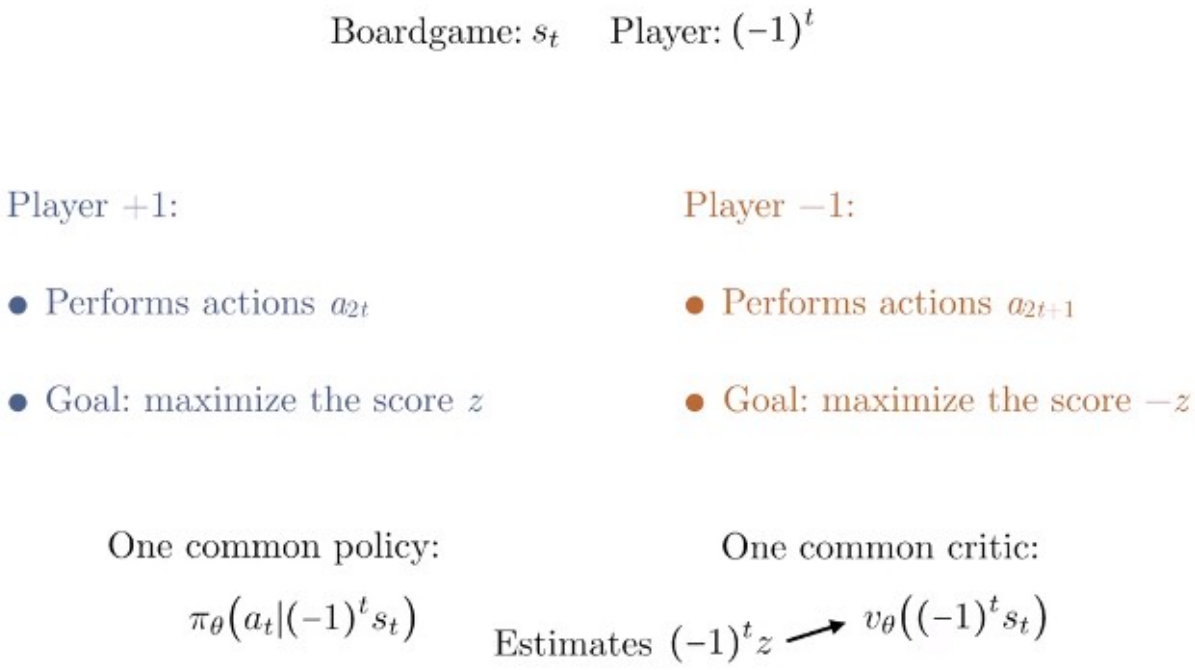
Basic concept

1)Multi-agent system
Multi-agent can share experiences and is robust

2)Multi-agent reinforce learning(MARL)
it is assumed that the environment is stationary(single-agent).
The matter agent approach : each agent have different observation of the environment state.
Mix cooperative and competitive behavior
maximize reward

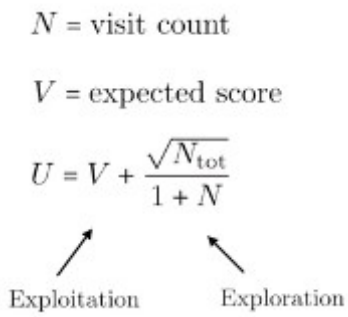
3)multi-agent DDPG
the normal agents are rewarded based on the least distance of any of the agents to the landmark.
Penalized based on the distance between adversary and the target landmark.

4)monte carlo tree search



(from udacity)

random sampling



(from udacity)

choose highest N, U for select branch
expansion ,back-propagation: update the statistic on the previous node,N→ V→ U
choose action by max U

MCTS Summary

Initialize top-node for current state, loop over actions for some N_{tot}:

1. Start from the top-node, repeatedly pick the child-node with the largest U
2. If $N = 0$ for the node, play a random game.
   Else, expand node, play a random game from a randomly selected child
3. Update statistics, back-propagate and update N and U as needed

Select move with highest visit counts

(from udacity)

5)guide tree search

Exploration guided by the Policy $\pi_\theta\left(a_t|(-1)^t s_t\right)$ = Expert Policy

Simulation done by the Critic $v_\theta\left((-1)s_t\right)^t$ = Expert Critic

$\pi_\theta\left(a_t|(-1)^t s_t\right)$ = Expert Policy

$v_\theta\left((-1)s_t\right)^t$ = Expert Critic

N = visit count

V = expected score

$$U = V + c\,\pi_\theta\left(a_t|(-1)^t s_t\right)\frac{\sqrt{N_{\text{tot}}}}{1+N}$$

Critic controlled exploitation

Policy guided exploration

Hyperparameter

(from udacity)

policy focus on exploring moves that an expert is likely to play.

Critic estimate the outcome of a game without running a simulation

4)self-play training

Action probability:

$$p_a^{(t)} = \frac{N_a^{(t)}}{\sum_a N_a^{(t)}}$$

(from udacity)

loss function to closer result of monte carlo tree search

Improves critic

$$L(\theta) = \sum_t \left\{ \left[v_\theta\left((-1)^t s_t\right) - (-1)^t z \right]^2 - \sum_a p_a^{(t)} \log \pi_\theta\left((-1)^t s_t\right) \right\}$$

Improves Policy

(from udacity)

5)alphazero

AlphaZero Algorithm

1. Initialize network for critic and policy (v_θ, π_θ)
2. Play a game using MCTS
3. Compute $L(\theta) = \sum_t \left\{ \left[v_\theta\left((-1)^t s_t\right) - (-1)^t z \right]^2 - \sum_a p_a^{(t)} \log \pi_\theta\left((-1)^t s_t\right) \right\}$, perform gradient descent
4. Repeat Step 2-3

(from udacity)

algorithm

1.Why you choose the MADDPG/DDPG algorithm?
1)DDPG average many trajectories to make model gradient noise reduction and dont care about each agent interaction.

2)MADDP is same with most parts of DDPG like gradient strategies , but may be different in agent reward and penalty define like some case use distance to define reward

3)in tennis case, because of tennis agent only care get ball by properly action, we select normal DDPG to learning the state and action and dont spend much time to change the each agent reward define . Thus we only utilize the experience to get information of two agent interaction .

2.Did you used replay buffer, epsilon-greedy etc approaches? (Also are both agents sharing same experience buffer or different)

1) need use replay buffer to get N-step states and actions for continuous control.

2)epsilon-greedy method can help model find convergence weight.

3)Soft update model parameters to help model have part properties of previous network

4)in tennis game ,multi-agent strategies is that save two agents experience ,then use the result to learn policy. And every iteration will add different noise to action and state for each agent

3.Is the noise added to agents?

Need add noise to agents for model robust by sampling each agent's action

4.Why you chose the particular model architecture for Actor/Critic?

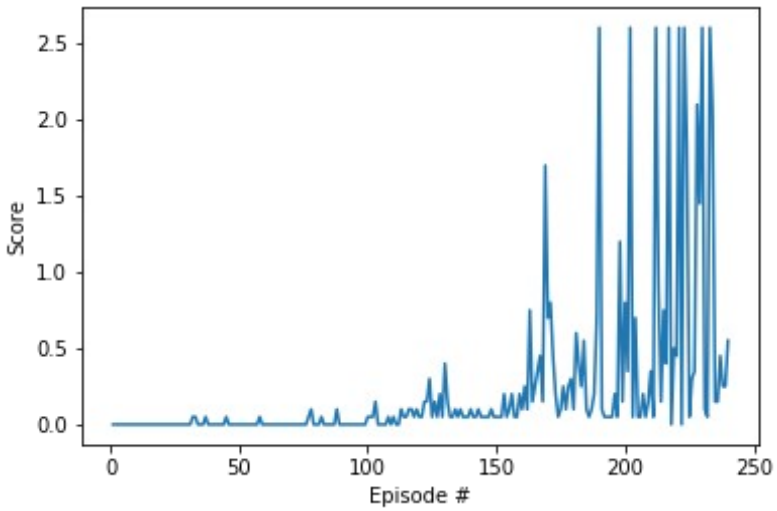
If Layer number is large , the model can get more feature of state and action. But it is not absolutely. For small training data ,the larger layer is bad for model convergence. So I choose 2 layer for actor and critic. And the reason of layer size is same as layer number.

code(reference to Tomas0413/Collaboration-and-Competition)

```
def step(self, state, action, reward, next_state, done, current_time_step):
        """Save experience in replay memory, and use random sample from buffer to learn."""
        # Save experience / reward
        self.memory.add(state, action, reward, next_state, done)

        # Learn, if enough samples are available in memory
        if len(self.memory) > BATCH_SIZE and current_time_step %
LEARN_EVERY_X_TIMESTEPS ==0:
            for update_step_num in range(UPDATES_PER_LEARN_STEP):
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)

def act(self, state, add_noise=True):
        """Returns actions for given state as per current policy."""
        state = torch.from_numpy(state).float().to(device)
        self.actor_local.eval()
        with torch.no_grad():
            action = self.actor_local(state).cpu().data.numpy()
            #print(action)
        self.actor_local.train()
        if add_noise:
            for i  in range(LEARN_EVERY_X_TIMESTEPS):
                action[i] += self.epsilon * self.noise.sample()
        return np.clip(action, -1, 1)
```



Parameters

LEARN_EVERY_X_TIMESTEPS = 2 # agent number
BUFFER_SIZE = int(1e5) # replay buffer size: more buffer can get more trajectories for reducing noise effect. But too large one may leak the store.
BATCH_SIZE = 128 # minibatch size:it is about sampling size . Size is depend on agent number and store size, just like BUFFER_SIZE. Besides, it is influence in convergence of training.
GAMMA = 0.99 # discount factor
TAU = 1e-3 # for soft update of target parameters
LR_ACTOR = 1e-4 # learning rate of the actor .in some extent , the small one will let actor learning slower than large one
LR_CRITIC = 1e-4 # learning rate of the critic
WEIGHT_DECAY = 0 # L2 weight decay

UPDATES_PER_LEARN_STEP = 10# the more sampling and learning, the better to improve robust. But if step too large , it will slow down the learning speed
EPSILON = 1.0 # about initialized adding noise percent of action
EPSILON_DECAY = 1e-6 # about decayed number for each adding noise percent of action. It helps reduce noise effect in each iteration. If value is 1e-3, model cannot convergence. if value is 1e-8, it is too small to effect in the noise. Thus, the variety of sample is decreased and the model may becomes divergence in the same training times like 400.

DDPG layer size
in tennis game , action is similar with critic. These are simple and dont exist large different in each agent. So we only set two layer number and size is 256 to 128 for fully capture feature. when first lay set 128 , convergence speed will slow down than 256. if first lay value is 400 , model r=will divergence.
actor : 256→ 128
critic : 256→128

Future

1) add optical flow to extract action feature for actor training in dynamic objects (reference : Motion Perception in Reinforcement Learning with Dynamic Objects)

2) use many critic penalties to make policies gentle(reference : Learning Gentle Object Manipulation with Curiosity-Driven Deep Reinforcement Learning)