Project #2 Report

Name: Patrick H. Rupp

Date: 2023/11/19

Course: Udacity Deep Reinforcement Learning

Repo: https://github.com/PHRupp/udacity-project2

Learning Algorithm

I implemented a deep deterministic policy gradient (DDPG) algorithm using the Torch framework for each of the two agents. The actor and critic neural networks are a simple forward pass network with four and three layers respectively, each using the RELU activation function except for the final layer in the actor which uses tanh to produce [-1,1] for each action dimension. The actor network has layers with [24x64, 64x128, 128x64, 64x2], and the critic network has layers with [24x256, 256x256, 256x1]. I included a memory buffer with 100,000 instances where a batch of 512 is sampled. This replay buffer has a shared memory to ensure that each individual agent learned collaborative actions. Additionally, I focused on using a large discount factor to focus on maximizing the value of future rewards. I set a maximum of 20,000 episodes but we reached >=0.5 at 786 and target at 886. I also implemented Ornstein-Uhlenbeck process for adding noise with a noise decay after each model update. However, I had some instability with it so the decay is set to 1 (no decay).

I also spent a few hours attempting a MADDPG algorithm with critic models with shared states/actions. Unfortunately, my implementation of this did not work. The MADDPG implementation failed to show any signs of learning, in fact, it failed to beat a random agent. I also tried to run the random agent to build up the replay buffer before training the MADDPG agents, but this also did not work. The double DDPG implementation provided the only means of solving this problem. Additionally, I ran the DDPG for a long while, and it even achieved up to +1.4 average score (Figure II).

main.py

```
params = {
    'STATE_SIZE': 8*3,
    'ACTION_SIZE': 2,
    'SEED': 8675309,
    'LR_ACTOR': 1e-4,
    'LR_CRITIC': 1e-3,
    'BUFFER_SIZE': int(1e5),
    'TRAIN_BATCH_SIZE': 512,
    'DISCOUNT_FACTOR': 0.99,
    'TAU': 1e-3,
    'WEIGHT_DECAY': 0.0001,
    'UPDATE_ITERATION': 10,
    'NUM_UPDATES_PER_INTERVAL': 10,
    'NOISE_DECAY': 1,
    'NUM_EPISODES': 20000,
    'MAX_TIMESTEPS': 999,
    'THRESHOLD': 0.5,
    'PLOT_ITER': 100,
}
```

Project #2 Report

Plot of Rewards

The results of the agent training is shown below with the average score in orange. These results are stagnant for nearly 500 episodes before they grow exponentially after 600 episodes. I had set the requirement for average score to be >= 0.5 over a window size of 100 episodes. The plots below shows that the problem is officially solved by episode 886. Figure II shows that as the agents continue to learn, they also periodically increase and decrease their performance. This is likely due to the instability of this collaborative problem.

Log Output:

Results were captured in the log file.

INFO: Episode: 1 Avg Scores: [0.00, -0.01] Scores: [0.00, -0.01]
INFO: Episode: 2 Avg Scores: [0.00, -0.01] Scores: [0.00, -0.01]
INFO: Episode: 3 Avg Scores: [0.00, -0.01] Scores: [0.00, -0.01]
INFO: Episode: 4 Avg Scores: [-0.00, -0.01] Scores: [-0.01, 0.00]
INFO: Episode: 5 Avg Scores: [-0.00, -0.01] Scores: [0.00, -0.01]
INFO: Episode: 6 Avg Scores: [-0.00, -0.01] Scores: [0.00, -0.01]
…
INFO: Episode: 881 Avg Scores: [0.44, 0.45] Scores: [2.29, 2.30]
INFO: Episode: 882 Avg Scores: [0.44, 0.45] Scores: [-0.01, 0.10]
INFO: Episode: 883 Avg Scores: [0.44, 0.45] Scores: [0.40, 0.39]
INFO: Episode: 884 Avg Scores: [0.45, 0.46] Scores: [1.59, 1.70]
INFO: Episode: 885 Avg Scores: [0.48, 0.49] Scores: [2.50, 2.60]
INFO: Episode: 886 Avg Scores: [0.50, 0.51] Scores: [2.60, 2.60]
INFO:
Environment solved in 786 episodes! Average Score: 0.50 | 0.51
INFO: Exiting...

Rewards Plot

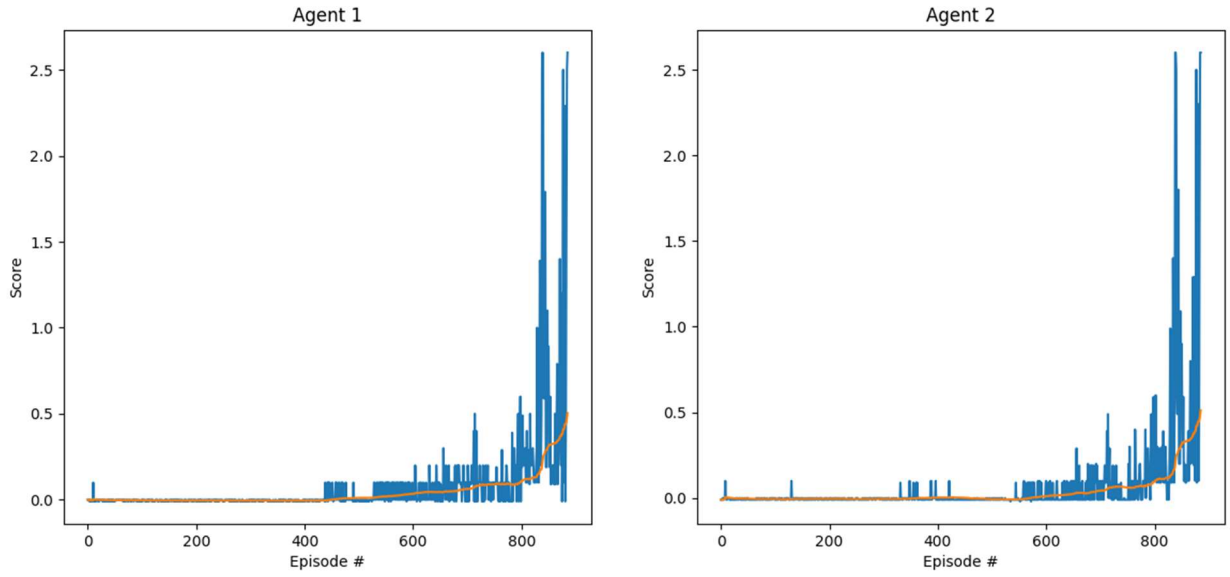

Figure I: Raw scores from each episode until training complete

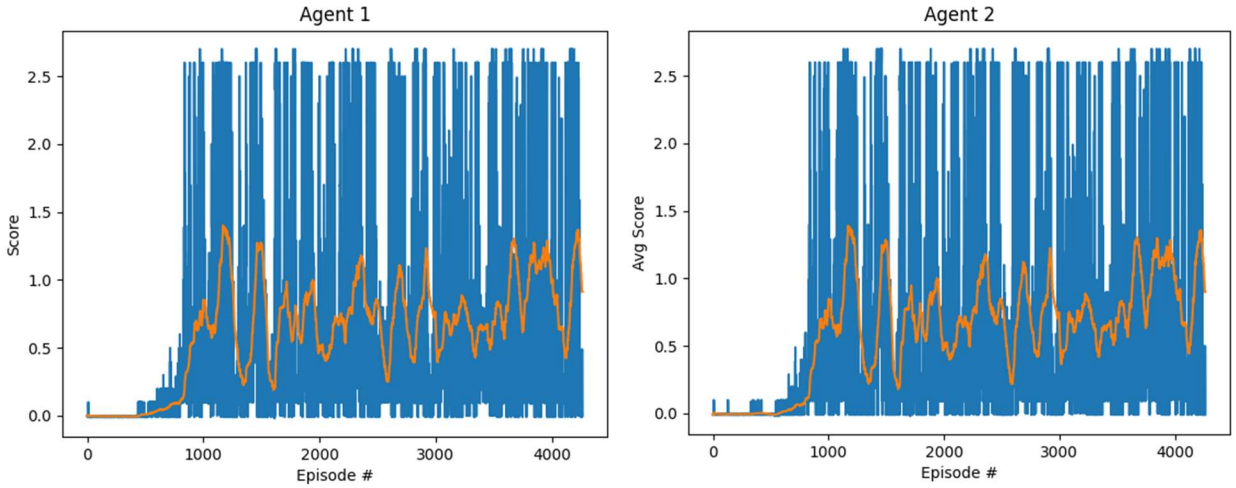

Figure II: Actual Scores and Avg Score Over last 100 episodes with user defined stops.

Ideas for Future Work

One thing that could make this approach significantly better is using an optimization technique for hyperparameter tuning. For this project, I started with some generic numbers and slightly tweaked by hand. This is inefficient and prone to sub-optimal performance. An optimization technique like particle swarm optimization (PSO) on the hyper parameters to algorithmically search for an optimal hyperparameter set which maximizes the score within a given time period or same score within smallest training cycles. Another method might be to use Ray-tune with its distributed processing capabilities to accelerate results maximizing the compute resources available. This would have saved me countless hours of parameter tweaking.