



# Table of contents

<b>1</b>		<b>3</b>
1.1	. . . . .	3
1.2	. . . . .	3
1.3	. . . . .	4
1.3.1	. . . . .	4
1.3.2	. . . . .	4
<b>2</b>		<b>5</b>
2.0.1	. . . . .	5
<b>3</b>	<b>:</b>	<b>6</b>
3.1	. . . . .	6
3.2	. . . . .	6
3.3	. . . . .	7
3.4	. . . . .	7
<b>4</b>		<b>15</b>
4.1	k . . . . .	15
4.2	. . . . .	15
4.3	10 . . . . .	15
4.4	. . . . .	16
4.5	. . . . .	17
<b>5</b>		<b>18</b>

# 1

:

## 1.1

```
```{python}  
1 + 1  
```
```

2

## 1.2

---

---

|       |
|-------|
| 11/30 |
| 12/5  |
| 12/12 |
| 12/19 |
| 12/26 |
| 1/2   |

---

book

## 1.3

1. git wsl
2. [rstudio quarto](#)
3. [https://github.com/Roku-3/rindoku\\_RL](https://github.com/Roku-3/rindoku_RL) main rstudio

### 1.3.1

```
git
git pull origin HEAD          # github
git add .                     #
git commit -m "edit chapter 2" #
                                # -m
git push origin HEAD          # github
2                               push
git branch main               push
push
```

### 1.3.2

github

## 2

### 2.0.1

197X

## 3 :

/

### 3.1

---

(policy)  
(reward)  
(value function)  
(model)

---

### 3.2

### 3.3

0~1                      0                      1                      0.5

$$V(s_t) \leftarrow V(s_t) + \alpha [V(s_{t+1}) - V(s_t)]$$

**TD** ; temporal-difference learning

### 3.4

## 1.1 self play

```

```{python}
'''
    import numpy as np
    import pickle
'''

'''
'\nimport numpy as np\nimport pickle\n'

```{python}
'''

BOARD_ROWS = 3
BOARD_COLS = 3
BOARD_SIZE = BOARD_ROWS * BOARD_COLS

class State:
    def __init__(self):
        self.data = np.zeros((BOARD_ROWS, BOARD_COLS))
        self.winner = None
        self.hashVal = None
        self.end = None

```

```

def getHash(self):
    if self.hashVal is None:
        self.hashVal = 0
        for i in self.data.reshape(BOARD_ROWS * BOARD_COLS):
            if i == -1:
                i = 2
            self.hashVal = self.hashVal * 3 + i
        return int(self.hashVal)

def isEnd(self):
    if self.end is not None:
        return self.end
    results = []
    for i in range(0, BOARD_ROWS):
        results.append(np.sum(self.data[i, :]))
    for i in range(0, BOARD_COLS):
        results.append(np.sum(self.data[:, i]))

    results.append(0)
    for i in range(0, BOARD_ROWS):
        results[-1] += self.data[i, i]
    results.append(0)
    for i in range(0, BOARD_ROWS):
        results[-1] += self.data[i, BOARD_ROWS - 1 - i]

    for result in results:
        if result == 3:
            self.winner = 1
            self.end = True
            return self.end
        if result == -3:
            self.winner = -1
            self.end = True
            return self.end

    sum = np.sum(np.abs(self.data))
    if sum == BOARD_ROWS * BOARD_COLS:
        self.winner = 0
        self.end = True
        return self.end

    self.end = False
    return self.end

```



```

def nextState(self, i, j, symbol):
    newState = State()
    newState.data = np.copy(self.data)
    newState.data[i, j] = symbol
    return newState

# print board
def show(self):
    for i in range(0, BOARD_ROWS):
        print('-----')
        out = '| '
        for j in range(0, BOARD_COLS):
            if self.data[i, j] == 1:
                token = '*'
            if self.data[i, j] == 0:
                token = '0'
            if self.data[i, j] == -1:
                token = 'x'
            out += token + ' | '
        print(out)
    print('-----')

def getAllStatesImpl(currentState, currentSymbol, allStates):
    for i in range(0, BOARD_ROWS):
        for j in range(0, BOARD_COLS):
            if currentState.data[i][j] == 0:
                newState = currentState.nextState(i, j, currentSymbol)
                newHash = newState.getHash()
                if newHash not in allStates.keys():
                    isEnd = newState.isEnd()
                    allStates[newHash] = (newState, isEnd)
                    if not isEnd:
                        getAllStatesImpl(newState, -currentSymbol, allStates)

def getAllStates():
    currentSymbol = 1
    currentState = State()
    allStates = dict()
    allStates[currentState.getHash()] = (currentState, currentState.isEnd())
    getAllStatesImpl(currentState, currentSymbol, allStates)
    return allStates

```

```

allStates = getAllStates()

class Judge:
    def __init__(self, player1, player2, feedback=True):
        self.p1 = player1
        self.p2 = player2
        self.feedback = feedback
        self.currentPlayer = None
        self.p1Symbol = 1
        self.p2Symbol = -1
        self.p1.setSymbol(self.p1Symbol)
        self.p2.setSymbol(self.p2Symbol)
        self.currentState = State()
        self.allStates = allStates

    def giveReward(self):
        if self.currentState.winner == self.p1Symbol:
            self.p1.feedReward(1)
            self.p2.feedReward(0)
        elif self.currentState.winner == self.p2Symbol:
            self.p1.feedReward(0)
            self.p2.feedReward(1)
        else:
            self.p1.feedReward(0.1)
            self.p2.feedReward(0.5)

    def feedCurrentState(self):
        self.p1.feedState(self.currentState)
        self.p2.feedState(self.currentState)

    def reset(self):
        self.p1.reset()
        self.p2.reset()
        self.currentState = State()
        self.currentPlayer = None

    def play(self, show=False):
        self.reset()
        self.feedCurrentState()
        while True:
            if self.currentPlayer == self.p1:
                self.currentPlayer = self.p2
            else:

```

```

        self.currentPlayer = self.p1
    if show:
        self.currentState.show()
    [i, j, symbol] = self.currentPlayer.takeAction()
    self.currentState = self.currentState.nextState(i, j, symbol)
    hashValue = self.currentState.getHash()
    self.currentState, isEnd = self.allStates[hashValue]
    self.feedCurrentState()
    if isEnd:
        if self.feedback:
            self.giveReward()
        return self.currentState.winner

# AI player
class Player:
    def __init__(self, stepSize = 0.1, exploreRate=0.1):
        self.allStates = allStates
        self.estimations = dict()
        self.stepSize = stepSize
        self.exploreRate = exploreRate
        self.states = []

    def reset(self):
        self.states = []

    def setSymbol(self, symbol):
        self.symbol = symbol
        for hash in self.allStates.keys():
            (state, isEnd) = self.allStates[hash]
            if isEnd:
                if state.winner == self.symbol:
                    self.estimations[hash] = 1.0
                else:
                    self.estimations[hash] = 0
            else:
                self.estimations[hash] = 0.5

    def feedState(self, state):
        self.states.append(state)

    def feedReward(self, reward):
        if len(self.states) == 0:
            return

```

```

        self.states = [state.getHash() for state in self.states]
        target = reward
        for latestState in reversed(self.states):
            value = self.estimatedions[latestState] + self.stepSize * (target - self.estimatedions[latestState])
            self.estimatedions[latestState] = value
            target = value
        self.states = []

def takeAction(self):
    state = self.states[-1]
    nextStates = []
    nextPositions = []
    for i in range(BOARD_ROWS):
        for j in range(BOARD_COLS):
            if state.data[i, j] == 0:
                nextPositions.append([i, j])
                nextStates.append(state.nextState(i, j, self.symbol).getHash())
    if np.random.binomial(1, self.exploreRate):
        np.random.shuffle(nextPositions)
        self.states = []
        action = nextPositions[0]
        action.append(self.symbol)
        return action

    values = []
    for hash, pos in zip(nextStates, nextPositions):
        values.append((self.estimatedions[hash], pos))
    np.random.shuffle(values)
    values.sort(key=lambda x: x[0], reverse=True)
    action = values[0][1]
    action.append(self.symbol)
    return action

def savePolicy(self):
    fw = open('optimal_policy_' + str(self.symbol), 'wb')
    pickle.dump(self.estimatedions, fw)
    fw.close()

def loadPolicy(self):
    fr = open('optimal_policy_' + str(self.symbol), 'rb')
    self.estimatedions = pickle.load(fr)
    fr.close()

```

```

# | 1 | 2 | 3 |
# | 4 | 5 | 6 |
# | 7 | 8 | 9 |
class HumanPlayer:
    def __init__(self, stepSize = 0.1, exploreRate=0.1):
        self.symbol = None
        self.currentState = None
        return
    def reset(self):
        return
    def setSymbol(self, symbol):
        self.symbol = symbol
        return
    def feedState(self, state):
        self.currentState = state
        return
    def feedReward(self, reward):
        return
    def takeAction(self):
        data = int(input("Input your position:"))
        data -= 1
        i = data // int(BOARD_COLS)
        j = data % BOARD_COLS
        if self.currentState.data[i, j] != 0:
            return self.takeAction()
        return (i, j, self.symbol)

def train(epochs=20000):
    player1 = Player()
    player2 = Player()
    judger = Judger(player1, player2)
    player1Win = 0.0
    player2Win = 0.0
    for i in range(0, epochs):
        print("Epoch", i)
        winner = judger.play()
        if winner == 1:
            player1Win += 1
        if winner == -1:
            player2Win += 1
        judger.reset()
    print(player1Win / epochs)
    print(player2Win / epochs)

```

```

        player1.savePolicy()
        player2.savePolicy()

def compete(turns=500):
    player1 = Player(exploreRate=0)
    player2 = Player(exploreRate=0)
    judger = Judger(player1, player2, False)
    player1.loadPolicy()
    player2.loadPolicy()
    player1Win = 0.0
    player2Win = 0.0
    for i in range(0, turns):
        print("Epoch", i)
        winner = judger.play()
        if winner == 1:
            player1Win += 1
        if winner == -1:
            player2Win += 1
        judger.reset()
    print(player1Win / turns)
    print(player2Win / turns)

def play():
    while True:
        player1 = Player(exploreRate=0)
        player2 = HumanPlayer()
        judger = Judger(player1, player2, False)
        player1.loadPolicy()
        winner = judger.play(True)
        if winner == player2.symbol:
            print("Win!")
        elif winner == player1.symbol:
            print("Lose!")
        else:
            print("Tie!")

    train()
    compete()
    play()
    """
...

'\n\nBOARD_ROWS = 3\nBOARD_COLS = 3\nBOARD_SIZE = BOARD_ROWS * BOARD_COLS\n\nclass State:\n

```

4

4.1 k

$k = 1, \dots, k$        $t = 1, \dots, t$        $A_t = 1, \dots, A_t$        $R_t = 1, \dots, R_t$        $X_a = 1, \dots, X_a$

$q_*(a) := \mathbb{E}[X_a]$

$t = 1, \dots, t$        $Q_t(a) = 1, \dots, Q_t(a)$        $q_*(a) = 1, \dots, q_*(a)$

4.2

$Q_t(a) := \frac{t \cdot a}{t \cdot a} = \frac{\sum_{i=1}^{t-1} R_i \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}$

$\mathbb{1}_{A_i=a} = 1$        $\mathbb{1}_{A_i=a} = 0$

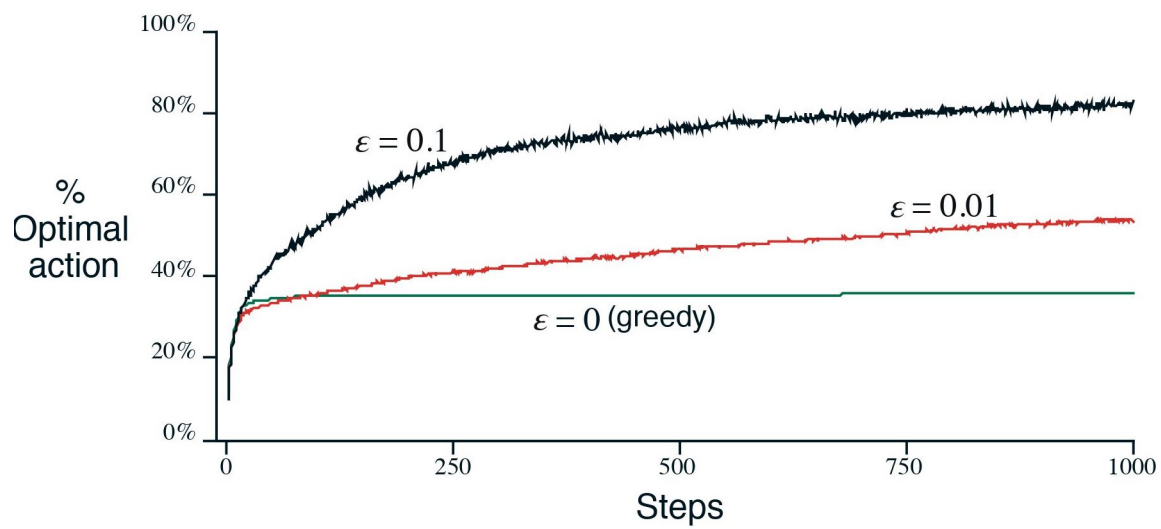
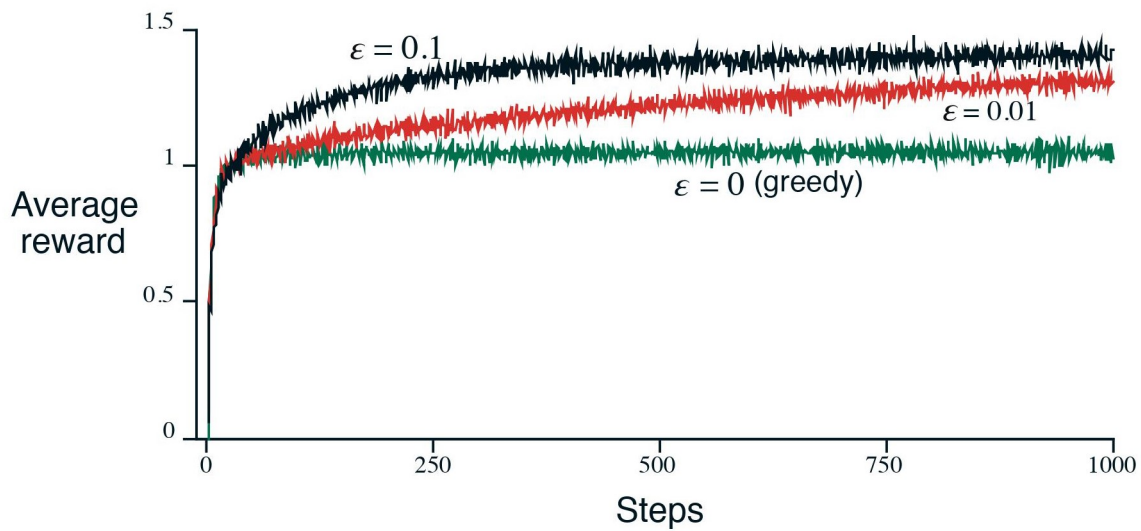
$A_t = \arg \max_a Q_t(a)$

$\arg \max_a Q_t(a) = a$        $\epsilon = 1$        $\epsilon = 0$

4.3 10

2000 k      k=10       $q_*(a) = (N(0, 1))$        $X_a = N(q_*(a), 1)$       1000      1

2000       $\epsilon = 1$       2000



$\epsilon = 0.1$     $\epsilon = 0.01$

$\epsilon = 0.01$

## 4.4

1

$n$     $Q_n$

$$Q_n := \frac{R_1 + R_2 + \dots + R_n}{n - 1}$$

$R_n$     $Q_n$



$$Q_{n+1} = \frac{1}{n} \sum_{i=1}^n R_i = \frac{1}{n} (R_n + \sum_{i=1}^{n-1} R_i) = \frac{1}{n} (R_n + (n-1)Q_n) = Q_n + \frac{1}{n} (R_n - Q_n)$$

$$Q_n \, n$$

$$NewEstimate \leftarrow OldEstimate + StepSize \, [Target - OldEstimate]$$

$$\begin{array}{llll} [Target-OldEstimate] & Target( \quad n \quad ) & StepSize \, 1/n & \alpha \, \alpha_t(a) \\ \epsilon - & k & & \end{array}$$

## 4.5

$$(\quad X_a) \quad , \quad \alpha \in (0,1]$$

$$Q_{n+1} := Q_n + \alpha[R_n - Q_n]$$

$$Q_{n+1}$$

$$Q_{n+1} = Q_n + \alpha[R_n - Q_n] = \alpha R_n + (1 - \alpha)Q_n = \alpha R_n + (1 - \alpha)[\alpha R_{n-1} + (1 - \alpha)Q_{n-1}] = \alpha R_n + (1 - \alpha)\alpha R_{n-1} + (1 - \alpha)^2 Q_{n-1}$$

$$(1-a)^n + \sum_{i=1}^n \alpha (1-\alpha)^{n-i} = 1 \quad Q_{n+1} \quad Q_1 \quad 1-\alpha$$

$$\alpha \quad \text{a n} \quad \alpha_n(a) \quad 1$$

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \quad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$$

```

{python}
print("Hello Python!")

```

Hello Python!

**5**