Twin Delayed DDPG for Reinforcement Learning

Francesco

Peracchia

1906895

Index

Introduction.....3

1.1 TD3.....3

1.2 MuJoCo engine & HumanoidStandup.....3

TD3 implementation.....4

2.1.1 Twin Critic Networks.....4

2.1.2 Delayed Updates.....4

2.1.3 Noise Regularisation.....4

2.2.1 Pipeline.....5

2.2.2 Implementation.....6

2.3.1 Conclusion.....8

reference.....8

Introduction

1.1 TD3

TD3 is an evolution of DDPG, this algorithm was proposed to avoid specific drawbacks of his predecessor, in particular DDPG fail when it tries to estimate the Q-values, that are overestimated, follow that the error grow up quickly and the process become unuseful. This error is the main cause of agent's failure, while finding a local optima. To reduce this overestimation are generally design to use this three features:

Clipped Double-Q Learning: a pair of critic networks, TD3 learns *two* Q-functions instead of one and use in the Bellman error loss functions the smaller of the two.

Delayed Policy Updates: updates of the actor are delayed , TD3 updates the policy less frequently than the Q-function.

Trick Three: target policy smoothing TD3 adds noise to the target action, this is used to do an Action noise regularisation, design to make less probable Q-function errors.

1.2 MuJoCo engine & HumanoidStandup

Multi-**Jo**int dynamics with **Co**ntact or MuJoCo is a physics engine used for fast and accurate simulation in research, development in robotics and other areas. This simulator is designed for the purpose of model-based optimization, and in particular optimization through contacts.

Is needed a license that can be free used for student purpose to obtain a 30-day free trial, this is simply obtained compiling a format in MuJoCo site, after is needed follow the process below to use MuJoco features in your project.

1. Download the MuJoCo version 2.0 binaries for Linux or OSX.
2. Unzip the downloaded `mujoco200` directory into `~/.mujoco/mujoco200`, and place your license key (the `mjkey.txt` file from your email) at `~/.mujoco/mjkey.txt`.

If you want to specify a nonstandard location for the key and package, use the env variables `MUJOCO_PY_MJKEY_PATH` and `MUJOCO_PY_MUJOCO_PATH`.

from

https://github.com/openai/mujoco-py

HumanoidStandUp is OpenAI environment based on MuJoCo, here a three-dimensional bipedal robot must stand up as fast as possible, a reward is related to these properties. We simply need to install Gym and then import the environment that we want use.

```
pip install gym
import gym
env = gym.make('HumanoidStandUp-v2')
```

TD3 implementation

2.1.1 Twin Critic Networks

In these section is presented how design a double Q-learning, and also an estimation of the current Q values using separate target value function. In this paper [1] is explained why is needed an older implementation of Double Q Learning respect to a new one. In particular the policy and target networks are updated so slowly that they look very similar, which brings bias back into the picture, use a clipped double Q learning where it takes the smallest value of the two critic networks is a better choice. Underestimate bias isn't a problem because, lower values will not be propagated through the algorithm like overestimate values. This grant a more stable approximation, improving the general stability of the algorithm.

2.1.2 Delayed Updates

Agents training use target networks for introducing stability but we need to consider that is caused by the interaction between the policy and critic networks. "The training of the agent diverges when a poor policy is overestimated" ,our agents policy will then continue to get worse as it is updating on states with a lot of error.

The value network become more stable and reduce errors before it is used to update the policy network.

"In practice, the policy network is updated after a fixed period of time steps, while the value network continues to update after each time step. These less frequent policy updates will have value estimate with lower variance and therefore should result in a better policy"

Actor network have a delayed update every 2 time steps instead of after each time step, resulting in more stable and efficient training.

2.1.3 Noise Regularization

Smoothing the target policy is another important goal to achieve, deterministic policy methods in generally produce target values with high variance when updating the critic, this is due to overfitting to spikes in the value estimate. Hence we have to minimize this problem reducing this variance, using a regularization technique known as **target policy smoothing.** TD3 reduces this variance by adding a small amount of random noise to the target and averaging over mini batches. The range of noise is clipped, this is done because we want to keep the target value close to the original action, also adding additional noise to the value estimate, our policies in general tend to be more stable as the target value is returning a higher value for actions that are more robust to noise and interference. Finally to the selected action is added **c**lipped noise when calculating the targets, this generate higher values for actions that are more robust.

2.2.1 Pipeline

This TD3 pseudocode is described in a paper of *Fujimoto et al., 2018 [2], here we can find the main structure of the algorithm and when the features above described are used.*

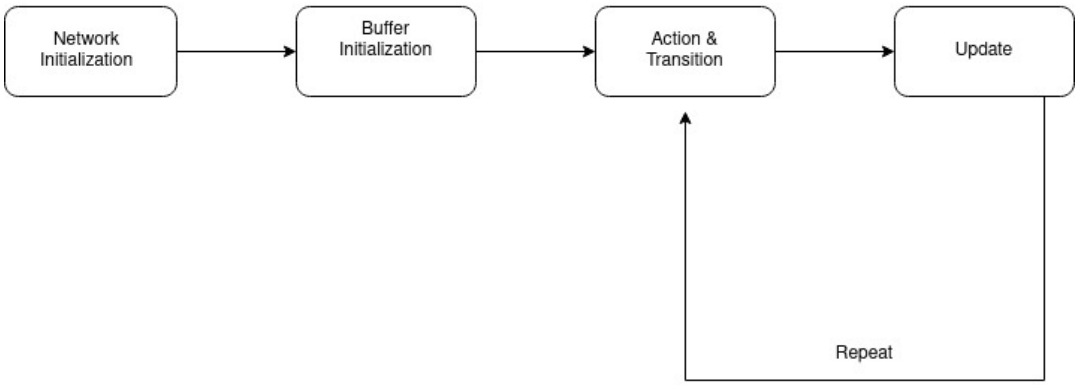
Algorithm 1 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ
with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
  Select action with exploration noise $a \sim \pi(s) + \epsilon$,
  $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
  Store transition tuple (s, a, r, s') in $\mathcal{B}$

  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\tilde{a} \leftarrow \pi_{\phi'}(s) + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$ Target policy smoothing
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$ Clipped Double Q-learning
  Update critics $\theta_i \leftarrow \min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  if t mod d **then** Delayed update of target and policy networks
    Update ϕ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
  end if
end for

In all the paper presented at [1] and the paper [2] the algorithm is generally described with this pipeline

- **Initialize networks & replay buffer**
- **Select and carry out action with exploration noise**
- **Store transitions**
- **Update critic, actor, target networks**



2.2.2 Implementation

Action & Transition

In a Markov decision process over this environment is needed like always choose the action, this depends over the observation Here the agent will pick an action with exploration noise that is added in the code below.

Call in main

```
action = agente.choose_action(observation)
```

this is the choose_action definition, this part of the code is taken from another implementation of TD3 [3]

```
def choose_action(self, observation):
if self.time_step < self.warmup:
mu = T.tensor(np.random.normal(scale=self.noise, size=(self.n_actions,)))
else:
state = T.tensor(observation, dtype=T.float).to(self.actor.device)
mu = self.actor.forward(state).to(self.actor.device)
mu_prime = mu + T.tensor(np.random.normal(scale=self.noise),
dtype=T.float).to(self.actor.device)
mu_prime = T.clamp(mu_prime, self.min_action[0], self.max_action[0])
self.time_step += 1
return mu_prime.cpu().detach().numpy()
```

The information must be stored after an action, are saved information about time step in the replay buffer, this is useful for updating our networks with this transitions.

```
def remember(self, state, action, reward, new_state, done):
    self.memory.store_transition(state, action, reward, new_state, done)
```

it is used in the main with the following code:

```
agente.remember(observation, action, reward, observation_, done)
```

Critic

A full time step is done through the environment, then is needed to train our model for several iterations. Is necessary have a mini batch of stored transitions from the replay buffer, the is selected an action for each of the states that we have pulled in from our mini batch.

To achieve a smooth results in action decision a target policy smoothing is applied, hence is necessary just picking an action with our target actor network and add noise to that action.

Note: It has been clipped in order to avoid that the noisy action isn't too far away from the original action value.

Double critic networks is then used hence get Q values for each target critic and then take the smallest of the two for our target Q value. Mean square error is used to compute the loss for the two current critic networks and is also necessary include the Q values, and finally a optimization of the critic is done.

Actor & Network

Like in the paper already mentioned a 2nd time step is used for update the actor the actor's loss function simply gets the mean of the Q values from our critic network with our actor choosing what action to take given the mini batch of states, back-propagation is used for optimize, then update the target networks using a soft update.

```
for i in range(episodi):
    observation = env.reset()

    episode_reward = 0


    done = False


    while not done:
        action = agente.choose_action(observation)
        observation_, reward, done, info = env.step(action)
        agente.remember(observation, action, reward, observation_, done)
        agente.learn()
        episode_reward += reward
        observation = observation_

    history.append(episode_reward)
    average_reward = np.mean(history[-100:])


    if average_reward > best_reward:
        best_reward = average_reward
        agente.save_models()


    print('episodio ', i, 'score %.2f' % episode_reward,
          '100 games  %.3f' % average_reward)
```

2.3.1 Conclusion

The results at first were very unsatisfactory, in a simulation the three-dimensional bipedal robot is not able to achieve the goal.

```
episodio  99 score 50390.88 100 games  50301.774
```

In order to obtain better result it tried to change hyper-parameters changing frequency in actor updating and the noise.

The result slightly improve until 8000.00, but inside the simulation the model is still not able to stand up, probably continuing in change parameters and iterate the training over more than 100 episodes the average reward can grow up. Before lunch the code is necessary create two specific folder “plots” and “tmp/td3”.

reference

[1] https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93

[2] https://arxiv.org/abs/1802.09477