

FightingGame实验报告

课程名称: 强化学习

报告题目: FightingGame实验报告

学生姓名: 李昊宸 任敏思 马时杰

报告时间: 2022年5月xx日

目 录

[第1章 模型描述 1](#_Toc103879279)

[1.1 问题介绍 1](#_Toc103879280)

[1.2 计算时间 2](#_Toc103879281)

[第2章 研究思路 3](#_Toc103879282)

[2.1 离散化奖励模型 3](#_Toc103879283)

[2.2 连续性状态空间 3](#_Toc103879284)

[2.3 离散化动作空间 3](#_Toc103879285)

[第3章 方法实现 4](#_Toc103879286)

[**3.1 Q-Learning** 4](#_Toc103879287)

[3.1.1 线性函数逼近器 4](#_Toc103879288)

[3.1.2 目标更新 4](#_Toc103879289)

[3.1.3 算法实现 5](#_Toc103879290)

[3.2 DoubleQ-Learning 6](#_Toc103879291)

[3.2.1 线性函数逼近器 6](#_Toc103879292)

[3.2.2 目标更新 6](#_Toc103879293)

[3.2.3 算法实现 7](#_Toc103879294)

[**3.3 SARSA** 9](#_Toc103879295)

[3.3.1 线性函数逼近器 9](#_Toc103879296)

[3.3.2 目标更新 9](#_Toc103879297)

[3.3.3 算法实现 10](#_Toc103879298)

[第4章 实验结果 12](#_Toc103879299)

[**4.1 Q-Learning** 12](#_Toc103879300)

[**4.2 SARSA** 14](#_Toc103879301)

[**4.3 Q-Learning与SARSA的学习过程对比** 16](#_Toc103879302)

# 模型描述

## 问题介绍

格斗游戏是一个典型的实时动作游戏，玩家在游戏中选择一定的动作，在规

定的时间内击败对方角色，赢得胜利。

本任务基于 FightingICE 格斗游戏平台，以已知的固定bot“MctsAi”作为游戏对手，利用课堂上讲述的强化学习方法设计游戏 AI，通过训练学习得到具有一定智能水平的格斗 AI。

将最终学到的强化学习 AI 与 MctsAi 对抗，统计100局中AI的胜率，以及每局结束时双方血量差的平均值，以此作为评判强化学习系统的性能优劣。

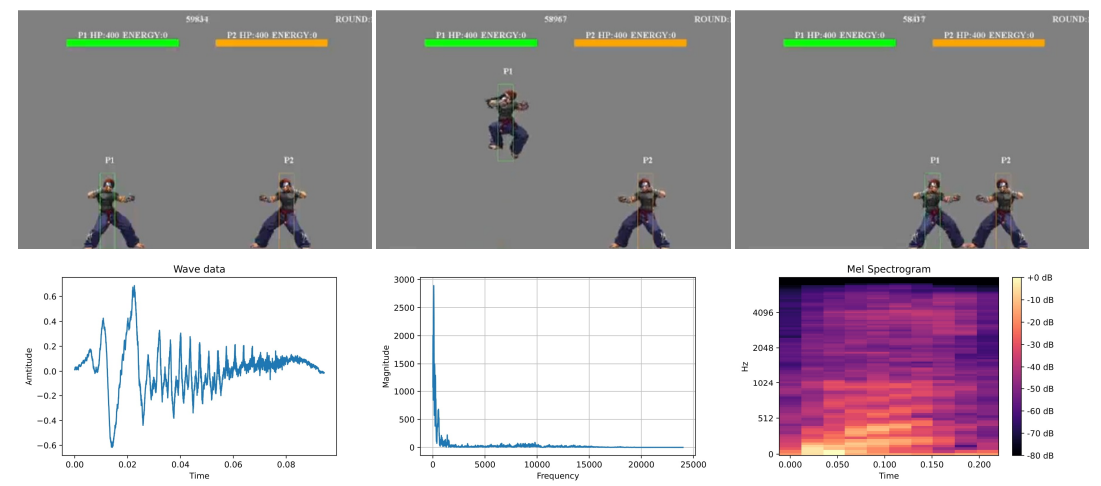


图1 FightingGame动作演示

游戏要求智能体在短时间内反应并决策。智能体的每个基本动作需要耗费一定的帧数，并且可被对手中途打断。

## 计算时间

游戏中的两个AI是同步实时计算，每秒刷新60帧。

在每帧中，AI都需要获取当前的状态输入，并在1帧（16.67ms）内完成动作决策，并将动作输出给状态更新器，更新对战环境的状态后，进行下一帧的运算。

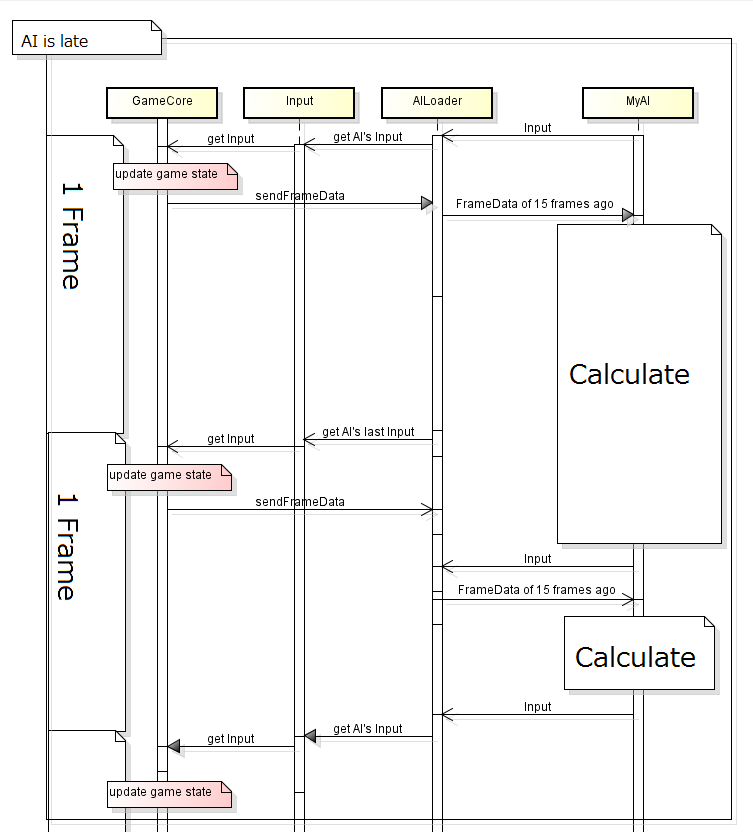
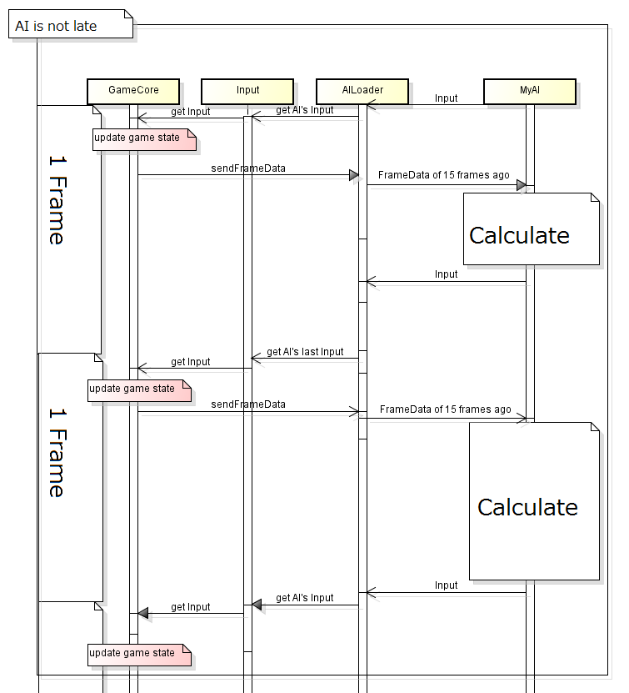


图2 FightingGame框架计算流程

# 研究思路

## 离散化奖励模型

游戏以每秒60帧为标准，将连续时间离散化，即智能体的采样时间。游戏框架给出了状态获取、更新，以及提供当前状态奖励函数的接口：

env = **FightingiceEnv**(port=4242)

obs = env.**reset**(env\_args=env\_args)

……

#act为选择的动作

new\_obs, reward, done, info = env.**step**(act)

## 连续性状态空间

游戏框架内部，提供了连续性的状态空间向量，包含血量、能量、x轴坐标、y轴坐标、位置状态（空中、地面、下蹲、击倒）等信息。该向量为一个144维的浮点型向量，记录了我方与地方的信息。

## 离散化动作空间

游戏框架内采用了离散的有限动作集，各种动作组合共有40种可选项（文档里说有56种，但在实际尝试中仅有40种稳定动作，有点奇怪）

act = **random**.randint(0, 39)

# 方法实现

## **Q-Learning**

首先，采用Q-Learning方法求解该模型。

### 线性函数逼近器

考虑到状态空间是连续的，普通的离散化对于格斗游戏而言，有以下弊端：1）粗粒度离散化，智能体在做决策时，无法精确处理各状态，进而做出不良决策 2）细粒度离散化，状态-动作价值函数表将变得非常稀疏，训练的收敛时间很长。

因此，我们采用线性函数逼近器，通过输入状态直接逼近状态-动作价值函数。

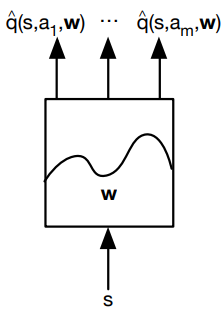


图3 价值函数逼近器示意

#线性逼近器权重

weight=**np**.zeros((40,144))

……

#动作选择 输入为144维的状态向量

act = **np**.argmax(**np**.dot(weight, obs))

### 目标更新

Q-Learning是一种离策略学习：选取行动采用ε-greedy(Q)策略，而更新目标的策略使用greedy(Q)。

迭代中，在每一步长的探索中，智能体通过ε-greedy(Q)策略选取当前状态对应的动作，并据此生成数据。

更新Q函数时，以贪心策略的Q值作为时间差分目标，即：



据此，得到Q函数的迭代公式：



考虑到，实际中我们使用线性函数逼近器逼近Q函数，那么逼近器的更新公式可以由下式给出：



#逼近器更新

delta = reward + gamma \* **np**.max(**np**.dot(weight,new\_obs))

**- np**.dot(obs,weight[act])

weight[act] = weight[act] + alpha \* delta \* obs

### 算法实现

def **train**(num\_alpha, num\_diff\_alpha):

env = **FightingiceEnv**(port=4242)

env\_args =……

#超参设置

gamma = 0.95

alpha = 0.01

epsilon = 0.1

#逼近器参数

weight=**np**.zeros((40,144))

#循环参数

n= 0

RewardData=[]

N = 500

NumWin = 0

Best\_Score = 0

abs\_score = 0

act = **random**.randint(0, 39)

while True:

obs = env.**reset**(env\_args=env\_args)

reward, done, info = 0, False, None

n = n+1

r = 0

while not done:

#参数保存

……

#ε-greedy策略

act = **np**.argmax(**np**.dot(weight, obs))

if **random**.random()<epsilon:

act = **random**.randint(0, 39)

else:

pass

new\_obs, reward, done, info = env.**step**(act)

#逼近器参数更新

if not done:

delta = reward + gamma \* **np**.max(**np**.dot(weight,new\_obs))

- **np**.dot(obs,weight[act])

weight[act] = weight[act] + alpha \* delta \* obs

obs = new\_obs

r = r + reward

#本轮迭代结束

elif info is not None:

**print**("round result: own hp {} vs opp hp {}, you {}".**format**(info[0], info[1],

'win' if info[0]>info[1] else 'lose'),'训练局数',n)

if info[0]>info[1]:

NumWin=NumWin+1

abs\_score = info[0] - info[1]

else:

pass

if n==N:

break

RewardData.**append**(r)

## **DoubleQ-Learning**

随后，采用DoubleQ-Learning方法求解该模型。

### 线性函数逼近器

采用与Q-Learning一致的线性函数逼近器。

### 目标更新

DoubleQ-Learning是一种离策略学习：选取行动采用ε-greedy(Q)策略，而更新目标的策略使用greedy(Q)。

在Q-Learning的迭代中，在每一步长的探索中，智能体通过ε-greedy(Q)策略选取当前状态对应的动作，并据此生成数据。更新Q函数时，以贪心策略的Q值作为时间差分目标。

但是，基于优先样本估计的Q函数，生成的贪心策略会导致最大化偏差。因此，DoubleQ-Learning的核心在于，将样本分为两组，分别定义两个独立的估计，交替的进行更新：使用其中一个Q函数计算max动作，并使用另一个Q函数估计动作的价值，以获得无偏的估计。

据此，得到DoubleQ函数的迭代公式：



考虑到，实际中我们使用线性函数逼近器逼近Q函数，那么逼近器的更新公式可以由下式给出：



#逼近器更新

if **random**.random() < 0.5:

*#更新Q1*

Q1\_act = **np**.argmax(**np**.dot(weight1, new\_obs))

delta = reward + gamma \* **np**.dot(new\_obs, weight2[Q1\_act]) - **np**.dot(obs, weight1[act])

weight1[act] = weight1[act] + alpha \* delta \* obs

else:

*#更新Q2*

Q2\_act = **np**.argmax(**np**.dot(weight2, new\_obs))

delta = reward + gamma \* **np**.dot(new\_obs, weight1[Q2\_act]) - **np**.dot(obs, weight2[act])

weight2[act] = weight2[act] + alpha \* delta \* obs

### 算法实现

def **train**(num\_alpha, num\_diff\_alpha):

env = **FightingiceEnv**(port=4242)

env\_args =……

#超参设置

gamma = 0.95

alpha = 0.01

epsilon = 0.1

#逼近器参数

weight1=**np**.zeros((40,144))

weight2=**np**.zeros((40,144))

#循环参数

n= 0

RewardData=[]

N = 500

NumWin = 0

Best\_Score = 0

abs\_score = 0

act = **random**.randint(0, 39)

while True:

obs = env.**reset**(env\_args=env\_args)

reward, done, info = 0, False, None

n = n+1

r = 0

while not done:

#参数保存

……

#ε-greedy策略

act = **np**.argmax(**np**.dot(weight, obs))

if **random**.random()<epsilon:

act = **random**.randint(0, 39)

else:

pass

new\_obs, reward, done, info = env.**step**(act)

#逼近器参数更新

if not done:

if **random**.random() < 0.5:

*#更新Q1*

Q1\_act = **np**.argmax(**np**.dot(weight1, new\_obs))

delta = reward + gamma \* **np**.dot(new\_obs, weight2[Q1\_act]) - **np**.dot(obs, weight1[act])

weight1[act] = weight1[act] + alpha \* delta \* obs

else:

*#更新Q2*

Q2\_act = **np**.argmax(**np**.dot(weight2, new\_obs))

delta = reward + gamma \* **np**.dot(new\_obs, weight1[Q2\_act]) - **np**.dot(obs, weight2[act])

weight2[act] = weight2[act] + alpha \* delta \* obs

obs = new\_obs

r = r + reward

#本轮迭代结束

elif info is not None:

**print**("round result: own hp {} vs opp hp {}, you {}".**format**(info[0], info[1],

'win' if info[0]>info[1] else 'lose'),'训练局数',n)

if info[0]>info[1]:

NumWin=NumWin+1

abs\_score = info[0] - info[1]

else:

pass

if n==N:

break

RewardData.**append**(r)

## **SARSA**

Q-Learning是离策略学习，在更新时永远选择奖励最高的动作，不考虑带来的其他后果，因此比较激进。为作比较，我们再采用SARSA方法求解该模型。

### 线性函数逼近器

采用与Q-Learning一致的线性逼近器。

### 目标更新

SARSA是一种在策略学习：选取行动和更新目标的策略都使用ε-greedy(Q)。

迭代中，在每一步长的探索中，智能体执行当前状态对应的动作，据此生成数据。

随后，根据策略ε-greedy(Q)采样动作，并根据更新Q，即：



考虑到，实际中我们使用线性函数逼近器逼近Q函数，那么逼近器的更新公式可以由下式给出：



#逼近器更新

delta = reward + gamma \* **np**.dot(new\_obs, weight[explore\_act])

- **np**.dot(obs,weight[act])

weight[act] = weight[act] + alpha \* delta \* obs

### 算法实现

def **train**(num\_alpha, num\_diff\_alpha):

env = **FightingiceEnv**(port=4242)

env\_args =……

#超参设置

gamma = 0.95

alpha = 0.01

epsilon = 0.1

#逼近器参数

weight=**np**.zeros((40,144))

#循环参数

n= 0

RewardData=[]

N = 500

NumWin = 0

Best\_Score = 0

abs\_score = 0

act = **random**.randint(0, 39)

while True:

obs = env.**reset**(env\_args=env\_args)

reward, done, info = 0, False, None

n = n+1

r = 0

while not done:

#参数保存

……

#ε-greedy策略

act = **np**.argmax(**np**.dot(weight, obs))

if **random**.random()<epsilon:

act = **random**.randint(0, 39)

else:

pass

new\_obs, reward, done, info = env.**step**(act)

#逼近器参数更新

if not done:

explore\_act = **np**.argmax(**np**.dot(weight, new\_obs))

if **random**.random() < epsilon:

explore\_act = **random**.randint(0, 39)

else:

pass

delta = reward + gamma \* **np**.dot(new\_obs, weight[explore\_act])

- **np**.dot(obs,weight[act])

weight[act] = weight[act] + alpha \* delta \* obs

obs = new\_obs

r = r + reward

#本轮迭代结束

elif info is not None:

**print**("round result: own hp {} vs opp hp {}, you {}".**format**(info[0], info[1],

'win' if info[0]>info[1] else 'lose'),'训练局数',n)

if info[0]>info[1]:

NumWin=NumWin+1

abs\_score = info[0] - info[1]

else:

pass

if n==N:

break

RewardData.**append**(r)

## **DuelingDQN**

为解决Q-Learning的维数灾问题，DQN方法使用神经网络作为Q函数的逼近器。Dueling DQN考虑将DQN的Q网络分成两部分，即价值函数部分和优势函数部分。

### 网络结构

价值函数仅仅与状态s有关，与具体要采用的动作a无关，记为，其中是公共部分的网络参数，是价值函数独有部分的网络参数。优势函数同时与状态s和动作a有关，记为，其中是优势网络独有的网络参数。

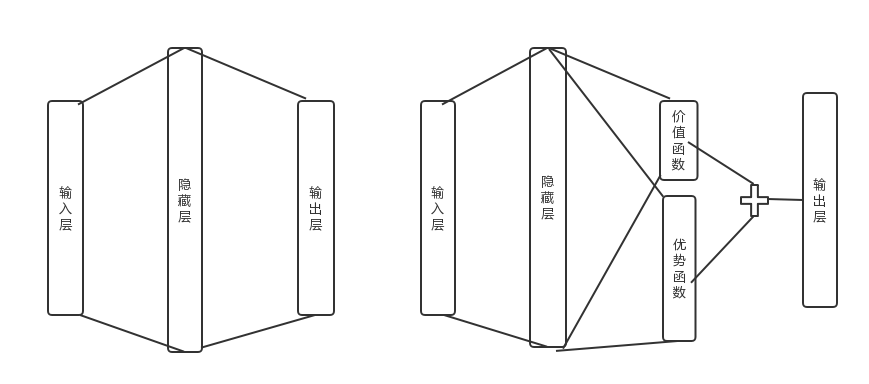


图4 DuelingDQN示意

因此，DuelingDQN网络的状态-动作价值函数由下式给出：



def **\_\_init\_\_**(self, input\_size, output\_size, mem\_len):

**super**(**DQN**, self).**\_\_init\_\_**()

self.input\_size = input\_size

self.output\_size = output\_size

self.memory = **collections**.**deque**(maxlen = mem\_len)

self.net = nn.Sequential(

nn.Linear(self.input\_size, 128),

nn.ReLU(),

nn.Linear(128, 256),

nn.ReLU(),

nn.Linear(256, 128),

nn.ReLU()

)

self.V = nn.Linear(128, 1)

self.A = nn.Linear(128, self.output\_size)

### 目标更新

配合forward，使用BP方法进行参数更新。

def **forward**(self, input):

net\_output = self.net(**input**)

v = self.V(net\_output)

advantage = self.A(net\_output)

advantage = advantage - **torch**.mean(advantage)

q\_value = v + advantage

return q\_value

loss使用传统的MSELoss



其中表示Q\_net，表示滞后更新的Q\_target。

def **train\_net**(Q\_net, Q\_target, optimizer, losses, loss\_list, replay\_time, gamma, batch\_size):

s, a, r, s\_next, done\_flag = Q\_net.sample\_memory(batch\_size)

q\_value = Q\_net(s)

a = **torch**.LongTensor(a)

q\_value = **torch**.gather(q\_value, 1, a)

q\_t = Q\_net(s\_next)

a\_index = **torch**.argmax(q\_t, 1)

a\_index = a\_index.reshape((a\_index.shape[0], 1))

q\_target = Q\_target(s\_next)

q\_target = **torch**.gather(q\_target, 1, a\_index)

q\_target = r + gamma \* q\_target \* done\_flag

loss = losses(q\_target, q\_value)

loss\_list.append(loss)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

### 算法实现

def **train**(num\_alpha, num\_diff\_alpha):

env = **FightingiceEnv**(port=4242)

env\_args = ……

gamma=0.95

alpha=0.01

epsilon=0.1

mem\_len = 30000

load\_parameter = True

learning\_rate = 1e-3

batch\_size = 32

train\_begin = 100

Q\_net = **DQN**(input\_size = 144, output\_size = 40, mem\_len = mem\_len)

Q\_target = **DQN**(input\_size = 144, output\_size = 40, mem\_len = mem\_len)

Q\_target.load\_state\_dict(Q\_net.state\_dict())

if load\_parameter:

**print**('Load parameter!')

Q\_net = **torch**.load(……)

Q\_target.load\_state\_dict(Q\_net.state\_dict())

optimizer = optim.Adam(Q\_net.parameters(), lr = learning\_rate)

losses = nn.MSELoss()

n = 0

p = 0

loss\_list = []

reward\_list = []

N = 500

NumWin = 0

Best\_Score = 0

abs\_score = 0

step\_count = 0

act = **random**.randint(0, 39)

while True:

obs = env.**reset**(env\_args=env\_args)

reward, done, info = 0, False, None

n = n+1

r = 0

epsilon = **max**(0.01, epsilon\*0.999)

while not done:

#参数保存

……

step\_count += 1

act = Q\_net.**sample\_action**(obs, epsilon)

new\_obs, reward, done, info = env.**step**(act)

if done:

done\_flag = 0

new\_obs = [0. for i in **range**(**len**(obs))]

else:

done\_flag = 1

Q\_net.**save\_trans**((obs, act, reward, new\_obs, done\_flag))

if not done and step\_count > train\_begin:

train\_flag = True

**train\_net**(Q\_net, Q\_target, optimizer, losses, loss\_list, 1, gamma, batch\_size)

r = r + reward

obs = new\_obs

if step\_count % 3000 == 0 :

Q\_target.load\_state\_dict(Q\_net.state\_dict())

if not done:

pass

elif info is not None:

**print**("round result: own hp {} vs opp hp {}, you {}".**format**(info[0], info[1],

'win' if info[0]>info[1] else 'lose'),'训练局数',n)

if info[0]>info[1]:

NumWin = NumWin + 1

abs\_score = info[0] - info[1]

else:

pass

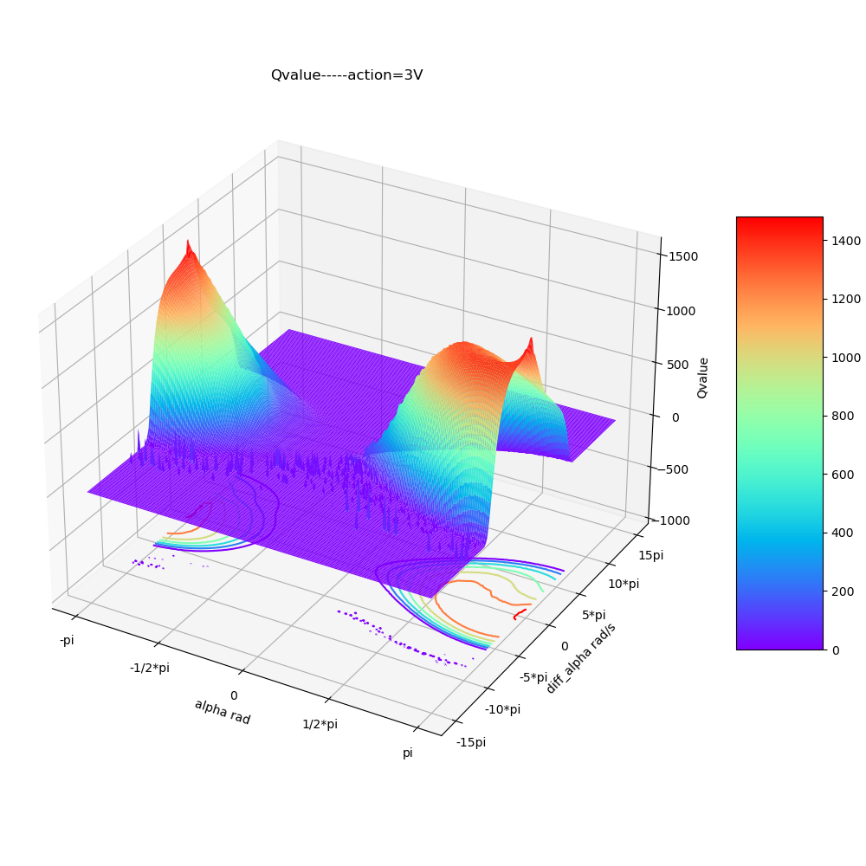
if n == N:

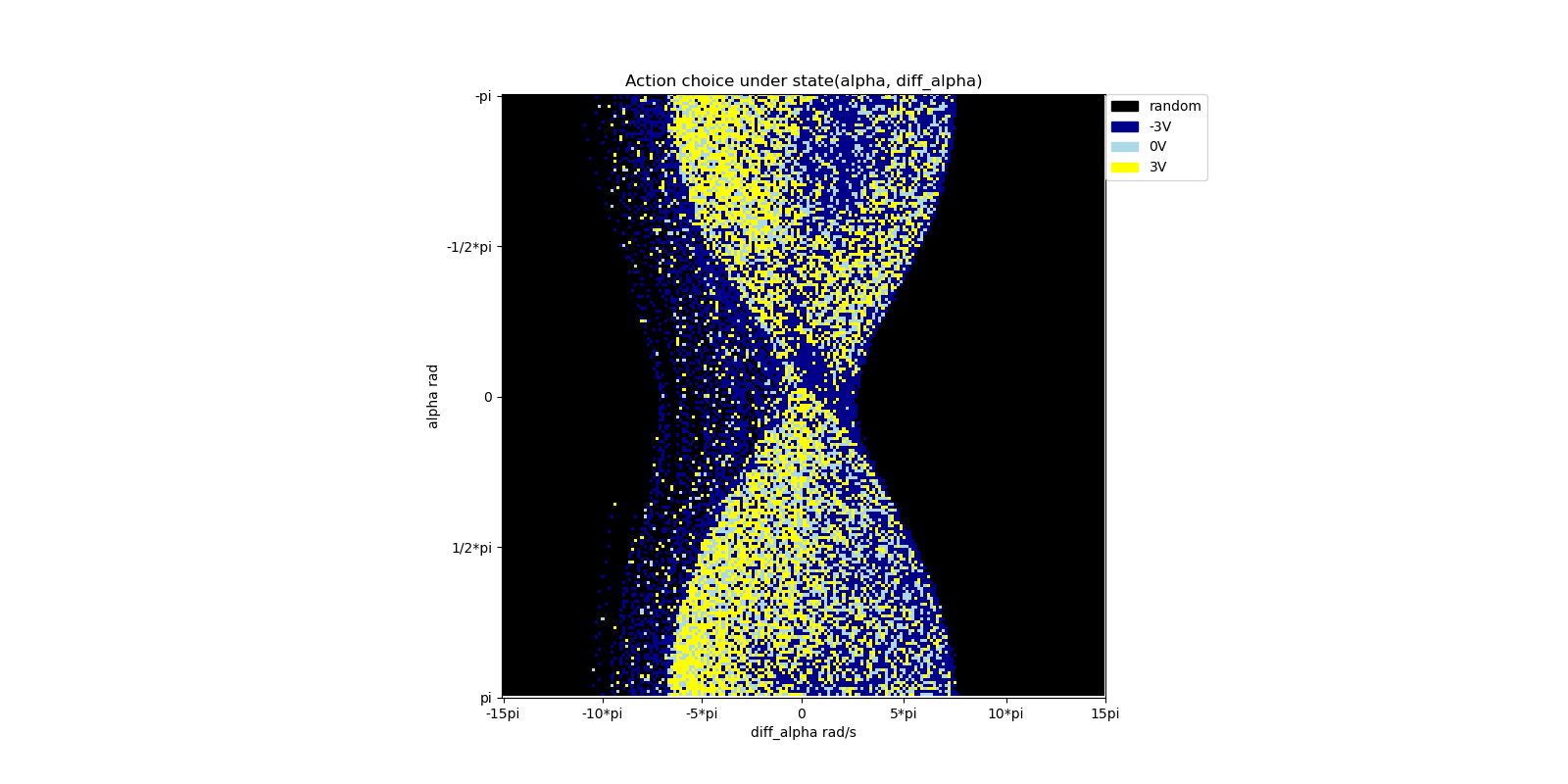
break

# 实验结果（没写）

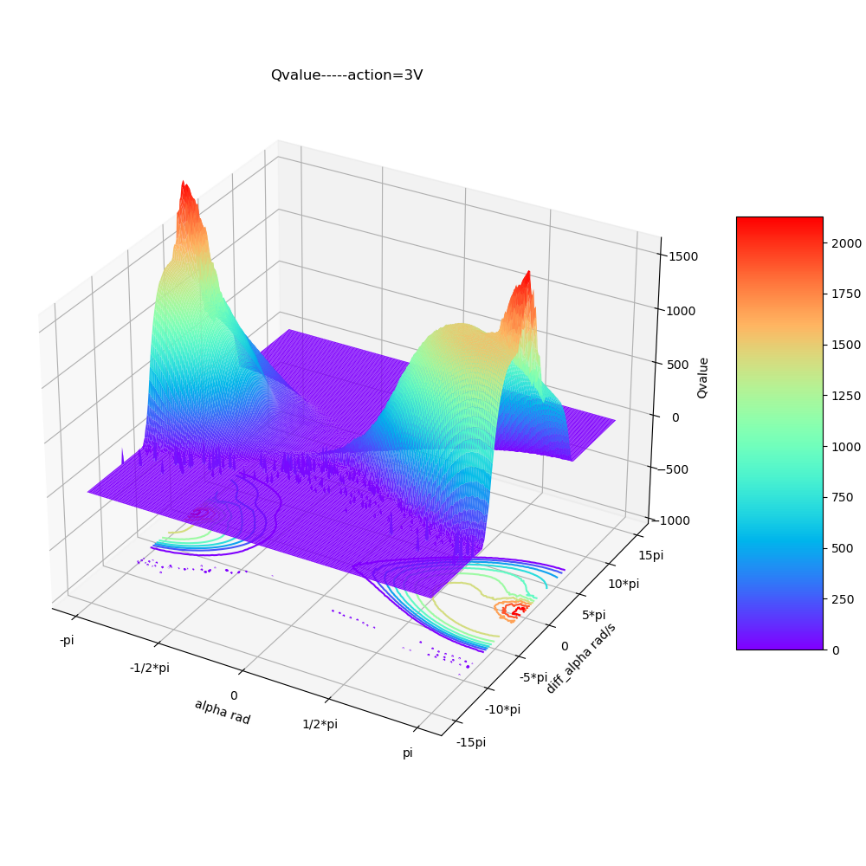
## **Q-Learning**

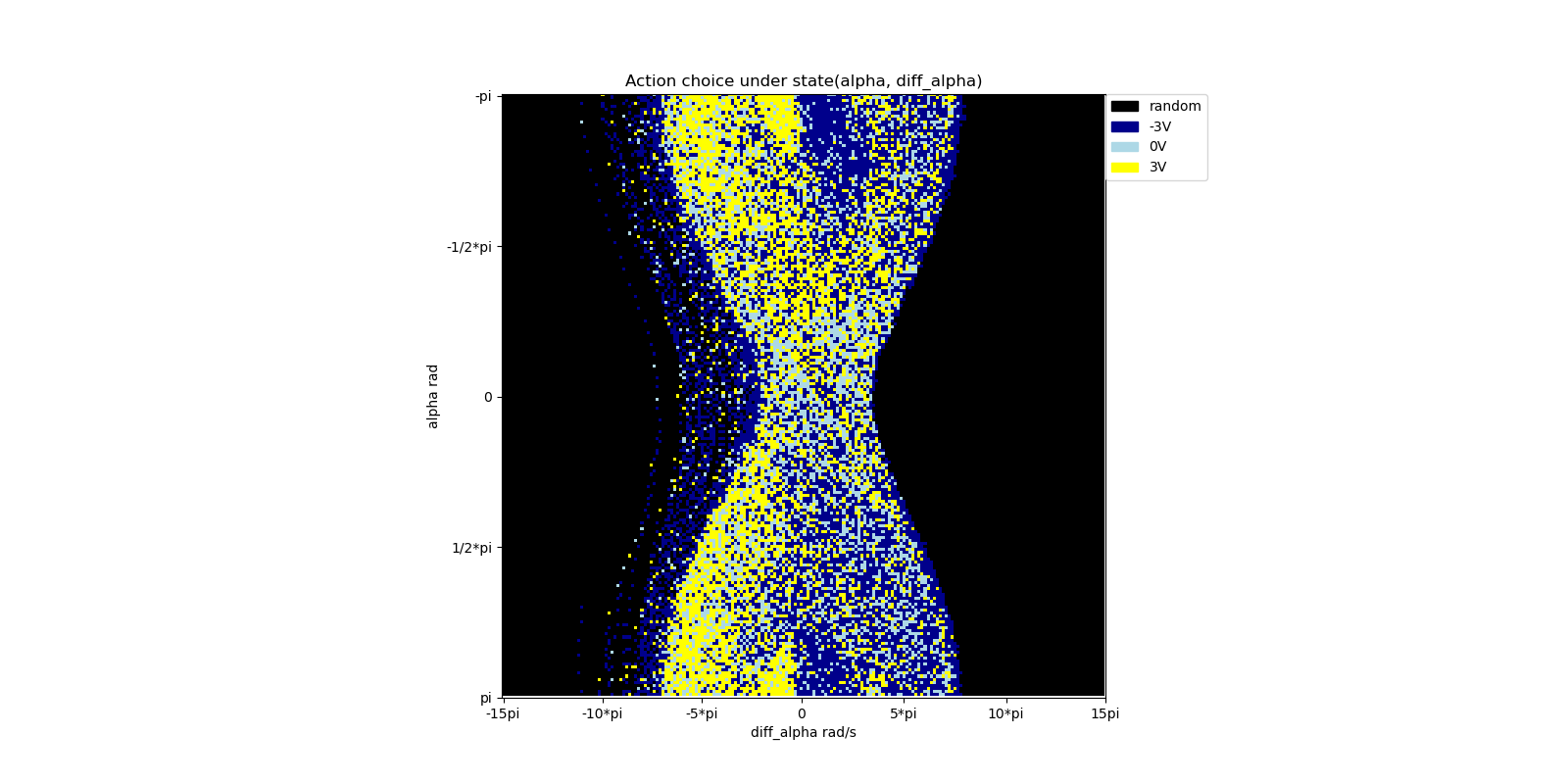
放了几个可以画的图的例子





## **SARSA**





## **Q-Learning与SARSA的学习过程对比**

