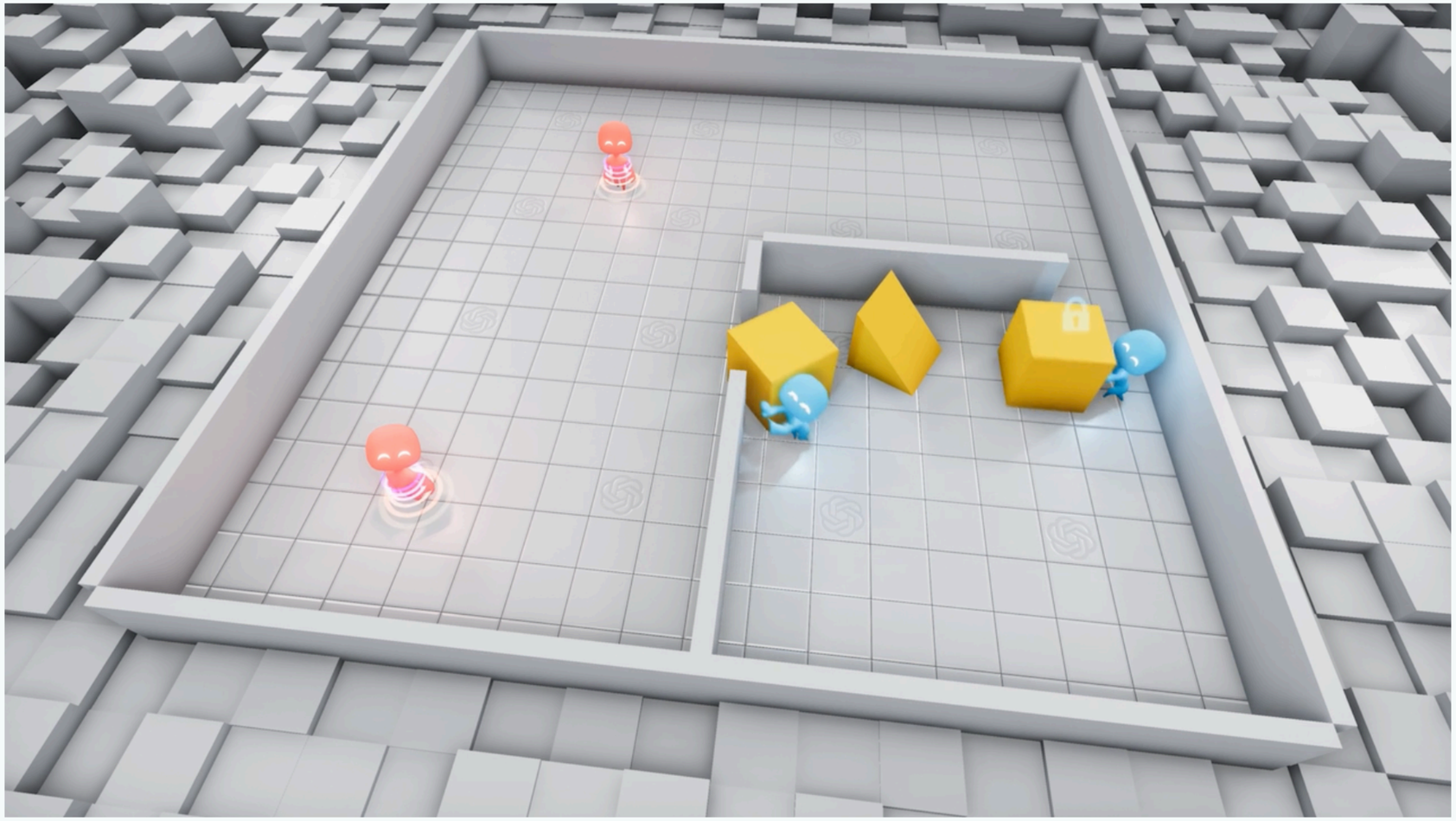
Multi-Agent Deep Deterministic Policy Gradient (MADDPG)

Marco Celoria

Trieste - July, 8th 2022

Emergent Tool Use From Multi-Agent Autocurricula (2019)

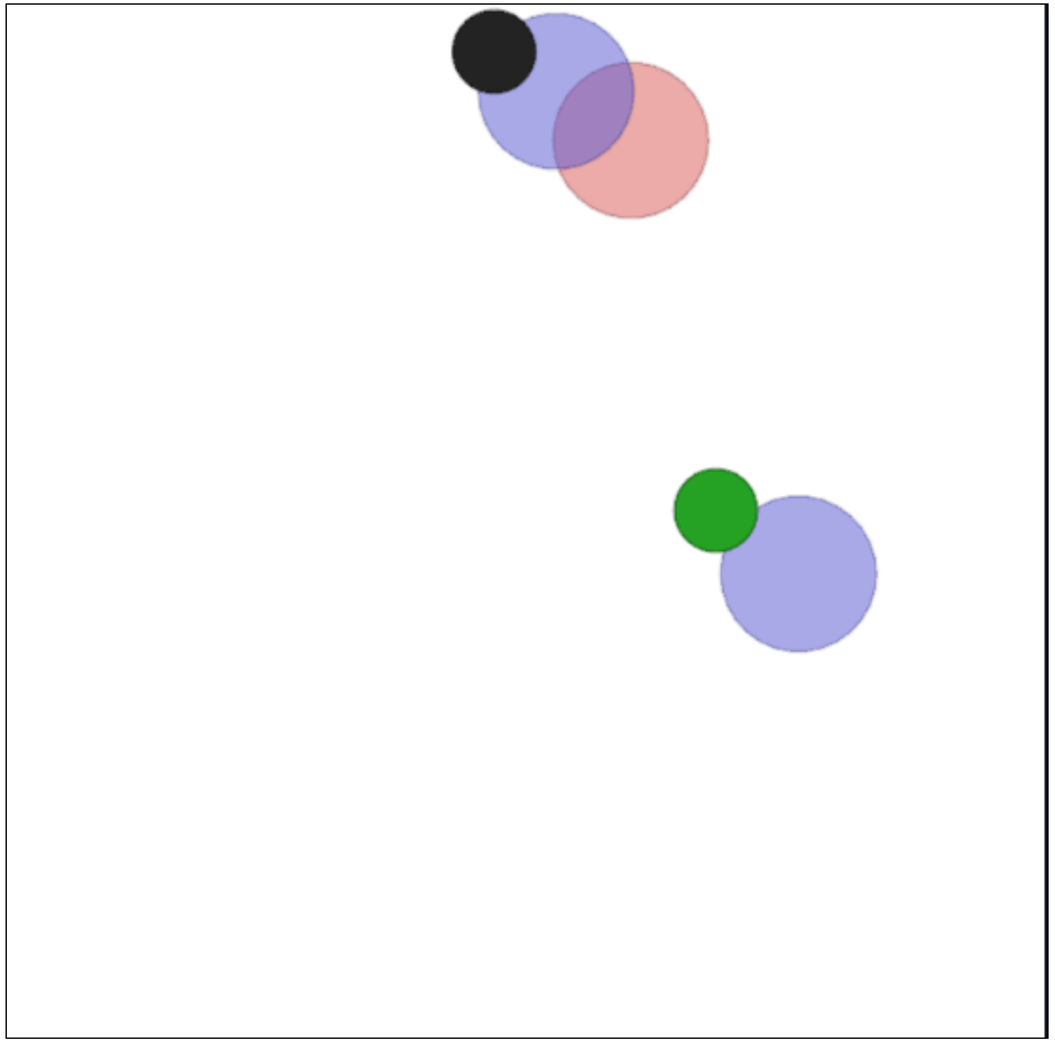
Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, Igor Mordatch



Additionally, hiders learn to **coordinate** who will block which door and who will go grab the ramp. In cases where the boxes are far from the doors, hiders **pass boxes to each other** in order to block the doors in time.

Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments (2017)

Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, Igor Mordatch



References:

- Deep RL Bootcamp [https://sites.google.com/view/deep-rl-bootcamp]
- OpenAI: Spinning Up [https://spinningup.openai.com/en/latest/]
- R. Lowe et al. “Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments”
- Lillicrap et al. “Continuous Control With Deep Reinforcement Learning”
- Silver et al. “Deterministic Policy Gradient Algorithms”
- Mnih et al. “Playing Atari with Deep Reinforcement Learning”
- R. Sutton & A. Barto Reinforcement Learning: An Introduction Second Edition

References:

- Phil Tabor: Multi-Agent-Deep-Deterministic-Policy-Gradients

  [https://github.com/philtabor/Multi-Agent-Deep-Deterministic-Policy-Gradients]

- MrSyee (Kyunghwan Kim): pg-is-all-you-need

  [https://github.com/MrSyee/pg-is-all-you-need]

- Andrew Gordienko: Reinforcement Learning: DQN w Pytorch

  [https://andrew-gordienko.medium.com/reinforcement-learning-dqn-w-pytorch-7c6faad3d1e]

- Eugenia Anello: Deep Q-network with Pytorch and Gym to solve the Acrobot game

  [https://towardsdatascience.com/deep-q-network-with-pytorch-and-gym-to-solve-acrobot-game-d677836bda9b]

Background: Markov Games

- A Markov game for N agents is defined by:

1. a set of states $\mathcal{S}$ describing the possible configurations of all agents
2. a set of actions $\mathcal{A}_1 \ldots \mathcal{A}_N$
3. a set of observations $\mathcal{O}_1 \ldots \mathcal{O}_N$ for each agent

Background: Markov Games

- To choose actions, each agent i uses a stochastic policy $\pi_{\theta_i} : \mathcal{O}_i \times \mathcal{A}_i \rightarrow [0,1]$
- The next state is produced according to the state transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A}_1 \times \ldots \times \mathcal{A}_N \rightarrow \mathcal{S}$
- Each agent i obtains rewards as a function of the state and agent's action $r_i : \mathcal{S} \times \mathcal{A}_i \rightarrow \mathbb{R}$
- Each agent i receives a private observation correlated with the state $\mathbf{o}_i : \mathcal{S} \rightarrow \mathcal{O}_i$
- The initial states are determined by a distribution $\rho : \mathcal{S} \rightarrow [0,1]$
- Each agent i aims to maximize its own total expected return

$$R_i = \sum_{t=0}^{T} \gamma^t r_i^t$$ where γ is a discount factor and T is the time horizon

State-action value function (Q function)

- For every $s \in \mathcal{S}, a \in \mathcal{A}$ and policy π define the action-value function

$$Q^{\pi}(s,a) = \mathbb{E}[R \,|\, s^t = s, a^t = a]$$

- This Q function can be recursively rewritten as

$$Q^{\pi}(s,a) = \mathbb{E}_{s'}\left[r(s,a) + \gamma\, \mathbb{E}_{a' \sim \pi}\left[Q^{\pi}(s',a')\right]\right]$$

- $Q^*(s,a)$ is the expected utility starting in s, taking action a, and acting optimally
- The Bellman equation is therefore

$$Q^*(s,a) = \sum_{s'} P(s' \,|\, s,a)\left(R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right)$$

Q-value Iteration: Algorithm

Algorithm 2: Value Iteration for Q-values

Result: Find $Q^{\pi^*}(s,a)$ and $\pi^*(s),\ \forall s$

$Q(s,a) \leftarrow 0,\ \forall s,a \in S \times A$;
$\Delta \leftarrow \infty$;
while $\Delta \geq \Delta_\circ$ **do**
    $\Delta \leftarrow 0$;
    for *each* $s \in S$ **do**
        for *each* $a \in A$ **do**
            $temp \leftarrow Q(s,a)$;
            $Q(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q(s',a')]$;
            $\Delta \leftarrow \max_a(\Delta, |temp - Q(s,a)|)$;
        end
    end
end
$\pi^*(s) \leftarrow argmax_a Q^*(s,a), \forall s \in S$;
return $\pi^*(s), \forall s \in S$

Model-free Q-learning

- What if we don't know the model $P(s' \mid s, a)$ and $R(s, a, s')$?

1. Model-based approaches: [We will not focus on these approaches]

2. Model-free approach:

- Don't learn a model, but rather learn value function (Q value) or policy directly

- Idea: Rewrite the update as expectation and replace expectation by samples

$$Q_{k+1}(s,a) \leftarrow \mathbb{E}_{s' \sim P(s'|s,a)} \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right] = \frac{1}{|\mathscr{D}|} \sum_{\mathscr{D} \text{ samples}} \left[R(s,a,s') + \gamma \max_{a'} Q(s',a') \right]$$

- Then, consider running average of $k+1$ samples of a quantity x:

$$\bar{x}_{k+1} = \frac{x_1 + x_2 + \ldots + x_k + x_{k+1}}{k+1} = \frac{x_1 + x_2 + \ldots + x_k}{k+1} \cdot \frac{k}{k} + \frac{x_{k+1}}{k+1} = \frac{k}{k+1} \cdot \frac{x_1 + x_2 + \ldots + x_k}{k} + \frac{x_{k+1}}{k+1} = \frac{k}{k+1} \cdot \bar{x}_k + \frac{x_{k+1}}{k+1}$$

$$\bar{x}_{k+1} = (1-\alpha)\ \bar{x}_k + \alpha\ x_{k+1} \qquad \text{where} \qquad \alpha = \frac{1}{k+1}$$

- and so the running average of $Q(s,a)$ is

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)\ Q_k(s,a) + \alpha\ \left[R(s,a,s') + \gamma\ \max_{a'}\ Q_k(s',a') \right]$$

(Tabular) 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 $Q(\mathit{terminal}, \cdot) = 0$

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'
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until S is terminal

Deep Q-Learning (also Deep Q-Network - DQN)

- High-level idea - make Q-learning look like supervised learning.
- Two main ideas for stabilizing Q-learning.

1. Apply Q-updates on batches of past experience instead of online
   - Experience replay makes the data distribution more stationary
2. Use an older set of weights to compute the targets (target network):
   - Keeps the target function from changing too quickly.

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim\mathcal{D}}\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2$$

- The network can end up chasing its own tail because of bootstrapping.

Deep Q-Learning: Algorithm

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Deep Q-Learning for Multi-Agent settings

- Q-Learning can be directly applied to multi-agent settings by having each agent i learn an independently optimal function Q_i
- However, because agents are independently updating their policies as learning progresses, the environment appears non-stationary from the view of any one agent, violating Markov assumptions required for convergence of Q-learning.
- Another difficulty is that the experience replay buffer cannot be used in such a setting since

$$P(s' \mid s, a, \pi_1, \ldots, \pi_N) \neq P(s' \mid s, a, \pi'_1, \ldots, \pi'_N)$$

when any $\pi_i \neq \pi'_i$

Policy Gradient

- Consider a stochastic, parameterized policy, π_θ. We aim to maximize the expected return

$$J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$$

- We would like to optimize the policy by gradient ascent $\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)\Big|_{\theta_k}$

- We get the following expectation (which means that we can estimate it with a sample mean)

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \,|\, s_t) R(\tau)\right]$$

- If we collect a set of trajectories $\mathcal{D} = \{\tau_i\}_{i=1,\ldots,N}$ where each trajectory is obtained by letting the agent act in the environment using the policy π_θ, the policy gradient can be estimated with

$$\hat{g} = \frac{1}{|\,\mathcal{D}\,|} \sum_{\tau \in \mathcal{S}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \,|\, s_t) R(\tau)$$

Policy Gradient: Reducing variance

- Causality: policy at time t' cannot affect reward at time t when $t < t'$. Consider

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \,|\, s_t) R(\tau) \right]$$

- This includes terms proportional to past rewards, all of which had zero mean, but nonzero variance. As a result, they would just add noise to sample estimates of the policy gradient.

- By removing them, we reduce the number of sample trajectories needed.

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \,|\, s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) \right]$$

Reward-to-go policy gradient: $\hat{R}_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})$

- Proof (https://spinningup.openai.com/en/latest/spinningup/extra_pg_proof1.html)

Policy Gradients: Baselines

- From EGLP lemma, it follows that for any function b which only depends on state,

$$\mathbb{E}_{a_t \sim \pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t \,|\, s_t)\right] = 0 \qquad \Longrightarrow \qquad \mathbb{E}_{a_t \sim \pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t \,|\, s_t) b(s_t)\right] = 0$$

- This allows us to add or subtract any number of terms like this from our expression for the policy gradient, without changing it in expectation:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \,|\, s_t)\left(\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t)\right)\right]$$

- Note: the following choice is also valid for the finite-horizon undiscounted return setting

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \left(\nabla_\theta \log \pi_\theta(a_t \,|\, s_t)\right) Q^{\pi_\theta}(s_t, a_t)\right]$$

- Proof (https://spinningup.openai.com/en/latest/spinningup/extra_pg_proof2.html)

Policy Gradient Algorithms: Summary

- The idea is to directly adjust the parameters θ of the policy in order to maximize the objective $J(\theta) = \mathbb{E}_{s\sim p^\pi, a\sim\pi_\theta}[R]$ by taking steps in the direction of $\nabla_\theta J(\theta)$

- Using the Q function defined previously, the gradient of the policy can be written

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim p^\pi, a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) Q^\pi(s,a)\right]$$

where p^π is the state distribution

- This has given rise to several algorithms, which differ in how they estimate Q

- One could learn an approximation of the true function $Q^\pi(s,a)$ by TD learning

- This $Q^\pi(s,a)$ is called the *critic* and leads to a variety of *actor-critic* algorithms

Policy Gradient for Multi-Agent setting

- Policy gradient methods are known to exhibit high variance gradient estimates
- This is exacerbated in multi-agent settings; since an agent's reward usually depends on the actions of many agents, the reward conditioned only on the agent's own actions (when the actions of other agents are not considered in the agent's optimization process) exhibits much more variability, thereby increasing the variance of its gradients

Deep Deterministic Policy Gradient (DDPG)

- It is possible to extend the policy gradient framework to deterministic policies $\mu_\theta : \mathcal{S} \rightarrow \mathcal{A}$
- Under certain conditions we can write the gradient $J(\theta) = \mathbb{E}_{s\sim p^\mu}[R(s,a)]$ as

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\mathcal{D}}\left[\nabla_\theta \mu_\theta(s)\, \nabla_a Q^\mu(s,a)\Big|_{a=\mu_\theta(s)}\right]$$

- Since this theorem relies on $\nabla_a Q^\mu(s,a)$ it requires that the action space $\mathcal{A}$ (and policy μ) be continuous
- In DDPG the policy μ and critic Q^μ are approximated with deep neural networks
- DDPG is an off-policy algorithm, and samples trajectories from a replay buffer of experiences that are stored throughout training
- DDPG also makes use of a target network, as in DQN

The Q-Learning Side of DDPG

- Let's recap the Bellman equation describing the optimal action-value function

$$Q^*(s,a) = \mathbb{E}_{s' \sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a') \right]$$

- This Bellman equation is the starting point for learning an approximator to $Q^*(s,a)$.

- Suppose the approximator is a neural network $Q_\phi(s,a)$, with parameters ϕ, and that we have collected a set $\mathcal{D}$ of transitions (s,a,r,s',d)

- We can set up a mean-squared Bellman error (MSBE) function, which tells us roughly how closely $Q_\phi(s,a)$ comes to satisfying the Bellman equation:

$$L(\phi, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}}\left[\left(Q_\phi(s,a) - (r + \gamma\ (1-d)\ \max_{a'} Q_\phi(s',a') \right)^2 \right]$$

- where $d == 1 ==$ True when s' is a terminal state

Trick One: Replay Buffers

- All standard algorithms for training a deep neural network to approximate $Q^*(s, a)$ make use of an experience replay buffer
- This is the set $\mathscr{D}$ of previous experiences
- In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep everything
- If you only use the very-most recent data, you overfit to that and things will break
- If you use too much experience, you may slow down your learning
- This may take some tuning to get right

Trick Two: Target Networks

- Q-learning algorithms make use of target networks. The term $r + \gamma\ (1 - d)\ \max_{a'} Q_{\phi}(s', a')$ is the target, because when we minimize the MSBE loss, we are trying to make the Q-function be more like this target
- The target depends on the same parameters we are trying to train. This makes MSBE minimization unstable!
- The solution is to use a set of parameters which comes close to ϕ, but with a time delay—i.e. a second network, called the target network ϕ_{targ}
- The target network is updated once per main network update by polyak averaging

$$\phi_{targ} \leftarrow \rho\ \phi_{targ} + (1 - \rho)\ \phi \qquad \rho \in (0,1) \qquad \rho \simeq 1$$

The Policy Learning Side of DDPG

- Policy learning in DDPG is fairly simple
- We want to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_\phi(s, a)$
- Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve

$$\max_\theta \mathbb{E}_{s \sim \mathcal{D}} \left[Q_\phi \left(s, \mu_\theta(s) \right) \right]$$

Note that the Q-function parameters are treated as constants here

DDPG: Exploration vs. Exploitation

- Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals

- To make DDPG policies explore better, we add noise to actions at training time.

- The authors of the original DDPG paper used time-correlated OU noise, but more recent results suggest that uncorrelated, mean-zero Gaussian noise works well

- To facilitate getting higher-quality training data, you may reduce the scale of the noise over the course of training

- At test time, to see how well the policy exploits what it has learned, we do not add noise to the actions

DDPG: Algorithm

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Multi-Agent Actor Critic: Assumptions

- Goal is to derive an algorithm able to operate under the following constraints:

1. The learned policies can only use local information (i.e. their own observations) at execution time

2. We do not assume a differentiable model of the environment dynamics

3. We do not assume any particular structure on the communication method between agents

- That is, we don’t assume a differentiable communication channel

Multi-Agent Actor Critic

- We adopt the framework of centralized training with decentralized execution
- We allow the policies to use extra information to ease training, so long as it is not used at test time

☐ In Q-learning, the Q function generally cannot contain different information at training and test time

☑ Extension of actor-critic policy gradient methods where the critic is augmented with extra information about the policies of other agents

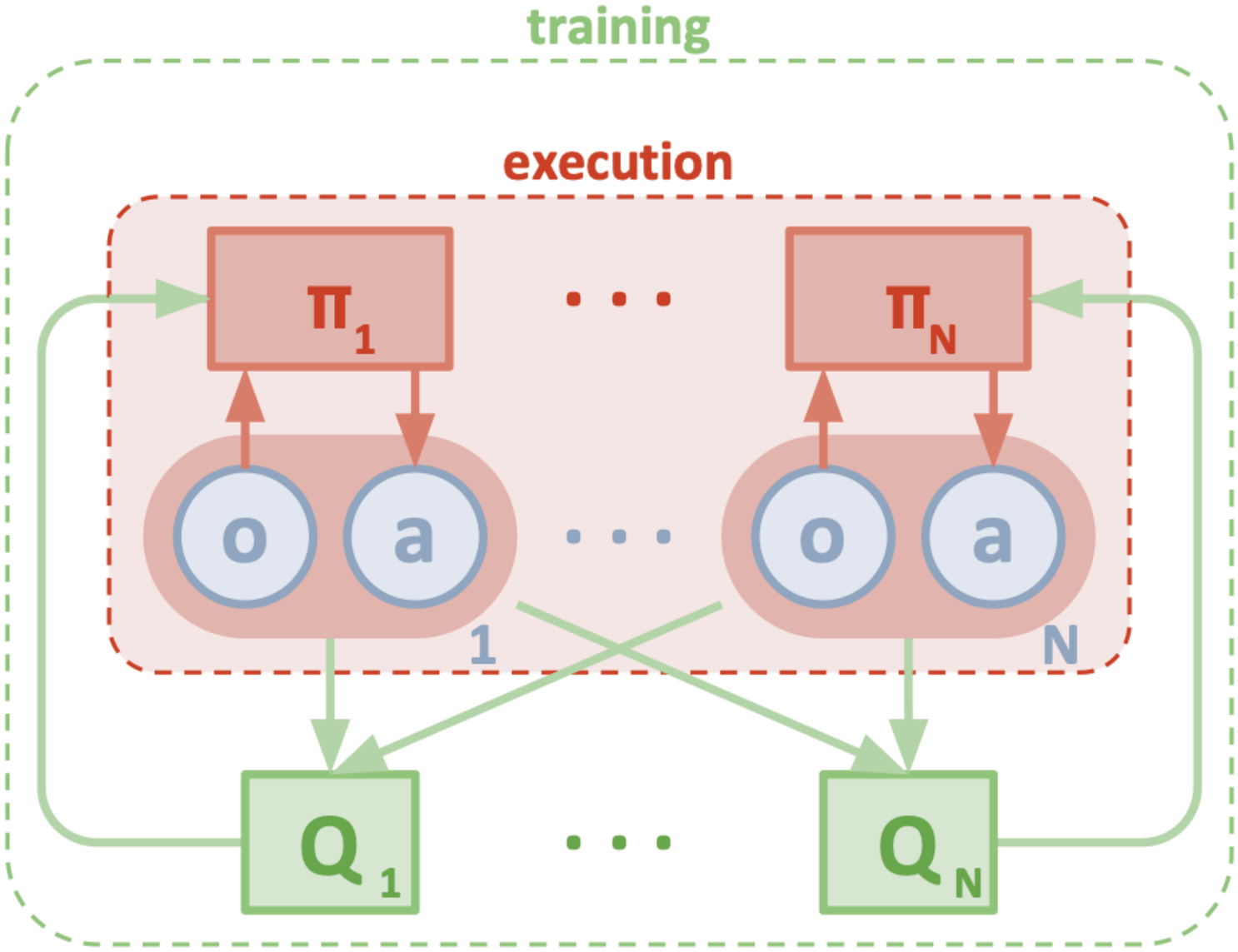
Multi-Agent Actor Critic: More concretely

- Consider a game with N agents with policies parameterized by $\theta = \{\theta_1, \ldots, \theta_N\}$, and let $\pi = \{\pi_1, \ldots, \pi_N\}$ be the set of all agent policies
- Then we can write the gradient of the expected return for agent i, $J(\theta_i) = \mathbb{E}[R_i]$ as:

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{s \sim p^{\mu}, a_i \sim \pi_i} \left[\nabla_{\theta_i} \log \pi_i(a_i \,|\, o_i) Q_i^{\pi}(\mathbf{x}, a_1, \ldots, a_N) \right]$$

- Here $Q_i^{\pi}(\mathbf{x}, a_1, \ldots, a_N)$ is a centralized action-value function that takes as input the actions of all agents, $a_1, \ldots, a_N$ in addition to state information $\mathbf{x}$, and outputs the Q-value for agent i
- In the simplest case, $\mathbf{x}$ could consist of the observations of all agents, $\mathbf{x} = (o_1, \ldots, o_N)$
- Since each Q_i^{π} is learned separately, agents can have arbitrary reward structures, including conflicting rewards in a competitive setting

Decentralized actor and Centralized critic



Multi-Agent Actor Critic: Deterministic policies

- Consider N continuous policies μ_{θ_i} w.r.t. parameters θ_i, the gradient can be written

$$\nabla_{\theta_i} J(\mu_i) = \mathbb{E}_{\mathbf{x}, a \sim \mathscr{D}} \left[\nabla_{\theta_i} \mu_i(a_i \,|\, o_i) \, \nabla_{a_i} Q_i^{\mu}(\mathbf{x}, a_1, \ldots, a_N) \Big|_{a_i = \mu_i(o_i)} \right]$$

- The experience replay buffer $\mathscr{D}$ contains the tuples $(\mathbf{x}, \mathbf{x}', a_1, \ldots, a_N, r_1, \ldots, r_N)$, recording experiences of all agents

- The centralized action-value function Q_i^{μ} is updated as:

$$\mathscr{L}(\theta_i) = \mathbb{E}_{\mathbf{x}, a, r, \mathbf{x}'} \left[\Big(Q_i^{\mu}(\mathbf{x}, a_1, \ldots, a_N) - y \Big)^2 \right], \qquad y = r_i + \gamma \, Q_i^{\mu'}(\mathbf{x}', a_1', \ldots, a_N') \Big|_{a_j' = \mu_j'(o_j)}$$

- where $\mu' = \{\mu_{\theta_1'}, \ldots, \mu_{\theta_N'}\}$ is the set of target policies with delayed parameters θ_i'

MADDPG: Motivations

- A primary motivation behind MADDPG is that, if we know the actions taken by all agents, the environment is stationary even as policies change, since for any $\pi_i \neq \pi_i'$

$$P(s' \mid s, a_1, \ldots, a_N, \pi_1, \ldots, \pi_N) = P(s' \mid s, a_1, \ldots, a_N) = P(s' \mid s, a_1, \ldots, a_N, \pi_1', \ldots, \pi_N')$$

- This is not the case if we do not explicitly condition on the actions of other agents
- Note that we require the policies of other agents to apply an update
- Knowing the observations and policies of other agents is not a restrictive assumption; if our goal is to train agents to exhibit complex communicative behaviour in simulation, this information is often available to all agents
- We can relax this assumption by learning policies of other agents from observations

MADDPG: Inferring Policies of Other Agents

- To remove this assumption, each agent i maintain an approximation $\hat{\mu}_{\phi_i^j}$ (henceforth $\hat{\mu}_i^j$) to the true policy of agent j, μ_j
- This approximate policy is learned by maximizing the log probability of agent j's actions

$$\mathscr{L}(\phi_i^j) = -\mathbb{E}_{o_j, a_j}\left[\log \hat{\mu}_i^j(a_j \mid o_j) + \lambda H(\hat{\mu}_i^j)\right]$$

where H is the entropy of the policy distribution.

- With the approximate policies, y can be replaced by an approximate value $\hat{y}$ calculated

$$y = r_i + \gamma\, Q_i^{\mu'}(\mathbf{x}', a_1', \ldots, a_N')\Big|_{a_j' = \mu_j'(o_j)}$$

where $\hat{\mu}'^j_i$ denotes the target network for the approximate policy $\hat{\mu}_i^j$

MADDPG: Algorithm

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma\, Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)\big|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(o_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

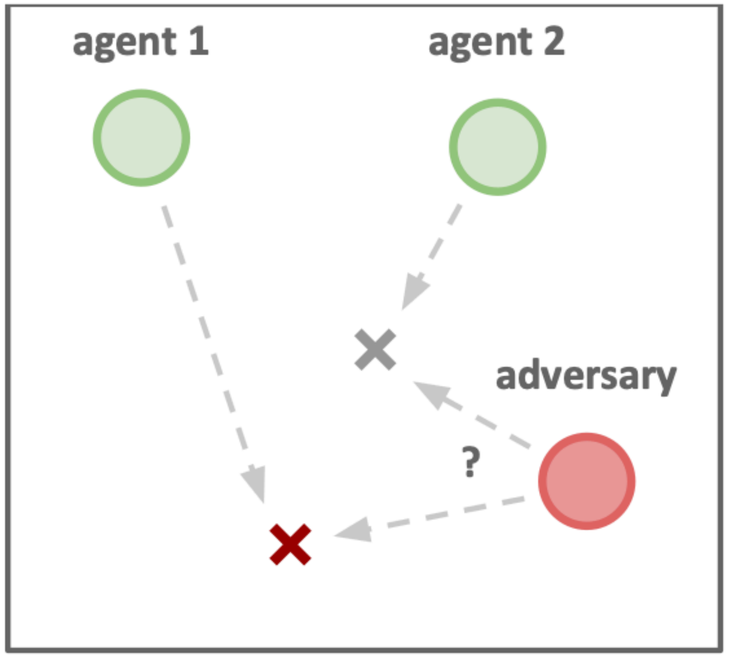
    end for
    Update target network parameters for each agent i:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$

  end for
end for

Multi-Agent Particle Environment: simple_adversary

- 1 adversary (red), 3 good agents (green), 2 landmarks
- All agents observe position of landmarks and other agents
- One landmark is the target landmark
- 5 actions: do nothing, move up, down, left, right

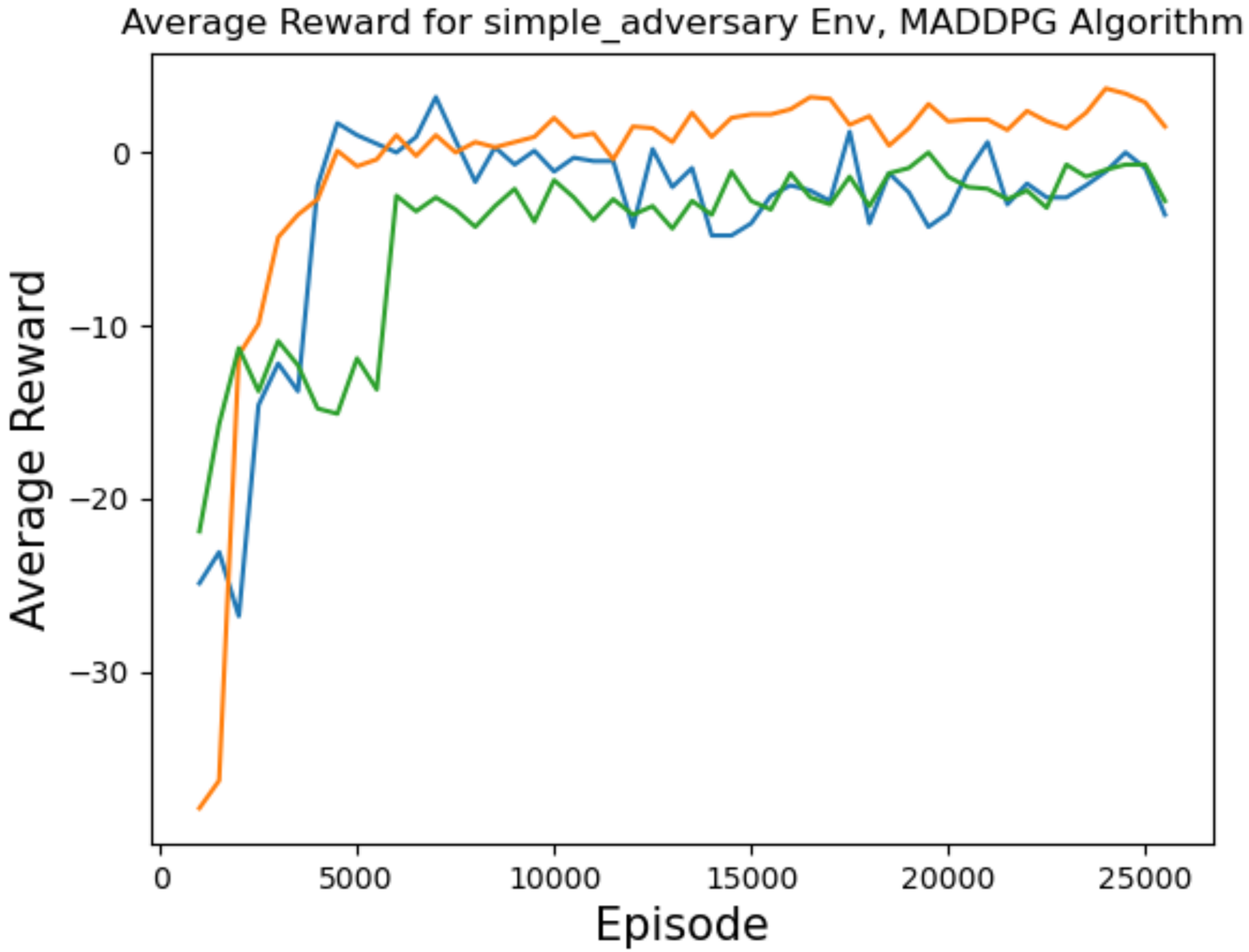


- Good agents rewarded based on how close one of them is to the target landmark, but negatively rewarded if the adversary is close to target landmark
- Adversary is rewarded based on how close it is to the target, but it doesn't know which landmark is the target landmark
- Good agents have to split and cover all landmarks to deceive the adversary

MADDPG: Implementation by Phil Tabor

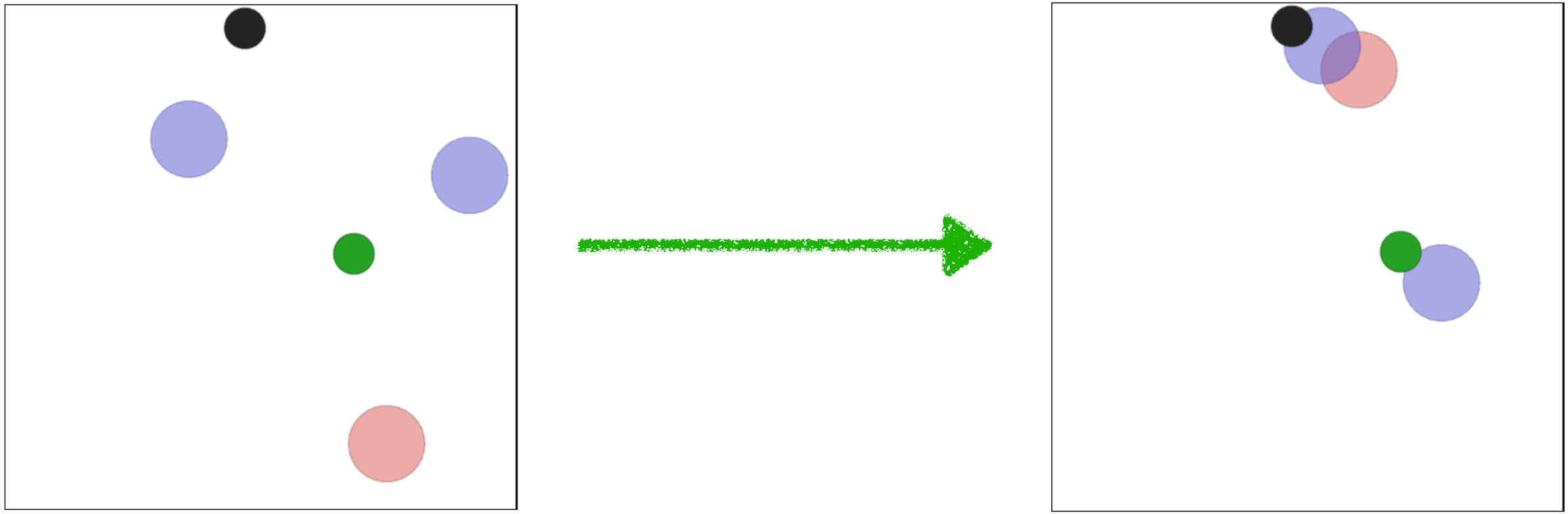
- https://github.com/philtabor/Multi-Agent-Deep-Deterministic-Policy-Gradients

- Results:



MADDPG: Implementation by Phil Tabor

- https://github.com/philtabor/Multi-Agent-Deep-Deterministic-Policy-Gradients
- Results:



MADDPG: class MultiAgentReplayBuffer

```
class MultiAgentReplayBuffer:

  def __init__(self, max_size, critic_dims, actor_dims, n_actions, n_agents, batch_size):
      self.mem_size = max_size
      self.mem_cntr = 0
      self.n_agents = n_agents
      self.actor_dims = actor_dims
      self.batch_size = batch_size
      self.n_actions = n_actions
      self.state_memory = np.zeros((self.mem_size, critic_dims))
      self.new_state_memory = np.zeros((self.mem_size, critic_dims))
      self.reward_memory = np.zeros((self.mem_size, n_agents))
      self.terminal_memory = np.zeros((self.mem_size, n_agents), dtype=bool)
      self.init_actor_memory()

  def init_actor_memory(self):
      self.actor_state_memory = []
      self.actor_new_state_memory = []
      self.actor_action_memory = []
      for i in range(self.n_agents):
        self.actor_state_memory.append(np.zeros((self.mem_size, self.actor_dims[i])))
        self.actor_new_state_memory.append(np.zeros((self.mem_size,self.actor_dims[i])))
        self.actor_action_memory.append(np.zeros((self.mem_size, self.n_actions)))

  def ready(self):
    if self.mem_cntr >= self.batch_size:
      return True
```

MADDPG: class MultiAgentReplayBuffer

```
def store_transition(self, raw_obs, state, action, reward, raw_obs_, state_, done):
   index = self.mem_cntr % self.mem_size
   for agent_idx in range(self.n_agents):
     self.actor_state_memory[agent_idx][index] = raw_obs[agent_idx]
     self.actor_new_state_memory[agent_idx][index] = raw_obs_[agent_idx]
     self.actor_action_memory[agent_idx][index] = action[agent_idx]
   self.state_memory[index] = state
   self.new_state_memory[index] = state_
   self.reward_memory[index] = reward
   self.terminal_memory[index] = done
   self.mem_cntr += 1

 def sample_buffer(self):
   max_mem = min(self.mem_cntr, self.mem_size)
   batch = np.random.choice(max_mem, self.batch_size, replace=False)
   states = self.state_memory[batch]
   rewards = self.reward_memory[batch]
   states_ = self.new_state_memory[batch]
   terminal = self.terminal_memory[batch]
   actor_states = []
   actor_new_states = []
   actions = []
   for agent_idx in range(self.n_agents):
     actor_states.append(self.actor_state_memory[agent_idx][batch])
     actor_new_states.append(self.actor_new_state_memory[agent_idx][batch])
     actions.append(self.actor_action_memory[agent_idx][batch])
   return actor_states, states, actions, rewards, actor_new_states, states_, terminal
```

MADDPG: class CriticNetwork

```
def obs_list_to_state_vector(observation):
    state = np.array([])
    for obs in observation:
        state = np.concatenate([state, obs])
    return state

class CriticNetwork(nn.Module):
    def __init__(self, beta, input_dims, fc1_dims, fc2_dims, n_agents, n_actions, name, chkpt_dir):
        super(CriticNetwork, self).__init__()

        self.chkpt_file = os.path.join(chkpt_dir, name)
        self.fc1 = nn.Linear(input_dims+n_agents*n_actions, fc1_dims)
        self.fc2 = nn.Linear(fc1_dims, fc2_dims)
        self.q = nn.Linear(fc2_dims, 1)
        self.optimizer = optim.Adam(self.parameters(), lr=beta)
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, state, action):
        x = F.relu(self.fc1(T.cat([state, action], dim=1)))
        x = F.relu(self.fc2(x))
        q = self.q(x)
        return q

    def save_checkpoint(self):
        T.save(self.state_dict(), self.chkpt_file)

    def load_checkpoint(self):
        self.load_state_dict(T.load(self.chkpt_file))
```

MADDPG: class ActorNetwork

```
class ActorNetwork(nn.Module):
  def __init__(self, alpha, input_dims, fc1_dims, fc2_dims, n_actions, name, chkpt_dir):
      super(ActorNetwork, self).__init__()
      self.chkpt_file = os.path.join(chkpt_dir, name)
      self.fc1 = nn.Linear(input_dims, fc1_dims)
      self.fc2 = nn.Linear(fc1_dims, fc2_dims)
      self.pi = nn.Linear(fc2_dims, n_actions)
      self.optimizer = optim.Adam(self.parameters(), lr=alpha)
      self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
      self.to(self.device)

  def forward(self, state):
      x = F.relu(self.fc1(state))
      x = F.relu(self.fc2(x))
      pi = T.softmax(self.pi(x), dim=1)
      return pi

  def save_checkpoint(self):
      T.save(self.state_dict(), self.chkpt_file)

  def load_checkpoint(self):
      self.load_state_dict(T.load(self.chkpt_file))
```

MADDPG: class Agent

```
class Agent:
  def __init__(self,actor_dims,critic_dims,n_actions,n_agents,agent_idx,chkpt_dir,
               alpha=0.01,beta=0.01,fc1=64,fc2=64,gamma=0.95,tau=0.01):

    self.gamma = gamma
    self.tau = tau
    self.n_actions = n_actions
    self.agent_name = 'agent_%s' % agent_idx
    self.actor=ActorNetwork(alpha,actor_dims,fc1,fc2,n_actions,chkpt_dir=chkpt_dir,name=self.agent_name+'_actor')
    self.critic=CriticNetwork(beta,critic_dims,fc1,fc2,n_agents,n_actions,chkpt_dir=chkpt_dir,name=self.agent_name+'_critic')
    self.target_actor=ActorNetwork(alpha,actor_dims,fc1,fc2,n_actions,chkpt_dir=chkpt_dir,name=self.agent_name+'_target_actor')
    self.target_critic=CriticNetwork(beta,critic_dims,fc1,fc2,n_agents,n_actions,chkpt_dir=chkpt_dir,
                                     name=self.agent_name+'_target_critic')

    self.update_network_parameters(tau=1)

  def update_network_parameters(self, tau=None):
    if tau is None:
      tau = self.tau
    target_actor_params = self.target_actor.named_parameters()
    actor_params = self.actor.named_parameters()
    target_actor_state_dict = dict(target_actor_params)
    actor_state_dict = dict(actor_params)
    for name in actor_state_dict:
      actor_state_dict[name] = tau*actor_state_dict[name].clone() + (1-tau)*target_actor_state_dict[name].clone()
    self.target_actor.load_state_dict(actor_state_dict)
    target_critic_params = self.target_critic.named_parameters()
    critic_params = self.critic.named_parameters()
    target_critic_state_dict = dict(target_critic_params)
    critic_state_dict = dict(critic_params)
    for name in critic_state_dict:
      critic_state_dict[name] = tau*critic_state_dict[name].clone() + (1-tau)*target_critic_state_dict[name].clone()
    self.target_critic.load_state_dict(critic_state_dict)
```

MADDPG: class Agent

```
  def choose_action(self, observation):
        state = T.tensor([observation],dtype=T.float).to(self.actor.device)
        actions = self.actor.forward(state)
        noise = T.rand(self.n_actions).to(self.actor.device)
        action = actions + noise
        return action.detach().cpu().numpy()[0]
```

```
  def save_models(self):
    self.actor.save_checkpoint()
    self.target_actor.save_checkpoint()
    self.critic.save_checkpoint()
    self.target_critic.save_checkpoint()

  def load_models(self):
    self.actor.load_checkpoint()
    self.target_actor.load_checkpoint()
    self.critic.load_checkpoint()
    self.target_critic.load_checkpoint()
```

MADDPG: class MADDPG

```
class MADDPG:
  def __init__(self,actor_dims,critic_dims,n_agents,n_actions,scenario='simple',
               alpha=0.01,beta=0.01,fc1=64,fc2=64, gamma=0.99, tau=0.01, chkpt_dir='tmp/maddpg/'):
    self.agents = []
    self.n_agents = n_agents
    self.n_actions = n_actions
    chkpt_dir += scenario
    for agent_idx in range(self.n_agents):
      self.agents.append(Agent(actor_dims[agent_idx],critic_dims,n_actions,n_agents,agent_idx,
                               alpha=alpha,beta=beta,chkpt_dir=chkpt_dir))

  def save_checkpoint(self):
    print('... saving checkpoint ...')
    for agent in self.agents:
      agent.save_models()

  def load_checkpoint(self):
    print('... loading checkpoint ...')
    for agent in self.agents:
      agent.load_models()

  def choose_action(self, raw_obs):
    actions = []
    for agent_idx, agent in enumerate(self.agents):
      action = agent.choose_action(raw_obs[agent_idx])
      actions.append(action)
    return actions
```

MADDPG: class MADDPG

```
    def learn(self, memory):
        if not memory.ready():
            return

        actor_states, states, actions, rewards, actor_new_states, states_, dones = memory.sample_buffer()
        device = self.agents[0].actor.device
        states = T.tensor(states, dtype=T.float).to(device)
        actions = T.tensor(actions, dtype=T.float).to(device)
        rewards = T.tensor(rewards).to(device)
        states_ = T.tensor(states_, dtype=T.float).to(device)
        dones = T.tensor(dones).to(device)
        all_agents_new_actions = []
        all_agents_new_mu_actions = []
        old_agents_actions = []

        for agent_idx, agent in enumerate(self.agents):
            new_states = T.tensor(actor_new_states[agent_idx], dtype=T.float).to(device)
            new_pi = agent.target_actor.forward(new_states)
            all_agents_new_actions.append(new_pi)
            mu_states = T.tensor(actor_states[agent_idx], dtype=T.float).to(device)
            pi = agent.actor.forward(mu_states)
            all_agents_new_mu_actions.append(pi)
            old_agents_actions.append(actions[agent_idx])

        new_actions = T.cat([acts for acts in all_agents_new_actions], dim=1)
        mu = T.cat([acts for acts in all_agents_new_mu_actions], dim=1)
        old_actions = T.cat([acts for acts in old_agents_actions],dim=1)

        for agent_idx, agent in enumerate(self.agents):
            critic_value_ = agent.target_critic.forward(states_, new_actions).flatten()
            critic_value_[dones[:,0]] = 0.0
            critic_value = agent.critic.forward(states, old_actions).flatten()
            target = rewards[:,agent_idx] + agent.gamma*critic_value_
            critic_loss = F.mse_loss(target, critic_value)
            agent.critic.optimizer.zero_grad()
            critic_loss.backward(retain_graph=True)
            agent.critic.optimizer.step()
            actor_loss = agent.critic.forward(states, mu).flatten()
            actor_loss = -T.mean(actor_loss)
            agent.actor.optimizer.zero_grad()
            actor_loss.backward(retain_graph=True)
            agent.actor.optimizer.step()
            agent.update_network_parameters()
```

MADDPG: Main loop

```
os.makedirs("tmp/maddpg/simple_adversary/", exist_ok=True)
scenario = 'simple_adversary'
env = make_env(scenario)
n_agents = env.n
actor_dims = []
for i in range(n_agents):
    actor_dims.append(env.observation_space[i].shape[0])

critic_dims = sum(actor_dims)
n_actions = env.action_space[0].n
maddpg_agents = MADDPG(actor_dims, critic_dims, n_agents, n_actions, fc1=64, fc2=64,  alpha=0.01,beta=0.01,
                       scenario=scenario, chkpt_dir='tmp/maddpg/')

memory = MultiAgentReplayBuffer(1000000, critic_dims, actor_dims, n_actions, n_agents, batch_size=1024)
PRINT_INTERVAL = 500
N_GAMES = 50000
MAX_STEPS = 25
total_steps = 0
score_history = []
evaluate = False
best_score = 0
if evaluate:
    maddpg_agents.load_checkpoint()
```

MADDPG: Main loop

```
for i in range(N_GAMES):
    obs = env.reset()
    score = 0
    done = [False]*n_agents
    episode_step = 0
    while not any(done):
        if evaluate:
            env.render()
        actions = maddpg_agents.choose_action(obs)
        obs_, reward, done, info = env.step(actions)
        state = obs_list_to_state_vector(obs)
        state_ = obs_list_to_state_vector(obs_)
        if episode_step >= MAX_STEPS:
            done = [True]*n_agents
        memory.store_transition(obs, state, actions, reward, obs_, state_, done)
        if total_steps % 100 == 0 and not evaluate:
            maddpg_agents.learn(memory)
        obs = obs_
        score += sum(reward)
        total_steps += 1
        episode_step += 1
    score_history.append(score)
    avg_score = np.mean(score_history[-100:])
    if not evaluate:
        if avg_score > best_score:
            maddpg_agents.save_checkpoint()
            best_score = avg_score
    if i % PRINT_INTERVAL == 0 and i > 0:
        print('episode', i, 'average score {:.1f}'.format(avg_score))
```