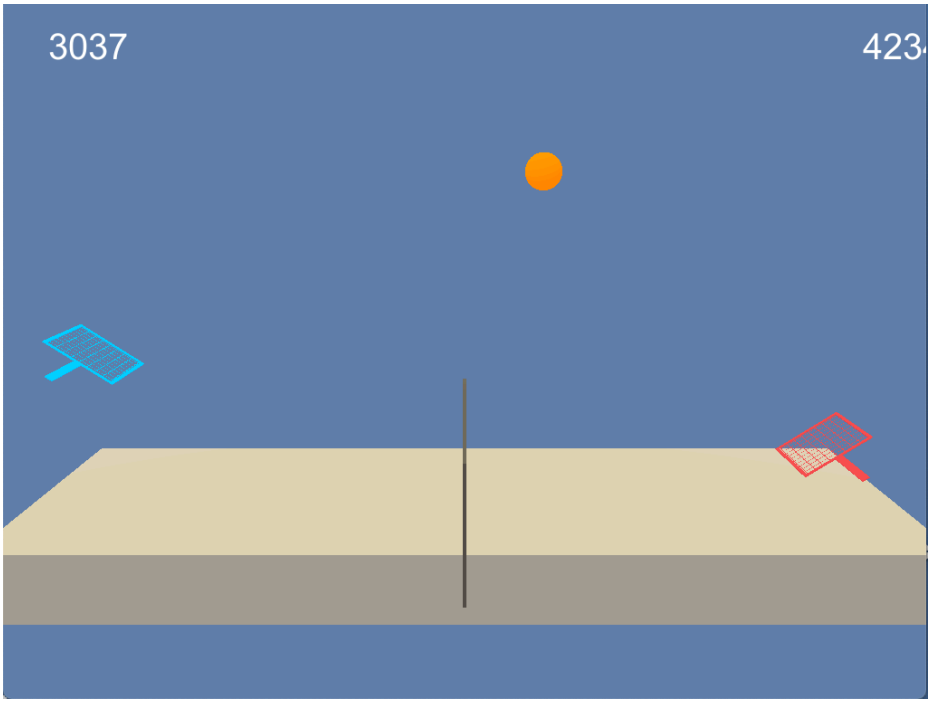
Project Report: Collaboration and Competition

Udacity Deep Reinforcement Learning Nanodegree.

7th January 2019. Simon Birrell.

Introduction

This Project aimed to solve the Unity reinforcement learning test environment called Tennis with two competing agents.



The reward function is as follows. For each agent, hitting the ball over the net gets a reward of +0.1. If the ball hits the ground or is outside the court, the reward is -0.01.

Two continuous action variables control movement towards the net (the game is 2D not 3D) and a jumping motion.

The observations (local to each agent) are 8 continuous variables representing ball and racket state (from the agent's point of view).

After each episode, the maximum score from the two agents is taken. This is then averaged over the last 100 episodes. When this trailing average passes +0.5, the task is considered solved.

The code uses the PPO (Proximal Policy Optimization) algorithm to arrive at a solution in 1500-1800 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-collaboration-and-competition.

The bulk of the code is identical to the PPO solution of the previous environment: Reach20. I therefore focused in this project on automating hyperparameter search.

The Agent

Both players 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 both agents, a larger set of sample trajectories can be gathered rapidly.

The agent's architecture is shown in Figure 1. A standard Actor-Critic architecture converts the 8 dimensional state into a predicted best action a and estimated value V. The layers are fully connected with a tanh activation function. The actual number of hidden layers is a hyperparameter, so the diagram illustrates an example network.

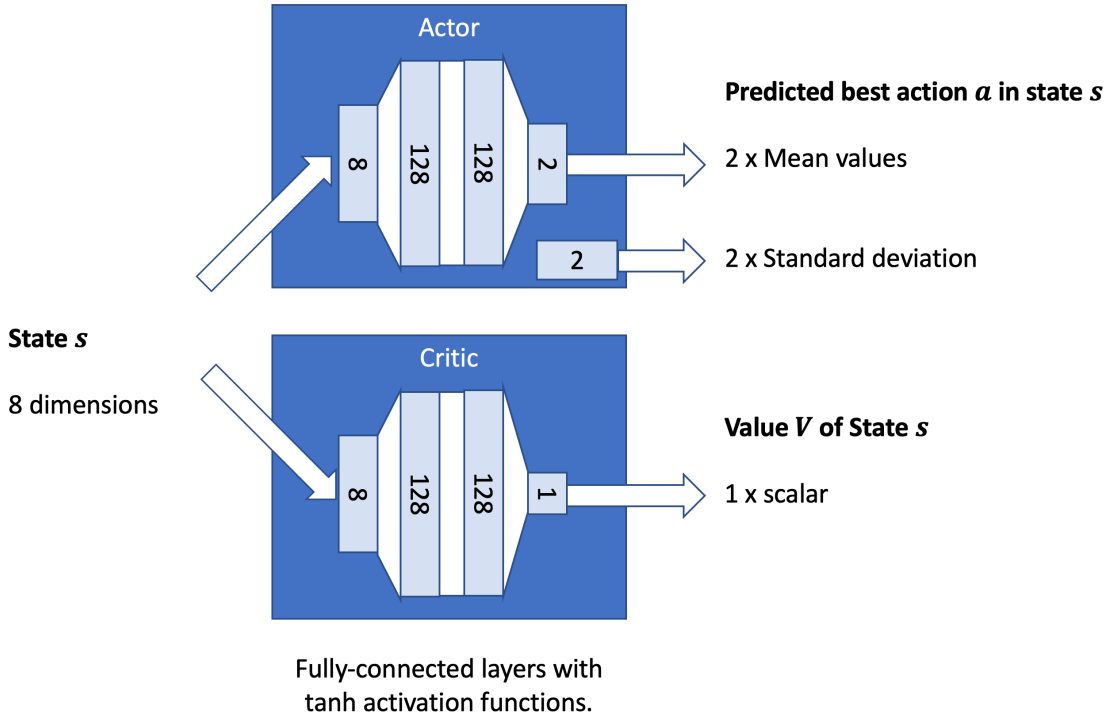


Figure 1: Agent architecture (actual number hidden units is a hyperparameter)

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 2 dimensional, so 4 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:

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

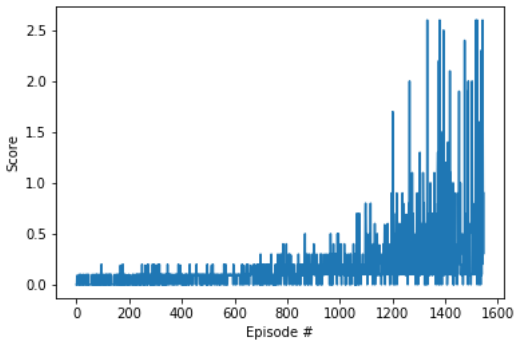
This allows us to use all the sampled state-action pairs, even if they were generated under a previous policy. The clipping mitigates errors caused by inaccuracies in this approximation.

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 0.50 in under 10000 episodes.
Rollout length: 250
GRADIENT_CLIP 0.75
PPO_RATIO_CLIP 0.1
GAE_LAMBDA 0.95
Finished 1 episodes (1.0 cycles). best score over agents    0.000 trailing    0.000
Finished 2 episodes (2.0 cycles). best score over agents    0.000 trailing    0.000
Finished 3 episodes (3.0 cycles). best score over agents    0.090 trailing    0.030
Finished 4 episodes (4.0 cycles). best score over agents    0.000 trailing    0.023
Finished 5 episodes (5.0 cycles). best score over agents    0.000 trailing    0.018
Finished 6 episodes (6.0 cycles). best score over agents    0.000 trailing    0.015
Finished 7 episodes (7.0 cycles). best score over agents    0.000 trailing    0.013
Finished 8 episodes (8.0 cycles). best score over agents    0.000 trailing    0.011
Finished 9 episodes (9.0 cycles). best score over agents    0.000 trailing    0.010
Finished 10 episodes (10.0 cycles). best score over agents    0.000 trailing    0.009
----- Episodes omitted for brevity --------
Finished 1537 episodes (1537.0 cycles). best score over agents    2.300 trailing    0.446
Finished 1538 episodes (1538.0 cycles). best score over agents    0.090 trailing    0.446
Finished 1539 episodes (1539.0 cycles). best score over agents    0.300 trailing    0.444
Finished 1540 episodes (1540.0 cycles). best score over agents    0.100 trailing    0.436
Finished 1541 episodes (1541.0 cycles). best score over agents    0.800 trailing    0.443
Finished 1542 episodes (1542.0 cycles). best score over agents    0.200 trailing    0.444
Finished 1543 episodes (1543.0 cycles). best score over agents    2.600 trailing    0.468
Finished 1544 episodes (1544.0 cycles). best score over agents    1.500 trailing    0.482
Finished 1545 episodes (1545.0 cycles). best score over agents    0.900 trailing    0.490
Finished 1546 episodes (1546.0 cycles). best score over agents    0.300 trailing    0.492
Finished 1547 episodes (1547.0 cycles). best score over agents    0.900 trailing    0.501
Breaking
Reached target! mean_last_100 0.501400007493794
======== Saving weights ==========
```



Optimization and Hyperparameters

Given that the code for my previous project worked with minimal alterations, I focused on improving the hyperparameter search.

First, I performed a manual search using the same technique as in the previous project. The highest impact hyperparameters appeared to be rollout length and the number of hidden layers, so I tried varying these independently:

Hyperparameters		Results			Analysis		
Rollout	Hidden Layers	Trial 1	Trial 2	Trial 3	Mean	STD	Results
10	32						No go at 10,000
250	32	1860	1917	1615	1797	160	
250	16	2570	2644	2077	2430	308	
250	**128**	**1547**	**1877**	**1547**	**1657**	**191**	**The best configuration**
250	256	1705	1670	1731	1702	31	
250	80	1462	2382	2178	2007	483	

Next, I worked on implementing the Hyperband method recommended by a Udacity reviewer of my previous project:

https://people.eecs.berkeley.edu/~kjamieson/hyperband.html

This method performs a grid-like search of the chosen hyperparameters, with the novelty that it initially starts with a large number of combinations but performs only a limited number of iterations on each one. It is a broad search, not a deep one. The assumption is that the configurations that do better than average on short runs are likely to contain the configurations that will eventually win in the long term. On the next round, the below-average configurations are discarded and the number of iterations increased. In this way, the search become progressively less broad and more deep.

The implemented code can be seen in the notebook in the "Hyperparameter Search" section.

The "value loss" function I defined as

$$L_{val} = \begin{cases} -R_{mean} & if\ target\ not\ reached \\ -\left(R_{mean} + \rho(T_{max} - T_{win})\right) & if\ target\ reached \end{cases}$$

where R_{mean} is the 100-episode trailing average at the end of the training session, T_{win} is the number of episodes to win (if the training session resulted in a win), T_{max} is the maximum number of episodes attempted in any training session (3000 in this case) and ρ is a scaling factor (0.005 here).

The resulting value loss is a small negative number (0.0 to -0.5) if the training session didn't win and a larger negative number (-0.5 to -15.5) if it did. As the search algorithm tries to minimize this, it will pick configurations that improve faster when trialling a small number of iterations, and those that win in fewer episodes for a larger number.

After some trials with both one (ROLLOUT_LENGTH) and two (ROLLOUT_LENGTH, HIDDEN_LAYERS) hyperparameters, I concluded that the stochasticity of the training results might be affecting the pruning of below-average configurations. With two hyperparameters it takes around 9 hours to do a full search, performing one training session per configuration in Hyperband's inner loop.

I then changed the code to perform three training sessions for each configuration in the inner loop and take the mean of the three values losses. This of course tripled the time to search the hyperparameter space, with the aim of finding better configurations.

Here is an extract of the hyperparameter search log:

```
New T:
[{'ROLLOUT_LENGTH': 502, 'HIDDEN_LAYERS': 305}, {'ROLLOUT_LENGTH': 535, 'HIDDEN_LAYERS': 216}, {'ROLLOUT_LENGTH': 3
44, 'HIDDEN_LAYERS': 377}, {'ROLLOUT_LENGTH': 449, 'HIDDEN_LAYERS': 144}]
--------------------
```

```
Inner loop i = 1
--------run_then_return_val_loss for 3000 iterations at {'ROLLOUT_LENGTH': 502, 'HIDDEN_LAYERS': 305}-----------
ASSIGNED ROLLOUT_LENGTH 502
Running training session 0 with max 3000 iterations
Attempting to reach 100 episode trailing average of 0.50 in under 3000 episodes.
ROLLOUT_LENGTH: 502
GRADIENT_CLIP 0.75
PPO_RATIO_CLIP 0.1
GAE_LAMBDA 0.95
HIDDEN LAYERS 305
Finished 500 episodes (500.0 cycles). best score over agents    0.000 trailing    0.038
Finished 1000 episodes (1000.0 cycles). best score over agents    0.090 trailing    0.078
Finished 1500 episodes (1500.0 cycles). best score over agents    0.100 trailing    0.114
Finished 2000 episodes (2000.0 cycles). best score over agents    0.100 trailing    0.160
Finished 2500 episodes (2500.0 cycles). best score over agents    0.000 trailing    0.447
Breaking
Breaking
Breaking
Reached target! mean_last_100 0.5346000079810619
======== Saving weights ==========
Value loss    -2.91
Running training session 1 with max 3000 iterations
Attempting to reach 100 episode trailing average of 0.50 in under 3000 episodes.
ROLLOUT_LENGTH: 502
GRADIENT_CLIP 0.75
PPO_RATIO_CLIP 0.1
GAE_LAMBDA 0.95
HIDDEN LAYERS 305
Finished 500 episodes (500.0 cycles). best score over agents    0.090 trailing    0.052
Finished 1000 episodes (1000.0 cycles). best score over agents    0.100 trailing    0.078
Finished 1500 episodes (1500.0 cycles). best score over agents    0.090 trailing    0.113
Finished 2000 episodes (2000.0 cycles). best score over agents    0.190 trailing    0.175
Breaking
Reached target! mean_last_100 0.504900007583201
======== Saving weights ==========
Value loss    -3.41
Running training session 2 with max 3000 iterations
Attempting to reach 100 episode trailing average of 0.50 in under 3000 episodes.
ROLLOUT_LENGTH: 502
GRADIENT_CLIP 0.75
PPO_RATIO_CLIP 0.1
GAE_LAMBDA 0.95
HIDDEN LAYERS 305
Finished 500 episodes (500.0 cycles). best score over agents    0.000 trailing    0.036
Finished 1000 episodes (1000.0 cycles). best score over agents    0.100 trailing    0.077
Finished 1500 episodes (1500.0 cycles). best score over agents    0.100 trailing    0.136
Finished 2000 episodes (2000.0 cycles). best score over agents    0.400 trailing    0.296
```

```
Breaking

Reached target! mean_last_100 0.5005000075139105

======== Saving weights ==========

Value loss    -4.86

Mean value loss   -3.728

--------run_then_return_val_loss for 3000 iterations at {'ROLLOUT_LENGTH': 535, 'HIDDEN_LAYERS': 216}----------
```

This is towards the end of the search, with four hyperparameter combinations remaining to be trained for the full 3000 maximum episodes (see first line). The combination "ROLLOUT_LENGTH=502, HIDDEN_LAYERS=305) is then trialled three times and the mean of the three value losses is taken.

As it turned out, the three trial version gave an identical result to the one trial version despite taking three times as long. It's possible that starting from a different random seed would help.

Hyperparameters		Results			Analysis		
Rollout	Hidden Layers	Trial 1	Trial 2	Trial 3	Mean	STD	Results
295	80	1574	1534	2067	1725	297	Optimizing Rollout only
317	390	1824	1984	2145	1984	161	Optimizing both
317	390	1881	1922	2212	2005	180	Taking mean three trials for each configuration

The results of the automated hyperparameter search are slightly inferior to my own manual search, although they do of course require less attention to perform. This could be because there is a broad minima for PPO and I had already found it.

My manual search method took fewer trials than the automated one, but would really only work where the searched hyperparameters are independent in the effects on performance. This does appear to be the case for ROLLOUT_LENGTH and HIDDEN_LAYERS, but would probably not be the case for, say ROLLOUT_LENGTH and PPO_RATIO_CLIP.

Ideas for Future Work

The hyperparameter search merits further development. It would be interesting to implement my own manual search method as code and compare it to Hyperband on a variety of hyperparameter selections. It would result in an ensemble of hyperparameter search methods!

It would also be interesting to try different configurations of PPO_RATIO_CLIP, GRADIENT_CLIP, GAE_LAMBDA and so on. As the number of hyperparameters increases, so does the necessary search time.

The training itself could probably be parallelized. Assuming that it's possible to load three Unity environments simultaneously, I would transfer the notebook code to a regular Python application and perform three training sessions simultaneously for each inner loop configuration. This should easily be achievable on my Macbook Pro, and would reduce hyperparameter search time from some 27 hours back to 9.