

A ADDITIONAL DETAILS ON THE METHODS AND EVALUATION METRICS

A.1 DETAILS ON THE CRAFTER SCORE METRIC

Crafter score measures how many of the following 22 achievements were obtained within the episode: “Collect Wood”, “Place Table”, “Eat Cow”, “Collect Sapling”, “Collect Drink”, “Make Wood Pickaxe”, “Make Wood Sword”, “Place Plant”, “Defeat Zombie”, “Collect Stone”, “Eat Plant”, “Defeat Skeleton”, “Make Stone Pickaxe”, “Make Stone Sword”, “Wake Up”, “Place Furnace”, “Collect Coal”, “Collect Iron”, “Collect Diamond”, “Make Iron Pickaxe”, “Make Iron Sword” (Hafner, 2021). Notably, some of the achievement are required to reach other achievement, for example “Place Plant” cannot be obtained unless “Collect Sapling” was already obtained.

A.2 INTERESTINGNESS SCORE

LLM-as-a-judge setup is used to judge the interest of generated goals using the following prompt and greedy sampling:

You are an expert game-design judge for Minecraft-like environments.
Rate each proposed goal on interestingness on a scale 0 10 (10 = highly interesting, novel, engaging; 0 = trivial or nonsensical).
Consider novelty, clarity, multi-step reasoning, and potential for skillful play.
Return a JSON object with ‘scores’: a list of numbers aligned to the task order. No extra commentary.

A.3 FILTERING INVALID GOALS

Before adding a goal generated by the LLM to the archive we are testing its validity by:

- Checking if the reward function is syntactically correct by executing its code
- Checking if the name of the reward state, the goal and subgoal names and the reward function is consistent.
- Checking if the reward state and the reward function is jit compatible by trying to compile and execute on a random input.

B HYPERPARAMETERS

B.1 PPO AGENT

We re-used the hyperparameters of the JAX implementation of PPO with the transformerXL architecture (Hamon, 2024)

Hyperparameter	Value
Learning Rate	2e-4
Batch Size	8
Number of Epochs	4
Clip Range	0.2
Discount Factor (Gamma)	0.999
GAE Lambda	0.8
Entropy Coefficient	2e-3
Value Function Coefficient	0.5
Max Gradient Norm	1.
Number of Layers	2
Number of Heads	8
Size hidden layers	256
QKV features	256
Window memory	128
Window gradient	64
Size embedding	256
max time step per rollout	100

Table 1: PPO transformerXL Hyperparameters

B.2 AUTOTELIC ARCHITECTURE HYPERPARAMETERS

Hyperparameter	Value
Update per generation K	500
Samples goals for training $\#S$	200
Number of steps per rollout p	100
Number of initial goal seed	36
Max size of the archive N	200

Table 2: Autotelic architecture hyperparameters

Overall the autotelic agent is tested on $128 \times 1024 \times 7 \times 150 = 16800000$ steps

C EXAMPLE OF REWARD FUNCTIONS GENERATED BY THE LLM

Here we present some examples of reward functions generated by the LLM.

The structure is the following:

- the reward state is defined with a class
- the reward function is a function
- the goal and subgoal names are stored in a list

[<name goal>, <name sugboal 1>, <name subgoal 2>, ...]

Craft and use wood swords:

```
@struct.dataclass
class RewardState_craft_and_use_wood_sword:
    counter_prompt: int = 0

def craft_and_use_wood_sword(state, action, rng, reward_state):
    """
    Check the sequenced goal that includes the sequence of subgoals: collect_wood, make_wood_sword, and
    defeat_monster.
    """
    # first subtask: collect wood
    subtask_1_achieved = state.inventory.wood >= 2
```

```

756 counter_updated = jax.lax.select(jnp.logical_and(subtask_1_achieved, reward_state.counter_prompt == 0), 1,
757 reward_state.counter_prompt)
758
759 # second subtask: make wood sword
759 subtask_2_achieved = state.inventory.wood_sword == 1
760 counter_updated = jax.lax.select(jnp.logical_and(subtask_2_achieved, reward_state.counter_prompt == 1), 2,
761 counter_updated)
762
763 # third subtask: defeat monster
762 monster_defeated = jnp.any(state.zombies.health <= 0)
763 counter_updated = jax.lax.select(jnp.logical_and(monster_defeated, reward_state.counter_prompt == 2), 3,
764 counter_updated)
765
766 task_achieved = counter_updated == 3
767
768 reward_state = reward_state.replace(counter_prompt=counter_updated)
769
770 return task_achieved, reward_state, reward_state.counter_prompt
771
772 prompt_sequence_craft_and_use_wood_sword = ['collect wood', 'make wood sword', 'defeat monster']

```

Build shelter:

```

771 @struct.dataclass
772 class RewardState_build_shelter:
773     has_placed_enough_blocks: bool = False
774     counter_prompt: int = 0
775
776 def build_shelter(state, action, rng, reward_state):
777     """
778     Check whether the agent builds a shelter using available resources (wood, stone) within a defined area.
779     """
780     def helper_in_bounds(state, position):
781         in_bounds_x = jnp.logical_and(0 <= position[0], position[0] < state.map.shape[0])
782         in_bounds_y = jnp.logical_and(0 <= position[1], position[1] < state.map.shape[1])
783         return jnp.logical_and(in_bounds_x, in_bounds_y)
784
785     # Define the shelter area (3x3 area around the player)
786     shelter_area = jnp.array([
787         [-1, -1], [-1, 0], [-1, 1],
788         [0, -1], [0, 0], [0, 1],
789         [1, -1], [1, 0], [1, 1]
790     ], dtype=jnp.int32)
791
792     # Check if the agent has placed blocks in the shelter area
793     def check_placed_blocks(unused, loc_add):
794         pos = state.player_position + loc_add
795         is_in_bounds = helper_in_bounds(state, pos)
796         is_placed = jnp.logical_and(is_in_bounds, state.map[pos[0], pos[1]] != BlockType.GRASS.value)
797         return None, is_placed
798
799     _, is_placed_blocks = jax.lax.scan(check_placed_blocks, None, shelter_area)
800
801     # Check if the agent has placed at least 5 blocks in the shelter area
802     has_placed_enough_blocks = is_placed_blocks.sum() >= 5
803
804     # Update the reward state
805     reward_state = reward_state.replace(has_placed_enough_blocks=has_placed_enough_blocks, counter_prompt=jax.
806 lax.select(has_placed_enough_blocks, 1, 0))
807
808     return has_placed_enough_blocks, reward_state, reward_state.counter_prompt
809
810 prompt_sequence_build_shelter = ['gather resources', 'place blocks', 'form shelter']

```

Build fortified shelter

```

811 @struct.dataclass
812 class RewardState_build_fortified_shelter:
813     has_placed_enough_blocks: bool = False
814     counter_prompt: int = 0
815
816 def build_fortified_shelter(state, action, rng, reward_state):
817     """
818     Check whether the agent builds a fortified shelter using stone and iron blocks, ensuring it is well-
819     protected.
820     """
821     # Define the shelter area (5x5 around the player)
822     shelter_area = jnp.array([
823         [-2, -2], [-2, -1], [-2, 0], [-2, 1], [-2, 2],
824         [-1, -2], [-1, -1], [-1, 0], [-1, 1], [-1, 2],
825         [0, -2], [0, -1], [0, 0], [0, 1], [0, 2],
826         [1, -2], [1, -1], [1, 0], [1, 1], [1, 2],
827         [2, -2], [2, -1], [2, 0], [2, 1], [2, 2]
828     ], dtype=jnp.int32)
829
830     # Check if the agent has placed blocks in the shelter area
831     def check_placed_blocks(unused, loc_add):
832         pos = state.player_position + loc_add

```

```

810     is_in_bounds = jnp.logical_and(0 <= pos[0], pos[0] < state.map.shape[0])
811     is_in_bounds = jnp.logical_and(is_in_bounds, jnp.logical_and(0 <= pos[1], pos[1] < state.map.shape[1])
812     )
813     is_placed = jnp.logical_and(is_in_bounds, jnp.logical_or(state.map[pos[0], pos[1]] == BlockType.STONE.
814     value, state.map[pos[0], pos[1]] == BlockType.IRON.value))
815     return None, is_placed
816
817     _, is_placed_blocks = jax.lax.scan(check_placed_blocks, None, shelter_area)
818
819     # Check if the agent has placed at least 10 blocks in the shelter area
820     has_placed_enough_blocks = is_placed_blocks.sum() >= 10
821
822     # Update the reward state
823     reward_state = reward_state.replace(has_placed_enough_blocks=has_placed_enough_blocks, counter_prompt=jax.
824     lax.select(has_placed_enough_blocks, 1, 0))
825
826     return has_placed_enough_blocks, reward_state, reward_state.counter_prompt
827
828     prompt_sequence_build_fortified_shelter = ['gather stone', 'gather iron', 'place stone blocks', 'place iron
829     blocks', 'form a 5x5 shelter']

```

Build mob trap:

```

828 @struct.dataclass
829 class RewardState_build_mob_trap:
830     task_achieved: bool = False
831     counter_prompt: int = 0
832
833 def build_mob_trap(state, action, rng, reward_state):
834     """
835     Check whether the agent builds a trap to catch mobs by strategically placing blocks and using resources.
836     """
837
838     # Define the trap area (3x3 around the player)
839     trap_area = jnp.array([
840         [-1, -1], [-1, 0], [-1, 1],
841         [0, -1], [0, 0], [0, 1],
842         [1, -1], [1, 0], [1, 1]
843     ], dtype=jnp.int32)
844
845     # Check if the agent has placed blocks in the trap area
846     def check_placed_blocks(unused, loc_add):
847         pos = state.player_position + loc_add
848         is_in_bounds = jnp.logical_and(0 <= pos[0], pos[0] < state.map.shape[0])
849         is_in_bounds = jnp.logical_and(is_in_bounds, jnp.logical_and(0 <= pos[1], pos[1] < state.map.shape[1])
850         )
851         is_placed = jnp.logical_and(is_in_bounds, state.map[pos[0], pos[1]] != BlockType.GRASS.value)
852         return None, is_placed
853
854     _, is_placed_blocks = jax.lax.scan(check_placed_blocks, None, trap_area)
855
856     # Check if the agent has placed at least 5 blocks in the trap area
857     has_placed_enough_blocks = is_placed_blocks.sum() >= 5
858
859     # Check if any mobs are trapped in the area
860     def check_trapped_mobs(unused, loc_add):
861         pos = state.player_position + loc_add
862         is_in_bounds = jnp.logical_and(0 <= pos[0], pos[0] < state.map.shape[0])
863         is_in_bounds = jnp.logical_and(is_in_bounds, jnp.logical_and(0 <= pos[1], pos[1] < state.map.shape[1])
864         )
865         is_mob = jnp.logical_and(is_in_bounds, state.mob_map[pos[0], pos[1]])
866         return None, is_mob
867
868     _, is_trapped_mobs = jax.lax.scan(check_trapped_mobs, None, trap_area)
869
870     # Check if there is at least one mob trapped
871     has_trapped_mobs = is_trapped_mobs.sum() > 0
872
873     # Task achievement is based on having placed enough blocks and trapping mobs
874     task_achieved = jnp.logical_and(has_placed_enough_blocks, has_trapped_mobs)
875
876     # Update the reward state
877     reward_state = reward_state.replace(task_achieved=task_achieved, counter_prompt=jax.lax.select(
878     task_achieved, 1, 0))
879
880     # Reward is given when the task is achieved
881     reward = jax.lax.select(task_achieved, 1., 0.)
882
883     return task_achieved, reward_state, reward_state.counter_prompt
884
885     prompt_sequence_build_mob_trap = ['gather resources', 'place blocks strategically', 'trap mobs']

```

D PROMPTS FOR THE LLM

D.1 GOAL CATEGORIZATION PROMPTS

LLM goal categorization is separated into two calls to the LLM. The first call receives 85 goals and is tasked with identifying 5 categories into which those goals can be separated. This step can be seen as clustering the archive. The prompt used for that call is as follows:

```
# Task Categorization for AI Agents in Crafter Environment

You are an AI expert tasked with creating a more nuanced categorization system for tasks in the Crafter
environment. The Crafter environment is a 2D open-world game where AI agents can perform various
activities like gathering resources, crafting tools, building structures, and interacting with entities.

## Your Task

1. Create {nbr_categories} new, more descriptive categories for classifying AI agent tasks in Crafter
2. Your categories should:
  - Capture different dimensions of complexity or skill requirements
  - Help in creating progressive learning curricula for AI agents
  - Consider both the cognitive and procedural aspects of tasks
3. After creating your categories, classify EVERY SINGLE task below (task indices 1 through {len(
  in_context_goals)}) into your new system using a dictionary format.

## Task Lists (Total: {len(in_context_goals)} tasks to classify)
"""
<tasks: python function name, index, task name, subgoals>
"""

## Response Format
Return ONLY a JSON object with this exact structure:
'''
{
  "categories": [
    {
      "name": "Category1Name",
      "definition": "Definition of category 1",
      "example_tasks": ["task_idx_1", "task_idx_2"] # One or Two Example tasks that fit this category
    },
    {
      "name": "Category2Name",
      "definition": "Definition of category 2",
      "example_tasks": ["task_idx_3", "task_idx_4"] # One or Two Example tasks that fit this category
    }
  ],
  "task_classifications": {
    "1": "Category1Name",
    "2": "Category2Name",
    "3": "Category1Name",
    ...
    "{len(in_context_goals)}": "CategoryXName"
  }
}
'''

CRITICAL REQUIREMENTS:
- You MUST classify ALL {len(in_context_goals)} tasks (task indices 1 through {len(in_context_goals)})
- Every task index from 1 to {len(in_context_goals)} must appear as a key in the task_classifications
  dictionary
- Each task can only be assigned to ONE category
- All category names in task_classifications must match exactly with the category names in the categories list
- Do not skip any task indices - the range is [1, 2, 3, ..., {len(in_context_goals)}]
- No category should be left empty without tasks assigned

Your entire response must be this JSON object only. Do not include markdown formatting or any explanatory text
outside the JSON.
Your categories must provide a framework for designing progressive curriculum learning that captures different
dimensions of task complexity better than the original categories.
```

The second call is tasked with assigning the remaining goals from the archive to the previously outlined categories. This step can be seen as classifying the goals into those categories. This is necessary in practice because the LLM often failed to cluster the entirety of the 200 goals in the archive on the first attempt. The prompt used for that call is as follows:

```
# Classify Tasks into Predefined Categories (Crafter)

You are given a fixed set of categories (names and definitions). Do NOT create new categories.
Classify EVERY SINGLE task below (task indices 1 through {len(task_indices)}) into exactly one of these
categories.

## Categories (Fixed)
"""
<categories>
"""

## Task List (Total: {len(task_indices)})
```

```

"""
<tasks: python function name, index, task name, subgoals>
"""
## Response Format
Return ONLY a JSON object with this exact structure:
'''
{{
  "task_classifications": {{
    "1": "CategoryName",
    "2": "CategoryName",
    ...,
    "{len(task_indices)}": "CategoryName"
  }}
}}
'''

CRITICAL REQUIREMENTS:
- You MUST classify ALL {len(task_indices)} tasks (task indices 1 through {len(task_indices)})
- Use ONLY the provided category names (must match exactly)
- Each task can only be assigned to ONE category
- Do not skip any task indices - the range is [1, 2, 3, ..., {len(task_indices)}]

Your entire response must be this JSON object only. Do not include markdown formatting or any explanatory text
outside the JSON.

```

D.2 GOAL GENERATION PROMPTS

Just like goal categorization, LLM goal generation is similarly separated into two calls to the LLM. The first call generates goal descriptions and the pseudocode for the reward function, while the second call generates the code of the reward function in jax.

The prompt used to generate the pseudocode is:

```

# Designing Reward Functions for AI Agents in Crafter

You are designing reward functions to train an AI agent in the Crafter environment. Your task is to create
novel, engaging tasks that progressively build the agent's capabilities through curriculum learning.
Each new task should be carefully calibrated - challenging enough to teach new skills while remaining
achievable based on the agent's current abilities.

Crafter is a 2D open-world game in which the agents can perform various tasks, such as foraging for food,
finding water, building tools and constructions, performing motion patterns, and defending against
monsters.

The documents provide:
1. Core requirements for good task design
2. Technical specifications of the Crafter environment
3. Reference examples of successful and unsuccessful tasks
4. Templates for submitting new task proposals

# 1. Task Design Requirements
The tasks should be designed by taking inspiration from the already tried tasks (cf. 4. Reference Examples).

## 1.1 Learnability
- Must be achievable given the agent's current capabilities
- Should be more challenging than mastered tasks (cf. 4. Reference Examples) but not overwhelmingly difficult
- Should have clear prerequisites and gradual skill progression
- It must be technically possible within environmental constraints.

## 1.2 Novelty and Interest
- Creates new challenges not covered by existing tasks
- Combines existing skills in creative ways
- Provides engaging scenarios (e.g., strategic resource gathering, clever tool use)
- Has clear success conditions that feel rewarding to achieve

## 1.3 Task Submission Format
For the description:
- Create multiple goals only if the previous are not precondition for the next one. When there's multiple
goals, it's because you want to create a sequence of goals that are not precondition for each other.
Example of single goal: Mine stone
1. main goal 1: Mine stone # all the subgoals are preconditions for this goal
  - subgoal 1: Craft a wooden pickaxe
    - completion condition: the agent has a new wooden pickaxe in the inventory
  - subgoal 2: Mine stone
    - completion condition: the agent has increased the stone in the inventory

Example of sequence of goals: Mine stone and go to the top of the map. The two goals in the sequence are not
preconditions for each other, the difficulty comes from the combination of the two goals.
1. main goal 1: Mine stone # all the subgoals are preconditions for this goal
  - subgoal 1: Craft a wooden pickaxe
    - completion condition: the agent has a wooden pickaxe in the inventory
  - subgoal 2: Mine stone
    - completion condition: the agent has increased the stone in the inventory
2. main goal 2: Go to the top of the map
  - subgoal 1: Go to the bottom of the map
    - completion condition: the agent has reached the bottom of the map

```

Whenever the goals you are creating is about combining that are not directly dependent on each other, you should create a sequence of goals. For example alternating between goals should be split into different goals.

- Only do multiple goals is the category is "Sequence of goals".
- The number of maximum subgoals is 10.
- Completion conditions should be precise and directly implementable in the environment. Don't use vague term like build a shelter or a bridge, but mention how would you test the condition.

1.5 Goal Categories

When designing a new task, you should categorize the task based on the type of goal the agent has to achieve. The categories are:

The categories are:

- Crafting tools
- Combat
- Collecting/Finding Things
- Construction (shelter, bridges)
- Navigation/Movement
- Sequence of goals (multiple goals that are not preconditions for each other): combination of goals from the other categories in original ways.

The complexity of the task is based on:

- The number of subgoals
- The objects required (wood, stone, iron, diamond)
- The constraints (e.g. time, resources, health conditions)
- The complexity of the subgoals: whether the agents has to follow complex instructions (complex patterns) or has to figure out the subgoals by itself.

1.6 Crafter Environment Overview

The Crafter Environment follow the following game logic:

- The player can only place blocks things that have a corresponding action in the Action enum (see section 2.1: place stone, place table, place furnace, place plant). The agent cannot place wood or water blocks.

2. Implementation Details of the Crafter Environment

2.1 Environment State Specification

The world-state is updated based on a state representation ("state") that is updated after each action of the agent by the following game logic. The state has the following structure:

```

python
# EnvState class to represent the entire game state, instantiated at the beginning of the game in the variable
"state"
class EnvState:
    map: jnp.ndarray = map # 64x64 grid representing the whole world
    mob_map: jnp.ndarray = jnp.zeros((64, 64), dtype=bool) # 64x64 grid representing the presence of mobs

    player_position: jnp.ndarray = player_position # 2D position of the player on the map that is used as
    index for the map
    player_direction: int = Action.UP.value

    # Player's vital stats (health, food, water, energy at the beginning of the game) at the beginning of the
    game
    player_health: int = 9 # Player's health is going down when attacked by mobs and goes directly to 0 if the
    player goes into the lava

    inventory: Inventory
    zombies: Mobs
    cows: Mobs
    skeletons: Mobs
    arrows: Mobs
    arrow_directions: jnp.ndarray

    # Tracking planted crops
    growing_plants_positions: jnp.ndarray
    growing_plants_age: jnp.ndarray
    growing_plants_mask: jnp.ndarray

# Inventory class to track player's collected resources and crafted tools
class Inventory:
    wood: int = 0
    stone: int = 0
    coal: int = 0
    iron: int = 0
    diamond: int = 0
    sapling: int = 0
    wood_pickaxe: int = 0
    stone_pickaxe: int = 0
    iron_pickaxe: int = 0
    wood_sword: int = 0
    stone_sword: int = 0
    iron_sword: int = 0

# Mobs class to represent creatures (zombies, cows, skeletons) in the game
class Mobs:
    position: jnp.ndarray = jnp.zeros((static_params.max_mobs, 2), dtype=jnp.int32) # (n, 2) array of mob
    positions (max_mobs is different for each mob type)
    health: jnp.ndarray = jnp.ones(static_params.max_mobs, dtype=jnp.int32) * params.mob_health # (n,) array
    of mob health (max_mobs and mob_health is different for each mob type)
    mask: jnp.ndarray = jnp.zeros(static_params.max_mobs, dtype=bool) # (n,) array of mob that are alive or
    dead (max_mobs is different for each mob type)

```

```

1026     attack_cooldown: jnp.ndarray # (n,) counter for mob attack cooldown (max_mobs is different for each mob
1027         type). When the counter reaches 0, the mob can attack again.
1028     just_died: jnp.zeros(static_params.max_mobs, dtype=bool) # (n,) array of mob that have just died (
1029         max_mobs is different for each mob type)
1030
1031 # Enum for different block types in the game world that are used to represent the map
1032 class BlockType(Enum):
1033     INVALID = 0
1034     OUT_OF_BOUNDS = 1
1035     GRASS = 2
1036     WATER = 3
1037     STONE = 4
1038     TREE = 5
1039     WOOD = 6
1040     PATH = 7 # Path is a block that the player can walk on. When the player collects a block of water, stone,
1041         diamond or iron, it becomes a path.
1042     COAL = 8
1043     IRON = 9
1044     DIAMOND = 10
1045     CRAFTING_TABLE = 11
1046     FURNACE = 12
1047     SAND = 13
1048     LAVA = 14
1049     PLANT = 15
1050     RIPE_PLANT = 16
1051
1052 # Blocks that the player cannot walk through (e.g the player position cannot be on one of these blocks)
1053 SOLID_BLOCKS = jnp.array(
1054     [
1055         BlockType.WATER.value,
1056         BlockType.STONE.value,
1057         BlockType.TREE.value,
1058         BlockType.COAL.value,
1059         BlockType.IRON.value,
1060         BlockType.DIAMOND.value,
1061         BlockType.CRAFTING_TABLE.value,
1062         BlockType.FURNACE.value,
1063         BlockType.PLANT.value,
1064         BlockType.RIPE_PLANT.value,
1065     ],
1066     dtype=jnp.int32,
1067 )
1068
1069 # Directions for movement and interaction
1070 DIRECTIONS = jnp.concatenate(
1071     (
1072         jnp.array([[0, 0], # no movement
1073                    [0, -1], # go left
1074                    [0, 1], # go right
1075                    [-1, 0], # go up
1076                    [1, 0]], # go down
1077                dtype=jnp.int32),
1078         jnp.zeros((11, 2), dtype=jnp.int32),
1079     ),
1080     axis=0,
1081 )
1082
1083 # Adjacent blocks for interaction
1084 CLOSE_BLOCKS = jnp.array(
1085     [
1086         [0, -1], # left
1087         [0, 1], # right
1088         [-1, 0], # up
1089         [1, 0], # down
1090         [-1, -1], # up-left
1091         [-1, 1], # up-right
1092         [1, -1], # down-left
1093         [1, 1], # down-right
1094     ],
1095     dtype=jnp.int32,
1096 )
1097
1098 # Enum for possible actions the player can take
1099 class Action(Enum):
1100     NOOP = 0
1101     LEFT = 1
1102     RIGHT = 2
1103     UP = 3
1104     DOWN = 4
1105     DO = 5 # DO is a general action that can be used for various tasks: mine, attack and drink water.
1106     SLEEP = 6
1107     PLACE_STONE = 7
1108     PLACE_TABLE = 8
1109     PLACE_FURNACE = 9
1110     PLACE_PLANT = 10
1111     MAKE_WOOD_PICKAXE = 11
1112     MAKE_STONE_PICKAXE = 12
1113     MAKE_IRON_PICKAXE = 13
1114     MAKE_WOOD_SWORD = 14
1115     MAKE_STONE_SWORD = 15
1116     MAKE_IRON_SWORD = 16
1117 ...

```



```

Here are some additional details about the environment state:
'''python
# Define the observation dimensions
# GAME CONSTANTS
OBS_DIM = (7, 9)
MAX_OBS_DIM = max(OBS_DIM)
assert OBS_DIM[0] % 2 == 1 and OBS_DIM[1] % 2 == 1

'''

## 2.2 Rules of the Crafter Environment
Placing blocks:
- The player can only place blocks that have a corresponding action in the Action enum (see section 2.1: place
  stone, place table, place furnace, place plant). The agent cannot place wood or water blocks.
- The player can only place blocks on the map that are not solid blocks (see section 2.1: SOLID_BLOCKS).

Rules to craft tools:
### pickaxe
- craft a wood pickaxe:
  - Action to do: MAKE_WOOD_PICKAXE
  - Inventory requirements: 1 wood
  - Block requirements: 1 crafting table is in CLOSE_BLOCKS
- craft a stone pickaxe:
  - Action to do: MAKE_STONE_PICKAXE
  - Inventory requirements: 1 stone, 1 wood
  - Block requirements: 1 crafting table is in CLOSE_BLOCKS
- craft an iron pickaxe:
  - Action to do: MAKE_IRON_PICKAXE
  - Inventory requirements: 1 iron, 1 coal, 1 stone
  - Block requirements: 1 crafting table and 1 furnace are in CLOSE_BLOCKS

### sword
- craft a wood sword:
  - Action to do: MAKE_WOOD_SWORD
  - Inventory requirements: 1 wood
  - Block requirements: 1 crafting table is in CLOSE_BLOCKS
- craft a stone sword:
  - Action to do: MAKE_STONE_SWORD
  - Inventory requirements: 1 stone, 1 wood
  - Block requirements: 1 crafting table is in CLOSE_BLOCKS
- craft an iron sword:
  - Action to do: MAKE_IRON_SWORD
  - Inventory requirements: 1 iron, 1 coal, 1 stone
  - Block requirements: 1 crafting table and 1 furnace are in CLOSE_BLOCKS

Rules to collect resources:
- collect wood:
  - Action to do: DO
  - Block requirements: 1 tree is ahead of the player
  - Inventory requirements: None
- collect stone:
  - Action to do: DO
  - Block requirements: 1 stone is ahead of the player
  - Inventory requirements: Have a wood pickaxe in the inventory
- collect coal:
  - Action to do: DO
  - Block requirements: 1 coal is ahead of the player
  - Inventory requirements: Have a wood pickaxe in the inventory
- collect iron:
  - Action to do: DO
  - Block requirements: 1 iron is ahead of the player
  - Inventory requirements: Have a stone pickaxe in the inventory
- collect diamond:
  - Action to do: DO
  - Block requirements: 1 diamond is ahead of the player
  - Inventory requirements: Have an iron pickaxe in the inventory

Rules to place things:
- place a stone block:
  - Action to do: PLACE_STONE
  - Block requirements: no solid block is ahead of the player
  - Inventory requirements: 1 stone is in the inventory
- place a crafting table:
  - Action to do: PLACE_TABLE
  - Block requirements: no solid block is ahead of the player
  - Inventory requirements: 1 wood is in the inventory
- place a furnace:
  - Action to do: PLACE_FURNACE
  - Block requirements: no solid block is ahead of the player
  - Inventory requirements: 1 stone is in the inventory
- place a plant:
  - Action to do: PLACE_PLANT
  - Block requirements: no solid block is ahead of the player
  - Inventory requirements: 1 sapling is in the inventory
  - Note: the plant will grow and become a plant block after a certain amount of time

## 2.3 Useful Functions to Implement the Reward Functions
Here are some useful functions that you can use to implement the reward functions:
'''python
def get_agent_view(state):
    """

```

```

1134     Get the agent's view of the map, centered around the agent's position.
1135
1136     Args:
1137         state (EnvState): The current state of the environment.
1138
1139     Returns:
1140         jnp.ndarray: The agent's view of the map with dimensions (7, 9). The agent's position is at
1141         map_view[3, 4].
1142     """
1143     # Convert the observation dimensions to a JAX array for easier manipulation
1144     obs_dim_array = jnp.array([OBS_DIM[0], OBS_DIM[1]], dtype=jnp.int32)
1145
1146     # Pad the map to ensure the view is within bounds, even if the agent is near the edge
1147     # We pad by MAX_OBS_DIM + 2 to ensure there's enough padding on all sides
1148     padded_grid = jnp.pad(
1149         state.map,
1150         (MAX_OBS_DIM + 2, MAX_OBS_DIM + 2),
1151         constant_values=BlockType.OUT_OF_BOUNDS.value,
1152     )
1153
1154     # Calculate the top-left corner of the view
1155     # We adjust the player's position by subtracting half the observation dimensions
1156     # and then adding the padding amount (MAX_OBS_DIM + 2) to account for the padding we added
1157     tl_corner = state.player_position - obs_dim_array // 2 + MAX_OBS_DIM + 2
1158
1159     # Extract the agent's view from the padded grid using dynamic slicing
1160     # This ensures that the view is always a 7x9 grid centered around the agent's position
1161     map_view = jax.lax.dynamic_slice(padded_grid, tl_corner, OBS_DIM)
1162     return map_view
1163
1164 # Example usage
1165 map_view = get_agent_view(state)
1166
1167 def is_mob_nearby(state, neighborhood_size=jnp.array([OBS_DIM[0], OBS_DIM[1]], dtype=jnp.int32)):
1168     """
1169     Return the number and type of mobs in the agent's neighborhood. By default, the neighborhood is the agent's
1170     view (7x9).
1171
1172     Args:
1173         state (EnvState): The current state of the environment.
1174         neighborhood_size (jnp.array): The size of the neighborhood around the agent.
1175
1176     Returns:
1177         tuple: A tuple containing the count of skeletons, cows, and zombies in the neighborhood.
1178     """
1179
1180     # Boundary of the neighborhood
1181     tl_corner = state.player_position - neighborhood_size // 2
1182     br_corner = state.player_position + neighborhood_size // 2
1183
1184     # Helper function to count mobs within the neighborhood
1185     def count_mobs(mob_positions, mob_mask, tl_corner, br_corner):
1186         # Check if the mob is within the neighborhood bounds for each dimension
1187         in_neighborhood = jnp.logical_and(
1188             jnp.logical_and(mob_positions[:, 0] >= tl_corner[0], mob_positions[:, 0] <= br_corner[0]),
1189             jnp.logical_and(mob_positions[:, 1] >= tl_corner[1], mob_positions[:, 1] <= br_corner[1])
1190         )
1191
1192         # Combine the neighborhood check with the mask to count only active mobs
1193         active_in_neighborhood = jnp.logical_and(in_neighborhood, mob_mask)
1194
1195         # Count the number of active mobs in the neighborhood
1196         return jnp.sum(active_in_neighborhood)
1197
1198     # Count skeletons
1199     skeleton_count = count_mobs(state.skeletons.position, state.skeletons.mask, tl_corner, br_corner)
1200
1201     # Count cows
1202     cow_count = count_mobs(state.cows.position, state.cows.mask, tl_corner, br_corner)
1203
1204     # Count zombies
1205     zombie_count = count_mobs(state.zombies.position, state.zombies.mask, tl_corner, br_corner)
1206
1207     return skeleton_count, cow_count, zombie_count
1208
1209 # Example usage
1210 skeleton_count, cow_count, zombie_count = is_mob_nearby(env_state, neighborhood_size=jnp.array([17, 17], dtype=jnp.int32))
1211
1212 def block_pattern_matching(
1213     map_: jnp.ndarray,
1214     pattern: jnp.ndarray,
1215     block_type: int,
1216     player_position: jnp.ndarray
1217 ) -> bool:
1218     """
1219     Checks if 'pattern' matches 'block_type' around 'player_position' in 'map_'.
1220
1221     Where pattern == 1, map_ must be block_type.

```

```

Where pattern == 0, we don't care.

Both 'map_' and 'pattern' must have statically known shapes if you
want to JIT this function without errors.
"""
# Shape info
H, W = map_.shape
P, Q = pattern.shape # must be fixed shape if you want to JIT

# Compute top-left corner in the map
# so that pattern is "centered" on the player
offset_row = player_position[0] - P // 2
offset_col = player_position[1] - Q // 2

# Check bounds: if we can't slice a full P,Q subarray, we return False
def in_bounds_fn(_):
    # Use lax.dynamic_slice to extract the sub-section of the map
    region = jax.lax.dynamic_slice(map_, (offset_row, offset_col), (P, Q))
    # region now has shape (P, Q)

    # Where pattern == 1, we require region == block_type
    matches = jnp.where(pattern == 1, region == block_type, True)
    return jnp.all(matches)

# Return False if out-of-bounds
in_bounds = (
    (offset_row >= 0) & (offset_col >= 0) &
    ((offset_row + P) <= H) & ((offset_col + Q) <= W)
)
return jax.lax.cond(in_bounds, in_bounds_fn, lambda _: False, operand=None)

# Example usage
pattern = jnp.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]], dtype=jnp.int32) # block on the left, right, up and
    down around the player
block_type = BlockType.STONE.value
player_position = jnp.array([10, 10], dtype=jnp.int32)
is_pattern_matched = block_pattern_matching(state.map, pattern, block_type, player_position)
'''

## 2.2 Bad practices to use in crafter

'''python
# Don't use state.map to check if the agent has placed a certain number of blocks, it contains blocks that are
    not placed by the agent
stone_blocks_placed = jnp.sum(state.map == BlockType.STONE.value) >= 5
# To check the number of blocks placed by the agent, you can use the decrease of blocks in the inventory or
    contrast the map with a map of the previous state that is saved in the reward state.
@struct.dataclass
class RewardState_build_very_basic_bridge:
    counter_prompt: int = 0
    previous_map: jnp.ndarray = field(default_factory=lambda: jnp.zeros((MAP_SIZE[0], MAP_SIZE[1]), dtype=jnp.
        int32))
# with the inventory
stone_placed = jnp.logical_and(state.inventory.stone < reward_state.previous_inventory.stone)
# with the map
block_ahead_position = state.player_position + DIRECTIONS[state.player_direction]
stone_placed = jnp.logical_and(state.map[block_ahead_position[0], block_ahead_position[1]] == BlockType.STONE.
    value, reward_state.previous_map[block_ahead_position[0], block_ahead_position[1]] != BlockType.STONE.
    value)
'''

# 3. Reference Examples of tasks
You should take inspiration from the reward functions below to create a new reward function that is
    interesting, novel, and achievable. For each reward function, a score between 0 and 100 is given to
    describe the difficulty based on the AI-agent experience in craftax. Don't re-use the same name of
    function as the ones in the reference examples. ## Learnable Tasks:
Below are some rewards that are learnable, you need to do tasks that are similar but still novel:

Task 1 (too easy): kill_two_cows
Current success rate for the agent on the following task : 0
Category: Combat
**Goal(s) decomposition**:
1. main goal 1: Kill two cows
    - subgoal 1: Kill a cow
        - completion condition: the agent has killed a cow
    - subgoal 2: Kill another cow
        - completion condition: the agent has killed a second cow

## Not learnable Tasks:
Below are some rewards that are not learnable, you need to do tasks that are more learnable:

# 4. Task Submission Format
When creating a task, you need to provide a detailed description of the task, including the goal, subgoals.
    Also be sure to not do the same tasks as the reference examples (section 3).

BE SURE TO FOLLOW THE TASK SUBMISSION FORMAT. The task submission format is as follows:

My proposed task is: \

```

```

**Goal reasoning**: <here, briefly reason about the next goal to create. Reason step by step before answering
the following questions: which goal is interesting to do now for the given category (it should not be
about multiple categories)? How is it different from all the previously tried tasks (section 3.)? Why is
it challenging (more difficult than the too-easy tasks)? Why is it achievable (easier than the too-
difficult ones)? Always mention the number of tasks you are building upon and how (be precise). >\
**Reason about subgoals**: <here based on the goal you just selected, reason about how the goal should be
decomposed into subgoals.>\
**Goal name**: <here mention the name of the goal. DO NOT RE-USE ONE OF THE TASK OF SECTION 3. Do not use
capital letters or spaces. Use underscores to separate words.>
**Goal(s) decomposition**: <here describe that main goals and subgoals in precise statements of the conditions
tested. Don't use general terms but be precise about how you want to test the condition. Subgoals are
for helping to achieve a goal. Use multiple goals for creating more complex goals (only in the case of '
Sequence of goals') by combining them when they are not preconditions for each other. Do not do more
than 10 subgoals in total.>\
1. main goal 1: description of the goal 1 single desired outcome
  - subgoal 1: description
    - completion condition
  - subgoal 2: description
    - completion condition
  - ...
  - subgoal n: description
    - completion condition
2. main goal 2: description of the goal 2 single desired outcome
  - subgoal n+1: description
    - completion condition
  - ...
>
**END**

--- end of the document ---

Now that you've taken into account the previous document create a new task from the category Combat.
My proposed task is:

```

The prompt used for generating the code is here:

```

# Designing Reward Functions for AI Agents in Crafter

You are designing reward functions to train an AI agent in the Crafter environment. Your task is to create
novel, engaging tasks that progressively build the agent's capabilities through curriculum learning.
Each new task should be carefully calibrated - challenging enough to teach new skills while remaining
achievable based on the agent's current abilities.

Crafter is a 2D open-world game in which the agents can perform various tasks, such as foraging for food,
finding water, building tools and constructions, performing motion patterns, and defending against
monsters.

The documents provide:
1. Core requirements for good task design
2. Technical specifications of the Crafter environment
3. Technical Implementation Guide
4. Reference examples code of successful and unsuccessful tasks
5. Templates for generating the code of a new task

# 1. Task Design Requirements
The tasks should be designed by taking inspiration from the already tried tasks (cf. 4. Reference Examples).

## 1.1 Learnability
- Must be achievable given the agent's current capabilities
- Should be more challenging than mastered tasks (cf. 4. Reference Examples) but not overwhelmingly difficult
- Should have clear prerequisites and gradual skill progression
- It must be technically possible within environmental constraints.

## 1.2 Novelty and Interest
- Creates new challenges not covered by existing tasks
- Combines existing skills in creative ways
- Provides engaging scenarios (e.g., strategic resource gathering, clever tool use)
- Has clear success conditions that feel rewarding to achieve

## 1.3 Crafter Environment Overview
The Crafter Environment follow the following game logic:
- The player can only place blocks things that have a corresponding action in the Action enum (see section
  2.1: place stone, place table, place furnace, place plant). The agent cannot place wood or water blocks.

## 1.4 Technical Implementation
- Must be well-defined with clear success criteria
- Must use available state information correctly
- Must be JAX/JIT compatible.
- Don't JIT anything; only write the code for the reward function and its associated reward state and prompt
  sequence.
- The only function you can re-use are the ones provided in the section 2.1 Useful Functions to Implement the
  Reward Functions. The rest of the code should be written from scratch.
- Pay attention to the state representation (EnvState and Inventory) and the game logic. The reward function
  should be compatible with the game logic and the EnvState.

# 2. Implementation Details of the Crafter Environment
## 2.1 Environment State Specification
The world-state is updated based on a state representation ("state") that is updated after each action of the
agent by the following game logic. The state has the following structure:

``python

```

```

1296 # EnvState class to represent the entire game state, instantiated at the beginning of the game in the variable
1297 "state"
1298 class EnvState:
1299     map: jnp.ndarray = map # 64x64 grid representing the whole world
1300     mob_map: jnp.ndarray = jnp.zeros((64, 64), dtype=bool) # 64x64 grid representing the presence of mobs
1301     player_position: jnp.ndarray = player_position # 2D position of the player on the map that is used as
1302         index for the map
1303     player_direction: int = Action.UP.value
1304     # Player's vital stats (health, food, water, energy at the beginning of the game) at the beginning of the
1305         game
1306     player_health: int = 9 # Player's health is going down when attacked by mobs and goes directly to 0 if the
1307         player goes into the lava
1308     # Rates of change for player's vital stats
1309     inventory: Inventory
1310     zombies: Mobs
1311     cows: Mobs
1312     skeletons: Mobs
1313     arrows: Mobs
1314     arrow_directions: jnp.ndarray
1315     # Tracking planted crops
1316     growing_plants_positions: jnp.ndarray
1317     growing_plants_age: jnp.ndarray
1318     growing_plants_mask: jnp.ndarray
1319 # Inventory class to track player's collected resources and crafted tools
1320 class Inventory:
1321     wood: int = 0
1322     stone: int = 0
1323     coal: int = 0
1324     iron: int = 0
1325     diamond: int = 0
1326     sapling: int = 0
1327     wood_pickaxe: int = 0
1328     stone_pickaxe: int = 0
1329     iron_pickaxe: int = 0
1330     wood_sword: int = 0
1331     stone_sword: int = 0
1332     iron_sword: int = 0
1333 # Mobs class to represent creatures (zombies, cows, skeletons) in the game
1334 class Mobs:
1335     position: jnp.ndarray = jnp.zeros((static_params.max_mobs, 2), dtype=jnp.int32) # (n, 2) array of mob
1336         positions (max_mobs is different for each mob type)
1337     health: jnp.ndarray = jnp.ones(static_params.max_mobs, dtype=jnp.int32) * params.mob_health # (n,) array
1338         of mob health (max_mobs and mob_health is different for each mob type)
1339     mask: jnp.ndarray = jnp.zeros(static_params.max_mobs, dtype=bool) # (n,) array of mob that are alive or
1340         dead (max_mobs is different for each mob type)
1341     attack_cooldown: jnp.ndarray # (n,) counter for mob attack cooldown (max_mobs is different for each mob
1342         type). When the counter reaches 0, the mob can attack again.
1343     just_died: jnp.zeros(static_params.max_mobs, dtype=bool) # (n,) array of mob that have just died (
1344         max_mobs is different for each mob type)
1345 # Enum for different block types in the game world that are used to represent the map
1346 class BlockType(Enum):
1347     INVALID = 0
1348     OUT_OF_BOUNDS = 1
1349     GRASS = 2
1350     WATER = 3
1351     STONE = 4
1352     TREE = 5
1353     WOOD = 6
1354     PATH = 7 # Path is a block that the player can walk on. When the player collects a block of water, stone,
1355         diamond or iron, it becomes a path.
1356     COAL = 8
1357     IRON = 9
1358     DIAMOND = 10
1359     CRAFTING_TABLE = 11
1360     FURNACE = 12
1361     SAND = 13
1362     LAVA = 14
1363     PLANT = 15
1364     RIPE_PLANT = 16
1365 # Blocks that the player cannot walk through (e.g the player position cannot be on one of these blocks)
1366 SOLID_BLOCKS = jnp.array(
1367     [
1368         BlockType.WATER.value,
1369         BlockType.STONE.value,
1370         BlockType.TREE.value,
1371         BlockType.COAL.value,
1372         BlockType.IRON.value,
1373         BlockType.DIAMOND.value,
1374         BlockType.CRAFTING_TABLE.value,
1375         BlockType.FURNACE.value,
1376         BlockType.PLANT.value,
1377         BlockType.RIPE_PLANT.value,
1378     ],

```

```

1350         dtype=jnp.int32,
1351     )
1352
1353     # Directions for movement and interaction
1354     DIRECTIONS = jnp.concatenate(
1355         (
1356             jnp.array([[0, 0], # no movement
1357                        [0, -1], # go left
1358                        [0, 1], # go right
1359                        [-1, 0], # go up
1360                        [1, 0], # go down
1361                        dtype=jnp.int32),
1362             jnp.zeros((11, 2), dtype=jnp.int32),
1363         ),
1364         axis=0,
1365     )
1366     # Adjacent blocks for interaction
1367     CLOSE_BLOCKS = jnp.array(
1368         [
1369             [0, -1], # left
1370             [0, 1], # right
1371             [-1, 0], # up
1372             [1, 0], # down
1373             [-1, -1], # up-left
1374             [-1, 1], # up-right
1375             [1, -1], # down-left
1376             [1, 1], # down-right
1377         ],
1378         dtype=jnp.int32,
1379     )
1380
1381     # Enum for possible actions the player can take
1382     class Action(Enum):
1383         NOOP = 0
1384         LEFT = 1
1385         RIGHT = 2
1386         UP = 3
1387         DOWN = 4
1388         DO = 5 # DO is a general action that can be used for various tasks: mine, attack and drink water.
1389         SLEEP = 6
1390         PLACE_STONE = 7
1391         PLACE_TABLE = 8
1392         PLACE_FURNACE = 9
1393         PLACE_PLANT = 10
1394         MAKE_WOOD_PICKAXE = 11
1395         MAKE_STONE_PICKAXE = 12
1396         MAKE_IRON_PICKAXE = 13
1397         MAKE_WOOD_SWORD = 14
1398         MAKE_STONE_SWORD = 15
1399         MAKE_IRON_SWORD = 16
1400
1401     ...
1402
1403     Here are some additional details about the environment state:
1404     ```python
1405     # Define the observation dimensions
1406     # GAME CONSTANTS
1407     OBS_DIM = (7, 9)
1408     MAX_OBS_DIM = max(OBS_DIM)
1409     assert OBS_DIM[0] % 2 == 1 and OBS_DIM[1] % 2 == 1
1410
1411     ...
1412
1413     ## 2.2 Rules of the Crafter Environment
1414     Placing blocks:
1415     - The player can only place blocks that have a corresponding action in the Action enum (see section 2.1: place
1416       stone, place table, place furnace, place plant). The agent cannot place wood or water blocks.
1417     - The player can only place blocks on the map that are not solid blocks (see section 2.1: SOLID_BLOCKS).
1418
1419     Rules to craft tools:
1420     ### pickaxe
1421     - craft a wood pickaxe:
1422         - Action to do: MAKE_WOOD_PICKAXE
1423         - Inventory requirements: 1 wood
1424         - Block requirements: 1 crafting table is in CLOSE_BLOCKS
1425     - craft a stone pickaxe:
1426         - Action to do: MAKE_STONE_PICKAXE
1427         - Inventory requirements: 1 stone, 1 wood
1428         - Block requirements: 1 crafting table is in CLOSE_BLOCKS
1429     - craft an iron pickaxe:
1430         - Action to do: MAKE_IRON_PICKAXE
1431         - Inventory requirements: 1 iron, 1 coal, 1 stone
1432         - Block requirements: 1 crafting table and 1 furnace are in CLOSE_BLOCKS
1433
1434     ### sword
1435     - craft a wood sword:
1436         - Action to do: MAKE_WOOD_SWORD
1437         - Inventory requirements: 1 wood
1438         - Block requirements: 1 crafting table is in CLOSE_BLOCKS
1439     - craft a stone sword:
1440         - Action to do: MAKE_STONE_SWORD
1441         - Inventory requirements: 1 stone, 1 wood

```

```

1404     - Block requirements: 1 crafting table is in CLOSE_BLOCKS
1405 - craft an iron sword:
1406     - Action to do: MAKE_IRON_SWORD
1407     - Inventory requirements: 1 iron, 1 coal, 1 stone
1408     - Block requirements: 1 crafting table and 1 furnace are in CLOSE_BLOCKS
1409 Rules to collect resources:
1410 - collect wood:
1411     - Action to do: DO
1412     - Block requirements: 1 tree is ahead of the player
1413     - Inventory requirements: None
1414 - collect stone:
1415     - Action to do: DO
1416     - Block requirements: 1 stone is ahead of the player
1417     - Inventory requirements: Have a wood pickaxe in the inventory
1418 - collect coal:
1419     - Action to do: DO
1420     - Block requirements: 1 coal is ahead of the player
1421     - Inventory requirements: Have a wood pickaxe in the inventory
1422 - collect iron:
1423     - Action to do: DO
1424     - Block requirements: 1 iron is ahead of the player
1425     - Inventory requirements: Have a stone pickaxe in the inventory
1426 - collect diamond:
1427     - Action to do: DO
1428     - Block requirements: 1 diamond is ahead of the player
1429     - Inventory requirements: Have an iron pickaxe in the inventory
1430 Rules to place things:
1431 - place a stone block:
1432     - Action to do: PLACE_STONE
1433     - Block requirements: no solid block is ahead of the player
1434     - Inventory requirements: 1 stone is in the inventory
1435 - place a crafting table:
1436     - Action to do: PLACE_TABLE
1437     - Block requirements: no solid block is ahead of the player
1438     - Inventory requirements: 1 wood is in the inventory
1439 - place a furnace:
1440     - Action to do: PLACE_FURNACE
1441     - Block requirements: no solid block is ahead of the player
1442     - Inventory requirements: 1 stone is in the inventory
1443 - place a plant:
1444     - Action to do: PLACE_PLANT
1445     - Block requirements: no solid block is ahead of the player
1446     - Inventory requirements: 1 sapling is in the inventory
1447     - Note: the plant will grow and become a plant block after a certain amount of time
1448
1449 ## 2.3 Useful Functions to Implement the Reward Functions
1450 Here are some useful functions that you can use to implement the reward functions:
1451 ```python
1452 def get_agent_view(state):
1453     """
1454     Get the agent's view of the map, centered around the agent's position.
1455
1456     Args:
1457         state (EnvState): The current state of the environment.
1458
1459     Returns:
1460         jnp.ndarray: The agent's view of the map with dimensions (7, 9). The agent's position is at
1461             map_view[3, 4].
1462     """
1463     # Convert the observation dimensions to a JAX array for easier manipulation
1464     obs_dim_array = jnp.array([OBS_DIM[0], OBS_DIM[1]], dtype=jnp.int32)
1465
1466     # Pad the map to ensure the view is within bounds, even if the agent is near the edge
1467     # We pad by MAX_OBS_DIM + 2 to ensure there's enough padding on all sides
1468     padded_grid = jnp.pad(
1469         state.map,
1470         (MAX_OBS_DIM + 2, MAX_OBS_DIM + 2),
1471         constant_values=BlockType.OUT_OF_BOUNDS.value,
1472     )
1473
1474     # Calculate the top-left corner of the view
1475     # We adjust the player's position by subtracting half the observation dimensions
1476     # and then adding the padding amount (MAX_OBS_DIM + 2) to account for the padding we added
1477     tl_corner = state.player_position - obs_dim_array // 2 + MAX_OBS_DIM + 2
1478
1479     # Extract the agent's view from the padded grid using dynamic slicing
1480     # This ensures that the view is always a 7x9 grid centered around the agent's position
1481     map_view = jax.lax.dynamic_slice(padded_grid, tl_corner, OBS_DIM)
1482     return map_view
1483
1484 # Example usage
1485 map_view = get_agent_view(state)
1486
1487 def is_mob_nearby(state, neighborhood_size=jnp.array([OBS_DIM[0], OBS_DIM[1]], dtype=jnp.int32)):
1488     """
1489     Return the number and type of mobs in the agent's neighborhood. By default, the neighborhood is the agent's
1490     view (7x9).

```

```

1458
1459
1460     Args:
1461         state (EnvState): The current state of the environment.
1462         neighborhood_size (jnp.array): The size of the neighborhood around the agent.
1463
1464     Returns:
1465         tuple: A tuple containing the count of skeletons, cows, and zombies in the neighborhood.
1466     """
1467
1468     # Boundary of the neighborhood
1469     tl_corner = state.player_position - neighborhood_size // 2
1470     br_corner = state.player_position + neighborhood_size // 2
1471
1472     # Helper function to count mobs within the neighborhood
1473     def count_mobs(mob_positions, mob_mask, tl_corner, br_corner):
1474         # Check if the mob is within the neighborhood bounds for each dimension
1475         in_neighborhood = jnp.logical_and(
1476             jnp.logical_and(mob_positions[:, 0] >= tl_corner[0], mob_positions[:, 0] <= br_corner[0]),
1477             jnp.logical_and(mob_positions[:, 1] >= tl_corner[1], mob_positions[:, 1] <= br_corner[1])
1478         )
1479
1480         # Combine the neighborhood check with the mask to count only active mobs
1481         active_in_neighborhood = jnp.logical_and(in_neighborhood, mob_mask)
1482
1483         # Count the number of active mobs in the neighborhood
1484         return jnp.sum(active_in_neighborhood)
1485
1486     # Count skeletons
1487     skeleton_count = count_mobs(state.skeletons.position, state.skeletons.mask, tl_corner, br_corner)
1488
1489     # Count cows
1490     cow_count = count_mobs(state.cows.position, state.cows.mask, tl_corner, br_corner)
1491
1492     # Count zombies
1493     zombie_count = count_mobs(state.zombies.position, state.zombies.mask, tl_corner, br_corner)
1494
1495     return skeleton_count, cow_count, zombie_count
1496
1497 # Example usage
1498 skeleton_count, cow_count, zombie_count = is_mob_nearby(env_state, neighborhood_size=jnp.array([17, 17], dtype
1499 =jnp.int32))
1500
1501 def block_pattern_matching(
1502     map_: jnp.ndarray,
1503     pattern: jnp.ndarray,
1504     block_type: int,
1505     player_position: jnp.ndarray
1506 ) -> bool:
1507     """
1508     Checks if 'pattern' matches 'block_type' around 'player_position' in 'map_'.
1509
1510     Where pattern == 1, map_ must be block_type.
1511     Where pattern == 0, we don't care.
1512
1513     Both 'map_' and 'pattern' must have statically known shapes if you
1514     want to JIT this function without errors.
1515     """
1516     # Shape info
1517     H, W = map_.shape
1518     P, Q = pattern.shape # must be fixed shape if you want to JIT
1519
1520     # Compute top-left corner in the map
1521     # so that pattern is "centered" on the player
1522     offset_row = player_position[0] - P // 2
1523     offset_col = player_position[1] - Q // 2
1524
1525     # Check bounds: if we can't slice a full P,Q subarray, we return False
1526     def in_bounds_fn(_):
1527         # Use lax.dynamic_slice to extract the sub-section of the map
1528         region = jax.lax.dynamic_slice(map_, (offset_row, offset_col), (P, Q))
1529         # region now has shape (P, Q)
1530
1531         # Where pattern == 1, we require region == block_type
1532         matches = jnp.where(pattern == 1, region == block_type, True)
1533         return jnp.all(matches)
1534
1535     # Return False if out-of-bounds
1536     in_bounds = (
1537         (offset_row >= 0) & (offset_col >= 0) &
1538         ((offset_row + P) <= H) & ((offset_col + Q) <= W)
1539     )
1540     return jax.lax.cond(in_bounds, in_bounds_fn, lambda _: False, operand=None)
1541
1542 # Example usage
1543 pattern = jnp.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]], dtype=jnp.int32) # block on the left, right, up and
1544 down around the player
1545 block_type = BlockType.STONE.value
1546 player_position = jnp.array([10, 10], dtype=jnp.int32)
1547 is_pattern_matched = block_pattern_matching(state.map, pattern, block_type, player_position)
1548 """

```



```

1512 # 3. Technical Implementation Guide
1513
1514 ## 3.1 JAX/JIT Compatibility Rules
1515 Follow these rules to ensure your code is compatible with JAX/JIT:
1516
1517 ```python
1518 # Use JAX conditionals
1519 has_resources = jnp.logical_and(state.inventory.stone >= 3, state.inventory.wood >= 2) # jnp.logical_and, jnp
1520 .logical_or takes only two arguments at a time. Alternatively, you can use & and | but add parentheses
1521 around the condition.
1522
1523 # No Python conditionals
1524 if state.inventory.stone >= 3 and state.inventory.wood >= 2: # Wrong
1525
1526 # No chained logical operations
1527 condition = a & b & c # Wrong
1528 condition = (a & b) & c # Correct
1529
1530 # Use JAX arrays
1531 position = jnp.array([0, 0], dtype=jnp.int32)
1532 # No Python lists
1533 position = [0, 0] # Wrong
1534
1535 # Use proper reward state structure
1536 @struct.dataclass
1537 class RewardState:
1538     counter: int = 0
1539     position: jnp.ndarray = field(default_factory=lambda: jnp.zeros(2))
1540
1541 # Don't call attributes that are not part of the state representation (see section 2)
1542 had_pickaxe_diamond = state.inventory.pickaxe_diamond # inventory does not have pickaxe_diamond attribute
1543 has_placed_furnace = state.inventory.furnace == 1 # inventory does not have furnace attribute
1544 ```
1545
1546 ## 3.2 Bad practices to use in crater
1547
1548 ```python
1549 # Don't use state.map to check if the agent has placed a certain number of blocks, it contains blocks that are
1550 not placed by the agent
1551 stone_blocks_placed = jnp.sum(state.map == BlockType.STONE.value) >= 5
1552 # To check the number of blocks placed by the agent, you can use the decrease of blocks in the inventory or
1553 contrast the map with a map of the previous state that is saved in the reward state.
1554 @struct.dataclass
1555 class RewardState_build_very_basic_bridge:
1556     counter_prompt: int = 0
1557     previous_map: jnp.ndarray = field(default_factory=lambda: jnp.zeros((MAP_SIZE[0], MAP_SIZE[1]), dtype=jnp.
1558 int32))
1559 # with the inventory
1560 stone_placed = jnp.logical_and(state.inventory.stone < reward_state.previous_inventory.stone)
1561 # with the map
1562 block_ahead_position = state.player_position + DIRECTIONS[state.player_direction]
1563 stone_placed = jnp.logical_and(state.map[block_ahead_position[0], block_ahead_position[1]] == BlockType.STONE.
1564 value, reward_state.previous_map[block_ahead_position[0], block_ahead_position[1]] != BlockType.STONE.
1565 value)
1566 ```
1567
1568 ## 3.2 Reward Function Implementation
1569
1570 To create a new task you should write a function with the same input and output as the previous examples (see
1571 section 4. Reference Examples). Also, add a reward class like the one in the reference examples. The
1572 reward class is a way to store information across different step updates and generate rewards based on
1573 information contained in a sequence of states of the world. Also, add a list of brief descriptions of
1574 the sequential subtasks with less than ten subtasks to guide the AI agent in the learning process.
1575
1576 # 4. Reference Examples of tasks
1577 Here are code the code examples to help you design new tasks.
1578 Task 1: kill_two_cows
1579 Category: Combat
1580 **Goal(s) decomposition**:
1581 1. main goal 1: Kill two cows
1582 - subgoal 1: Kill a cow
1583 - completion condition: the agent has killed a cow
1584 - subgoal 2: Kill another cow
1585 - completion condition: the agent has killed a second cow
1586
1587 ```python
1588 prompt_sequence_kill_two_cows = ['kill two cows', 'kill cow', 'kill cow']
1589
1590 def kill_two_cows(state, action, rng, reward_state):
1591     """
1592     Check whether the agent kills at least 2 cows.
1593     """
1594     reward_state.current_task = reward_state.kill_two_cows
1595
1596     # Check if a cow is killed
1597     cow_count = state.cows.just_died.sum()
1598     cows_killed = reward_state.current_task.cows_killed + cow_count
1599
1600     reward_state.current_task = reward_state.current_task.replace(

```

```

        cows_killed=cows_killed,
        counter_prompt=cows_killed
    )

    task_achieved = cows_killed >= 2
    reward_state = reward_state.replace(kill_two_cows=reward_state_current_task)

    return task_achieved, reward_state, reward_state_current_task.counter_prompt

@struct.dataclass
class RewardState_kill_two_cows:
    counter_prompt: int = 0
    cows_killed: int = 0
...

# 5. Templates for generating the code of a new task
To create a new task you should write a function with the same input and output as the previous examples (see
section 3. Reference Examples). Also, add a reward class like the one in the reference examples. The
reward class is a way to store information across different step updates and generate rewards based on
information contained in a sequence of states of the world. Also, add a list of brief descriptions of
the sequential subtasks with less than ten subtasks to guide the AI agent in the learning process. These
descriptions are used to guide the agent in the learning process and they correspond to the goals and
subgoals of the task defined in the reward function.

The task submission format is as follows. DO NOT IMPORT MODULES, you can only use jax and jax.numpy as jnp and
the Useful Functions (from section 2.3):

My proposed code for the reward function and its reward state and prompt sequence is:
'''python
<goal_description = [<description of the goals together>, <description of subgoal 1>, <description of subgoal
2>, ...]> # less than 10 subgoals

<reward function> # Don't re-use the same exact names of functions as the ones in reference examples

<reward state>
'''

--- end of the document ---

Now that you've taken into account the previous document create the code of the following task:
Name: kill_skeleton_with_wood_sword
**Goal(s) decomposition**:

1. main goal 1: Kill a skeleton
    - subgoal 1: Craft a wood sword.
      - completion condition: the agent has a wood sword in the inventory.
    - subgoal 2: Find a skeleton.
      - completion condition: a skeleton is within the agent's observable area (e.g., 17x17 area centered on
the agent), can be implemented by testing the output of is_mob_nearby.
    - subgoal 3: Kill the skeleton.
      - completion condition: the agent has killed a skeleton (skeleton.mask becomes false and skeleton.
just_died become true in the 'state.skeletons' ).

.
My proposed task is:

```