A. Experiments on More Minecraft Tasks

A.1. Environment Setup

This section conducts experiments on more Minecraft tasks to verify the effectiveness of the proposed reward function design mechanism. The experiments mainly includes the approaching task of three animals including pig , sheep and chicken , as well as the attacking task of cow .

- **Approaching animals in plains biome.** The task is considered successful if the animal is within the agent's visible range and the distance is less than 2 distance unit. The difficulty of the task is that the animals are constantly moving and small (*e.g.* chicken). The maximum number of steps is limited to 2,000 steps.
- **Attacking cow in plains biome.** The task is considered successful if the agent successfully kills a cow and obtains beef or leather . The maximum number of steps is limited to 4,000 steps.

A.2. Results

Table I compares the performance between imitation learning model and ours. As the task of approaching for animals does not appear in the training data of imitation learning, our method achieves a significant improvement in success rate in fewer steps compared to imitation learning model's zero shot. In the more difficult task of attacking cow, our method also achieves clear advantages. By further checking the videos, we find most of the successful cases are due to good luck without intention to actively approach the target. In the supplementary materials, we have prepared several demo videos, in which we can clearly see that the agent trained by our method has a clear awareness of completing the corresponding tasks.

Table I. Compare imitation learning with our method on four Minecraft tasks.

Method	**Approach Pig on Plains**		**Approach Sheep on Plains**		**Approach Chicken on Plains**		**Attack Cow on Plains**	
	avg. step ↓	succ. (%) ↑	avg. step ↓	succ. (%) ↑	avg. step ↓	succ. (%) ↑	avg. step ↓	succ. (%) ↑
Imitation Learning	1664	25.0	1734	23.5	1496	36.4	3522	13.3
Ours	**830**	**67.3**	**1024**	**60.6**	**1391**	**54.5**	**3133**	**40.8**

B. Experiments of scalability on MiniGrid Environment

To verify the scalability and effectiveness of Auto MC-reward when applied on new environments other than Minecraft, we conduct preliminary experiments on the cross-lava task of the MiniGrid environment [8], where the agent is required to reach the goal point on the other corner of the room while avoiding rivers of deadly lava. We choose two environment settings of different difficulty levels: `LavaCrossingS9N1` with map size 9×9 and 1 lava stream, and `LavaCrossingS9N3` with map size 9×9 and 3 lava streams. We modify the prompts of game information, task description, and input parameters to adapt to the MiniGrid environment, but other parts (*e.g.*, reward function requirements and format) remain unchanged. The example environment visualization and results are shown in Figure I. Auto MC-Reward shows its superiority to original sparse reward, indicating its effectiveness in new environments.

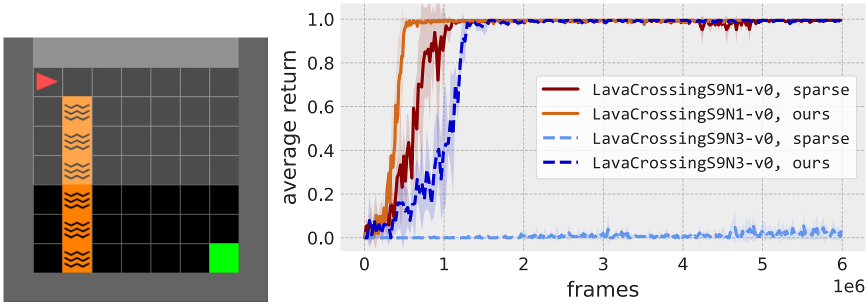

Figure I. (Left) Example environment visualization of MiniGrid `LavaCrossingS9N1` environment. (Right) Performance comparison on MiniGrid.

C. Experiments with Different LLMs

Our framework leverages different LLM capabilities: code comprehension and generation for Designer and Critic, and JSON-like data interpretation for Analyzer. In Fig. II (left), we investigate different LLMs. We use the case of avoiding lava from Sec. H to test whether each LLM can analyze that lava is the failure reason, and whether it can add the penalty for lava into the reward function. Notably, small LLMs like Mistral Medium show better performance than GPT-4. We hope that future work could provide more comprehensive evaluation to guide the selection of LLMs.

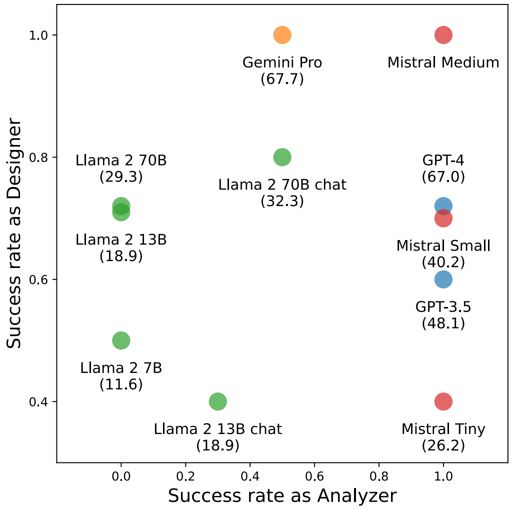

Figure II. Effect of different LLMs. Numbers in the parentheses are benchmark scores of HumanEval on code generation.

D. Foundation Model Imitation Learning

D.1. Pretraining Data

When large labeled datasets do not exist, the canonical strategy for training capable agents is RL, which is inefficient and expensive to sample for hard-exploration problems [3, 4, 21], *e.g.* mining diamond in Minecraft. Therefore, in order to more efficiently explore the effectiveness of the LLM-based reward function design mechanism proposed in this paper, we pre-trained some foundation models through imitation learning as done by VPT [3]. Specifically, we use GITM [53] to continuously perform Diamond Mining task and record important observation data of each frame, such as RGB, action, inventory, compass (*e.g.* y_level), structured action (action plans), etc. The following is the format of the collected data:

```
{
  "action_plans": [
    {
      "name": "mine",
      "args": {
        "object": {"log": 1},
        "tool": None
      },
      "expectation": "Obtain 1 log"
    },
    ...
  ],

  "actions": [
    [0, 0, 0, 12, 12, 3, 0, 0], ...
  ],

  "y_level": [66, ...],
```

```
    "action_indices": [100, ...],

    "biomes": ["Forest", ...],

    "deaths": [0, ...],

    "visible_blocks": [
      [
        {"name": "wood", "nearest_distance": 0.71}, ...
      ],
      ...
    ],

    "visible_entities": [
      [
        {"name": "cow", "nearest_distance": 4.57}, ...
      ],
      ...
    ],

    "inventory": [{"log": 1}, ...],

    "task_names": ["Obtain 1 log", ...],

    "task_indices": [88, ...],
}
```

Finally, we collect approximately 20.5/5.3 million image data in forest/plains biome, as listed in Table II. To reduce training costs, we only use 9 million and 2 million image data for training and validation in forest and plains biomes, respectively, totaling about 153 hours of game video (the control frequency is 20 Hz).

Table II. Detailed statistics of generated trajectories.

Item	**Forest Biome**				**Plains Biome**			
	trajectories	frames	max frames / traj.	min frames / traj.	trajectories	frames	max frames / traj.	min frames / traj.
log	14,790	5,209,246	13,806	8	3,254	2,425,602	14,373	10
planks	16,452	16,452	1	1	3,647	3,647	1	1
stick	7,198	9,076	2	1	1,550	1,908	2	1
crafting table	3,452	3,452	1	1	837	837	1	1
wooden pickaxe	2,972	408,056	2,263	72	685	97,529	2,328	106
cobblestone	3,047	1,704,098	13,107	180	681	341,737	6,681	184
stone pickaxe	2,729	414,750	4,473	57	607	86,806	2,486	108
iron ore	2,177	6,498,252	14,833	200	461	1,394,251	12,783	235
furnace	1,879	280,604	7,912	86	362	55,101	2,885	28
iron ingot	1,632	497,513	7,334	17	281	91,585	4,916	254
iron pickaxe	1,588	189,718	877	38	274	32,450	656	106
diamond	756	5,285,204	14,989	1,154	119	766,785	14,634	1,226
total	58,672	20,516,421	-	-	12,758	5,298,238	-	-

D.2. Action Space

We select 14 actions to train imitation learning for four tasks (obtain log/cobblestone/iron ore/diamond), as shown in Table III.

D.3. Training Details

Subsequently, we conduct fully supervised training on the four main tasks of the Diamond mining Task, *i.e.* log, cobblestone, iron ore, diamond, by using Impala CNN [13] and Transformer [44] as backbone, and obtained several foundation models. The main differences between the foundation models are different biomes (forest and plains), temporal frames (16 and 128), and whether goal and y_level embeddings are used as conditions. We use 32 A800 GPUs for foundation model training, and

Table III. Action space used in our method.

Index	Action Name	Action Box	Index	Action Name	Action Box
0	no_op	[0, 0, 0, 12, 12, 0, 0, 0]	7	jump	[0, 0, 1, 12, 12, 0, 0, 0]
1	turn_up	[0, 0, 0, 11, 12, 0, 0, 0]	8	back	[2, 0, 0, 12, 12, 0, 0, 0]
2	turn_down	[0, 0, 0, 13, 12, 0, 0, 0]	9	move_left	[0, 1, 0, 12, 12, 0, 0, 0]
3	turn_left	[0, 0, 0, 12, 11, 0, 0, 0]	10	move_right	[0, 2, 0, 12, 12, 0, 0, 0]
4	turn_right	[0, 0, 0, 12, 13, 0, 0, 0]	11	attack	[0, 0, 0, 12, 12, 3, 0, 0]
5	forward	[1, 0, 0, 12, 12, 0, 0, 0]	12	equip	[0, 0, 0, 12, 12, 5, 0, 0]
6	forward_jump	[1, 0, 1, 12, 12, 0, 0, 0]	13	jump_place	[0, 0, 1, 12, 12, 6, 0, 0]

initialize the network using the foundation model of VPT to speed up the training of imitation learning. The hyperparameters are listed in Table IV.

Table IV. Hyperparameters for imitation learning.

Hyperparameter	Value
Learning rate	1e-4
Batch size	128
Epochs	25
Optimizer	Adam

The obtained foundation models, especially the goal-conditioned one, already have satisfactory basic behavioral capabilities. Based on these models, this paper uses the proposed methods to further improve existing behaviors (*e.g.* avoid lava while exploring diamond) and complete new tasks (*e.g.* approaching / killing animals).

E. Reinforcement Learning Training

We use proximal policy optimization (PPO) algorithm [39] with generalized advantage estimation (GAE) [38] implemented in TorchRL [5] library to train our RL model. The main hyperparameters used for RL training are listed in Table V.

Table V. Hyperparameters for reinforcement learning.

Hyperparameter	**Value**
Learning rate	5e-5
Total training frames	256,000
Batches per iteration	64
Discount factor (γ)	0.99
GAE λ	0.95
Value pre-training iteration	2
PPO clip	0.1
Entropy loss weight	0.1
Critic loss type	L_2
Critic coefficient	1
Optimizer	Adam

In addition to the above hyperparameters, several training techniques are applied to all of our experiments:

- We follow VPT [3] to apply an auxiliary Kullback-Leibler (KL) divergence loss between the RL model and the frozen pre-trained policy to prevent catastrophically forgetting or overly aggressive policy update during RL training, *e.g.*, maintaining the ability to dig horizontal tunnels. This loss is defined as:

  $$L_{kl} = \alpha \mathrm{KL}(\pi_{pt}, \pi_\theta) \tag{1}$$

  where π_θ is the the policy being trained, π_{pt} is the frozen pretrained policy, $\mathrm{KL}(\pi_{pt}, \pi_\theta)$is the Kullback-Leibler divergence between the policy being trained and the pretrained policy, and α is the loss weight that is set to 0.125 by default.
- We also normalize the reward based on the trajectory returns to constrain the return scales of different tasks. The formula is as follows:

  $$\begin{aligned} R' &= \mathrm{mean}(\mathrm{abs}(R)) \\ R'_{cur} &= \beta R'_{cur} + (1-\beta) R'_{pre} \\ r_{norm} &= \frac{r}{\max(R'_{cur}, 1)} \end{aligned} \tag{2}$$

  where r and R are reward and trajectory returns. β is a momentum parameter, set as 0.7 by default.

F. Full Text Prompts

We provide all the LLM input prompts of our method in this section.

F.1. Task Description

The task description contains task objective, initial status, success criteria and procedure of each task. It describe basic information of a task without prior knowledge like game strategies.

Exploring Diamond Ore

Task description

- Objective: Find and approach diamond with higher success rate and avoid death.

- Initial Status:
  1. The agent is under ground at y level 11, the most common level to find diamonds.
  2. The agent already has an iron pickaxe.
- Success criteria: The distance to the nearest diamond block is less than 2 meters.

- Procedure: Explore horizontally to find a diamond, face it and approach it.

Approaching a Tree on Plains

Task description

- Objective: Find and approach wood in the plains biome.

- Initial Status: The agent is on the ground.

- Success Criteria: The distance to the nearest wood block is less than 1 meters.

- Procedure: Find a wood, face it and approach it.

- Additional Information: Wood is sparse in plains biome. The agent may need to walk a long distance to find and approach a wood.

Approaching a Cow on Plains

Task description

- Objective: Find and approach cow in the plains biome.

- Initial Status: The agent is on the ground.

- Success Criteria: The distance to the nearest cow is less than 2 meters.

- Procedure: Find a cow, face it and approach it.

Attacking a Cow on Plains

Task description

- Objective: Find, approach and kill cow in the plains biome.

- Initial Status: The agent is on the ground.

- Success Criteria: The agent successfully kills a cow.

- Procedure: Find a cow, face it, approach it and kill it.

F.2. Reward Designer

Prompts for Reward Designer include prompts for initializing reward function, updating reward function, and handling critic failure and execution error.

Prompt for Initializing Reward Function

You are now a proficient Minecraft player. You should help me write proper reward functions to train a Minecraft agent with reinforcement learning to complete the described task.

`{task_description}`

Input parameters of the reward function

- current_nearest_blocks: nearby block types and nearest distance of each block type at the **current step**. It is a dictionary, consisting of the names, relative distances with the agent, and relative angles (yaw and pitch) with the agent, in the form of: {NAME_1: (DIST_1, YAW_1, PITCH_1), NAME_2: (DIST_2, YAW_2, PITCH_2)}. For example, {"wood": (24.7, 1.48, -1.57), "cobblestone": (1.3, -0.17, 1.05), "iron_ore": (4.5, 0.61, -0.17)}. If a certrain block type does not exist in the field of view, it is not present in the dictionary. The yaw and pitch here is relative to the agent's front, i.e., pitch = 0 and yaw = 0 is the front of the agent.

- previou_nearest_blocks: nearby block types and nearest distance of each block type at the **previous step**, with the same format as the current_nearest_blocks.

- inventory_change: the change of the agent's inventory from the previous step to current step, in the form of a dictionary: {NAME_1: CHANGE_NUM_1, NAME_2: CHANGE_NUM_2}. Positive values mean increase and negative values mean decrease. For example, {"wood": 2, "dirt": 3, "stone_pickaxe": -1}.

- health: an integer value in range 0 to 10 indicating the health level of the agent. 0 means death and 10 means full health.

- past_agent_positions: the history of location of agent, in the form of a list: [[x1, y1, z1, yaw1, pitch1], [x2, y2, z2, yaw2, pitch2], ...]. The yaw and pitch here are relative to the agent's initial forward direction, i.e., pitch = 0 and yaw = 0 is the front of the agent when it was born. The length of the list is the number of steps the agent has taken. The last element of the list is the current location of the agent.

- GLOBAL_DATA: a global variable. It is initialized as a dictionary. You can save necessary information between different steps with it.

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Reward function requirements

- You should write a dense reward function `dense` and a sparse reward function `sparse`. The sparse reward indicates achieving the goal or receiving heavy punishment. The dense reward provides intermediate signal to guide the agent in the process of achieving the goal. The magnitude of the return value does not matter, but the sign (positive or negative) is important. The final reward will be `np.sign(sparse(...)) * 1 + np.sign(dense(...)) * 0.1`.

Output Requirements

- The reward function should be written in Python 3.9.

- Output the code block only. **Do not output anything else outside the code block**.

- You should include **sufficient comments** in your reward function to explain your thoughts, the objective and **implementation details**. The implementation can be specified to a specific line of code.

- If you need to import packages (e.g. math, numpy) or define helper functions, define them at the beginning of the function. Do not use unimported packages and undefined functions.

Output format

Strictly follow the following format. **Do not output anything else outside the code block**.

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    # Thoughts:
    # ...
    # (import packages and define helper functions)
    import numpy as np
    ...
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
    ...
```

Now write a reward function. Then in each iteration, I will use the reward function to train an RL agent, and test it in the environment. I will give you possible reasons of the failure found during the testing, and you should modify the reward function accordingly.

Prompt for Updating Reward Function

You are now a proficient Minecraft player. I now have a reward function to train a Minecraft agent with reinforcement learning to complete the described task. The reward function is used to train the reinforcement learning agent for several times. I will provide you the analysis of failure and inefficiency and suggestions. You should help me modify the reward function.

`{task_description}`

Input parameters of the reward function

- current_nearest_blocks: nearby block types and nearest distance of each block type at the **current step**. It is a dictionary, consisting of the names, relative distances with the agent, and relative angles (yaw and pitch) with the agent, in the form of: {NAME_1: (DIST_1, YAW_1, PITCH_1), NAME_2: (DIST_2, YAW_2, PITCH_2)}. For example, {"wood": (24.7, 1.48, -1.57), "cobblestone": (1.3, -0.17, 1.05), "iron_ore": (4.5, 0.61, -0.17)}. If a certrain block type does not exist in the field of view, it is not present in the dictionary. The yaw and pitch here is relative to the agent's front, i.e., pitch = 0 and yaw = 0 is the front of the agent.

- previou_nearest_blocks: nearby block types and nearest distance of each block type at the **previous step**, with the same format as the current_nearest_blocks.

- inventory_change: the change of the agent's inventory from the previous step to current step, in the form of a dictionary: {NAME_1: CHANGE_NUM_1, NAME_2: CHANGE_NUM_2}. Positive values mean increase and negative values mean decrease. For example, {"wood": 2, "dirt": 3, "stone_pickaxe": -1}.

- health: an integer value in range 0 to 10 indicating the health level of the agent. 0 means death and 10 means full health.

- past_agent_positions: the history of location of agent, in the form of a list: [[x1, y1, z1, yaw1, pitch1], [x2, y2, z2, yaw2, pitch2], ...]. The yaw and pitch here are relative to the agent's initial forward direction, i.e., pitch = 0 and yaw = 0 is the front of the agent when it was born. The length of the list is the number of steps the agent has taken. The last element of the list is the current location of the agent.

- GLOBAL_DATA: a global variable. It is initialized as a dictionary. You can save necessary information between different steps with it.

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Reward function requirements

- You should write a dense reward function `dense` and a sparse reward function `sparse`. The sparse reward indicates achieving the goal or receiving heavy punishment. The dense reward provides intermediate signal to guide the agent in the process of achieving the goal. The magnitude of the return value does not matter, but the sign (positive or negative) is important. The final reward will be `np.sign(sparse(...)) * 1 + np.sign(dense(...)) * 0.1`.

Current reward function

`{current_reward_function}`

Reward function test result

The reward function is used to train the reinforcement learning agent for several times. Here is some analysis of failure and inefficiency and suggestions:

```
{analysis}
```

Requirements

Please consider the analysis and suggestions above and modify the reward function.

1. You can both modify the current lines and add new lines.

2. If necessary, you can write a **totally different** reward function than the current one.

3. Consider modifing the reward and penalty values in the current reward function to balance them.

4. In the first part of the reward function, you should provide your thoughts of modifying the reward function. **The thoughts should be concise.**

Output format

Strictly follow the following format. **Do not output anything else outside the code block**.

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    # Thoughts:
    # ...
    # (import packages and define helper functions)
    import numpy as np
    ...
```

```
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
    ...
```

Prompt for Critic Failure

The reward function fails to pass the verification. The reasoning is:

```
{reasoning}
```

The critique is:

```
{critique}
```

Requirements

Please consider the reasoning and critique, and modify the reward function.

1. If necessary, you can write a totally different reward function than the current one.

2. In the first part of the reward function, you should provide your thoughts of modifying the reward function. The thoughts should not directly copy the given reasoning or critique. **The thoughts should be concise.**

Reward function requirements

- You should write a dense reward function `dense` and a sparse reward function `sparse`. The sparse reward indicates achieving the goal or receiving heavy punishment. The dense reward provides intermediate signal to guide the agent in the process of achieving the goal. The magnitude of the return value does not matter, but the sign (positive or negative) is important. The final reward will be `np.sign(sparse(...)) * 1 + np.sign(dense(...)) * 0.1`.

Output format

Strictly follow the following format. **Do not output anything else outside the code block**.

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    # Thoughts:
    # ...
    # (import packages and define helper functions)
    import numpy as np
    ...
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
```

```
GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA)
    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
    ...
```

Prompt for Code Execution Error

You are now a proficient Minecraft player. I now have a reward function to train a Minecraft agent with reinforcement learning to complete the described task. The reward function now encounters an execution error. I will provide you the execution error and the reward function. You should help me modify the reward function.

`{task_description}`

Input parameters of the reward function

- current_nearest_blocks: nearby block types and nearest distance of each block type at the **current step**. It is a dictionary, consisting of the names, relative distances with the agent, and relative angles (yaw and pitch) with the agent, in the form of: {NAME_1: (DIST_1, YAW_1, PITCH_1), NAME_2: (DIST_2, YAW_2, PITCH_2)}. For example, {"wood": (24.7, 1.48, -1.57), "cobblestone": (1.3, -0.17, 1.05), "iron_ore": (4.5, 0.61, -0.17)}. If a certrain block type does not exist in the field of view, it is not present in the dictionary. The yaw and pitch here is relative to the agent's front, i.e., pitch = 0 and yaw = 0 is the front of the agent.

- previou_nearest_blocks: nearby block types and nearest distance of each block type at the **previous step**, with the same format as the current_nearest_blocks.

- inventory_change: the change of the agent's inventory from the previous step to current step, in the form of a dictionary: {NAME_1: CHANGE_NUM_1, NAME_2: CHANGE_NUM_2}. Positive values mean increase and negative values mean decrease. For example, {"wood": 2, "dirt": 3, "stone_pickaxe": -1}.

- health: an integer value in range 0 to 10 indicating the health level of the agent. 0 means death and 10 means full health.

- past_agent_positions: the history of location of agent, in the form of a list: [[x1, y1, z1, yaw1, pitch1], [x2, y2, z2, yaw2, pitch2], ...]. The yaw and pitch here are relative to the agent's initial forward direction, i.e., pitch = 0 and yaw = 0 is the front of the agent when it was born. The length of the list is the number of steps the agent has taken. The last element of the list is the current location of the agent.

- GLOBAL_DATA: a global variable. It is initialized as a dictionary. You can save necessary information between different steps with it.

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Reward function requirements

- You should write a dense reward function `dense` and a sparse reward function `sparse`. The sparse reward indicates achieving the goal or receiving heavy punishment. The dense reward provides intermediate signal to guide the agent in the process of achieving the goal. The magnitude of the return value does not matter, but the sign (positive or negative) is important. The final reward will be `np.sign(sparse(...)) * 1 + np.sign(dense(...)) * 0.1`.

Current reward function

`{current_reward_function}`

Execution error traceback

```
{execution_error}
```

Requirements

Please consider the reason of error and modify the reward function.

In the first part of the reward function, you should write your analysis of the error in comments, and provide your thoughts of modifying the reward function. **The analysis and thoughts should be concise.**

Output format

Strictly follow the following format. **Do not output anything else outside the code block**.

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA):
  # Thoughts:
  # ...
  # (import packages and define helper functions)
  import numpy as np
  ...
  def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA):
    ...
  def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA):
    ...
  dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA)
  sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA)
  return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
  ...
```

F.3. Reward Critic

The prompt for Reward Critic judges whether the current reward function meets the requirements, and provides critique when the judgement fails.

You are now a proficient Minecraft player. I will give you a reward function written in Python, which is used to train an RL agent to play Minecraft.

You should help me to evaluate whether the reward function has met the requirements.

`{task_description}`

Input parameters of the reward function

- current_nearest_blocks: nearby block types and nearest distance of each block type at the **current step**. It is a dictionary, consisting of the names, relative distances with the agent, and relative angles (yaw and pitch) with the agent, in the form of: {NAME_1: (DIST_1, YAW_1, PITCH_1), NAME_2: (DIST_2, YAW_2, PITCH_2)}. For example, {"wood": (24.7, 1.48, -1.57), "cobblestone": (1.3, -0.17, 1.05), "iron_ore": (4.5, 0.61, -0.17)}. If a certrain block type does not exist in the field of view, it is not present in the dictionary. The yaw and pitch here is relative to the agent's front, i.e., pitch = 0 and yaw = 0 is the front of the agent.

- previou_nearest_blocks: nearby block types and nearest distance of each block type at the **previous step**, with the same format as the current_nearest_blocks.

- inventory_change: the change of the agent's inventory from the previous step to current step, in the form of a dictionary: {NAME_1: CHANGE_NUM_1, NAME_2: CHANGE_NUM_2}. Positive values mean increase and negative values mean decrease. For example, {"wood": 2, "dirt": 3, "stone_pickaxe": -1}.

- health: an integer value in range 0 to 10 indicating the health level of the agent. 0 means death and 10 means full health.

- past_agent_positions: the history of location of agent, in the form of a list: [[x1, y1, z1, yaw1, pitch1], [x2, y2, z2, yaw2, pitch2], ...]. The yaw and pitch here are relative to the agent's initial forward direction, i.e., pitch = 0 and yaw = 0 is the front of the agent when it was born. The length of the list is the number of steps the agent has taken. The last element of the list is the current location of the agent.

- GLOBAL_DATA: a global variable. It is initialized as a dictionary. You can save necessary information between different steps with it.

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Reward function requirements

- You should write a dense reward function `dense` and a sparse reward function `sparse`. The sparse reward indicates achieving the goal or receiving heavy punishment. The dense reward provides intermediate signal to guide the agent in the process of achieving the goal. The magnitude of the return value does not matter, but the sign (positive or negative) is important. The final reward will be `np.sign(sparse(...)) * 1 + np.sign(dense(...)) * 0.1`.

Requirements

1. You should check whether the reward function meets the **reward function requirements** above.

2. Your judgement should consider whether the comments and the detailed implementation code are consistent. You can judge whether a line is correct by its comment, but do only rely on the comments.

3. You should also check the format of the reward function. It should be like:

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    # Thoughts:
    # ...
    # (import packages and define helper functions)
    import numpy as np
    ...
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
    ...
```

4. Consider the **Information about the game state** above.

5. **You don't need to check whether the reward function meets the task description**.

6. You should first provide a reasoning and decision. If the reward function fails, provide a critique to help to improve the reward function.

Output format

You should only respond in JSON format as described below:

```
{
    "reasoning": "reasoning",
    "success": boolean,
    "critique": critique,
}
```

Ensure the response can be parsed by Python `json.loads`, e.g. no trailing commas, no single quotes, etc. Do not output anything else like explanations.

Reward function to be evaluated

`{reward_function}`

F.4. Trajectory Analyzer

The prompt for Trajectory Analyzer analyzes the trajectory information and statistics, and provides possible reasons of failure and suggestions on game strategy.

You are now a proficient Minecraft player. I have trained an RL agent and tested for several times in the Minecraft game environment.

I will give you the information of the failed test results, i.e. trajectories of actions, rewards and observations. You should help me write proper analysis of possible reasons of failure and inefficiency, and your suggestion on the game strategy.

`{task_description}`

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Input format

Failed trajectories

The failed trajectories are shown as a list, where each entry is a dictionary representing a trajectory. Note that **only the failed cases are provided.**

The each item of "history" is a list representing statistics of each single step. When the history is too long, it will be truncated to the last 32 steps, indicated by the value of "truncated".

"dead" indicates whether the agent has died.

The format is:

```
[
  {
    "history": {
      "rewards": [reward1, ...],
      "actions": [action1, action2, ...],
      "locations": [[x1, y1, z1, pitch1, yaw1], ...],
      "inventory_change": {item1: delta_num1, ...},
      "truncated": True or False,
    },
    "final_health": health,
    "final_inventory": {item1: num1, ...},
    "final_nearest_blocks": {block_name1: nearest_distance1, block_name2: nearest_distance2, ...},
    "block_under_foot": block_name,
    "dead": True or False,
  },
  ...
]
```

where "final_health" is in range 0 and 10. "block_under_foot" is the block right under the agent.

Statistics

The statistics is a dictionary of all the test results, including **both successful and failed trajectories**.

The format is:

```
{
  "success_rate" : success_rate,
}
```

Requirements

Please write proper analysis of possible reasons of failure and inefficiency, and your suggestion on the game strategy. You should not not be limited to the task description above, but also come up with other failure cases based on the test results.

The analysis and suggestion should be concise.

Test results

Failed trajectories

`{trajectories}`

Statistics

`{statistics}`

We also try to replace game-specific feedback information (i.e. trajectory information and statistics) with captions from multi-modal large language model (e.g. GPT-4V) to enhance the generality of the agent. First, we input the video sequence into GPT-4V and let it describe the sequence in as much detail as possible. The prompt is as follows:

This is a Minecraft game video clip. Please describe the scene and the player's actions in detail. Note that describe the entire sequence of images rather than describing each image individually.

Then, the output of GPT-4V will be further combined with the prompt of the above trajectory analysis for another question and answer, allowing LLM to analyze the possible reasons for the failure. See Section I for detailed examples.

G. Examples of LLM-Designed Reward Functions

We provide reward function examples for four tasks. For the attacking cow task, we add two extra input parameters *damage_dealt* and *kill_entity*, which records the amount of damage the agent caused and the amounts of each entity the agent killed, respectively.

Algorithm I Reward Function Example of Exploring Diamond Ore

```python
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions
     , GLOBAL_DATA):
    """
    Thoughts:
    In order to make the agent approach the diamond, we need to give positive rewards for actions that
    make the agent get closer to the diamond and negative rewards for actions that make the agent get further from
     the diamond.
    We also need to give a negative reward if the agent's health decreases to encourage the agent to avoid
     dangerous situations.
    The dense reward function will be based on the distance to the diamond and health.
    The sparse reward function will give a large positive reward when the agent reaches the diamond and a large
     negative reward when the agent dies.
    In addition, we should implement a threat detection mechanism to avoid unnecessary deaths.
    """
    import numpy as np

    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
     GLOBAL_DATA):
        # Initialize reward
        reward = 0

        # If the agent is closer to diamond than before, give a positive reward
        if "diamond" in current_nearest_blocks and "diamond" in previous_nearest_blocks:
            if current_nearest_blocks["diamond"][0] < previous_nearest_blocks["diamond"][0]:
                reward += 1
            elif current_nearest_blocks["diamond"][0] > previous_nearest_blocks["diamond"][0]:
                reward -= 1

        # If the agent is close to lava, give a negative reward
        if "lava" in current_nearest_blocks:
            reward -= 1

        # If the agent is not in pitch range 0 to 45, give a negative reward
        cur_pitch = past_agent_positions[-1][-1]
        if cur_pitch < 0 or cur_pitch > 45:
            reward -= 2

        # If the agent's health decreases, give a negative reward
        if "health" in GLOBAL_DATA:
            if health < GLOBAL_DATA["health"]:
                reward -= 1
        GLOBAL_DATA["health"] = health

        return reward

    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
     GLOBAL_DATA):
        # Initialize reward
        reward = 0

        # If the agent reaches the diamond, give a large positive reward
        if "diamond" in current_nearest_blocks and current_nearest_blocks["diamond"][0] < 2:
            reward += 100

        # If the agent dies, give a large negative reward
        if health == 0:
            reward -= 100

        return reward

    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health,
     past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health,
     past_agent_positions, GLOBAL_DATA)

    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
```

Algorithm II Reward Function Example of Approaching a Tree on Plains .

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions
     , GLOBAL_DATA):
    # Thoughts:
    # Objective: The agent should approach the nearest wood block in the plains biome.
    # We want to encourage the agent to:
    # - Make progress towards wood
    # - Keep health high
    # - Avoid circling around or staying still
    # We use a dense reward to encourage the above behaviors, and a sparse reward to give a big bonus when the
     agent achieves the objective.
    import numpy as np
    import math

    # Helper function to calculate the Euclidean distance between two points
    def calculate_distance(point1, point2):
        return math.sqrt(sum([(a - b) ** 2 for a, b in zip(point1, point2)]))

    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
     GLOBAL_DATA):
        reward = 0

        # Encourage the agent to approach wood
        if "wood" in current_nearest_blocks:
            current_distance = current_nearest_blocks["wood"][0]
            if "wood" in previous_nearest_blocks:
                previous_distance = previous_nearest_blocks["wood"][0]
                # Encourage the agent to decrease the distance to the nearest wood
                if current_distance < previous_distance:
                    reward += 1
                # Discourage the agent to increase the distance to the nearest wood
                else:
                    reward -= 1
            # Encourage the agent to find wood
            else:
                reward += 1

        # Encourage the agent to keep health high
        if health <= 3:
            reward -= 1

        # Discourage the agent from circling around or staying still
        if len(past_agent_positions) > 1:
            current_position = past_agent_positions[-1][:3]
            previous_position = past_agent_positions[-2][:3]
            # Encourage the agent to move
            if calculate_distance(current_position, previous_position) < 0.1:
                reward -= 1

        return reward

    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
     GLOBAL_DATA):
        reward = 0

        # Give a big bonus when the agent achieves the objective
        if "wood" in current_nearest_blocks and current_nearest_blocks["wood"][0] < 1:
            reward += 10

        return reward

    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health,
     past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health,
     past_agent_positions, GLOBAL_DATA)

    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
```

Algorithm III Reward Function Example of Approaching a Cow on Plains

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions
    , GLOBAL_DATA):
    """
    Thoughts:
    - The agent needs to find a cow and approach it. So, the primary factor affecting the reward is the distance to
      the nearest cow.
    - The agent should avoid getting hurt (losing health).
    - We do not need to consider inventory change in this task.
    - We need to encourage the agent to keep exploring, so we also consider the agent's movement.
    """

    import numpy as np

    # Helper function to calculate the Euclidean distance between two positions
    def calculate_distance(pos1, pos2):
        return np.sqrt((pos1[0] - pos2[0]) ** 2 + (pos1[1] - pos2[1]) ** 2 + (pos1[2] - pos2[2]) ** 2)

    def dense(current_nearest_blocks, previous_nearest_blocks, health, past_agent_positions):
        # Reward for approaching the cow
        cow_reward = 0
        if "cow" in current_nearest_blocks:
            if "cow" in previous_nearest_blocks:
                # If the agent is getting closer to the cow, give a positive reward
                if current_nearest_blocks["cow"][0] < previous_nearest_blocks["cow"][0]:
                    cow_reward = 1
                # If the agent is getting farther from the cow, give a negative reward
                elif current_nearest_blocks["cow"][0] > previous_nearest_blocks["cow"][0]:
                    cow_reward = -1

        # Reward for moving (exploring)
        move_reward = 0
        if len(past_agent_positions) > 1:
            # If the agent is moving, give a positive reward
            if calculate_distance(past_agent_positions[-1], past_agent_positions[-2]) > 0:
                move_reward = 1
            # If the agent is not moving, give a negative reward
            else:
                move_reward = -1

        return cow_reward + move_reward

    def sparse(current_nearest_blocks, health):
        # If the agent finds a cow and approaches it successfully, give a large positive reward
        if "cow" in current_nearest_blocks and current_nearest_blocks["cow"][0] < 2:
            return 100
        # If the agent gets hurt, give a large negative reward
        elif health < 10:
            return -100
        else:
            return 0

    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, health, past_agent_positions)
    sparse_reward = sparse(current_nearest_blocks, health)

    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
```

Algorithm IV Reward Function Example of Attacking a Cow on Plains

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions
     , GLOBAL_DATA, damage_dealt, kill_entity):
    """
    Thoughts:
    - The agent needs to find a cow and approach and kill it. We can give a positive dense reward for approaching
     or attacking a cow.
    - We can give a sparse reward for killing a cow.
    """

    import numpy as np

    def dense(current_nearest_blocks, previous_nearest_blocks, damage_dealt):
        # Reward for approaching the cow
        approach_reward = 0
        if "cow" in current_nearest_blocks and "cow" in previous_nearest_blocks:
            if current_nearest_blocks["cow"][0] < previous_nearest_blocks["cow"][0]:
                approach_reward = 1
            elif current_nearest_blocks["cow"][0] > previous_nearest_blocks["cow"][0]:
                approach_reward = -1

        # Reward for attacking the cow
        if damage_dealt > 0:
            attack_reward = 5
        else:
            attack_reward = 0

        return approach_reward + attack_reward

    def sparse(kill_entity):
        # Reward for killing the cow
        if kill_entity["cow"] > 0:
            return 100
        else:
            return 0

    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, damage_dealt)
    sparse_reward = sparse(kill_entity)

    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
```

H. Example of Full Interactions with LLM

We show an example of interactions with the LLM for the diamond exploration task, including initializing reward function and one iteration of updating reward function. **`INPUT`** is the input prompt and **`OUTPUT`** is the response from GPT-4. We highlight some lines that are related to critic revising and reward updating in yellow.

`DESIGNER INPUT:`

You are now a proficient Minecraft player. You should help me write proper reward functions to train a Minecraft agent with reinforcement learning to complete the described task.

Task description

- Objective: Find and approach diamond with higher success rate and avoid death.

- Initial Status:
  1. The agent is under ground at y level 11, the most common level to find diamonds.
  2. The agent already has an iron pickaxe.
- Success criteria: The distance to the nearest diamond block is less than 2 meter.

- Procedure: Explore horizontally to find a diamond, face it and approach it.

Input parameters of the reward function

- current_nearest_blocks: nearby block types and nearest distance of each block type at the **current step**. It is a dictionary, consisting of the names, relative distances with the agent, and relative angles (yaw and pitch) with the agent, in the form of: {NAME_1: (DIST_1, YAW_1, PITCH_1), NAME_2: (DIST_2, YAW_2, PITCH_2)}. For example, {"wood": (24.7, 1.48, -1.57), "cobblestone": (1.3, -0.17, 1.05), "iron_ore": (4.5, 0.61, -0.17)}. If a certrain block type does not exist in the field of view, it is not present in the dictionary. The yaw and pitch here is relative to the agent's front, i.e., pitch = 0 and yaw = 0 is the front of the agent.

- previou_nearest_blocks: nearby block types and nearest distance of each block type at the **previous step**, with the same format as the current_nearest_blocks.

- inventory_change: the change of the agent's inventory from the previous step to current step, in the form of a dictionary: {NAME_1: CHANGE_NUM_1, NAME_2: CHANGE_NUM_2}. Positive values mean increase and negative values mean decrease. For example, {"wood": 2, "dirt": 3, "stone_pickaxe": -1}.

- health: an integer value in range 0 to 10 indicating the health level of the agent. 0 means death and 10 means full health.

- past_agent_positions: the history of location of agent, in the form of a list: [[x1, y1, z1, yaw1, pitch1], [x2, y2, z2, yaw2, pitch2], ...]. The yaw and pitch here are relative to the agent's initial forward direction, i.e., pitch = 0 and yaw = 0 is the front of the agent when it was born. The length of the list is the number of steps the agent has taken. The last element of the list is the current location of the agent.

- GLOBAL_DATA: a global variable. It is initialized as a dictionary. You can save necessary information between different steps with it.

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Reward function requirements

- You should write a dense reward function `dense` and a sparse reward function `sparse`. The sparse reward indicates achieving the goal or receiving heavy punishment. The dense reward provides intermediate signal to guide the agent in the process of achieving the goal. The magnitude of the return value does not matter, but the sign (positive or negative) is important. The final reward will be `np.sign(sparse(...)) * 1 + np.sign(dense(...)) * 0.1`.

Output Requirements

- The reward function should be written in Python 3.9.

- Output the code block only. **Do not output anything else outside the code block**.

- You should include **sufficient comments** in your reward function to explain your thoughts, the objective and **implementation details**. The implementation can be specified to a specific line of code.

- If you need to import packages (e.g. math, numpy) or define helper functions, define them at the beginning of the function. Do not use unimported packages and undefined functions.

Output format

Strictly follow the following format. **Do not output anything else outside the code block**.

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    # Thoughts:
    # ...
    # (import packages and define helper functions)
    import numpy as np
    ...
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
    ...
```

Now write a reward function. Then in each iteration, I will use the reward function to train an RL agent, and test it in the environment. I will give you possible reasons of the failure found during the testing, and you should modify the reward function accordingly.

DESIGNER OUTPUT:

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    """
    Thoughts:
    In order to make the agent approach the diamond, we need to give positive rewards for actions that
    make the agent get closer to the diamond and negative rewards for actions that make the agent get further from the diamond.
    We also need to give a negative reward if the agent's health decreases to encourage the agent to avoid dangerous situations.
    The dense reward function will be based on the distance to the diamond and health.
    The sparse reward function will give a large positive reward when the agent reaches the diamond and a large negative reward when the agent dies.
    """
    import numpy as np
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        # Initialize reward
```

```
        reward = 0
        # If the agent is closer to diamond than before, give a positive reward
        if "diamond" in current_nearest_blocks and "diamond" in previous_nearest_blocks:
            if current_nearest_blocks["diamond"][0] < previous_nearest_blocks["diamond"][0]:
                reward += 1
            elif current_nearest_blocks["diamond"][0] > previous_nearest_blocks["diamond"][0]:
                reward -= 1
        # If the agent's health decreases, give a negative reward
        if health < GLOBAL_DATA["health"]:
            reward -= 1
        GLOBAL_DATA["health"] = health
        return reward
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        # Initialize reward
        reward = 0
        # If the agent reaches the diamond, give a large positive reward
        if "diamond" in current_nearest_blocks and current_nearest_blocks["diamond"][0] < 1:
            reward += 100
        # If the agent dies, give a large negative reward
        if health == 0:
            reward -= 100
        return reward
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(dense_reward) * 1 + np.sign(sparse_reward) * 0.1
```

CRITIC INPUT:

You are now a proficient Minecraft player. I will give you a reward function written in Python, which is used to train an RL agent to play Minecraft.

You should help me to evaluate whether the reward function has met the requirements.

Task description

- Objective: Find and approach diamond with higher success rate and avoid death.

- Initial Status:
  1. The agent is under ground at y level 11, the most common level to find diamonds.
  2. The agent already has an iron pickaxe.
- Success criteria: The distance to the nearest diamond block is less than 2 meter.

- Procedure: Explore horizontally to find a diamond, face it and approach it.

Input parameters of the reward function

- current_nearest_blocks: nearby block types and nearest distance of each block type at the **current step**. It is a dictionary, consisting of the names, relative distances with the agent, and relative angles (yaw and pitch) with the agent, in the form of: {NAME_1: (DIST_1, YAW_1, PITCH_1), NAME_2: (DIST_2, YAW_2, PITCH_2)}. For example, {"wood": (24.7, 1.48, -1.57), "cobblestone": (1.3, -0.17, 1.05), "iron_ore": (4.5, 0.61, -0.17)}. If a certrain block type does not exist in the field of view, it is not present in the dictionary. The yaw and pitch here is relative to the agent's front, i.e., pitch = 0 and yaw = 0 is the front of the agent.

- previou_nearest_blocks: nearby block types and nearest distance of each block type at the **previous step**, with the same format as the current_nearest_blocks.

- inventory_change: the change of the agent's inventory from the previous step to current step, in the form of a dictionary: {NAME_1: CHANGE_NUM_1, NAME_2: CHANGE_NUM_2}. Positive values mean increase and negative values mean decrease. For example, {"wood": 2, "dirt": 3, "stone_pickaxe": -1}.

- health: an integer value in range 0 to 10 indicating the health level of the agent. 0 means death and 10 means full health.

- past_agent_positions: the history of location of agent, in the form of a list: [[x1, y1, z1, yaw1, pitch1], [x2, y2, z2, yaw2, pitch2], ...]. The yaw and pitch here are relative to the agent's initial forward direction, i.e., pitch = 0 and yaw = 0 is the front of the agent when it was born. The length of the list is the number of steps the agent has taken. The last element of the list is the current location of the agent.

- GLOBAL_DATA: a global variable. It is initialized as a dictionary. You can save necessary information between different steps with it.

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Reward function requirements

- You should write a dense reward function `dense`and a sparse reward function `sparse`. The sparse reward indicates achieving the goal or receiving heavy punishment. The dense reward provides intermediate signal to guide the agent in the process of achieving the goal. The magnitude of the return value does not matter, but the sign (positive or negative) is important. The final reward will be `np.sign(sparse(...)) * 1 + np.sign(dense(...)) * 0.1`.

Requirements

1. You should check whether the reward function meets the **reward function requirements** above.

2. Your judgement should consider whether the comments and the detailed implementation code are consistent. You can judge whether a line is correct by its comment, but do only rely on the comments.

3. You should also check the format of the reward function. It should be like:

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    # Thoughts:
    # ...
    # (import packages and define helper functions)
    import numpy as np
    ...
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
    ...
```

4. Consider the **Information about the game state** above.

5. **You don’t need to check whether the reward function meets the task description**.

6. You should first provide a reasoning and decision. If the reward function fails, provide a critique to help to improve the reward function.

Output format

You should only respond in JSON format as described below:

```
{
  "reasoning": "reasoning",
  "success": boolean,
  "critique": critique,
}
```

Ensure the response can be parsed by Python `json.loads`, e.g. no trailing commas, no single quotes, etc. Do not output anything else like explanations.

Reward function to be evaluated

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    """
    Thoughts:
    In order to make the agent approach the diamond, we need to give positive rewards for actions that
    make the agent get closer to the diamond and negative rewards for actions that make the agent get further from the diamond.
    We also need to give a negative reward if the agent’s health decreases to encourage the agent to avoid dangerous situations.
    The dense reward function will be based on the distance to the diamond and health.
    The sparse reward function will give a large positive reward when the agent reaches the diamond and a large negative reward when the agent dies.
    """
    import numpy as np
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        # Initialize reward
        reward = 0
        # If the agent is closer to diamond than before, give a positive reward
        if "diamond" in current_nearest_blocks and "diamond" in previous_nearest_blocks:
            if current_nearest_blocks["diamond"][0] < previous_nearest_blocks["diamond"][0]:
                reward += 1
            elif current_nearest_blocks["diamond"][0] > previous_nearest_blocks["diamond"][0]:
                reward -= 1
        # If the agent’s health decreases, give a negative reward
        if health < GLOBAL_DATA["health"]:
            reward -= 1
        GLOBAL_DATA["health"] = health
        return reward
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        # Initialize reward
        reward = 0
        # If the agent reaches the diamond, give a large positive reward
        if "diamond" in current_nearest_blocks and current_nearest_blocks["diamond"][0] < 1:
            reward += 100
        # If the agent dies, give a large negative reward
        if health == 0:
            reward -= 100
        return reward
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
```

```
GLOBAL_DATA)
    return np.sign(dense_reward) * 1 + np.sign(sparse_reward) * 0.1
```

CRITIC OUTPUT:

```
{
    "reasoning": "The reward function does not consider that `health` is not initialized in GLOBAL_DATA. It will raise a error because the key `health` is not found.",
    "success": false,
    "critique": "The reward function needs to be revised to initialize `health` in GLOBAL_DATA."
}
```

DESIGNER INPUT:

The reward function fails to pass the verification. The reasoning is:

```
The reward function does not consider that `health` is not initialized in GLOBAL_DATA. It will raise a error because the key `health` is not found.
```

The critique is:

```
The reward function needs to be revised to initialize `health` in GLOBAL_DATA.
```

Requirements

Please consider the reasoning and critique, and modify the reward function.

1. If necessary, you can write a totally different reward function than the current one.

2. In the first part of the reward function, you should provide your thoughts of modifying the reward function. The thoughts should not directly copy the given reasoning or critique. **The thoughts should be concise.**

Reward function requirements

- You should write a dense reward function `dense` and a sparse reward function `sparse`. The sparse reward indicates achieving the goal or receiving heavy punishment. The dense reward provides intermediate signal to guide the agent in the process of achieving the goal. The magnitude of the return value does not matter, but the sign (positive or negative) is important. The final reward will be `np.sign(sparse(...)) * 1 + np.sign(dense(...)) * 0.1`.

Output format

Strictly follow the following format. **Do not output anything else outside the code block**.

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    # Thoughts:
    # ...
    # (import packages and define helper functions)
    import numpy as np
    ...
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
```

```
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
    ...
```

DESIGNER OUTPUT:

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    """
    Thoughts:
    In order to make the agent approach the diamond, we need to give positive rewards for actions that
    make the agent get closer to the diamond and negative rewards for actions that make the agent get further from the diamond.
    We also need to give a negative reward if the agent's health decreases to encourage the agent to avoid dangerous situations.
    The dense reward function will be based on the distance to the diamond and health.
    The sparse reward function will give a large positive reward when the agent reaches the diamond and a large negative reward when the agent dies.
    """
    import numpy as np
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        # Initialize reward
        reward = 0
        # If the agent is closer to diamond than before, give a positive reward
        if "diamond" in current_nearest_blocks and "diamond" in previous_nearest_blocks:
            if current_nearest_blocks["diamond"][0] < previous_nearest_blocks["diamond"][0]:
                reward += 1
            elif current_nearest_blocks["diamond"][0] > previous_nearest_blocks["diamond"][0]:
                reward -= 1
        # If the agent's health decreases, give a negative reward
        if "health" in GLOBAL_DATA:
            if health < GLOBAL_DATA["health"]:
                reward -= 1
        GLOBAL_DATA["health"] = health
        return reward
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        # Initialize reward
        reward = 0
        # If the agent reaches the diamond, give a large positive reward
        if "diamond" in current_nearest_blocks and current_nearest_blocks["diamond"][0] < 1:
            reward += 100
        # If the agent dies, give a large negative reward
        if health == 0:
            reward -= 100
        return reward
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(dense_reward) * 1 + np.sign(sparse_reward) * 0.1
```

CRITIC INPUT:

You are now a proficient Minecraft player. I will give you a reward function written in Python, which is used to train an RL agent to play Minecraft.

You should help me to evaluate whether the reward function has met the requirements.

Task description

- Objective: Find and approach diamond with higher success rate and avoid death.

- Initial Status:
  1. The agent is under ground at y level 11, the most common level to find diamonds.
  2. The agent already has an iron pickaxe.
- Success criteria: The distance to the nearest diamond block is less than 2 meter.

- Procedure: Explore horizontally to find a diamond, face it and approach it.

Input parameters of the reward function

- current_nearest_blocks: nearby block types and nearest distance of each block type at the **current step**. It is a dictionary, consisting of the names, relative distances with the agent, and relative angles (yaw and pitch) with the agent, in the form of: {NAME_1: (DIST_1, YAW_1, PITCH_1), NAME_2: (DIST_2, YAW_2, PITCH_2)}. For example, {"wood": (24.7, 1.48, -1.57), "cobblestone": (1.3, -0.17, 1.05), "iron_ore": (4.5, 0.61, -0.17)}. If a certrain block type does not exist in the field of view, it is not present in the dictionary. The yaw and pitch here is relative to the agent's front, i.e., pitch = 0 and yaw = 0 is the front of the agent.

- previou_nearest_blocks: nearby block types and nearest distance of each block type at the **previous step**, with the same format as the current_nearest_blocks.

- inventory_change: the change of the agent's inventory from the previous step to current step, in the form of a dictionary: {NAME_1: CHANGE_NUM_1, NAME_2: CHANGE_NUM_2}. Positive values mean increase and negative values mean decrease. For example, {"wood": 2, "dirt": 3, "stone_pickaxe": -1}.

- health: an integer value in range 0 to 10 indicating the health level of the agent. 0 means death and 10 means full health.

- past_agent_positions: the history of location of agent, in the form of a list: [[x1, y1, z1, yaw1, pitch1], [x2, y2, z2, yaw2, pitch2], ...]. The yaw and pitch here are relative to the agent's initial forward direction, i.e., pitch = 0 and yaw = 0 is the front of the agent when it was born. The length of the list is the number of steps the agent has taken. The last element of the list is the current location of the agent.

- GLOBAL_DATA: a global variable. It is initialized as a dictionary. You can save necessary information between different steps with it.

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Reward function requirements

- You should write a dense reward function `dense` and a sparse reward function `sparse`. The sparse reward indicates achieving the goal or receiving heavy punishment. The dense reward provides intermediate signal to guide the agent in the process of achieving the goal. The magnitude of the return value does not matter, but the sign (positive or negative) is important. The final reward will be `np.sign(sparse(...)) * 1 + np.sign(dense(...)) * 0.1`.

Requirements

1. You should check whether the reward function meets the **reward function requirements** above.

2. Your judgement should consider whether the comments and the detailed implementation code are consistent. You can judge whether a line is correct by its comment, but do only rely on the comments.

3. You should also check the format of the reward function. It should be like:

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA):
    # Thoughts:
    # ...
    # (import packages and define helper functions)
    import numpy as np
    ...
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA):
        ...
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA):
        ...
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA)
    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
    ...
```

4. Consider the **Information about the game state** above.

5. **You don’t need to check whether the reward function meets the task description**.

6. You should first provide a reasoning and decision. If the reward function fails, provide a critique to help to improve the reward function.

Output format

You should only respond in JSON format as described below:

```
{
    "reasoning": "reasoning",
    "success": boolean,
    "critique": critique,
}
```

Ensure the response can be parsed by Python `json.loads`, e.g. no trailing commas, no single quotes, etc. Do not output anything else like explanations.

Reward function to be evaluated

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA):
    """
    Thoughts:
    In order to make the agent approach the diamond, we need to give positive rewards for actions that
    make the agent get closer to the diamond and negative rewards for actions that make the agent get further from the diamond.
    We also need to give a negative reward if the agent’s health decreases to encourage the agent to avoid dangerous situations.
    The dense reward function will be based on the distance to the diamond and health.
    The sparse reward function will give a large positive reward when the agent reaches the diamond and a large negative reward
when the agent dies.
    """
    import numpy as np
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions,
GLOBAL_DATA):
        # Initialize reward
        reward = 0
```

```
        # If the agent is closer to diamond than before, give a positive reward
        if "diamond" in current_nearest_blocks and "diamond" in previous_nearest_blocks:
            if current_nearest_blocks["diamond"][0] < previous_nearest_blocks["diamond"][0]:
                reward += 1
            elif current_nearest_blocks["diamond"][0] > previous_nearest_blocks["diamond"][0]:
                reward -= 1
        # If the agent's health decreases, give a negative reward
        if "health" in GLOBAL_DATA:
            if health < GLOBAL_DATA["health"]:
                reward -= 1
        GLOBAL_DATA["health"] = health
        return reward
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        # Initialize reward
        reward = 0
        # If the agent reaches the diamond, give a large positive reward
        if "diamond" in current_nearest_blocks and current_nearest_blocks["diamond"][0] < 1:
            reward += 100
        # If the agent dies, give a large negative reward
        if health == 0:
            reward -= 100
        return reward
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(dense_reward) * 1 + np.sign(sparse_reward) * 0.1
```

CRITIC OUTPUT:

```
{
    "reasoning": "The reward function meets the requirements. It considers sparse reward of completing the objective and death, and dense reward of getting closer to diamond and health decrease.",
    "success": true,
    "critique": null
}
```

(TRAIN AN AGENT WITH THE REWARD FUNCTION)

ANALYZER INPUT:

You are now a proficient Minecraft player. I have trained an RL agent and tested for several times in the Minecraft game environment.

I will give you the information of the failed test results, i.e. trajectories of actions, rewards and observations. You should help me write proper analysis of possible reasons of failure and inefficiency, and your suggestion on the game strategy.

Task description

- Objective: Find and approach diamond with higher success rate and avoid death.

- Initial Status:
  1. The agent is under ground at y level 11, the most common level to find diamonds.
  2. The agent already has an iron pickaxe.
- Success criteria: The distance to the nearest diamond block is less than 2 meter.

- Procedure: Explore horizontally to find a diamond, face it and approach it.

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Input format

Failed trajectories

The failed trajectories are shown as a list, where each entry is a dictionary representing a trajectory. Note that **only the failed cases are provided.**

The each item of "history" is a list representing statistics of each single step. When the history is too long, it will be truncated to the last 32 steps, indicated by the value of "truncated".

"dead" indicates whether the agent has died.

The format is:

```
[
  {
    "history": {
      "rewards": [reward1, ...],
      "actions": [action1, action2, ...],
      "locations": [[x1, y1, z1, pitch1, yaw1], ...],
      "inventory_change": {item1: delta_num1, ...},
      "truncated": True or False,
    },
    "final_health": health,
    "final_inventory": {item1: num1, ...},
    "final_nearest_blocks": {block_name1: nearest_distance1, block_name2: nearest_distance2, ...},
    "block_under_foot": block_name,
    "dead": True or False,
  },
  ...
]
```

where "final_health" is in range 0 and 10. "block_under_foot" is the block right under the agent.

Statistics

The statistics is a dictionary of all the test results, including **both successful and failed trajectories**.

The format is:

```
{
  "success_rate" : success_rate,
}
```

Requirements

Please write proper analysis of possible reasons of failure and inefficiency, and your suggestion on the game strategy. You should not not be limited to the task description above, but also come up with other failure cases based on the test results.

The analysis and suggestion should be concise.

Test results

Failed trajectories

```
[{’block_under_foot’: ’lava’,
  ’dead’: True,
  ’final_health’: 0.5,
  ’final_inventory’: {’coal’: 3,
                      ’cobblestone’: 84,
                      ’crafting_table’: 1,
                      ’furnace’: 1,
                      ’gravel’: 3,
                      ’iron_pickaxe’: 1,
                      ’planks’: 2,
                      ’stone’: 17,
                      ’stone_pickaxe’: 10,
                      ’wooden_pickaxe’: 1},
  ’final_nearest_blocks’: {’lava’: [0.01, -0.87, -0.63]},
  ’history’: {’actions’: [’noop’, ’attack’, ’noop’, ’attack’, ’noop’, ’attack’,
                          ’attack’, ’turn_down’, ’attack’, ’attack’, ’noop’,
                          ’attack’, ’forward’, ’forward’, ’noop’, ’noop’,
                          ’forward_jump’, ’forward_jump’, ’attack’, ’attack’,
                          ’attack’, ’noop’, ’jump’, ’noop’, ’noop’, ’jump’,
                          ’attack’, ’turn_down’, ’turn_up’, ’forward_jump’,
                          ’turn_down’, ’turn_up’],
              ’inventory_change’: [{}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {},
                                   {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {},
                                   {}, {}, {}, {}, {}, {}, {}, {}, {}, {}],
              ’locations’: [[-205.42, 9.27, 474.4, 15.0, -90.0],
                            [-205.41, 9.24, 474.4, 15.0, -90.0],
                            [-205.41, 9.2, 474.4, 15.0, -90.0],
                            [-205.41, 9.16, 474.4, 15.0, -90.0],
                            [-205.41, 9.12, 474.4, 15.0, -90.0],
                            [-205.41, 9.08, 474.4, 15.0, -90.0],
                            [-205.41, 9.04, 474.4, 15.0, -90.0],
                            [-205.41, 9.0, 474.4, 15.0, -90.0],
                            [-205.41, 8.96, 474.4, 30.0, -90.0],
                            [-205.41, 8.92, 474.4, 30.0, -90.0],
                            [-205.41, 8.88, 474.4, 30.0, -90.0],
                            [-205.41, 8.84, 474.4, 30.0, -90.0],
                            [-205.41, 8.8, 474.4, 30.0, -90.0],
                            [-205.39, 8.76, 474.4, 30.0, -90.0],
                            [-205.36, 8.72, 474.4, 30.0, -90.0],
                            [-205.34, 8.68, 474.4, 30.0, -90.0],
                            [-205.34, 8.64, 474.4, 30.0, -90.0],
                            [-205.31, 8.64, 474.4, 30.0, -90.0],
                            [-205.3, 8.66, 474.4, 30.0, -90.0],
                            [-205.3, 8.65, 474.4, 30.0, -90.0],
                            [-205.3, 8.63, 474.4, 30.0, -90.0],
                            [-205.3, 8.6, 474.4, 30.0, -90.0],
                            [-205.3, 8.56, 474.4, 30.0, -90.0],
                            [-205.3, 8.56, 474.4, 30.0, -90.0],
                            [-205.3, 8.54, 474.4, 30.0, -90.0],
                            [-205.3, 8.51, 474.4, 30.0, -90.0],
```

```
                  [-205.3, 8.52, 474.4, 30.0, -90.0],
                  [-205.3, 8.5, 474.4, 30.0, -90.0],
                  [-205.3, 8.47, 474.4, 45.0, -90.0],
                  [-205.3, 8.43, 474.4, 30.0, -90.0],
                  [-205.3, 8.44, 474.4, 30.0, -90.0],
                  [-205.3, 8.42, 474.4, 45.0, -90.0]],
          ’nearest_blocks’: [{’lava’: [0.25, 0.87, -0.63]},
                             {’lava’: [0.31, 0.87, -0.63]},
                             {’lava’: [0.38, 0.87, -0.63]},
                             {’lava’: [0.41, 0.07, -0.03]},
                             {’lava’: [0.41, 0.07, -0.03]},
                             {’lava’: [0.41, 0.07, -0.03]},
                             {’lava’: [0.41, 0.07, -0.03]},
                             {’lava’: [0.41, 0.35, -0.1]},
                             {’lava’: [0.41, 0.35, -0.1]},
                             {’lava’: [0.41, 0.35, -0.1]},
                             {’lava’: [0.41, 0.35, -0.1]},
                             {’lava’: [0.41, 0.35, -0.1]},
                             {’lava’: [0.38, -1.57, 0.0]},
                             {’lava’: [0.34, -1.57, 0.0]},
                             {’lava’: [0.3, -1.57, 0.0]},
                             {’lava’: [0.26, -1.57, 0.0]},
                             {’lava’: [0.26, -1.57, 0.0]},
                             {’lava’: [0.28, -1.57, 0.0]},
                             {’lava’: [0.27, -1.57, 0.0]},
                             {’lava’: [0.25, -0.87, -0.63]},
                             {’lava’: [0.22, -0.87, -0.63]},
                             {’lava’: [0.18, -0.87, -0.63]},
                             {’lava’: [0.18, -0.87, -0.63]},
                             {’lava’: [0.16, -0.87, -0.63]},
                             {’lava’: [0.13, -0.87, -0.63]},
                             {’lava’: [0.14, -0.87, -0.63]},
                             {’lava’: [0.12, -0.87, -0.63]},
                             {’lava’: [0.09, -0.87, -0.63]},
                             {’lava’: [0.05, -1.57, 0.0]},
                             {’lava’: [0.06, -0.87, -0.63]},
                             {’lava’: [0.04, -0.87, -0.63]},
                             {’lava’: [0.01, -0.87, -0.63]}],
          ’rewards’: [],
          ’truncated’: True}},
{’block_under_foot’: ’stone’,
 ’dead’: False,
 ’final_health’: 10.0,
 ’final_inventory’: {’coal’: 5,
                     ’cobblestone’: 133,
                     ’crafting_table’: 1,
                     ’dirt’: 18,
                     ’furnace’: 1,
                     ’iron_pickaxe’: 1,
                     ’planks’: 2,
                     ’stone’: 37,
                     ’stone_pickaxe’: 9,
                     ’wooden_pickaxe’: 1},
 ’final_nearest_blocks’: {’stone’: [0.5, 0.87, -0.63]},
 ’history’: {’actions’: [’attack’, ’attack’, ’attack’, ’attack’, ’attack’,
                         ’attack’, ’attack’, ’attack’, ’attack’, ’attack’,
```

```
            'attack', 'attack', 'attack', 'attack', 'attack',
            'noop', 'turn_up', 'turn_up', 'turn_up', 'noop',
            'forward', 'forward', 'forward', 'forward', 'forward',
            'forward', 'back', 'noop', 'noop', 'attack', 'attack',
            'attack'],
'inventory_change': [{}, {}, {}, {'cobblestone': 1}, {}, {}, {},
                     {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {},
                     {}, {}, {}, {}, {}, {}, {'cobblestone': 1},
                     {}, {}, {}, {}, {}, {}, {}],
'locations': [[611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 45.0, -180.0],
              [611.54, 11.0, 212.67, 30.0, -180.0],
              [611.54, 11.0, 212.67, 15.0, -180.0],
              [611.54, 11.0, 212.67, 0.0, -180.0],
              [611.54, 11.0, 212.67, 0.0, -180.0],
              [611.54, 11.0, 212.57, 0.0, -180.0],
              [611.54, 11.0, 212.42, 0.0, -180.0],
              [611.54, 11.0, 212.24, 0.0, -180.0],
              [611.54, 11.0, 212.04, 0.0, -180.0],
              [611.54, 11.0, 211.84, 0.0, -180.0],
              [611.54, 11.0, 211.63, 0.0, -180.0],
              [611.54, 11.0, 211.61, 0.0, -180.0],
              [611.54, 11.0, 211.6, 0.0, -180.0],
              [611.54, 11.0, 211.6, 0.0, -180.0],
              [611.54, 11.0, 211.6, 0.0, -180.0],
              [611.54, 11.0, 211.6, 0.0, -180.0]],
'nearest_blocks': [{'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
                   {'stone': [0.78, 0.73, -0.63]},
```

```
                    {'stone': [0.78, 0.73, -0.63]},
                    {'stone': [0.78, 0.49, -0.63]},
                    {'stone': [0.66, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]},
                    {'stone': [0.5, 0.87, -0.63]}],
        'rewards': [],
        'truncated': True}}]
```

Statistics

{'success_rate': 0.0}

ANALYZER OUTPUT:

Analysis of Possible Reasons of Failure and Inefficiency: In one of the failed trajectories, the agent died due to lava. The agent did not take any action to avoid the lava or move away from it. The agent should be programmed to detect and avoid dangerous blocks, such as lava, by turning or moving in a different direction.

Suggestion on Game Strategy: Train the agent to detect and avoid dangerous blocks, such as lava or deep holes. The agent should be programmed to prioritize safety and avoid situations that could lead to death or loss of health.

By implementing the suggestion and improving the agent's training, the success rate of finding diamonds can be increased.

DESIGNER INPUT:

You are now a proficient Minecraft player. I now have a reward function to train a Minecraft agent with reinforcement learning to complete the described task. The reward function is used to train the reinforcement learning agent for several times. I will provide you the analysis of failure and inefficiency and suggestions. You should help me modify the reward function.

Task description

- Objective: Find and approach diamond with higher success rate and avoid death.

- Initial Status:
  1. The agent is under ground at y level 11, the most common level to find diamonds.
  2. The agent already has an iron pickaxe.
- Success criteria: The distance to the nearest diamond block is less than 2 meter.

- Procedure: Explore horizontally to find a diamond, face it and approach it.

Input parameters of the reward function

- current_nearest_blocks: nearby block types and nearest distance of each block type at the **current step**. It is a dictionary, consisting of the names, relative distances with the agent, and relative angles (yaw and pitch) with the agent, in the form of: {NAME_1: (DIST_1, YAW_1, PITCH_1), NAME_2: (DIST_2, YAW_2, PITCH_2)}. For example, {"wood": (24.7, 1.48, -1.57), "cobblestone": (1.3, -0.17, 1.05), "iron_ore": (4.5, 0.61, -0.17)}. If a certrain block type does not exist in the field of view, it is not present in the dictionary. The yaw and pitch here is relative to the agent's front, i.e., pitch = 0 and yaw = 0 is the front of the agent.

- previou_nearest_blocks: nearby block types and nearest distance of each block type at the **previous step**, with the same format as the current_nearest_blocks.

- inventory_change: the change of the agent's inventory from the previous step to current step, in the form of a dictionary: {NAME_1: CHANGE_NUM_1, NAME_2: CHANGE_NUM_2}. Positive values mean increase and negative values mean decrease. For example, {"wood": 2, "dirt": 3, "stone_pickaxe": -1}.

- health: an integer value in range 0 to 10 indicating the health level of the agent. 0 means death and 10 means full health.

- past_agent_positions: the history of location of agent, in the form of a list: [[x1, y1, z1, yaw1, pitch1], [x2, y2, z2, yaw2, pitch2], ...]. The yaw and pitch here are relative to the agent's initial forward direction, i.e., pitch = 0 and yaw = 0 is the front of the agent when it was born. The length of the list is the number of steps the agent has taken. The last element of the list is the current location of the agent.

- GLOBAL_DATA: a global variable. It is initialized as a dictionary. You can save necessary information between different steps with it.

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Reward function requirements

- You should write a dense reward function `dense`and a sparse reward function `sparse`. The sparse reward indicates achieving the goal or receiving heavy punishment. The dense reward provides intermediate signal to guide the agent in the process of achieving the goal. The magnitude of the return value does not matter, but the sign (positive or negative) is important. The final reward will be `np.sign(sparse(...)) * 1 + np.sign(dense(...)) * 0.1`.

Current reward function

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    """
    Thoughts:
    In order to make the agent approach the diamond, we need to give positive rewards for actions that
    make the agent get closer to the diamond and negative rewards for actions that make the agent get further from the diamond.
    We also need to give a negative reward if the agent's health decreases to encourage the agent to avoid dangerous situations.
    The dense reward function will be based on the distance to the diamond and health.
    The sparse reward function will give a large positive reward when the agent reaches the diamond and a large negative reward when the agent dies.
    """
    import numpy as np
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        # Initialize reward
        reward = 0
        # If the agent is closer to diamond than before, give a positive reward
        if "diamond" in current_nearest_blocks and "diamond" in previous_nearest_blocks:
            if current_nearest_blocks["diamond"][0] < previous_nearest_blocks["diamond"][0]:
                reward += 1
            elif current_nearest_blocks["diamond"][0] > previous_nearest_blocks["diamond"][0]:
```

```
                reward -= 1
        # If the agent’s health decreases, give a negative reward
        if "health" in GLOBAL_DATA:
            if health < GLOBAL_DATA["health"]:
                reward -= 1
        GLOBAL_DATA["health"] = health
        return reward
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        # Initialize reward
        reward = 0
        # If the agent reaches the diamond, give a large positive reward
        if "diamond" in current_nearest_blocks and current_nearest_blocks["diamond"][0] < 1:
            reward += 100
        # If the agent dies, give a large negative reward
        if health == 0:
            reward -= 100
        return reward
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(dense_reward) * 1 + np.sign(sparse_reward) * 0.1
```

Reward function test result

The reward function is used to train the reinforcement learning agent for several times. Here is some analysis of failure and inefficiency and suggestions:

```
Analysis of Possible Reasons of Failure and Inefficiency: In one of the failed trajectories, the agent died due to lava. The agent did not take any action to avoid the lava or move away from it. The agent should be programmed to detect and avoid dangerous blocks, such as lava, by turning or moving in a different direction.

Suggestion on Game Strategy: Train the agent to detect and avoid dangerous blocks, such as lava or deep holes. The agent should be programmed to prioritize safety and avoid situations that could lead to death or loss of health.

By implementing the suggestion and improving the agent’s training, the success rate of finding diamonds can be increased.
```

Requirements

Please consider the analysis and suggestions above and modify the reward function.

1. You can both modify the current lines and add new lines.

2. If necessary, you can write a **totally different** reward function than the current one.

3. Consider modifing the reward and penalty values in the current reward function to balance them.

4. In the first part of the reward function, you should provide your thoughts of modifying the reward function. **The thoughts should be concise.**

Output format

Strictly follow the following format. **Do not output anything else outside the code block**.

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    # Thoughts:
    # ...
    # (import packages and define helper functions)
    import numpy as np
    ...
```

```
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
    ...
```

DESIGNER OUTPUT:

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    """
    Thoughts:
    In order to make the agent approach the diamond, we need to give positive rewards for actions that
    make the agent get closer to the diamond and negative rewards for actions that make the agent get further from the diamond.
    We also need to give a negative reward if the agent's health decreases to encourage the agent to avoid dangerous situations.
    The dense reward function will be based on the distance to the diamond and health.
    The sparse reward function will give a large positive reward when the agent reaches the diamond and a large negative reward when the agent dies.
    In addition, we should implement a threat detection mechanism to avoid unnecessary deaths.
    """
    import numpy as np
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        # Initialize reward
        reward = 0
        # If the agent is closer to diamond than before, give a positive reward
        if "diamond" in current_nearest_blocks and "diamond" in previous_nearest_blocks:
            if current_nearest_blocks["diamond"][0] < previous_nearest_blocks["diamond"][0]:
                reward += 1
            elif current_nearest_blocks["diamond"][0] > previous_nearest_blocks["diamond"][0]:
                reward -= 1
        # If the agent is close to lava, give a negative reward
        if "lava" in current_nearest_blocks:
            reward -= 1
        # If the agent's health decreases, give a negative reward
        if "health" in GLOBAL_DATA:
            if health < GLOBAL_DATA["health"]:
                reward -= 1
        GLOBAL_DATA["health"] = health
        return reward
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        # Initialize reward
        reward = 0
        # If the agent reaches the diamond, give a large positive reward
        if "diamond" in current_nearest_blocks and current_nearest_blocks["diamond"][0] < 1:
            reward += 100
        # If the agent dies, give a large negative reward
        if health == 0:
            reward -= 100
```

```
        return reward
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(dense_reward) * 1 + np.sign(sparse_reward) * 0.1
```

`CRITIC INPUT:`

You are now a proficient Minecraft player. I will give you a reward function written in Python, which is used to train an RL agent to play Minecraft.

You should help me to evaluate whether the reward function has met the requirements.

Task description

- Objective: Find and approach diamond with higher success rate and avoid death.

- Initial Status:
  1. The agent is under ground at y level 11, the most common level to find diamonds.
  2. The agent already has an iron pickaxe.
- Success criteria: The distance to the nearest diamond block is less than 2 meter.

- Procedure: Explore horizontally to find a diamond, face it and approach it.

Input parameters of the reward function

- current_nearest_blocks: nearby block types and nearest distance of each block type at the **current step**. It is a dictionary, consisting of the names, relative distances with the agent, and relative angles (yaw and pitch) with the agent, in the form of: {NAME_1: (DIST_1, YAW_1, PITCH_1), NAME_2: (DIST_2, YAW_2, PITCH_2)}. For example, {"wood": (24.7, 1.48, -1.57), "cobblestone": (1.3, -0.17, 1.05), "iron_ore": (4.5, 0.61, -0.17)}. If a certrain block type does not exist in the field of view, it is not present in the dictionary. The yaw and pitch here is relative to the agent's front, i.e., pitch = 0 and yaw = 0 is the front of the agent.

- previou_nearest_blocks: nearby block types and nearest distance of each block type at the **previous step**, with the same format as the current_nearest_blocks.

- inventory_change: the change of the agent's inventory from the previous step to current step, in the form of a dictionary: {NAME_1: CHANGE_NUM_1, NAME_2: CHANGE_NUM_2}. Positive values mean increase and negative values mean decrease. For example, {"wood": 2, "dirt": 3, "stone_pickaxe": -1}.

- health: an integer value in range 0 to 10 indicating the health level of the agent. 0 means death and 10 means full health.

- past_agent_positions: the history of location of agent, in the form of a list: [[x1, y1, z1, yaw1, pitch1], [x2, y2, z2, yaw2, pitch2], ...]. The yaw and pitch here are relative to the agent's initial forward direction, i.e., pitch = 0 and yaw = 0 is the front of the agent when it was born. The length of the list is the number of steps the agent has taken. The last element of the list is the current location of the agent.

- GLOBAL_DATA: a global variable. It is initialized as a dictionary. You can save necessary information between different steps with it.

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Reward function requirements

- You should write a dense reward function `dense` and a sparse reward function `sparse`. The sparse reward indicates achieving the goal or receiving heavy punishment. The dense reward provides intermediate signal to guide the agent in the process of achieving the goal. The magnitude of the return value does not matter, but the sign (positive or negative) is important. The final reward will be `np.sign(sparse(...)) * 1 + np.sign(dense(...)) * 0.1`.

Requirements

1. You should check whether the reward function meets the **reward function requirements** above.

2. Your judgement should consider whether the comments and the detailed implementation code are consistent. You can judge whether a line is correct by its comment, but do only rely on the comments.

3. You should also check the format of the reward function. It should be like:

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    # Thoughts:
    # ...
    # (import packages and define helper functions)
    import numpy as np
    ...
    def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
        ...
    dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
    return np.sign(sparse_reward) * 1 + np.sign(dense_reward) * 0.1
    ...
```

4. Consider the **Information about the game state** above.

5. **You don't need to check whether the reward function meets the task description**.

6. You should first provide a reasoning and decision. If the reward function fails, provide a critique to help to improve the reward function.

Output format

You should only respond in JSON format as described below:

```
{
    "reasoning": "reasoning",
    "success": boolean,
    "critique": critique,
}
```

Ensure the response can be parsed by Python `json.loads`, e.g. no trailing commas, no single quotes, etc. Do not output anything else like explanations.

Reward function to be evaluated

```
def reward_function(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    """
    Thoughts:
```

```
In order to make the agent approach the diamond, we need to give positive rewards for actions that
make the agent get closer to the diamond and negative rewards for actions that make the agent get further from the diamond.
We also need to give a negative reward if the agent's health decreases to encourage the agent to avoid dangerous situations.
The dense reward function will be based on the distance to the diamond and health.
The sparse reward function will give a large positive reward when the agent reaches the diamond and a large negative reward when the agent dies.
In addition, we should implement a threat detection mechanism to avoid unnecessary deaths.
"""
import numpy as np
def dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    # Initialize reward
    reward = 0
    # If the agent is closer to diamond than before, give a positive reward
    if "diamond" in current_nearest_blocks and "diamond" in previous_nearest_blocks:
        if current_nearest_blocks["diamond"][0] < previous_nearest_blocks["diamond"][0]:
            reward += 1
        elif current_nearest_blocks["diamond"][0] > previous_nearest_blocks["diamond"][0]:
            reward -= 1
    # If the agent is close to lava, give a negative reward
    if "lava" in current_nearest_blocks:
        reward -= 1
    # If the agent's health decreases, give a negative reward
    if "health" in GLOBAL_DATA:
        if health < GLOBAL_DATA["health"]:
            reward -= 1
    GLOBAL_DATA["health"] = health
    return reward
def sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA):
    # Initialize reward
    reward = 0
    # If the agent reaches the diamond, give a large positive reward
    if "diamond" in current_nearest_blocks and current_nearest_blocks["diamond"][0] < 1:
        reward += 100
    # If the agent dies, give a large negative reward
    if health == 0:
        reward -= 100
    return reward
dense_reward = dense(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
sparse_reward = sparse(current_nearest_blocks, previous_nearest_blocks, inventory_change, health, past_agent_positions, GLOBAL_DATA)
return np.sign(dense_reward) * 1 + np.sign(sparse_reward) * 0.1
```

CRITIC OUTPUT:

```
{
    "reasoning": "The reward function meets the requirements. It considers sparse reward of completing the objective and death, and dense reward of getting closer to diamond, having lava nearby and health decrease.",
    "success": true,
    "critique": null
}
```

I. Example of GPT-4V as a Captioner

We show an example of using GPT-4V as a captioner to assist LLM in trajectory analysis of Figure III.

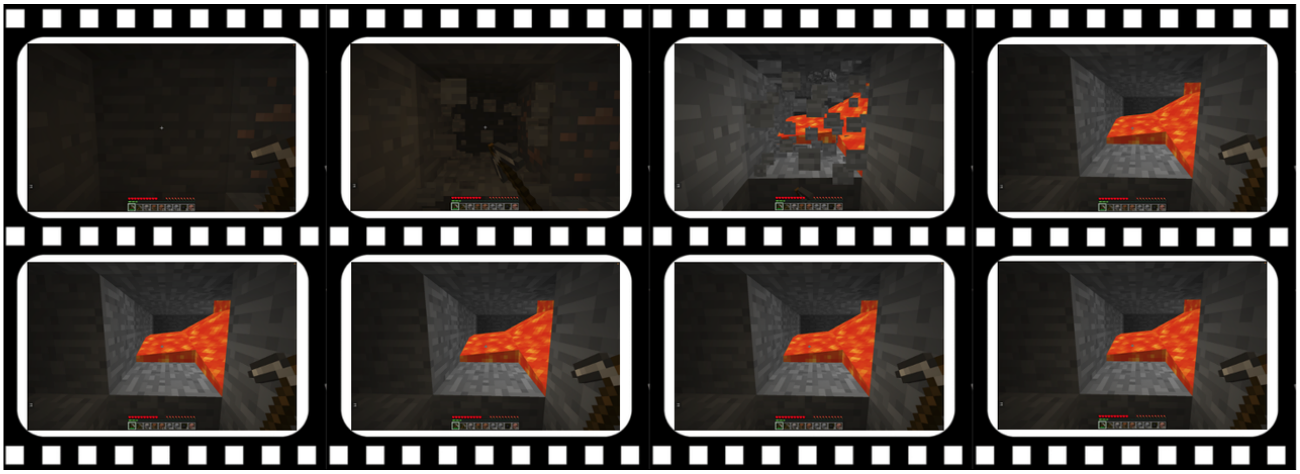

Figure III. The last few frames of trajectory images of failed diamond mining.

GPT-4V INPUT:

This is a Minecraft game video clip. Please describe the scene and the player's actions in detail. Note that describe the entire sequence of images rather than describing each image individually.

GPT-4V OUTPUT:

The sequence of images depicts a typical scenario in the game Minecraft where a player is mining underground. Initially, the player is in a dimly lit area, facing a wall of stone blocks, indicating they are inside a tunnel or a cave. The player's health bar shows a slight reduction in health, and the inventory is equipped with essential items for mining, such as pickaxes and cobblestone blocks.

As the player begins to mine through the stone with the pickaxe, they make progress with each swing, as indicated by the developing cracks in the stone block. This action is part of the core gameplay in Minecraft, where players gather resources by mining various types of blocks.

Suddenly, the player breaks through to an unexpected hazard a flow of bright orange lava appears behind the mined block. Lava is dangerous in Minecraft; it can cause damage to the player and destroy items. The player quickly steps back to avoid getting burned.

In the next moments, the player cautiously assesses the situation by looking at the lava from different angles, probably searching for a safe way to navigate around or contain it. The subsequent images indicate that the player is trying to deal with the lava flow, which is a common and risky part of mining deep underground in Minecraft.

Throughout the sequence, the player remains close to the lava, suggesting they may be attempting to block it with the cobblestone from their inventory or create a barrier to prevent the lava from spreading. This series of images captures the unpredictability and the survival aspects of Minecraft gameplay, where players must quickly adapt to the challenges they encounter while exploring and gathering resources.

ANALYZER INPUT:

You are now a proficient Minecraft player. I have trained an RL agent and tested for several times in the Minecraft game environment. I will give you a description of the player's scene and behavior at the end of the failure in text form.

Task description

- Objective: Find and approach diamond with higher success rate and avoid death.

- Initial Status:
  1. The agent is under ground at y level 11, the most common level to find diamonds.
  2. The agent already has an iron pickaxe.
- Success criteria: The distance to the nearest diamond block is less than 2 meter.

- Procedure: Explore horizontally to find a diamond, face it and approach it.

General game information

- The version of Minecraft is 1.11.

- Common block names : dirt, cobblestone, iron, diamond, wood, coal, water, air, lava, leaves, ... Collected item names are the same.

- FOV is set to -35 to 35 degrees for yaw and -30 to 30 degrees for pitch. The max visible distance of blocks is 64.

- Length of each block is 1 meter.

- The y coordinate of the agent is the agent's height. The larger the y, the higher the agent. The ground level is around y = 63, but is not fixed.

- At each step, the agent selects one operation in is action space. The action space includes doing nothing (i.e. staying still), moving forward and backward, jumping and attacking. The action space also includes turning left, right, up and down.

- The attack range is 2 meters in front of the agent. The agent can move 0.2 meters in each step. The agent can turn 15 degrees in each step. Typically, the agent needs to stay still and attack for **60 successive steps** to break a block.

- The hunger value is always at the max level.

Description of the player's scene and behavior at the end of the failure

- The sequence of images depicts a typical scenario in the game Minecraft where a player is mining underground. Initially, the player is in a dimly lit area, facing a wall of stone blocks, indicating they are inside a tunnel or a cave. The player's health The bar shows a slight reduction in health, and the inventory is equipped with essential items for mining, such as pickaxes and cobblestone blocks.

- As the player begins to mine through the stone with the pickaxe, they make progress with each swing, as indicated by the developing cracks in the stone block. This action is part of the core gameplay in Minecraft, where players gather resources by mining various types of blocks.

- Suddenly, the player breaks through to an unexpected hazard a flow of bright orange lava appears behind the mined block. Lava is dangerous in Minecraft; it can cause damage to the player and destroy items. The player quickly steps back to avoid getting burned. - In the next moments, the player cautiously assesses the situation by looking at the lava from different angles, probably searching for a safe way to navigate around or contain it. The subsequent images indicate that the player is trying to deal with the lava flow, which is a common and risky part of mining deep underground in Minecraft.

- Throughout the sequence, the player remains close to the lava, suggesting they may be attempting to block it with the cobblestone from their inventory or create a barrier to prevent the lava from spreading. This series of images captures the unpredictability and the survival aspects of Minecraft gameplay, where players must quickly adapt to the challenges they encounter while exploring and gathering resources.

Suppose the player dies after going through the above scenarios and actions. You should help me write the most likely reason of failure, and the most feasible suggestion on the game strategy based on the above reason.

ANALYZER OUTPUT:

Based on the description of the player's scene and behavior at the end of the failure, the most likely reason for failure appears to be the encounter with lava while mining. Lava is a significant hazard in Minecraft, especially when mining at deeper levels where diamonds are commonly found. The sudden appearance of lava behind a mined block can catch players off guard, leading to damage from burns or even death if they are unable to escape or protect themselves in time. The player's attempt to manage the lava flow suggests awareness of the danger but may have been insufficiently cautious or effective in containing or avoiding the lava.

Most Likely Reason of Failure:

The failure was likely due to the player's inability to safely navigate the sudden lava flow encountered during mining. Despite attempts to assess and manage the situation, the proximity to the lava and possibly inadequate measures to block or divert it might have resulted in sustained damage leading to death.

Suggestion on Game Strategy:

1. **Advance Preparedness**: Always carry water buckets in the inventory. A water bucket can be crucial for quickly converting lava into obsidian or stone, neutralizing the immediate threat. This quick response can save the player from burning and provides a safer path to continue mining.

2. **Strategic Mining Approach**: When mining at levels where lava is common, adopt a cautious approach:

- **Mine at a Safe Distance**: Instead of directly mining the block in front, mine while standing back or to the side to avoid sudden lava flows.

- **Use Blocks to Block Lava**: Have blocks ready in the hotbar to quickly place in front of a lava source. Cobblestone or dirt blocks can be used to block the flow or create barriers.

- **Listen for Lava Sounds**: Pay attention to audio cues. The sound of bubbling lava can alert the player to its presence behind or adjacent to blocks being mined.

3. **Environmental Awareness**: Before breaking a block, check the surroundings for signs of lava, such as light emanating from cracks or the sound of flowing lava. This can preemptively alert the player to potential dangers.

4. **Escape Plan**: Always have an escape route planned when mining deep underground. This means knowing the way back to safer areas or having a clear path to retreat if faced with hazards like lava.

5. **Practice Lava Management**: Familiarize with techniques to manage lava flows, such as placing blocks to divert the flow away from the mining area or using water to solidify lava. Practicing these techniques in safer conditions can prepare the player for real encounters.

By following these strategies, the player can significantly reduce the risk of death from lava while mining for diamonds and improve their overall success rate in finding valuable resources.