Programming Assignment 2:
MADDPG for a Ranger/Poacher Environment

Introduction

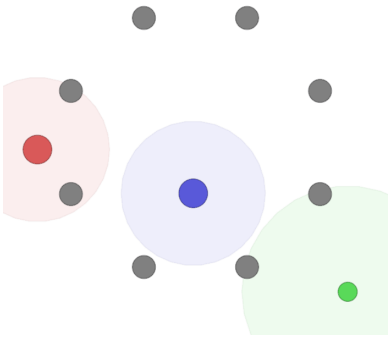
In this assignment, you will be implementing the Multi-Agent Deep Deterministic Policy Gradient (MADDPG) algorithm and testing it in a ranger/poacher environment.

In MADDPG, each agent maintains an actor network (i.e., their policy) and a critic network (i.e., their Q-function). MADDPG works with deterministic, continuous policies μ. This means that the output of a policy is an action in some continuous action space $\mathcal{A}$ and not a probability distribution over actions. Additionally, MADDPG has a centralized critic, meaning that each agent's Q-function takes as input the observations and actions of all agents. Each agent still maintains their own critic network, since agents can receive different rewards.

For more details on the algorithm, you can refer to the original paper[1] or to the slides from Lecture 8. The code for this assignment is sourced from the original MADDPG paper.

Environment: Ranger/Poacher Scenario

We will be using MADDPG in a scenario from the popular Multi-Agent Particle Environment[2]. The scenario is implemented in the file "multiagent-particle-envs/multiagent/scenarios/ranger_poacher_uav.py". Below is an image of the environment.



This environment includes 3 agents: 1 poacher (red), 1 ranger (blue), and 1 UAV (green). Each agent's actions correspond to movement in a 2-dimensional world. In addition, there are 8 wildlife animals (grey) that do not move. Each agent has a particular movement speed and observation radius. Agents are able to observe the positions and velocities of any other agents within their observation radius. The locations of the animals are public information and are known to all agents from the beginning of the game. Additionally, the ranger and the UAV are in the same team with real-time information sharing and shared rewards. Thus, the ranger will know the poacher's location when the UAV observes the poacher.

The ranger can catch the poacher immediately when it collides (in Euclidean distance) with the poacher. When the ranger catches the poacher, the ranger/UAV gets a reward of 1, the poacher gets a reward of -1 and the game ends.

[1] `https://arxiv.org/pdf/1706.02275.pdf`
[2] `https://github.com/openai/multiagent-particle-envs`

The poacher can successfully capture an animal if he is in the same location as the animal for 20 consecutive timesteps. When the poacher captures an animal, the ranger/UAV gets a reward of -5, the poacher gets a reward of 1.25, and the game ends.

Notation

We use the following notation.

- $o_i \in \mathcal{O}$ is the observation for agent i in some timestep. $o = (o_1, \ldots, o_N)$.
- $a_i \in \mathcal{A}$ is the action of agent i in some timestep. $a = (a_1, \ldots, a_N)$.
- $r_i \in \mathbb{R}$ is the reward of agent i in some timestep. $r = (r_1, \ldots, r_N)$.
- $\mathcal{D}$ is the experience replay buffer, which contains tuples of experiences (o, a, r, o').
- $\mu_i : \mathcal{O} \to \mathcal{A}$ is the policy for agent i. $\mu = (\mu_1, \ldots, \mu_N)$.
- $Q_i^{\mu} : \mathcal{O}^N \times \mathcal{A}^N \to \mathbb{R}$ is the Q-function for agent i, given that agents follow policies μ. This function takes as input the observations and actions of all agents.
- $\overline{Q}_i^{\overline{\mu}}$ and $\overline{\mu}_i$ are the target Q-function and target policy for agent i.
- γ is the discount factor.

Code Setup

To ensure the correct dependencies for the code, we will be using Google Colab. To setup the Colab environment, follow these steps:

1. Download the "PA2" folder and upload it to your Google Drive. This folder contains the Python implementations of MADDPG and the Multi-Agent Particle Environment that you will be modifying.
2. Open the Colab notebook here, and make a copy to your Google Drive: `https://colab.research.google.com/drive/1ASW2c9Y3UAC_nGxJk_AJRMDloCMZrhf4?usp=sharing`. This notebook contains commands to (a) connect to your Google Drive, (b) install the required dependencies, and (c) run the code in the "PA2" folder.

If you prefer to run the code locally, you can instead create an Anaconda environment (with Python 3.7) and install the dependencies as in the Colab notebook.

During the assignment, you'll need to modify the Python code in the "PA2" folder. The easiest way to do this is to open the files in Colab using the file browser and edit them there. After modifying this code, you can run cells in the Colab notebook to execute the code.

Submission

Please submit a zip file with the following items:

- A writeup in pdf format with answers to the requested questions (including the 3 learning curve plots)
- The 2 gifs of learned policies (with and without reward shaping)
- The modified "maddpy.py", "ranger_poacher_uav.py", and "train.py" files

1 Implementing MADDPG [45 points]

In this section, you'll implement parts of the MADDPG algorithm. All of the functions to implement are in the file "maddpg/maddpg/trainer/maddpg.py". **Include this modified code file in your submission.**

The critic is constructed in the function "q_train", and the actor is similarly constructed in the function "p_train". These functions build the computation graph for the Q-function and policy, which later operations and parameter updates will use. You will be implementing several helper functions used to create these computation graphs and pass inputs into them.

You may find the TensorFlow functions "tf.reduce_mean", "tf.concat", and "tf.square" useful during your implementation.

Task 1.1 [5 points] Implement "construct_q_function", which is used to create the Q-function for each agent i: $Q_i^\mu : \mathcal{O}^N \times \mathcal{A}^N \to \mathbb{R}$. "construct_q_function" returns the input to the critic network and the number of outputs that the critic network should have. The input to the critic network should be the concatenation of the actions and observations of all agents. (Note: The code currently uses an MLP as the network architecture for both the critic and actor. Often, a RNN-type architecture is instead used for these networks in order to incorporate past observations.)

Task 1.2 [10 points] Implement "compute_q_loss", which computes the loss used to train the critic network:

$$\mathbb{E}_{(o,a,r,o')\sim\mathcal{D}}\left[\left(Q_i^\mu(o,a_1,\ldots,a_N) - y_i(o,a,r,o')\right)^2\right].$$

$y_i(o,a,r,o')$ is the target Q-value for each sample.

Task 1.3 [5 points] Implement "construct_p_function", which is used to create the policy function for each agent i: $\mu_i : \mathcal{O} \to \mathcal{A}$. Since the action space is continuous in MADDPG, $\mathcal{A} = \mathbb{R}^d$ where d is the dimension of the action space. "construct_p_function" returns the input to the actor network and the number of outputs that the actor network should have.

Task 1.4 [10 points] Implement "compute_p_loss", which computes the loss used to train the actor network. Policies are chosen to maximize the expected reward-to-go, which is equivalent to minimizing the following loss:

$$\mathbb{E}_{(o,a,r,o')\sim\mathcal{D}}\left[-Q_i^\mu(o,a_1,\ldots,a_N)\right].$$

Once this loss is defined, the computation of the policy gradient will be handled automatically by Tensorflow.

Task 1.5 [10 points] Implement "compute_q_targets", which computes the target Q-values that are passed into the loss calculation:

$$y_i(o,a,r,o') = r_i + \gamma \overline{Q}_i^{\overline{\mu}}(o', a'_1, \ldots, a'_N)|_{a'_j = \overline{\mu}_j(o'_j)}.$$

Unlike the previous functions you implemented, which are used to construct the computation graph, this function will be called every time new target Q-values need to be computed.

Task 1.6 [5 points] Implement "update_target_param", which updates the parameters of the target Q and policy networks maintained by each agent in a slightly different manner than was stated in lecture. Target network parameters are updated using a weighted sum of the current target network parameter values and the current actual network parameters: $\overline{\theta} \leftarrow 0.99\overline{\theta} + 0.01\theta$, where θ are the actual network parameters and $\overline{\theta}$ are the corresponding target network parameters.

2 Testing the implementation [15 points]

Task 2.1 [5 points] First, test that your implementation works on the "simple" scenario using the command in the Colab notebook. This scenario features one agent who gets a reward if they navigate to a landmark. Our implementation receives a reward of about -6 after training for 5000 episodes, taking 10-15 seconds per 1000 episodes. Plot the learning curve to show how the mean episode reward changes over time. The x-axis should be the number of episodes trained (up to 5000 episodes), and the y-axis should be an agent's average reward in the last 1000 episodes. You can use the data saved in "results/maddpg_simple/test_agrewards.pkl" for this plot. **Include this plot in your writeup.**

Task 2.2 [5 points] Then, train your implementation on the ranger-poacher scenario for 3000 episodes using the command in the Colab notebook. Our trained policies receive rewards of [-0.06, 0.06, 0.06] after training for all 3000 episodes, taking 60-100 seconds per 500 episodes. Plot the learning curve for each agent (i.e., 3 curves on the same plot), showing on the y-axis each agent's average reward in the last 500 episodes. You can again use the data saved in "results/maddpg_uav/test_agrewards.pkl" for this plot. **Include this plot in your writeup.**

Task 2.3 [5 points] Finally, evaluate your trained policies using the command in the Colab notebook, which also saves a gif of the played episodes. **In your writeup,** describe what the agents' learned policies seem to be doing. **Include the saved gif in your submission.**

3 Reward shaping [40 points]

One issue that makes training difficult in this environment is a sparse reward. Agents only receive a reward at the end of the episode and get no other guidance as to which actions are taking them in the right direction. "Reward shaping" involves defining a custom reward function to use during training, with the goal of learning better policies for the original reward function.

Task 3.1 [15 points] First, design a custom reward function for use in this scenario. **In your writeup,** describe your reward function and explain why you think it will make training easier in this scenario.

Task 3.2 [15 points] Next, implement reward shaping in this scenario with your custom reward. You should modify the scenario "multiagent-particle-envs/multiagent/scenarios/ranger_poacher_uav.py" to add a function that computes your custom reward in the environment. Then, you should modify the file "maddpg/experiments/train.py" so that agents are trained with the rewards you've designed. **Include these modified code files in your submission.**

Task 3.3 [5 points] Train your implementation with reward-shaping on the ranger-poacher scenario for at least 3000 episodes (you can train for longer if you want), and plot the learning curve for each agent as you did in the previous part. Note: Since agents are still evaluated on the original reward function, this plot should use the original reward function. **Include this plot in your writeup.**

Task 3.4 [5 points] Evaluate your trained policies as before, and examine the saved gif. **In your writeup,** answer the following: Does your reward shaping seem to help the agents learn better policies as compared to the case with no reward shaping? Why do you think this might be the case? **Include the saved gif in your submission.**