Project #2 Report

Name: Patrick H. Rupp

Date: 2023/11/05

Course: Udacity Deep Reinforcement Learning

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

Learning Algorithm Version 1

I implemented a deep deterministic policy gradient (DDPG) algorithm using the Torch framework. 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]. The actor network has layers with [33x64, 64x128, 128x64, 64x4], and the critic network has layers with [33x256, 256x256, 256x1]. I included a memory buffer with 10,000 instances where a batch of 256 is sampled. Additionally, I focused on using a somewhat large discount factor to focus on maximizing the value of future rewards, but to incentivize recent actions. I set a maximum of 2,000 episodes but we reached >=30.0 at 135 and target at 235. I also implemented Ornstein-Uhlenbeck process for adding noise. I originally tried to add a decay mechanism for slowly minimizing the noise over time, but I had a hard time making that work so this final iteration had the noise_decay at 1 (no decay). I spent a few hours tweaking the various parameters, but most other parameter configurations lead to poor performance.

main.py

```
scores = train(
    env=env,
    agent=DDPGAgent(
        state_size=33,
        action_size=4,
        seed=546879,
        lr_actor=1e-4,
        lr_critic=1e-3,
        buffer_size=int(1e5),
        train_batch_size=256,
        discount_factor=0.95,
        TAU=1e-3,  # update of best parameters
        update_iteration=20,
        weight_decay=0.0001,
        num_updates_per_interval=10,
        noise_decay=1,
    ),
    num_episodes=250,
    max_timesteps=999,
    threshold=30.0,
)
```

Project #2 Report

Plot of Rewards

The results of the agent training is shown below with the score of the last 100 episodes. These results grow fairly linearly from episode 100 until about episode 500 where it begins to plateau slightly with more varying results. The first 100 episodes were slow to learn because of it focusing mostly on random actions to build enough data/experience to learn from. I had set the requirement for average score to be >= 14.00 over a window size of 100 episodes. The plot below shows the actuals scores of each episode and that it consistently grows above the 13.00 threshold starting from episode 512.

Log Output:

Results were captured in the excel document saved within the repo.

INFO: Episode 1 Average Score: 0.96
INFO: Episode 2 Average Score: 1.10
INFO: Episode 3 Average Score: 0.80
INFO: Episode 4 Average Score: 0.67
INFO: Episode 5 Average Score: 0.71
INFO: Episode 6 Average Score: 0.79
…
INFO: Episode 230 Average Score: 28.92
INFO: Episode 231 Average Score: 29.11
INFO: Episode 232 Average Score: 29.35
INFO: Episode 233 Average Score: 29.57
INFO: Episode 234 Average Score: 29.90
INFO: Episode 235 Average Score: 30.12
INFO:TEST:
Environment solved in 135 episodes! Average Score: 30.12
INFO: Exiting...

Rewards Plot

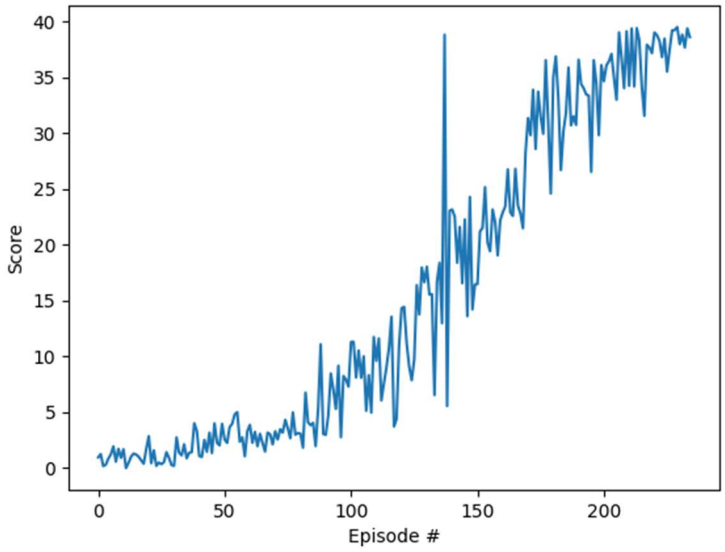

Figure I: Raw scores from each episode until training complete

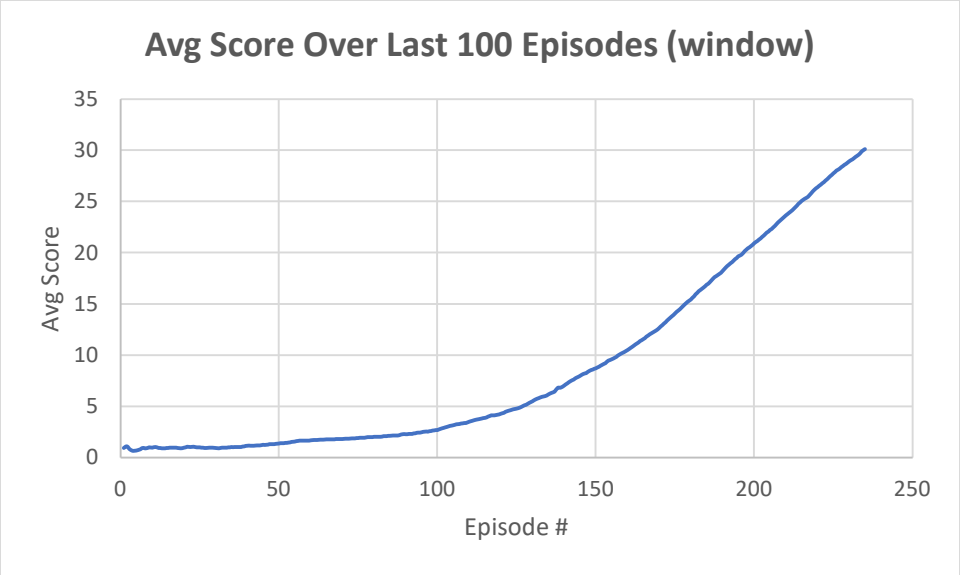

Figure II: Avg Score Over last 100 episodes

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.