Report – P2 Continuous Control @Unity ML Agent Toolkit

Krishna Sankar
10/19/18

1. Project & General Notes

This is another very interesting problem – to train a double-jointed arm to move to targeted locations as indicated by a green ball. The ball moves, clockwise, counter-clockwise and even static occasionally! The environment is the reacher environment using the Unity ML-Agents toolkit.

The environment has a four-dimensional continuous action space, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

The environment returns a state space of 33 dimensions and consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm.

The environment terminates after 1000 steps.

A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of the agent is to maintain its position at the target location for as many time steps as possible.

In order to solve the environment, an agent has to get a minimum score of 30 averaged over a window of 100 consecutive episodes.

Note : One can run the iPython notebook & train the agent (slow) or use the iPython notebook to run a saved model (fast) or watch the two videos viz. p2_cc_no_learning_02.mov and p2_cc_after_learning_02.mov (fastest)

2. Algorithm

2.1. Reinforcement Learning Model

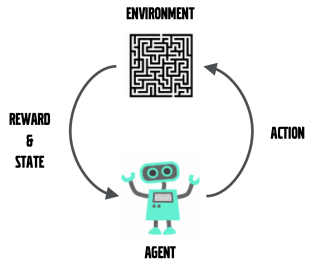


Figure 1

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.

2.2. 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[1] 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[1] 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:

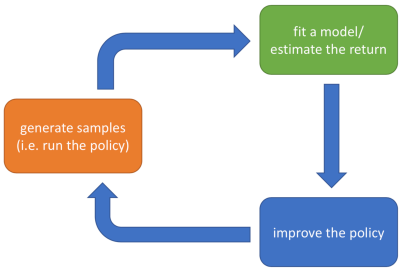
$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

2.3. System Architecture

I liked the simple systems diagram that Prof. Sergey Levine from UC Berkeley uses for his CS294 Deep Reinforcement Learning class[6].

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.

2.4. Implementation - Fundamental Components

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

2.5. 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.

2.6. 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 128 input units and an output layer with 4 units. Both the actor and critic have the same network parameters as shown below.

```
Actor(
  (model): Sequential(
    (0): Linear(in_features=33, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=128, bias=True)
    (3): ReLU()
    (4): Linear(in_features=128, out_features=4, bias=True)
    (5): Tanh()
  )
)
Critic(
  (hc_1): Sequential(
    (0): Linear(in_features=33, out_features=128, bias=True)
    (1): ReLU()
    (2): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (hc_2): Sequential(
    (0): Linear(in_features=132, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=1, bias=True)
  )
)
```

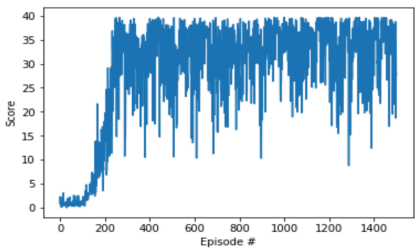
3. Results

The environment always terminated after 1000 steps. I tried max_t > 1500. I didn't see any complex adaptive temporal behavior.
The environment was declared solved if an agent can get a score of > 30 (averaged over 100 consecutive episodes). My implementation was able to solve it by 215 episodes (using the -100 method) or in 315 absolute episodes and was able to reach a max average score of 35.36.

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

```
Episode 100     Average Score: 0.95
Episode 200     Average Score: 6.55
Episode 300     Average Score: 27.73
Episode 315     Average Score: 30.06
Environment solved in 215 episodes!
Episode 400     Average Score: 31.26
Episode 500     Average Score: 32.39
Episode 600     Average Score: 31.73
Episode 700     Average Score: 32.58
Episode 800     Average Score: 30.36
Episode 900     Average Score: 32.20
Episode 1000    Average Score: 34.17
Episode 1100    Average Score: 32.92
Episode 1200    Average Score: 35.36
Episode 1300    Average Score: 31.67
Episode 1400    Average Score: 33.16
Episode 1500    Average Score: 34.33
Elapsed : 4:25:37.281611
2018-10-19 03:56:31.655755
```

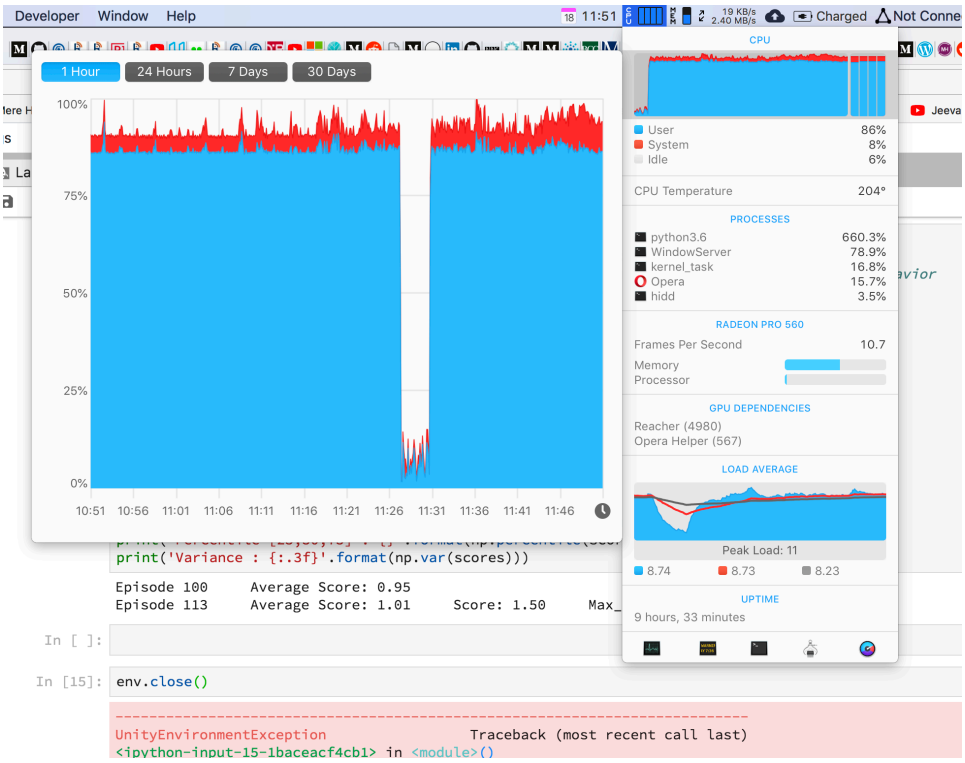
```
Score: 25.23    Max_steps : 1000
Average Score: 30.06
```




```
Actor(
  (model): Sequential(
    (0): Linear(in_features=33, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=128, bias=True)
    (3): ReLU()
    (4): Linear(in_features=128, out_features=4, bias=True)
    (5): Tanh()
  )
)
Max Score 39.689999 at 1000
Percentile [25,50,75] : [24.98499944 32.81499927 36.83249918]
Variance : 131.885
```

Even though I could have stopped after 400 episodes, I wanted to see if the network is stable and doesn't degrade. It kept the average above 30 after that.

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 FC16-FC16-FC4 to the monstrous atrocity (as it relates to this project – there are huge networks with 100s of layers in the Resnet world) FC300-FC400-FC4. The metrics are captured as a summary in the iPython notebook (Section 2.3) itself for easy reference. The best architecture turned out to be FC128-FC128-FC4. 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 It took > 2 hrs to run. So for experimentation, I used 500 to start experimentation. The runs were ~ 45 min max. As I was able to solve the environment within 500 episodes, I tried the final run with 1500 to make sure the network didn't revert back. It didn't.
2. The Batch Size was initially 256, but I found 64 was a good number which gave consistent results.
3. The Replay Buffer size initially was 1e6. I tried a smaller number 1e5 and it didn't make any difference. So, 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.

4. The Learning Rate initially was 1e-3 for both Actor and Critic. Based on the DDPG paper[1] I tried the learning rate of 1e-4 and 3e-4 and they worked out much better. Finally, I chose a middle ground with 5e-4 and 6e-4 which worked out very well.
5. One of the things that made a lot of difference was the addition of the Batchnorm layer to the Critic. The DDOPG paper mentions the use of batchnorm layer. Before the layer, I used to get rewards of only < 1.0 ! But the batchnorm layer changed that !
6. One of the metrics that I was printing was the max steps. With max_t at 500, the max steps was also 500 and the reward couldn't go beyond ~19. Once I increased the max_t to 1500, was able to run 1000 steps and the reward increased to > 30!

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
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.

4.2. Twenty Agent Version

I used the one agent environment to develop the algorithm. After grading of this project, I plan to work on the 20-agent version as well. I already have a skeleton working.

4.3. Challenge Crawl

I also plan to work with the crawler to teach the four-legged creature to walk. As it is optional, I wanted to submit this first.

4.4. Algorithms

I also want to try the reacher with PPO and understand how PPO can be applied in a continuous action environment. There are a few interesting ways to approach with PPO. Another potentially interesting algorithm is the Distributed Distributional Deterministic Policy Gradients[5]. Plan to try that as well.

4.5. 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.

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. This proved to be true in project 1 and more so in project 2.

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

6. References

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

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

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

[4] 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

[5] 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

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