Explore different reward design for 2d robot arm task and analyzing SAC and TD3's performance on those rewards

Hantang Li hantang.li@mail.utoronto.ca
1004302890

Xuan Du xuand.du@mail.utoronto.ca
1004079256

Abstract

In this paper, we explained a simple 2d robot arm environment and constructed three reward functions to improve the default negative euclidean distance reward. We tested those rewards on the Soft Actor-Critic algorithm and the Twin Delayed DDPG algorithm then evaluated their performance. After experiment and analysis, we found that the Exponential negative Euclidean distance reward performs the best in sample efficiency and performance.

1. Introduction

Deep reinforcement learning (RL) has emerged as a promising approach for robotic control. It is believed that analyzing a simple low-level 2d environment could help understand the more complex 3d environments. Considering the importance of reward function in the deep reinforcement learning algorithm's training process, we would like to handcraft different reward functions and test them on Algorithms that focus on exploration and stability. We will craft reward functions under two major categories: discrete/continuous. Those reward functions will evaluate the reward of a robot arm to reach a ball. We will test those reward functions on Soft Actor-Critic (SAC) and Twin Delayed DDPG (TD3) and analyze the performance.

Code:`https://github.com/Hantang-Li/CSC498_Project`

2. Algorithm and Environment

2.1 Twin Delayed DDPG (TD3)

Twin delayed DDPG (TD3) Fujimoto et al. (2018) is built on Double Q-learning and improved on Deep deterministic policy gradient decent (DDPG). TD3 updates the policy less frequently, called delayed policy updates and adds noise to action to smooth q function errors. DDPG has a problem: if the target is overestimated or underestimated initially, it will keep over/underestimating these targets until the end. By using TD3 in our task, we can observe the benefit of policy smoothing and delayed updates and how our reward design will affect that.

2.2 Soft Actor-Critic (SAC)

Soft Actor-Critic (SAC) Haarnoja et al. (2018) is an off-policy algorithm that uses an actor-critic approach to learn policy and value function separately. Similar to TD3, it uses a clipped Double-Q Learning trick to reduce maximization bias. The major difference between

SAC and TD3 is the use of entropy regularization and stochastic policy. This is also the central feature of SAC. This feature allows SAC to explore more solutions that could reach the goal, and it has a stronger exploration ability compared to TD3. The implementation of this feature is by adding an entropy that evaluates how surprised encounter a new state action choice (the higher the entropy). By doing that, the agent will prefer to explore new actions for each state.

2.3 Purpose of choosing SAC and TD3

The purpose of choosing SAC to evaluate our reward is because we trust its exploration ability, we are confident that more reward functions will work on SAC, and we can evaluate different reward functions by comparing SAC's, and we will start to test our reward on TD3 after knowing SAC is able to converge. And we would like to investigate how Entropy-Regularized Reinforcement Learning algorithms explore the environment using different reward functions. If one of the reward functions we implemented works on both SAC and TD3, we are able to compare their performance.

The purpose of choosing TD3 is to investigate how much the target policy smoothing, delayed policy update, and other tricks to minimize maximization bias that SAC does not have will change the result of the convergence performance.

2.4 OpenAI Gym 2D Robot Arm Environment

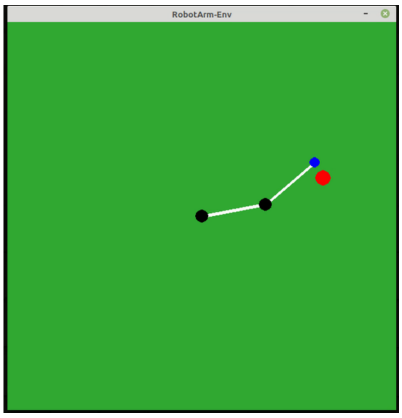


Figure 1: 2D Robot Arm Environment ekorudiawan (2019)

The environment is inside a 600*600 pixel frame. A robot arm consists of two joints and two links; each link is 100 pixels in length. The joints' maximum angle is 90 degrees. When the task begins, a red ball will be randomly generated on the frame that is within the robot arm's range, and then the robot arm will take actions that lead its tip to the ball.

State: A tuple that includes the euclidean distance between the robot arm's tip to the ball (in x, y direction), and the robot arm's two joint's angle in terms of radius.

Action: It specifies the robot's position at the next state. The data encodes the robot arm's angle that the robot will be switched to in the next iteration. Its data type is A tuple consists of the following value:

(joint 1 value (in range -1 to 1), joint 2 value (in range -1 to 1))

Those values within range 1 and -1 will then be mapped to a degree between 0 to 90 and passed to the robot arm. The robot arm's two angles will rotate based on the value. ekorudiawan (2019)

2.5 Challenges of the environment

The challenge is making the robot arm reach the goal as soon as possible and as accurately as possible. The original euclidean distance reward function does not perform quite well, so we want to modify the reward function and see if we can achieve some better results.

3. Investigation

We will investigate how each algorithm performed under different reward implementations. First, we will list the "reward road map." We will explain each reward's thought process and the intuition and reasoning on how the reward is connected to the reinforcement learning agent. Second, we will show SAC and TD's performance on each reward and provide a detailed explanation.

All the reward we are using is based on the Euclidean distance between the robot arm's tip and the red ball. We will use *distance_error* to represent the Euclidean distance between the ball and the arm tip.

Note: To detect if the robot arm's tip reaches the ball, we check whether the Euclidean distance between the ball and the tip is within 5 pixels.

3.1 Negative Euclidean distance reward

```
reward = - distance_error / 100
```

The environment will reset after the tip reaches the goal.

The reward is the default implementation that the environment's author constructs. We want to penalize the agent if the robot arm's tip is far from the goal. Here we use the euclidean distance divide by 100 because we want to scale the reward at each step to be close to one to avoid numerical overflow problems if the trajectory is too long. We use negative rewards here to let the agent minimize the total cost of reaching the target, which means we want the agent to approach the target as soon as possible.

3.2 Exponential negative Euclidean distance reward

```
reward = - (distance_error / max_arm_length)^0.5
```

The environment will reset after the tip reaches the goal.

The reward is built to improve the default reward function. We kept the reward at each step to be negative and made two modifications.

An exponentially increasing reward as the arm approaches its goal, where distance error is the euclidean distance between the end effector of the arm and the goal, and

max_arm_length is the maximum distance between the goal and the maximum possible edge that the robot arm could reach.

1. We use the max_arm_length as the denominator. By doing this, we can scale the reward to be always within one, so algorithms that take a long time to explore will worry less about summing too many rewards at one time.

2. We take a square root of the reward to make the cost shrink exponentially, so the agent will be rewarded more as it comes closer to the target.

3.3 Counting reward

A reward will be given by comparing the current step's euclidean distance to the last step's euclidean distance. And the environment will reset when either the reward accumulates to 10 or -10.

If the distance increases or keeps the same, the arm tip is moving further from the goal, a -1 reward will be given to the agent.

If the distance decreases, which means the arm tip moves toward the goal, a 1 reward will be given to the agent.

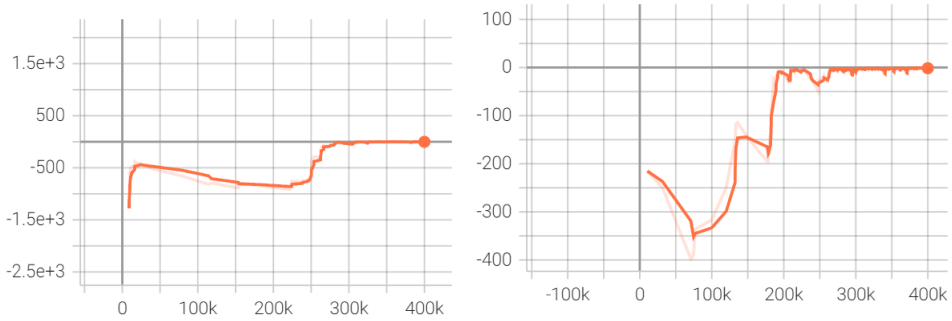
The reward is designed to encourage the agent to move toward the goal, if the agent learns always to move forward, then it will eventually touch the ball. We also enable early termination if the agent always makes a bad move, which the continuous reward mentioned above could not do because the reward for previous reward functions is always negative, early termination will confuse the agent and makes the algorithm learns to maximize the reward in a wrong way. However,

3.4 Level wise counting reward

We inherited the advantage of counting rewards and made this sparse reward. This reward function will penalize the agent by giving a reward of -1 if it moves back to the outer level, just like counting reward does, but it will not encourage the agent to move forward continuously. Instead, it gives a step-wise increasing reward that can only obtain once for each donut-shaped level based on how close the tip is to the ball. Note that the environment will reset when the reward accumulates to -10 or the tip reaches the goal.

For example, if the tip is 50 distances away from the ball, but the distance is less than 100, a reward of one will be given; if the tip is 10 distances away from the ball, but the distance is less than 20, a reward of five will be given. We encourage the agent to move the robot arm tip to the goal, instead of just increasing or decreasing the distance between the tip and the ball.

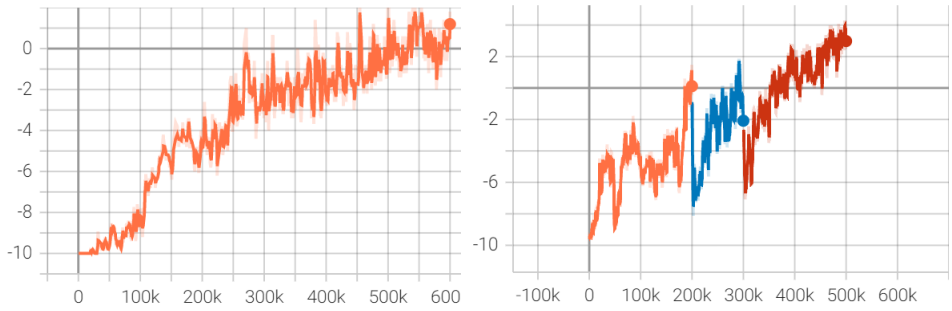
3.5 SAC's performance on each rewards



(a) Negative Euclidean distance reward (b) Exponential negative Euclidean distance reward

Figure 2: Continuous rewards, X-axis(time steps), Y-axis(mean reward on each episode)

The training graph shows that our modified "Exponential negative Euclidean distance reward" has a lower convergence time than the original euclidean distance reward. By looking at the reward change among all the time steps, we observe that the reward fluctuates similarly on a large scale, which shows the algorithm explores the environment in a similar manner. So, in terms of the training curve, the Exponential euclidean distance reward performs better on the algorithm's convergence.



(a) Counting reward (b) Level wise counting reward

Figure 3: Discrete rewards, X-axis(time steps), Y-axis(mean reward on each episode)

Due to technical difficulties, the level-wise counting reward's training curve appears to be discontinuous because we stopped and resumed training at 200k and 300k, respectively. Since we are still using the same environment and reward function to resume training, the affection on models performance can be ignored. The training graph shows that level-wise counting reward has a stepper learning curve and converges faster than the counting reward. So, the Level wise counting reward performs better on the algorithm's convergence and has better sample efficiency in terms of the training curve.

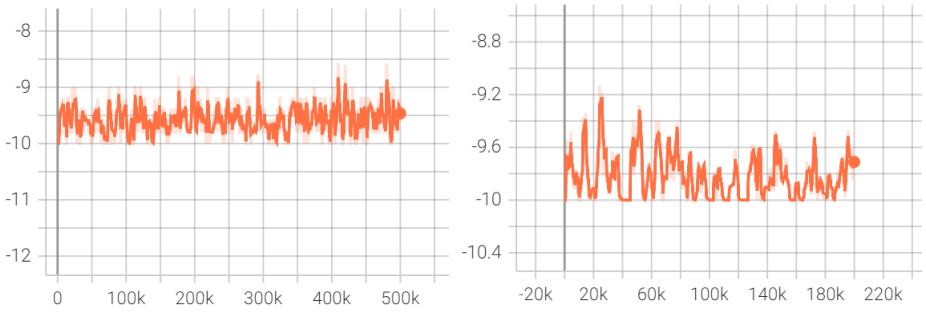
We test the training result by running the model 50 episodes. Each episode consists of 50 timesteps. The environment we use to evaluate the model will not calculate any rewards, only returns $done = True$ if the robot arm tip reaches the goal. We will count the number

of episodes that the agent successfully reached the ball within 50 timesteps, then calculate its ratio. The result for each reward function is shown below.

Reward function / Random seed	0	1	7
Negative euclidean distance reward	0.88	0.94	0.92
Exponential negative Euclidean distance reward	1.0	1.0	0.98
Counting reward	0.7	0.74	0.64
Level wise counting reward	0.98	0.96	0.94

Based on the testing result, the Exponential euclidean distance reward outperformed the regular euclidean distance reward in all three random seed settings. Level-wise counting reward outperformed the regular Counting reward in all three different random seed settings. Unexpectedly, the level-wise counting reward also outperforms the regular euclidean distance reward. This could indicate that sparse reward could be a good choice if we don't need to care about sample efficiency since the discrete reward we tested is not as sample efficient as continuous rewards. In summary, the Exponential negative Euclidean distance reward performs the best out of all the other rewards we tested. It indicates while designing a reward function for SAC, a continuous reward that grows exponentially as approaching the goal is preferred.

3.6 TD3's performance on each rewards



(a) Counting reward (b) Level wise counting reward

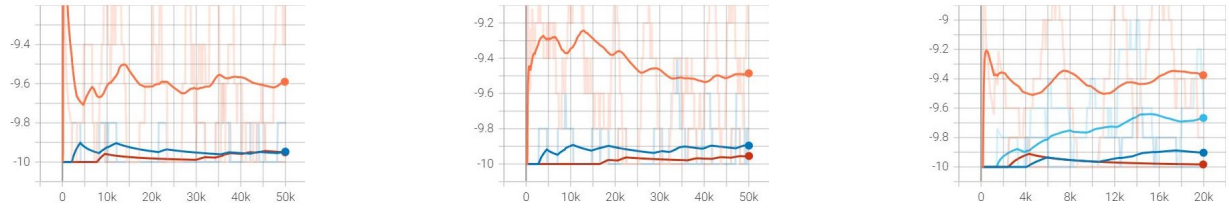
Figure 4: Discrete rewards, X-axis(time steps), Y-axis(mean reward on each episode)

The TD3 algorithm was stuck in the first episode and could not explore the ball, so it failed to learn the task while using euclidean distance rewards and exponential euclidean distance rewards. And from the observation of the training curve, the algorithm is stuck at an average reward per episode around - 9.8 and fails to learn the policy for counting reward and level-wise counting reward. One possible cause is TD3's exploration ability. Since TD3 uses deterministic policy, it only explores when random noise is added to its action, and in our case, the random noise we added might be too small, and TD3 is not able to explore properly. Another possible cause is the (Guillaume et al.) stated that when the algorithm finds the reward early in the training session, it is also more successful in converging to the optimal policy Guillaume Matheron (2020). From our observation, the early training

of episodes does not find the pattern of getting a continuous reward. The replay memory gets polluted and will keep producing non-sense policy. So, we will try to fine-tune its hyper-parameter.

The reward function we will use to fine-tune the hyper-parameter is the Counting reward because the agent will receive a solid +1 or -1 reward at each step and the agent only need to learn how to move closer. And we will run each experiment for 50k steps to observe whether the algorithm starts to converge.

Param	exp1 (Orange)	exp2 (Blue)	exp3 (Red)
Target Noise, Target Clip	[0.0, 0.0]	[0.2, 0.5](default)	[1.0, 1.0]
Update Freq	1	2(default)	5
TAU	0.001	0.005(default)	0.1



(a) Target Policy Smoothing (b) Update frequency (c) Tau values

Figure 5: TD3 tune hyper parameters, X-axis(time steps), Y-axis(mean reward on each 10 episode)

In the experiment, we tuned hyper parameters by varying target policy smoothing with target noise and target clip, updating frequency and tau values. All of these did not affect the not converge problem. For example, 'Target Policy Smoothing' is added to reduce the variance of the learned policies. The paper suggests using a small Gaussian noise with a standard deviation of 0.2 clipped within the range of -0.5 to +0.5 Fujimoto et al. (2018). However, to make them less brittle, we make the critic gradient is clipped with a maximum value being 1 to enhance the training stability in Figure5. When TAU is set to 1, actor and critic network are directly copied to target network, which makes negative effect, the suggested tau in paperFujimoto et al. (2018) is 0.005, making 1 indeed get a negative effect. However, even smaller tau(0.001) make odes not learning worse but have more volatile noises.

3.7 Conclusion

Under this 2d robot arm environment, we built different reward functions and used them to train on SAC and TD3 with measuring its performance. We've shown Exponential negative Euclidean distance reward, and level-wise counting rewards outperform Negative Euclidean distance reward and counting reward, respectively. We discussed the benefit of using these reward functions. The TD3 failed on this task. We discussed the potential reason, including the random noise might too small, causing exploration failed, or the early training polluted replay memory and agent constantly getting negative effects. We

also tried tuning hyperparameters and it is not helpful. As we found the exponential euclidean distance reward works best among all the rewards we proposed, Future researchers could explore different root functions' effects on SAC's performance, and people also could investigate more on the reason behind TD3's bad performance.

References

ekorudiawan. OpenAI Gym 2D Robot Arm Environment, 12 2019. URL `https://github.com/ekorudiawan/gym-robot-arm`.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018.

Olivier Sigaud Guillaume Matheron, Nicolas Perrin. The problem with ddpg: Understanding failures in deterministic environments with sparse rewards. *ICLR*, 2020.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018. URL `http://arxiv.org/abs/1801.01290`.

3.8 Appendix

Code:`https://github.com/Hantang-Li/CSC498_Project`