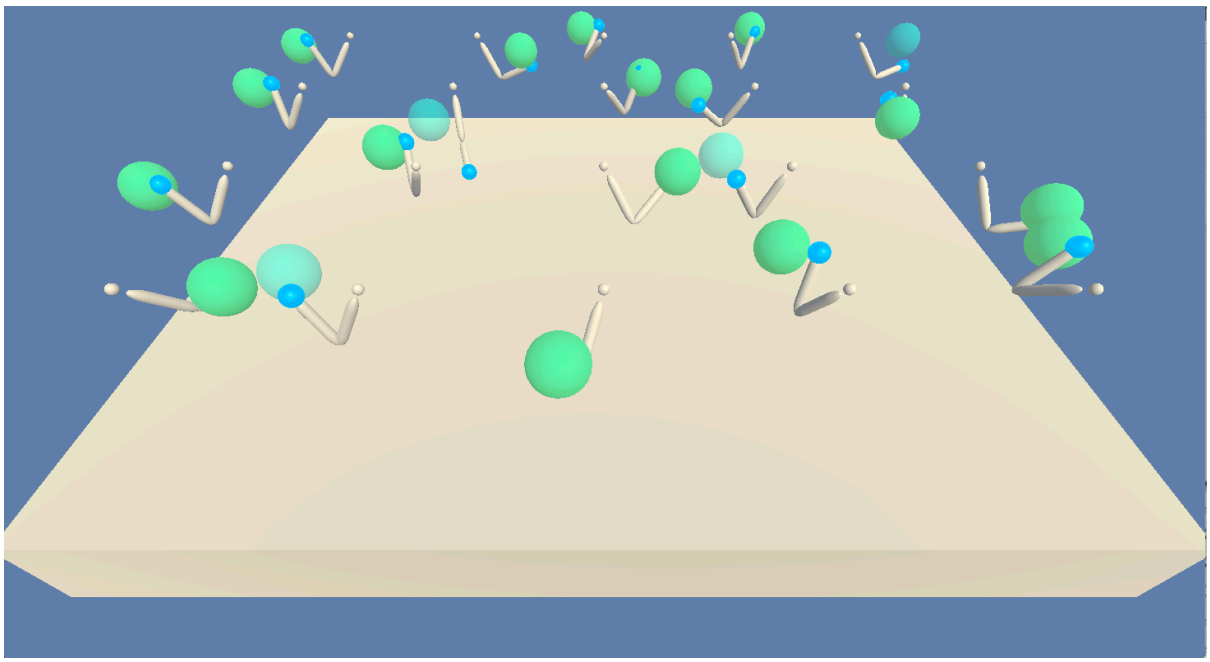
Project Report: Continuous Control

Udacity Deep Reinforcement Learning Nanodegree.

19th December 2018. Simon Birrell.

Introduction

This Project aimed to solve the Unity reinforcement learning test environment called Reacher with 20 simulated robotic arms. The task is for the arms to track a moving target with their end effectors. The code uses the PPO (Proximal Policy Optimization) algorithm to arrive at a solution in 141-250 episodes.



The Code

The Python 3.6 code is packaged within a Jupyter notebook. Full instructions are in the repository at https://github.com/SimonBirrell/ppo-continuous-control-project/

The Agent

All 20 robotic arms are controlled by a single agent using the same policy. The policy is derived using the PPO algorithm (Schulman, Wolski et al, 2017). By using 20 agents, a large set of sample trajectories can be gathered rapidly.

The agent's architecture is shown in Figure 1. A standard Actor-Critic architecture converts the 33 dimensional state into a predicted best action a and estimated value V. The layers are

fully connected with a tanh activation function.

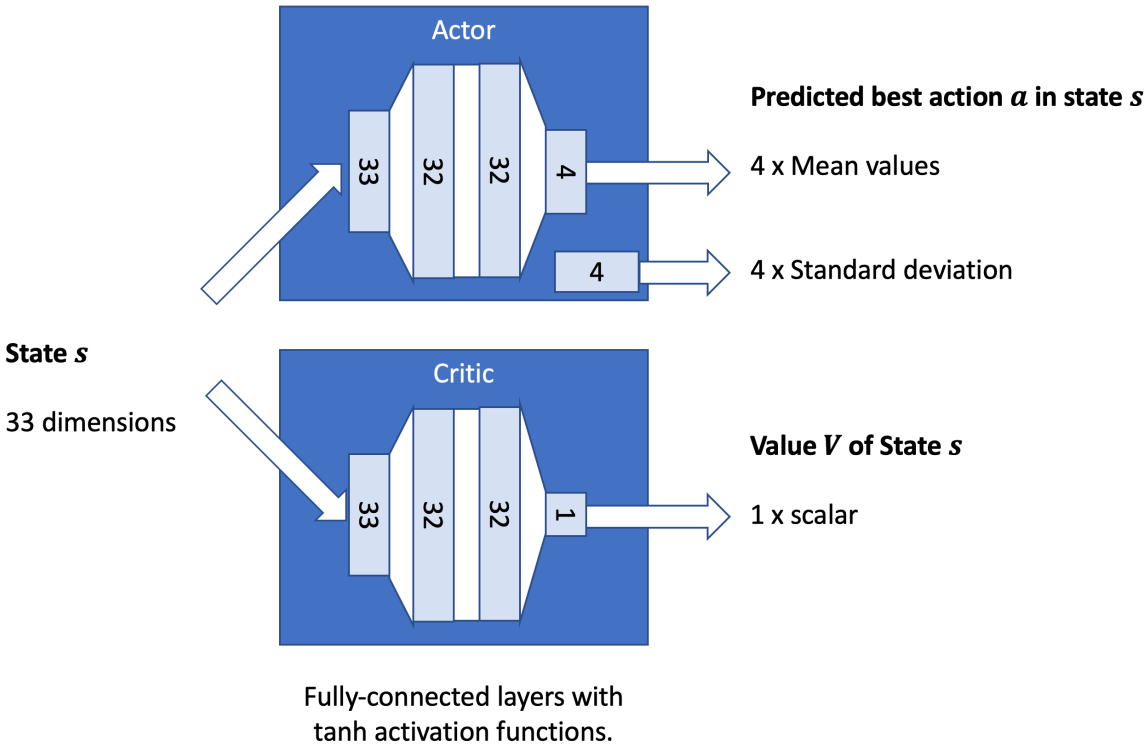


Figure 1: Agent architecture

The Actor's output represents the mean and standard deviation of the predicted distribution of best actions to be taken in the current state. The action space is 4 dimensional, so 8 outputs are required. The actual best action is then sampled from this distribution.

The Critic's output is a single number, representing the estimated value V of the input state. This is converted into the Advantage A, using the formula:

$$A(s,a) = r + \gamma V(s';\ \theta_V) - V(s;\ \theta_V)$$

The loss function for the overall network has two parts: the policy loss (for the Actor) and the value loss (for the critic).

The policy loss is clipped on the upside, according to the PPO algorithm:

```
# This is the core of PPO
# ratio = new prob / old prob for all workers
ratio = (log_prob_action - sampled_log_probs_old).exp()
# Clip loss on the upside
clamped_ratio = ratio.clamp(1.0 - PPO_RATIO_CLIP, 1.0 + PPO_RATIO_CLIP)
obj = ratio * sampled_advantages
obj_clipped = clamped_ratio * sampled_advantages
policy_loss = -torch.min(obj, obj_clipped).mean()
return policy_loss
```

This clipping keeps the surrogate function close to the underlying reward function, avoiding high dimensional cliffs caused by inaccuracies in the reweighted advantages.

$$L_{sur}^{clip}(\theta', \theta) = \sum_t min\left\{\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} R_t^{future},\ clip_\epsilon\left(\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right) R_t^{future}\right\}$$

The return R_t^{future} itself is estimated using Generalized Advantage Estimation, in this case with hyperparameter λ of 0.95.

The value loss is simply the mean squared error:

```
value_loss = 0.5 * (sampled_returns - prediction['v']).pow(2).mean()
```

Training the Agent

Rollout size is defined independently from the environment-signalled episode length. This allows us to optimize the rollout size as one of the hyperparameters (see below).

During training, mini-batches of sampled states, actions, returns, advantages and log-probabilities are sampled. The advantages are reweighted by the ratio mentioned above, to account for the different likelihood of the state-actions under previous and current policies:

$$ratio = \frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}$$

This allows us to use all the sampled state-action pairs, even if they were generated under a previous policy. The clipping mitigates errors caused by inaccuracies in this approximation.

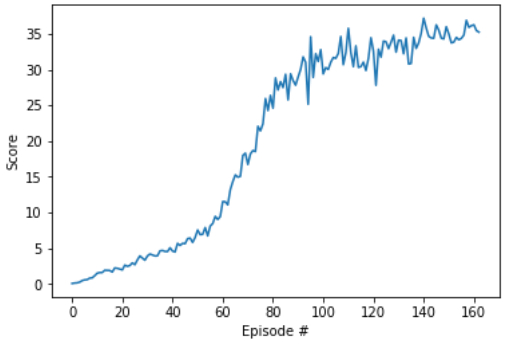
This ratio is implemented with the following line, by subtracting log probabilities (equivalent to division of the probabilities):

```
# ratio = new prob / old prob for all workers
ratio = (log_prob_action - sampled_log_probs_old).exp()
```

A sample training run can be seen below:

```
Attempting to reach 100 episode trailing average of 30.00 in under 300 episodes.
Rollout length: 250
GRADIENT_CLIP 0.75
PPO_RATIO_CLIP 0.1
GAE_LAMBDA 0.95

Finished 1 episodes (1.0 cycles). mean_score_over_agents 0.11949999732896685 trailing 0.11949999732896685
Finished 2 episodes (2.0 cycles). mean_score_over_agents 0.22099999506026508 trailing 0.17024999619461595
Finished 3 episodes (3.0 cycles). mean_score_over_agents 0.305999993160367 trailing 0.21549999518319965
Finished 4 episodes (4.0 cycles). mean_score_over_agents 0.580999987013638 trailing 0.30687499314080924
Finished 5 episodes (5.0 cycles). mean_score_over_agents 0.7829999824985862 trailing 0.40209999101236465
Finished 6 episodes (6.0 cycles). mean_score_over_agents 0.6689999850466848 trailing 0.44658332335141804
Finished 7 episodes (7.0 cycles). mean_score_over_agents 1.0259999770671129 trailing 0.5293571310250887
Finished 8 episodes (8.0 cycles). mean_score_over_agents 1.1459999743849039 trailing 0.6064374864450656
Finished 9 episodes (9.0 cycles). mean_score_over_agents 1.1054999752901495 trailing 0.6618888740945194
Finished 10 episodes (10.0 cycles). mean_score_over_agents 1.475999967008829 trailing 0.7432999833859504
-- Episodes deleted for brevity
```

```
Finished 153 episodes (153.0 cycles). mean_score_over_agents 33.84199924357235 trailing 27.48305438570585

Finished 154 episodes (154.0 cycles). mean_score_over_agents 34.493999228999016 trailing 27.74919937975705

Finished 155 episodes (155.0 cycles). mean_score_over_agents 34.16549923634157 trailing 28.023759373620155

Finished 156 episodes (156.0 cycles). mean_score_over_agents 34.36149923196062 trailing 28.286529367746787


Finished 157 episodes (157.0 cycles). mean_score_over_agents 34.82899922151118 trailing 28.550444361847827

Finished 158 episodes (158.0 cycles). mean_score_over_agents 36.89049917543307 trailing 28.824889355713502

Finished 159 episodes (159.0 cycles). mean_score_over_agents 35.89899919759482 trailing 29.093764349703683

Finished 160 episodes (160.0 cycles). mean_score_over_agents 36.136999192275105 trailing 29.360954343731517

Finished 161 episodes (161.0 cycles). mean_score_over_agents 36.25849918955937 trailing 29.60835933820158

Finished 162 episodes (162.0 cycles). mean_score_over_agents 35.44749920768663 trailing 29.84792433284689

Finished 163 episodes (163.0 cycles). mean_score_over_agents 35.23649921240285 trailing 30.089759327441456

Breaking

Reached target! mean_last_100 30.089759327441456
```



Optimization and Hyperparameters

The method chosen for optimizing the hyperparameters was to systematically vary them one at a time, choosing the best combination before moving on to the next parameter.

Unfortunately, after some searching I discovered that the number of episodes to win varied greatly from trial to trial, even when keeping hyperparameters constant. I repeated the exercise, taking three trials for each parameter combination, and taking the mean and standard deviation of the number of episodes to win. I set a limit of 300 episodes for each trial and in some cases the algorithm failed to win within this constraint.

Hyperparameters					Episodes to Win			Results		
Rollout length	Layers	GRAD_CLIP	PPO_CLIP	GAE_TAU	Trial 1	Trial 2	Trial 3	Mean	STD	Robustness
125	64	0.75	0.1	0.95	X	X	X			0%
250	64	0.75	0.1	0.95	X	180	235	207.5	38.9	67%
375	64	0.75	0.1	0.95	151	140	X	145.5	7.8	67%
500	64	0.75	0.1	0.95	237	175	263	225.0	45.2	100%
500	32	0.75	0.1	0.95	184	240	151	191.7	45.0	100%
250	32	0.75	0.1	0.95	250	141	163	184.7	57.6	100%

The hyperparameter that had the biggest effect on the result was the length of the rollout. Low values like 125 and 250 didn't always complete within 300 episodes, and a "Robustness" metric is shown (percentage of successful trials). A rollout length of 500 consistently completed the task.

Next, I tried varying the number of hidden layers in the actor and critic sub-networks. I had originally selected 64, but it turned out that with only 32 units the task was on average completed faster.

I then again tried reducing the rollout length from 500 to 250 and obtained slightly faster training while maintaining robustness. The final chosen combination had hidden layers of 32 units with a rollout of 250. It is likely that better still results could be obtained with more trials.

Ideas for Future Work

The search for good hyperparameters was tedious. With more time for the project I would put the code into a pure Python format and run multiple sessions in parallel. That would speed up the testing of each hyperparameter combination.

Beyond this, I would attempt to systematically search the hyperparameter space, either with a grid search or a Bayesian one.

It may also be interesting to search for hyperparameter combinations that not only reduce the number of episodes to successful completion, but also reduce the variance in that number. In other words, a robust and repeatable algorithm may be more desirable than one that is on average faster, assuming it exists.