تمرین سری اول هوش مصنوعی

کیمیا اسماعیلی - 610398193

مقدمه:

بازی مارپیچ در مطالعات و آموزش، معمولا با استفاده از الگوریتم کوتاه‌ترین راه دایکسترا که منطقی‌ترین راه است، مطرح می‌شود. به دلیل مشخصه‌های این مسئله، برای یادگیری تقویت‌یافته هم امکان استفاده از آن وجود دارد.

بازی استاندارد مارپیچ:

المانهای استاندارد یک مارپیچ خانه‌ها، خانه "پنیر"(هدف) و یک "موش" (agent) است. در این مسئله ما یک جدول 10 در 10 داریم که پنیر در گوشه پایین سمت راست صفحه وجود دارد. خانه‌هایی داریم که آزاد و سفید رنگ هستند و موانعی که با رنگ قرمز معیین شده‌اند. ما تنها مجاز به عبور از خانه‌های آزاد هستیم.

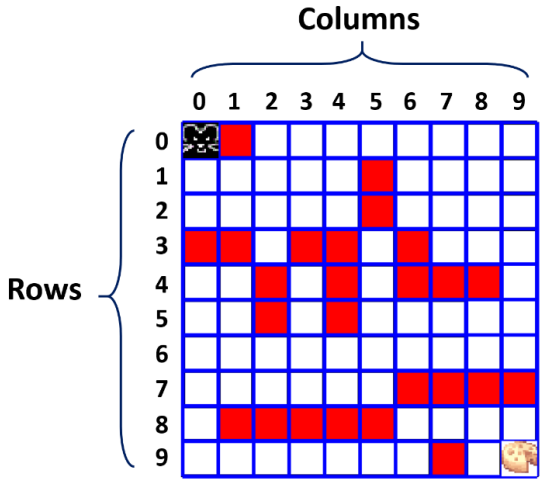
برای پیاده‌سازی کد خود، ابتدا کتابخانه‌های لازم را اضافه می‌کنیم:

In [1]:
```
from __future__ import print_function
import os, sys, time, datetime, json, random
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD , Adam, RMSprop
from keras.layers.advanced_activations import PReLU
import matplotlib.pyplot as plt
%matplotlib inline
```

ماتریس 10 در 10 مورد استفاده‌مان را به وسیله numPy میسازیم:

In [2]:
```
maze = np.array([
    [ 1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
    [ 1.,  1.,  1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.],
    [ 1.,  1.,  1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.],
    [ 0.,  0.,  1.,  0.,  0.,  1.,  0.,  1.,  1.,  1.],
    [ 1.,  1.,  0.,  1.,  0.,  1.,  0.,  0.,  0.,  1.],
    [ 1.,  1.,  0.,  1.,  0.,  1.,  1.,  1.,  1.,  1.],
    [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
    [ 1.,  1.,  1.,  1.,  1.,  1.,  0.,  0.,  0.,  0.],
    [ 1.,  0.,  0.,  0.,  0.,  0.,  1.,  1.,  1.,  1.],
    [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  0.,  1.,  1.]
])
```

موش از (0,0) (گوشه بالای سمت چپ) شروع میکند و پنیر در (9,9) (گوشه پایین سمت راست) قرار گرفته است.



در نامپای هر cell (خانه) با استفاده از یک زوج مرتب به صورت (row , col) مشخص می‌شود (ردیفها از بالا به پایین و ستونها از چپ به راست شمارش میشوند).

چگونگی environment ما:

یک فریمورک برای پروسه تصمیم‌گیری مارکوف (MDP) از یک environment و یک agent تشکیل شده است. در مسئله ما environment مارپیچ ماست که سه نوع خانه دارد:

1. اشغال‌شده
2. آزاد
3. هدف

Agent ما موش است که هدفش بدست آوردن پنیر است و تنها از خانه‌های آزاد می‌تواند بگذرد.

ما یک روش امتیازدهی (rewarding) برای موش داریم که به صورت زیر عمل میکند:
ما دقیقا 4 عملگر که با عدد نمایش داده میشوند داریم:

- 0: چپ
- 1:بالا
- 2: راست
- 3: پایین

امتیازهای ما نقاط شناور بین 1.0- و 1.0 هستند.
هر حرکت از یک موقعیت (state) به موقعیتی دیگر امتیاز میگیرد و موش امتیاز کسب میکند.
این امتیاز میتواند مقادیر مثبت (تشویق) یا منفی (تنبیه) باشند.
هر حرکت از یک موقعیت به بعدی 0.04- امتیاز برای موش هزینه میبرد.

بیشترین امتیاز مثبت 1.0 است که تنها زمانی به موش تعلق میگیرد که به پنیر برسد.
هر تلاش برای عبور از خانه‌های قرمز 0.75- امتیاز برای موش هزینه میبرد. به این صورت موش در طی زمان یاد میگیرد از موانع نگذرد. همچنین این انتخاب حرکتی در بر نداشته و تنها امتیاز کم میکند.
به همین صورت تلاش برای گذشتن از لبه‌های مارپیچ، 0.8- امتیاز از موش میگیرد.
گذشتن از هر خانه تکراری 0.25- هزینه میبرد.
برای جلوگیری از به وجود آمدن لوپ بی‌نهایت و بی‌هدف نبودن بازی، اگر امتیاز موش از [0.5- * سایز مارپیچ] کمتر شود باخت اعلام میشود. در این حالت ما در نظر میگیریم که موش گم شده است و به اندازه کافی اشتباه کرده و از آنها یاد گرفته است، حال باید بازی جدیدی را شروع کند.

مقادیر ثابت اصلی ما:

In [3]:
```
visited_mark = 0.8  # Cells visited by the rat will be painted by gray 0.8
rat_mark = 0.5      # The current rat cell will be painteg by gray 0.5
LEFT = 0
UP = 1
RIGHT = 2
DOWN = 3

# Actions dictionary
actions_dict = {
    LEFT: 'left',
    UP: 'up',
    RIGHT: 'right',
    DOWN: 'down',
}

num_actions = len(actions_dict)

# Exploration factor
epsilon = 0.1
```

اکتشاف و بهره‌وری (exploitation and exploration):

ما قبلا توضیح دادیم min_reward و actions_dict به چه معنا هستند. حال در مورد اپسیلون که فاکتور اکشاف در Q-Learning هم نامیده میشود، صحبت میکنیم:

1. هدف اصلی در Q-training آن است که به یک Policy مانند π برسیم که به خوبی در مارپیچ مسیریابی کند. فرض میکنیم بعد از هزاران بار بازی کردن، agent ما یعنی موش میتواند یک policy دترمینیستیک واضح برای عمل کردن در هر موقعیت ممکن داشته باشد. (چه action ای را در هر استیت پیش بگیرد.)

2. Policy را به زبان ساده تابع π میگیریم که یک کپی از وضعیت فعلی مارپیچ (Envstate) به عنوان ورودی میگیرد و عملی که ایجنت ما باید انجام دهد را برمیگرداند. ورودی شامل استیت همه خانه‌های مارپیچ هستند. $\text{next action}=\pi(\text{state})$
3. هنگام شروع ما یک پالیسی کاملا رندم را انتخاب میکنیم. بعد با استفاده از آن هزاران بازی را انجام میدهیم تا یاد بگیریم چطور آن را بی‌نقص کنیم. واضحا در مراحل اولیه یادگیری پالیسی π موجب ارورها و باختن های زیادی میشود. اما پالیسی امتیازدهی ما به گونه‌ای بازخورد میدهد تا بتواند خود را بهتر کند. موتور یادگیری ما یک شبکه عصبی feed-forward ساده است که یک استیت از اینوایرومنت ما میگیرد و یک امتیاز برای هر بردار اکشنی که انجام دادیم برمیگرداند.
4. برای آن که بتوانیم Q-learning را بهتر کنیم دو نوع حرکت داریم:
بهره وری: حرکاتی هستند که پالیسی ما با توجه به تجربه قبلی به ما اعمال میکند. تابع پالیسی ما در 90 درصد مواقع قبل از اینکه حرکتی کامل شود، استفاده میشود.
اکتشاف: در حدود 10 درصد مواقع ما یک اکشن کاملا رندم استفاده میکنیم تا یک تجربه جدید داشته باشیم (و احتمالا امتیاز بهتری بگیریم) در صورتی که تابع استراتژی ما ممکن است اجازه این حرکت را به دلیل ویژگی محدودکننده اش ندهد.
5. اپسیلون، عامل اکتشاف، مشخص میکند که چه مقدار اکتشاف داشته باشیم. معمولا 0.1 است که یعنی در هر 10 حرکت، یک حرکت ایجنت رندم میشود. میتوان اپسیلون را در حین یادگیری بهینه‌تر کرد.

کلاس Qmaze:

In [4]:
```
# maze is a 2d Numpy array of floats between 0.0 to 1.0
# 1.0 corresponds to a free cell, and 0.0 an occupied cell
# rat = (row, col) initial rat position (defaults to (0,0))

class Qmaze(object):
    def __init__(self, maze, rat=(0,0)):
        self._maze = np.array(maze)
        nrows, ncols = self._maze.shape
        self.target = (nrows-1, ncols-1)   # target cell where the "cheese" is
        self.free_cells = [(r,c) for r in range(nrows) for c in range(ncols) if self._maze[r,c] == 1.0]
        self.free_cells.remove(self.target)
        if self._maze[self.target] == 0.0:
            raise Exception("Invalid maze: target cell cannot be blocked!")
        if not rat in self.free_cells:
            raise Exception("Invalid Rat Location: must sit on a free cell")
        self.reset(rat)

    def reset(self, rat):
        self.rat = rat
        self.maze = np.copy(self._maze)
        nrows, ncols = self.maze.shape
        row, col = rat
        self.maze[row, col] = rat_mark
        self.state = (row, col, 'start')
        self.min_reward = -0.5 * self.maze.size
        self.total_reward = 0
        self.visited = set()
```

```
    def update_state(self, action):
        nrows, ncols = self.maze.shape
        nrow, ncol, nmode = rat_row, rat_col, mode = self.state

        if self.maze[rat_row, rat_col] > 0.0:
            self.visited.add((rat_row, rat_col))  # mark visited cell

        valid_actions = self.valid_actions()

        if not valid_actions:
            nmode = 'blocked'
        elif action in valid_actions:
            nmode = 'valid'
            if action == LEFT:
                ncol -= 1
            elif action == UP:
                nrow -= 1
            if action == RIGHT:
                ncol += 1
            elif action == DOWN:
                nrow += 1
        else:                  # invalid action, no change in rat position
            mode = 'invalid'

        # new state
        self.state = (nrow, ncol, nmode)
```

```
    def get_reward(self):
        rat_row, rat_col, mode = self.state
        nrows, ncols = self.maze.shape
        if rat_row == nrows-1 and rat_col == ncols-1:
            return 1.0
        if mode == 'blocked':
            return self.min_reward - 1
        if (rat_row, rat_col) in self.visited:
            return -0.25
        if mode == 'invalid':
            return -0.75
        if mode == 'valid':
            return -0.04

    def act(self, action):
        self.update_state(action)
        reward = self.get_reward()
        self.total_reward += reward
        status = self.game_status()
        envstate = self.observe()
        return envstate, reward, status

    def observe(self):
        canvas = self.draw_env()
        envstate = canvas.reshape((1, -1))
        return envstate
```

```
    def draw_env(self):
        canvas = np.copy(self.maze)
        nrows, ncols = self.maze.shape
        # clear all visual marks
        for r in range(nrows):
            for c in range(ncols):
                if canvas[r,c] > 0.0:
                    canvas[r,c] = 1.0
        # draw the rat
        row, col, valid = self.state
        canvas[row, col] = rat_mark
        return canvas

    def game_status(self):
        if self.total_reward < self.min_reward:
            return 'lose'
        rat_row, rat_col, mode = self.state
        nrows, ncols = self.maze.shape
        if rat_row == nrows-1 and rat_col == ncols-1:
            return 'win'

        return 'not_over'
```

```
    def valid_actions(self, cell=None):
        if cell is None:
            row, col, mode = self.state
        else:
            row, col = cell
        actions = [0, 1, 2, 3]
        nrows, ncols = self.maze.shape
        if row == 0:
            actions.remove(1)
        elif row == nrows-1:
            actions.remove(3)

        if col == 0:
            actions.remove(0)
        elif col == ncols-1:
            actions.remove(2)

        if row>0 and self.maze[row-1,col] == 0.0:
            actions.remove(1)
        if row<nrows-1 and self.maze[row+1,col] == 0.0:
            actions.remove(3)

        if col>0 and self.maze[row,col-1] == 0.0:
            actions.remove(0)
        if col<ncols-1 and self.maze[row,col+1] == 0.0:
            actions.remove(2)

        return actions
```

با استفاده از matplotlib imshow میتوانیم مارپیچ را به تصویر بکشیم. موش توسط 50 درصد gray level و پنیر با 90 درصد gray level نمایش داده میشود. به کل صفحه مارپیچ اینوایرومنت میگوییم که در اصطلاح یادگیری تقویت یافته، ایجنت/اینوایرومنت خود را داریم.

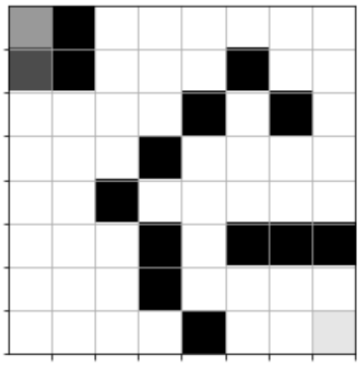
In [5]:
```
def show(qmaze):
    plt.grid('on')
    nrows, ncols = qmaze.maze.shape
    ax = plt.gca()
    ax.set_xticks(np.arange(0.5, nrows, 1))
    ax.set_yticks(np.arange(0.5, ncols, 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    canvas = np.copy(qmaze.maze)
    for row,col in qmaze.visited:
        canvas[row,col] = 0.6
    rat_row, rat_col, _ = qmaze.state
    canvas[rat_row, rat_col] = 0.3   # rat cell
    canvas[nrows-1, ncols-1] = 0.9 # cheese cell
    img = plt.imshow(canvas, interpolation='none', cmap='gray')
    return img
```

In [6]:
```
maze = [
    [ 1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.],
    [ 1.,  0.,  1.,  1.,  1.,  0.,  1.,  1.],
    [ 1.,  1.,  1.,  1.,  0.,  1.,  0.,  1.],
    [ 1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.],
    [ 1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.],
    [ 1.,  1.,  1.,  0.,  1.,  0.,  0.,  0.],
    [ 1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.],
    [ 1.,  1.,  1.,  1.,  0.,  1.,  1.,  1.]
]
```

In [7]:
```
qmaze = Qmaze(maze)
canvas, reward, game_over = qmaze.act(DOWN)
print("reward=", reward)
show(qmaze)
```

```
reward= -0.04
```

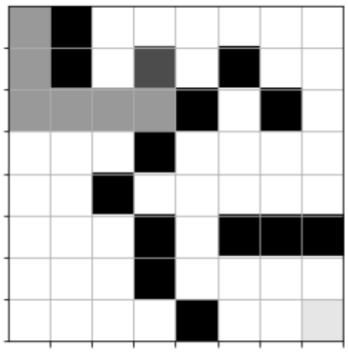
Out[7]:
```
<matplotlib.image.AxesImage at 0x135328b36a0>
```



اینجا تصویری از مارپیچ پس از 5 حرکت زیر را داریم:

```
In [8]: qmaze.act(DOWN)  # move down
        qmaze.act(RIGHT)  # move right
        qmaze.act(RIGHT)  # move right
        qmaze.act(RIGHT)  # move right
        qmaze.act(UP)  # move up
        show(qmaze)
```

```
Out[8]: <matplotlib.image.AxesImage at 0x135329090f0>
```



یک فانکشن مینویسیم که این سه آرگومان را میگیرد:

1. **Model :** یک شبکه عصبی train شده که اکشن بعدی را حساب میکند.
2. **qmaze:** یک آبجکت Qmaze
3. **rat_cell:** خانه‌ای که ایجنت از آن شروع میکند.

```
In [9]: def play_game(model, qmaze, rat_cell):
            qmaze.reset(rat_cell)
            envstate = qmaze.observe()
            while True:
                prev_envstate = envstate
                # get next action
                q = model.predict(prev_envstate)
                action = np.argmax(q[0])

                # apply action, get rewards and new state
                envstate, reward, game_status = qmaze.act(action)
                if game_status == 'win':
                    return True
                elif game_status == 'lose':
                    return False
```

Completion Check

completion check:

برای مارپیچ های کوچک میتوانیم یک completion check انجام دهیم که در آن همه بازی های ممکن را simulate میکنیم و چک میکنیم آیا مدل ما همه را برنده میشود یا نه. این روش برای مارپیچ های بزرگ قابل اجرا نیست چون یادگیری را به طور چشمگیری کند میکند.

In [10]:
```
def completion_check(model, qmaze):
    for cell in qmaze.free_cells:
        if not qmaze.valid_actions(cell):
            return False
        if not play_game(model, qmaze, cell):
            return False
    return True
```

MDP برای اینوایرومنت مارپیچ:

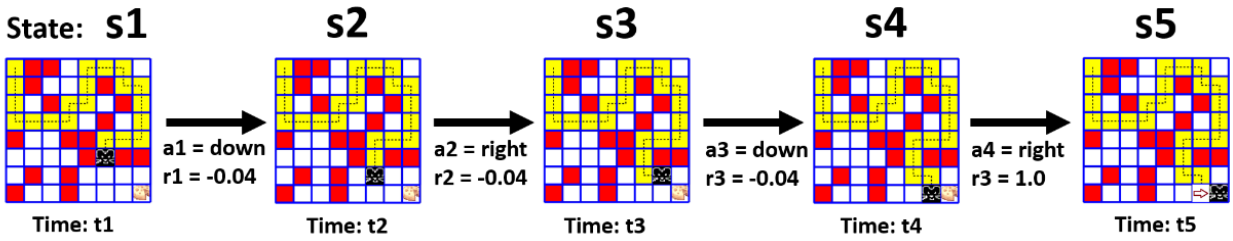
یک سیستم یادگیری تقویت‌یافته از یک اینوایرومنت و یک ایجنت پویا (dynamic) که در اینوایرومنت ما در مراحل گسسته متناهی رفتار میکند، تشکیل شده است.

1. در هر مرحله t ایجنت وارد یک استیت s میشود و باید یک اکشن مانند a از مجموعه مشخصی از اکشن ها انتخاب کند. انتخاب اینکه چه اکشنی را انتخاب کنیم تنها به اینکه الان در چه استیتی هستیم مرتبط است و اکشن های قبلی اتخاذ شده اینجا مهم نیستند. این در واقع همان MDP یا زنجیره مارکوف است.
2. انجام اکشن a در زمان t، منجر به یک ترنزیشن از استیت فعلی s در زمان t به استیت (s' $=T(s,a$) در زمان t+1 شده و امتیاز ($r=R(s,a$) که توسط ایجنت بعد از انجام اکشن گرفته میشود و ممکن است منفی باشد(جریمه). T معمولا تابع ترنزیشن نامیده میشود و R تابع پاداش یا امتیازدهی است:

$s' = T(s,a)$

$r = R(s,a)$

3. هدف ایجنت آن است که بیشترین مقدار پاداش را در یک "بازی" جمع آوری کند. پالیسی حریصانه انتخاب اکشنی که بیشترین امتیاز آنی در استیت s به ما میدهد، ممکن است به بهترین مجموع امتیازها در کل نیانجامد چون ممکن است از یک حرکت شانسی امتیاز گرفته باشد و حرکات بعدی ممکن است به ما جریمه بدهند. به این دلیل یافتن بهینه ترین راه بزرگترین چالش ماست از آنجا که در زندگی عادی هم ما با پاداش متاخر به سختی کنار می آییم.
4. در شکل زیر ما یک زنجیره مارکوف از 5 استیت یک موش در بازی مارپیچ میبینیم . امتیاز برای هر حرکت قانونی 0.04- است که در واقع یک جریمه کوچک است. دلیل آن این است که ما میخواهیم مسیر موش تا پنیر را کوتاه‌ترین حد ممکن کنیم. هر چه موش در مسیر وقت تلف کند و مسیر اضافی برود، امتیاز کل کمتری میگیرد. پس تعداد استیت ها بستگی به میزان اشتباهاتی که موش میکند و بهینه‌ترین حالت برای رسیدن به خانه هدف، دارد. زمانی که به خانه پنیر میرسد بیشترین امتیاز یعنی 1.0 را میگیرد(همه امتیازات از بازه 0.1- تا 1.0 هستند).



در نظر داریم که هر استیت دارای همه اطلاعات قابل دسترسی خانه هاست، که شامل مختصات موش هم میشود. در کد ما هر استیت را با یک بردار از طول 64 (برای یک مارپیچ 8*8) با مقادیر خاکستری 0.0 تا 1.0 نمایش داده میشود: 0.0 نمایانگر خانه اشغال شده و 1.0 خانه آزاد و قابل استفاده است. حرکت بعدی به کا حرکات قبلی مرتبط نیست، پس لازم نیست آنها را به خاطر بسپاریم.

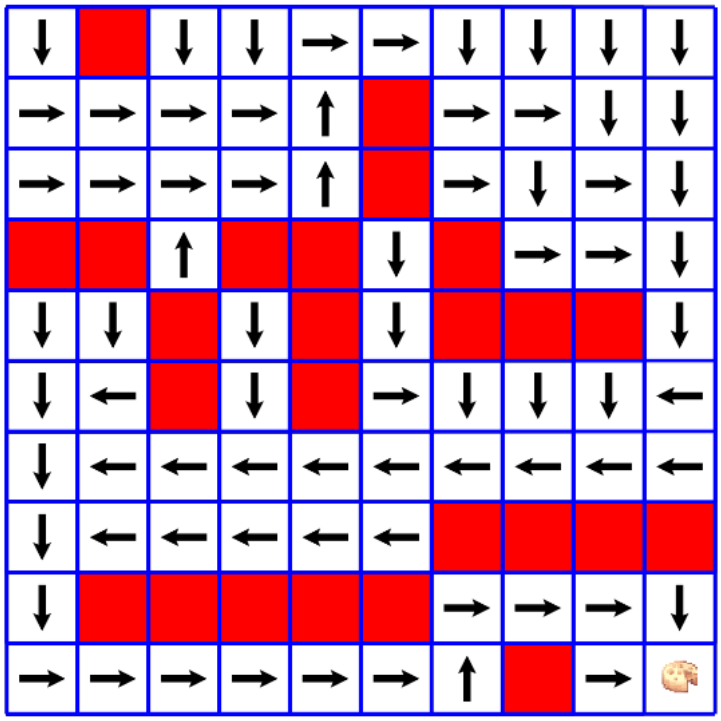
5. اگر ایجنت ما چنین دنباله اکشن هایی را پیش بگیرد(از استیت s1 شروع شده و تا زمانی که بازی تمام شود ادامه دارد): a1, a2, a3, …, an
آنگاه نتیجه برای مجموع امتیازات این دنباله به این صورت است:

$$A = R(s1,a1) + R(s2,a2) + \dots + R(sn,an)$$

هدف ما آن است که یک تابع پالیسی مانند π بیابیم که استیت s مارپیچ را به یک اکشن بهینه a متناظر کند که ما باید اتخاذ کنیم تا A را بیشینه کنیم. پالیسی π به ما میگوید که چه اکشنی را در هر استیت s که هستیم اتخاذ کنیم (با استفاده از:

$$action = \pi(s))$$

یک تابع پالیسی معمولا توسط یک نمودار پالیسی نمایش داده میشود:



6. هنگامی که تابع پالیسی π را داریم، تنها لازم است به راحتی آن را دنبال کنیم:

$a1 = \pi(s1)$
$s2 = T(s1,a1)$
$a2 = \pi(s2)$
$\ldots = \ldots$
$an = \pi(sn-1)$

پس بازی کردن این مارپیچ خودکار میشود. ما به راحتی از π میپرسیم که از چه اکشنی در هر استیت استفاده کنیم تا مطمئن باشیم در آخر بازی بیشترین مقدار امتیازات را داشته باشیم.

7. برای بدست آوردن π، اگر از منظر تئوری بازیها نگاه کنیم، کار سختی در پیش داریم علی الخصوص برای بازیهای بزرگی مانند گو (که برای آن هیچ راه حل کلاسیکی در نظریه بازیها مطرح نشده است.)

استفاده از معادله بلمن در Q-Learning:

ما میتوانیم برای یافتن π ابتدا از تابع متفاوتی به فرم Q(s,a) استفاده کنیم که تابع best utility نامیده میشود (بعضی اوقات تابع best quality نامیده میشود که حرف کیو در کیو لرنینگ از این عبارت می آید.).
تعریف Q(s,a) ساده است:
Q(s,a) : ماکسیمم بیشترین امتیازی که میتوانیم با انتخاب اکشن a در استیت s داشته باشیم.
حداقل در این مسئله مطمئن هستیم چنین تابعی وجود دارد، هرچند برای ما محاسبه آن به طور بهینه آن مشخص نباشد (به استثنای حالتی که از همه زنجیره های مارکوف ممکن که از استیت s شروع میشوند، بگذریم که اصلا بهینه نیست.) اما میتوان آن را از دیدگاه ریاضی برای تمام سیستم های مارکوف مشابه اثبات کرد که هدف ما در اینجا نیست، هرچند در زیر نمونه آن آمده است:
https://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html

زمانی که Q(s,a) را بدست آوردیم، بدست آوردن تابع پالیسی ساده میشود:

$$\pi(s) = \underset{i=0,1,\ldots,n-1}{\operatorname{argmax}} Q(s, a_i)$$

که یعنی ماQ(s,ai)a را برای همه اکشن های ai,i = 0,1,...,n-1 که در آن n تعداد اکشن ها است.
در اینجا ما aiای را انتخاب میکنیم که مقدارQ(s,ai)a آن ماکسیمم باشد. این یک راه برای انجام این کار است. اما بدون داشتن تابع Qمان چگونه این کار را انجام میدهیم؟
تابع Q به طور یک تابع بازگشتی ساده هم نوشته میشود که کمک به تخمین آن میکند و معادله بلمن نامیده میشود، به صورت:

$$Q(s, a) = R(s, a) + \max_{i=0,1,\ldots,n-1} Q(s', a_i), \quad (\text{where } s' = T(s, a))$$

به زبان ساده، مقدار تابع کیو مساوی با امتیاز آنیR(s,a)a به علاوه مقدارQ(s',a)aاست که s' استیت بعدی و ai یک اکشن است.
علاوه بر آن معادله بلمن یک نمایش یکتا از تابع best utility است. پس اگر تابع کیو بدست آمده در معادله بلمن صدق کند، حتما تابع best utility است.
برای تخمین تابع کیو باید یک شبکه عصبی N بسازیم که استیت s را به عنوان ورودی میگیرد و در خروجی به ما یک بردار q از q-valueهای متناظر با n اکشن ما میدهد:
q = (q0, q1, q2,...,qn-1)k

که qi باید مقدار Q(s,ai)a را برای هر اکشن ai تخمین بزند. زمانی که شبکه ما به طور کافی آموزش دیده و از دقت خوبی برخوردار است، از آن استفاده میکنیم تا یک پالیسی (derived

policy) را به این صورت تعریف کنیم:

$$q = N[s]$$
$$j = \underset{i=0,1,\ldots,n-1}{\operatorname{argmax}} (q_0, q_1, \ldots, q_{n-1})$$
$$\pi(s) = a_j$$

:Q-Training

چگونگی train کردن شبکه عصبی N: راه معمول آن است که یک دیتاست به اندازه کافی بزرگ از زوجهای (e,q) درست کنیم، که در آن e استیت اینوایرومنت یعنی مارپیچ ما و q = (q0, q1,...,qn-1)w اکشن های صحیح q-value هستند.برای انجام این کار ما باید هزاران بازی را اجرا کنیم تا مطمئن شویم همه حرکت‌ها بهینه هستند یا در غیر این صورت مقادیر q-value ممکن است درست نباشد که این راه بسیار سخت میشود.
راه آسانتری هم هست که قابل اجراست.

1. ما سمپل های ترین خود را از ساختن خود شبکه عصبی بدست می آوریم.ما از **derived policy** خود یعنی π، برای 90 درصد بازی ها، اقدام به عمل بهره وری میکنیم و 10 درصد دیگر را برای اکتشاف استفاده میکنیم. در این حالت، تابع هدف شبکه عصبیمان را تابع سمت راست معادله بلمن میگذاریم. اگر فرض را بر همگرایی شبکه عصبیمان بگیریم، یک تابع کیو به وجود می آورد که در معادله بلمن صدق میکند و به این دلیل بهترین تابع best utility است که ما داریم.
2. ترین کردن شبکه عصبی بعد از هر بازی توسط وارد کردن رندم تعدادی از سمپل های آخرین ترین ما انجام میشود. با فرض اینکه توانایی بازی ما هر بار بهتری میشود، از تعداد کمی از این سمپل های استفاده میکنیم. ما سمپل های قدیمی را فراموش میکنیم (چون احتمالا عملکرد بدی دارند.).
3. بعد از هر حرکت بازی ما یک اپیزود میسازیم و آن را در یک دنباله حافظه کوتاه مدت ذخیره میکنیم. هر اپیزود یک تاپل متشکل از 5 عنصر است:

episode = [envstate, action, reward, envstate_next, game_over]a

envstate استیت اینوایرومنت ما است. در مارپیچ ما این به معنی یک تصویر کلی از خانه ها با نشان دادن وضعیت موش و پنیر است. برای آنکه کار با آن برای شبکه عصبی راحت تر شود، ما مارپیچ را به صورت یک بردار یک بعدی فشرده میکنیم تا به عنوان ورودی شبکه مان قابل قبول باشد.

action: یکی از چهار حرکتی که موش ما میتواند انجام دهد:

```
0 - left
1 - up
2 - right
3 - down
```

reward: امتیاز دریافت شده از اکشن است.
envstate_next: استیت جدید اینوایرومنت که از اکشن قبلی بدست می آید.
game_over: یک مقدار بولین صحیح یا غلط است که نشان میدهد موش به پنیر رسیده(برد) یا مجموع امتیازات منفی از حد مجاز خود بیشتر شده است(باخت).

بعد از هر حرکت در بازی، ما این اپیزود 5 عنصری را میسازیم و در دنباله حافظه مان وارد میکنیم. اگر اندازه دنباله حافظه از یک مقدار معین بیشتر شد، شروع به حذف کردن عناصر از دم میشویم تا اندازه از حد تعیین شده بیشتر نشود.
وزن های شبکه ما به وسیله مقادیر رندم به وجود می آیند، پس شبکه در ابتدا نتایج بدی میدهد اما اگر پارامترهای مدل ما به درستی انتخاب شده باشند، باید به یک راه حل برای معادله بلمن همگرا شود و به این دلیل انتظار میرود آزمایش های بعدی دقیق تر باشند.
اینجا Q-table و اپیزودهای موجود را برای مسئله داریم:

state	'up'	'down'	'left'	'right'
(0, 0)	0.0	0.0	0.0	0.0
(0, 1)	0.0	0.0	0.0	0.0
(0, 2)	0.0	0.0	0.0	0.0
(0, 3)	0.0	0.0	0.0	0.0
(1, 0)	0.0	0.0	0.0	0.0
(1, 1)	0.0	0.0	0.0	0.0
(1, 2)	0.0	0.0	0.0	0.0
(1, 3)	0.0	0.0	0.0	0.0
(2, 0)	0.0	0.0	0.0	0.0
(2, 1)	0.0	0.0	0.0	0.0
(2, 2)	0.0	0.0	0.0	0.0
(2, 3)	0.0	0.0	0.0	0.0
(3, 0)	0.0	0.0	0.0	0.0
(3, 1)	0.0	0.0	0.0	0.0
(3, 2)	0.0	0.0	0.0	0.0
(3, 3)	0.0	0.0	0.0	0.0

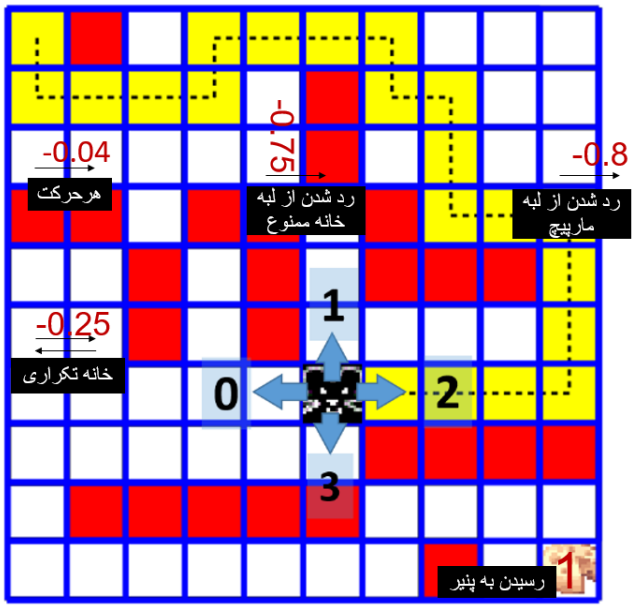
state	'up'	'down'	'left'	'right'
(0, 0)	0.000002	6.561000e-06	2.131669e-06	4.287023e-05
(0, 1)	0.000091	6.541390e-04	5.867699e-06	3.249000e-02
(0, 2)	0.039051	8.100000e-04	8.100000e-04	3.439000e-01
(0, 3)	0.000000	0.000000e+00	0.000000e+00	0.000000e+00
(1, 0)	0.000002	5.904900e-07	1.246590e-05	3.055969e-03
(1, 1)	0.001654	3.490452e-05	3.018060e-05	1.744149e-01
(1, 2)	0.039780	7.900173e-04	5.147395e-03	4.045116e-01
(1, 3)	0.814698	2.195100e-03	8.580411e-03	3.249000e-02
(2, 0)	0.000012	0.000000e+00	5.904900e-07	2.681199e-04
(2, 1)	0.062038	1.254579e-05	5.904900e-07	1.385100e-04
(2, 2)	0.000810	3.431403e-05	2.203318e-04	1.389849e-02
(2, 3)	0.150792	1.975590e-04	7.290000e-05	2.195100e-03
(3, 0)	0.000001	1.816146e-06	1.396565e-06	4.158873e-03
(3, 1)	0.017853	6.348465e-04	9.121653e-07	2.827266e-07
(3, 2)	0.000251	0.000000e+00	3.141407e-06	0.000000e+00
(3, 3)	0.010342	0.000000e+00	0.000000e+00	1.975590e-04

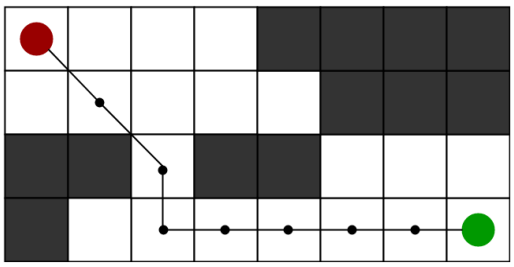
```
([[ 9000.        ,  8671.347625  ,  8775.8975    ,  8871.2       ,
    9236.7725    , 10000.        ,  9000.        ,  8952.6875    ,
   10000.        ,  9000.        ,  9705.        ,  8755.62438125,
    9000.        ,  9266.        ,  9268.5       ,  9495.        ,
    9458.214375  ,  8559.4875    ,  9000.        ,  9626.33625   ,
    9643.71192859, 10000.        ,  8050.        , 10000.        ,
    8050.        ,  8886.94596875,  8220.824375  ,  7715.18746875,
    8907.246875  , 10000.        ,  1974.73878477,  7715.18746875,
    8278.513125  ,  9047.76967721,  7448.23304531, 10000.        ],
  [ 9000.        ,  9000.        ,  9000.        ,  8050.        ,
    9000.        , 10000.        ,  9469.84      ,  9106.575     ,
   10000.        ,  9247.3151875 ,  9660.750625  , 16384.66332678,
    9356.        ,  9231.338     ,  9000.        ,  9501.14159375,
   10711.79216094, 19128.30205103,  8536.56875   , 10230.70884375,
    8984.01875   , 10000.        , 20925.90512458, 10000.        ,
    8600.62907188,  9426.70438723,  9436.13706597,  8050.        ,
   21973.79127787, 10000.        ,  8354.29473414, 28125.30266815,
```

```
  8135.31190625,  9132.52758582,  7855.93326797, 10000.        ],
[ 9013.84      ,  9106.172     ,  9112.49      ,  9267.05634375,
  9421.6525    , 10000.        ,  9476.8975    ,  9000.        ,
 10000.        ,  9015.25      , 11601.95187014,  8050.        ,
  9143.5       , 10207.95912578,  9306.        ,  9214.75      ,
 19643.34640256,  9000.        ,  9089.028125  ,  9509.6       ,
  9000.        , 10000.        ,  7147.5       , 10000.        ,
  8687.88734375, 26655.12276917, 24432.4976389 , 22679.49029107,
  4701.8378125 , 10000.        , 30534.13416024, 10166.50748639,
 11055.14597954, 11058.49358369,  8928.99026458, 10000.        ],
[ 8916.325     ,  9096.        ,  9000.        ,  9015.25      ,
  9379.227075  , 10000.        ,  9495.        ,  8859.45      ,
 10000.        ,  8559.4875    ,  9388.5905    ,  8008.40971875,
  9015.25      ,  9238.5       , 10117.33449113,  9000.        ,
  8126.513125  ,  8050.        ,  9096.94113754,  8700.67303517,
 10254.34705686, 10000.        ,  8126.513125  , 10000.        ,

 8126.513125  ,  7607.64740585, 10440.45189188,  9954.5224679 ,
 9211.21903544, 10000.        ,  6290.125     ,  6290.125     ,
 8050.        ,  8050.        ,  2604.98819449, 10000.        ]])
```

اگر هر خانه را یک راس گراف بگیریم، هر حرکت را هم یال در نظر بگیریم، وزن ها به این صورت امتیاز اعمال میشوند. نمایش ساده ای داریم:





بقیه خانه های مشابه به همین صورت امتیازدهی میشوند.

کلاس Experience:

ما در این کلاس اپیزودهای بازیمان(Experienceهای بازیمان) را در یک لیست حافظه ذخیره میکنیم. ابتدا نیاز به مواردی داریم:

1. **modal:** یک مدل شبکه عصبی
2. **max_memory:** ماکسیمم طول اپیزودها برای نگه داشتن در حافظه، اگر به این طول برسیم با اضافه شدن هر اپیزود جدید، قدیمی ترین اپیزود حذف میشود.
3. discount factor: این مورد یک ضریب مخصوص است که معمولا توسط γ نمایش داده میشود که در مدل بلمن برای اینوایرومنت استوکاستیک ما (یعنی ترنزیشن استیت های ما از راه احتمال بدست می آیند.) لازم است. یک نوع نگارش بهتر از معادله بلمن:

$$Q(s,a) = R(s,a) + \gamma \cdot \max_{i=0,..,n-1} Q(s', a_i), \quad (\text{where } s' = T(s,a))$$

In [11]:
```
class Experience(object):
    def __init__(self, model, max_memory=100, discount=0.95):
        self.model = model
        self.max_memory = max_memory
        self.discount = discount
        self.memory = list()
        self.num_actions = model.output_shape[-1]

    def remember(self, episode):
        # episode = [envstate, action, reward, envstate_next, game_over]
        # memory[i] = episode
        # envstate == flattened 1d maze cells info, including rat cell (see method: observe)
        self.memory.append(episode)
        if len(self.memory) > self.max_memory:
            del self.memory[0]

    def predict(self, envstate):
        return self.model.predict(envstate)[0]
```

```
    def get_data(self, data_size=10):
        env_size = self.memory[0][0].shape[1]   # envstate 1d size (1st element of episode)
        mem_size = len(self.memory)
        data_size = min(mem_size, data_size)
        inputs = np.zeros((data_size, env_size))
        targets = np.zeros((data_size, self.num_actions))
        for i, j in enumerate(np.random.choice(range(mem_size), data_size, replace=False)):
            envstate, action, reward, envstate_next, game_over = self.memory[j]
            inputs[i] = envstate
            # There should be no target values for actions not taken.
            targets[i] = self.predict(envstate)
            # Q_sa = derived policy = max quality env/action = max_a' Q(s', a')
            Q_sa = np.max(self.predict(envstate_next))
            if game_over:
                targets[i, action] = reward
            else:
                # reward + gamma * max_a' Q(s', a')
                targets[i, action] = reward + self.discount * Q_sa
        return inputs, targets
```

تاثیرات گاما و آلفا:

مقادیر آلفا (نرخ یادگیری)، گاما (نرخ تخفیف)، اپسیلون (احتمال انتخاب یک عمل تصادفی) و تعداد گام های هر اپیزود تعیین می شود. مقدار متداول برای ضریب یادگیری 0.1 است ولی میتوانیم ضریب یادگیری را تغییر دهیم و تغییر رفتار عامل در یادگیری و افزایش یا کاهش یادگیری کامل را با ایجاد تغییر در ضریب یادگیری مشاهده کنیم. ضمنا در مثال های اجرا شده مقدار اپسیلون نیز 0.1 در نظر گرفته شده است.

ضریب یادگیری زیاد و نزدیک به یک باعث کند شدن یادگیری عامل می شود. زیرا عامل به یادگیری هایی که از هر مرحله به دست آورده توجهی نکرده و فقط به یادگیری های جدید توجه می کند بنابراین همگرایی دیرتر صورت می گیرد. اینکه در نهایت عامل می تواند یاد بگیرد به این خاطر است که این ضریب یادگیری طبق ماهیت الگوریتم یادگیری Q به صفر همگرا شده و کم کم کاهش پیدا می کند بنابراین در آخر کار یادگیری تسریع می شود.
ضریب یادگیری ارتباط بین حافظه قدیم و جدید را توصیف میکند. مقدار بالای آن نشان دهنده این

است که اطلاعات جدید (حافظه جدید) بر حافظه قدیم ارجحیت دارد. ضریب یادگیری می گوید که Action های آتی چقدر باید مورد توجه قرار گیرند. ضریب یادگیری یک می گوید تنها اطلاعات خیلی جدید در نظر گرفته شود. نرخ یادگیری بالاتر در ابتدای امر بد نیست ولی در انتها مقدار کمتر بهتر است.
ضریب یادگیری بالاتر در ابتدا منحنی یادگیری با شیب بیشتری ایجاد می کند زیرا هدف این پارامتر همین است. نتایج نهایی اغلب با نرخ یادگیری پایین تر بهتر است. احتمالا به این دلیل که هنگام استفاده از ضریب یادگیری بالاتر تجربیات قدیمی خیلی زود با تجربیات جدید جایگزین می شوند و یادگیری را مختل می کنند که به این معنی است که عامل یادگیرنده حافظه خوبی ندارد. برای نتایج نهایی خوب یک نرخ یادگیری نسبتا پایین بهتر است. نرخ یادگیری بالاتر برای شروع کار بهتر است. ولی در کل استفاده از نرخ های یادگیری پایین تر نتایج بهتری در بر دارد. یعنی مثلا نرخ یادگیری 0.8 نتایج کوتاه مدت بدی نداشته ولی نرخ یادگیری مثل 0.2 نتایج بلند مدت بهتری دارد.
الگوریتم یادگیری Q نسبت به الگوریتم سارسا کندتر همگرا می شود ولی قابلیت این را دارد که همزمان با تغییر policyها به یادگیری ادامه دهد.
برای آلفا میتوان تصور کرد که یک تابع امتیازدهی داریم که برای حرکت ترکیبی SA امتیازهای 1 یا 0 میدهد. حالا هر زمان این حرکت را اجرا کنیم، صرفا 1 یا 0 میگیریم. اگر آلفا را یک نگه داریم، Q-value ها 1 یا 0 میشوند. اگر 0.5 باشد، 0.5+ یا 0 میگیریم و تابع همیشه بین این دو مقدار است. با این حال اگر هر دفع آلفا را 50 % کم کنیم، امیتازها به صورت1 و0 و1 و0و... میشود و Q-value ها به صورت 1 و 0.5 و 0.75 و 0.9 و 0.8 و ... میشوند و نهایتا به سمت 0.5 همگرا میشوند.(در بینهایت 0.5 میشود که از نظراحتمالاتی نتیجه مورد انتظار است.)

اگر تابع ترنزیشن مانند این حالت استوکستیک یا رندم باشد، آنگاه آلفا باید در طول زمان تغییر کند و به صفر در بینهایت نزدیک شود. این به دلیل تخمین خروجی مورد انتظار یک ضرب داخلی a(T(transition)*R(reward))است(زمانی که یکی یا هر دو رفتار رندم داشته باشند.)
گاما مقدار امتیاز آینده ما است و میتواند مقداری داینامیک یا استاتیک باشد. این مقدار باید متناظر با اندازه فضای مشاهده ما باشد. میتوان گفت گاما نرخ انقضای یک امتیاز از آخرین استیت موفق است. گامای بزرگتر(نزدیک 1) برای فضای بزرگتر و گامای کوچکتر هم معمولا برای فضای کوچکتر استفاده میشود. در این وضعیت اگر گاما مساوی با یک باشد، مقدار امتیاز بعدی ایجنت دقیقا مقدار فعلی امتیاز گرفته شده توسط او میشود. یعنی در 10 اکشن کار مثبت ایجنت با کار خنثی او تفاوتی ندارد و یادگیری حرکات خوب در مقادیر بالای گاما انجام نمیشود.
به همین صورت گامای مساوی با صفر باعث میشود که ایجنت تنها امتیازات آنی را مهم بداند که این حالت تنها با یک تابع امیتازدهی بسیار جزئی شده امکانپذیر است.

برای رفتار اکتشافی میتوان از یک سرچ جزئی تر استفاده کرد و با استفاده از درخت تصمیم و راه حل های مرتبط با بهینه تر کردن پالیسی هم میتوان کارهای مثبتی در این زمنیه انجام داد.

الگوریتم Q-Training برای QMaze ما:

الگوریتم یک لیست آرگومان های کلید واژه ای میگیرد. انتخاب‌های مهم در آن:

n-epoch: تعداد بارهای ترین کردن

max_memory: ماکسیمم تعداد experienceهایی که در مموری نگه میداریم.

data_size: تعداد سمپل هایی که در هر epoch ترینینگ استفاده میکنیم. این تعداد اپیزودهاست که ما به طور رندم از experience انتخاب میکنیم.

In [12]:
```python
def qtrain(model, maze, **opt):
    global epsilon
    n_epoch = opt.get('n_epoch', 15000)
    max_memory = opt.get('max_memory', 1000)
    data_size = opt.get('data_size', 50)
    weights_file = opt.get('weights_file', "")
    name = opt.get('name', 'model')
    start_time = datetime.datetime.now()

    # If you want to continue training from a previous model,
    # just supply the h5 file name to weights_file option
    if weights_file:
        print("loading weights from file: %s" % (weights_file,))
        model.load_weights(weights_file)

    # Construct environment/game from numpy array: maze (see above)
    qmaze = Qmaze(maze)

    # Initialize experience replay object
    experience = Experience(model, max_memory=max_memory)

    win_history = []   # history of win/lose game
    n_free_cells = len(qmaze.free_cells)
    hsize = qmaze.maze.size//2   # history window size
    win_rate = 0.0
    imctr = 1

    for epoch in range(n_epoch):
        loss = 0.0
        rat_cell = random.choice(qmaze.free_cells)
        qmaze.reset(rat_cell)
        game_over = False
```

```
        # get initial envstate (1d flattened canvas)
        envstate = qmaze.observe()

        n_episodes = 0
        while not game_over:
            valid_actions = qmaze.valid_actions()
            if not valid_actions: break
            prev_envstate = envstate
            # Get next action
            if np.random.rand() < epsilon:
                action = random.choice(valid_actions)
            else:
                action = np.argmax(experience.predict(prev_envstate))

            # Apply action, get reward and new envstate
            envstate, reward, game_status = qmaze.act(action)
            if game_status == 'win':
                win_history.append(1)
                game_over = True
            elif game_status == 'lose':
                win_history.append(0)
                game_over = True
            else:
                game_over = False

            # Store episode (experience)
            episode = [prev_envstate, action, reward, envstate, game_over]
            experience.remember(episode)
            n_episodes += 1
```

```
            # Train neural network model
            inputs, targets = experience.get_data(data_size=data_size)
            h = model.fit(
                inputs,
                targets,
                epochs=8,
                batch_size=16,
                verbose=0,
            )
            loss = model.evaluate(inputs, targets, verbose=0)

        if len(win_history) > hsize:
            win_rate = sum(win_history[-hsize:]) / hsize

        dt = datetime.datetime.now() - start_time
        t = format_time(dt.total_seconds())
        template = "Epoch: {:03d}/{:d} | Loss: {:.4f} | Episodes: {:d} | Win count: {:d} | Win rate: {:.3f} | time: {}"
        print(template.format(epoch, n_epoch-1, loss, n_episodes, sum(win_history), win_rate, t))
        # we simply check if training has exhausted all free cells and if in all
        # cases the agent won
        if win_rate > 0.9 : epsilon = 0.05
        if sum(win_history[-hsize:]) == hsize and completion_check(model, qmaze):
            print("Reached 100%% win rate at epoch: %d" % (epoch,))
            break
```

```
    # Save trained model weights and architecture, this will be used by the visualization code
    h5file = name + ".h5"
    json_file = name + ".json"
    model.save_weights(h5file, overwrite=True)
    with open(json_file, "w") as outfile:
        json.dump(model.to_json(), outfile)
    end_time = datetime.datetime.now()
    dt = datetime.datetime.now() - start_time
    seconds = dt.total_seconds()
    t = format_time(seconds)
    print('files: %s, %s' % (h5file, json_file))
    print("n_epoch: %d, max_mem: %d, data: %d, time: %s" % (epoch, max_memory, data_size, t))
    return seconds

# This is a small utility for printing readable time strings:
def format_time(seconds):
    if seconds < 400:
        s = float(seconds)
        return "%.1f seconds" % (s,)
    elif seconds < 4000:
        m = seconds / 60.0
        return "%.2f minutes" % (m,)
    else:
        h = seconds / 3600.0
        return "%.2f hours" % (h,)
```

ساختن شبکه عصبی:

انتخاب کردن پارامترهای درست برای یک مدل مناسب آسان نیست. در این مسئله یافتیم:

1. مناسب ترین activation تابع، SReLU است که srelu ی s شکل است.
2. بهینه کننده ما RMSProp است.
3. تابع loss برای ما mse است (mean squared error).

ما از دو لایه پنهان استفاده میکنیم که اندازه هرکدام به اندازه مارپیچ ماست. لایه ورودی هم دارای اندازه ای مشابه مارپیچ است چون استیت مارپیچ را به عنوان ورودی میگیرد. اندازه لایه خروجی با تعداد اکشن ها یکسان است (دراین مسئله 4 تا) چون q-value ی تخمین زده شده برای هر اکشن را خروجی میدهد (باید اکشنی که بیشتری q-value را دارد برای بازی کردن انتخاب کنیم.).

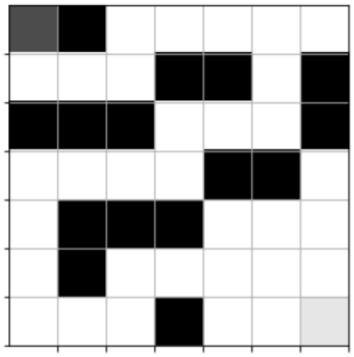
In [13]:
```
def build_model(maze, lr=0.001):
    model = Sequential()
    model.add(Dense(maze.size, input_shape=(maze.size,)))
    model.add(PReLU())
    model.add(Dense(maze.size))
    model.add(PReLU())
    model.add(Dense(num_actions))
    model.compile(optimizer='adam', loss='mse')
    return model
```

امتحان کردن:

In [14]:
```
maze =  np.array([
    [ 1.,  0.,  1.,  1.,  1.,  1.,  1.],
    [ 1.,  1.,  1.,  0.,  0.,  1.,  0.],
    [ 0.,  0.,  0.,  1.,  1.,  1.,  0.],
    [ 1.,  1.,  1.,  1.,  0.,  0.,  1.],
    [ 1.,  0.,  0.,  0.,  1.,  1.,  1.],
    [ 1.,  0.,  1.,  1.,  1.,  1.,  1.],
    [ 1.,  1.,  1.,  0.,  1.,  1.,  1.]
])

qmaze = Qmaze(maze)
show(qmaze)
```

Out[14]: <matplotlib.image.AxesImage at 0x13532983978>



In [15]:
```
model = build_model(maze)
qtrain(model, maze, epochs=1000, max_memory=8*maze.size, data_size=32)
```

```
Epoch: 000/14999 | Loss: 0.0013 | Episodes: 108 | Win count: 0 | Win rate: 0.000 | time: 4.2 seconds
Epoch: 001/14999 | Loss: 0.0043 | Episodes: 102 | Win count: 0 | Win rate: 0.000 | time: 8.0 seconds
Epoch: 002/14999 | Loss: 0.0300 | Episodes: 106 | Win count: 0 | Win rate: 0.000 | time: 11.9 seconds
Epoch: 003/14999 | Loss: 0.0046 | Episodes: 50 | Win count: 1 | Win rate: 0.000 | time: 13.7 seconds
Epoch: 004/14999 | Loss: 0.0263 | Episodes: 104 | Win count: 1 | Win rate: 0.000 | time: 17.5 seconds
Epoch: 005/14999 | Loss: 0.0170 | Episodes: 105 | Win count: 1 | Win rate: 0.000 | time: 21.5 seconds
Epoch: 006/14999 | Loss: 0.0159 | Episodes: 7 | Win count: 2 | Win rate: 0.000 | time: 21.7 seconds
Epoch: 007/14999 | Loss: 0.0073 | Episodes: 104 | Win count: 2 | Win rate: 0.000 | time: 25.6 seconds
Epoch: 008/14999 | Loss: 0.0064 | Episodes: 10 | Win count: 3 | Win rate: 0.000 | time: 26.0 seconds
Epoch: 009/14999 | Loss: 0.0203 | Episodes: 3 | Win count: 4 | Win rate: 0.000 | time: 26.1 seconds
Epoch: 010/14999 | Loss: 0.0052 | Episodes: 6 | Win count: 5 | Win rate: 0.000 | time: 26.3 seconds
Epoch: 011/14999 | Loss: 0.0026 | Episodes: 104 | Win count: 5 | Win rate: 0.000 | time: 30.1 seconds
Epoch: 012/14999 | Loss: 0.0120 | Episodes: 109 | Win count: 5 | Win rate: 0.000 | time: 34.1 seconds
Epoch: 013/14999 | Loss: 0.0088 | Episodes: 105 | Win count: 5 | Win rate: 0.000 | time: 38.0 seconds
Epoch: 014/14999 | Loss: 0.0168 | Episodes: 110 | Win count: 5 | Win rate: 0.000 | time: 42.0 seconds
Epoch: 015/14999 | Loss: 0.0191 | Episodes: 2 | Win count: 6 | Win rate: 0.000 | time: 42.1 seconds
Epoch: 016/14999 | Loss: 0.0018 | Episodes: 12 | Win count: 7 | Win rate: 0.000 | time: 42.6 seconds
Epoch: 017/14999 | Loss: 0.0014 | Episodes: 8 | Win count: 8 | Win rate: 0.000 | time: 42.9 seconds
Epoch: 018/14999 | Loss: 0.0119 | Episodes: 3 | Win count: 9 | Win rate: 0.000 | time: 43.0 seconds
Epoch: 019/14999 | Loss: 0.0060 | Episodes: 109 | Win count: 9 | Win rate: 0.000 | time: 47.0 seconds
Epoch: 020/14999 | Loss: 0.0061 | Episodes: 4 | Win count: 10 | Win rate: 0.000 | time: 47.1 seconds
Epoch: 021/14999 | Loss: 0.0028 | Episodes: 104 | Win count: 10 | Win rate: 0.000 | time: 51.0 seconds
Epoch: 022/14999 | Loss: 0.0066 | Episodes: 103 | Win count: 10 | Win rate: 0.000 | time: 54.8 seconds
Epoch: 023/14999 | Loss: 0.0021 | Episodes: 106 | Win count: 10 | Win rate: 0.000 | time: 58.6 seconds
Epoch: 024/14999 | Loss: 0.0015 | Episodes: 106 | Win count: 10 | Win rate: 0.417 | time: 62.5 seconds
Epoch: 025/14999 | Loss: 0.0030 | Episodes: 8 | Win count: 11 | Win rate: 0.458 | time: 62.8 seconds
Epoch: 026/14999 | Loss: 0.0025 | Episodes: 109 | Win count: 11 | Win rate: 0.458 | time: 66.8 seconds
Epoch: 027/14999 | Loss: 0.0009 | Episodes: 101 | Win count: 11 | Win rate: 0.417 | time: 70.5 seconds
Epoch: 028/14999 | Loss: 0.0013 | Episodes: 8 | Win count: 12 | Win rate: 0.458 | time: 70.8 seconds
Epoch: 029/14999 | Loss: 0.0071 | Episodes: 20 | Win count: 13 | Win rate: 0.500 | time: 71.6 seconds
Epoch: 030/14999 | Loss: 0.0029 | Episodes: 102 | Win count: 13 | Win rate: 0.458 | time: 75.3 seconds
Epoch: 031/14999 | Loss: 0.0448 | Episodes: 99 | Win count: 14 | Win rate: 0.500 | time: 79.0 seconds
Epoch: 032/14999 | Loss: 0.0042 | Episodes: 3 | Win count: 15 | Win rate: 0.500 | time: 79.1 seconds
Epoch: 033/14999 | Loss: 0.0328 | Episodes: 1 | Win count: 16 | Win rate: 0.500 | time: 79.1 seconds
Epoch: 034/14999 | Loss: 0.0025 | Episodes: 5 | Win count: 17 | Win rate: 0.500 | time: 79.3 seconds
Epoch: 035/14999 | Loss: 0.1047 | Episodes: 3 | Win count: 18 | Win rate: 0.542 | time: 79.4 seconds
Epoch: 036/14999 | Loss: 0.0130 | Episodes: 14 | Win count: 19 | Win rate: 0.583 | time: 79.9 seconds
Epoch: 037/14999 | Loss: 0.0475 | Episodes: 7 | Win count: 20 | Win rate: 0.625 | time: 80.2 seconds
Epoch: 038/14999 | Loss: 0.0063 | Episodes: 7 | Win count: 21 | Win rate: 0.667 | time: 80.5 seconds
Epoch: 039/14999 | Loss: 0.0377 | Episodes: 11 | Win count: 22 | Win rate: 0.667 | time: 80.9 seconds
Epoch: 040/14999 | Loss: 0.0039 | Episodes: 20 | Win count: 23 | Win rate: 0.667 | time: 81.6 seconds
Epoch: 041/14999 | Loss: 0.0043 | Episodes: 2 | Win count: 24 | Win rate: 0.667 | time: 81.7 seconds
Epoch: 042/14999 | Loss: 0.0518 | Episodes: 3 | Win count: 25 | Win rate: 0.667 | time: 81.8 seconds
Epoch: 043/14999 | Loss: 0.0016 | Episodes: 107 | Win count: 25 | Win rate: 0.667 | time: 85.7 seconds
Epoch: 044/14999 | Loss: 0.0015 | Episodes: 13 | Win count: 26 | Win rate: 0.667 | time: 86.2 seconds
Epoch: 045/14999 | Loss: 0.0058 | Episodes: 5 | Win count: 27 | Win rate: 0.708 | time: 86.4 seconds
Epoch: 046/14999 | Loss: 0.0016 | Episodes: 96 | Win count: 28 | Win rate: 0.750 | time: 89.9 seconds
Epoch: 047/14999 | Loss: 0.0042 | Episodes: 3 | Win count: 29 | Win rate: 0.792 | time: 90.0 seconds
Epoch: 048/14999 | Loss: 0.0024 | Episodes: 27 | Win count: 30 | Win rate: 0.833 | time: 91.0 seconds
Epoch: 049/14999 | Loss: 0.0021 | Episodes: 10 | Win count: 31 | Win rate: 0.833 | time: 91.4 seconds
Epoch: 050/14999 | Loss: 0.0016 | Episodes: 6 | Win count: 32 | Win rate: 0.875 | time: 91.6 seconds
Epoch: 051/14999 | Loss: 0.0013 | Episodes: 1 | Win count: 33 | Win rate: 0.917 | time: 91.6 seconds
Epoch: 052/14999 | Loss: 0.0036 | Episodes: 2 | Win count: 34 | Win rate: 0.917 | time: 91.7 seconds
Epoch: 053/14999 | Loss: 0.0024 | Episodes: 9 | Win count: 35 | Win rate: 0.917 | time: 92.0 seconds
Epoch: 054/14999 | Loss: 0.0076 | Episodes: 3 | Win count: 36 | Win rate: 0.958 | time: 92.1 seconds
Epoch: 055/14999 | Loss: 0.0014 | Episodes: 24 | Win count: 37 | Win rate: 0.958 | time: 93.0 seconds
Epoch: 056/14999 | Loss: 0.0029 | Episodes: 3 | Win count: 38 | Win rate: 0.958 | time: 93.1 seconds
Epoch: 057/14999 | Loss: 0.0006 | Episodes: 113 | Win count: 39 | Win rate: 0.958 | time: 97.4 seconds
Epoch: 058/14999 | Loss: 0.0012 | Episodes: 19 | Win count: 40 | Win rate: 0.958 | time: 98.1 seconds
Epoch: 059/14999 | Loss: 0.0021 | Episodes: 21 | Win count: 41 | Win rate: 0.958 | time: 98.9 seconds
Epoch: 060/14999 | Loss: 0.0014 | Episodes: 5 | Win count: 42 | Win rate: 0.958 | time: 99.1 seconds
Epoch: 061/14999 | Loss: 0.0016 | Episodes: 25 | Win count: 43 | Win rate: 0.958 | time: 100.0 seconds
```

```
Epoch: 062/14999 | Loss: 0.0044 | Episodes: 20 | Win count: 44 | Win rate: 0.958 | time: 100.7 seconds
Epoch: 063/14999 | Loss: 0.0012 | Episodes: 44 | Win count: 45 | Win rate: 0.958 | time: 102.4 seconds
Epoch: 064/14999 | Loss: 0.0007 | Episodes: 2 | Win count: 46 | Win rate: 0.958 | time: 102.5 seconds
Epoch: 065/14999 | Loss: 0.0007 | Episodes: 9 | Win count: 47 | Win rate: 0.958 | time: 102.8 seconds
Epoch: 066/14999 | Loss: 0.0049 | Episodes: 4 | Win count: 48 | Win rate: 0.958 | time: 102.9 seconds
Epoch: 067/14999 | Loss: 0.0004 | Episodes: 21 | Win count: 49 | Win rate: 1.000 | time: 103.7 seconds
Epoch: 068/14999 | Loss: 0.0018 | Episodes: 2 | Win count: 50 | Win rate: 1.000 | time: 103.8 seconds
Reached 100% win rate at epoch: 68
files: model.h5, model.json
n_epoch: 68, max_mem: 392, data: 32, time: 104.0 seconds
```

Out[15]: 103.995127

مدل در کمتر از 2 دقیقه ترین شد.