Collaboration and Competition Project Report

1. Project Overview and Environment Introduction

The Environment

For this project, you will work with the Tennis environment.

In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

The task is episodic, and in order to solve the environment, your agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically,

After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores. This yields a single score for each episode. The environment is considered solved, when the average (over 100 episodes) of those scores is at least +0.5.

2. Learning Algorithm

There are a number of applications that invlove interaction between multiple agents, for example, multi-robot control, multiplayer games, social analysis,trading etc.

Traditional reinforcement learning approaches such as Q-Learning or policy gradient are poorly suited to multi-agent environments.

The OpenAI baselines Tensorflow implementation and Ilya Kostrikov's Pytorch implementation of DDPG were used as references. After the majority of this codebase was complete, OpenAI released their code for MADDPG. Our reference includes the MADDPG algorithm presented in OpenAI's publication "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments" (Lowe et al., https://arxiv.org/pdf/1706.02275.pdf (https://arxiv.org/pdf/1706.02275.pdf))

MADDPG (Multi Agent Deep Deterministic Policy Gradient (MADDPG))

The primary motivation behind MADDPG is that if we know the actions taken by all agents, the environment is stationary even as the policies change, since $P(s'|s,a1,a2,\pi1,\pi2) = P(s'|s,a1,a2) = P(s'|s,a1,a2,\pi'1,\pi'2)$ for any $\pi i \neq \pi' i$. This is not the case if we do not explicitly condition on the actions of other agents, as done by most traditional RL algorithms.

In MADDPG, each agent's critic is trained using the observations and actions from all the agents, whereas each agent's actor is trained using just its own observations. This allows the agents to be effectively trained without requiring other agents' observations during inference (because the actor is only dependent on its own observations). Here is the disgram of the MADDPG algorithm.

The oseudo-code of the algorithm looks like this:

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

Experience replay: a replay buffer samples experience to update neural network parameters. During each trajectory roll-out, we save all the experience tuples (state, action, reward, next_state) and store them in a finite-sized cache — a “replay buffer.” Then, we sample random mini-batches of experience from the replay buffer when we update the value and policy networks. In optimization asks, we want the data to be independently distributed. This fails to be the case when we optimize a sequential decision process in an on-policy way, because the data then would not be independent of each other. When we store them in a replay buffer and take random batches for training, we overcome this issue.

Actor(Policy) & Critic(Value) network updates: The value network is updated similarly as is done in Q-learning. The updated Q value is obtained by the Bellman equation. However, in DDPG, the next-state Q values are calculated with the target value network and target policy network. Then, we minimize the mean-squared loss between the updated Q value and the original Q value. For the policy function, our objective is to maximize the expected return. To calculate the policy loss, we take the derivative of the objective function with respect to the policy parameter.

The Ornstein–Uhlenbeck process is a stationary Gauss–Markov process, which means that it is a Gaussian process, a Markov process, and is temporally homogeneous. In fact, it is the only nontrivial process that satisfies these three conditions, up to allowing linear transformations of the space and time variables. Over time, the process tends to drift towards its mean function: such a process is called mean-reverting.

3. Training Process of the project

The neural network architecture used for this project can be found here in the model.py file of the source code.

The network architecture is defined using pytorch.

It is observed that an average score of +-.5019 over 1673 consecutive episodes.

For each agent's actor, a two-layer neural network with 24 units(24 states) in the input layer, 400 in the first hidden layer, 300 units in the second hidden layer, and 2 units(2 actions) in the output layer is used. For each agent's critic, a two-layer neural network with 48 units in the input layer, 400 units in the first hidden layer (and both agents' actions are concatenated with the output of the input layer), 300 units in the second hidden layer, and 1 unit in the output layer.

Epsiode 100 Average Score 0.00

Epsiode 200 Average Score 0.02

Epsiode 300 Average Score 0.02

Epsiode 400 Average Score 0.00

Epsiode 500 Average Score 0.00

Epsiode 600 Average Score 0.00

Epsiode 700 Average Score 0.00

Epsiode 800 Average Score 0.00

Epsiode 900 Average Score 0.00

Epsiode 1000 Average Score 0.00

Epsiode 1100 Average Score 0.01

Epsiode 1200 Average Score 0.07

Epsiode 1300 Average Score 0.02

Epsiode 1400 Average Score 0.01

Epsiode 1500 Average Score 0.07

Epsiode 1600 Average Score 0.16

Epsiode 1700 Average Score 0.23

Episode 1773 Average Score 0.50

hyperparameters

lr_actor = 1e-3 # learning rate of Actor

lr_critic = 1e-3 # learning rate of Critic

weight_decay = 0. # L2 weight decay

gamma = 0.99 # discount factor

tau = 1e-3 # soft update parameter

batch_size = 512 # batch size to sample from replay buffer

buffer_size = int(1e5) # max size (capacity) of the replay buffer

n_agents = 2 # number of agents

The actor network's learning rate was 1e-3 and the critic's was 1e-3. Adjusting the learning rate can affect how fast we can converge. or the target networks, a soft update factor of τ=1e-3 was used — this value was experimentally determined to find the right balance between learning speed and learning stability. A batch size of 512 was used. Additionally, a discount factor of 0.99 was used to force the agents to be "cognizant" of their actions' long term consequences.

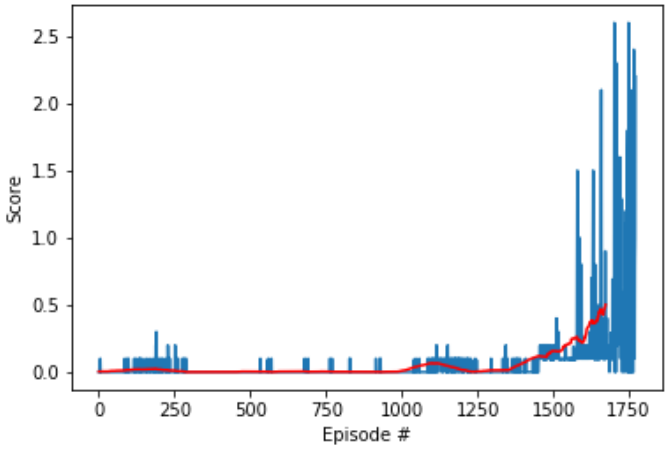
Then the neural network architecture is studied to see how to improve the training process. The initial network consists of two layers with ReLu as the activation function.

Batch normalization is used. Running computations on large inputs can inhibut learning. Normalization scales the features to be within the same range, typically between 0 and 1. It could be placed in between first layer and second layer. Or it can be done after the second layer or after each layer.

Gradient Clipping can help learning by clipping outsized gradients. By using torch.nn.utils.clip_grad*norm* function and setting the function to "clip" the norm of the gradients at 1, it places an upper limit on the size of the parameter updates, and prevents them from growing exponentially.

results

Environment solved in 1673 episodes! Average Score: 0.501900



4. Ideas for Future Improvement

1. **further tuning of the model**

We can tune the parameters in the nueral network to see if it will improve the convergence and performance of the training.

1. **Other algorithms**

We can train the agents using Multi Agent Proximal Policy Optimization (MAPPO) to see if it is more robust compared to MADDPG. Additionally, I would like to incorporate Prioritized Experience Replay to better utilize important but less frequently seen experiences.

1. **Prioritized Experience Replay**

Prioritized experienced replay is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability. It will be good to incorporate Prioritized Experience Replay to better utilize important but less frequently seen experiences.