(sum(b),"correct of",len(mush_train))
print("Accuracy:", sum(b)/len(mush_train))
```

```
7563 correct of 7600
Accuracy: 0.9951315789473684
```

شکل 1-1 : دقت در دادگان آموزش

Test data

```
#Test data
b = []
for i in mush_test .index:
    #print(classify(mush.loc[i,features]),mush.loc[i,target])
    b.append(classify(mush_test .loc[i,features]) == mush_test .loc[i,target])
print(sum(b),"correct of",len(mush_test ))
print("Accuracy:",sum(b)/len(mush_test ))
```

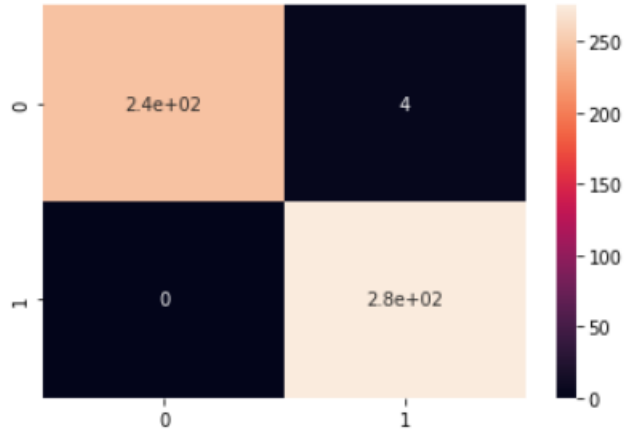
```
520 correct of 524
Accuracy: 0.9923664122137404
```

شکل 1-2 : دقت در دادگان تست

در نهایت طبق خواسته سوال ، ماتریس آشفتگی را رسم می‌کنیم :

```
sns.heatmap(A,annot=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3134442d90>
```



شکل 1-3 : ماتریس آشفتگی در دادگان تست

سوال ۲ – عنوان سوال

پیاده سازی :

در ابتدا اطلاعات داده شده توسط مساله را در زیر خلاصه می‌کنیم :

Initialization

Maximum

- MAX_Capacity = 20
- MAX_transfer = 5

FIRST of the appointment

- A_REQUEST_FIRST = 3
- B_REQUEST_FIRST = 4

END of the appointment

- A_REQUEST_END = 3
- B_REQUEST_END = 2

gamma factor(default 0.9)

- DISCOUNT = 0.9

Define reward

- CREDIT = 10
- COST = 2

شکل 2-1-1 : خلاصه اطلاعات حاصل از مدل سازی

در ادامه از سه اصطلاح زیر در کد بهره می‌بریم :

Return states

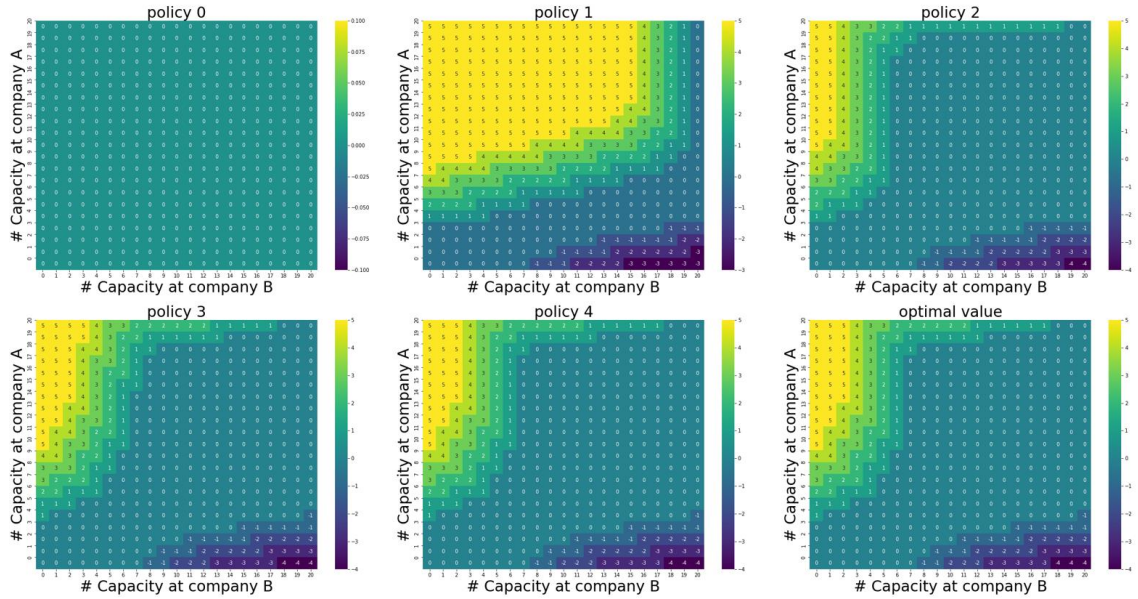
Overall info

- state: [# of capacity in first company , # of capacity in second company]
- action:
  - positive if transferin from first company to second one,
  - negative if transfering from second company to frist one,
- stateValue: state value matrix

شکل 2-1-2 : تعریف اطلاحات استفاده شده در کد

$\gamma = 0.9$:

نمودار سیاست های اتخاذ شده بر حسب تعداد ظرفیت شرکت A و B به کمک کتابخانه seaborn و دستور sns.heatmap در زیر رسم شده است :

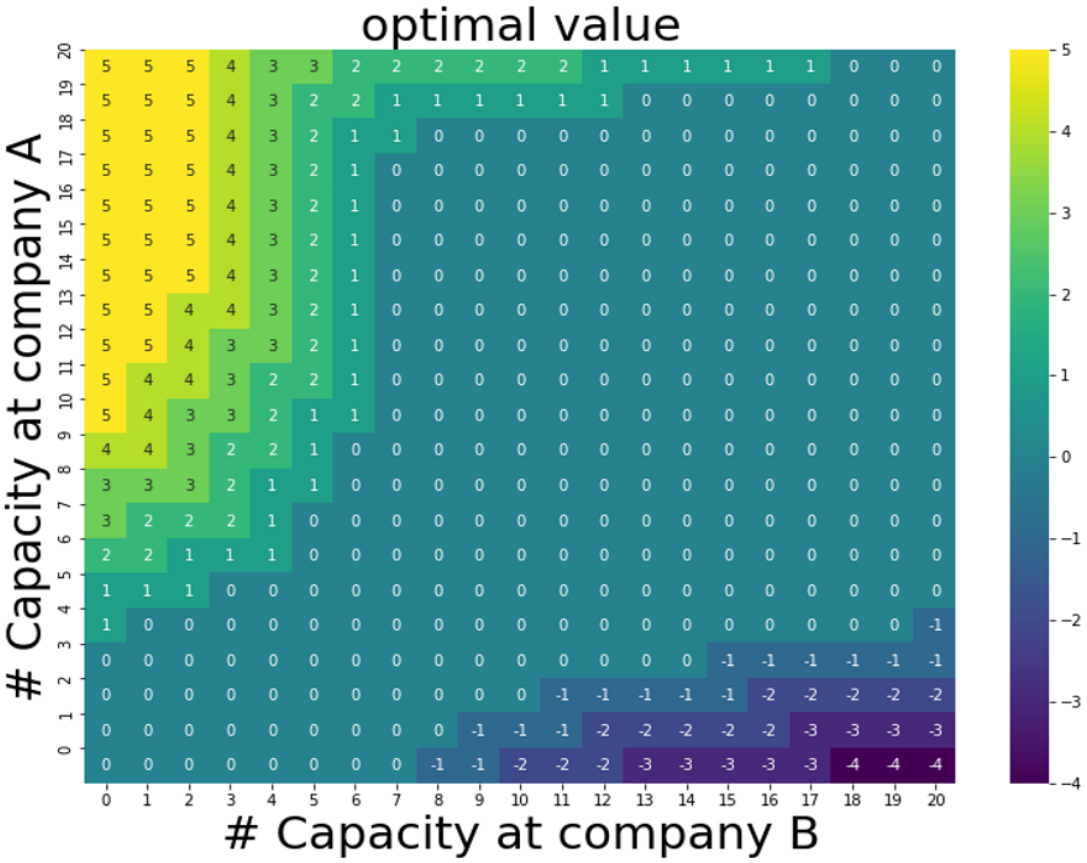


شکل 2-1-3 : heat map حاصل از سیاست بدست آمده در هر تکرار($\gamma = 0.9$)

طبق نمودار های بالا ، بعد از 6 بار بروز رسانی تابع V به همراه سیاست اتخاذ شده ، در 5 امین تکرار به سیاست بهینه می‌رسیم . اعداد نشان داده شده در هر heat map ، سیاست متناظر با ظرفیت شرکت A و B را نشان می‌دهند.

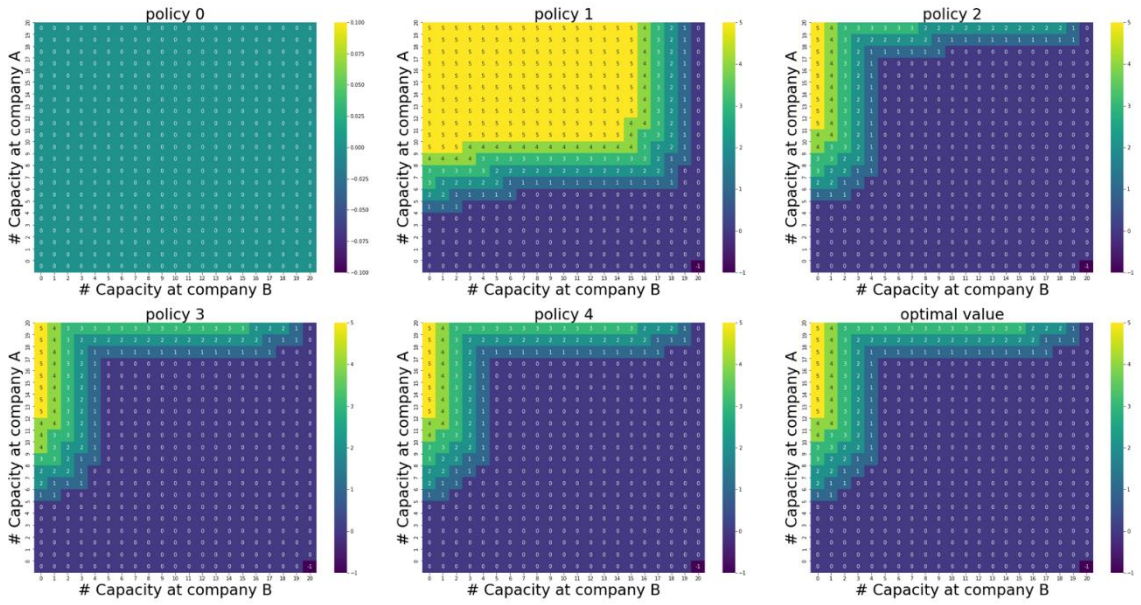
برای مثال ، در صورتی که شرکت B ، تنها 2 ظرفیت داشته باشد و شرکت A بزرگتر مساوی 14 عدد ظرفیت داشته باشد ، سیاست بهینه این خواهد بود که شرکت B از تمام ظرفیت خود برای خرید هر 5 ظرفیت شرکت A استفاده کند .

نمای دقیق تر سیاست بهینه نهایی به صورت زیر می‌باشد :



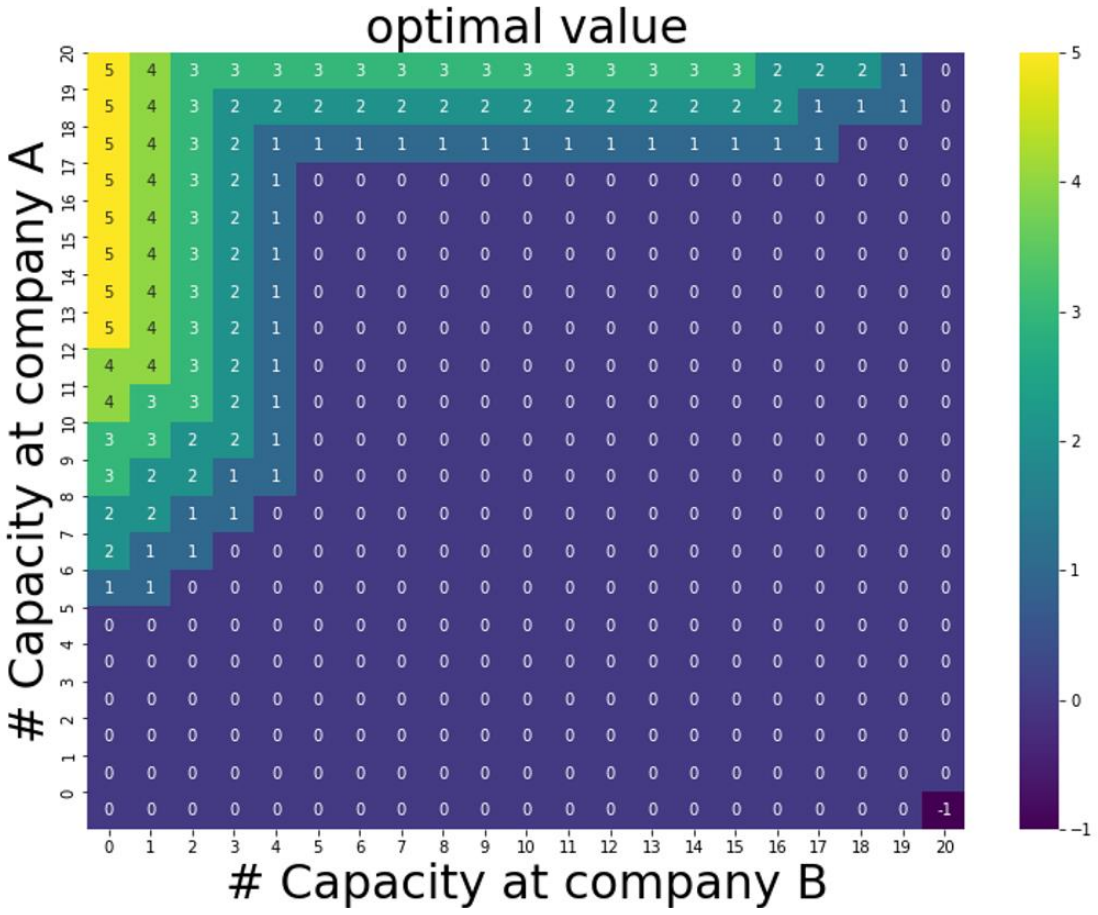
شکل 2-1-4 : heat map حاصل از سیاست بهینه همگرا شده($\gamma = 0.9$)

$\gamma = 1$:



شکل 2-1-5 : heat map حاصل از سیاست بدست آمده در هر تکرار ($\gamma = 1$)

نمای دقیق تر سیاست بهینه نهایی به صورت زیر می‌باشد :

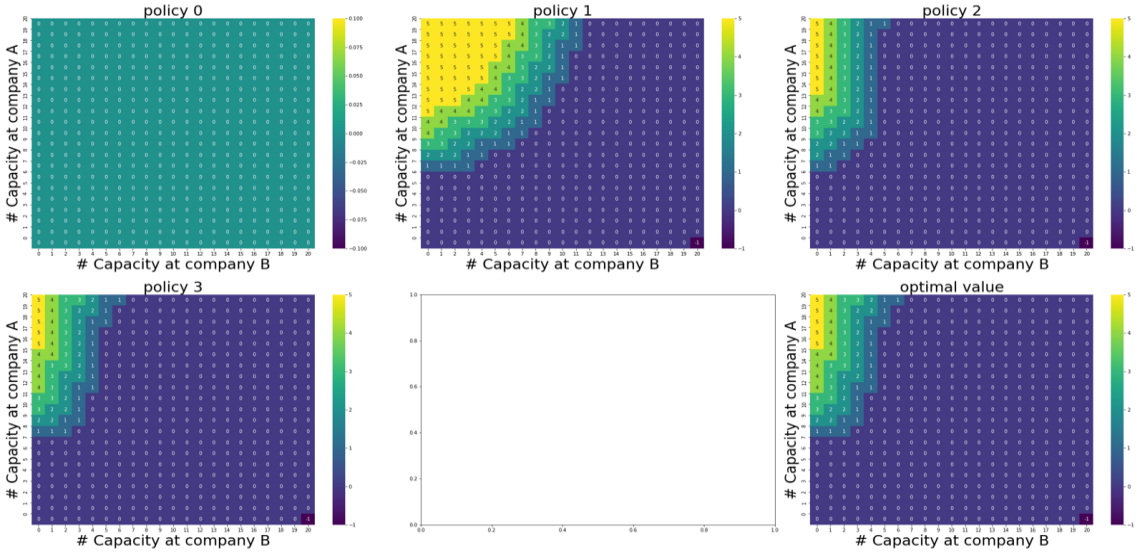


شکل 2-1-4 : heat map حاصل از سیاست بهینه همگرا شده($\gamma = 0.9$)

طبق نمودار های بدست آمده برای ضریب تخفیف برابر 1 ، تعداد سیاست های مبنی بر انتقال بین دو شرکت کاهش می‌یابد . همچنین شرکت A تنها زمانی که تعداد ظرفیت آن برابر صفر باشد ، 1 واحد از شرکت B خریداری می‌کند .

تاثیر افزایش هزینه :

طبیعتا افزایش هزینه با $\gamma = 1$ ، باید مانند افزایش γ هنگامی که هزینه ها ثابت است ، عمل کند. چرا که در رابطه bellman ، هر دو باعث افزایش تابع هزینه مربوط به حالات جدید می‌شوند. ادعای گفته شده را در زیر نیز با رسم نمودار ها می‌توان برسی کرد :

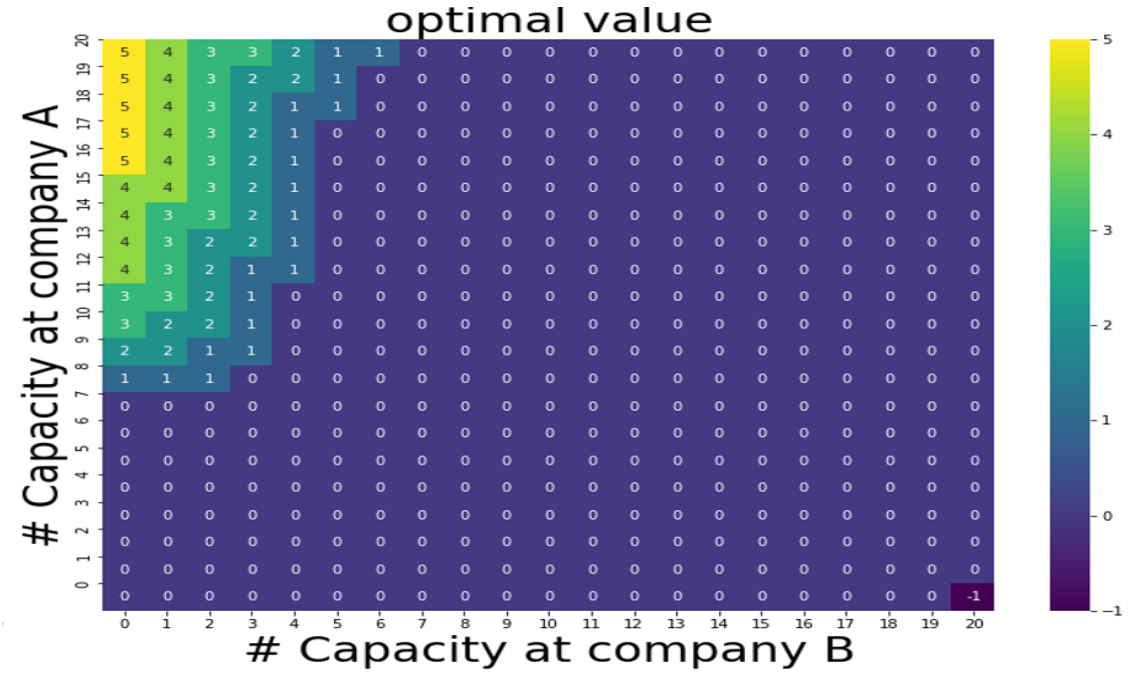


شکل 2-1-5 : heat map حاصل از سیاست بدست آمده در هر تکرار ($Cost = 6$)

طبق نمودار بالا ، با افزایش هزینه از 2 به 4 ، همگرایی از حالت $\gamma = 1$ هم سریع تر شده ولی سیاست های اتخاذ شده در هر دو تقریبا یکسان می‌باشند . (با کمی دقت می‌توان متوجه شد که تعداد سیاست های برابر صفر (action=0) در این حالت بیشتر نیز می‌باشد) .

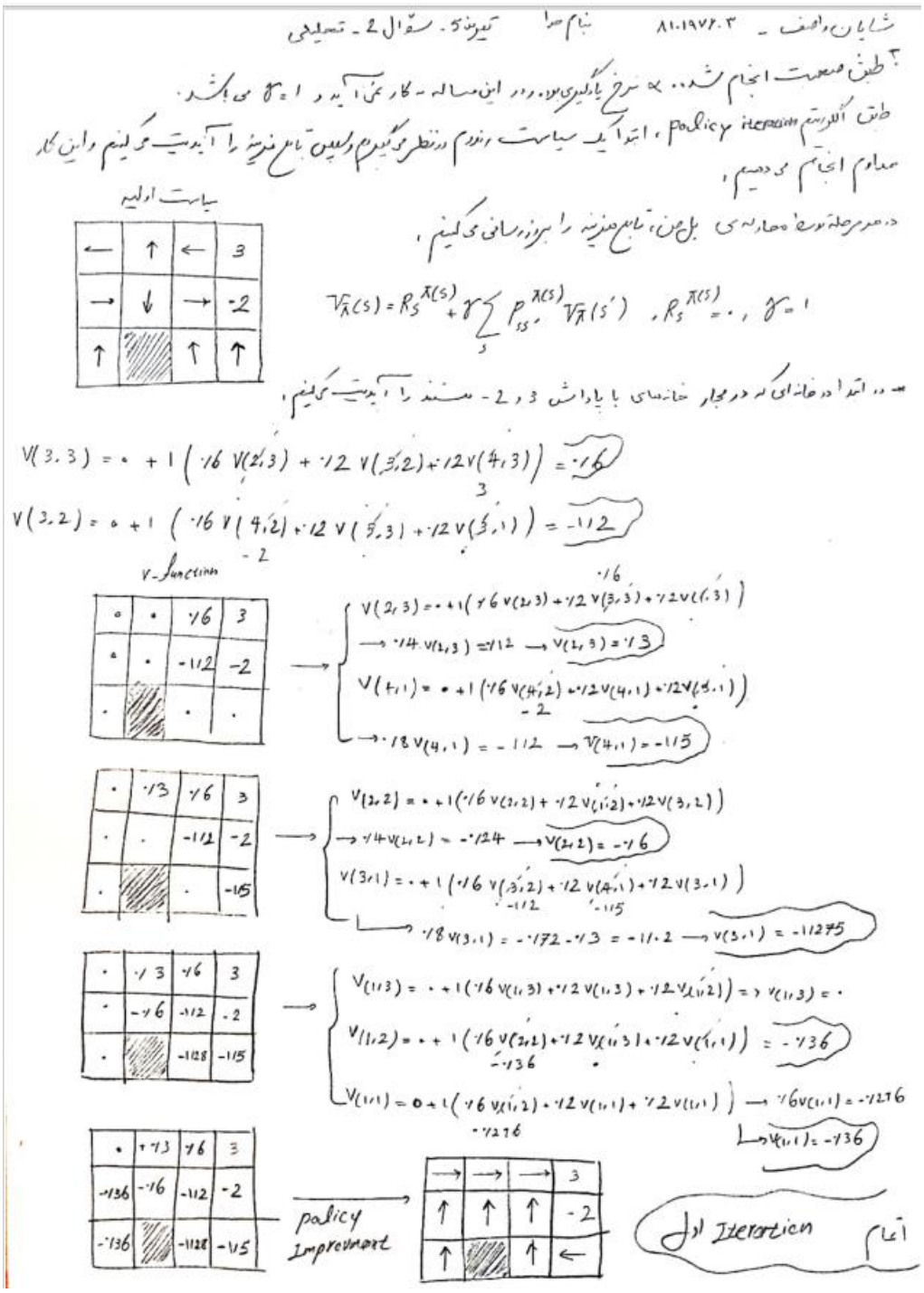
در واقع گویی با افزایش ضرر و ایجاد پاداش منفی بزرگتر ، دو شرکت ترغیب کمتری نسبت به خرید یا فروش کالا به هم دارند چون در هر صورت یکی از آنها ضرر زیادی متحمل خواهد شد.

نمای دقیق تر سیاست بهینه نهایی به صورت زیر می‌باشد :

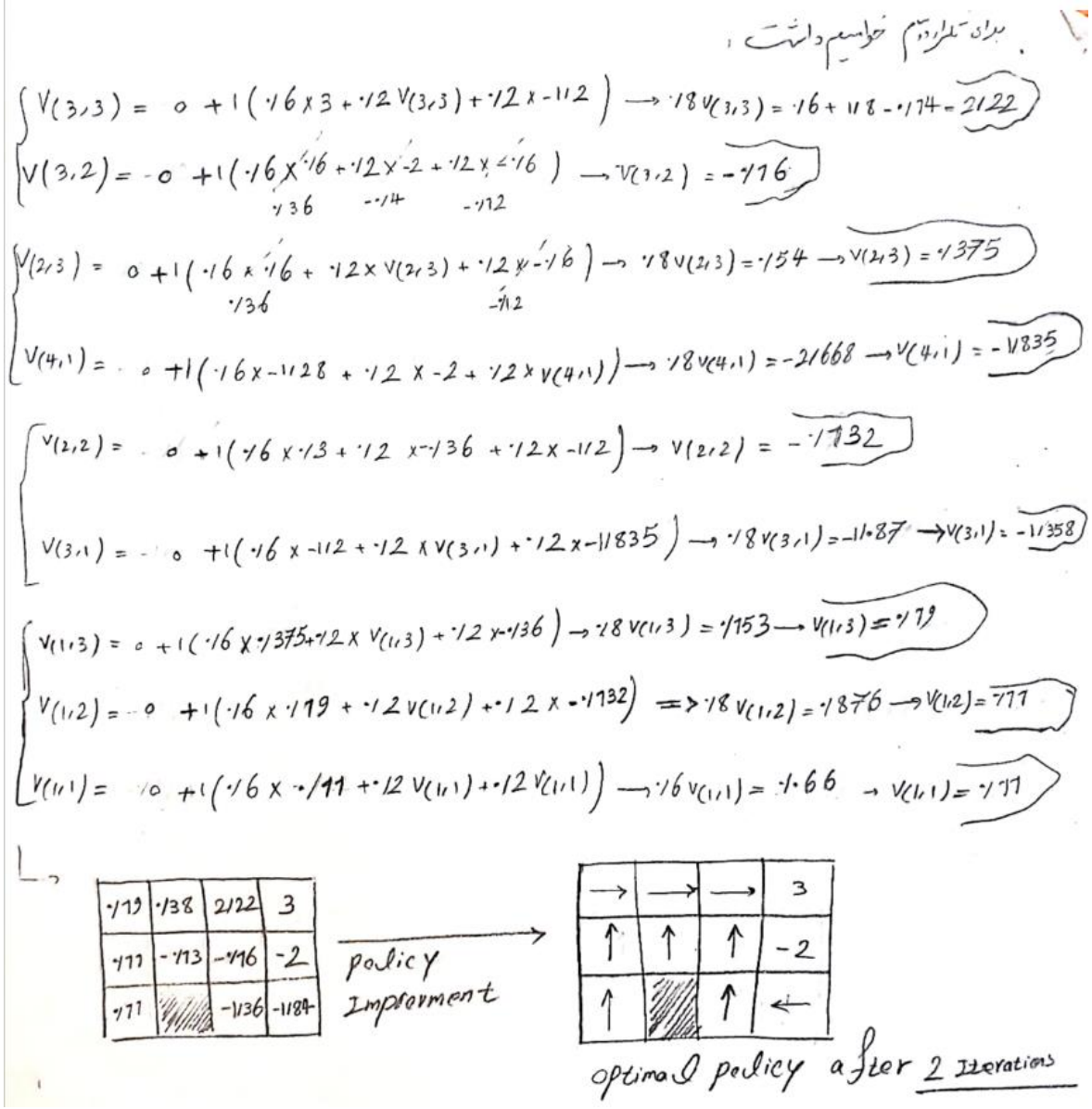


شکل 2-1-6 : heat map حاصل از سیاست بهینه همگرا شده($Cost = 6$)

تحلیلی :



شکل 2-2-1 : محاسبه Value function و Policy در اولین تکرار



شکل 2-2-2 : محاسبه Value function و Policy در دومین تکرار

سوال 3 – عنوان سوال

در ابتدا طبق کد داده شده ، محیط بازی که در آن پاداش ها و موانع تعریف شده اند را پیاده سازی می‌کنیم . در ادامه به کمک صدا زدن کلاس GridWorldEnv ، تعداد حالت ها (States) و اعمال (actions) و در نتیجه ماتریس Q را به صورت تمام یک تشکیل می‌دهیم :

```
env =  GridworldEnv()
action_space_size = env.action_space.n
state_space_size = env.observation_space.n
Q_table = np.ones((state_space_size, action_space_size))
```

```
print(action_space_size)
print(state_space_size)
print(Q_table.shape)
```

```
4
100
(100, 4)
```

شکل 4-1 : ایجاد محیط بازی و تشکیل ماتریس Q

دلیل این موضوع که از ماتریس تمام 1 به جای تمام صفر به عنوان مقدار دهی Q استفاده کردیم ، تسریع در فرآیند همگرایی الگوریتم می‌باشد . طبیعتا ، در مجموع 100 حالت و 4 عمل مختلف را می‌توانیم انجام دهیم .

در ادامه Hyper-parameter ها را تعریف می‌کنیم :

```
num_of_episodes = 20000
max_steps_per_episode = 200

learning_rate = 0.7
discount_rate = 0.99

exploration_rate = 1
```

شکل 4-2 : مقدار دهی Hyper-parameter های محیط بازی

در زیر هر کدام از پارامتر های بالا را توضیح می‌دهیم :

- Number of episodes: هر episode ، به یک دور حرکت Agent در محیط شبیه سازی شده می‌گویند. ما در هر episode ، Agent رو در جای اول قرار داده و باتوجه به ماتریس Q آپدیت

شده در مرحله قبل ، reward های جدید را بدست آورده و مشابه قبل ماتریس Q را آپدیت می‌کنیم .

- Maximum steps per episode: Agent ما در هر episode تنها قادر است تعداد محدودی گام بردارد و بعد از آن دوباره به جای اول باز میگردد.
- Learning rate: LR نرخ بروز رسانی ماتریس Q می‌باشد که تعیین می‌کند که به چه مقدار از q مرحله قبل استفاده شود و چه مقدار از مقادیر بدست آمده تاثیر پذیرد و برای مثال اگر LR برابر 1 باشد ، ماتریس Q جدید کاملا مستقل از ماتریس Q در مرحله قبل بدست می‌آید .
- Discount rate : همان γ می‌باشد که بر روی مقدار ماکسیمم q مرحله جدید تاثیر می‌گذارد.
- Exploration rate: در هر مرحله از حرکت Agent ، مساله ای بنام Exploration Vs Exploitation مطرح می‌باشد.

Exploitation از دانسته های ما نشات می‌گیرد و در واقع ما بر دانسته های خود تکیه می‌کنیم. Exploration از نادانسته ها می‌آید و مربوط به کشف حالت های ناشناخته و رسیک پذیری می‌باشد. بنابراین اگر در تمتمی مراحل تنها بر دانسته های خود تکیه کنیم ، امکان دارد به جواب بهینه اصلی همگرا نشویم و برای مثال امکان تعداد پاداش های بازی زیاد باشد و به یک local optimum همگرا شویم .

در ابتدا که Agent چیزی از اطلاعات اطراف نمی‌داند ، Exploration را برابر 1 قرار می‌دهیم و به مرور زمان ، با افزایش دانسته های خود از محیط ، Exploration کاهش می‌یابد و Exploitation افزایش می‌یابد . به میزان کاهش Exploration ، Exploration_decay_rate می‌گوییم که یا می‌توان آن را دستی تنظیم کرد و یا می‌توان آنرا معکوس episode تعریف کرد که ما از این روش استفاده می‌کنیم . در واقع با افزایش episode ، دانش ما از محیط اطراف افزایش می‌باشد و بنابراین نیاز ما به Exploration کمتر می‌باشد .پس با افزایش Episode ، Exploration باید کاهش یابد که به صورت زیر انجام می‌شود :

```
exploration_rate = 1 - np.log(episode + 1) / np.log(num_of_episodes + 1)
```

شکل 3-4 : نحوه تعریف Exploration_decay_rate

بنابراین در هر مرحله یک عدد رندوم بین صفر و یک از توزیع uniform انتخاب می‌کنیم و آنرا با Exploration آن مرحله مقایسه می‌کنیم . در صورتی که مقدار آن عدد از Exploration بیشتر باشد ، Action ای را انتخاب می‌کنیم که متناظر با بیشترین مقدار q باشد و در عوض اگر کمتر باشد ، یک action تصادفی را انتخاب می‌کنیم که کد مربوط به آن در زیر آمده است :

```
random_number = random.uniform(0,1)

if random_number > exploration_rate:
    action = np.argmax(Q_table[state,:])

else:
    action = env.action_space.sample()
```

شکل 4-4 : نحوه انتخاب action به کمک Exploration-rate

بعد از توضیحات بالا ، به هسته اصلی کد یعنی آپدیت ماتریس Q می‌پردازیم که با پارامتر های بالا به صورت زیر تعریف می‌شود :

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{learned value}}$$

شکل 4-4 : نحوه بروز رسانی ماتریس Q به کمک پارامتر های داده شده

که در کد به صورت زیر ایجاد می‌کنیم :

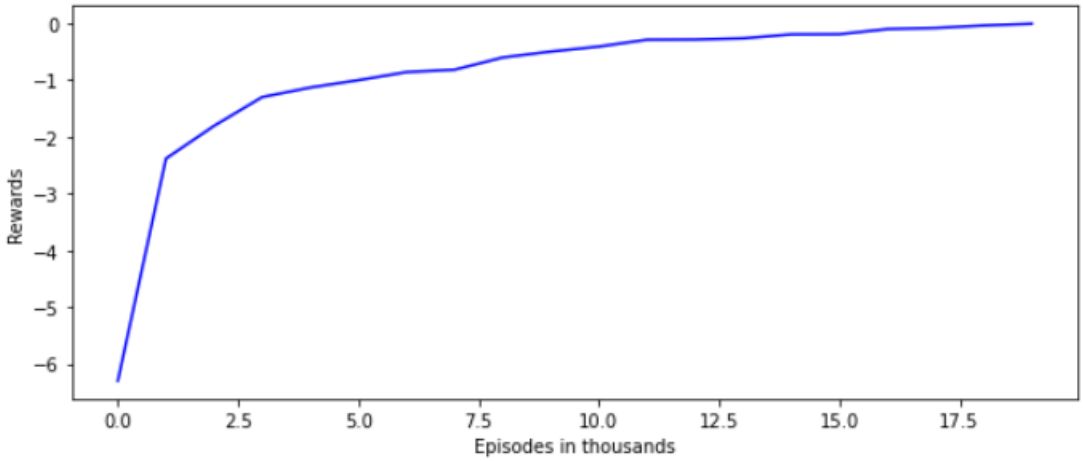
```
Q_table[state, action] = Q_table[state, action] * (1 - learning_rate) \
+ learning_rate * (reward + discount_rate * np.max(Q_table[new_state,:]))
```

شکل 4-4 : نحوه بروز رسانی ماتریس Q در کد

در نهایت پاداش را بر حسب هر 1000 تعداد Episode خروجی می‌گیریم و رسم می‌کنیم :

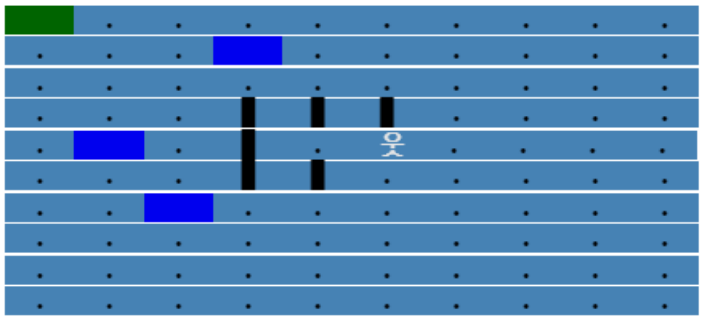
```
Average reward per thousand episodes

1000 : -6.295
2000 : -2.381
3000 : -1.805
4000 : -1.298
5000 : -1.13
6000 : -1.0
7000 : -0.858
8000 : -0.816
9000 : -0.601
10000 : -0.496
11000 : -0.408
12000 : -0.288
13000 : -0.285
14000 : -0.263
15000 : -0.193
16000 : -0.19
17000 : -0.1
18000 : -0.081
19000 : -0.037
20000 : -0.006
```

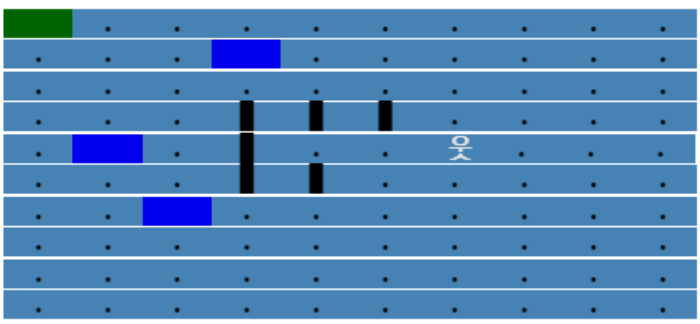
شکل 4-5 : میانگین پاداش بدست آمده در هر 1000 episode

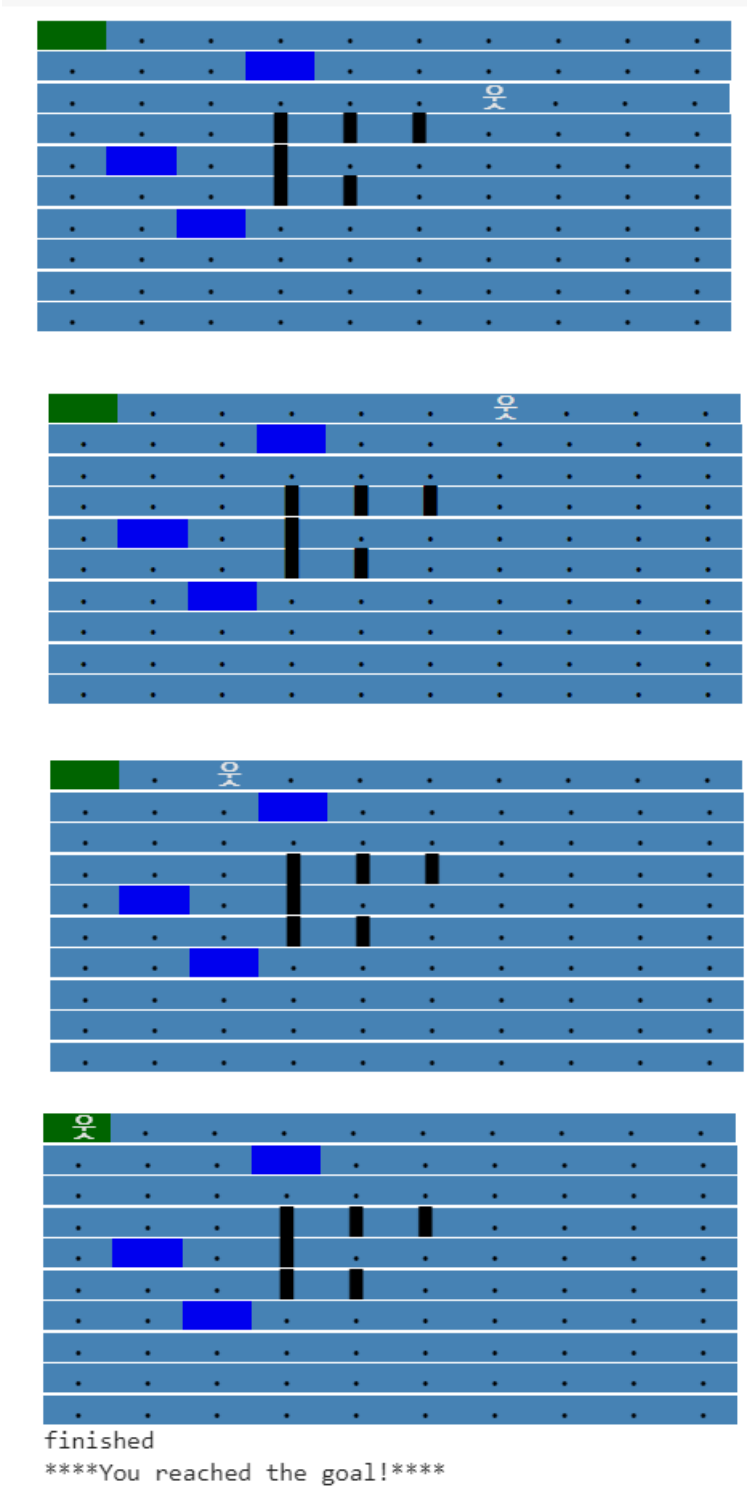


شکل 4-6 : نمودار میانگین پاداش بدست آمده در هر 1000 episode

طبق خروجی بالا ، میانگین پاداش بدست آمده ار یک عدد منفی بزرگ به سمت صفر نزدیک می‌شود تا در نهایت در صفر پایدار می‌شود . بنابراین طبق پاداش های داده شده ، در بهترین حالت ، agent ما پاداش منفی دریافت نمی‌کند و در واقع یاد می‌گیرد که به صخره ها و مکان های پر عمق نرود و همچنین در نهایت به جزیره برسد . در زیر روند حرکت agent و حالت نهایی توقف او را نشان می‌دهد (لازم به ذکر است که agent در حرکت های خود طبق آخرین ماتریس آپدیت شده Q تصمیم می‌گیرد) :







شکل 4-7 : نحوه حرکت agent تا رسیدن به جزیره

امتیازی :

طبق قسمت قبل ، agent به جزیره همگرا شد ولی قصد داریم تا سرعت آنرا کمی بهبود بخشیم . بنابراین نیاز است تا انگیزه ای برای agent ایجاد کنیم تا به سمت جزیره برود . اگر به ابتدای نمودار ها

نگاه کنید ، پیک مهمی در ابتدای کار می‌خورد که به نظر می‌رسد اگر این پیک بیشتر باشد ، agent در تعداد Episode کمتری محیط را یاد می‌گیرد. بنابراین چون در ابتدای کار با صخره ها برخورد داریم و بعد به مناطق عمیق می‌رسیم ، پاداش منفی مربوط به این دو را با هم جابجا می‌کنیم . از طرفی در تعدادی نقاط ، پاداش نسبتا کوچکی قرار می‌دهیم تا میانگین پاداش ها در هر Episode همچنان سه سمت صفر نزدیک شود (به جزیره همگرا شویم) ولی تا حدی agent را در رسیدن به جزیره بیشتر کمک کند . در کد زیر نحوه ایجاد پاداش های مثبت و منفی را آورده ایم :

```
#Rewards of each state
reward = 1.0 if is_done(s) else 0.0
reward = -103.0 if pit(s) else reward
#Generating rewards
pre_help1=lambda s :s in [40,41,42,28,48]
pre_help2=lambda s :s in [10]
reward = -1 if pre_help1(s) else reward
reward = 0.01 if pre_help2(s) else reward
```

شکل 4-8 : نحوه ایجاد پاداش منفی و مثبت در حالت جدید

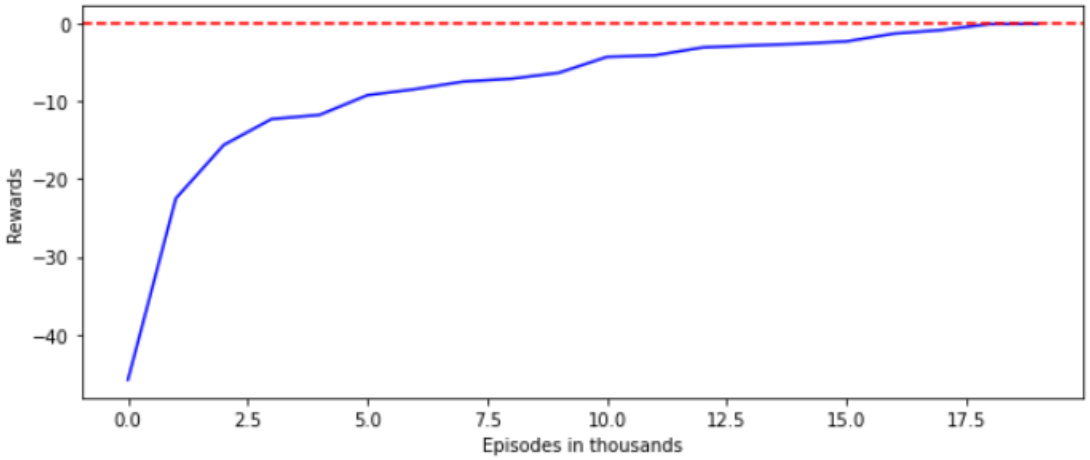
با اجرای الگوریتم ، میانگیم مقادیر پاداش بر حسب هر 1000 Episode به صورت زیر می‌باشد :

```
11000 : -4.1412999999999
12000 : -4.400279999999903
13000 : -3.6583699999998918
14000 : -2.6641999999999646
15000 : -2.5042199999999735
16000 : -2.643149999999923
17000 : -1.639160000000008
18000 : -1.4831100000000086
19000 : -0.718000000000007
20000 : -0.16301000000000318
```

```
Average reward per thousand episodes

1000 : -52.276979999999774
2000 : -21.46601000000013
3000 : -15.363779999999847
4000 : -15.590669999999834
5000 : -11.86682999999984
6000 : -8.401799999999852
7000 : -7.284429999999881
8000 : -6.798479999999858
9000 : -7.525419999999853
10000 : -5.810429999999886
```

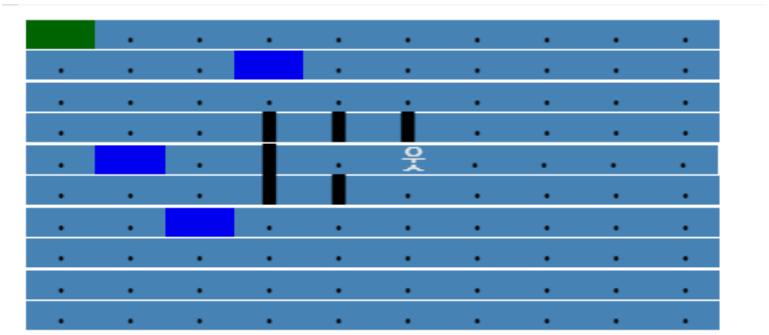
شکل 4-9 : مقادیر میانگین پاداش بر حسب Episode

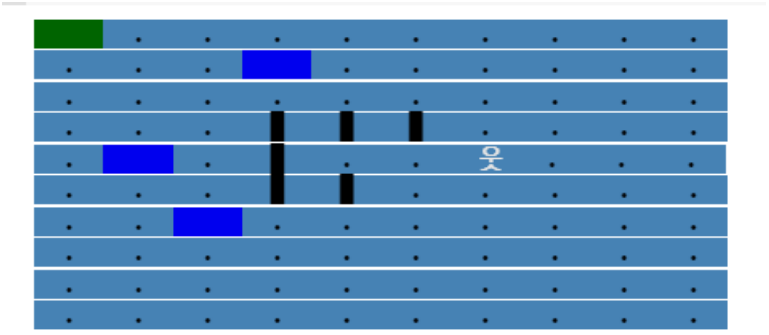
و نمودار پاداش بر حسب Episode به صورت زیر می‌باشد :

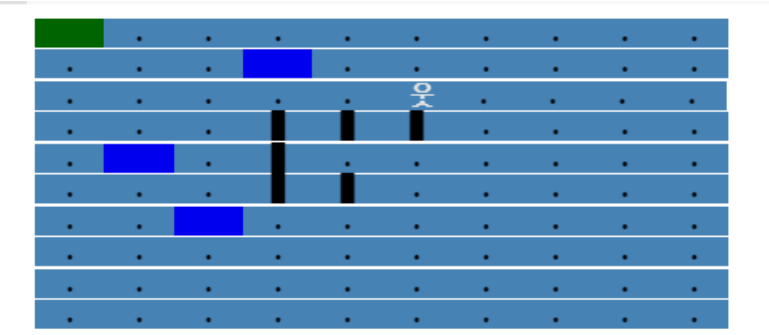


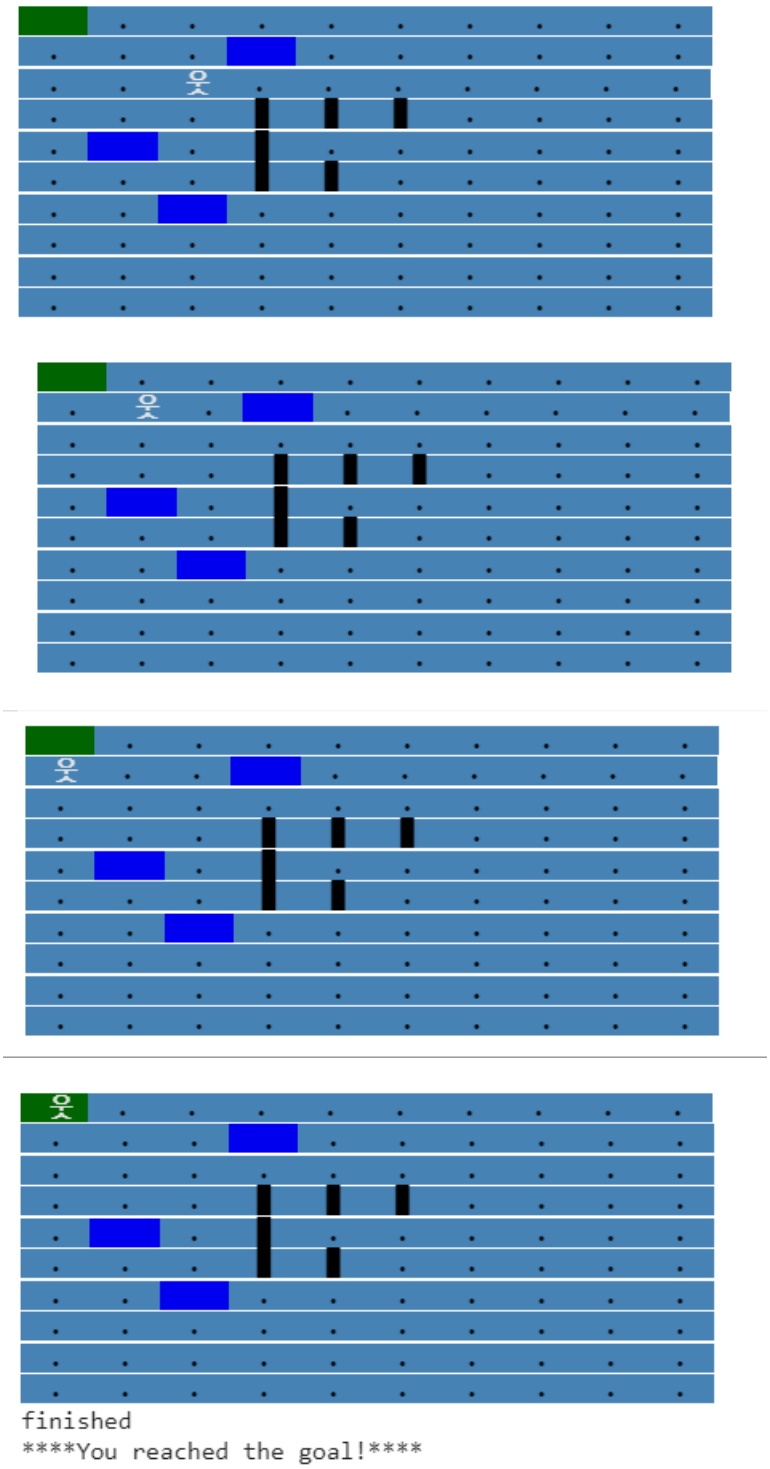
شکل 4-10 : نمودار پاداش بر حسب Episode با پاداش های جدید

حال در زیر ، نحوه جابجایی agent را در محیط رسم می‌کنیم (لازم به ذکر است که agent در حرکت های خود طبق آخرین ماتریس آپدیت شده Q تصمیم می‌گیرد) :









شکل 4-8 : نحوه حرکت agent تا رسیدن به جزیره

همانطور که مشاهده می‌شود ، در این حالت با تغییر در روند پاداش ها ، agent نسبت به قبل از اطلاعات بیشتری نسبت به محیط برخوددار است و در تنیجه با سرعت بیشتری همگرا می‌شود .