Experiments on Olympic Running Games

Qirui Chen [*] **Bo Peng** [*] **Yuxi Wei** [*] **Zi Wang** [*]

Abstract

Reinforcement learning algorithms can be used to address problems in a variety of domains, including robotics, distributed control, and so on. In the reinforcement learning process, the agents must discover a solution in their exploration, using learning. However, there are few chances to train a reinforcement learning-based system in the real world. Luckily, Jidi provides us a platform to simulate Olympic running, where two agents compete to arrive at the terminal. We have run the PPO algorithm, a baseline method, on each map and shuffle. As for other attempts, we changed the algorithm to some other powerful ones, including DDPG, TD3, and SAC. In addition, we propose a geometry-based method to modify the reward mechanism based on four rules we made artificially. Also, we followed the suggestions of training strategies on Github to improve the performance of the controlled agent. The detailed experiment results are shown in the paper. The four of us contribute equally, and the authors are in alphabetical order.

1. Introduction

In the last several years, game theory and reinforcement learning have attracted rapidly increasing interest in the machine learning and artificial intelligence communities. It programs agents by reward and punishment without needing to specify how the task is to be achieved. The agents must learn behavior through trial-and-error interactions with a dynamic environment. In this project, Jidi provides us with a platform to simulate Olympic running games so that we can train agents based on reinforcement learning. All the process is done in this simulation environment instead of the physical environment in our real world.

[*]Equal contribution . Correspondence to: Qirui Chen <519030910365>, Bo Peng <519030910366>, Yuxi Wei <519030910369>, Zi Wang <519030910345>.

Proceedings of the 39th International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022. Copyright 2022 by the author(s).

First, we have trained and evaluated the baseline PPO (Schulman et al., 2017) algorithm in all maps and every single map. Then we replaced the PPO algorithm with several different algorithms, such as DDPG (Deep Deterministic Policy) (Lillicrap et al., 2015), TD3 (Twin Delayed DDPG) (Fujimoto et al., 2018), and SAC (Soft Actor-Critic) (Haarnoja et al., 2018). DDPG combines ideas from DPG (Deterministic Policy Gradient) and DQN (Deep Q-Network), expanding DQN into continuous action space. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy. TD3 is an algorithm that addresses the overestimation of Q-values in DDPG by introducing three critical tricks: Clipped Double-Q Learning, "Delayed" Policy Updates, and Target Policy Smoothing. Soft Actor-Critic (SAC) is an algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and DDPG-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. We have employed the above algorithms in this environment and conducted a lot of experiments to evaluate their performance compared with the PPO algorithm.

We also modified the reward mechanism based on four rules we made artificially. The detailed rules are as follows: 1) We want agents to go along the landmarks (arrows). Thus we calculate the direction of arrows through a geometry-based method. We encourage the angle of agents' action to be close to the direction of arrows. Otherwise, we punish agents. 2) Observing that agents sometimes get stuck in corners or keep hitting the walls, we want agents to avoid these behaviors. Hence we calculate the distance to the walls from the agents' view. The larger the distance, the better. 3) Noticing that sometimes agents are almost crossing the finish line, while they turn around and miss such valuable opportunities. To tackle this issue, we give a penalty to agents when this happens. 4) The race should not be won or lost by crossing the finish line! We want to encourage the agents to move as far as possible. Therefore, we use BFS (Breadth-first search) to calculate each position's distance to the finish line. Then we will give a reward to agents according to this distance. We validate our updated reward mechanism separately and the results will be shown in section 4.

In addition to the exploration of algorithms and reward mechanisms, we have also considered the training strategy and proposed several methods to improve the performance and generalization of the model on all maps when evaluated. Concretely, we have tried the following strategies: 1) train the model on all 11 maps **in turn**, instead of random shuffle; 2) change the discrete action space of the PPO algorithm into a smaller grained division; 3) use self-play based training, substituting the random opponent; 4) shuffle the control index when training in a new episode. We have found these methods can help the agent adapt to different positions and maps, with better convergence and generalization.

In a nutshell, our main contributions can be summarized as follows:

• Besides PPO algorithm, we tried some other algorithms, including DDPG, TD3, SAC.

• We proposed four ways of calculating rewards, which are more reasonable under the setting of Olympic running games.

• We followed the suggestions on Github to improve performance by changing the training strategy.

• We conducted many experiments to validate the effectiveness of our proposed methods.

2. BackGround

2.1. Competition Olympics Running

Platform Jidi provides an environment for the Olympic running game of agents. In this series of races, two intelligent agents compete in running races with the goal of reaching the finish line as quickly as possible.

The environment rule is:

1. In a random map, each player controls an elastic ball agent with the same mass and radius.

2. Agents can hit each other and walls, but at a loss of speed

3. The agent has an energy of its own, and the energy consumed at each step is proportional to the force and displacement applied.

4. Intelligent energy regenerates at a fixed rate, and if energy decays to zero, no force can be applied.

5. The observation of the agent is the rectangular area of itself facing forward 25*25, and the observation values include the wall, finish line, its opponent and the track direction auxiliary arrow.

6. At the beginning, the agent is located at the starting line of the map, and its initial orientation is parallel to the direction of the runway.

7. When an agent reaches the finish line (red line) or the environment reaches the maximum number of steps of 500, the environment ends, and the one who crosses the finish line first wins. f neither side crosses the line, it is a draw.

8. An agent needs to be generalized to fit different maps, and the evaluation will randomly select one map from all the maps.

2.2. Proximal Policy Optimization(PPO)

Proximal Policy Optimization comes from policy gradient algorithm.

When making Policy Gradient, an agent, a policy and an actor are required. First, the actor is to interact with the environment and collect data (sampling), then it updates the parameters of the policy according to the information collected. Once the parameters have been updated, the previously collected data is unusable anymore and new data need to be collected. Therefore, it is greatly time-consuming.

To improve Policy Gradient, off-policy was adopted. Suppose the model parameter θ can learn from the data collected by a fixed parameter θ' interacting with the environment. Then we can save time in collecting data by using fixed-parameter θ' to interact with the environment and use the collected data to train θ. As θ' is fixed, the collected data can be reused.

Recall that the goal of sampling is to determine the expectation, for a function f(x), the expectation of $f(x)$ under distribution p can be evaluated by the following formula according to Importance Sampling:

$$\mathrm{E}_{\mathrm{x}\sim \mathrm{p}}[\mathrm{f}(\mathrm{x})] \approx \frac{1}{\mathrm{N}}\sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{f}\left(\mathrm{x}^{\mathrm{i}}\right)$$

If we cannot sample from p, but can instead sample from q, then we can evaluate the expectation by:

$$\begin{aligned}\mathrm{E}_{\mathrm{x}\sim \mathrm{p}}[\mathrm{f}(\mathrm{x})] &= \int \mathrm{f}(\mathrm{x})\mathrm{p}(\mathrm{x})\mathrm{dx} = \int \mathrm{f}(\mathrm{x})\frac{\mathrm{p}(\mathrm{x})}{\mathrm{q}(\mathrm{x})}\mathrm{q}(\mathrm{x})\mathrm{dx} \\ &= \mathrm{E}_{\mathrm{x}\sim \mathrm{q}}\left[\mathrm{f}(\mathrm{x})\frac{\mathrm{p}(\mathrm{x})}{\mathrm{q}(\mathrm{x})}\right]\end{aligned}$$

Then the gradient to update becomes

$$= \mathrm{E}_{(\mathrm{st},\mathrm{at})\sim \pi_{\theta'}}\left[\frac{\mathrm{p}_\theta\left(\mathrm{a}_\mathrm{t} \mid \mathrm{s}_\mathrm{t}\right)}{\mathrm{p}_{\theta'}\left(\mathrm{a}_\mathrm{t} \mid \mathrm{s}_\mathrm{t}\right)}\mathrm{A}^{\theta'}\left(\mathrm{s}_\mathrm{t}, \mathrm{a}_\mathrm{t}\right)\nabla \log \mathrm{p}_\theta\left(\mathrm{a}_\mathrm{t}^\mathrm{n} \mid \mathrm{s}_\mathrm{t}^\mathrm{n}\right)\right]$$

by assuming that $p_\theta(s_t)$ is mostly the same as $p_\theta^{'}(s_t)$ which is usually the case.

Therefor the objective function inducted from gradient is:

$$\mathrm{J}^{\theta'}(\theta)=\mathrm{E}_{(\mathrm{st}_t,\mathrm{a_t})\sim\pi_{\theta'}}\left[\frac{\mathrm{p}_\theta\left(\mathrm{a_t}\mid\mathrm{s_t}\right)}{\mathrm{p}_{\theta'}\left(\mathrm{a_t}\mid\mathrm{s_t}\right)}\mathrm{A}^{\theta'}\left(\mathrm{s_t},\mathrm{a_t}\right)\right]$$

2.2.1. PPO1 (PPO-Penalty)

PPO1 first initialize a policy parameter θ_0, then during each iteration, use θ^k which is the parameter in the previous iteration to interact with the environment to sample. Then compute $A^{\theta^k}(s_t,a_t)$ according to the sampling and update θ several times before the next iteration. To make sure the similarity of θ^k and θ, KL divergence is added to the objective function:

$$J_{PPO}^{\theta^k}(\theta)=J^{\theta^k}(\theta)-\beta KL\left(\theta,\theta^k\right)$$

β can be adjusted dynamically by setting KL_{max} and KL_{min}. After update, if $KL\left(\theta,\theta^k\right)>KL_{max}$ increase β and decrease β if $KL\left(\theta,\theta^k\right)<KL_{min}$.

2.2.2. PPO2 (PPO-Clip)

Instead of KL divergence, PPO2 use clip to optimize objective function:

$$\mathrm{J}_{\mathrm{PPO2}}^{\theta^{\mathrm{k}}}(\theta)\approx\sum_{(\mathrm{st},\mathrm{at})}\min\left(\frac{\mathrm{p}_\theta\left(\mathrm{a_t}\mid\mathrm{s_t}\right)}{\mathrm{p}_{\theta^{\mathrm{k}}}\left(\mathrm{a_t}\mid\mathrm{s_t}\right)}\mathrm{A}^{\theta^{\mathrm{k}}}\left(\mathrm{s_t},\mathrm{a_t}\right),\right.$$
$$\left.\operatorname{clip}\left(\frac{\mathrm{p}_\theta\left(\mathrm{a_t}\mid\mathrm{s_t}\right)}{\mathrm{p}_{\theta^{\mathrm{k}}}\left(\mathrm{a_t}\mid\mathrm{s_t}\right)},1-\varepsilon,1+\varepsilon\right)\mathrm{A}^{\theta^{\mathrm{k}}}\left(\mathrm{s_t},\mathrm{a_t}\right)\right)$$

What the expression do is to make sure the model used for demonstration should not be too different from the model that is actually studied after optimization.

If A is greater than 0, then we want to increase the probability of that state-action pair. That is to say, we want to make $p_\theta(a_t|s_t)$ as large as possible. But to make sure θ^k is similar to θ, the ratio of $p_\theta(a_t|s_t)$ and $p_{\theta^k}(a_t|s_t)$ cannot exceed $1+\epsilon$.

If A is smaller than 0, $p_\theta(a_t|s_t)$ should become as small as possible on the condition that $\frac{p_\theta(a_t|s_t)}{p_\theta^k(a_t|s_t)}$ is larger than $1-\epsilon$. (Figure 1)

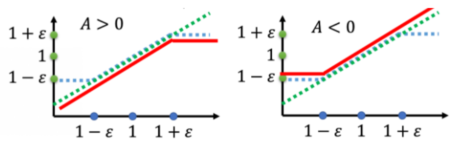


Figure 1: PPO2

2.3. Deep Deterministic Policy Gradient (DDPG)

DDPG combines value-based method and strategy-based method, it makes use of the advantages of DQN and Policy Gradient network and expands DQN into continuous space.

DDPG has two networks, one is Q network, which is used as the ”judge” to score the strategy, and the other is a strategy network. The strategy network uses the score of Q network for training, and the optimization of Q network is similar to DQN. The real reward RR and the next Q, namely Q’, are used to fit the future income Q_target. Therefore, the loss function constructed is to directly calculate the Mean Squared Error (MSE) of the two values.

However, the optimization of Q network has the same problem as DQN, that is, the Q_target is unstable. In addition, $Q_{\bar{w}}(s',a')$is not stable either, as it is just an estimation.

To stabilize the Q_target, DDPG sets up target networks for both Q network and policy network.Target_Q network is used to calculate $Q_{\bar{w}}(s',a')$ while Target_p network is used to calculate the next action a' for $Q_{\bar{w}}(s',a')$.

2.4. Twin Delayed DDPG (TD3)

Although DDPG sometimes performs well, it is often sensitive to hyperparameters and other types of adjustments. A common problem with DDPG is that the already learned Q function starts to significantly overestimate Q, which then results in the policy being broken because it exploits errors in the Q function.

TD3 addresses this problem by introducing three key techniques: Clipped Double Q-learning, “Delayed” Policy Updates and Target Policy smoothing.

Clipped Dobule Q-learning TD3 learns two q-functions Q_{ϕ_1} Q_{ϕ_2} simultaneously by minimizing the mean square error. The two q-functions share the same Q-target defined as:

$$\mathrm{y}\left(\mathrm{r},\mathrm{s}',\mathrm{d}\right)=\mathrm{r}+\gamma(1-\mathrm{d})\min_{\mathrm{i}-1,2}\mathrm{Q}_{\phi_{\mathrm{i},\mathrm{targ}}}\left(\mathrm{s}',\mathrm{a}_{\mathrm{TD3}}\left(\mathrm{s}'\right)\right)$$

“Delayed” Policy Updates TD3 algorithm updates the action network at a lower frequency and the evaluation network at a higher frequency, usually updating the evaluation network once every two times.

Target Policy smoothing TD3 introduces the idea of Smoothing. TD3 adds noise to the target action and makes it more difficult for the strategy to take advantage of the error of the Q function by smoothing the change of Q along with the action. The target policy smoothing works as follows:

$$\mathrm{a}_{\mathrm{TD3}}\left(\mathrm{s}'\right)=\operatorname{clip}\left(\mu_{\theta,\mathrm{targ}}\left(\mathrm{s}'\right)+\operatorname{clip}(\epsilon,-\mathrm{c},\mathrm{c}),\mathrm{a}_{\mathrm{low}},\mathrm{a}_{\mathrm{high}}\right)$$

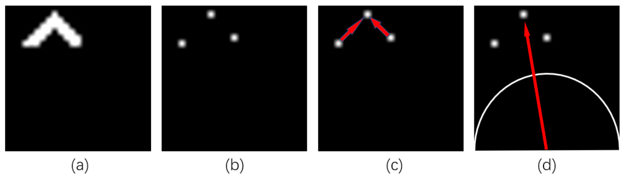


Figure 2: The process showing how we get the angle of an arrow. (a) is the original arrow in observation matrix. (b) shows three points we extracted. (c) demonstrates how to calculate the angle of an arrow according to those three points. (d) presents a fan-shaped area we use to determine which direction the agent should go.

Where $\in$ is a noise sampled from the normal distribution $\epsilon \sim \mathrm{N}(0, \sigma)$.

2.5. Soft Actor-Critic (SAC)

Soft Actor-Critic (SAC) is an algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and DDPG-style approaches. It isn't a direct successor to TD3 (having been published roughly concurrently), but 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.

3. Methodology

In this section, we will introduce four adaptations of reward. Noticing that the original reward (whether cross the finish line or not) is too simple for agents to learn the information about maps and structures. Aiming to provide our algorithm with more information about the rules of running competition, we change agents' reward according to these four cases: 1) Direction of Arrow appeared in the observation matrix can be calculated. 2) The agents keep hitting walls. 3) Finish line is just in front of the agents yet they turn around instead of dashing to the line. 4) The game ends in a tie yet the distances to the end of the two agents are different.

3.1. Calculate the Direction of Arrow

Noticing that there are many landmarks (arrows) on maps, while the original reward cannot guarantee the agents learn the semantic information of these arrows. Thus we propose a geometry-based method to calculate the direction, i.e. the angle of arrows. Then we can use the difference between the calculated angle and the angle of the agent's action to give reward to agents through a specific formula. In the following paragraphs, we will introduce how to calculate the angles according to the observation matrix.

To begin with, we need to get a mask of the arrow that appeared in the observation matrix, as shown in Fig. 2(a). There are some corner cases, for instance, there are two arrows in an observation matrix. For simplicity, we find **strong components** in this matrix and fetch only one component (each component corresponds to one arrow) to calculate the angle.

Then we need to find three angular points of an arrow, as shown in Fig. 2(b). We first find intersection points of the arrow and its **peripheral rectangle**. Then we use distance-based NMS (non-maximum-suppression) to select three (expected) angular points.

Given the assumption that the vertex of this kind of arrow is closest to its **mass center**, we can easily make sure which point is its vertex. Then we can calculate the angles of two edges of an arrow, respectively. We average these two angles and finally get the angle of an arrow, as shown in Fig. 2(c). However, there are some corner cases, for example, if the arrow is too far away from the agent, following the angle of the arrow is not a bad choice for the agent because possibly it will hit the wall. Thus we also propose a method to solve this issue. We specify a fan area (Fig. 2(d)), within which agents can follow the angle of the arrow. Outside the area, the agents just need to run towards the vertex of the arrow.

We denote the calculated angle (two cases in total) as θ_{arrow} and the angle in the agent's action space as θ_{action}. The reward can be calculated as follows:

$$R_{angle} = \alpha \cdot e^{-|\theta_{arrow} - \theta_{action}|}$$

where α represents an adjustable coefficient.

3.2. Avoid Hitting Walls

Observing that the agents are sometimes stuck in corners or keep hitting the walls, we want to punish these kinds of behaviors. However, just calculating the distance to the wall in the observation matrix is not enough, we also need to consider the agents' action direction.

Hence, we first find a ray in the observation area according to action direction. Then we can get the intersection of walls and this ray. Apparently, we can also calculate the distance between the agent and the intersection point. The process is shown in Fig. 3. We denote this distance as d_{wall}, then the

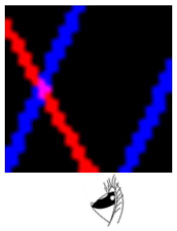


Figure 3: This figure shows how we calculate the distance between wall and agent in the direction of observation. Blue lines represent walls and red line represents a ray with angle of agent's action. The intersection point is exactly the point whose distance to the agent we need to calculate.

reward can be formulated as

$$R_{wall} = -\beta \cdot e^{-d_{wall}}$$

where β represents an adjustable coefficient.

3.3. Dash to finish line

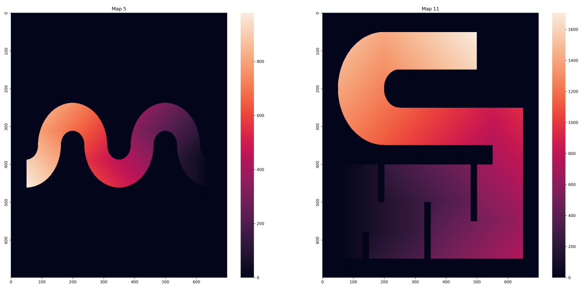
Similar to the previous subsection, we also need to consider the action direction of agents. We use exactly the same strategy to get the ray in the observation matrix. Then the intersection of the wall and this ray, and the intersection of the finish line and this ray can be obtained. After comparing the distance between these two intersection points and agent points, we can judge whether the agent needs to dash to the finish line. (We compare these distances because there are some cases where there exist walls between the finish line and agent. In this situation, our agent cannot dash to the finish line.)

If the distance between the finish line and the agent is smaller than the distance between wall and agent, we think the agent is dashing to the finish line and we will give it a positive reward, denoted as R_{dash}, a constant.

3.4. Distance to finish line

The existing rewards cannot encourage the agent to approach the finish line all the time. The direction of the arrow can guide the agent to a certain extent, but it doesn't work always. The observation of the agent usually doesn't contain any parts of an arrow, and the observed parts cannot implicate the direction. So how to let the agent know if it is close to the finish line is an important aspect that can guide the agent.

Calculating the Euclidean distance between the agent and the finish line is not correct because there may be some walls obstructing the path. So we need to calculate the minimum distance between the agent and the finish line with the walls considered. We choose **BFS(Breadth-first search)** as our algorithm. We have a map matrix in which 1 represents that



(a) map5 (b) map11

Figure 4: Distance matrix on selected maps. More results are available in Appendix A

.

this pixel is a wall and 0 represents that this pixel is not a wall. First, we start at a point of the finish line and note its distance is 0. This point is pushed into a queue. Then, every time we pop the first item in the queue and store its distance. And the 4 neighbors of the item will be checked if they are walls or if they have been visited. The neighbors which are not walls and haven't been visited will be pushed into the queue's tail with a distance of D+1, D is the distance of the current point. The process will repeat until nothing is in the queue.

There are some details in this procedure. The map matrix with the wall is reconstructed by the maps.json file. And because the finish line is a line but not a single point, we have to calculate not only the distances of one point. We calculate the distance between a position and every point in the finish line, and the final result is the minimum value. At last, we get a map matrix with every position's value representing the distance between the position and the finish line. This process only needs to happen once, so we calculate and store the matrices for the follow-up uses. The matrices are shown in Fig. 4. We denote this distance as d_{goal} and the coefficient as γ, then the reward can be formulated as:

$$R_{goal} = -\gamma \cdot d_{goal}$$

After we have the distance matrix, the reward of the agent minus the distance between the agent's current position and the finish line. It means we encourage the agent to approach the finish line. To make the penalty reasonable, we multiply a coefficient to the distance as the final penalty.

This method can guide the agent effectively. But it has a fatal defect. This penalty is given by the environment but not the observation and the action of the agent, so different maps will return different penalty even with the same observation and action. But our agent need to make policy by observation. Therefore,if the training and evaluating stages contain more than 1 map, the agent will be confused and cannot have an expected performance.

Algorithm 1 BFS for distance calculation

```
1: Initialize a queue **Q** with a start position(x,y), and distance d=0. An item in the queue is a tuple((x,y),d)
2: Initialize a result matrix **R** contains the final distance and an occupancy matrix **O** to record if the position have been visited
3: while Q is not empty do
4:     pop the first item in the queue and record the position(x,y)'s distance d into R and record (x,y) has been visited in O
5:     for $(n_x, n_y)$ in (x,y)'s neighbors do
6:         if $(n_x, n_y)$ hasn't been visited and $(n_x, n_y)$ is not wall then
7:             push($(n_x, n_y)$,d+1) into Q
8:         end if
9:     end for
10: end while
```

4. Experiments

There are 11 maps in total, the training speed is about 4h/map on SJTU A100. In order to compromise the model's generalization and training time, we may select several maps for training.The metric we use in the following experiments is the number of victories in 100 episodes.For instance, 30/70 means the two agents win 30 and 70 times respectively.The sum of the two numbers may not be 100 because both of them may don't reach the finish line.

4.1. Baseline

The performance of baseline compared with random strategy is shown in Table 1(baseline). In training, we shuffle maps and train for 1500 episodes in total. Then we evaluate the model for 100 episodes with each episode fixing a random map from all 11 maps. In other words, we evaluate the model on all maps.

Method	random / PPO
baseline	29 / 10
action space	13 / 24
maps in turn	**7 / 61**
self-play	6 / 52

Table 1: Trick Result

Then we train and evaluate the baseline on every single map. The win rate of each is shown in Table 2. It can be seen that the baseline PPO agent can not handle most of the maps. In some difficult maps, neither the random agent nor the PPO agent can pass the final, so the sum of the win rate of two agents is below 100%. The baseline framework can't generate enough useful transitions for the agent to learn because of the lack of guidance. In some simple situations, the agent can obtain some useful transitions in which it gets to the finish line by random decision. The agent can learn from these victory transitions. But if the situation is difficult, a random decision almost cannot make the agent win. Therefore, the buffer doesn't contain enough useful transitions for the agent to learn, which makes PPO get a poor result.

map	random/PPO
1	2.0 / 98.0
2	25.0 / 5.0
3	46.0 / 23.0
4	17.0 / 1.0
5	13.0 / 2.0
6	0.0 / 0.0
7	12.0 / 0.0
8	25.0 / 9.0
9	33.0 / 24.0
10	0.0 / 0.0
11	1.0 / 10.0

Table 2: Win rate of baseline in each map

4.2. DDPG & SAC

We only train and evaluate the DDPG&SAC algorithm on map1,4,10 and all, which represent the different degrees of difficulty. The result in Table 3 shows that both of them are weaker than baseline (PPO). The reason may be that we cannot adjust the parameters to proper ranges, so the agent can't explore useful transitions for itself to update the networks. And PPO is a more robust algorithm due to its low dependency on parameters.

As DDPG Does not work well, we stopped our experiment on the TD3 (for their methodology have a lot in common) and look for other implementation solutions.

maps/algo	random / DDPG	random / SAC
map1	32/5	30/14
map4	13/2	12/5
map10	0/0	0/0
shuffle	12/2	13/5

Table 3: Algorithms Result

4.3. Adding Training Tricks

4.3.1. Fine-grained action space

In the PPO algorithm of the baseline, the continuous 2-dim action space (force and angle) are divided into $6 \times 6 = 36$ discrete actions, such as $[-200, -30°], [-200, 20°] \dots [100, 30°]$. Therefore, we changed the action space into a smaller grain, the number of actions that the PPO can choose changes from 36 to 256. As the baseline, we shuffle maps in training for 1500 episodes, then we evaluate the model on all maps for 100 trials and get the following result shown in Table 1.

It can be seen that the smaller-grained action space indeed improves the performance of the PPO agent. Compared to the baseline, the PPO agent can defeat the random agent now with a smaller-grained discrete action space.

4.3.2. Train on all maps in turn

In the baseline training program, if we choose shuffle maps and set max episodes to be 1500, we may train the agent only $1500/11$ episodes in each map on average. Therefore, for better convergence in each map and the generalization in all maps, we let the agent train on each map for 3000 episodes from map 1 to map 11 continuously, which means training $3000 \times 11 = 33000$ episodes in total. We evaluate this strategy under the same setting as before, then get the following result shown in Table 1.

This learning schedule has more implications. In many maps, as mentioned above, the agent cannot explore useful transitions just by initialized networks. But map1 is simple for the agent to learn. Under these circumstances, the agent can learn some useful policy for basic purposes, for instance, it can learn to go along the wall after training in map1. And in this way, the environment is more stable compared to shuffle maps which lead to the agent being confused and hard to converge.

It can be seen that this training strategy has greatly exploited the potential of the PPO agent, which has overpassed the baseline for a large margin.

4.3.3. Self-play based training

We change the opponent from the random agent to our model (PPO) itself. Concretely, we use the current training model to choose an action for the opponent agent in each step but without gradient backward or buffer push. Then we trained the model on all maps in turn for total 33000 episodes as in the previous experiment. The result is shown in Table 1.

It can be seen that self-play based training seems to be useless when training on all maps in turn. This may be because a stronger opponent will make the agent fail more times which obstructs the way for the agent to explore.

4.3.4. Some other tricks

There are some other tricks for the training process such as take some random actions with a certain probability , setting the opponent as a lazy agent which means it doesn't take any actions and setting the opponent as different agents for weak to strong(lazy-¿random-¿self). The former may make the agent experience more valuable situations, and the latter can give the agent a more stable environment for exploration and . But limited to the lack of time and computing resources, we didn't run some tenable experiments to verify their effect.

4.4. Experiments of different rewards

As the experiments of algorithms, we choose map 1, 4, 10 and all as our settings. The results are shown in 5. In fact, the 4 rewards seem don't work. The reason we infer is that the changed rewards still cannot guide the agent to the finish line. They just restrain some bad actions of the agent, but only by this still cannot make the agent explore enough useful transitions.

maps/rewards	Angle	Wall	Dash	Dist
map1	4/95	3/97	**1/99**	**1/99**
map4	14/2	**18/14**	11/7	13/3
map10	2/0	0/0	1/0	1/0
shuffle	29/4	16/13	25/4	**18/16**

Table 4: Rewards result with original victory judgment(random/ppo), 100 episodes

On this basis, we change the judgement of win. In the original version, the agent win if it reaches the finish line before the opponent. But in the early stage, the agent cannot usually win especially on some difficult maps. It leads to the lack of useful experiences. We change it as: at the end of an episode, if the two players cannot reach the finish line either, the one whose distance to the finish line is shorter wins this episode. The distance can be obtained by the same matrix got in the distance reward. This revision enable the agent to learn from more transitions even though it seldom reaches the finish line. The agent can just learn how to approach the finish line little by little but not learn to reach the finish line directly, which is more challenging.

We can get some conclusions from this table. First,distance reward can improve the performance in single map but cannot work in shuffle maps. The reason has been stated above. Second, wall reward seems still don't work. Third, dash reward and angle reward can have a better performance. Especially on map 4, it can get a marvelous result comparing to others.

maps/rewards	Angle	Wall	Dash	Dist
map1	0/95	1/99	**0/100**	2/98
map4	**0/95**	14/0	12/62	9/24
map10	1/0	0/0	0/0	**2/22**
shuffle	**18/31**	15/13	25/31	13/14

Table 5: Rewards result with modified victory judgment,(random/ppo), 100 episodes

Still limited to the lack of time and computing sources, the experiments which combine the rewards and the training tricks haven't been run well-organized.

5. Conclusion

In this work, we first trained and evaluated the baseline PPO agent in the Competition 1v1running environment proposed by Jidi platform. Then, we replaced the PPO algorithm with other advanced algorithms, such as DDPG, TD3, and SAC. We employed these algorithms in this environment and conducted lots of experiments to evaluate their performance.

Then, we modified the reward mechanism in four ways to help the agent learn to move as far as possible toward the finish line, encouraging its exploration. We validated the effectiveness of our updated reward mechanism, compared to the sparse and unstable reward mechanism used in the baseline method.

Furthermore, we considered the original training strategy and proposed several useful methods and tricks to improve the performance and generalization of the agent when evaluated on all maps with a shuffle.

Our approaches have shown positive results in this environment according to the experiments. We hope that our agent can compete with other groups' agents to further show its capacity.

References

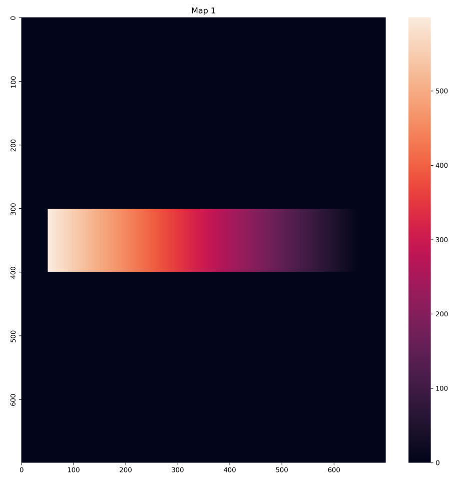
Fujimoto, S., van Hoof, H., and Meger, D. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018. URL `http://arxiv.org/abs/1802.09477`.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. 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`.

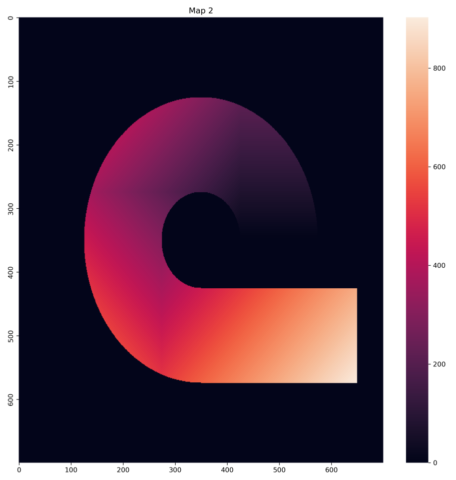
Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL `http://arxiv.org/abs/1707.06347`.

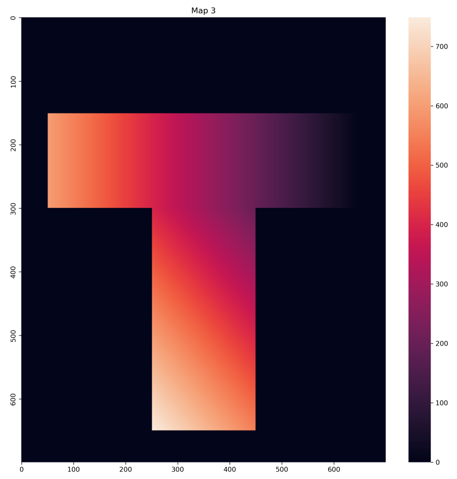
A. Distance Matrix on Different Maps



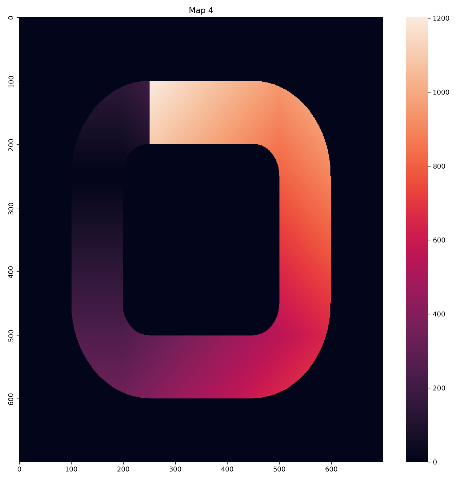
(a) Fig1



(b) Fig2

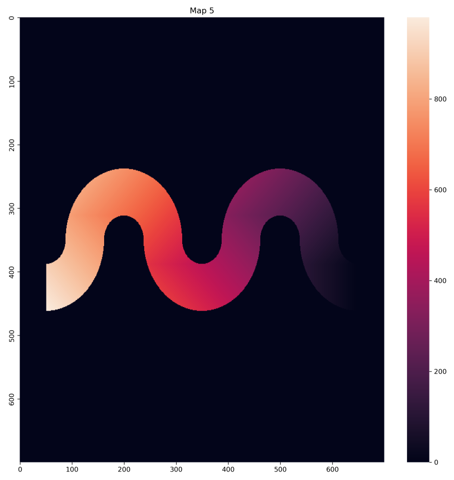


(c) Fig3

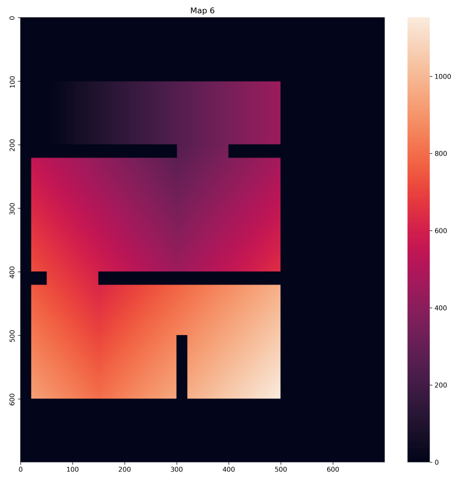


(d) Fig4

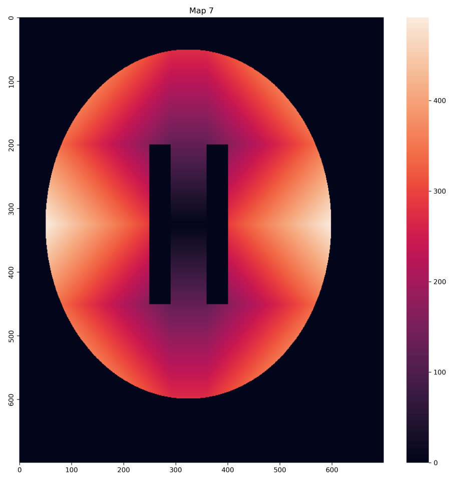
Figure 5: Distance matrix on different maps.



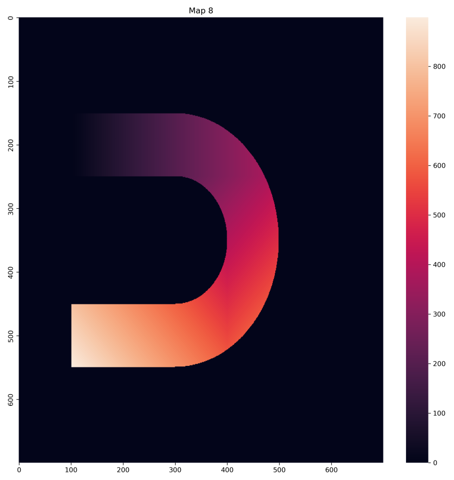
(a) Fig5



(b) Fig6

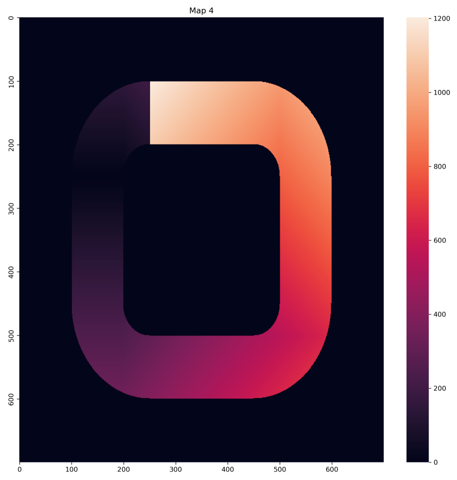


(c) Fig7

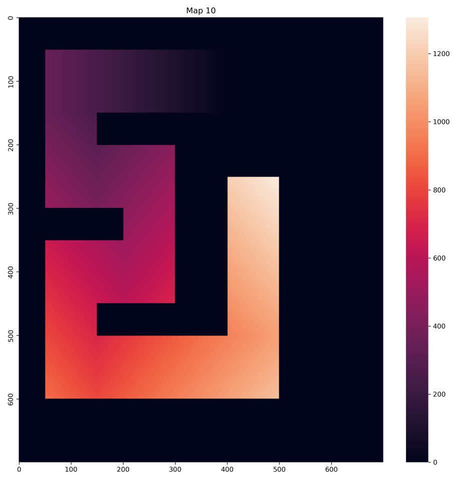


(d) Fig8

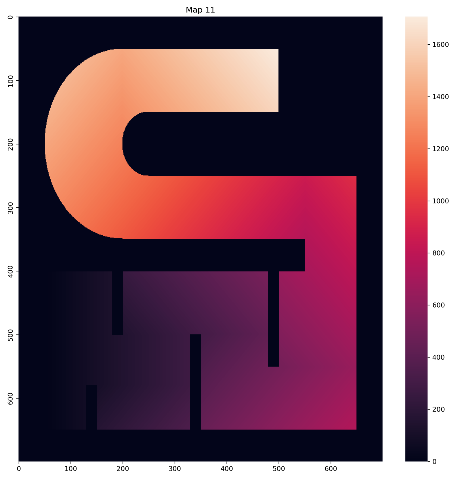
Figure 6: Distance matrix on different maps.



(a) Fig9



(b) Fig10



(c) Fig11

Figure 7: Distance matrix on different maps.