AI3617-Final-Project:A Rule-based Agent on Olympics Running

Jialin LI [*] **Yanyu Huang** [*]

Abstract

Based on the Olympics-running environment, we explored data augmentation and feature engineering for training reinforcement learning intelligences, and further obtained our Rule-based agent by map stitching reconstruction, energy constraints, judging thedirection of the arrow and searching endpoints in edges. Such a rule-based agent achieves significant advantages against random agent and RL agent.

1. Task Description

We choose Olympics-Running Competition of RLChina as our final project task. In this competiton, we need to control an agent to race against another agent with any other strategy in a given map, with the goal of reaching the finish line earlier. The specific rules are as follows:

- In given maps, both sides control an elastic ball agent with the same mass and radius.
- Agents can collide with each other and walls, but will **lose a certain speed**.
- The agent itself has energy, and the energy consumed in each step is proportional to the applied driving force and displacement. The agent's energy is recovered at a fixed rate, and if the energy decays to zero, no afterburner is possible.
- The observation of the agent is **a** 25×25 **matrix ahead**, and the observation targets include **walls**, **finish lines**, **opponents**, and **arrows** that shows the direction to the finish line.
- When an agent reaches the finish line or steps are over 500, the environment end, and if neither side crosses the line, it will be a draw.

2. Basic Task Modeling

In this task, our agent cannot observe the global map information, but can only observe the rectangular area of 25×25 ahead, so the state space can be defined as a matrix of 25×25 per frame, which is a discrete space with 625 dimensions, where the value of the relative position of the matrix represents the category of the pixel (like walls or arrows).

$$State\ Space = \begin{pmatrix} x_{1\,1} & x_{1\,2} & \dots & x_{1\,25} \\ x_{2\,1} & x_{2\,2} & \dots & x_{2\,25} \\ \vdots & \vdots & \vdots & \vdots \\ x_{25\,1} & x_{25\,2} & \dots & x_{25\,25} \end{pmatrix}$$

We can think that in each step, the control of the agent is completed by a vector representing the force, so the action space is the size and direction of the vector, which can be modeled as a two-dimensional continuous space. It should be noted here that the size and direction of the force are limited. Specifically, the size range of the force is $[-100, 200]$, and the range of the angle is $[-30°, 30°]$. That is to say, the agent can only complete a small angle of rotation and a limited acceleration and deceleration in each frame.

$$Actor\ Space = \vec{F} = (force, angel)$$

From the simplest point of view, we can directly set the reward function according to whether the agent reaches the end point.In this task, because we only have two outcomes of reaching the finish line (our agent or opponent) or not, in the end,as long as we haven't reached the finish line, the score is 0 no matter how far we are, so we set the reward function as follows($K \gg k > 0$):

$$Reward = \begin{cases} K & \text{Our agent reaches the finish line} \\ -K & \text{Opponent reaches the finish line} \\ -k & step = 500 \\ 0 & step < 500 \end{cases}$$

3. Challenges and Thoughts

3.1. Local State Observation

The first difficulty that comes to our mind is a series of problems caused by local state observation, such as the lack of information of historical sequence states and the possibility of falling into local minima. Because in each frame we

only get the information in the matrix box in front of the agent, then the agent's decision is only based on a small part of the global map. To make matters worse, due to the lack of global information, using simple modeling methods will cause the agent to not know its position and speed at all, which may cause great problems in the energy distribution of the race, such as running out of energy before game ends. Of course, we can avoid this situation by limiting force, but this will inevitably have a great impact on whether the agent can reach the finish line first. Another obvious problem is that only having the information of the current frame and ignoring the historical information may affect the understanding of the arrows in the map, which are important reference targets for determining the direction, as shown in Figure 1. A possible solution is to make a superimposed input of historical observed states. This increases the dimension size of the state space to obtain more information.

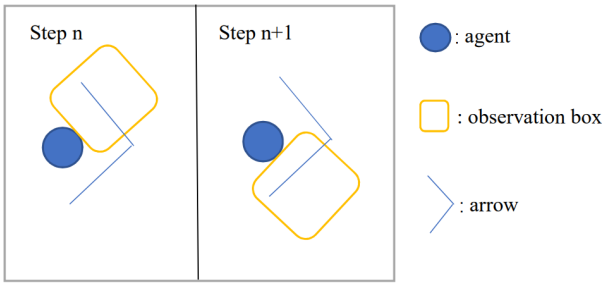

Figure 1. Part of the arrow is observed in the two frames before and after, but the direction of the arrow cannot be judged in a single frame, but if the angle of rotation is used, it is possible to reconstruct the complete arrow and get its direction

3.2. Sparse Reward

Another challenge is the sparse reward, which is also a classic challenge in RL. The sparsity of the reward function will lead to the following problems: in the process of policy update, because there is no effective signal in the middle to guide the agent to learn the right policy, relying only on randomness, the model is difficult to converge or the convergence is very slow.

In this problem, the reward criterion is to rely solely on reaching the finish line. Then we cannot effectively guide the agent to respond effectively to some specific situations in the game, such as the direction determination when an arrow is observed. In fact, there are many intermediate control signals in this task that can enrich our reward function. For example, the collision with the wall will lose speed, which is obviously not conducive to the agent to complete the game faster.

There are also arrows in the map that directly indicate the correct forward direction of the agent, which is a strong oriented control signal. If the direction of the agent is consistent with the direction of the arrow, it is more likely to reach the finish line, and vice versa.

3.3. New Reward Function

We try to define a new reward function as follows:

$$Reward^* = \begin{cases} K & \text{Our agent reaches the finish line} \\ -K & \text{Opponent reaches the finish line} \\ -k & step = 500 \\ \alpha^n P & \text{Direction matches for n times} \\ -P & \text{Direction mismatches} \\ -W & \text{Collision occurs} \\ 0 & \text{Otherwise} \end{cases}$$

In this reward function, the judgment of the collision event and the direction of the arrow is added. Since it is difficult for us to judge whether the agent collides with the wall through the change of speed, we can set a threshold for the shortest distance between the wall in the observation matrix and the edge of the agent. For example, if a wall in the observed matrix is only a pixel or two away from the agent, we judge that a collision will occur. Because it is not certain whether the collision with the opponent is beneficial to our own side, it will not be considered here.

For arrow direction matching, we can first find the offset of arrow position in adjacent frame, and then compare the offset direction with the arrow direction. If more than one arrows appear, we search for any match or all mismatches. Adding an α decay to the reward function is mainly to prevent the agent from being too aggressive in finding the arrow and continuing to follow the arrow instead of completing the game as shown in Figure 2.

But there are many difficulties in this step. One is how to determine arrow and its direction through the distribution of some arrow pixels, which is actually a traditional computer version task. Another difficulty is the previously mentioned problem that only part of the arrow can be observed. Perhaps the direction of the arrow can be reconstructed to the greatest extent through the historical observation sequence.

4. Feature Engineering

Although neither data augmentation nor internal reward design seems realistic, we expect to do some work for reinforcement training. First we try to get the current position data of the smart agent.

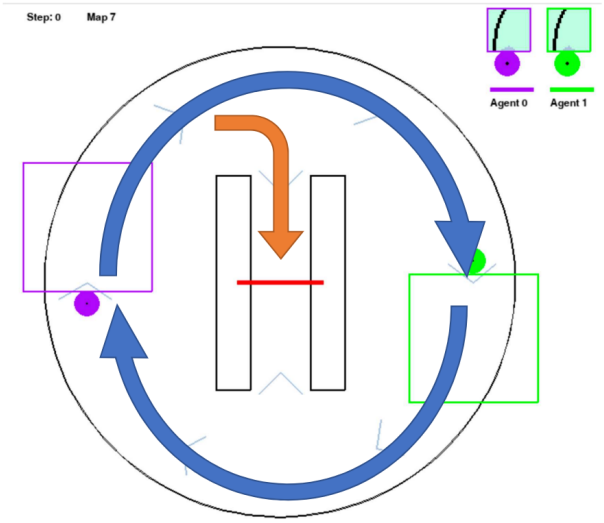

Figure 2. Take map 7 as an example. If the matching reward is not decayed, the agent may tend to take the blue route, since the number of arrows the agent has passed in 500 steps is enough to make the matching reward greater than the reward for completing the game. Setting the decay can guide the agent towards the orange route and learn the right policy.

4.1. Get Location

We observe that the position of the intelligent agent is difficult to calculate due to the presence of collisions, but we can calculate the current angle of motion by the cumulative method. We also know that the maximum angular variation per frame is 30 degrees, which means that there must be a large part of the overlapping interval between the observations of the two frames. Take figure 6 as an example.

In this case, after using rotation to eliminate the angle difference and observation position difference between the two observation maps, finding the amount of change in position becomes finding the translational phase of the two maps. Here, we choose search matching to solve the translational phase solution problem due to the sparsity of the image data coupled with the low movement speed of the intelligent agent.

Since we are matching to get the amount of position change of the two observation maps, each calculation will produce a certain error, and the final position measurement will be seriously affected by the accumulated error. To solve this problem, we use a map stitching reconstruction method, specifically, we select not only two consecutive frames of observation maps, but also multiple consecutive frames of observation maps, which are all correlated with the current observation map, and combine all these observation maps to form a "whole" map when matching.

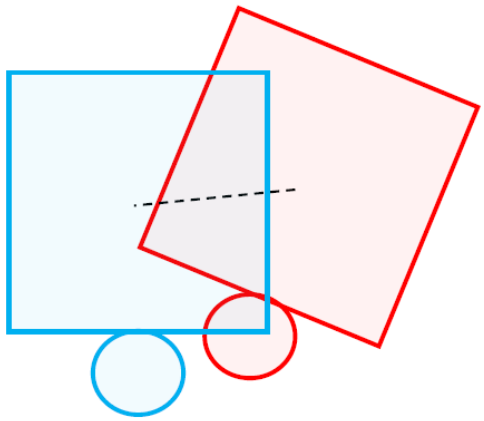

Figure 3. Schematic diagram of the overlap of the intelligent agent observation area between two frames and the distance between them

4.2. Additional Information

After the above operation, our agent now has the following information:

- Angle α (cumulative calculation with no error)
- Position (x, y) (with errors)

Due to the presence of collisions, the velocity cannot be extrapolated with force, so we can estimate the velocity from the known position information, although there will be errors. Next, we will be able to make a cumulative estimate of force and velocity, so that the intelligent body is informed of its current energy, which is, of course, the one with the largest positional error.

Theoretically, after using these methods for number augmentation, the effect of performing reinforcement learning will be much better than the original one, including internal rewards can be achieved in various ways. However, after our practical tests, in some special cases (e.g., when moving parallel to a wall), there is no difference between the two frames, which can lead to the inability of the intelligence to recognize the changing distance, a situation that can seriously affect reinforcement learning, which needs to explore all states.

This situation prompted us to think, since we have reconstructed the map, why not just plan the path for the intelligent agent? So we turned our exploration to rule-based agents.

5. Method: A Rule-based Agent

5.1. Overview

The rules of our agent are shown in Figure 4. In each step, we update the position of the agent itself and update the map information according to the newly obtained observation box of the agent. How to determine the agent's position has been explained before. The map information includes walls, arrows, and undetected areas. We did not include opponents in the step of building the map. After that, we look for whether there is an arrow in the observation box. If there is, the direction is determined according to the first arrow found. If the existing arrow information can determine the direction, we can get the forward direction of the agent. The judgment of the direction of the arrow will be introduced later.If no arrow is found, we search the wall in the observation frame, that is, the edge part, to find its endpoint. Without arrow assistance, the agent chooses to move towards the nearest endpoint. In this way, we get the direction of the agent, and the force is determined by the displacement of the agent in the previous step, that is, the speed of the agent. We need to ensure that the product of the agent's force and displacement is equal to the energy it recovers in each round to avoid running out of energy in the middle of the game.

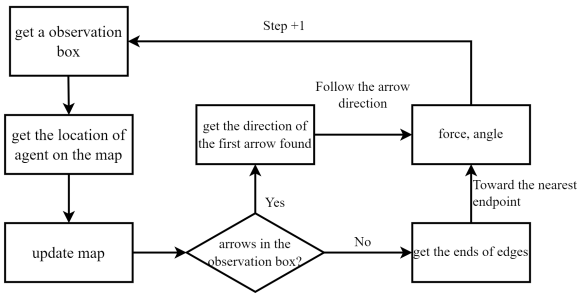

Figure 4. The general process of our algorithm. In each step, the agent updates its position and map information after acquiring a new observation box, then judges the direction according to the arrows and edges, and finally decides the force by its own speed.

5.2. Arrow Direction

There are actually many classic algorithms for judging the direction of the arrow, such as Houghline, etc. But here we use a very naive method to only judge the general direction of the arrow. The arrows in this question are two right-angled sides of an isosceles right triangle, which means that any straight line alone cannot effectively determine the direction. But as long as we have the information of the two sides, no matter whether it is complete or not, we can confirm its general direction through the change law of x, y coordinates of the pixels.

During map construction, we update the set of arrows at each step. For each arrow pixel, we traverse it in four directions to build a single arrow then save it in the arrow set. If only one straight line is known for the arrow, then the y-coordinate of its pixel point on the x-axis direction (or x-coordinate on the y-axis direction) must increase or decrease. If the maximum or minimum values appear in the middle rather than at the ends, both lines of the arrow are observed. Then we can get the general direction of the arrow (up, down, left or right) by extreme value (max or min) and the traversed rule (compare x on y-axis or compare y on x-axis).

5.3. Search Endpoints in Edges

The edge part here mainly refers to walls. The establishment of the edge map is almost the same as the establishment of the arrow set, both of which are traversed in four directions when the wall pixels are detected. With the edge map, we can easily find endpoint in the map by using a convolution kernel K.

$$K = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

If the convolution result equals to 1, the pixel refers to the endpoint. For each endpoint, we calculate its distance from the agent and choose the closest endpoint as the target to move forward. Note that if there are edges on the agent's path to the endpoint, then we go to the next closest endpoint and repeat same operation. If all endpoints are not reachable in a straight line, we will give a direction randomly.

5.4. Compute Force

After we have the angle of advance, the next thing to solve is the magnitude of the applied force. In order to keep the agent in a state of sufficient energy, the energy lost from the force applied at each step should be the same as the energy recovered by the agent at each step. Knowing the position of the agent in each round, we can find its displacement as the speed of the agent.

$$force = \frac{energy}{speed}$$

6. Experiment

6.1. About Baseline: PPO

Proximal Policy Optimisation (PPO) is a recent advancement in the field of Reinforcement Learning, which provides an improvement on Trust Region Policy Optimization (TRPO). This algorithm was proposed in 2017, and showed remarkable performance when it was implemented by OpenAI. To understand and appreciate the algorithm, we first need to understand what a policy is.

A policy, in Reinforcement Learning terminology, is a mapping from action space to state space. It can be imagined to be instructions for the RL agent, in terms of what actions it should take based upon which state of the environment it is currently in. When we talk about evaluating an agent, we generally mean evaluating the policy function to find out how well the agent is performing, following the given policy. This is where Policy Gradient methods play a vital role. When an agent is learning and doesnt really know which actions yield the best result in the corresponding states, it does so by calculating the policy gradients. It works like a neural network architecture, whereby the gradient of the output, i.e, the log of probabilities of actions in that particular state, is taken with respect to parameters of the environment and the change is reflected in the policy, based upon the gradients.

While this tried and tested method works well, the major disadvantages with these methods is their hypersensitivity to hyperparameter tuning such as choice of stepsize, learning rate, etc , along with their poor sample efficiency. Unlike supervised learning which has a guaranteed route to success or convergence with relatively less hyperparameter tuning, reinforcement learning is a lot more complex with various moving parts that need to be considered. PPO aims to strike a balance between important factors like ease of implementation, ease of tuning, sample complexity,sample efficiency and trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small. PPO is in fact, a policy gradient method that learns from online data as well. It merely ensures that the updated policy isnt too much different from the old policy to ensure low variance in training. The most common implementation of PPO is via the Actor-Critic Model which uses 2 Deep Neural Networks, one taking the action(actor) and the other handles the rewards(critic). The mathematical equation of PPO is shown below:

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

- θ is the policy parameter
- $\hat{E}_t$ denotes the empirical expectation over timesteps
- r_t is the ratio of the probability under the new and old policies, respectively
- $\hat{A}_t$ is the estimated advantage at time t
- ϵ is a hyperparameter, usually 0.1 or 0.2

The following important inferences can be drawn from the PPO equation:

- It is a policy gradient optimization algorithm, that is, in each step there is an update to an existing policy to seek improvement on certain parameters.
- It ensures that the update is not too large, that is the old policy is not too different from the new policy (it does so by essentially clipping the update region to a very narrow range).
- Advantage function is the difference between the future discounted sum of rewards on a certain state and action, and the value function of that policy.
- Importance Sampling ratio, or the ratio of the probability under the new and old policies respectively, is used for update.
- ϵ is a hyperparameter denotes the limit of the range within which the update is allowed.

Algorithm 1 is how the working PPO algorithm looks, in its entirety when implemented in Actor-Critic style:

Algorithm 1 PPO, Actor-Critic Style

for $iteration = 1, 2, \ldots$ **do**
  for $actor = 1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end for

What we can observe, is that small batches of observation are used for updation, and then thrown away in order to incorporate new a new batch of observations,aka minibatch. The updated policy will be ϵ-clipped to a small region so as to not allow huge updates which might potentially be irrecoverably harmful. In short, PPO behaves exactly like other policy gradient methods in the sense that it also involves the calculation of output probabilities in the forward pass based on various parameters and calculating the gradients to improve those decisions or probabilities in the backward pass. It involves the usage of importance sampling

ration like its predecessor, TRPO. However, it also ensures that the old policy and new policy are at least at a certain proximity (denoted by ϵ), and very large updates are not allowed. It has become one of the most widely used policy optimization algorithms in the field of reinforcement learning.

6.2. Compare With Baseline by Random Agent

In order to test the performance of our smart agent, we conducted the following experiments. First, let Rule-Based Agent and RL Agent play a total of 500 episodes against Random Agent on each of the 11 maps and count the average number of steps and the winning rate, the results obtained are shown in Figure 5 below.

Where the model parameters loaded by RL Agent are the pre-trained baseline model(PPO model).

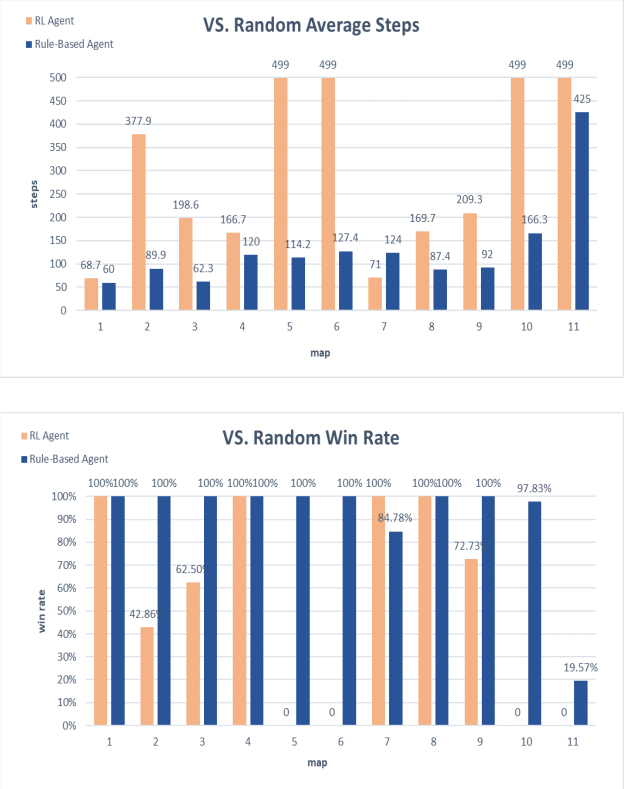

Figure 5. Results of Confrontation with Random Agent

6.3. Compare With Baseline in Racing Environment

At the same time, we also set up a comparative experiment in the same environment. We set 50 epoches on each map and get the result. We can see that our rule-based agent performs much better than the baseline agent on all maps.

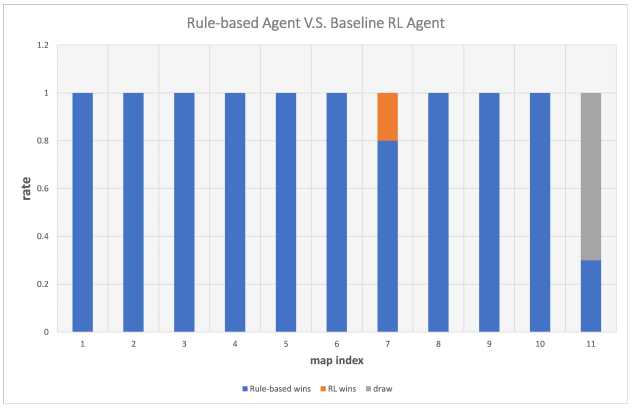

Figure 6. Result of rule-based agent v.s. Baseline RL agent

7. Member Contribution

Because our team only have two members, both of us are involved in every step of the project such as coding, running experiments, writing papers. So the contribution is basically the same.

Acknowledgements

Thanks for the course for providing a great project to work further on game theory. In the process of completing the project, we reviewed a lot of reinforcement learning content, even some key points on computer vision.

Thanks for the algorithms sharing of the 1st and 2nd place team in this competition. Their rule-based algorithms has given us great inspiration in our work.