Locomotion Learning for Quadruped Robots Based on Reinforcement Learning

LIU QINGZHEN

A0225465A

E0576099@u.nus.edu

Abstract

Unlike wheeled-robots and crawler-robots, for whom motion will not be a big problem, a significant skill that multi-legged robots are supposed to learn is the stable locomotion. Design of locomotion for multi-legged robots can be a mission requiring a large amount of manual tuning with traditional control methods. While, with reinforcement learning, by proposing different reward functions, we can achieve the learning of multiple kinds of gaits for the robot. Compared with problems of discrete action space, although the continuous action space will take more computing cost, it will increase the flexibility and fluency for the motion of robots and creates more possibilities in learning. In this project, I applied algorithms such as DDPG, TD3 and PPO with *Pybullet Minitaur* environment to achieve the locomotion learning for a ghost robot and tried to apply the idea of 'Transfer Learning' into this project.

Environment and Algorithms

In this project, I use the *Minitaur* environment by *Pybullet*. This environment is a 3rd party gym environment which works like how Gym environments were used.

As it is a continuous action-space problem, I tested this environment with DDPG (Deep Deterministic Policy Gradient) and TD3 (Twin Delayed Deep Deterministic Policy Gradient) written by myself, plus PPO1 (Proximal Policy Optimization) with Stable Baselines.

Environment

Observation and Actions

Observation Space: totally 28 states for this model.

Motor angle observation: 8

Motor velocity observation: 8

Motor torque observation: 8

Quaternion: 4

Action Space: Continuous space from -1 to 1

Torque of each motor: 8

The observation space is different from the method applied in the paper *"Sim-to-Real: Learning Agile Locomotion For Quadruped Robots"* introduced in *Pybullet* for this ghost robot simulation environment. What the paper uses is another environment called *MinitaurTrottingEnv*. The dimension of the observation space of that environment is 4, which are 4 angles for each pair of motors of each leg. We can easily try that environment by changing the environment ID and the input dimensions of the neural network.

Reward Function:

The reward function can be determined by adjusting the weights parameters in the environment.

There are 4 parts of reward provided for us to tune in the basic environment.

Encourage moving forward:

Distance weight * Forward reward

Forward reward is the difference of the current Roll coordinate value and before current action was taken

Discourage shaking and drifting:

- Drift weight * Drift reward – Shake weight * Shake reward

Drift reward and Shake reward are the differences of the current Yaw and Pitch coordinate values and values before current action was taken.

Discourage energy cost:

- Energy weight * Energy reward

ME5406 PROJECT II

Energy reward refers to Torque * velocity * time of one step

Final Reward

*Distance weight * Forward reward - Energy weight * Energy reward* - Drift weight * Drift reward – Shake weight * Shake reward

Max Steps: 1000

Tuning of weights:

How to tune weights of different parts of reward depends on what kind of the motion we want and the which algorithm we use. For example, I tuned the *Shake weight* to be large so as to make the robot move more stable vertically. Thus, the robot will be less likely to fall quickly for each episode. But one bad effect is that the robot may try to use one side of the triangle of each leg to move instead of using the angle part to support the motion.

By the way, we should keep the proportion of each weight so as not to make one reward too large to make other rewards useless.

Algorithms

DDPG (Deep Deterministic Policy Gradient)

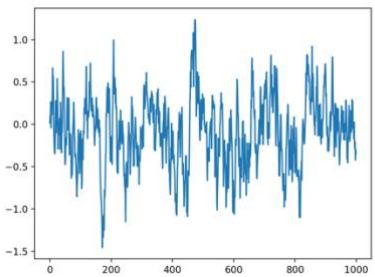
As its name suggests, DDPG makes the process of choosing actions deterministic. Unlike PG (policy gradient), whose Actor Network will return parameters for probability distribution such as Gaussian Distribution to choose action, DDPG's Actor Network returns deterministic actions directly.

Another intuition from comparing the structure of DDPG and DQN is that DDPG is kind of the implementation of DQN in continuous action space. Both DDPG and DQN use Replay Buffer and Target Network to improve the performance on training. Due to the implementation of Replay Buffer, they are all off-policy algorithms, which improves the sample efficiency of their model. Using target Network solves the moving target problem which makes the training more stable. Besides Replay Buffer and Target Network, DDPG uses Critic Network to measure the TD error to update parameters in Actor Network.

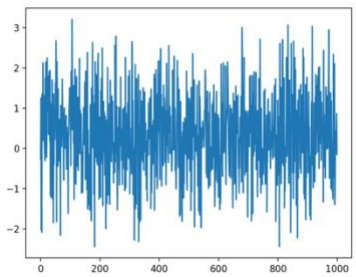
Features of DDPG

Ornstein-Uhlenbeck Noise

Similar to Epsilon-Greedy method used in discrete action space, DDPG uses Ornstein-Uhlenbeck noise in order to add exploration for training. DDPG uses Ornstein-Uhlenbeck Noise, which is a noise that has correlation in time sequence. We can see from the pictures below that Gaussian Noise are independent, but Ornstein-Uhlenbeck Noise doesn't.



Ornstein-Uhlenbeck Noise



Gaussian Noise

Action: Action equals the policy from Actor Network with noise added

$$A = \pi_\theta(s) + N$$

Replay Buffer and Target Network: Similar to DQN

Loss Function:

$$y_i = R + \gamma Q'(S', A', w')$$

$$J(\theta) = -\frac{1}{m\sum_{j=1}^{m} Q(s_i, a_i, w)}; J(w) = 1/m\sum_{j=1}^{m}\left(y_j - Q\left(\varphi(S_j), A_j, w\right)\right)^2$$

y_i is the target Q value; *J(θ)* is the loss function of Actor Network; *J(w)* is the loss function of Critic Network.

As we can see, the Critic Network uses MSE (Mean Square Error) between the target Q value and Q value to update parameters. The critic loss would be small if the difference between target Q value is small. If the critic loss is zero, then the critic network can perfectly predict the Q value of each action. The Actor Network uses the mean value of the Q values to update parameters. This means if the action has a larger Q value from Critic Network, then the loss would be smaller.

"Soft" Target Updates

DDPG uses the parameter τ (0<τ<1) to update parameters in Target Networks:

θ : parameters of Actor Network w : parameters of Critic Network

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \quad w' \leftarrow \tau w + (1-\tau)w'$$

TD3

TD3 algorithm looks like an evolution from DDPG. They have many similar features such as loss function, Replay Buffer, Target Network, Adding noises to actions, etc. Based on these features, TD3 makes some improvements and achieves a better performance.

Features of TD3

Clipped Double Q learning

Difference between TD3 and DDPG is very similar to the difference between Double DQN and DQN. The key idea of their algorithm is the same. They both estimate 2 Q or V values and choose the minimum. This is called '***Clipped Double Q learning***'. They are like 2 pairs of twins in discrete action space and continuous action space. Research has shown that the Q values are often estimated larger by neural networks than the true Q value should be due to the accumulation of noises. So TD3 uses 2 Critic Network to estimate the V value and DDQN uses 2 networks to estimate Q value. They finally choose the smaller one as their V and Q value.

Target Policy Smoothing

Similar to DDPG, TD3 also adds noises to actions chosen for exploration. However, TD3 doesn't apply Ornstein-Uhlenbeck noise as noises in its paper. Actually, we can try to replace the noises with Ornstein-Uhlenbeck noise. Another place TD3 adds noises to is the target actions. This idea is intuitive. We add noises to the output of Actor Network, which is the actions, then correspondingly we should add noises to the target actions to make it 'fair'. This added noise will make the returned Q value smooth and thus decrease the error in learning.

Delayed Policy Updates

Another skill TD3 uses is the Delayed Policy Updates. The agent will update the parameters in the Actor Network every certain steps instead of each step. This trick is easy to apply in coding. I like it.

Training

Prove the correctness of my code

Similar to how I trained the agent in FrozenLake 10x10 map with Monte-Carlo method in project I, this training of quadruped robot also bothered me for a long time at the beginning. The problem is whether the training with the algorithm I use will converge? Does it work for my training? There is no sign in the beginning of training that it will succeed. Reward has no

increase and the robot stays at the origin with weird actions. For example, the robot looks dumb with TD3 algorithm at the beginning. It works better with actions chosen by noise than actions chosen by neural networks. The robot repeatedly chose the similar actions that will make the environment terminate its episode.

To solve this issue, I came up with 2 methods. One is to use an easier gym environment to test. So I chose *Gym Pendulum* and *HopperBullet* environment to test the algorithm and it works. Another is to use the baselines to test the *Minitaur* environment and check whether the robot looks as weird as my program works. The result is YES. So the problem should be with the environment. In other words, for this environment, the robot will look dumb in the exploring period. What we need to do is just let the agent train for more episodes.

Tuning parameters of environment and agent

As the video in the link shows, every episode was terminated quickly after the robot chose the 'wrong' actions. This too quick termination will lead to lack of training data. With the default setting of reward weights, the robot's dumb actions will receive a positive reward, which is not what we would like to see. The noises added for exploration seems doesn't help at all.

To solve the quick termination problem, I tuned the parameters in the environment. These can be tricks to improve the speed of training.

- Increase the Shake weight to discourage shaking
- Modify the ***is_fallen*** and ***termination*** functions to make it terminate harder:
  - This will make the agent learn a good gait in the changed environment, then we can change the parameters back to the default environment to train. However, sometimes the transplantation to the default environment doesn't work well. The default parameters make it too strict to view the gait as fallen.
- Fine-tune the reward function

There are also many parameters to tune for the agent:

- Learning rate of the optimizer: If the learning rate is too small, then the agent may fall into local optimum; if it is too large, then the agent may learn nothing.
- Batch size and replay buffer size: Theoretically, the larger the batch size and replay buffer size are, the better the performance will be. Because the agent will get more experience. Larger batch size will help to reduce the bias of training samples. However, if the batch size is too large, then there will be problem with our memory and training speed.
- Scale of noise: The noise plays as the role to improve exploration. It should be set depending on the value of action space. In fact, it is possible to use 'noise-decay' which could work the same as the 'epsilon-decay' method.
- Warmup steps: I set it to 1000 steps which is the largest number of steps for an episode.

Increase the Replay Buffer initial size

The original setting of my model was when the data in the buffer is larger than the batch size, then the agent will sample from the buffer to update parameters in neural networks. I found that this may lead to the outcome of not enough training examples to make the training efficiency low. So I set a parameter called ***Replay Initial***, which is 1000 or 2000, which equals the steps of 1 or 2 full training episodes of training. Only when the buffer received more than this number of data will the agent update parameters in neural networks.

Concepts from 'Transfer Learning'

In my bachelor final year project, I tried transfer learning for image classification. The idea is to borrow CNN parameters of weights from the model trained on large dataset such as ImageNet. Then we train parameters of some layers with our dataset. It turned out to work very well and saved a lot of time. However, there's a difference in this project. In reinforcement learning, there is no such environment as datasets like ImageNet, Coco in computer vision.

My idea is to train a new model based on the already well-trained model with different reward function weights and environment setting. This operation will make the agent escape the hard time at the beginning of a new training, thus saving our computing cost. And it really worked in my training, which will help to speed up the beginning part of training, which can be seen in the Results part of this report.

Network Structure

CriticNetwork: (fc1): Linear(in_features = 36, out_features = 400, bias = True) (relu1): ReLU() (fc2): Linear(in_features = 400, out_features = 300, bias = True) (relu2): ReLU() (q1): Linear(in_features = 300, out_features = 1, bias = True) ActorNetwork: (fc1): Linear(in_features = 28, out_features = 400, bias = True) (relu1): ReLU() (fc2): Linear(in_features = 400, out_features = 300, bias = True) (relu2): ReLU() (action): Linear(in_features = 300, out_features = 8, bias = True) (tanh): Tanh()	CriticNetwork: (fc1): Linear(in_features = 36, out_features = 256, bias = True) (relu1): ReLU() (fc2): Linear(in_features = 256, out_features = 256, bias = True) (relu2): ReLU() (fc3): Linear(in_features = 256, out_features = 256, bias = True) (fc4): Linear(in_features = 256, out_features = 256, bias = True) (q1): Linear(in_features = 256, out_features = 1, bias = True) ActorNetwork: (fc1): Linear(in_features = 28, out_features = 400, bias = True) (relu1): ReLU() (fc2): Linear(in_features = 256, out_features = 256, bias = True) (relu2): ReLU() (action): Linear(in_features = 256, out_features = 8, bias = True) (tanh): Tanh()
Network I	*Network II*

I used Pytorch to build the neural network structure. TD3 has 1 Actor Network with 1 target network and 2 Critic Networks with 2 target networks. DDPG will add batch normalization after each fully connected layer, but only one target network for Actor Network and Critic Network respectively.

One of my network structure of the Actor Network and Critic Network follows the parameters used in the paper of TD3. Just the input and output dimensions of each FC layer is different. The original TD3 code uses (256,256). In another one, I added the number of layers to test the training, which makes the training faster.

Because there is no image in our input data, so we don't need the network to be so deep, MLP will be able to handle this problem. As to the number of neurons in each layer, it doesn't need to be too large, too.

Results

DDPG

Agent parameters: Actor network learning rate = 3e-4; Critic network learning rate = 3e-4; input dimensions = [28]; number of actions = 8; batch size = 64; gamma = 0.99; tau = 0.005; Replay buffer size: 5000; Noise: ($\sigma = 0.15;\ \theta = 0.2; dt = 0.2$)

DDPG is the first algorithm I tried to use for this environment. Due to the not so good performance of DDPG, I changed to TD3. The training record could be found in my uploaded files.

TD3

Agent parameters: Actor network learning rate = 3e-4; Critic network learning rate = 3e-4; input dimensions = [28]; number of actions = 8; batch size = 128; gamma = 0.99; tau = 0.005; Replay buffer size: 100000; Noise = Normal Distribution ($\sigma = 0.2$); Warmup steps = 2000; Replay initial = 1000

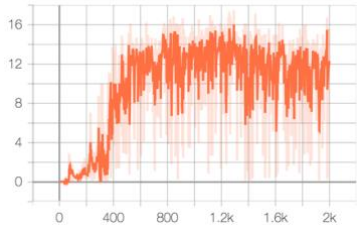
Training I (Nov 7th): Trained with Network I

TD3 is the algorithms I used for most of my trainings. Below are the records of a successful training which played for 2000 episodes and took about 3-4 hours on RTX 2080Ti with my network I.

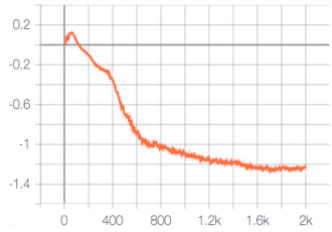
As we can see, it started to converge at around the 600th episode. After that, the score and the actor loss of each episode became stable. Because of the noises we add to improve exploration, the critic loss has more fluctuation than actor loss looks like.

Reward weight:

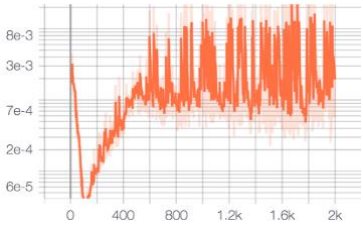
Distance weight=1.0; Energy weight=0.0005; Shake weight=0.2; Drift weight=0.1,



Score of Each Episode



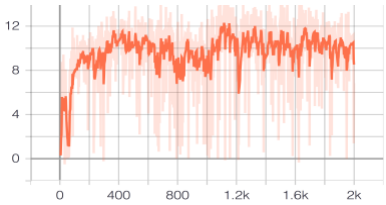
Actor Loss



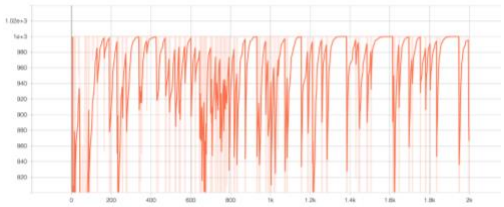
Critic Loss

Training II: 'Transfer Learning' Based on Training I with different reward function (Nov 11th)

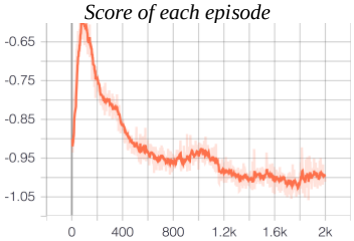
This training was also a 2000 episodes training. In this training, I changed the reward function setting of the environment. As we can see from records from TensorBoard, it only took about 200 episodes to start to converge under a new reward setting with the model we trained before. This benefit came from the idea of 'Transfer Learning' I talked about in Training part.



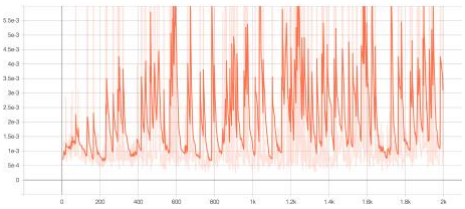
Score of each episode



Steps of each episode



Actor loss



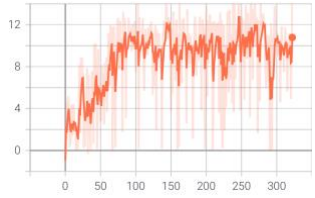
Critic loss

Reward weight:

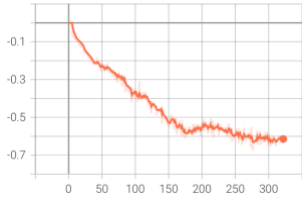
Distance weight=1.0; Energy weight=0.005; Shake weight=0.10; Drift weight=0.25

Training III: Trained with Network II (Nov 17th)

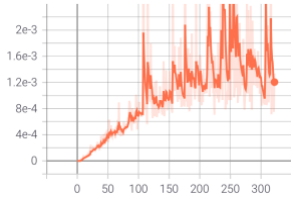
In this training, I applied the second neural network structure. The outcome is interesting. It started to converge extremely quickly than training with Network I before. As you can see, the score of each episode starts to growing from its first 10 episodes. In Training I, the score starts to leaving zero after about 400 episodes of training. However, the time of training each episode become longer. It took about 1.5 hours to train for 300 episodes, 260,000 steps. But generally, the time consuming is less than training with Network I, which is a good result.



Score of each episode



Critic loss

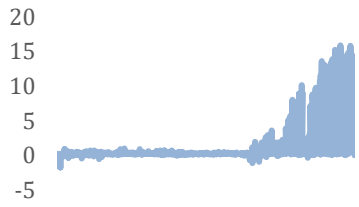


Actor loss

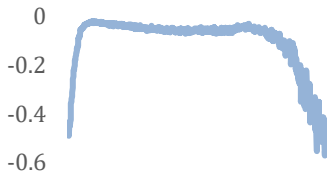
Training Records IV: Reduce the Penalty on Shaking and Drifting

In this 2000-episodes training. I cut down on the weight for discouraging shaking and drifting compared with Training I. Changing of reward weights made the model hard to train. The score started to increase after 1500 episodes of training. And at the end of 2000th training, this training didn't converge yet. The score is still ascending and the actor loss, which reflects the

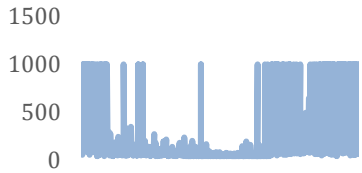
negative mean Q values of actions, is still going down. This means the agent is choosing actions better and better. As we can see, steps of each episode are very low before about 1300 episodes. This means the actions the agent chose made it fall very quickly, thus resulting the low score of that episode.



Score of Each Episode



Actor Loss



Step of Each Episode

If we let the agent train more episodes, then it should converge.

Reward weight:

Distance weight=1.0; Energy weight=0.0005; Shake weight=0.1; Drift weight=0.1,

---

Agent's Performance

Beginning of Training:

Choose actions based on normal distribution in the first certain steps and store the states, new states, actions, reward, done into the replay buffer. With these noise-based actions, the agent will move slowly but smoothly forward or backward.

Steps larger than max of *Replay Initial* and *Batch Size*

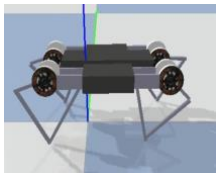
When the number of steps is larger than the value of parameter *Replay Initial*, then the agent will start to learn something. We sample data of *Batch Size* and transfer the data into Tensors for Pytorch to update parameters of networks. Then, the gait starts to look really weird. It jumped ahead and got stuck with its hip in the air stretching its hind legs. Or it lost balance and got fallen.

After tens or hundreds of episodes training

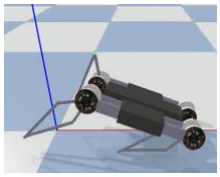
It learned to walk with a weird gait. Its left legs kept on the floor and right legs kicked the floor to move forward but slowly. Or it also learned to move with 3 legs with right hind leg in the air.

2000 episodes training

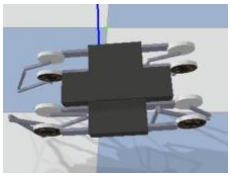
It learned a kind of good gait. But still need to be improved.



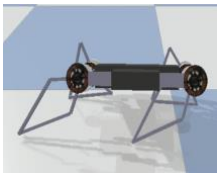
Gait with normal distributional noise



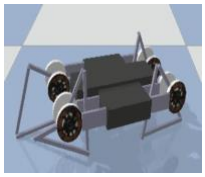
Start of training: 1



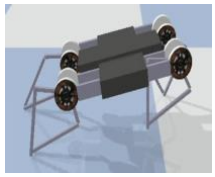
Start of training: 2



Weird Gait 1



Weird Gait 2: 3-legs moving

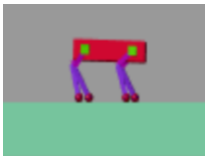


Final Gait

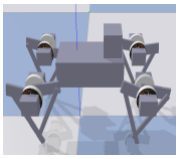
Comparison Between RL TD-3 and Conventional Control Raibert Controller

Conventional control method requires us to build the model of forward kinematics and inverse kinematics. Because it is based on mechanics, so there will be high requirement for simulation environment to simulate the real ground contact models. Raibert controller is a very good conventional quadruped robot locomotion control algorithm. Marc Raibert is well-know as the co-founder of Boston Dynamics. I tested the Raibert Controller with a 2-d Matlab simulation and Pybullet Raibert Controller. The result is it moves well in a low speed. But as the speed increases, its gait started to get bad. Maybe there should be another controller designed for speed changed.

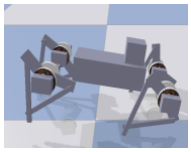
What's good of reinforcement learning is that it can adjust its gait with the reward feedback. And we don't need much time to design a new mechanics model or controller by hand.



Raibert controller with MATLAB



Raibert controller with speed=0.35



Raibert controller with speed=2

Discussion

Advantages and Limitations:

The advantage is clear. Compared with conventional methods, RL provides an end-to-end way to help the robot learn the gait. Compared with algorithms such as Policy Gradient, DDPG, TD3 performs better on this continuous problem with this large dimension of observations and actions. Another advantage is that this training could be done in simulation, which will decrease the cost of training in real environment.

One of the limitations could be that the RL method cannot directly control how the gait looks like. To make the robot learn the gait that we want. To achieve this, I think we should introduce Imitation Learning to this model.

Challenges:

Continuous action space and high dimensions of observation

One challenge is to use policy gradient algorithm for continuous action space, another is the high computing cost for this problem.

Solution

To change the mind-set of value-based methods to policy gradient methods takes some time. But not too hard. What troubles me a lot is the poor configuration of my computer, which is an old, no-Nvidia-GPU-supported MacBook. Firstly, the computer at my home with RTX2060 comes to my mind. I taught my parents to use TeamViewer but I gave up due to the low quality of connection and the ban from NVIDIA of CUDA and CUDNN in China. Then I asked about how to use NSCC. It could work, but hard to configure the environment. I pulled the docker and build my own container. I gave up again for I cannot render my training on server. Finally, I found that I can use the GPU in the lab. This is wonderful.

Robot got fallen too fast resulting in lack of exploration

This problem was caused by both the environment setting and the actions the agent choose. The default setting of the environment will perceive the robot as fallen If the up directions between the base and the world is larger (the dot product is smaller than 0.85) or the base is very low on the ground (the height is smaller than 0.13 meter). This criterion is too strict for a dumb robot trying weird actions at the start of training, which will make the robot end its episode very fast. This is very bad for the robot because it will be hard for the robot to get enough observations of the world.

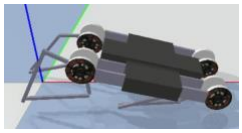
Solution

Solution is easy, just tune the parameters lower to make it harder to be seen as fallen. I tuned it to 0.70 and 0.10 and it looks good. But

---

Robot tends to choose actions with positive reward but make it fall on the earth

Robot will choose to jump ahead at the beginning. This action will bring positive reward because of it moves forward for a certain distance, but it will actually make the robot hard to run. This problem looks very like overfitting in supervised learning.



Solution

I increased the Shake weight to offset the positive reward of moving forward. Noises we add in choosing actions will also help to solve the problem.

Can't not load the trained zip model with my computer; TensorBoard doesn't work on my base environment

One reason is because my model was trained on a GPU and I want to load it with GPU. Another it that it seems the version of my Pytorch cannot load the zip file, so I have to save the network weights file with _use_new_zipfile_serialization=False. TensorBoard worked after I created a new conda environment. Maybe it is because there are some conflicting.

Lessons learned:

This project is time-consuming due to its relatively large input scale and continuous action space. However, because of this, I was able to face many situations that cannot be met in an easily, fast-trained problem, such as the

We have to provide the agent with enough experience and data to learn the best policy. it works for a person too. One person should not be short eyesight, we need to explore more of the world even if we will encounter a variety of difficulties and setbacks. The more one experience, the more one will learn.

Increasing the depth of neural network to a suitable may cut down on the episodes of training, even though the training time for each episode became longer because of the computation of neural network increases too, the total time it cost was lower. However, if the depth is too large, then the time may increases instead.

The idea of transfer learning is also great to save our time for training.

My questions and ideas:

How do we judge what is a good gait and what is the optimal gait for the robot?

For example, the weird-looking gait and the normal-looking gait will all get positive reward for each action, but there is definitely a difference between how we judge them. Another interesting thing is that the robot may appear to have a good gait after training, but when I tested with CPU, which is slow in choosing actions, then the gait will have some problem. It's like the overfitting problem in supervised learning. However, there is a test set and labels of each element in supervised learning. But in reinforcement learning, there is no such thing. We do not know whether we have reached the optimal policy in problem like this. I talked with my friend of my bachelor school about this problem. It baffled us for a long time. Finally, I recall the idea of 'exploration and exploitation'. The noises we added will help the robot to explore.

How do we set reward?

If we want the robot to move fast, then we can calculate the distance/steps to give reward to it.

If we want the robot to move far without falling, then we should give reward based on the distance of each episode. It could be a sparse reward.

However, what if the goal is to let the robot handle a good-looking gait? I find it hard. For this problem, maybe we need to add friction reward between torque and such things. Maybe imitation learning is a good solution.

What does negative loss mean? Why is the critic loss going up?

As we can see from the loss in training records, although they will finally converge, the actor loss is negative and the critic loss goes higher. This looks weird if we are familiar with supervised learning problems such as image classification. The loss should descend and be positive in that kind of problems.

However, in DDPG and TD3, the actor loss was negative. Negative loss means we will update the parameters to make it goes to the direction to make it more negative. It means it will make the actor choose the actions with greater Q value.

Why is critic loss going up? Shouldn't the loss go down? The answer is it depends on what our loss function is and what our goal is. In training, due to the changing of increasing reward, the loss will get larger. When our reward of actions is stable, then the MSE loss would be stable too.

Anyway, to judge the performance of a reinforcement learning model, maybe reward or score is the better criterion compared with loss.

Future scope:

After learning how to run, we can add some obstacles in the environment which is the URDF file. I will add a camera to the robot to increase its perception ability. Thus, the neural network should be deeper. CNN is necessary for dealing with images input. Then we let the robot learn obstacle avoidance. In this step, we don't even make big change to our reward function. And then we can set a goal to reach with a large reward to make it a navigation problem.

As to the algorithms of reinforcement learning, I should take more experiments by tuning the hyperparameters. Tuning of learning rate, batch size, etc. is a really troublesome process. But it is definitely important for our training.

Disclosion

Code

Firstly, I referred to the code of Machine Learning with Phil on Youtube in its network structure, hyperparameters and saving and loading models. Then I wrote my DDPG code. Due to the performance of DDPG is not good, I upgraded my code to TD3 algorithm. In writing the final code, I referred to the TD3 paper code and tried to modified my neural network structure and hyperparameters such as noise, learning rate, batch size, replay buffer size, etc. I introduced Tensorboard to log my training.

Reference

[1] Jie Tan, Tingnan Zhang, Erwin Coumans,etc.Sim-to-Real: Learning Agile Locomotion For Quadruped Robots arXiv:1804.10332v2 [cs.RO]

[2] Fujimoto, S., van Hoof, H., and Meger, D. Addressing function approximation error in actor-critic methods. arXiv preprint arXiv:1802.09477, 2018.