Report – P3 Multi Agent RL @Unity ML Agent Toolkit

Krishna Sankar
10/21/18

1. Project & General Notes

- This is another very interesting problem – to train a two-player game where agents control rackets to bounce ball over a net interacting with the Tennis environment in the Unity ML-Agents toolkit. The goal of the environment is that the agents must bounce ball between one another while not dropping or sending ball out of bounds.

  The environment has a two-dimensional continuous action space, corresponding to movement toward (or away from) the net, and jumping. Every entry in the action vector should be a number between -1 and 1.

  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.

  While it is not specified, I found out by experimentation that the environment terminates after 1000 steps.

  The reward structure is much simpler than the normal tennis - 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.

  In order to solve the environment, an agent has to get a minimum score of 0.5 averaged over a window of 100 consecutive episodes.

  Note : One can run the iPython notebook (DDPG_Tennis.ipynb) & train the agent (slow) or use the iPython notebook to run a saved model (fast) or watch the three videos viz. Before_RL-01.m4v, After_RL_Slow-01.m4v and After_RL_Fast-01.m4v (fastest)

2. Algorithm

2.1. Problem Discussion

This is inherently a multi-agent problem, but without the cooperative or competitive interactions. As a result, this problem can be viewed as a multi-agent self-play i.e. two distinct agents with shared replay buffer, centralized critic and a centralized actor. As the agents are on opposite sides of the court, there is very little cooperative or competitive opportunities. Even sharing the policy doesn't add any advantage or disadvantage. If the action also included vectors for direction and force, then each agent separately could anticipate where the ball would land and can prepare beforehand. Because each agent receives its own local observation, the actor is decentralized because the view is different for each agent. Moreover, even the reward calculation is geared towards self-play i.e. max reward of the two agents.

For this project, I used the basic DDPG algorithm with appropriate adjustments from the Multi-agent Actor-Critic paper[1].

2.2. Reinforcement Learning Model

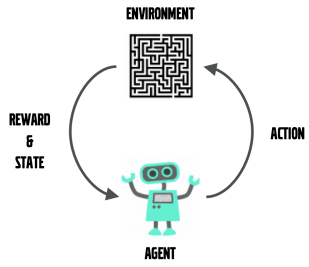


Figure 1

In the classical Reinforcement Learning scenario, an agent interacts with an environment by taking actions. The environment returns a reward as well as an observation snapshot of the state space. The state space observation encapsulates all the cumulative information of all the previous actions. Hence the current state space observation is enough for the agent to take next action. The agent's goal is to select actions that maximizes cumulative future reward. This fits very well with the current problem.

2.3. Deep Deterministic Policy Gradient (DDPG) Algorithm

The challenge in this problem is the continuous action space. So, we need an algorithm that can handle continuous action space. DDPG is an interesting algorithm classified as an "Actor-Critic" method. The authors [2] introduce the algorithm as "a model-free, off-policy *actor-critic algorithm* using deep function approximators that can learn policies in high-dimensional, continuous action spaces".

Some researchers think DDPG is best classified as a DQN method for continuous action spaces. As we will see in the program DDPG uses a replay buffer, it has the target and local networks, usually updated via the polyak-averaging – all are reminiscent of the DQN algorithm. Without going into too much detail, the DDPG algorithm from the paper [2] is as follows. We will look at the implementation in the sections below:

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:

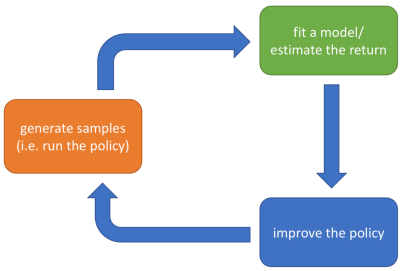
$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

2.4. System Architecture

I liked the simple systems diagram that Prof. Sergey Levine from UC Berkeley uses for his CS294 Deep Reinforcement Learning class [7].

The anatomy of a reinforcement learning algorithm



Let us map this architecture to DDPG.

- The sample generation is via interaction with the environment; but DDPG uses the replay buffer to sample a batch.
  - In that sense, the interaction between the environment and the sample consumption are asynchronous and theoretically can be parallelized
- The policy evaluation is interesting – it is a critic, very similar to the DQN where a Deep Neural Network predicts the Q(s,a) i.e. a q-value for a state and all the actions.
- The actor does the policy improvement using the policy gradient methods.

In short, an interesting combination of methods.

2.5. Implementation - Fundamental Components

The three main components actor, critic and the replay buffer are straightforward and are implemented as three classes

2.6. Agent Implementation

The agent has 3 main components viz:

1. The DDPG Orchestrator which interacts with the environment by taking actions and then unpacking the returned package to rewards, state space et al.
2. It also has to do housekeeping like tracking scores, store high performant models and check when the problem is solved
3. The 3rd component is the most interesting one, which gives the agent the capability to hunt for the right policy.

2.7. Multi-Agent Changes

Because this is a self-play with each agent getting it's own observation of the shared environment, the changes required were to parse the observation into each agent, update the shared replay buffer and then train the actor and the critic. The code does the management of the agents – parse the correct observation, trigger the actor (and critic), store in the replay buffer, get the action and finally package the action array to step the environment.

Note : As a future enhancement, the decentralized actor, centralized critic can be implemented very easily, by creating an actor for each agent and passing the agent id as a parameter. Then the agent class can invoke the right actor while still keeping the replay buffer and the critic common. The soccer challenge will be interesting in this regard, but probably it will need a little more refactoring.

2.8. Experimentation

Once the code is functionally correct the work is to find the right hyper parameters. I did experimentation with the network architecture and the hyper parameters as discussed in the hyperparameters sub section under Results below.

After some hyperparameter search, the optimum network for the actor and critic is a fully connected network with two layers with 32 input units and an output layer with 2 units. Both the actor and critic have the same network parameters as shown below.

```
Actor(
  (model): Sequential(
    (0): Linear(in_features=24, out_features=32, bias=True)
    (1): ReLU()
    (2): Linear(in_features=32, out_features=32, bias=True)
    (3): ReLU()
    (4): Linear(in_features=32, out_features=2, bias=True)
    (5): Tanh()
  )
)
Critic(
  (hc_1): Sequential(
    (0): Linear(in_features=24, out_features=32, bias=True)
    (1): ReLU()
    (2): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (hc_2): Sequential(
    (0): Linear(in_features=34, out_features=32, bias=True)
    (1): ReLU()
    (2): Linear(in_features=32, out_features=1, bias=True)
  )
)
```

2.9. README.md file

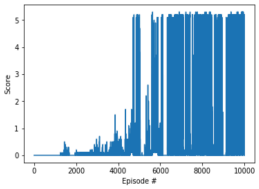
One of the feedbacks I got from the Project 2 was to make the readme.md file specific to this project. That was an excellent feedback and I immediately restructured the readme for both project 1 and project 2 to reflect the reviewer's suggestion. I carried that to this project as well and created a customized README.md.

3. Results

The environment always terminated after 1000 steps. The environment was declared solved if an agent can get a score of > 0.5 (averaged over 100 consecutive episodes). My implementation was able to solve it by 4724 episodes (using the -100 method) and was able to reach a max average score of 5.3.

A formal capture of the results as well as run stats like percentile, variance, absolute max score and a score plot is below:

```
Episode 500     Average Score: 0.00     Score Stats: [0. 0. 0. 0.]      Steps Stats : [13. 13. 13. 14.]
Episode 1000    Average Score: 0.00     Score Stats: [0. 0. 0. 0.]      Steps Stats : [13. 13. 13. 14.]
Episode 1500    Average Score: 0.06     Score Stats: [0.         0.         0.1        0.50000001]      Steps Stats : [ 13.  16.  33. 112.]
Episode 2000    Average Score: 0.00     Score Stats: [0.  0.  0.  0.1]  Steps Stats : [13. 13. 13. 31.]
Episode 2500    Average Score: 0.03     Score Stats: [0.  0.  0.1 0.1]  Steps Stats : [16. 16. 32. 33.]
Episode 3000    Average Score: 0.06     Score Stats: [0.         0.         0.1        0.60000001]      Steps Stats : [ 15.  17.  32. 128.]
Episode 3500    Average Score: 0.06     Score Stats: [0.         0.         0.1        0.50000001]      Steps Stats : [ 13.   14.5  32.  111. ]
Episode 4000    Average Score: 0.15     Score Stats: [0.         0.1        0.2        0.90000001]      Steps Stats : [ 15.75  32.    51.25 186.  ]
Episode 4500    Average Score: 0.17     Score Stats: [0.1        0.1        0.2        0.60000001]      Steps Stats : [ 31.   45.   61.5 141. ]
Episode 4724    Average Score: 0.51     Score: 5.100    Max_steps : 1000        Steps Avg : 113.180
Environment solved in 4724 episodes!    Average Score:  0.51
Episode 5000    Average Score: 1.17     Score Stats: [0.         0.2        1.72500003 5.20000008]      Steps Stats : [  16.   51.  338. 1000.]
Episode 5500    Average Score: 0.30     Score Stats: [0.         0.2        0.40000001 2.60000004]      Steps Stats : [ 14.   53.   99.5 561. ]
Episode 6000    Average Score: 0.21     Score Stats: [0.         0.1        0.3        1.20000002]      Steps Stats : [ 13.75  45.5   82.5  303.  ]
Episode 6500    Average Score: 2.99     Score Stats: [0.3        4.00000006 5.10000008 5.20000008]      Steps Stats : [  75.   790.5 1000.  1000. ]
Episode 7000    Average Score: 2.76     Score Stats: [0.1        3.10000005 5.20000008 5.30000008]      Steps Stats : [  33.   596.5 1000.  1000. ]
Episode 7500    Average Score: 0.70     Score Stats: [0.         0.1        0.1        5.30000008]      Steps Stats : [  16.    31.    33.75 1000.  ]
Episode 8000    Average Score: 1.82     Score Stats: [0.1        0.35000001 5.10000008 5.20000008]      Steps Stats : [  32.   80. 1000. 1000.]
Episode 8500    Average Score: 1.95     Score Stats: [0.1        1.00000001 4.60000007 5.20000008]      Steps Stats : [  32.  210. 1000. 1000.]
Episode 9000    Average Score: 0.85     Score Stats: [0.         0.         0.25       5.20000008]      Steps Stats : [  13.   13.   64. 1000.]
Episode 9500    Average Score: 2.31     Score Stats: [0.2        1.15000002 5.10000008 5.20000008]      Steps Stats : [  52.   237.5 1000.  1000. ]
Episode 10000   Average Score: 2.72     Score Stats: [0.3        2.05000003 5.20000008 5.30000008]      Steps Stats : [  70.  406. 1000. 1000.]
Elapsed : 6:00:50.424233
2018-10-21 03:39:40.714783
```




```
Actor(
  (model): Sequential(
    (0): Linear(in_features=24, out_features=32, bias=True)
    (1): ReLU()
    (2): Linear(in_features=32, out_features=32, bias=True)
    (3): ReLU()
    (4): Linear(in_features=32, out_features=2, bias=True)
    (5): Tanh()
  )
)
Max Score 5.300000 at 5634
Percentile [25,50,75] : [0.  0.1 0.3]
Variance : 2.949
```

Even though I could have stopped after 6000 episodes, I wanted to see if the network is stable and doesn't degrade. It kept the average above 0.5 most of the time after that. The more interesting metrics are the percentiles – the Score Stats and the Step Stats. I have captured the 25th, 50th 75th and 100th percentiles. You see, the number of steps is directly related to the learning – if the agents can keep the ball in play for long time, we will get larger steps. And the scores also will be higher – max of ~5.3 for 1000 steps. Initially the agents couldn't keep the ball in play and so the steps are ~13. After learning for 10,000 episodes, the 75th and 100th percentiles are 1000 steps.

Note : I have captured a log of the experimentations in the notebook in section 2.3 Run Logs and Notes. I also have the screen shots of the CPU usage et al in a separate file (not included in the git hub)

The CPU load was very high – the program used all the CPU threads available!

3.1. Network Architecture alternates and Hyperparameters

One of the more interesting part (in addition to writing the code) is the search for optimum architecture and hyperparameters.

I tried a few network architectures – from a minimalist FC8-FC8-FC2 to FC128-FC128-FC2. The metrics and statistics are captured as a summary in the iPython notebook (Section 2.3) itself for easy reference. The best architecture turned out to be FC32-FC32-FC2. This architecture was used for the final results.

3.2. Hyperparameters

I did an informal search on the important hyperparameters – seeking optimum values within a range. It made sense for this problem as the effect of the hyper parameters were not as varied as in other do mains like object detection, behavior cloning for autonomous driving et al.

1. The number of episodes was initially 2000, but the system went thru very fgast and didn't learn anything! I realized that this problem needs a larger number of episodes and kept it at 10,000. The runs do take 1 hr to 4 hr to finally 6 hrs. In the beginning as the agents are not skilled, the games would end very fast with < 20 steps, but as they get proficient, many games would go to 1000 steps and the clock time got longer
2. Which leads to the max_t parameter. It was 500 which meant the game stopped at 500 steps, easier to debug and do the functionality test. Then I increased the max_t to 1000. Later when running the stored model, I found out that the system terminates at 1000 steps.
3. The Batch Size was initially 64 and I found that it was a good number which gave consistent results.

4. The Replay Buffer size initially was 1e5. I kept the smaller number to increase the chances of getting more newer data. A case can be made to keep older data so that the agent doesn't forget bad results.
5. The Learning Rate initially was 5e-4 and 6e-4 for both Actor and Critic, based on the DDPG paper [2]. I tried the learning rate of [1e-3, 3e-3] as well as [1e-4,3e-4] and they didn't work out at all. Finally, I reverted back to 5e-4 and 6e-4 which worked out very well.
6. One change from last projects in terms of the metrics was to track the percentiles of scores and steps. As the number of steps are related to the skill of the agents, It was interesting to track the stats of the scores and the steps. The scores are directly related to the number of steps ie the more steps the agents can keep the ball in the air, the more scores they got. But the steps is an easy metric to track. This helped in getting an intuition about the learning.

4. Ideas for future work

There are lots of vectors one can follow. Am listing the most important one here. Have a few ideas how to improve the code.

1. A more automated, systemic and formal way of exploring hyperparameters, like the caret package in R, is needed in the future. The current exploration is fine for this problem, but for a larger complex problem, an automated hyperparameter search is required
2. This program runs on CPU – I do want to add the device statements and run the notebook on a GPU machine and compare the performance. The intuition is that GPU might not make that much of a difference, as the neural networks are very small. More CPU core might make the program run faster.
3. Another idea to explore is to run the code in the cloud i.e. AWS/Azure.
4. As I mention in the program, the save and load model could be more generic. The recommendation, from stackoverflow et al, was that this might be unstable. But with the release of PyTorch 1.0 at the PyTorch Developer Conference, this might be streamlined. This is one area to explore further.

4.1. Refactor current notebook code

The code works and I even have a systemic way of just running a saved model without training. And if we train, it will save the best model. This worked out very well, especially to capture the videos.

4.2. Challenge : Play Soccer

The optional soccer challenge is also very interesting. I plan to solve that challenge after this project is accepted.

4.3. Algorithms

There are two potential algorithms to try –

- 1st one is to try out PPO and understand how PPO can be applied in a continuous action environment. There are a few interesting ways to approach with PPO.

- 2nd is to try the full MADDPG as depicted in paper[1]. I think the soccer game has more competitive/cooperative aspects. Plan to try the MADDPG on the soccer challenge.

4.4. NIPS Competition!

Plan to participate in the NIPS 2018 competition AI driving Olympics [https://www.duckietown.org/research/AI-Driving-olympics]. The Fleet management Task [https://www.duckietown.org/research/ai-driving-olympics/challenges] requires interesting Reinforcement Algorithms – plan to use the DQN in Project 1 and the DDPG in this project plus other Reinforcement Learning techniques. The NIPS challenge, for now, is a single agent. Am sure future challenges would add multi-agent interaction for mobility.

5. One More Thing !!

Getting a handle on Reinforcement learning is not easy. One has to code and understand a few DRL algorithms to get confidence. After working on project 1 and project 2, project 3 proved to be easier in terms of confidence.

"Confidence is recursive ! You have to come thru a few times to be certain and confident that you can beat it !"

6. References

[1] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, Igor Mordatch, Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments; https://arxiv.org/abs/1706.0227

[2] T. P. Lillicrap, J. J. Hunt, A. Pritzel et al., "Continuous control with deep reinforcement learning," preprint https://arxiv.org/abs/1509.02971

[3] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," https://arxiv.org/abs/1604.06778

[4] Hado van Hasselt and Marco A. Wiering, Reinforcement Learning in Continuous Action Spaces https://ieeexplore.ieee.org/document/4220844

[5] Chris Gaskett, David Wettergreen, and Alexander Zelinsky, "Q-Learning in Continuous State and Action Spaces" http://users.cecs.anu.edu.au/~rsl/rsl_papers/99ai.kambara.pdf

[6] Gabriel Barth-Maron˚, Matthew W. Hoffman˚, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess, Timothy Lillicrap, Distributed Distributional Deterministic Policy Gradients https://openreview.net/forum?id=SyZipzbCb

[7] UCB CS294-112 Deep Reinforcement Learning http://rail.eecs.berkeley.edu/deeprlcourse/