Reinforcement Learning Project 2019/2020

Giovanbattista Abbate • 1875271

13th January 2020

1 Introduction

In this project, experiments on the MuJoCo framework[1] together with the gym[2] environment *Ant-v2*[3] will be made, using TensorFlow 1.12[4] and the Soft-Actor Critic algorithm (SAC)[5], provided by `stable-baselines`[6].

1.1 MuJoCo

MuJoCo stands for Multi-Joint dynamics with Contact. It is a physics engine for detailed, efficient rigid body simulations with contact forces. It is a dynamic library with C/C++ API, with an excellent Python porting[7].

1.2 Gym: Ant-v2

Gym is a toolkit for developing and comparing reinforcement learning algorithms. The gym library is a collection of test problems (environments) that can be used to work on reinforcement learning algorithms. One of these environments is the *Ant*, a four-legged creature, which target is to walk as far as possible, as fast as possible.

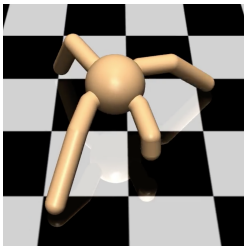


Figure 1: Gym's Ant

By searching into the documentation, it is possible to find the file `ant.xml`, from which is possible to identify the main object of the Ant's hierarchical structure, the `torso`, and his four children:

[1] http://www.mujoco.org/
[2] https://gym.openai.com/
[3] https://gym.openai.com/envs/Ant-v2/
[4] Since baselines only works with TF 1.x
[5] https://arxiv.org/abs/1812.05905
[6] https://github.com/hill-a/stable-baselines
[7] https://github.com/openai/mujoco-py

- `front_left_leg`
- `front_right_leg`
- `back_leg`
- `right_back_leg`

From this structure is possible to define a 111-dim observation space[8]:

Torso Height	1
Torso Orientation	4
Joint Angles	8
Velocities (Angular + Linear)	6
Joint Velocities	8
External Forces	84
Total dimension	111

Table 1: Observation space dimensionality

And a 8-dim action space, made by the torque[9] of each joint.

1.2.1 Reward Function

The reward function is defined in the file `ant.py`, specifically inside the function `step` of the class `AntEnv`, and is defined as:

```
[...]
def step(self, a):
  xposbefore = self.get_body_com("torso")[0]
  self.do_simulation(a, self.frame_skip)
  xposafter = self.get_body_com("torso")[0]
  forward_reward = (xposafter - xposbefore)/self.dt
  ctrl_cost = .5 * np.square(a).sum()
  contact_cost = 0.5 * 1e-3 * np.sum(
    np.square(np.clip(self.sim.data.cfrc_ext, -1, 1)))
  survive_reward = 1.0
  reward = forward_reward - ctrl_cost - contact_cost +
    + survive_reward
  [...]
```

This means that the reward is computed considering the variation of the position in the time, subtracting the action cost and the cost of the contact forces, and adding the survive reward for each step in which Ant is not flipping or stopping. The environment is considered *solved* when the average reward over 100 episodes reaches 6000.

[8] Regarding external forces: (force x,y,z + torque x,y,z) applied to the CoM of each link. Ant has 14 links: ground+ torso + 12 (3 links for 4 legs): (3+3)*(14)=84

[9] Value from -1 to 1

1.3 Soft-Actor Critic Algorithm

Soft Actor Critic (SAC) is an algorithm which optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and Deep Deterministic Policy Gradient (DPPG)-style approaches. It incorporates the clipped double-Q trick, and due to the inherent stochasticity of the policy in SAC, it also winds up benefiting from something like target policy smoothing. A central feature of SAC is *entropy regularization*. The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. This has a close connection to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on. It can also prevent the policy from prematurely converging to a bad local optimum. The pseudocode is:

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, V-function parameters ψ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\psi_{\text{targ}} \leftarrow \psi$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for Q and V functions:

$$y_q(r, s', d) = r + \gamma(1 - d)V_{\psi_{\text{targ}}}(s')$$
$$y_v(s) = \min_{i=1,2} Q_{\phi_i}(s, \tilde{a}) - \alpha \log \pi_\theta(\tilde{a}|s), \qquad \tilde{a} \sim \pi_\theta(\cdot|s)$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi,i}(s, a) - y_q(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update V-function by one step of gradient descent using

$$\nabla_\psi \frac{1}{|B|} \sum_{s \in B} \left(V_\psi(s) - y_v(s)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(Q_{\phi,1}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target value network with

$$\psi_{\text{targ}} \leftarrow \rho\psi_{\text{targ}} + (1 - \rho)\psi$$

    end for
  end if
until convergence

Figure 2: SAC Algorithm Pseudocode

2 Experiments

The following experiments were made in Python 3.6.9 with Jupyter Notebook, using TensorFlow 1.12, MuJoCo 2.0, stable-baselines 2.9.0, Ubuntu 18.04, on a notebook with a CPU Intel Core i3-5005U CPU @ 2.00GHz and a GPU Nvidia GeForce 920M. Each run took almost 13 hours for the 2M time-steps experiments, and almost 19 hours for the 3M time-steps experiments.

2.1 2M time-steps Experiments

Apart from three tuning experiments, five experiments using 2M time-steps were made. All of these experiments share the following hyperparameters:

Optimizer	Adam
Learning Rate	$3 * 10^{-4}$
Buffer Size	10^6
Batch Size	256
Target update interval	1
Gradient steps	1
Learning starts	10^5
time-steps	$2 * 10^6$
Number of hidden layers	2
Number of hidden units per layer	256

Table 2: Shared Hyperparameters

While, for each configuration, these parameters were taken[10]:

	base	ent-coef-02	tau-0005	gamma-099	combined
Gamma	0.98	0.98	0.98	0.99	0.99
Tau	0.01	0.01	0.005	0.01	0.005
Ent.Coef.	auto	0.2	auto	auto	0.2

Table 3: Specific Hyperparameters

In the following table is it possible to see the results of each configuration[11] on an interval of 100 episodes:

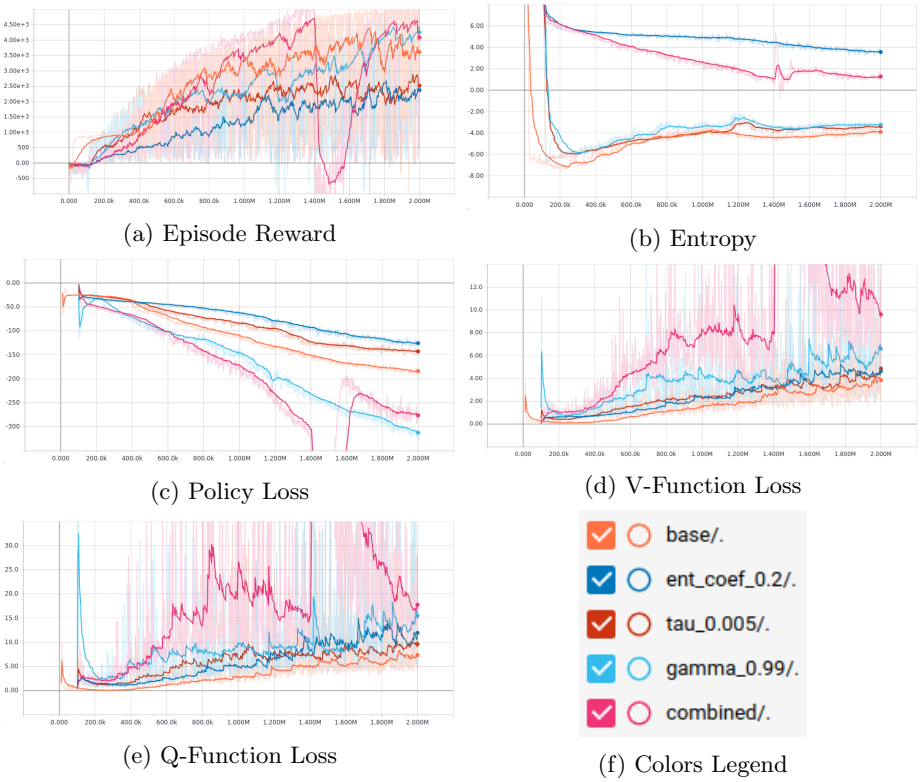
	mean	STD	SEM
base	4318	1590	159
ent-coef-02	3162	1620	162
tau-0005	3070	1415	142
gamma-099	4674	1134	113
combined	**5383**	**655**	**66**

Table 4: 2M time-steps Results

From the table, it is possible to see that the *combined* model is the best among all the 2M time-steps models, but this is not enough to consider the environment solved.

[10] For indentation reason, the names were abbreviated. They are: discount factor (gamma), target smoothing coefficient (tau), entropy regularization coefficient (ent.coef.)

[11] Mean: average of the rewards taken in 100 episodes. STD: standard deviation. SEM: standard error of the mean.



(a) Episode Reward

(b) Entropy

(c) Policy Loss

(d) V-Function Loss

(e) Q-Function Loss

(f) Colors Legend

Figure 3: Tensorboard Graphs for 2M-TS models (zoom in for more details)

From **Figure 3a** it is possible to see that the best models is *combined*. However, looking at the losses **Figure 3c/d/e** it is possible to see that also *gamma-099* could improve over time. Moreover, results in **Table 4** confirm this hypothesis.

2.2 3M time-steps Experiments

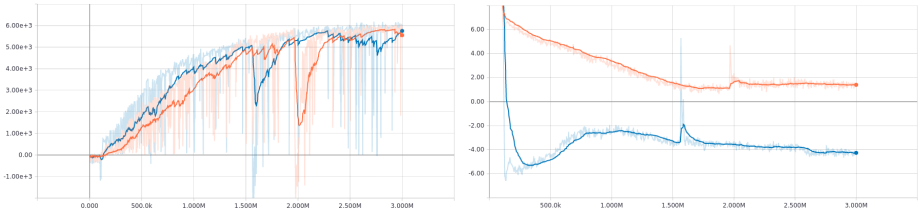
Considering previous statements, new experiments were made on the two best models: *combined* and *gamma-099*. The configurations were the same reported in **Table2/3**, except for the parameter `time-steps` set to $3 * 10^6$. The results are:

	Best Model	End of Training	time-step of BM
combined	$6289 \pm 534(53)$	$6269 \pm 1144(114)$	2828451
gamma-099	**6388±76(8)**	$6048 \pm 948(95)$	**2186203**

Table 5: 3M time-steps Results

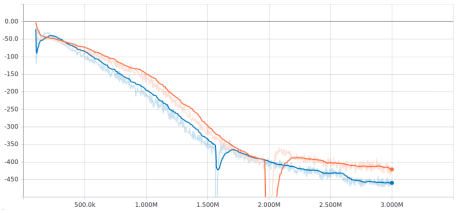
Where the results are reported in the form `mean±STD(SEM)`, and the last column is the time-step of the best[12] model taken during the training.

[12]According to the callback function defined in the Jupyter Notebook file attached to this report.

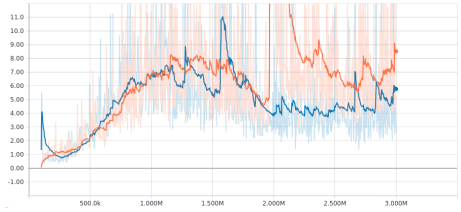


(a) Episode Reward

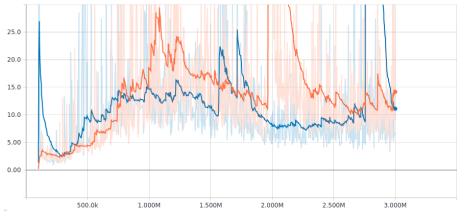
(b) Entropy



(c) Policy Loss



(d) V-Function Loss



(e) Q-Function Loss

Figure 4: Tensorboard Graphs for 3M-TS models (zoom in for more details).

From **Figure 4** and **Table 5** is it possible to see that the hypothesis made in **Section 2.1** were correct: the model *gamma-099* (reported in blue in the graphs, while *combined* is reported in orange) is the best model overall after 2186203 time-steps, contrary to 2M-TS models where the best model was *combined*.

3 Conclusion

At the end of all of these experiments, it is possible to say that SAC algorithm is quite unstable on the results and during the training, since it is not converging not even after 19 hours and 3 millions time-steps of training. According to other experiments[13] it seems that the best algorithm for quadruped-walk is Trust Region Policy Optimization (TRPO) using gradient descent with KL loss terms[14], but this is out of the purpose of this report.

According to results in **Table 5** and premises in **Section 1.2.1**, it is possible to say that the *Ant-v2* problem has been resolved in this work.

[13] https://gist.github.com/pat-coady/bac60888f011199aad72d2f1e6f5a4fa
[14] https://github.com/pat-coady/trpo/blob/master/README.md