MARL

SAEYEON HWANG

REINFORCEMENT LEARNING

2022.05.11

University at Buffalo The State University of New York

University at Buffalo The State University of New York

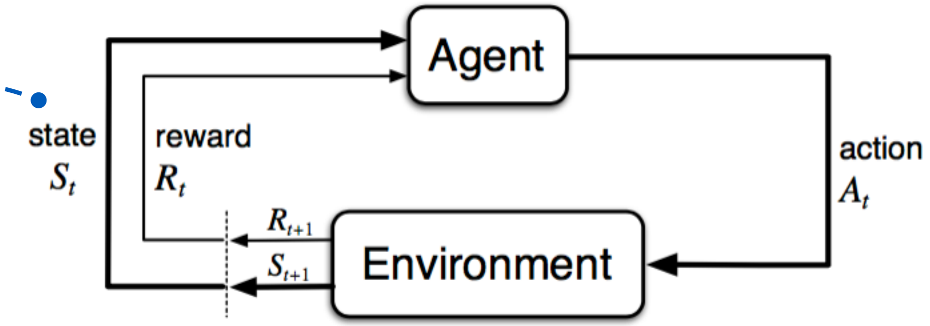
PROJECT DESCRIPTION

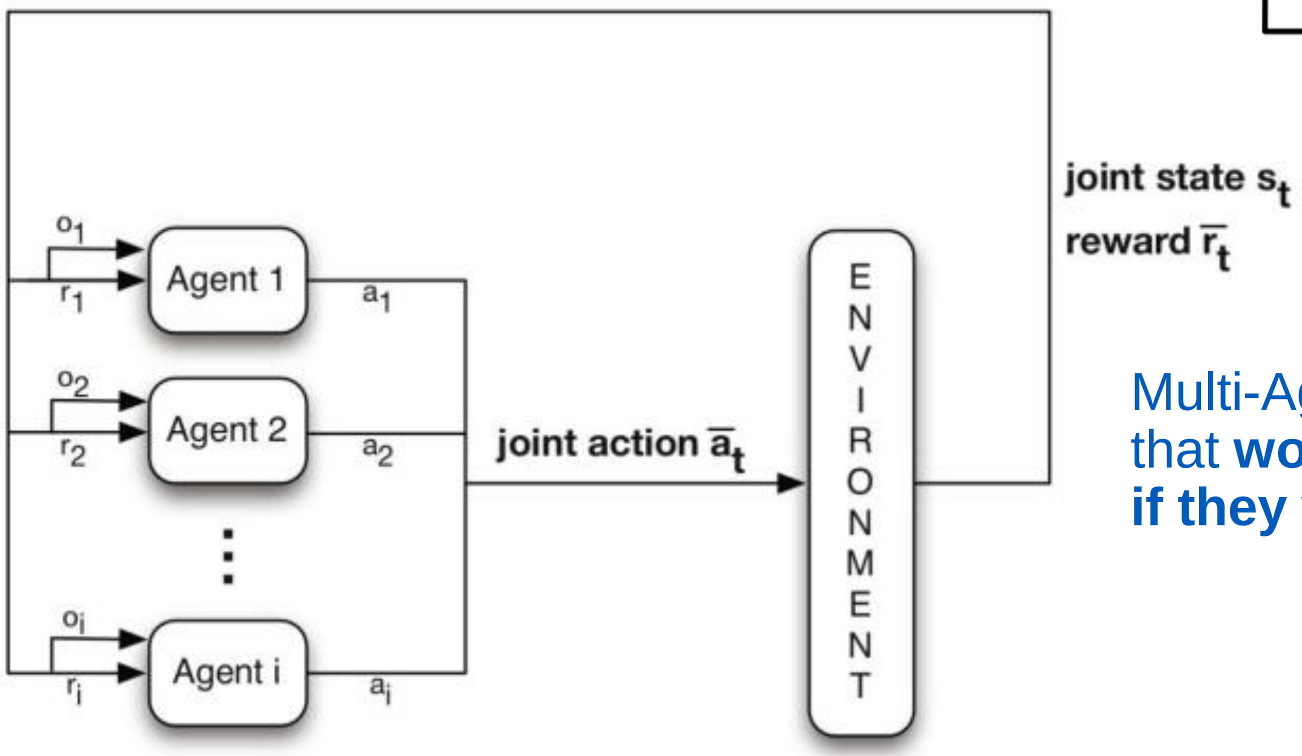
Project Description

- **Project Goal : To solve a ‘Cooperative Multi-Agent’s Task’**
- Step1 : Build a small multi-agent environment with two agents
- Step2 : Solve the environment with tabular method (Q-learning)
- Step3 : Solve the environment with using Deep RL methods (Actor-Critic)
- Step4 : Apply algorithms from Part3 to solve existing MARL problem (Pressure Plate)

BACKGROUND

Multi-Agent RL



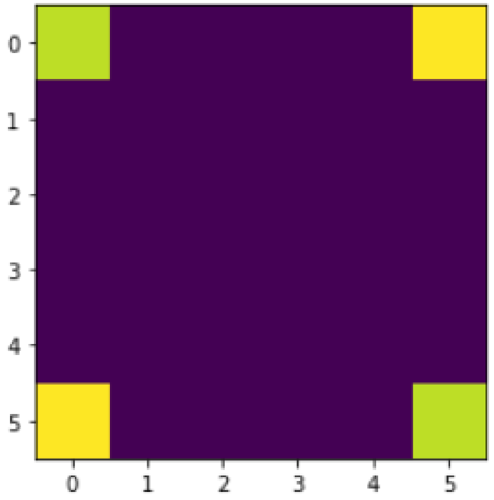


Source: Nowe, Vrancx & De Hauwere 2012

Multi-Agent RL environment solves the problem that **would not have been possible if they were a single agent**

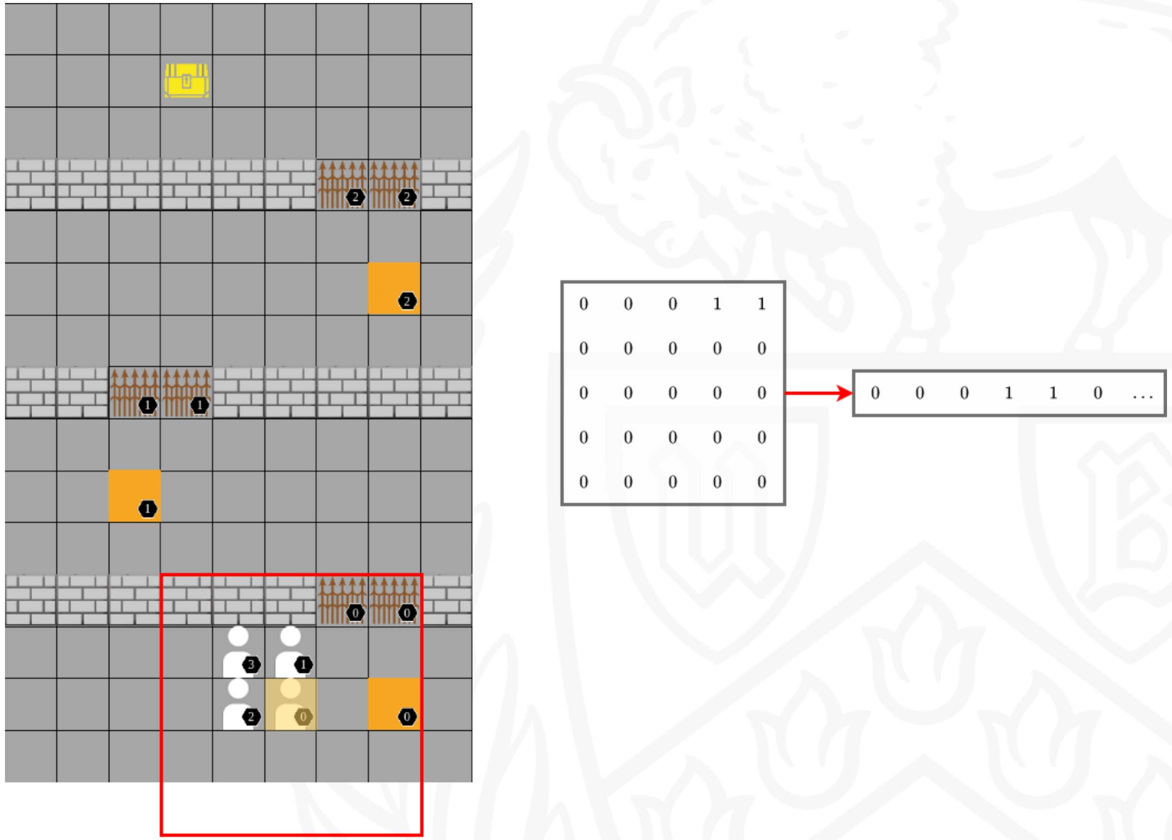
Environments

GRID WORLD



- 6x6 Grid World
- Two Agents
- Goal Position : [[0,0],[0,5]]
- State : [[5,0],[5,5]]

PRESSUREPLATE



Type	Observations	Actions	Code	Papers
Collaborative	Discrete	Discrete	Environment	/

IMPLEMENTATION

Q-Learning

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'

$$Q(S, A) \leftarrow Q(S, A) + \alpha\Big[\underbrace{\overbrace{\boxed{\underset{\text{Immediate Reward}}{R} + \gamma \max_a Q(S', a)}}^{\text{Target}} - \overbrace{\boxed{Q(S, A)}}^{\text{Prediction}}}_{\text{loss}}\Big]$$

    $S \leftarrow S'$
  until S is terminal

Epsilon-greedy:
With probability **epsilon**, choose random action **(exploration)**
With probability **1-epsilon**, choose greedy action **(exploitation)**

```
env = MAGridWorld(size=2,n_agents=6)
Q1 = np.zeros([36,5]) #obs, action
Q2 = np.zeros([36,5]) #obs, action
obs = env.reset()
epsilon=1
#repeat for each episode
for ep in range(1000):
  #observe the initial state s
  obs = env.reset()
  done = [False,False]
  epsilon = epsilon * 0.99
  #repeat for each step of episode
  for t in range(200):
    state = obs
    state1 = 6*obs[0][0] + obs[0][1]
    state2 = 6*obs[1][0] + obs[1][1]
    #select an action a from state s(e.g. epsilon-greedy) and execute it
    if np.random.random() < epsilon:
      action = [np.random.choice(5) for _ in range(2)] #exploration
    else:
      action1 = np.argmax(Q1[state1,:])
      action2 = np.argmax(Q2[state2,:])
      action = [action1, action2] #exploit
    #Receive immediate reward r, Observe the new state s'
    next_obs, reward, done, _ = env.step(action)
    #Update the table entry for Q(s,a) as follows:
    next_state1 = 6*next_obs[0][0] + next_obs[0][1]
    next_state2 = 6*next_obs[1][0] + next_obs[1][1]
    Q1[state1,action[0]] = Q1[state1,action[0]] + (reward[0]+0.9*np.max(Q1[next_state1,:])-Q1[state1,action[0]])
    Q2[state2,action[1]] = Q2[state2,action[1]] + (reward[1]+0.9*np.max(Q2[next_state2,:])-Q2[state2,action[1]])
    #state=state'
    obs = next_obs
```

Actor-Critic Method

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\boxed{\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S,\mathbf{w})}$
    $\boxed{\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla\ln\pi(A|S,\boldsymbol{\theta})}$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

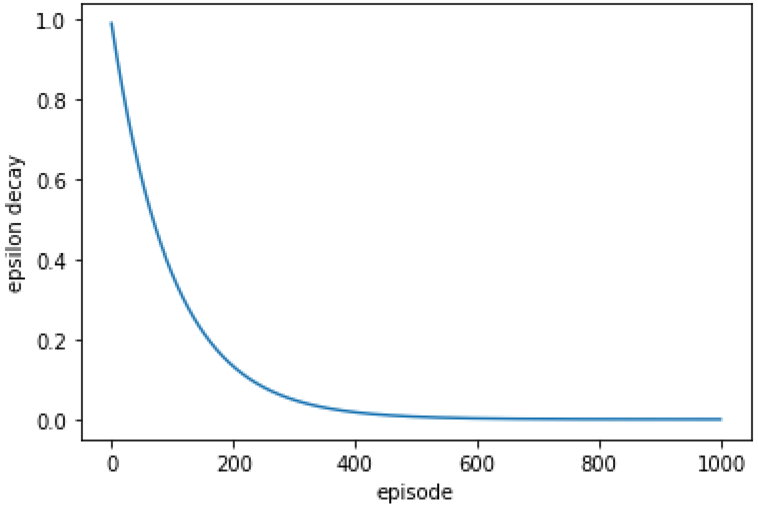
```
#Neural Network Model for Actor and Critic
class ActorCritic(nn.Module):
  def __init__(self,obs_space,action_space):
    super(ActorCritic, self).__init__()
    self.obs_space = obs_space
    self.action_space = action_space
    self.linear = nn.Linear(self.obs_space,128)
    self.actor = nn.Linear(128,self.action_space)
    self.critic = nn.Linear(128,1)

  def Actor(self, state):
    output = F.relu(self.linear(state))
    output = F.softmax(self.actor(output))
    return output

  def Critic(self, state):
    output = F.relu(self.linear(state))
    output = self.critic(output)
    return output
```

```
ac1 = ActorCritic(36, action_space)
optimizer1 = optim.Adam(ac1.parameters(),3e-4)

ac2 = ActorCritic(36, action_space)
optimizer2 = optim.Adam(ac2.parameters(),3e-4)

for ep in range(1000): #for each episode

  #Initialize S
  obs = env.reset()

  loss1 = 0
  loss2 = 0
  cumulative_rewards = 0
  reward1 = 0
  reward2 = 0

  for t in range(100): #for each timesteps
    state = obs
    state1 = torch.zeros([1,36])
    integer_1 = 6*state[0][0] + state[0][1]
    state1[0,integer_1] = 1

    integer_2 = 6*state[1][0] + state[1][1]
    state2 = torch.zeros([1,36])
    state2[0,integer_2] = 1

    #Take action A, observe S', R
    probs1 = ac1.Actor(state1)
    m1 = Categorical(probs1)
    action1 = m1.sample()

    probs2 = ac2.Actor(state2)
    m2 = Categorical(probs2)
    action2 = m2.sample()
    next_obs, reward, done, _ = env.step([action1,action2])


    #advantage function = TD error
    #a <- R + gamma * v(S',w) - v(S,w)
    next_state = next_obs

    integer_1 = 6*next_state[0][0] + next_state[0][1]
    next_state1 = torch.zeros([1,36])
    next_state1[0,integer_1] = 1
    value1 = ac1.Critic(state1)
    next_value1 = ac1.Critic(next_state1)

    integer_2 = 6*next_state[1][0] + next_state[1][1]
    next_state2 = torch.zeros([1,36])
    next_state2[0,integer_2] = 1
    value2 = ac2.Critic(state2)
    next_value2 = ac2.Critic(next_state2)

    target1 = torch.tensor(reward[0]) + 0.9 * next_value1
    target2 = torch.tensor(reward[1]) + 0.9 * next_value2
    if done[0] == True :
      target1 = torch.tensor(reward[0])
    if done[1] == True:
      target2 = torch.tensor(reward[1])

    #update critic by minimizing loss
    lossC1 = F.smooth_l1_loss(target1, value1)
    lossC2 = F.smooth_l1_loss(target2, value2)

    #update actor by minimizing loss
    lossA1 = -m1.log_prob(action1) * (target1 - value1)
    loss1 += lossC1 + lossA1.sum()
    lossA2 = -m2.log_prob(action2) * (target2 - value2)
    loss2 += lossC2 + lossA2.sum()

    #S<-S'
    obs = next_obs

    #break
    if done[0] == True and done[1] == True:
      print(state)
      break

  #backpropagation
  optimizer1.zero_grad()
  optimizer2.zero_grad()
  loss1.backward()
  loss2.backward()
  optimizer1.step()
  optimizer2.step()

env.close()
```

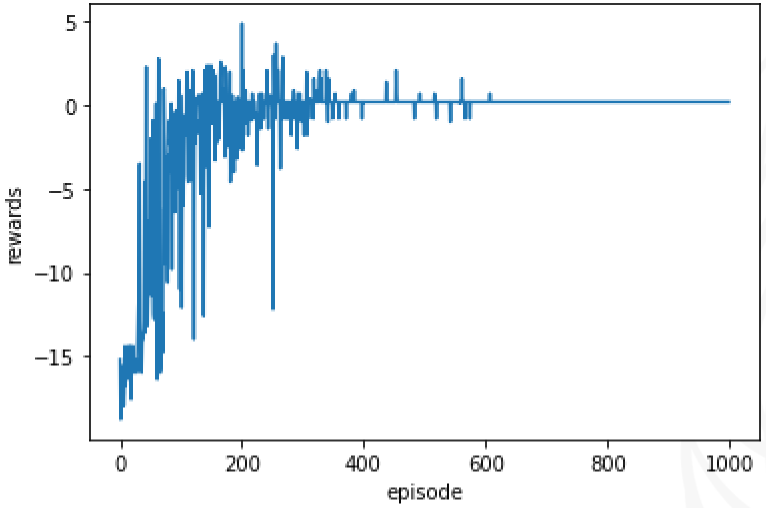
Actor: Updates policy parameters θ, in direction suggested by critic (acts)
Critic: Updates action-value function parameters w (action is good or bad)

RESULTS

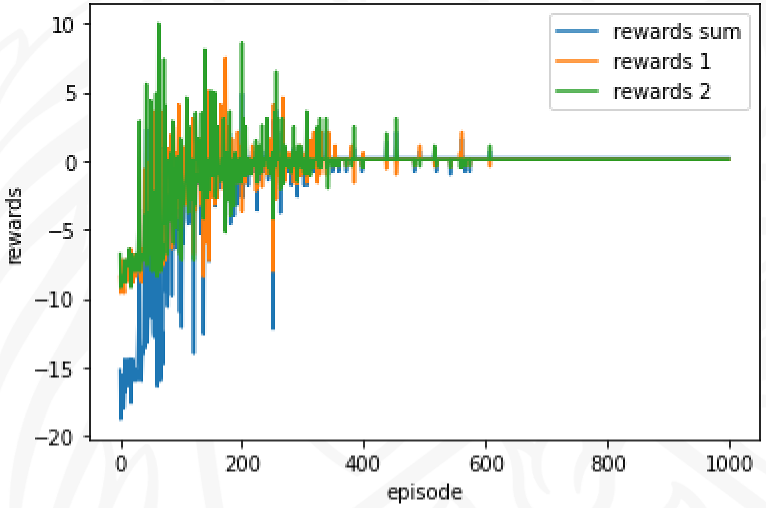
Step2 : Solve the environment with tabular method (Q-learning)



Epsilon decay per episode

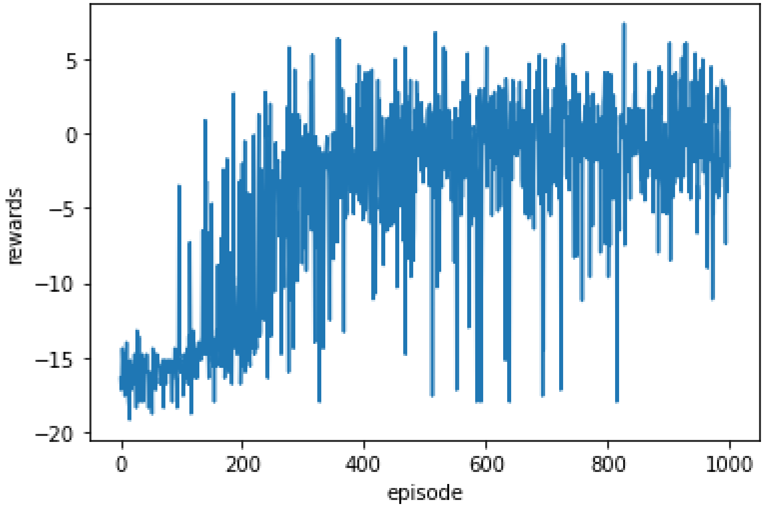


Rewards per episode

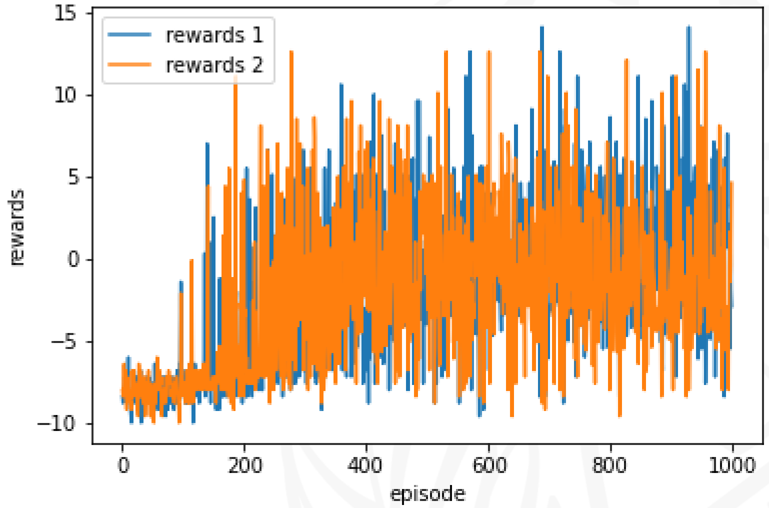


Compare rewards per agent

Step3 : Solve the environment with using Deep RL methods (Actor-Critic)



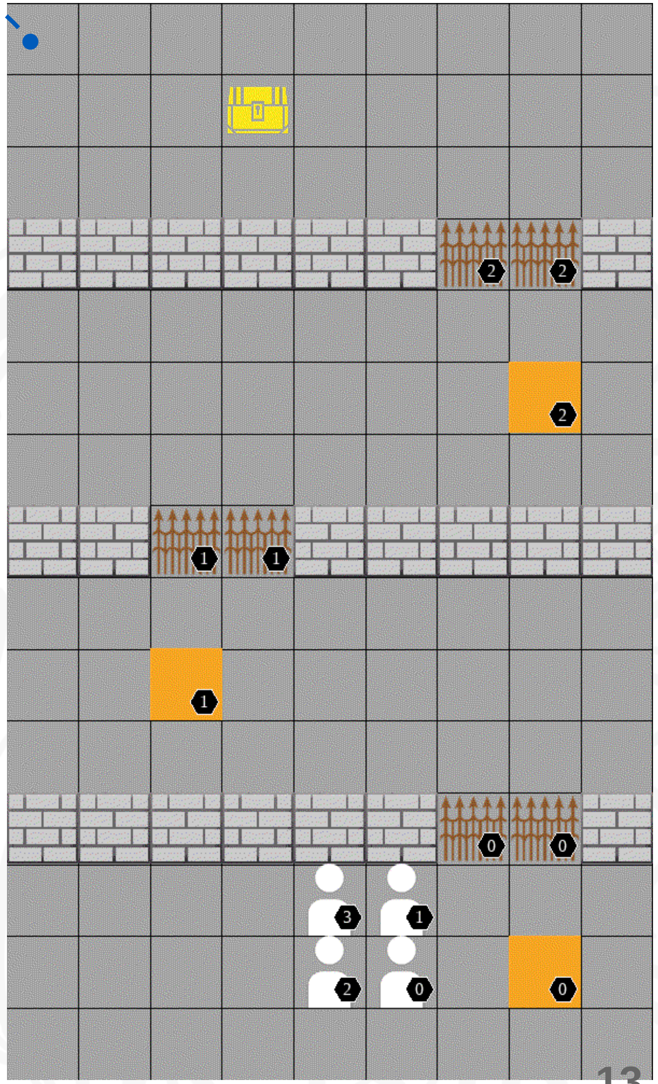
Rewards per episode

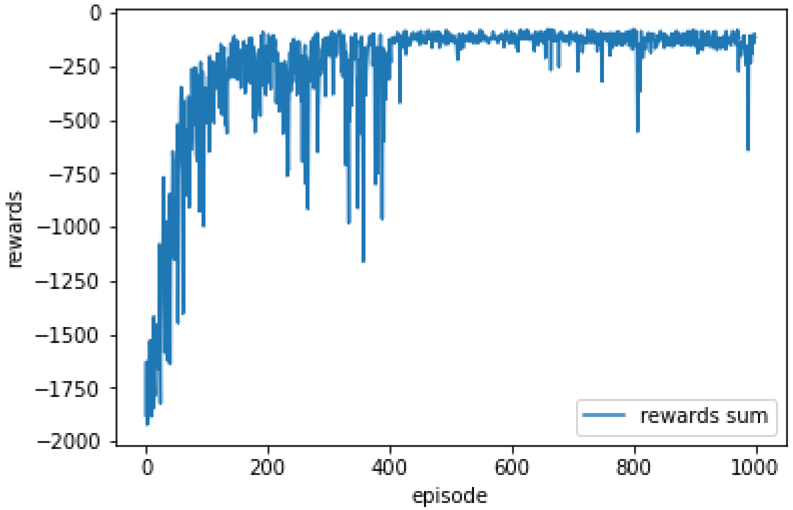


Compare rewards per agent

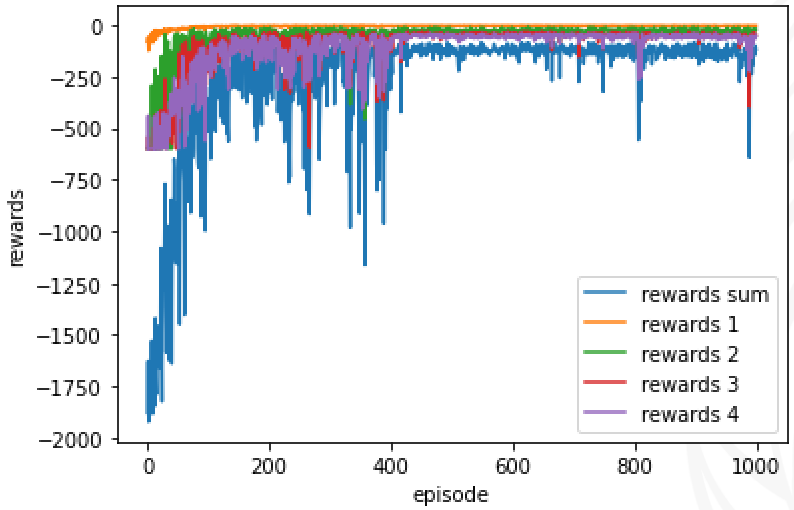
Step4: Solve PRESSUREPLATE with Q-learning

PRESSUREPLATE Visualization



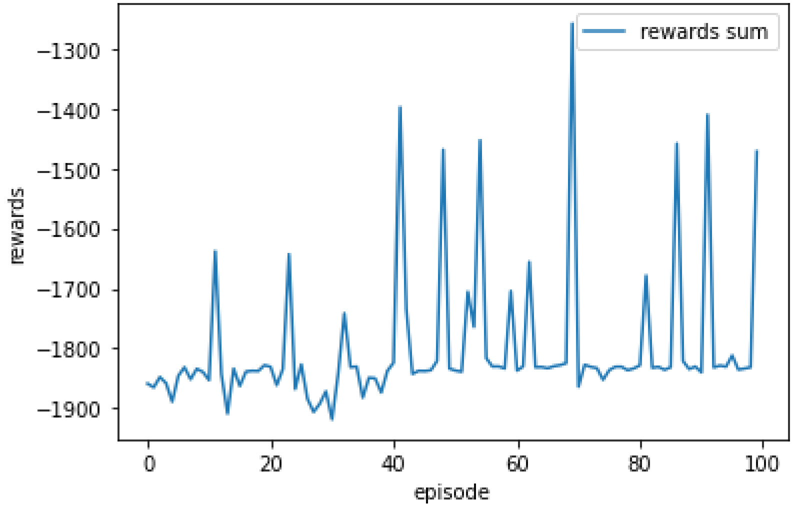


Rewards per episode (Q-learning)

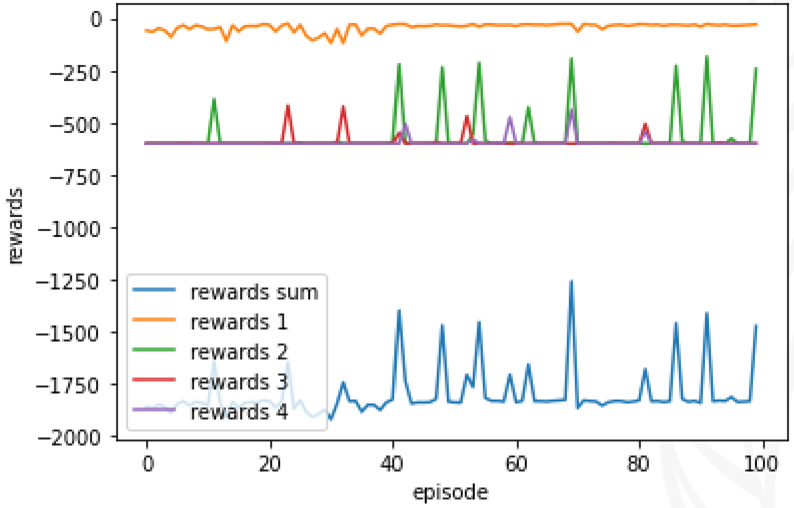


Compare rewards per agent(Q-learning)

Step4: Solve PRESSUREPLATE with Actor-Critic

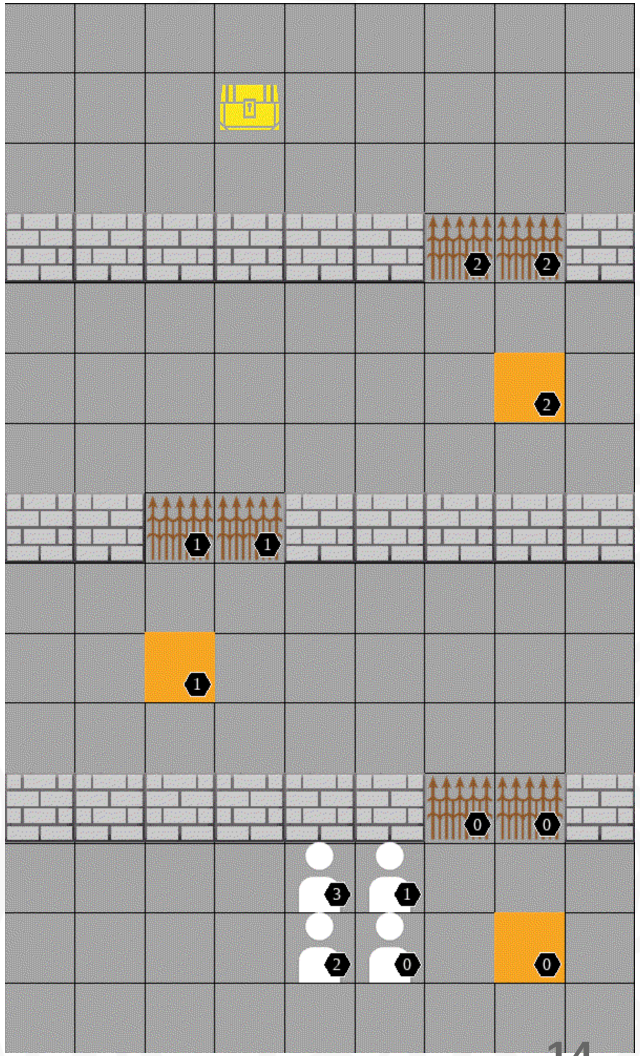


Rewards per episode (Actor-Critic)



Compare rewards per agent(Actor-Critic)

PRESSUREPLATE Visualization



KEY OBSERVATIONS / SUMMARY

Key Observations & Summary

Multi Agent Environment

- In MARL, each agent has its own observation, reward, and action, but when we implement these into the environment, we should join the actions, and then we could get joint state and rewards.
- Multi-Agent RL environment solves the problem that would not have been possible if they were a single agent

Q-learning vs Actor-Critic

- Actor-Critic trained well in simple multi agent grid world than Q-learning
- However, in pressureplate where the number of agents is four, rewards per episode in Q-learning exceeds the rewards in Actor-Critic
- The reason of this is the environment is too complicate to be solved by actor-critic
- We need modification in code to improve performance of Actor-Critic

Reference:

https://agents.inf.ed.ac.uk/blog/multiagent-learning-environments/#pressureplate
https://github.com/uoe-agents/pressureplate#customizing-scenarios
https://piazza.com/class_profile/get_resource/kyxmabvv7jc3q4/l20vt1vz4m17jy
https://piazza.com/class_profile/get_resource/kyxmabvv7jc3q4/l02l9hc4wq268
https://piazza.com/class_profile/get_resource/kyxmabvv7jc3q4/l1e1arkrv51y9

THANK YOU