Quantum Machine Learning, Homework 1

- Author: National Chiao Tung University 0712238 Yan-Tong Lin
- Lecturer: Dr. Alexey Melnikov

In [5]:

```
import numpy as np
import matplotlib.pyplot as plt
import abc
```

Note for "Projective simulation for artificial intelligence"

- https://arxiv.org/abs/1104.3787 (https://arxiv.org/abs/1104.3787)
  - Projective simulation for artificial intelligence
- https://arxiv.org/pdf/1504.02247 (https://arxiv.org/pdf/1504.02247)
  - Projective simulation with generalization

Model

- Policy
  - $P^{(t)}(a \mid s)$
- How to learn a better policy? By ECM(episodic compositional memory).
- ECM
  - a network of episodes (or clips), which are sequences of remembered percepts and actions.
  - (i) Encounter of percept $s \in S$ which happens with a certain probability $P^{(t)}(s)$. The encounter of percept s triggers the excitation of memory clip $c \in C$ according to a fixed **input-coupler** probability function $I(c \mid s)$.
  - (ii) Random walk through memory/clip space C, which is described by conditional probabilities $p^{(t)}(c' \mid c)$ of calling/exciting clip c' given that c was excited.
  - (iii) Exit of memory through activation of action a, described by a fixed **output-coupler** function $O(a =\mid c)$

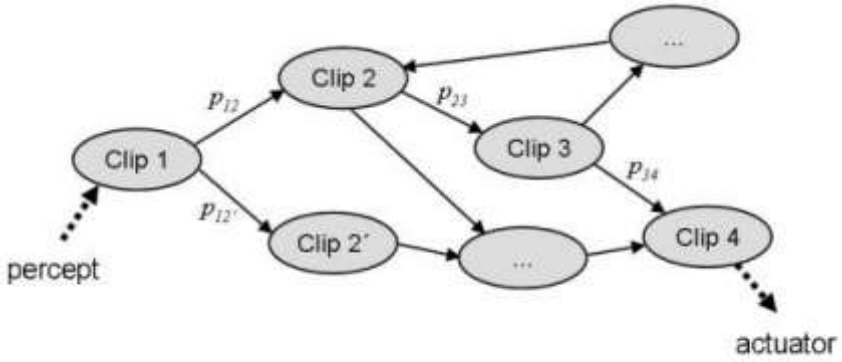


Figure 2 | Model of episodic memory as a network of clips.

- percepts and actions as product of features

- *Percept space:*
  $s \equiv (s_1, s_2, \ldots, s_N) \in S_1 \times \cdots \times S_N \equiv S$, $s_i = 1, \ldots, |S_i|$. The structure of the percept space S, a cartesian product of sets, reflects the compositional (categorical) structure of percepts (objects). For example, s_1 could label the category of shape, s_2 category of color, s_3 category of size, etc. The maximum number of distinguishable input states is given by the product $|S| = |S_1| \cdots |S_N|$.

- *Actuator space:*
  $a \equiv (a_1, a_2, \ldots, a_M) \in A_1 \times \cdots \times A_M \equiv A$, $a_j = 1, \ldots, |A_j|$. The structure of the actuator space A reflects the categories (or, in physics terminology, the degrees of freedom) of the agent's actions. For example a_1 could label the state of motion, a_2 the state of a shutter, a_3 the state of a warning signal, etc. All of this depends on the specification of the agent and the environment. The maximum number of different possible actions is given by the product $|A| = |A_1| \cdots |A_M|$.

- clips are remembered percect/action time-sequences, in the following we consider clips with length 1 for simplicity

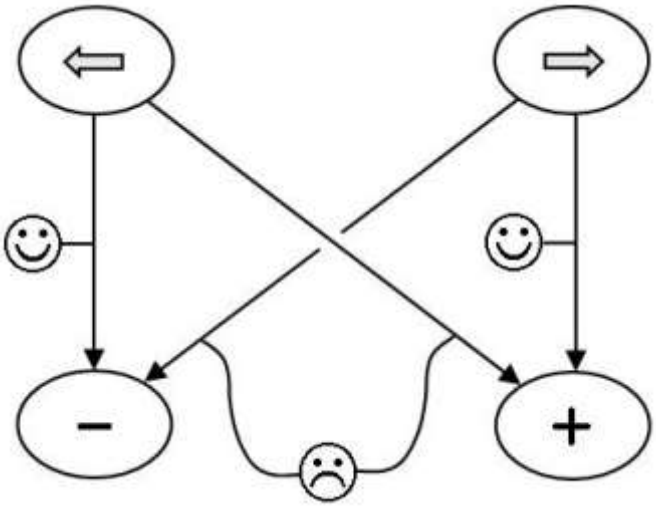
- *Clip space:*
  $c \equiv \left(c^{(1)}, c^{(2)}, \ldots, c^{(L)}\right) \in C$; $c^{(l)} \in \mu(S) \cup \mu(A)$. The index L specifies the length of the clip. A simple example for $L = 2$ is the clip $c = (\mu(s), \mu(a)) \equiv$ (🧐,🎯), which corresponds to a simple percept-action pair. Clips of length $L = 1$ consist of a single remembered percept or action, respectively. In the subsequent examples, we will mainly consider probabilistic networks of such simple clips.

- emotions are remembered rewards

- *Emotion space:*
  $e \equiv (e_1, e_2, \ldots, e_K) \in E_1 \times \cdots \times E_K \equiv E$, $e_k = 1, \ldots, |E_k|$. In the simplest case $K = 1$ and $|E_1| = 2$, with a two-valued emotion state $e_1 \equiv e \in \{$☺, ☹$\}$. Emotional states are *tags*, attached to transitions between different clips in the episodic memory. The state of these tags can be changed through feedback (e.g. reward) from the environment. They are internal parameters and should be distinguished from the reward function itself, which is defined externally. Informally speaking, emotional states are *remembered rewards* for previous actions, they have thus a similar status as the clips.

- true reward function $\Lambda : S \times A \to I \in \mathbb{R}$

The reward function Λ is a mapping from $S \times A$ to $I \subset \mathbb{R}$ (real numbers), where in most subsequent examples we consider the case $I = 0, 1, \ldots, \lambda$. In the simplest case, $\lambda = 1$: If $\Lambda(s, a) = 1$ then the transition $s \to a$ is rewarded; if $\Lambda(s, a) = 0$, it is not rewarded. A rewarded (unrewarded) transition will set certain emotion tags in the episodic memory to ☺ (☹), as discussed previously. We shall also consider situations where the externally defined reward function changes in time, which leads to an adaptation of the flags in the agent's memory.



An Instance of Update Rules

- frequency rule
  - here s, a can be replaced by c_i, c_j to be more general
  - $h^{(1)}(s,a) = 1 \forall s \in S, a \in A$
  - $P^{(n)}(a \mid s) = \frac{h^{(n)}(s,a)}{h^{(n)}(s)}$ where $h^{(n)}(s) = \sum_a h^{(n)}(s,a)$
    - can use softmax to avoid a negative probability
  - let $\lambda^{(n)} = \Lambda(s^{(n)}, a^{(n)})$
  - $h^{(n+1)}(s,a) = h^{(n)}(s,a) + \lambda^{(n)}\delta(s,s^n)\delta(a,a^n) - \gamma(h^{(n)}(s,a) - 1))$
    - the γ part means forgetting part (decay to uniform)
      - not used for non-changing MDPs
    - δ is the delta function (to decide whether the s, a pair is the current pair)
- to model delayed reward: **glow**
  - $h^{(n+1)}(s,a) = h^{(n)}(s,a) + \lambda^{(n)} g^{(n+1)}(s,a) - \gamma(h^{(n)}(s,a) - 1))$
    - $g^{(n+1)}(s,a) = 1$ if s, a was traversed at time step n
    - $g^{(n+1)}(s,a) = (1-\eta) g^{(n)}(s,a)$ otherwise
  - consider delayed reward 100 at time step 3 when a, b is traversed, the edge c, d was traversed earlier in time step 1 with $\eta = 0.1$ then edge c, d by definition will be rewarded with increase in h value by $0.9^2 \times 100$
- choose of meta-params
  - γ is not required for non-chaning environment (e.g. MDP)
  - In theory, we can choose proper η to make the policy to converge to optimal

Reflection time R

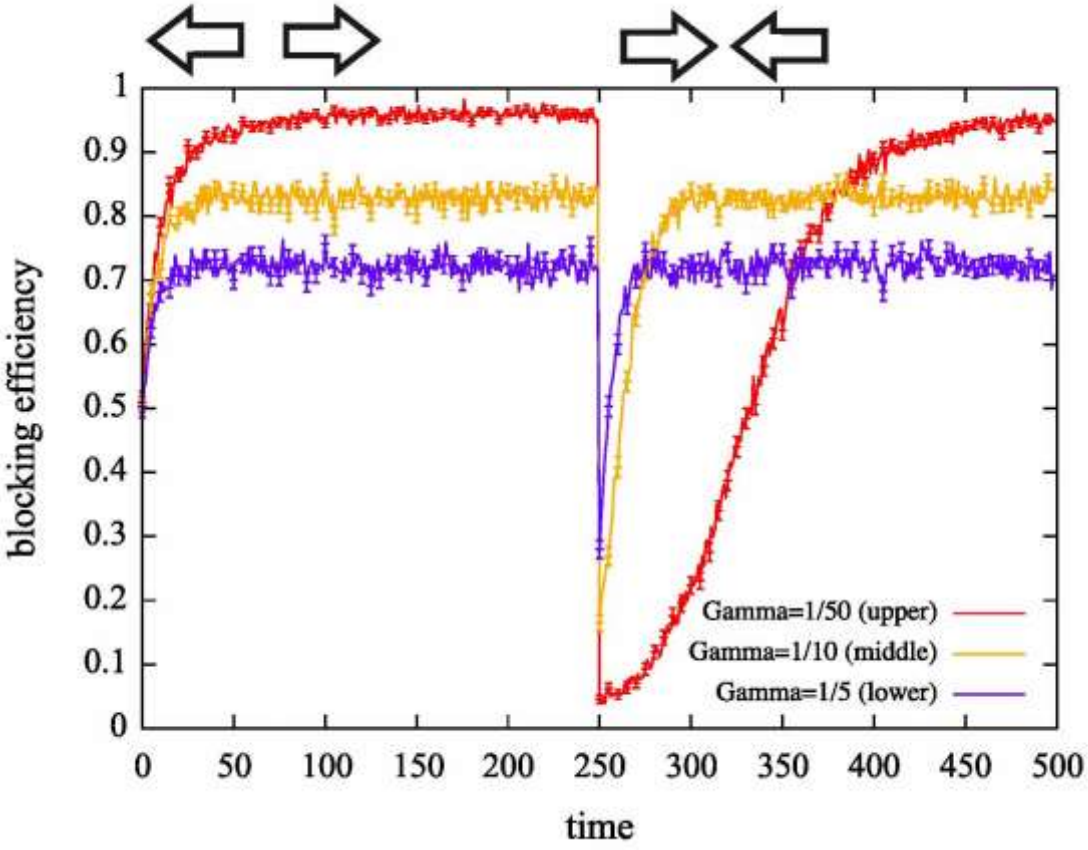
- here $\rightarrow$ means activate with prob
- if $R = 1$
  - $\mu(s) \xrightarrow{p} \mu(a)$
- if $R > 1$
  - same as $\mu(s) \xrightarrow{p} \mu(a)$
  - but if the emotion assigned to $\mu(s), \mu(a)$ is not good enough, repeat til the R_{th} time

Efficiency, the Learning Time / Maximal Efficiency Trade-off

success probability (averaged over different percepts, i.e. symbols shown by the attacker). After the nth round, the blocking efficiency is thus given by

$$r^{(n)} = \sum_{s \in S} P^{(n)}(s) P^{(n)}(a_s^*|s). \qquad (10)$$

In a similar way one can define the *learning time* $\tau(r_{th})$ for a given strategy as the time it takes on average (over an ensemble of identical agents) until the blocking efficiency reaches a certain threshold value r_{th}.



Q1

Go through the material which we considered during the lectures. Think of what is not clear and ask questions.

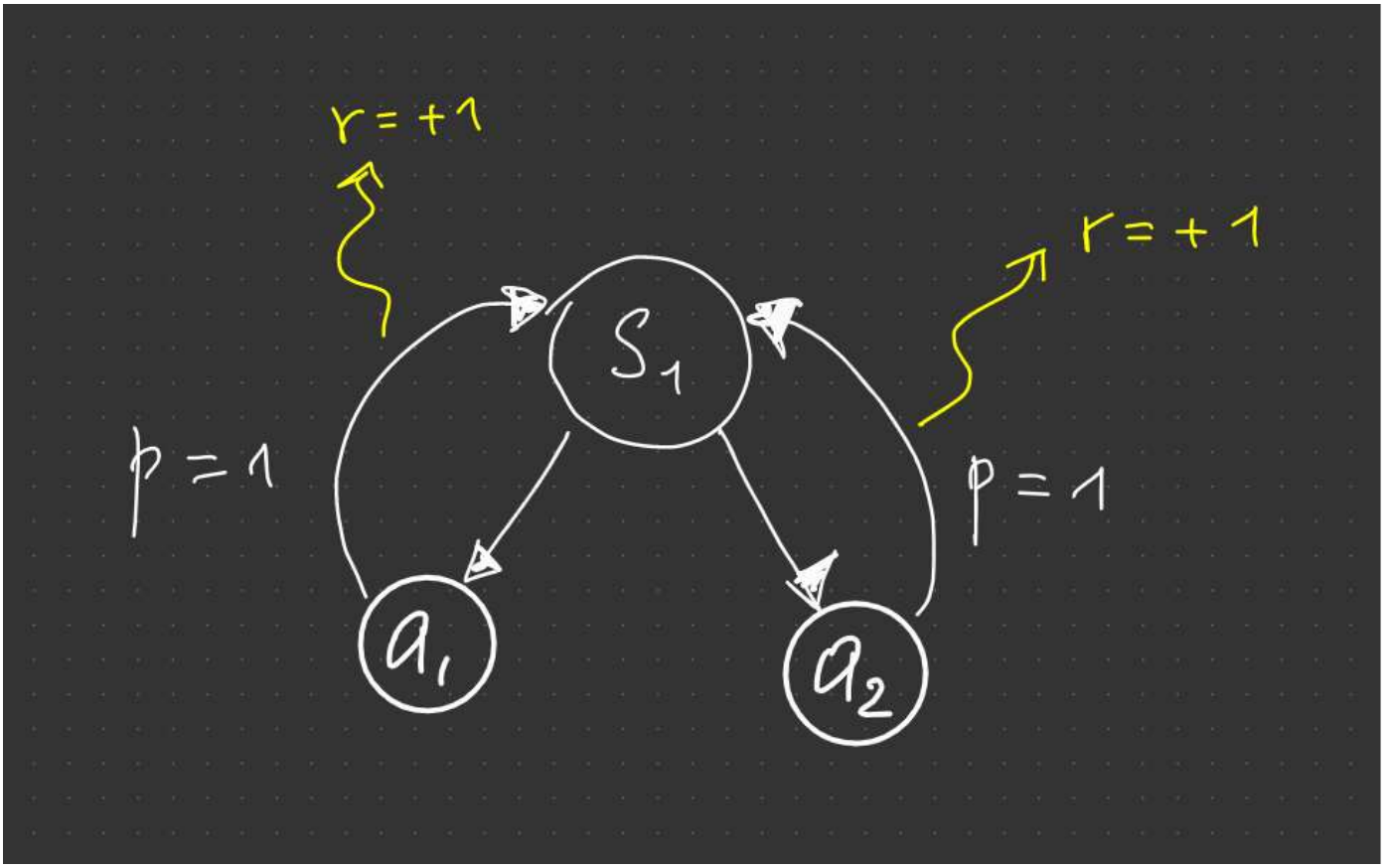
A1

- I go see the original paper for a more rigorous description instead. A note is shown above. If I made any mistake in my note, please let me know. Thanks!
- The equation 5 in the paper "Projective simulation with generalization" is different from the one in the slide by λ^t or λ^{t+1}, is there a reason to define them differently?

- The definition of glow seems confusing at the first glace, I think that adding the time step (i.e. $g^{(n+1)}(s,a) = 1$ if s, a was traversed "at time step n") will make it more clear.

Q2 + Q3

Let us consider the projective simulation (PS) agent. The agent has a memory construction with only percept clips and action clips. No other clips are created in the PS memory. The learning algorithm is such that it is the simplest (the first rule that we studied): there is no forgetting, and there is no glow.



1. How this will this agent perform in an environment described by the following MDP?
2. How will the h-values change in time?
3. What are the asymptotic h-values at t=infinity?
4. What are the asymptotic probabilities of all the actions?
5. Will all agents behave the same, or some agents will have a preference towards a particular action?

Answer the same questions as above given that the h-values are not initialized as h(0)=1 at time step t=0, but h(0)=1 for action a1 and h(0)=2 for action a2.

A2, use simulation

In [6]:

```
class Env(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def get_r():
        return NotImplemented
```

In [7]:

```
class Q23(Env):
    def __init__(self):
        self.r = np.ones((1, 2), dtype=np.float64)
    def get_r(self, s: int, a: int):
        return self.r[s][a]
```

In [8]:

```
class PSAgent():
    def __init__(self, n_s: int, n_a: int, gamma: float = 0.0, eta: float = 1.0):
        """
        init a naive PS agent with only state and action clips
        default gamma = 0.0 (no forgetting)
        default eta = 1.0 (no glowing)
        """
        # params
        self.n_s = n_s
        self.n_a = n_a
        self.gamma = gamma
        self.eta = eta
        # initialize g, h
        self.h = np.ones((self.n_s, self.n_a), dtype=np.float64)
        self.g = np.zeros((self.n_s, self.n_a), dtype=np.float64)

    def update(self, s: int, a: int, r: int):
        # produce an action and get a reward from the environment
        self.g = (1.-self.eta)*self.g
        self.g[s][a] = 1.
        self.h = self.h - self.gamma*(self.h-1) + self.g*r

    def get_p(self, s):
        h_s = self.h[s]
        p_s = h_s / np.sum(h_s)
        return p_s

    def act(self, s):
        return np.random.choice(self.n_a, p=self.get_p(s))
```

In [262]:

```
def simulate(T: int = 1000):
    """
    env: Env, agent: PSAgent, for scalibility if required
    """
    agent = PSAgent(1, 2)
    env = Q23()
    state = 0
    for i in range(T):
        action = agent.act(state)
        reward = env.get_r(state, action)
        agent.update(state, action, reward)
        #print(agent.h)
    return agent
ag = simulate()
ag.get_p(0)
```

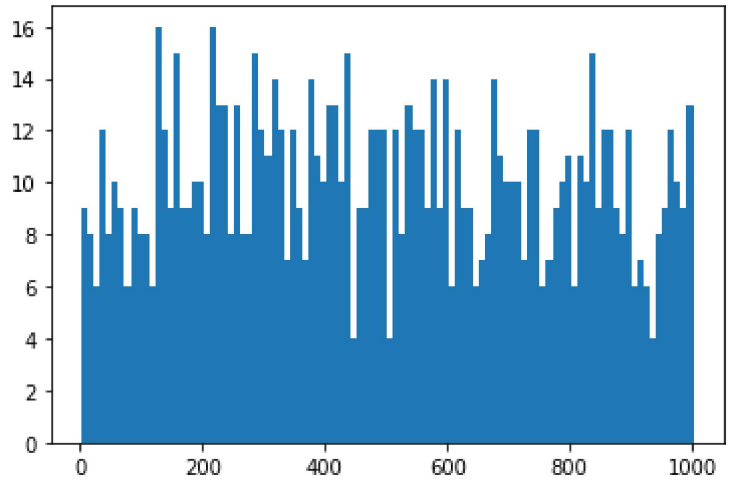
Out[262]:

```
array([0.23053892, 0.76946108])
```

In [10]:

```
def get_distribution_of_hs(T: int = 1000):
    """
    to plot a histogram of each
    """
    x = []
    for i in range(T):
        ag = simulate()
        x += [ag.h[0][0]]
    plt.hist(x, bins = 100)
    plt.show()
    return x

get_distribution_of_hs()
```



A2 + A3, use dynamic programming

- let $P(i, j)$ be Pr(do i+j-2 times and end up with $h_{0,0} = i, h_{0,1} = j$)
- $P(1, 1) = 1$ or $P(1, 2) = 1$
- $P(a, b) = \frac{a-1}{a+b-1} P(a-1, b) + \frac{b-1}{a+b-1} P(a, b-1)$

1. According to the model description and I will discuss how its h value changes as time goes to infinity
2. The dynamic programming transition shows it
   - $P(a,b) = \frac{a-1}{a+b-1}P(a-1,b) + \frac{b-1}{a+b-1}P(a,b-1)$
3. What are the asymptotic h-values at t=infinity?
   - the figure clearly shows the two cases
   - case 1
     - any h value pairs are equally possible
     - $[h_0] = \frac{1}{2}n + 1$
   - case 2
     - A h value pair with lower $h(0)$ is more possible
     - $[h_0] = \frac{1}{3}n + 1$
4. What are the asymptotic probabilities of all the actions?
   - case 1
     - $\frac{1}{2}, \frac{1}{2}$
   - case 2
     - $\frac{1}{3}, \frac{2}{3}$
5. Will all agents behave the same, or some agents will have a preference towards a particular action?
   - Agents will have their preferences, with uniform prob in case 1, and with higher prob to prefer action 2 in case 2

In [300]:

```
def DP(n, a, b):
    dp = np.zeros((n, n), dtype=np.float64)
    dp[a][b] = 1
    for i in range(a, n):
        for j in range(b, n):
            if(i == a and j == b):
                continue
            dp[i][j] = (i-1)*1.0/(i+j-1)*dp[i-1][j] + (j-1)*1.0/(i+j-1)*dp[i][j-1]
    return dp
```

In [301]:

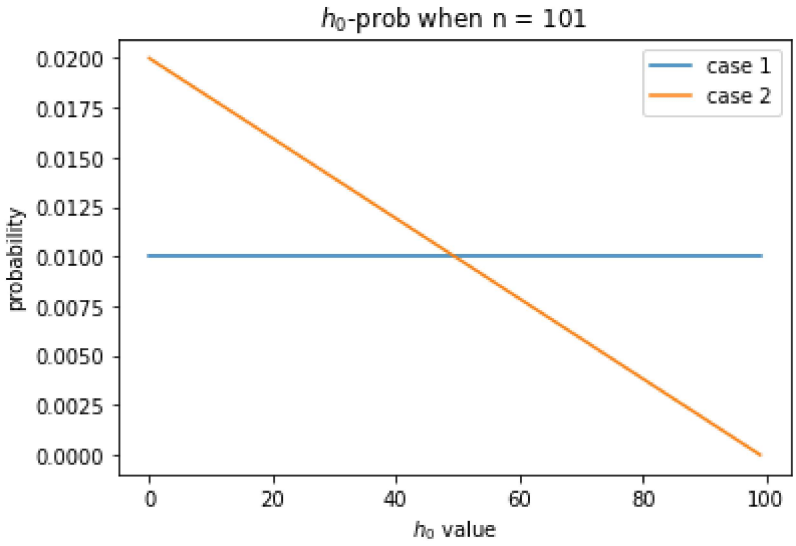
```
dp = DP(101, 1, 1)
dp2 = DP(101, 1, 2)
```

In [302]:

```
x = []
for i in range(1, 101):
    x += [dp[i][101-i]]

x2 = []
for i in range(1, 101):
    x2 += [dp2[i][101-i]]

plt.title("$h_0$-prob when n = 101")
plt.xlabel('$h_0$ value')
plt.ylabel('probability')
plt.plot(x, label='case 1')
plt.plot(x2, label='case 2')
plt.legend()
plt.show()
```



In [308]:

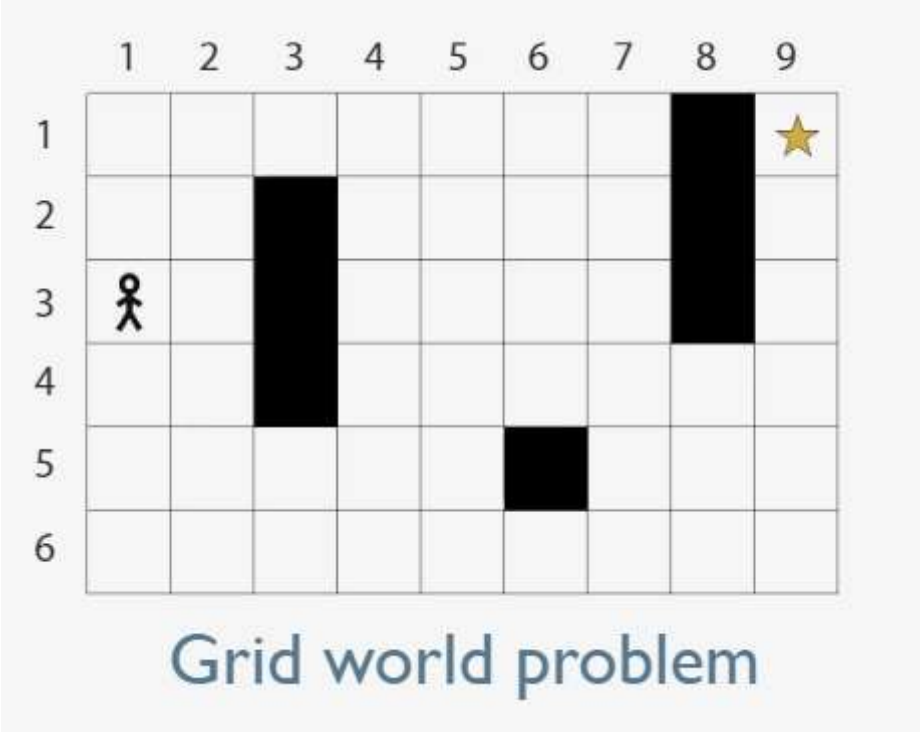
```
exp1 = 0
for i in range(1, 101):
    exp1 += x[i-1]*i
print("case1 [h_0] ~= ", exp1, sep=" ")

exp2 = 0
for i in range(1, 101):
    exp2 += x2[i-1]*i
print("case2 [h_0] ~= ", exp2, sep=" ")
```

```
case1 [h_0] ~=  50.5
case2 [h_0] ~=  33.66666666666665
```

Q4

Consider the grid-world problem.



Program PS agent in this environment (the same maze as in the presentation slides containing 54 positions). Use glow in this environment and no forgetting. Which meta parameter do you find to be optimal? Use any programming language you prefer.

In [248]:

```
class GridWorld(Env):
    def __init__(self):
        """
        (0,0) is top left
        b = {(0,0), (0,1), ...}
        a = {up, down, left, right}
        """
        self.r = np.zeros((54, 4), dtype=np.float64)
        self.r[17][0] = 1. # only state (1,8) with index 8 + action up is of reward 1
        self.nogo = np.zeros((6,9))
        self.dx = [-1, 1, 0, 0]
        self.dy = [0, 0, -1, 1]
        obstacles = [(1,2), (2,2), (3,2), (4,5), (0,7), (1, 7), (2,7)]
        for obs in obstacles:
            self.nogo[obs[0]][obs[1]] = 1

    def to_xy(self, idx):
        return [idx//9, idx%9]

    def to_idx(self, xy):
        return xy[0]*9+xy[1]
    def ok(self, xy):
        return xy[0] >= 0 and xy[0] < 6 and xy[1] >= 0 and xy[1] < 9 and (not self.nogo[xy[

    def get_r(self, s: int, a: int):
        return self.r[s][a]
    def get_next_s(self, s: int, a: int):
        xy = self.to_xy(s)
        nxy = (xy[0]+self.dx[a], xy[1]+self.dy[a])
        if self.ok(nxy):
            return self.to_idx(nxy)
        else:
            return self.to_idx(xy)
```

In [310]:

```
def sim_grid(agent, env, T=100):
    """
    starting state is 0, ending state is always 8
    """
    for i in range(T):
        state = 18
        while state != 8:
            action = agent.act(state)
            reward = env.get_r(state, action)
            next_state = env.get_next_s(state, action)
            agent.update(state, action, reward)
            state = next_state
    # the agent is trained T step after the function call
    return

def get_learning_curve(env, eta, P=100, T=300):
    """
    starting state is 0, ending state is always 8
    """
    curves = np.zeros((P, T), dtype=np.float64)
    for p in range(P):
        agent =  PSAgent(54, 4, 0.0, eta=eta)
        for i in range(T):
            state = 18
            t = 0
            while state != 8:
                t += 1
                action = agent.act(state)
                reward = env.get_r(state, action)
                next_state = env.get_next_s(state, action)
                agent.update(state, action, reward)
                state = next_state
            curves[p][i] = t
    return curves

def show_policy(h, nogo=None):
    """
    0 up, 1 down, 2 left, 3 right
    """
    policy = np.argmax(h, axis=1).reshape(6,9)
    char_policy = [['' for j in range(9)] for i in range(6)]
    arrows = [u"\u2191", u"\u2193", u"\u2190", u"\u2192"]
    for i in range(6):
        for j in range(9):
            if nogo is not None and nogo[i][j]:
                char_policy[i][j] = 'x'
            else:
                char_policy[i][j] = arrows[policy[i][j]]
    for i in range(6):
        for j in range(9):
            print(char_policy[i][j], end=" ")
        print()
    return policy
```

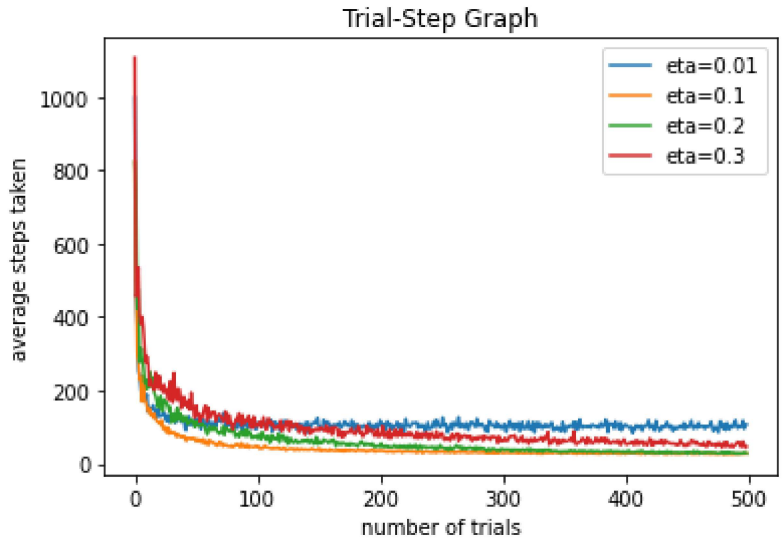
In [314]:

```
curves = get_learning_curve(env, eta=0.1, P=10, T=300)
```

In [322]:

```
etas = [0.01, .1, .2, .3]
env = GridWorld()

plt.title("Trial-Step Graph")
plt.xlabel('number of trials')
plt.ylabel('average steps taken')
for eta in etas:
    curves = get_learning_curve(env, eta=eta, P=50, T=500)
    curve = curves.mean(axis=0)
    plt.plot(curve, label=f"eta={eta}")
plt.legend()
plt.show()
```



In [246]:

```
env = GridWorld()
agent = PSAgent(54, 4, 0.0, eta=.1)
print(np.array(env.nogo, dtype=int))
sim_grid(agent, env, T=100)
pi = show_policy(agent.h, env.nogo)
```

```
[[0 0 0 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
→ → → → ↓ ↓ ↓ x ↑
↑ ↑ x → ↓ → ↓ x ↑
↑ ↓ x → → → ↓ x ↑
→ ↓ x → → → → → ↑
→ → → ↑ ↑ x → → ↑
→ → → ↑ → → ↑ → ↑
```

In [250]:

```
env = GridWorld()
agent = PSAgent(54, 4, gamma=0.0, eta=.1)
print(np.array(env.nogo, dtype=int))
sim_grid(agent, env, T=1000)
pi = show_policy(agent.h, env.nogo)
```

```
[[0 0 0 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
→ → → → → ↓ ↓ x ↑
→ ↑ x ↓ ↓ ↓ ↓ x ↑
↓ ↓ x → → ↓ ↓ x ↑
↓ ↓ x → → → → → ↑
→ → → → ↑ x ↑ ↑ ↑
→ → → ↑ ↑ → → ↑ ↑
```

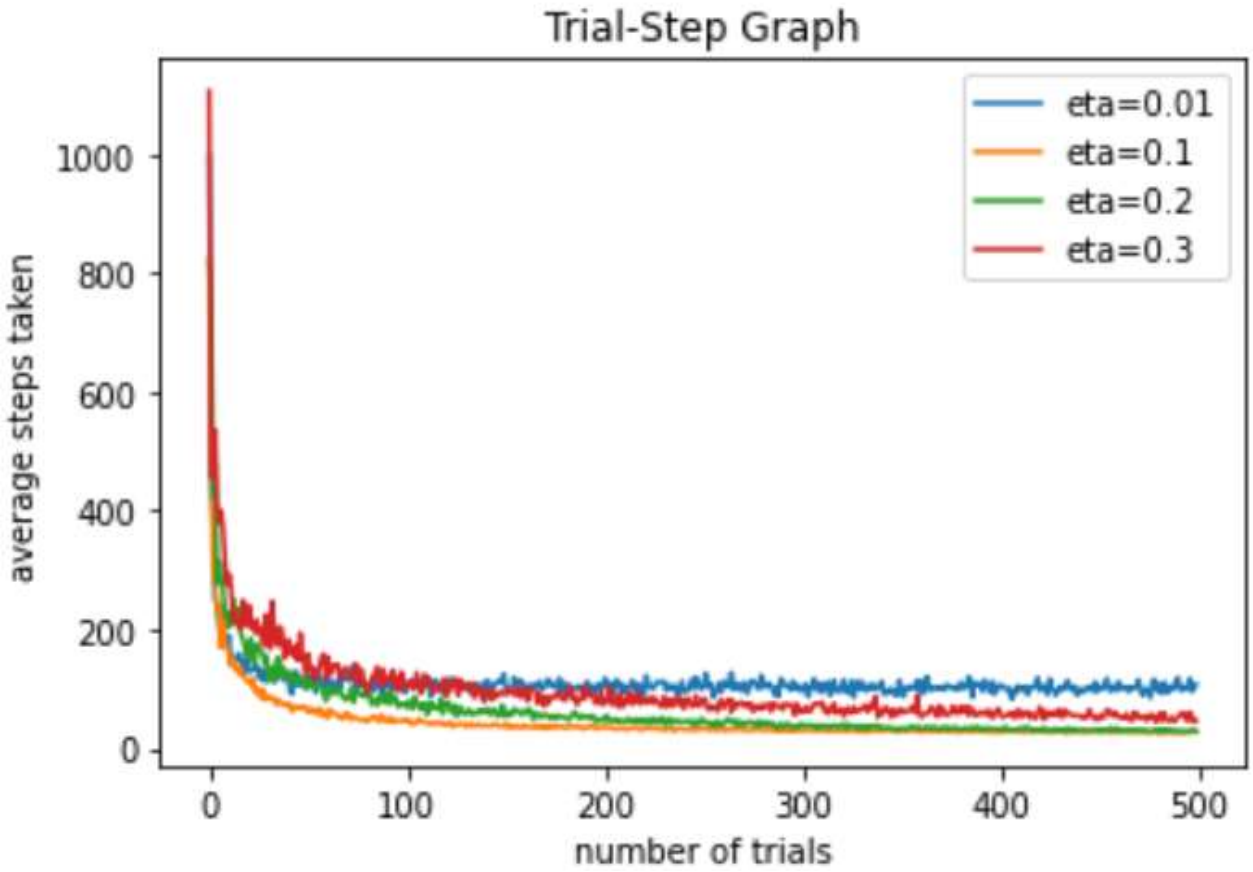
In [252]:

```
env = GridWorld()
agent = PSAgent(54, 4, gamma=0.0, eta=.2)
print(np.array(env.nogo, dtype=int))
sim_grid(agent, env, T=100)
pi = show_policy(agent.h, env.nogo)
```

```
[[0 0 0 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
→ → → → ↓ ↓ ↓ x ↑
→ ↓ x ↓ → ↓ ↓ x ↑
→ ↓ x ↓ → ↓ ↓ x ↑
↓ ↓ x → → → → → ↑
→ → → → ↑ x ↑ → ↑
↑ → ↑ ↑ ↑ → → → ↑
```

A4

- The optimal can be defined on ourself. (confirmed with Dr. Alexey Melnikov)
- In this case, my observation is that agents with different ηs in certain range all converge to η as the number of trials grow.
- For the observation, I would define the optimal as "the η that converges to the optimal policy with high probability with the least number of trials"
- In practice, I plot a $\text{number of trials - average steps}$ graph to show which value among $\{0.01, 0.1, 0.2, 0.3\}$ is optimal
  -



    - note: the population is set to 50
  - By the graph, it seems that $\eta = 0.1, 0.2$ are good
  - My observation is that agents with larger η converge to smaller value, but takes more trials.
- Also, an example of the policy (from its h-values) of an $\eta = 0.2$ agent after 100 trials
  -

In [252]:
```
env = GridWorld()
agent = PSAgent(54, 4, gamma=0.0, eta=.2)
print(np.array(env.nogo, dtype=int))
sim_grid(agent, env, T=100)
pi = show_policy(agent.h, env.nogo)
```

```
[[0 0 0 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
→ → → → ↓ ↓ ↓ x ↑
→ ↓ x ↓ → ↓ ↓ x ↑
→ ↓ x ↓ → ↓ ↓ x ↑
↓ ↓ x → → → → → ↑
→ → → → ↑ x ↑ → ↑
↑ → ↑ ↑ ↑ → → → ↑
```

In [ ]:

```
-
```