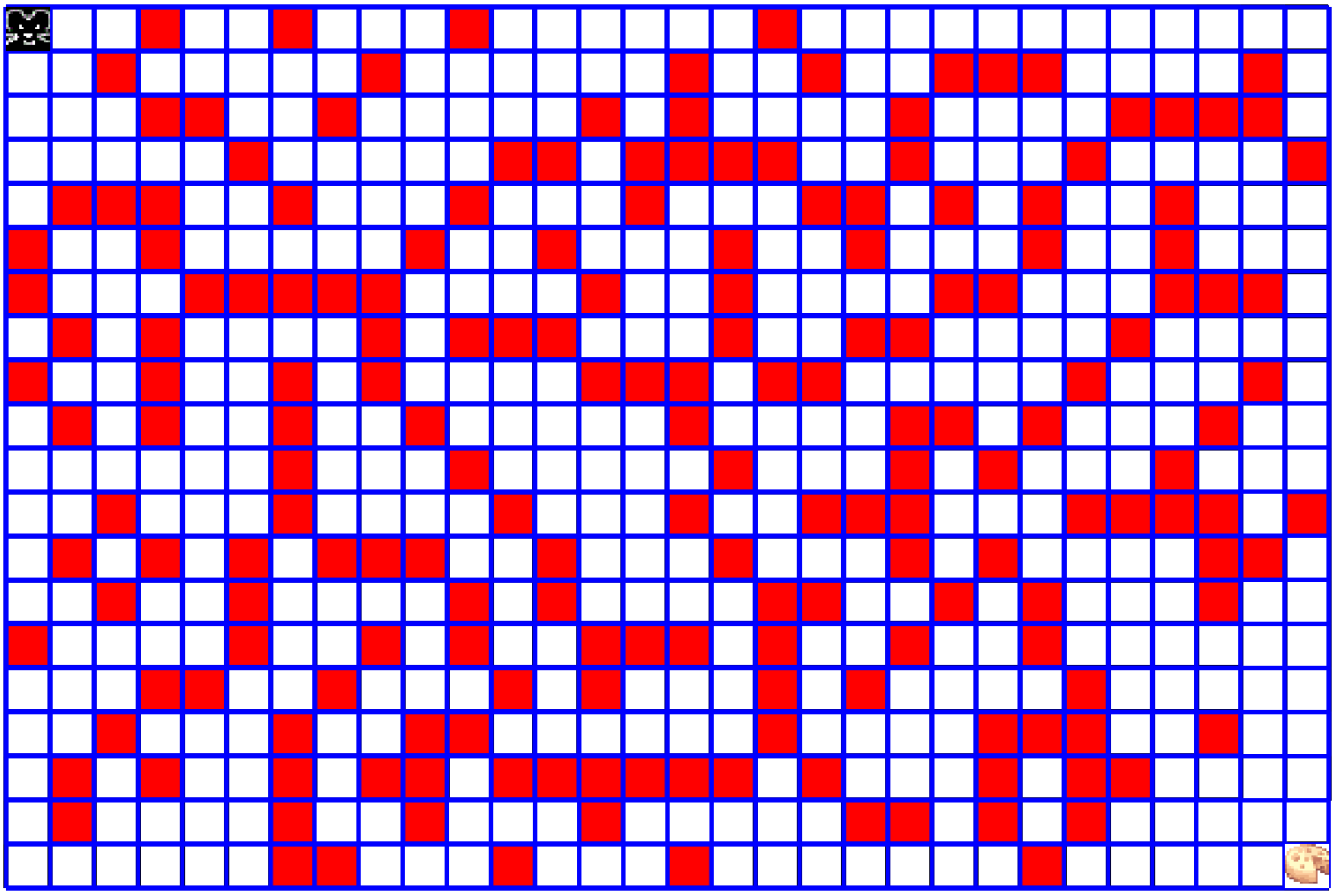
Deep Reinforcement Learning for Maze Solving



Deep Reinforcement Learning got a lot of publicity recently due to Google's acquired AI Startup DeepMind For More than 500M$ (https://techcrunch.com/2014/01/26/google-deepmind) and Intel's 15B$ Mobileye deal (https://www.technologyreview.com/s/603850/intels-15-billion-mobileye-buyout-puts-it-in-the-autonomous-car-drivers-seat). Since then, Google has also patented the mathematical aparatus (https://www.reddit.com/r/MachineLearning/comments/3c5f5j/google_patented_deep_qlearning) of using Q-learning with deep neural networks that was developed by its **DeepMind** division. In this notebook we will try to explain the main ideas behind deep reinforcement learning (also called deep **Q-learning**) by a simple application for solving classical mazes.

Before you start, it is advised that you skim through the following resources on which our code is based. Special thanks to Eder Santana (https://gist.github.com/EderSantana) for his Keras plays catch code example (https://edersantana.github.io/articles/keras_rl/) which we have copied and adapted to maze solving.

Our Python 3 code uses Keras version 1.2.2 on top of Google's **TensorFlow** version 0.12.1.

Resources

1. Demystifying Deep Reinforcement Learning (https://www.nervanasys.com/demystifying-deep-reinforcement-learning)
2. Keras plays catch, a single file Reinforcement Learning example (http://edersantana.github.io/articles/keras_rl)
3. Q-learning (https://en.wikipedia.org/wiki/Q-learning)
4. Keras plays catch - code example (https://gist.github.com/EderSantana/c7222daa328f0e885093)
5. Reinforcement learning using chaotic exploration in maze world (http://ieeexplore.ieee.org/document/1491636)
6. A Reinforcement Learning Approach Involving a Shortest Path Finding Algorithm (http://incorl.hanyang.ac.kr/xe/paper/ic/ic2003-3.pdf)
7. Neural Combinatorial Optimization with Reinforcement Learning (https://arxiv.org/abs/1611.09940)
8. Google Acquires Artificial Intelligence Startup DeepMind For More Than $500M (https://techcrunch.com/2014/01/26/google-deepmind)
9. How DeepMind’s Memory Trick Helps AI Learn Faster (https://www.technologyreview.com/s/603868/how-deepminds-memory-trick-helps-ai-learn-faster/?imm_mid=0ef03f&cmp=em-data-na-na-newsltr_20170320)
10. Methods and apparatus for reinforcement learning US 20150100530 A1 (Patent Registration) (https://www.google.com/patents/US20150100530)
11. Introduction to Reinforcement Learning (http://www.cs.indiana.edu/~gasser/Salsa/rl.html)
12. Reward function and initial values: Better choices for accelerated Goal-directed reinforcement learning (https://hal.archives-ouvertes.fr/hal-00331752/document)
13. Andrej Karpathi blog: Deep Reinforcement Learning: Pong from Pixels (http://karpathy.github.io/2016/05/31/rl/)

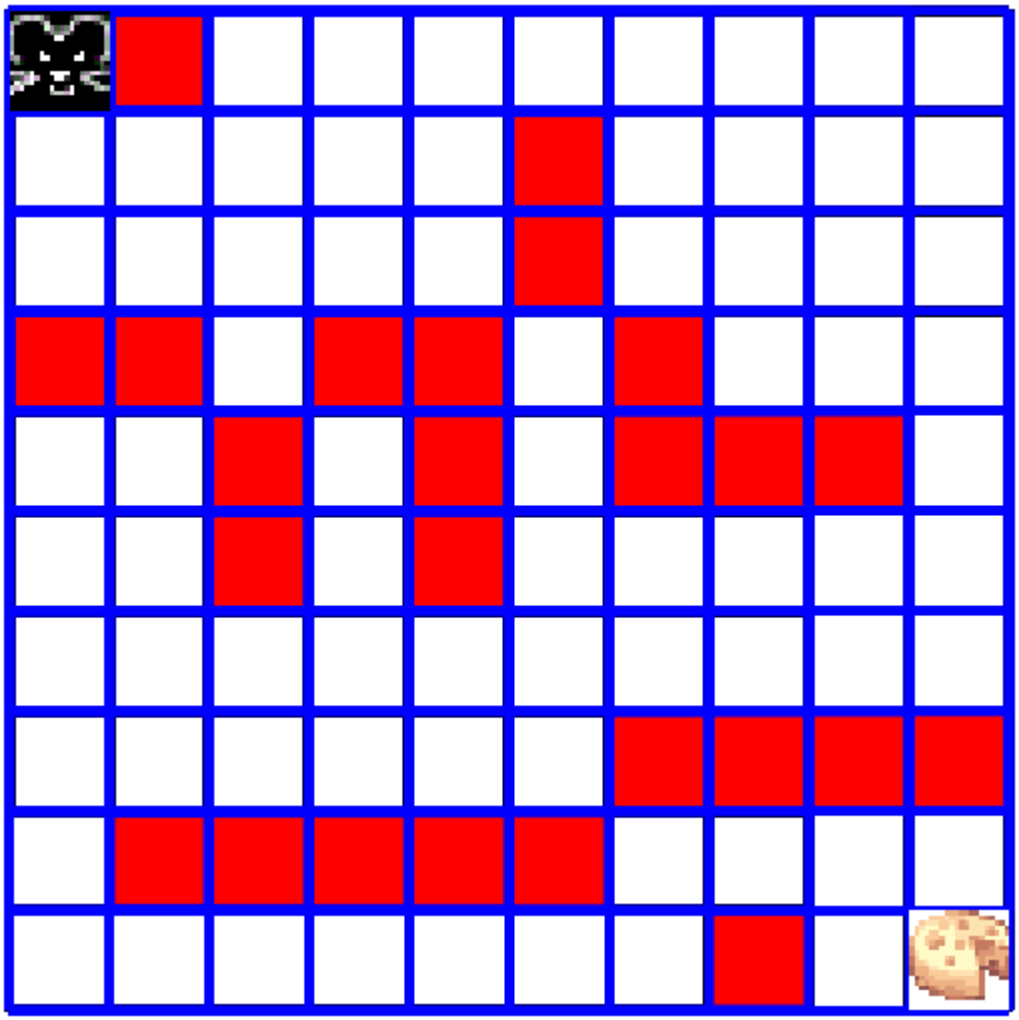
Background

Reinforcement learning is a machine learning technique for solving problems by a feedback system (rewards and penalties) applied on an **agent** which operates in an **environment** and needs to move through a series of states in order to reach a pre-defined final state. A classical example is a rat (agent) which is trying to find the shortest route from a starting cell to a target cheese cell in a maze (environment). The agent is **experimenting** and **exploiting** past experiences (**episodes**) in order to achieve its goal. It may fail again and again, but hopefully, after lots of trial and error (rewards and penalties) it will arrive to the solution of the problem. The solution will be reached if the agent finds the optimal sequence of states in which the **accumulated sum of rewards** is maximal (in short, we lure the agent to accumulate a maximal reward, and while doing so, he actually solves our problem). Note that it may happen that in order to reach the goal, the agent will have to endure many penalties (negative rewards) on its way. For example, the rat in the above maze gets a small penalty for every legal move. The reason for that is that we want it to get to the target cell in the shortest possible path. However, the shortest path to the target cheese cell is sometimes long and winding, and our agent (the rat) may have to endure many penalties until he gets to the "cheese" (sometimes called "delayed reward").

Maze Solving

Traditional maze puzzles have been used a lot in data structures and algorithms research and education. The well-known **Dijkstra shortest path algorithm** is still the most practical method for solving such puzzles, but due to their familiarity and intuititive nature, these puzzles are quite good for demonstrating and testing Reiforcement Learning techniques.

A simple maze consists of a rectangular grid of cells (usually square), a rat, and a "cheese" cell.



To keep it as simple as possible:

1. We will use small 7x7, 8x8, and 10x10 mazes as examples
2. The cheese is always at the bottom right cell of the maze
3. We have two types of cells: free cells (white) and occupied cells (red or black).

4. The rat can start from any free cell and is allowed to travel on the free cells only

Load Libraries

We will use the **keras** Python deep learning library on top of Google's **Tensorflow** version 0.12.1. We will need **matplotlib** module for drawing mazes.

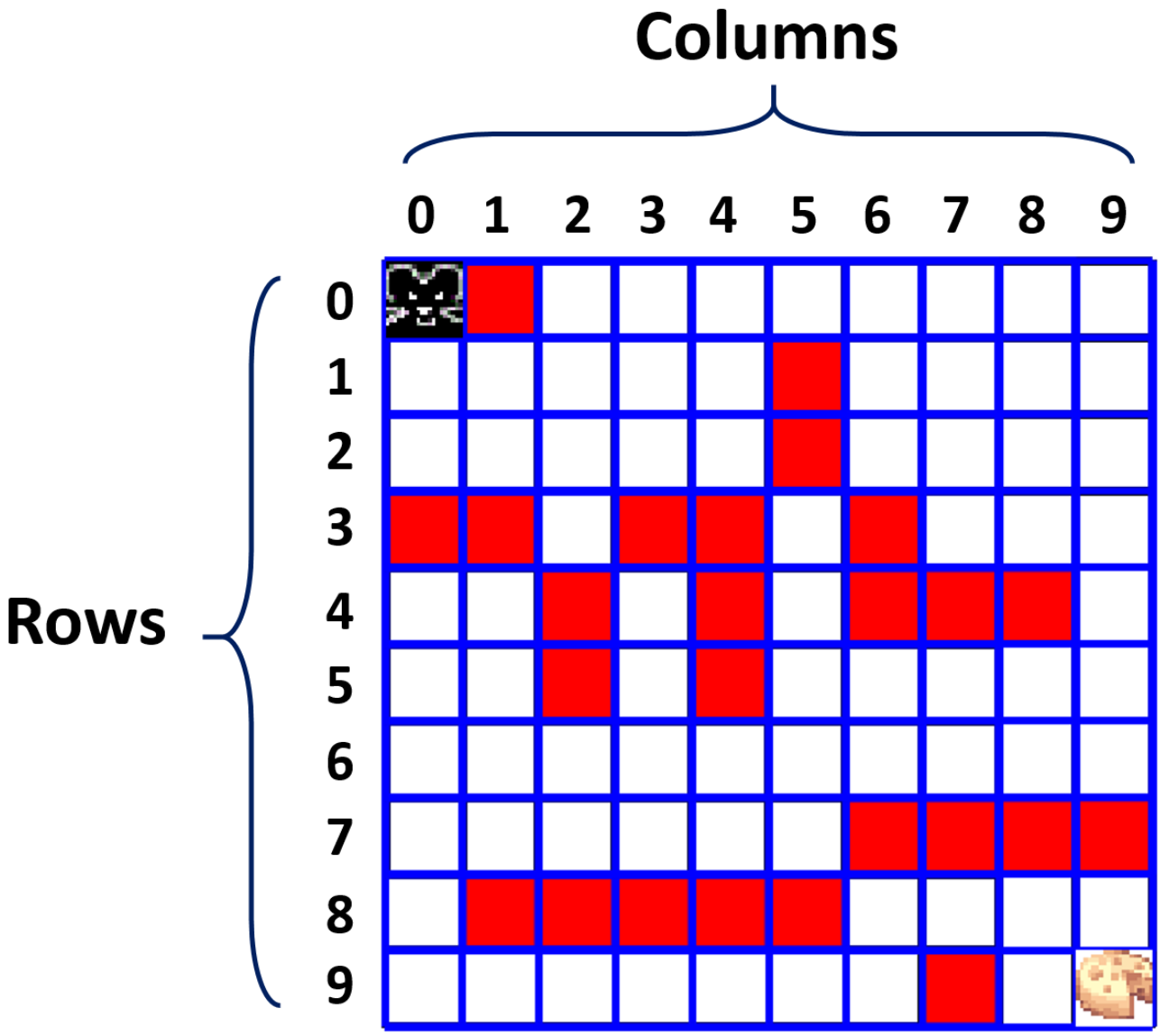
In [1]:
```
from __future__ import print_function
import os, sys, time, datetime, json, random
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD , Adam, RMSprop
from keras.layers.advanced_activations import PReLU
import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

```
Using TensorFlow backend.
```

Our 10x10 maze can be easily coded as a 10x10 Numpy matrix

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.]
])
```

This is how it looks on a **tkinter** canvas. The rat is starting at cell (0,0) (top-left cell) and the cheese is placed at cell (9,9) (bottom-right cell)



Note the usual Numpy row and column numbering conventions: each cell is represented by a pair of integers **(row, col)** where **row** is the row number (counted from the top!) and **col** is the column number (counted left to right).

Environment Description

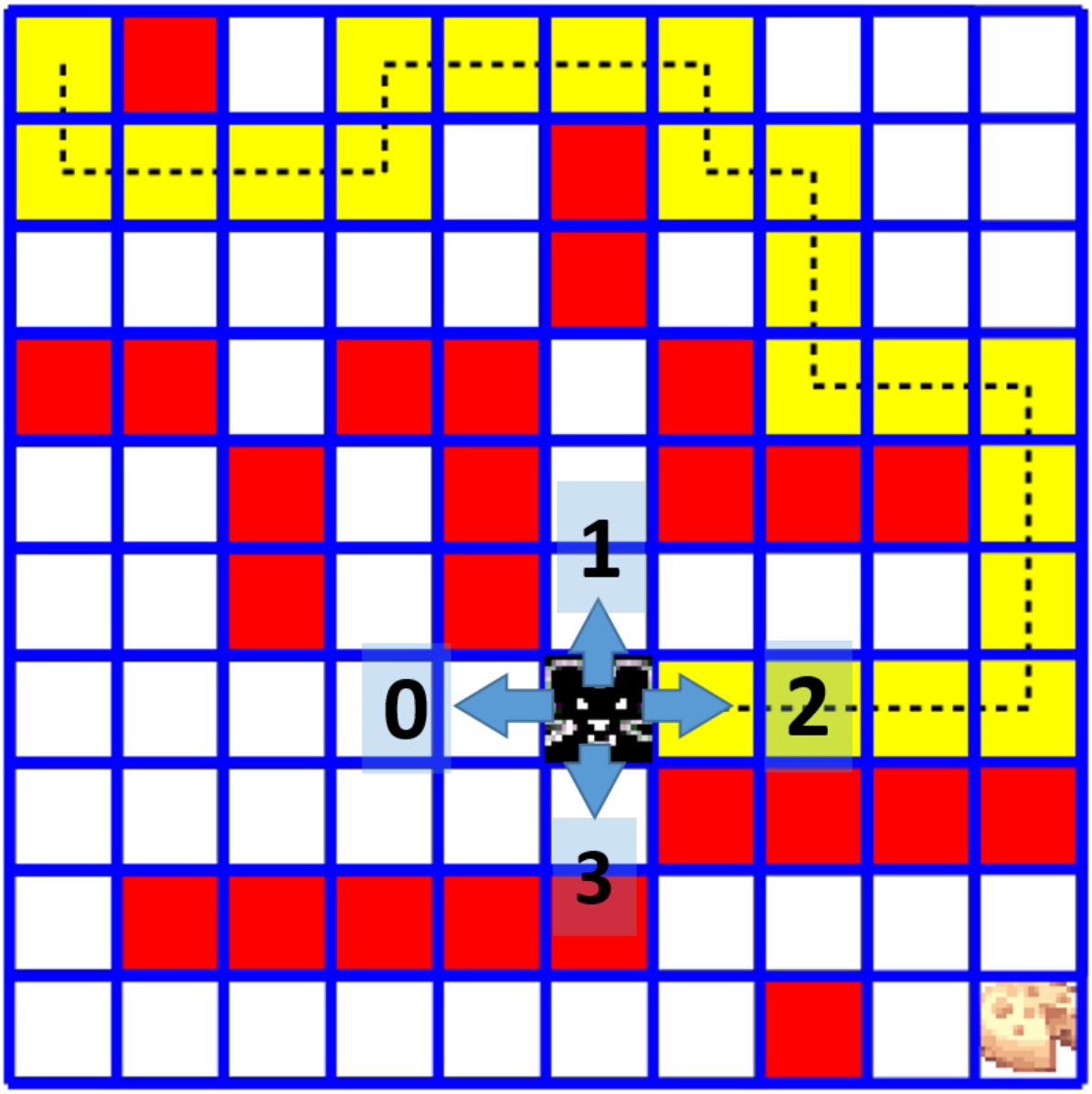
A framework for an **MDP** (Markov Decision Process) consists of an **environment** and an **agent** which acts in this environment. In our case the environment is a classical square maze with three types of cells:

1. **Occupied cells**
2. **Free cells**
3. **Target Cell** (in which the cheese is located)

Our agent is a **rat** (or a mouse if you prefer) which is allowed to move only on free cells, and whose sole purpose in life is to get to the cheese.

In our model, the rat will be "encouraged" to find the shortest path to the target cell by a simple rewarding scheme:

1. We have exactly 4 actions which we must encode as integers 0-3:
    - 0 - left
    - 1 - up
    - 2 - right
    - 3 - down



2. Our rewards will be floating points ranging from -1.0 to 1.0.
3. Each move from one state to the next state will be rewarded (the rat gets points) by a positive or a negative (penalty) amount.
4. Each move from one cell to an adjacent cell will cost the rat -0.04 points. This should discourage the rat from wandering around and get to the cheese in the shortest route possible.
5. The maximal reward of 1.0 points is given when the rat hits the cheese cell.
6. An attempt to enter a blocked cell ("red" cell) will cost the rat -0.75 points! This is a severe penalty, so hopefully the rat will learn to avoid it completely. Note that an attempt to move to a blocked cell is invalid and will not be executed. But it will incur a -0.75 penalty if attempted.
7. Same rule hold for an attempt to move outside the maze boundaries: **-0.8 points penalty**.

8. The rat will be penelized by -0.25 points for any move to a cell which he has already visited. This is clealy a counter productive action that should not be taken at all.
9. To avoid infinite loops and senseless wandering, the game is ended (**lose**) once the total reward of the rat is below the negative threshold: (-0.5 * maze.size). We assume that under this threshold, the rat has "lost its way" and already made too many errors from which he has learned enough, and should proceed to a new fresh game.

Here are our main constants:

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
```

Exploitation vs. Exploration

We already explained what **min_reward** and **actions_dict** are. The story of **epsilon**, also called **exploration factor** is part of the larger **Q-learning** story:

1. The main objective of Q-training is to develope a **policy π** for navigating the maze successfully. Presumably, after playing hundreds of games, the **agent** (rat in our case) should attain a clear deterministic policy for how to act in every possible situation. Which action to take in every possible maze state?
2. The term **policy** should be understood simply as a function π that takes a maze snapshot (**envstate**) as input and returns the action to be taken by the **agent** (rat in our case). The input consists of the full nxn maze cells state and the rat location.
$$\mathbf{next\ action} = \boldsymbol{\pi}(\mathbf{state})$$
3. At start, we simply choose a completely random policy. Then we use it to play thousands of games from which we learn how to perfect it. Surely, at the early training stages, our policy π will yield lots of errors and cause us to lose many games, but our rewarding policy should provide feedback for it on how to improve itself. Our learning engine is going to be a simple feed-forward neural network which takes an environment state (maze cells) as input and yields a reward per action vector (see later for better description).
4. In order to enhance the Q-learning process, we shall use two types of moves:
    - **Exploitation**: these are moves that our policy π dictates based on previous experiences. The policy function is used in about 90% of the moves before it is completed.
    - **Exploration**: in about 10% of the cases, we take a completely random action in order to acquire new experiences (and possibly meet bigger rewards) which our strategy function may not allow us to make due to its restrictive nature. Think of it as choosing a completey random new restaurant once in a while instead of choosing the routine restaurants that you already familiar with.
5. The exploration factor **epsilon** is the the frequency level of how much **exploration** to do. It is usually set to 0.1, which roughly means that in one of every 10 moves the agent takes a completely random action. There are however many other usage schemes you can try (you can even tune epsilon during training!)

The Qmaze Class

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 "chees
e" is
        self.free_cells = [(r,c) for r in range(nrows) for c in range(nco
ls) 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 blocke
d!")
        if not rat in self.free_cells:
            raise Exception("Invalid Rat Location: must sit on a free cel
l")
        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 positio
n
            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
```

Example of 8x8 maze

Drawing on a **tkinter** canvas is hard and requires prior knowledge. For practical work, it is best to use **matplotlib imshow** method. The rat is represented by a 50% gray level, and the cheese by a 90% gray level cell. We will use the term "enviroment" to describe the full image (with the rat and cheese thrown in). This is the standard agent/environment terminology used in most **Reinforcement Learning** discussions

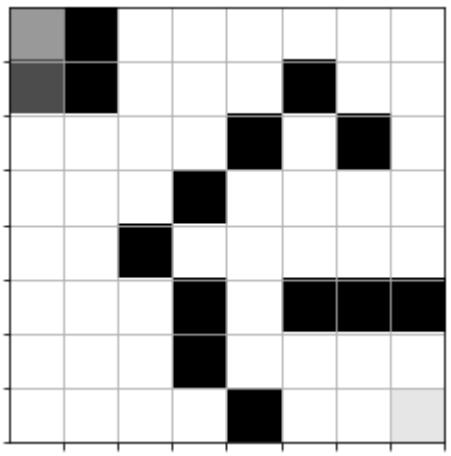
In [5]:
```
def show(qmaze):
    plt.grid('on')
    nrows, ncols = qmaze.maze.shape
    ax = plt.gca()
    ax.set_xticks(np.arange(0.5, nrows, 1))
    ax.set_yticks(np.arange(0.5, ncols, 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    canvas = np.copy(qmaze.maze)
    for row,col in qmaze.visited:
        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
```

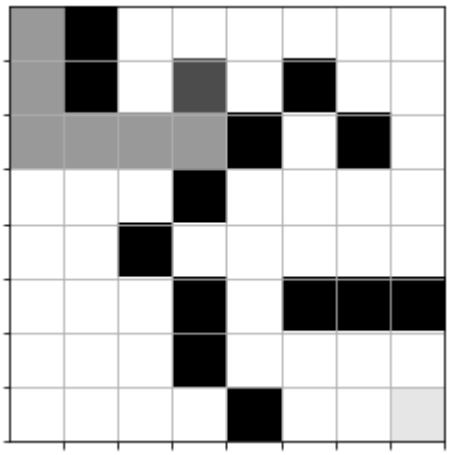
Out[7]:
```
<matplotlib.image.AxesImage at 0x135328b36a0>
```



Here is a drawing of the maze after performing the following 5 moves:

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>



Indeed, it is now very simple to write a fucntion for simulating a full game according to a given model (trained neural network). This function acceps the following three arguments:

1. **model** - a trained neural network which calculates the next action
2. **qmaze** - A Qmaze object
3. **rat_cell** - the starting cell of the rat

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

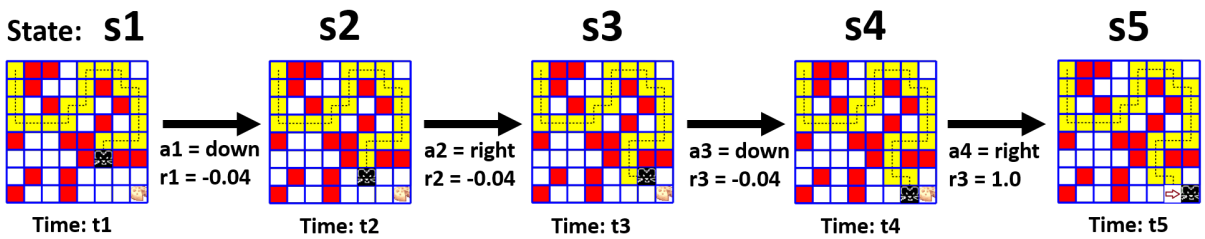
For small mazes we can allow ourselves to perform a **completion check** in which we simulate all possible games and check if our model wins the all. This is not practical for large mazes as it slows down training significantly.

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
```

Markov Decision Process for Maze Environment

A Reinforcement Learning system consists of an **environment** and a dynamic **agent** which acts in this environment in finite discrete list of time steps.

1. At every time step t, the agent is entering a state s, and needs to choose an action a from a fixed set of possible actions. The decision about which action to take should depend on the current state only (previous actions history is irrelevant). This is sometimes reffered to as **MDP: Markov Decision Process** (or shortly Markov Chain).
2. The result of performing action a at time t will result in a transition from a current state s at time t to a new state $s' = T(s,a)$ at time $t+1$, and an immediate reward $r = R(s,a)$ (numerical value) which is collected by the agent after each action (could be called a "penalty" in case the reward is negative). T is usually calle the **transition function**, and R is the **reward function**:
$$s' = T(s,a)$$
$$r = R(s,a)$$
3. The agent's goal is to collect the maximal total reward during a "game". The greedy policy of choosing the action that yields the highest immediate reward at state s, may not lead to the best possible total reward as it may happend that after one "lucky" strike all the subsequent moves will yield poor rewards or even penalties. Therefore, selecting the optimal route is a real and difficult challenge (just as it is in life, delayed rewards are hard to get by). 4. In the following figure we see a Markov chain of 5 states of a rat in a maze game. The reward for every legal move is -0.04 which is actually a "small penalty". The reason for this is that we want to minimize the rat's route to the cheese. The more the rat wonders around and wastes time, the less reward he gets. When the rat reaches the cheese cell, he gets the maximal reward of 1.0 (all rewards are ranging from -1.0 to 1.0)



Note that each state consists of all the available cells information, including the rat location. In our Python code, each state is represented by a vector of length 64 (for an 8x8 maze) with gray values 0.0 to 1.0: occupied cells is 0.0, a free cell is 1.0,

and the rat cell is 0.5. History (yellow cells) is not recorded since the next move should not depend on past moves!

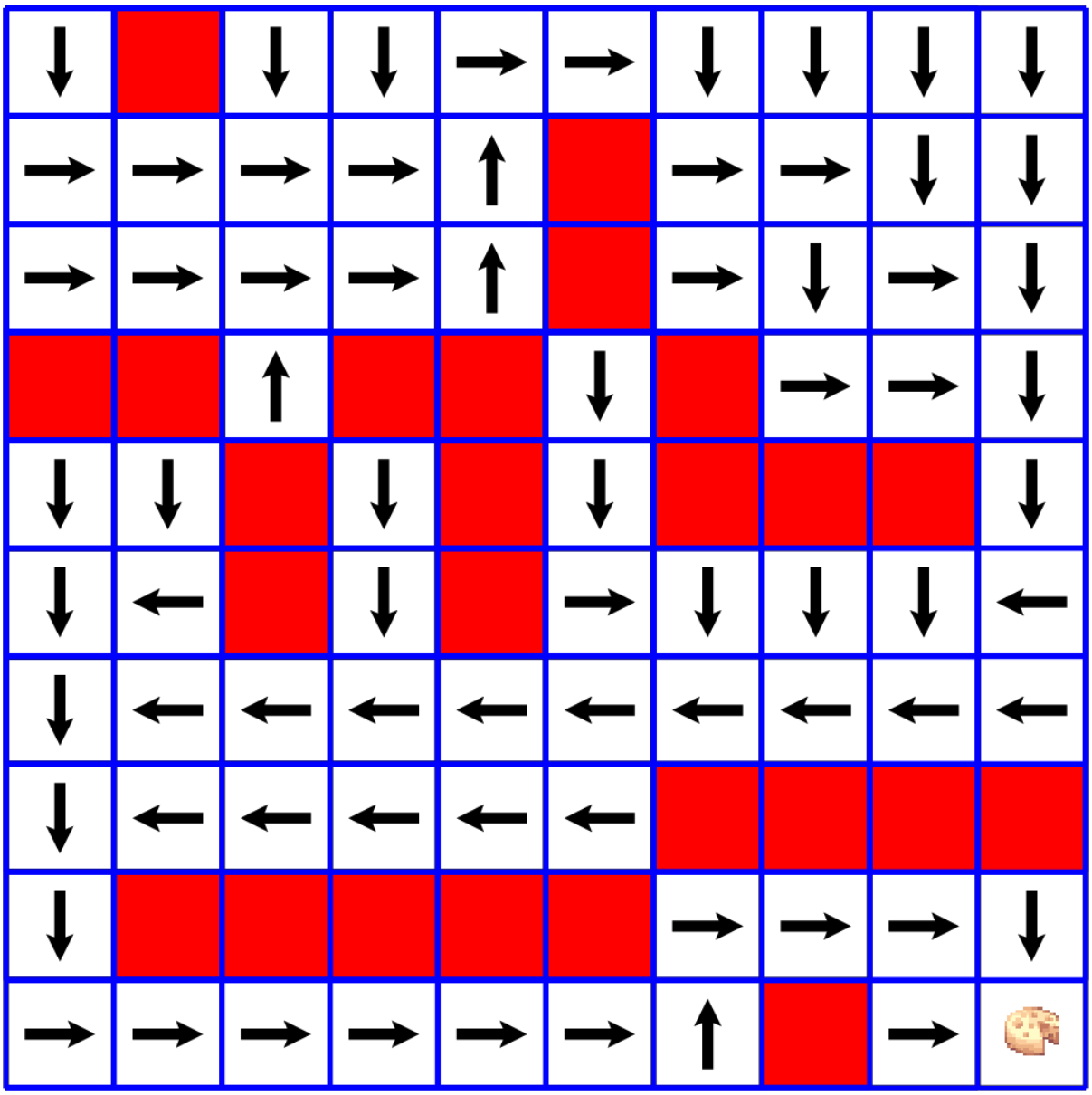
4. If our agent takes the action sequence (starting at state $s1$ till the game end): a_1, a_2, a_3, ..., a_n, then the resulting total reward for this sequence is:

$$A = R(s_1, a_1) + R(s_2, a_2) + \cdots + R(s_n, a_n)$$

Our goal is to find a **policy** function π that maps a maze state s to an "optimal" action a that we should take in order to maximize our total reward A. The policy π tells us what action to take in whatever state s we are in by simply applying it on the given state s:

$$\text{action} = \pi(s)$$

A **policy** function is sometimes best illustrated by a **policy diagram**:



5. Once we have a policy function π, all we need to do is to follow it blindly:

$$
\begin{aligned}
a_1 &= \pi(s_1) \\
s_2 &= T(s_1, a_1) \\
a_2 &= \pi(s_2) \\
\cdots &= \cdots \\
a_n &= \pi(s_{n-1})
\end{aligned}
$$

So playing the maze is now becoming an automatic pilot flight. We simply ask π what to do in each state and we're guaranteed to end the game with the maximal reward.

6. But how are we supposed to find π? From a game-theoretic point of view, it is know to be quite a difficult challenge, especially for large board games such as **GO** (for which no classical game theoretic solution is known).

Q-Learning and Bellman Equation

The trick that was used by startups such as Google DeepMind for finding π was to start with a different kind of function $Q(s,a)$ called **best utility function** (and sometimes **best quality function**, from which the **Q** letter and **Q-learning** terms were coined).

The definition of $Q(s,a)$ is simple:

$$Q(s,a) = \text{the maximum total reward we can get by choosing action } a \text{ in state } s$$

At least for our maze solving, it is easy to be convinced that such function exists, although we have no idea how to compute it efficiently (except for going through all possible Markov chains that start at state s, which is insanely inefficient). But it can also be proved mathematically for all similar Markov systems. Look for example in:
https://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html (https://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html)

Once we have $Q(s,a)$ at hand, finding a policy function is easy!

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

That is: we calculate $Q(s,a_i)$ for all actions a_i, $i = 0, 1, \ldots, n-1$ (where n is the number of actions), and select the action a_i for which $Q(s,a_i)$ is maximal. This is obviously the way to go. But we do not have the function $Q(s,a)$ yet ... how do we get it?

It turns out that the function $Q(s,a)$ has a simple **recursive property** which characterizes it, and also helps to approximate it. It is called **Bellman's Equation** and it is obvious from first sight:

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

In simple words: the value $Q(s,a)$ is equal to the immediate reward $R(s,a)$ plus the maximal value of $Q(s',a_j)$, where s' is the next state and a_i is an action.

In addition, Bellman's Equation is also a unique characterization of the best utility function. That is, if a function $Q(s,a)$ satisfies the Bellman Equation the it must be the best utility function.

To approximate $Q(s,a)$ we will build a neural network N which accepts a state s as input and outputs a vector q of **q-values** corresponding to our n actions:
$q = (q_0, q_1, q_2, \cdots, q_{n-1})$, where q_i should approximate the value $Q(s,a_i)$, for each action a_i. Once the network is sufficiently trained and accurate, we will use it to define a policy, which we call **the derived policy**, as follows

$$
\begin{aligned}
q &= N[s] \\
j &= \underset{i=0,1,\ldots,n-1}{\operatorname{argmax}} \left(q_0, q_1, \ldots, q_{n-1}\right)
\end{aligned}
$$

```
```

Q-Training

The question now is how do we train our neural network N? The usual arrangement (as we've seen a lot) is to generate a sufficiently large dataset of (e, q) pairs, where e is an **environment state** (or maze state in our case), and $q = (q_0, q_1, \ldots, q_{n-1})$ are the correct actions q-values. To do this, we will have to simulate thousands of games and make sure that all our moves are optimal (or else our q-values may not be correct). This is of course too tedious, too hard, and impractical in most real life cases.

Deep learning startups (like Google DeepMind) came up with more practical and surprisingly elegant schemes for tackling this problem. We will explore one of them (thanks to Eder Santana (https://gist.github.com/EderSantana) post which included a small and clear demonstration).

1. We will generate our training samples from using the neural network N itself, by simulating hundreds or thousands of games. We will **exploit** the **derived policy** π (from the last section) to make 90% of our game moves (the other 10% of the moves are reserved for exploration). However we will set the target function of our neural network N to be the function in the right side of Bellman's equation! Assuming that our neural network N converges, it will define a function $Q(s, a)$ which satisfies Bellman's equation, and therefore it must be the best utility function which we seek.
2. The training of N will be done after each game move by injecting a random selection of the most recent training samples to N. Assuming that our game skill will get better in time, we will use only a small number of the most recent training samples. We will forget old samples (which are probably bad) and will delete them from memory.
3. In more detail: After each game move we will generate an **episode** and save it to a short term memory sequence. An **episode** is a tuple of 5 elements that we need for one training:

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

   (a) **envstate** - environment state. In our maze case it means a full picture of the maze cells (the state of each cell including rat and chees location) To make it easier for our neural network, we squash the maze to a 1-dimensional vector that fits the networks input.

   (b) **action** - one of the four actions that the rat can do to move on the maze:

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

(c) **reward** - is the reward received from the action

(d) **envstate_next** - this is the new maze environment state which resulted from the last action

(e) **game_over** - this is a boolean value (True/False) which indicates if the game is over or not. The game is over if the rat has reached the cheese cell (win), or of the rats has reached a negative reward limit (lose).

After each move in the game, we form this 5-elements episode and insert it to our memory sequence. In case that our memory sequence size grows beyond a fixed bound we delete elements from its tail to keep it below this bound.

The weights of network N are initialized with random values, so in the beginning N will produce awful results, but if our model parameters are chosen properly, it should converge to a solution of the Bellman Equation, and therefore later experiments are expected to be more truthful. Currently, building model that converge quickly seems to be very difficult and there is still lots of room for improvements in this issue.

The Experience Class

This is the class in which we collect our game episodes (or game experiences) within a memory list. Note that its initialization methods need to get a

1. **model** - a neural network model
2. **max_memory** - maximeal length of episodes to keep. When we reach the maximal lenght of memory, each time we add a new episode, the oldest episode is deleted
3. **discount factor** - this is a special coefficient, usually denoted by γ which is required for the Bellman equation for stochastic environments (in which state transitions are probabilistic). Here is a more practical version of the Bellman equation:

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

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 (s
ee 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 e
lement 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.mem
ory[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
```

The Q-Training Algorithm for Qmaze

Following is the algorithm fortraining a our neural network model to solve the maze. It accepts a keyword argument list. Here are the most significant options:

1. **n_epoch** - Number of training epochs
2. **max_memory** - Maximum number of game experiences we keep in memory (see the **Experince** class above)
3. **data_size** - Number of samples we use in each training epoch. This is the number episodes (or game experiences) which we randomly select from our experiences repository (again, see the **Experience** class above)

In [12]:
```
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 t
he 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_me
mory, 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,)
```

Building a Neural Network Model

Choosing the correct parameters for a suitable model is not easy and requires some experience and many experiments. In the case of a maze we found that:

1. The most suitable activation function is **SReLU** (the S-shaped relu)
2. Our optimizer is **RMSProp**
3. Our loss function is **mse** (Mean Squared Error).

We use two hidden layers, each of size equals to the maze size. The input layer has also the same size as the maze since it accepts the maze stae as input. The output layer size is the same as the number of actions (4 in our case), since its outputs the estimated q-value for each action. (we need to take the action with the maximal q-value for playing in the game).

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
```

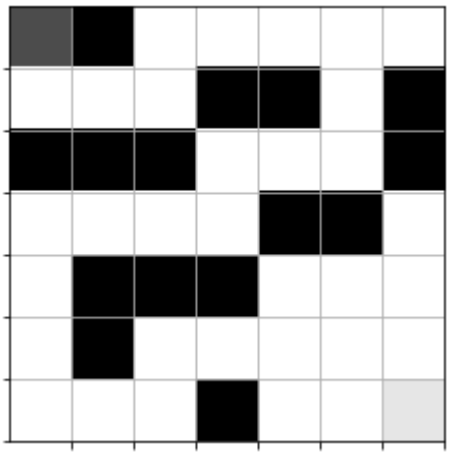
Small Q-test

Lets test our algorithm on the following small (7x7) maze

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)
```

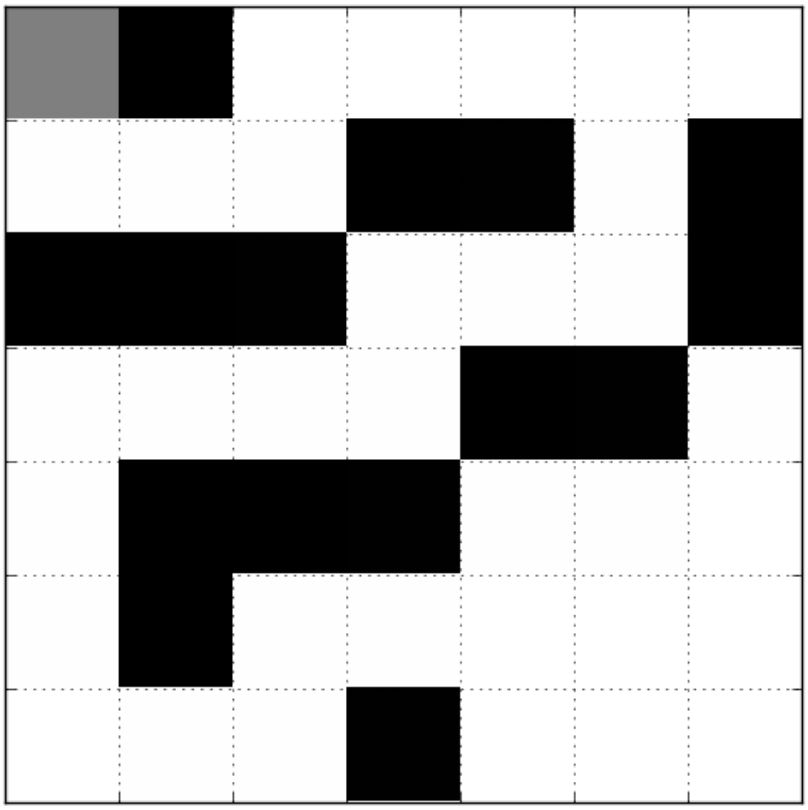
```
Epoch: 000/14999 | Loss: 0.0013 | Episodes: 108 | Win count: 0 | Win rat
e: 0.000 | time: 4.2 seconds
Epoch: 001/14999 | Loss: 0.0043 | Episodes: 102 | Win count: 0 | Win rat
e: 0.000 | time: 8.0 seconds
Epoch: 002/14999 | Loss: 0.0300 | Episodes: 106 | Win count: 0 | Win rat
e: 0.000 | time: 11.9 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 rat
e: 0.000 | time: 17.5 seconds
Epoch: 005/14999 | Loss: 0.0170 | Episodes: 105 | Win count: 1 | Win rat
e: 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 rat
e: 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 rat
e: 0.000 | time: 30.1 seconds
Epoch: 012/14999 | Loss: 0.0120 | Episodes: 109 | Win count: 5 | Win rat
e: 0.000 | time: 34.1 seconds
Epoch: 013/14999 | Loss: 0.0088 | Episodes: 105 | Win count: 5 | Win rat
e: 0.000 | time: 38.0 seconds
Epoch: 014/14999 | Loss: 0.0168 | Episodes: 110 | Win count: 5 | Win rat
e: 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 rat
e: 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 rat
e: 0.000 | time: 51.0 seconds
Epoch: 022/14999 | Loss: 0.0066 | Episodes: 103 | Win count: 10 | Win rat
e: 0.000 | time: 54.8 seconds
Epoch: 023/14999 | Loss: 0.0021 | Episodes: 106 | Win count: 10 | Win rat
e: 0.000 | time: 58.6 seconds
Epoch: 024/14999 | Loss: 0.0015 | Episodes: 106 | Win count: 10 | Win rat
e: 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 rat
e: 0.458 | time: 66.8 seconds
Epoch: 027/14999 | Loss: 0.0009 | Episodes: 101 | Win count: 11 | Win rat
e: 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 rat
e: 0.500 | time: 71.6 seconds
Epoch: 030/14999 | Loss: 0.0029 | Episodes: 102 | Win count: 13 | Win rat
e: 0.458 | time: 75.3 seconds
Epoch: 031/14999 | Loss: 0.0448 | Episodes: 99 | Win count: 14 | Win rat
e: 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 rat
e: 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 rat
e: 0.667 | time: 80.9 seconds
Epoch: 040/14999 | Loss: 0.0039 | Episodes: 20 | Win count: 23 | Win rat
e: 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 rat
e: 0.667 | time: 85.7 seconds
Epoch: 044/14999 | Loss: 0.0015 | Episodes: 13 | Win count: 26 | Win rat
e: 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 rat
e: 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 rat
e: 0.833 | time: 91.0 seconds
Epoch: 049/14999 | Loss: 0.0021 | Episodes: 10 | Win count: 31 | Win rat
e: 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 rat
e: 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 rat
e: 0.958 | time: 97.4 seconds
Epoch: 058/14999 | Loss: 0.0012 | Episodes: 19 | Win count: 40 | Win rat
e: 0.958 | time: 98.1 seconds
Epoch: 059/14999 | Loss: 0.0021 | Episodes: 21 | Win count: 41 | Win rat
e: 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 rat
e: 0.958 | time: 100.0 seconds
Epoch: 062/14999 | Loss: 0.0044 | Episodes: 20 | Win count: 44 | Win rat
e: 0.958 | time: 100.7 seconds
Epoch: 063/14999 | Loss: 0.0012 | Episodes: 44 | Win count: 45 | Win rat
e: 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 rat
e: 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
```

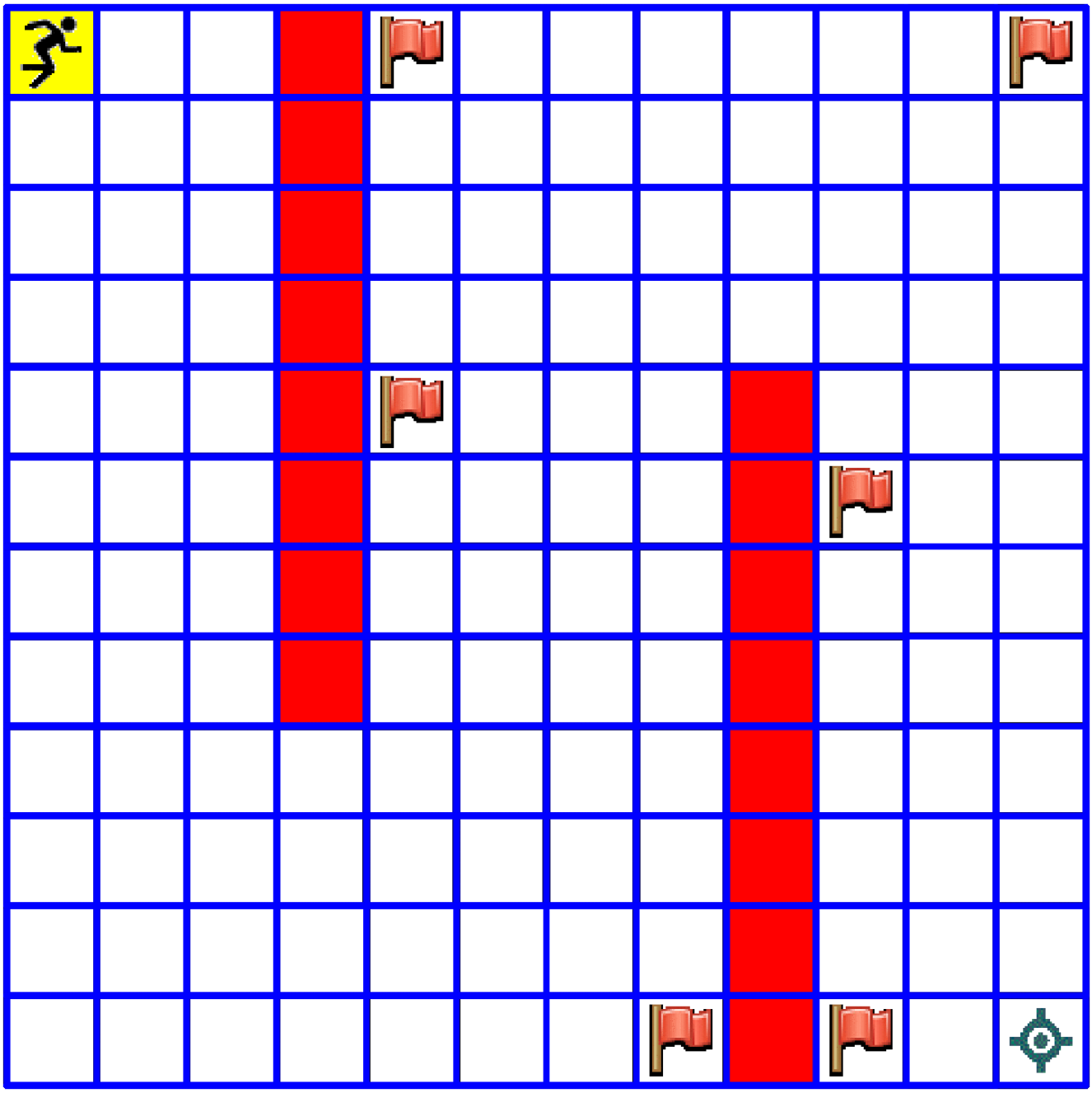
Out[15]: 103.995127

It took less than two minutes to train the model. We can now simulate a few games with this model and visualize them with matplotlib and even create gifs for the solutions (more on that in next versions of the notebook ...)



Additional Ideas for Course Projects

1. A more challenging maze problem is a cat/mouse chase puzzle in which the blocked cells, as well as the mouse are moving in time. This is the type of problems in which classical algorithms are starting to get rough and deep learning techniques could provide better answers.
2. There are plenty of variations on the above type of games. We can for example add a cheese to the maze and then we have a double chase scene: the cat chases the mouse, and the mouse chases the moving cheese. This is also known as the Theseus and Minotaur Maze (http://lafarren.com/theseus-minotaur-solver), and it belongs to the multi-agent type of problems. Current Reinforcement Learning techniques are still in their infancy and there's a vast room for further reseach and development.
3. We can throw several pieces of cheese to the maze and the rat will have to find the shortest route for collecting all of them. Of course, we can also add complications like moving cells and moving cheese, but it seems like the static version is hard enough.
4. There are also a 3-dimensionl versions of these maze problems, which are hard to visualize, and probably much harder to solve, but sooner or later they must be used for testing, validating, and enhancing deep reinforcement learning techniques.
5. In our next notebook we will explore the "Tour De Flags" maze in which an agent has to collect a group of flags and deliver them to a target cell. The agent must find a shortest route for doing so. The agent receives bonus points for collecting each flag, and receives the full award when he arrives to the target cell. Here is a small glimpse to this topic:



In [1]:
```
# Fancy Notebook CSS Style
from nbstyle import *
HTML('<style>%s</style>' % (fancy(),))
```

Out[1]: