**Media Engineering and Technology Faculty**
**German University in Cairo**

# Learning of Symbolic Representations for Rescue Scenarios in Disaster Zones

**Bachelor Thesis**

Author: Ahmed Khaled Ahmed Gaber Diab

Supervisors: Dr. Nourhan Ehab Azab

Submission Date: 01 June, 2023

This is to certify that:

(i) The thesis comprises only my original work toward the Bachelor Degree.

(ii) Due acknowledgement has been made in the text to all other material used.

Ahmed Khaled Ahmed Gaber Diab
01 June, 2023

# Acknowledgments

First and foremost, I would like to thank my mentors, Dr. Nourhan Ehab and Eng. Reem Adel. They've been there for me through this entire thesis journey, always ready to guide and advise.

I also want to thank my parents for always encouraging me to achieve my dreams and being the greatest role models one could ask for. I must mention my younger sisters as well, who have kept my spirits high during the length of this project.

I'm grateful for my friends and colleagues. They've stood by me, providing support and motivation when I needed it the most. They've helped keep me grounded, and I'll always appreciate that.

Lastly, I want to thank the online community of researchers and coders. They've made it so much easier to share and find knowledge, making my research less of a burden. They've done an amazing job and it's truly inspiring.

# Abstract

The development of intelligent search and rescue systems is of paramount importance in disaster response scenarios, as rapid and effective assistance can save lives and mitigate damage. In this thesis, I present a novel approach that combines reinforcement learning with symbolic knowledge extraction to create an autonomous agent capable of navigating complex environments and making informed decisions in a disaster zone, specifically in my custom-designed DisasterZone environment. By leveraging the strengths of Q-learning and Ron Sun's [11] approach to extracting IF-THEN rules from the learned policy, I aim to develop a more interpretable and explainable model for disaster response tasks, enabling better collaboration between humans and artificial intelligence systems. My findings reveal that the training time of the agent is proportional to the complexity of the environment, showcasing the efficiency of my approach. However, limitations and challenges arise from the lack of computational power, as training is conducted using a CPU-based approach rather than a GPU one.

# Contents

# Chapter 1

# Introduction

Disaster response and search and rescue operations are among the most critical and time-sensitive tasks faced by emergency management teams. These operations often involve navigating treacherous environments with limited visibility, obstacles, and potential hazards, which can challenge even the most experienced responders. Disaster zones pose significant challenges to relief organizations due to the complex and unpredictable nature of disasters. The need for timely and efficient response is crucial in these situations, which can be difficult to achieve with human resources alone. Therefore, creating AI agents to assist in disaster zones has become increasingly important. These agents are designed to learn from their environment and make decisions based on their observations, allowing them to adapt to changing conditions in real-time. With their ability to process large amounts of data and make informed decisions, agents can significantly enhance the capabilities of relief organizations in disaster zones. The importance of these agents in disaster response cannot be overstated, as they have the potential to save countless lives and minimize the devastating effects of disasters that are present nearly everywhere around the globe as shown in the figure below.

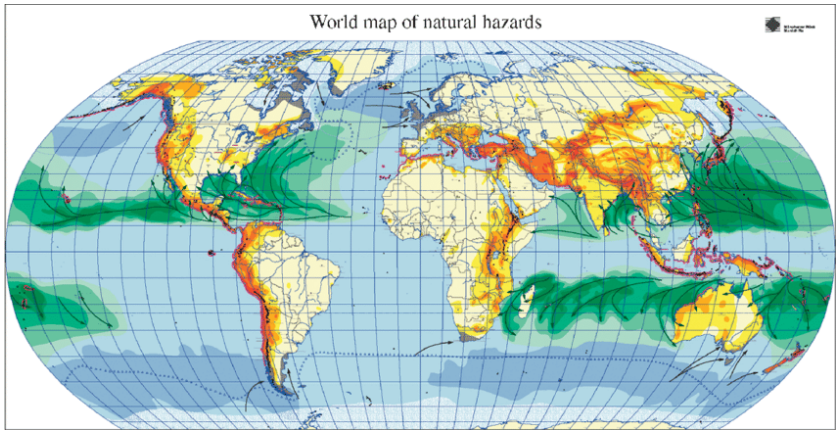


Figure 1.1: World Map of Natural Hazards [10]

## 1.1 Motivation

In recent years, the application of artificial intelligence and robotics in disaster response has shown promising results, offering the potential to augment human capabilities and save lives. The recent Turkish earthquake disaster, which caused significant loss of life and property, further highlights the urgency of enhancing disaster response capabilities. Reinforcement learning, a branch of machine learning, provides an opportunity to develop intelligent agents capable of learning optimal behaviors through trial and error. However, traditional reinforcement learning methods often produce opaque policies that lack interpretability, which can hinder collaboration between humans and artificial intelligence systems. Consequently, there is a growing need to develop approaches that combine the strengths of reinforcement learning with the interpretability of symbolic knowledge extraction.

## 1.2 Problem Statement

The primary challenge in disaster response and search and rescue operations is to develop a neuro-symbolic agent capable of learning optimal behaviors to navigate complex environments while maintaining interpretability to facilitate human understanding and collaboration. Traditional reinforcement learning algorithms, such as Q-learning, can enable agents to learn optimal policies, but the resulting Q-tables often lack transparency and are difficult for humans to interpret. To bridge this gap, the thesis aims to leverage symbolic knowledge extraction techniques that transform the learned policy into a set of human-readable IF-THEN rules. These rules can provide insights into the agent's decision-making process and enhance human-AI collaboration in disaster response scenarios.

## 1.3 Scope and Objectives

The scope of this thesis is primarily focused on the development and evaluation of a Q-learning agent in a grid-based disaster zone environment to extract symbolic knowledge from its learned policy. The main objectives of this thesis are twofold:

1. I aim to create a reinforcement learning agent for navigating complex disaster zones, like earthquake-affected areas. To this end, I will design a grid-based environment called the *DisasterZone* to simulate search and rescue missions, including obstacles, survivors, and limited battery life. Using Ron Sun's approach [11], we will extract human-readable symbolic knowledge from the agent's learned policy. This knowledge extraction will involve generating IF-THEN rules that represent the agent's optimal policy, making its decision-making process more transparent and interpretable.

2. I intend to evaluate the quality of the extracted symbolic knowledge, assess its implications for real-world applications, and identify areas for future improvement and research.

## 1.4 Outline

This thesis is organized as follows:

### Chapter 2

*Chapter 2* offers a comprehensive background on reinforcement learning, symbolic knowledge extraction, and related work in disaster response and search and rescue.

### Chapter 3

Next, *Chapter 3* describes the *DisasterZone* custom environment, its design, and implementation details.

### Chapter 4

*Chapter 4* delves into the Q-learning agent, detailing the algorithm, and its implementation.

### Chapter 5

In *Chapter 5*, we discuss the symbolic knowledge extraction process, its algorithm, and implementation details.

### Chapter 6

*Chapter 6* presents the results and discussion, encompassing training performance, quality of symbolic knowledge, implications, and insights.

### Chapter 7

Finally, *Chapter 7* concludes the thesis, summarizing the contributions, talking about the limitations faced, the space for future work, and providing final remarks.

# Chapter 2

# Background

This chapter provides an overview of the fundamental concepts and techniques used in this thesis. I begin by discussing reinforcement learning and its different approaches, with a focus on Q-learning. Next, I explore symbolic knowledge extraction and its importance in understanding and generalizing RL policies, concentrating on Ron Sun's [11] approach. Finally, I review related work in disaster response and search and rescue, highlighting the importance of these applications in the context of the thesis' study.

## 2.1 Reinforcement Learning

Reinforcement Learning (RL) is a sub-field of machine learning that focuses on training agents to make decisions by interacting with their environment [12]. The agent learns to perform actions that maximize its cumulative reward over time. The RL framework involves an agent, an environment, states, actions, and rewards. In each time step, the agent observes the current state of the environment, takes an action, receives a reward, and transitions to the next state. RL has been widely applied to a variety of problems, ranging from robotics and control systems to recommendation systems and game playing.

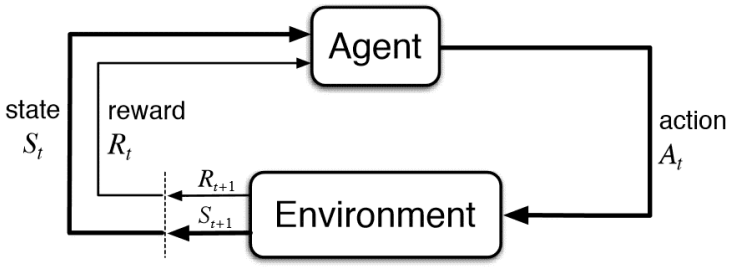


Figure 2.1: Reinforcement Learning [12]

### Markov Decision Processes

Reinforcement learning problems are often modeled as Markov Decision Processes (MDPs), which can be represented by the tuple $(S, A, P, R, \gamma)$, where:

- $S$ is the set of states,
- $A$ is the set of actions,
- $P$ is the state transition probability function: $P(s'|s, a)$,
- $R$ is the reward function: $R(s, a)$,
- $\gamma$ is the discount factor.

The objective of reinforcement learning is to find an optimal policy $\pi^*$ that maps states to actions, maximizing the expected cumulative reward.

## 2.1.1 RL Approaches

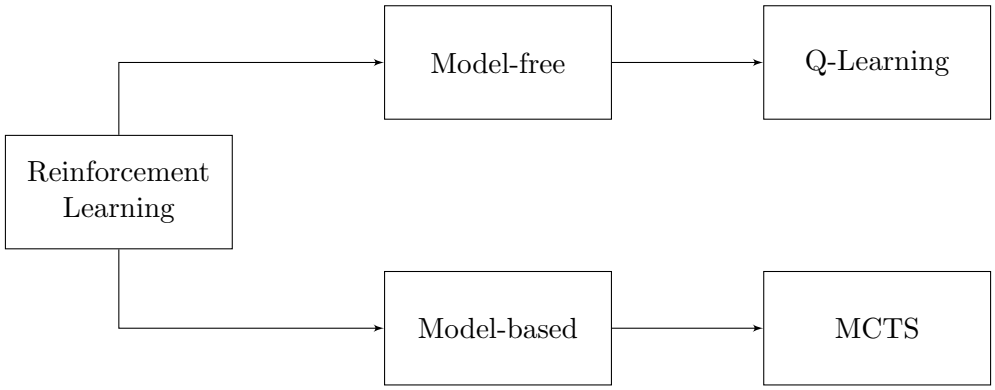


Figure 2.2: RL Approaches [12]

There are two main approaches in RL: (i) model-based and (ii) model-free. Model-based methods learn an explicit model of the environment, including transition probabilities and reward functions. The agent uses this model to plan and make decisions.

On the other hand, model-free methods do not require an explicit model of the environment. Instead, they directly learn the optimal policy or value function from the agent's interactions with the environment. Model-free approaches, such as Q-learning, have proven to be particularly effective in various complex environments and tasks.

### 2.1.2 Q-Learning

Q-learning is a popular model-free, off-policy reinforcement learning algorithm that estimates the action-value function, also known as the Q-function $Q(s, a)$, which represents the expected cumulative reward an agent can obtain by taking a specific action $a$ in a given state $s$ and then following a certain policy $\pi$ thereafter [13].

Q-learning uses a Q-table to store the Q-values for each state-action pair $(s, a)$. The agent iteratively updates its Q-table based on the rewards $R$ it receives from the environment, eventually converging to the optimal Q-function $Q^*(s, a)$. The agent can then use the learned Q-values to make optimal decisions in each state.

The Q-learning algorithm updates the Q-values according to the following equation:

$$\underbrace{Q(s,a)}_{\substack{\text{New}\\\text{Q-Value}}} = \underbrace{Q(s,a)}_{\substack{\text{Current}\\\text{Q-Value}}} + \underbrace{\alpha}_{\substack{\text{Learning}\\\text{Rate}}} \Big[ \underbrace{R(s,a)}_{\text{Reward}} + \underbrace{\gamma}_{\substack{\text{Discount}\\\text{Rate}}} \overbrace{\max_a Q'(s',a')}^{\substack{\text{Maximum predicted reward, given}\\\text{new state and all possible actions}}} - Q(s,a) \Big] \tag{2.1}$$

which is used and utilised in the following learning algorithm:

**Algorithm 1:** Q-Learning (off-policy TD control) for estimating $\pi \approx \pi^*$ [13]

**Data:** $\alpha \in (0, 1]$, $\gamma \in [0, 1]$, *epsilon* $> 0$
**Result:** A Q-table containing $Q(S, A)$ pairs defining estimated optimal policy
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except $Q(\text{terminal}, \cdot)$;
$Q(\text{terminal}, \cdot) \leftarrow 0$;
**foreach** *episode* **do**
    Initialize state S;
    **foreach** *step in episode* **do**
        **do**
            Choose action $A$ from $S$ using ε-greedy policy derived from $Q$;
            Take action $A$, then observe reward $R$ and next state $S'$;
            $Q(S, A) \leftarrow Q(S, A) + \alpha[R(S, A) + \gamma \max_a Q'(s', a') - Q(S, A)]$;
            $S \leftarrow S'$;
        **while** *$S$ is not terminal*;
    **end foreach**
**end foreach**

## 2.2 Symbolic Knowledge Extraction

Symbolic knowledge extraction refers to the process of extracting human-understandable rules and representations from trained machine learning models, such as reinforcement learning agents. The extracted rules can help researchers and practitioners understand the decision-making process of the agent and provide insights into its learned policy. There are several approaches to symbolic knowledge extraction, one of which is Ron Sun's [11] approach, which I will shed some light on in the following section.

### 2.2.1 Ron Sun's Approach

Ron Sun's [11] approach involves converting the learned Q-values from a Q-learning RL agent into symbolic rules in the form of IF-THEN statements. The process consists of three main steps:

1. Normalize the Q-values in the Q-table.
2. Perform clustering on the normalized Q-values to group similar states.
3. For each cluster, generate an IF-THEN rule that represents the best action for the states in the cluster.

This approach allows for the extraction of human-understandable rules that represent the agent's learned policy, which can be helpful for analysis and interpretation. However, the quality of the extracted rules depends on the chosen clustering algorithm, the number of clusters, and how well the agent learned the optimal policy.

### 2.2.2 Other Approaches

There are various other approaches to symbolic knowledge extraction, which have been extensively researched in recent years, such as decision tree induction, rule induction, and neuro-symbolic methods. Some of the relevant works in this area include:

#### Learning Symbolic Representations from Unlabeled Data

Researchers have proposed alternative methods for learning symbolic representations from data. For example, Zhang et al. combined unsupervised feature learning with symbolic rule mining, enabling the extraction of meaningful symbolic representations from unannotated data [14]. This approach overcomes the limitations of traditional rule-based and logic-based methods in handling complex and unstructured data.

### Agent-Based Learning for Heuristic Discovery

Apeldoorn and Kern-Isberner proposed an agent-based learning approach for finding and exploiting heuristics in unknown environments [1]. By training agents to explore and learn from unknown environments, this approach overcomes the limitations of traditional rule-based and logic-based methods in handling complex and unstructured data. This approach can be applied to rescue scenarios in disaster zones, where heuristics can aid in decision-making and planning for efficient rescue operations.

### Cognitive Architectures for Neuro-Symbolic AI

Kramer proposed a cognitive architecture for Neuro-Symbolic AI, called the Subsumption Architecture [7]. The Subsumption Architecture combines symbolic and sub-symbolic representations in a layered architecture, allowing for the integration of symbolic and sub-symbolic processing in a flexible and scalable way. This architecture enables the development of interpretable and adaptable AI systems that can reason about both low-level sensory data and high-level abstract concepts.

## 2.3 Disaster Response and Search and Rescue

Disaster response and search and rescue (SAR) operations are critical areas of research and practice, as they can save lives and minimize damage in emergency situations [9]. Reinforcement learning and other machine learning techniques have been applied to various aspects of disaster response, such as path planning, coordination among multiple agents, and resource allocation.

Previous studies have used reinforcement learning for search and rescue tasks in simulated environments, focusing on different aspects like multi-agent collaboration, efficient exploration, and real-time decision-making [4]. These studies have demonstrated the potential of reinforcement learning in addressing complex and dynamic problems in the context of disaster response.

In this thesis, I build on this existing research by developing a reinforcement learning agent for disaster response in a grid-based environment and extracting symbolic knowledge from the learned policy to facilitate better understanding and analysis of the agent's decision-making process.

# Chapter 3

# Environment

The environment is designed to simulate a search and rescue scenario in the aftermath of a disaster. The agent's objective is to navigate a grid-like environment, rescuing survivors while managing its energy reserves and avoiding obstacles. This chapter provides an overview of the environment design, implementation details, and evaluation criteria.

## 3.1 Environment Design

In this work, I introduce the *DisasterZone* environment, a grid-based simulation platform for disaster response and search and rescue scenarios. The *DisasterZone* environment is designed to be a challenging, complex, and dynamic domain that requires agents to navigate through a grid filled with obstacles, locate survivors, and bring them back to a safe zone while managing their energy levels.

### 3.1.1 Grid Representation

The *DisasterZone* environment is represented as a discrete grid, with each cell representing a location that can be traversed by a rescue agent $A$. The grid is defined by its dimensions, width $W$ and height $H$, and contains a mix of obstacles $X$, survivors $S$, and charging stations $C$, which will be further described in the following sections.

| | | | |
|---|---|---|---|
| A | | C | S |
| X | X | | X |
| | S | C | X |
| X | | | S |

$$\text{DisasterZone} = (W \times H),\ X,\ S,\ C \tag{3.1}$$

The agent's task is to navigate the environment, locate survivors, and provide assistance while avoiding obstacles and maintaining its energy reserves.

### 3.1.2 Elements

The *DisasterZone* environment contains three types of elements:

#### Obstacles

They represent impassable areas such as debris, collapsed buildings, or other hazards that the agent must avoid during navigation.

#### Survivors

They represent individuals in need of assistance and are the primary objective for the agent.

#### Charging Stations

They represent locations where the agent can recharge its energy reserves.

Obstacles, survivors, and charging stations are randomly placed within the environment based on predefined probabilities. The probabilities can be adjusted to create different scenarios with varying levels of difficulty, allowing for the evaluation of the agent's performance under diverse conditions.

### 3.1.3 Agent Dynamics

The agent's dynamics within the *DisasterZone* environment are governed by its actions, energy reserves, and interactions with survivors and obstacles. The agent can perform four possible actions at each time step: move up $\uparrow$, move down $\downarrow$, move left $\leftarrow$, or move right $\rightarrow$. When the agent moves, its energy reserves are depleted by a predefined amount. The agent must manage its energy reserves carefully, as running out of energy will result in mission failure.

The agent's interactions with the environment's elements are defined as follows:

- If the agent encounters a survivor, the survivor is considered to be rescued, and the agent receives a positive reward based on the distance to the survivor, the smaller the distance, the higher the reward.
- If the agent encounters an obstacle, the agent is penalized with a predefined negative reward value and is unable to move to the obstacle's location.
- If the agent reaches a charging station, its energy reserves are replenished to their maximum capacity.
- If the agent encounters an empty cell, the agent receives a negative reward.

The agent's goal is to maximize the total reward accumulated during a mission by rescuing as many survivors as possible while avoiding obstacles and managing its energy reserves.

## 3.2 Environment Implementation Details

The *DisasterZone* environment is implemented in Python using the OpenAI Gym library [3], which provides a standardized interface for reinforcement learning environments. The grid is represented with NumPy arrays [5] to enable efficient manipulation and computation of the agent's dynamics and interactions within the environment.

The agent's actions, energy reserves, and interactions with survivors and obstacles are managed within the custom Gym environment. The agent's state is represented as a tuple consisting of its position, energy level, and the locations of survivors, obstacles, and charging stations. The agent's reward function is designed to encourage the rescue of survivors, avoidance of obstacles, and efficient energy management.

A significant part of the implementation is the `step` function, which defines how the environment responds to the agent's actions. This function takes an action as input, updates the state of the environment, and returns the new state, the reward, and a boolean indicating whether the episode has ended. The Python code [6] for the `step` function is presented below:

```python
import numpy as np

def step(self, action):
    # Copy of the current agent position
    old_pos = self.agent_pos

    # Calculate the new position
    new_pos = tuple(map(add, self.agent_pos, self.actions_dict[action]))

    # Ensure the agent does not move outside the grid
    new_pos = (max(min(new_pos[0], self.grid_size - 1), 0), max(min(new_pos[1], self.grid_size - 1), 0))

    # Default battery consumption for moving to a new position
    battery_consumption = 1

    # Reward for moving into an empty cell
    reward = -2
    done = False

    hit_obstacle = False  # Flag to track if the agent attempted to move into an obstacle

    if self.grid[new_pos] == 1:  # Obstacle
        reward = -3
        hit_obstacle = True  # The agent attempted to move into an obstacle
    elif self.grid[new_pos] == 2:  # Survivor
        euclidean_distance = np.linalg.norm(
            np.array(new_pos) - np.array(old_pos))
        reward = max(5 - int(euclidean_distance), 1)
        self.grid[new_pos] = 0
        if new_pos in self.survivors:
```

```
            del self.survivors[new_pos]
        done = len(self.survivors) == 0
    elif self.grid[new_pos] == 3:  # Charging Station
        # Charging the battery instead of consuming it
        battery_consumption = -3

    # Update agent's battery life
    self.battery_life -= battery_consumption

    # Check if the battery life has run out
    if self.battery_life <= 0:
        done = True

    # Calculate reward penalty for low battery life
    reward -= self.max_battery_life - self.battery_life

    # Update agent's position
    self.agent_pos = new_pos

    return self.grid, reward, done, {"hit_obstacle": hit_obstacle}
```

Listing 3.1: Environment Step Function

The `step` function begins by checking the action taken by the agent. Depending on the action (move up, down, left, or right), the agent's position is updated within the boundaries of the grid. If the agent attempts to move outside the grid, the action is ignored and the agent remains in the same position. If the agent moves onto a cell with a survivor, the survivor is rescued and removed from the grid.

The energy level of the agent is then updated. Each movement action consumes one unit of energy, and moving into a charging station recharges the agent's energy to its maximum capacity.

The reward is computed based on the action taken. The agent receives a ***positive*** reward for rescuing a survivor, a higher reward for moving towards the closest survivor, and a ***negative*** reward for moving into an obstacle or an empty cell, attempting to move to the same position, or when running on very low battery levels.

Finally, the function checks if the episode has ended. This occurs if the agent has run out of energy, or if all survivors have been rescued. The function then returns the new state, the reward, and the termination flag.

By implementing the *DisasterZone* environment as a custom Gym environment, it allows for seamless integration with various reinforcement learning algorithms and libraries, which can facilitate the development and testing of the rescue agent.

# Chapter 4

# Reinforcement Learning Agent

In the vast and dynamic landscape of artificial intelligence, the reinforcement learning agent stands as a beacon of autonomous decision-making. Drawing inspiration from the principles of behavioral psychology, RL agents learn by interacting with their environment, seeking to optimize a reward signal through their actions. Within the context of our study, I employ a Q-learning based RL agent, a variant of the value-iteration based methods. Q-learning, known for its off-policy characteristic, holds the promise of learning an optimal policy regardless of the agent's current exploratory strategies [12]. This approach is particularly fitting for our disaster response and search and rescue scenario, where the stakes are high, and the environment is fraught with uncertainty. The agent's task is formidable: to navigate the perilous terrain, discover survivors, and ensure their safe extraction, all while constantly learning and adjusting its strategy.

## 4.1 Q-Learning Algorithm

Reinforcement Learning (RL) algorithms provide a powerful framework for training intelligent agents to make optimal decisions in dynamic environments. Among the RL algorithms, Q-Learning stands out as a versatile and effective method for learning an optimal policy in unknown environments [13]. This characteristic makes Q-Learning particularly suitable for our application in the *DisasterZone*, where the agent needs to navigate an unpredictable environment. In this section, I present a detailed overview of the Q-Learning algorithm, its key components, and its application within our RL agent.

### Model-Free

Q-Learning is a model-free RL algorithm that learns from direct interaction with the environment without requiring a model of the environment dynamics. It falls under the category of Temporal Difference (TD) learning methods which combines ideas from Monte Carlo methods and Dynamic Programming [2]. The algorithm estimates the value of state-action pairs, known as Q-values, which represent the expected cumulative reward an agent can obtain by taking a specific action in a given state and following a certain policy thereafter, allowing the agent to learn from incomplete episodes, making it quite powerful for our needs [13].

### Off-Policy

The Q-Learning algorithm is an off-policy method, meaning that it can learn an optimal policy while following a different exploration policy. This characteristic allows the agent to balance exploration and exploitation effectively. By exploring different actions and learning from the observed rewards, the agent gradually improves its Q-values and converges to the optimal Q-values that correspond to the optimal policy.

### Algorithm

To illustrate the Q-Learning algorithm, I provide a flowchart in Figure 4.1, which outlines the main steps involved in the learning process. Starting with the initialization of the Q-values, the agent interacts with the environment, selects actions based on an exploration strategy (e.g., $\epsilon$-greedy), receives rewards, and updates the Q-values accordingly. The process continues until the agent converges to an optimal policy.

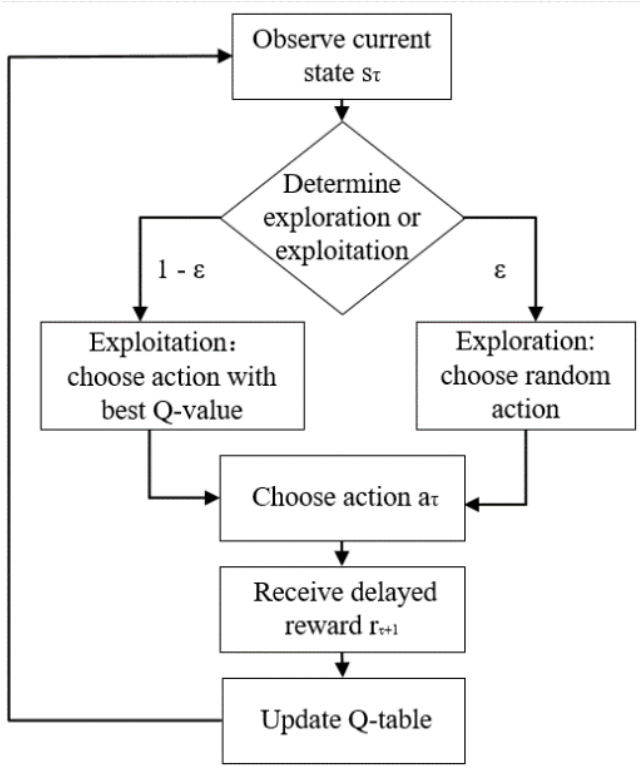


Figure 4.1: Q-Learning Algorithm Flowchart [13]

As this algorithm is fundamental to our agent, I encourage readers to refer to Section 2.1.2 for a comprehensive understanding of its operation, including the iterative update of Q-values and Algorithm 1.

### 4.1.1 Learning Rate, Discount Factor, and Exploration Rate

The learning rate ($\alpha$), discount factor ($\gamma$), and exploration rate ($\epsilon$) are key parameters in the Q-Learning algorithm, and the choice of their values is instrumental in shaping the agent's learning behavior.

- The learning rate $\alpha \in (0, 1]$ determines how much of the newly acquired information will override the old information. A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the most recent information.
- The discount factor $\gamma \in [0, 1]$ determines the importance of future rewards. A factor of 0 will make the agent short-sighted by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward.
- The exploration rate $\epsilon > 0$ is used in the $\epsilon$-greedy policy where the agent will exploit the best known action with a probability of $1 - \epsilon$ or explore a random action with a probability of $\epsilon$ controlling the exploration-exploitation trade-off.

The optimal values for these parameters depend on the specific problem and may require tuning through experimentation. In our implementation, as provided in Section 4.2 I set the learning rate to $\alpha = 0.1$, the discount factor to $\gamma = 0.99$, and the exploration rate to $\epsilon = 0.1$. These values were selected to ensure an optimal balance between rapid learning, long-term reward prioritization, and the exploration-exploitation trade-off [13] in our *DisasterZone* environment.

### 4.1.2 Q-Table and State-Action Pairs

The Q-table is a simple lookup table where I store the Q-values for every state-action pair. This table guides the agent to the best action at each state. As previously elaborated in Section 2.1.2, the Q-table forms the crux of the Q-learning algorithm.

| **State/Action** | **Action 1** | **Action 2** | ... | **Action $N$** |
|---|---|---|---|---|
| State 1 | $Q(s_1, a_1)$ | $Q(s_1, a_2)$ | ... | $Q(s_1, a_N)$ |
| State 2 | $Q(s_2, a_1)$ | $Q(s_2, a_2)$ | ... | $Q(s_2, a_N)$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| State $M$ | $Q(s_M, a_1)$ | $Q(s_M, a_2)$ | ... | $Q(s_M, a_N)$ |

Table 4.1: Q-Table [13]

Table 4.1 illustrates how the Q-values are organized in the Q-table for $M$ states and $N$ actions. Each cell's entry, $Q(s, a)$, in the table represents the expected future reward for taking a particular action $a$ in a specific state $s$, reflecting the agent's current understanding of the environment. The Q-values are updated, using equation 4.1, during the learning process as the agent explores the environment and receives rewards, gradually converging to the optimal values that reflect the optimal policy $\pi^*$.

The Q-function satisfies the famous Bellman equation [2]:

$$Q^{new}(s_t, a_t) = \mathbb{E} \cdot \left[ r_t + \gamma \cdot \max_{a'} Q(s', a') \mid s_t, a_t \right] \tag{4.1}$$

$$\mathbb{E} = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \tag{4.2}$$

where $\mathbb{E}$ denotes the expected value, $r$ is the immediate reward, $s'$ is the new state after taking action $a$, and $\gamma$ is the discount factor.

For our *DisasterZone* environment, the agent can choose from four actions at each state: go up, go left, go right, or go down. Let's consider a subset of the environment consisting of $M$ states. An example of how the Q-table might look like is as follows:

| **State/Action** | **Up** | **Left** | **Right** | **Down** |
|---|---|---|---|---|
| State 1 | $Q(s_1, \text{Up})$ | $Q(s_1, \text{Left})$ | $Q(s_1, \text{Right})$ | $Q(s_1, \text{Down})$ |
| State 2 | $Q(s_2, \text{Up})$ | $Q(s_2, \text{Left})$ | $Q(s_2, \text{Right})$ | $Q(s_2, \text{Down})$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| State $M$ | $Q(s_M^*, \text{Up})$ | $Q(s_M^*, \text{Left})$ | $Q(s_M^*, \text{Right})$ | $Q(s_M^*, \text{Down})$ |

Table 4.2: Sample *DisasterZone* Q-Table [6]

*It is important to note that the state $s_M$ belongs to a set of matrices with different combinations of elements. In our *DisasterZone* environment, the grid-like environment can be represented as a set of matrices, denoted as $\mathcal{S} = \{\text{Matrix}_1, \text{Matrix}_2, \ldots, \text{Matrix}_M\}$, where each matrix corresponds to a specific configuration of the environment.

Each matrix in the set $\mathcal{S}$ represents a different arrangement of elements within the environment, including agents, obstacles, survivors, and charging stations.

The environment state $s$ corresponds to one of the matrices in the set $\mathcal{S}$. By considering $s \in \mathcal{S}$, the agent can perceive the current configuration of the environment and make informed decisions based on the Q-values in the Q-table.

To illustrate this, let's consider a set of matrices $\mathcal{S}$ containing different configurations of the environment:

$$s \in \mathcal{S} = \left[ \overbrace{\begin{array}{|c|c|c|c|}\hline A & & C & S \\ \hline X & X & & X \\ \hline & S & C & X \\ \hline X & & & S \\ \hline \end{array}}^{s_1}, \overbrace{\begin{array}{|c|c|c|c|}\hline S & & C & X \\ \hline X & A & S & X \\ \hline & X & C & X \\ \hline S & & & S \\ \hline \end{array}}^{s_2}, \ldots, \overbrace{\begin{array}{|c|c|c|c|}\hline C & & S & X \\ \hline S & X & & X \\ \hline & S & C & X \\ \hline X & & & A \\ \hline \end{array}}^{s_M} \right] \tag{4.3}$$

Each matrix in the set $\mathcal{S}$ represents a specific configuration of the *DisasterZone* environment, where the agent (represented by A) is located in a particular position, obstacles (represented by X) are present, survivors (represented by S) are scattered, and charging stations (represented by C) are available at specific locations. The agent's decisions and actions will be based on these matrices in the set $\mathcal{S}$.

## 4.2 Agent Implementation Details

Our RL agent, implemented in the Python class `QLearningAgent`, includes several key components that come together to create a learning system which is both powerful and adaptable. Our approach [6] was based on the mathematical and scientific foundations proposed and discussed in Sections 2.1.2, 4.1.

### Initialization

The constructor method `__init__` initializes the agent with its associated environment, the learning rate $\alpha$, the discount factor $\gamma$, and the exploration rate $\epsilon$. It also prepares an empty dictionary to serve as the Q-table.

```
def __init__(self, env, alpha=0.1, gamma=0.99, epsilon=0.1):
    self.env = env
    self.alpha = alpha
    self.gamma = gamma
    self.epsilon = epsilon
    self.q_table = {}
```

Listing 4.1: Initialization Function

### State Key Creation

A crucial aspect of our implementation is the creation of a unique key for each state-action pair for storing in the Q-table. This is handled by the `_state_key` method, which flattens the state and combines it with the agent's position to form a unique key. See Equation 4.3

```
def _state_key(self, state, agent_pos):
    return (tuple(state.flatten()), agent_pos)
```

Listing 4.2: State-Key Function

### Q-table Management

The agent's Q-table is managed using the `_get_q` method. If a state-action key does not exist in the Q-table, this method initializes it with an array of zeros corresponding to the size of the action space.

```
def _get_q(self, state, agent_pos, action):
    state_key = self._state_key(state, agent_pos)
    if state_key not in self.q_table:
        self.q_table[state_key] = np.zeros(self.env.action_space.n)
    return self.q_table[state_key]
```

Listing 4.3: Get-Q Function

### Action Selection

Our agent selects its actions using the `_choose_action` method. It applies an $\epsilon$-greedy policy for balancing exploration and exploitation, using the `np.random.rand()` function to decide whether to explore or exploit based on the epsilon value.

```
def _choose_action(self, state, agent_pos):
    if np.random.rand() < self.epsilon:
        return self.env.action_space.sample()
    else:
        q_values = self._get_q(state, agent_pos, None)
        return np.argmax(q_values)
```

Listing 4.4: Choose Action Function

### Q-table Update

The core Q-learning algorithm is implemented in the `_update_q` method. Here, the agent updates the Q-value of the current state-action pair based on the reward received and the maximum Q-value of the next state, scaled by the learning rate. The discount factor is also applied to the future reward. See Equations 2.1 and 4.1.

```
def _update_q(self, state, agent_pos, action, next_state,
    next_agent_pos, reward, done):
    q_values = self._get_q(state, agent_pos, action)
    next_q_values = self._get_q(next_state, next_agent_pos, None)
    next_action = self._choose_action(next_state, next_agent_pos)

    if done:
        q_values[action] += self.alpha * (reward - q_values[action])
    else:
        q_values[action] += self.alpha * \
            (reward + self.gamma *
             next_q_values[next_action] - q_values[action])
```

Listing 4.5: Update-Q Function

### Learning Process

The main learning loop of the agent is contained in the `learn` method. This method controls the agent's interaction with the environment over a given number of episodes. In each episode, the agent repeatedly selects actions, updates the Q-table, and moves to the next state until the episode ends. The cumulative reward of each episode is tracked for evaluation purposes.

```
def learn(self, num_episodes):
    rewards = []
    for episode in range(num_episodes):
        state = self.env.reset()
        done = False
        episode_reward = 0

        while not done:
            action = self._choose_action(state, self.env.agent_pos)
            next_state, reward, done, _ = self.env.step(action)
            self._update_q(state, self.env.agent_pos, action,
                           next_state, self.env.agent_pos, reward, done
    )
            state = next_state

            episode_reward += reward

            self.visualize_training(reward, episode_reward, done)

        rewards.append(episode_reward)

        print_mode = 1

        if print_mode == 0: # Print all episodes stats
            print(
                f"#### Episode {episode + 1}/{num_episodes} finished
    with cumulative reward: {episode_reward} and survivors rescue rate:
    {1 - len(self.env.survivors) / self.env.num_survivors} ####")
            time.sleep(0.8)
            print()
            clear_output(wait=True)
        elif print_mode == 1: # Print only final episode stats
            if episode == num_episodes - 1:
                print(
                    f"#### Episode {episode + 1}/{num_episodes}
    finished with cumulative reward: {episode_reward} and survivors
    rescue rate: {1 - len(self.env.survivors) / self.env.num_survivors}
    ####")
        else: # Print nothing
            pass

    return rewards
```

Listing 4.6: Q-Learning Function

### Visualization

The agent's learning process can be visualized using the `visualize_training` method. This method renders the environment and prints out important information at each step, such as the reward, battery life, and the number of survivors left.

```
def visualize_training(self, reward, episode_reward, done):
    self.env.render(mode='human')
    print(
        f"Step Reward: {reward}, Cumulative Episode Reward: {
    episode_reward}, Battery Life: {self.env.battery_life}, Survivors
    Left: {len(self.env.survivors)}")
    print()

    clear_output(wait=True)

    if not done:
        time.sleep(0.5)
```

Listing 4.7: Learning Visualization Function

### Q-table Retrieval

Finally, the agent's Q-table can be retrieved using the `get_q_table` method for further analysis or for use in decision-making processes.

```
def get_q_table(self):
    return self.q_table
```

Listing 4.8: Q-table Retrieval Function

In summary, I have presented a comprehensive description of our RL agent, outlining key components and their respective functions. This implementation, though complex, provides a robust and adaptable system that demonstrates the effectiveness of Q-learning in achieving optimal policies. Our agent leverages the mathematical concepts explained in Sections 2.1.2, 4.1, and combines them with Python programming to interact and learn from its environment. I have strived for clarity and reproducibility in our work, and I believe that our agent's design and methodology could serve as a valuable reference for future research and development in reinforcement learning. Our focus now shifts towards analyzing the performance and results generated by our agent in the experiments that follow.

# Chapter 5

# Symbolic Knowledge Extraction

In this chapter, I delineate the extraction of symbolic knowledge, which is the main goal of our research, from the learned policy of our Q-learning agent, discussed in Section 4.1.2. The emphasis is on the conversion of implicit knowledge, contained in the agent's Q-table, into explicit symbolic IF-THEN rules that succinctly encapsulate the agent's learned policy. Our rule extraction algorithm draws inspiration from the symbolic knowledge extraction approach proposed by Ron Sun [11], adapted to our specific context. It encompasses several steps: normalization of Q-values, k-means clustering, and IF-THEN rule generation.

## 5.1 Overview

As discussed in Section 2.2, all the proposed knowledge extraction approaches have their own advantages and disadvantages, depending on the problem domain and the structure of the learned model. In this thesis, I focus on Ron Sun's [11] approach due to its simplicity and effectiveness in extracting rules from Q-learning agents. However, the other approaches mentioned also offer valuable insights and advancements in the field of neuro-symbolic AI and can be considered for future work and research.

### Ron Sun's Model

One of the essential concepts I leverage in this chapter comes from the two-level model developed by Sun [11]. This model incorporates a bottom level implementing Q-learning and a top level, a localist rule net. The interaction between these two levels allows for the extraction of rules which can be used in sequential decision making tasks.

For my purposes, I specifically focus on the bottom level, which calculates Q-values. Although Sun utilized a four-layered network in his original model, my algorithm employs a simpler Q-learning approach. This approach, in combination with k-means clustering, enables us to extract the rules from Q-values.

The core of the rule extraction process is described in Sun's high-level pseudocode algorithm:

**Algorithm 2:** Knowledge Extraction from Reinforcement Learning [11]

1 **do**
2 Observe the current state $x$;
3 Compute in the bottom level the Q-values of $x$ associated with each of all the possible actions $a_1, a_2, \ldots, a_n$: $Q(x, a_1), Q(x, a_2), \ldots, Q(x, a_n)$;
4 Find out all the possible actions $b_1, b_2, \ldots, b_m$ at the top level based on the input $x$ and the rules in place;
5 Compare the values of $a_i$'s with those of $b_j$'s, and choose an appropriate action $b$;
6 Perform the action $b$, and observe the next state $y$ and (possibly) the reinforcement $r$;
7 Update the bottom level in accordance with Q-Learning;
8 Update the top level with *Rule-Extraction-Revision*;
9 **while** *Termination condition is met*;

While steps 3, 4, and 7 of Sun's model involve the interaction with the top level (localist rule net), in my approach, these steps are replaced with k-means clustering and the generation of IF-THEN rules. These modifications are due to my goal of making the rules easily interpretable without requiring integration with a neural network.

The rule extraction in Sun's model is guided by the concept of Information Gain (IG) which is calculated as follows:

$$IG(A, B) = log_2 \frac{PM_a(A) + 1}{PM_a(A) + NM_a(A) + 2} - log_2 \frac{PM_a(B) + 1}{PM_a(B) + NM_a(B) + 2} \tag{5.1}$$

Where $A$ and $B$ are two different conditions that lead to the same action $a$. $PM_a$ represents Positive Matches and $NM_a$ Negative Matches.

In my context, the ***positive*** and ***negative*** matches are replaced with clustering and labelling processes.

The specifics of rule extraction, revision, expansion, shrinking, and deletion in Sun's model provide an interesting perspective but are replaced in my model by the generation of simple IF-THEN rules based on the clusters found in Q-values.

### K-means Clustering

K-means is a widely used clustering algorithm due to its simplicity, efficiency, and empirical success. The objective of k-means clustering is to partition $n$ observations into $k$ clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.

Mathematically, given a set of observations $(x_1, x_2, ..., x_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the $n$ observations into $k(\leq n)$ sets $S = \{S_1, S_2, ..., S_k\}$ so as to minimize the within-cluster sum of squares (WCSS) (i.e., variance). The objective function is:

$$\arg\min_{S} \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2 \tag{5.2}$$

where $\mu_i$ is the mean of points in $S_i$.

The algorithm proceeds as follows:

**Algorithm 3:** K-means Clustering Algorithm [8]

**Input:** Dataset $X$, Number of Clusters $k$
**Output:** Cluster assignments for each data point in $X$

```
Randomly initialize k centroids: c_1, c_2, ..., c_k
repeat
    foreach data point x in dataset X do
        Assign x to the nearest centroid c
        This forms k clusters
    end foreach
    foreach cluster i = 1, 2, ..., k do
        Compute the new centroid μ_i as the mean of all data points in cluster i
        Set c_i = μ_i
    end foreach
until assignment of data points to clusters no longer changes
```

The reason for choosing the k-means clustering in my approach is its ability to handle large datasets efficiently, which is important in reinforcement learning scenarios where the state-action space can be large. It creates clear, non-overlapping clusters that map well to the IF-THEN rules I aim to extract. However, the number of clusters $k$ is an input parameter: inappropriate choice of $k$ can lead to poor clustering performance.

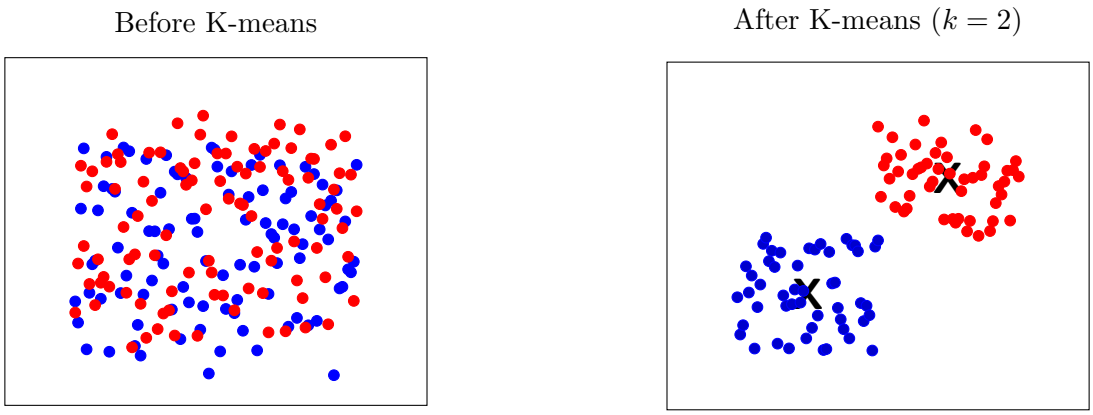


Figure 5.1: K-means Clustering [8]

## 5.2 Algorithm

My symbolic knowledge extraction algorithm is composed of three primary steps. First, I normalize the Q-values of the Q-table. Then, we apply k-means clustering on the normalized Q-values to segregate them into distinct clusters. Finally, I transform these clusters into a set of interpretable IF-THEN rules. The following subsections detail each of these steps.

### 5.2.1 Normalizing Q-values

As discussed in Section 4.1.2, the Q-values in the Q-table represent the expected return of taking a particular action in a specific state. As the agent interacts more with the environment, these Q-values keep evolving until they eventually converge. However, these Q-values may range from very small to very large values, which might make the clustering process challenging.

Normalization is a technique used to change the values of numeric columns in the dataset to use a common scale, without distorting differences in the ranges of values or losing information. This process is essential before performing clustering because it ensures that each feature contributes approximately proportionately to the final distance.

I employ min-max normalization, which transforms the Q-values so that they fall within the range of 0 and 1. The normalization function is defined as:

$$Q'_{sa} = \frac{Q_{sa} - \min(Q_{sa})}{\max(Q_{sa}) - \min(Q_{sa})} \tag{5.3}$$

Where $Q'_{sa}$ is the normalized Q-value of action $a$ in state $s$, and $Q_{sa}$ is the original Q-value. $\min(Q_{sa})$ and $\max(Q_{sa})$ are the minimum and maximum Q-values of action $a$ in state $s$, respectively.

### 5.2.2 K-means Clustering

I employ the k-means clustering algorithm on the normalized Q-values. K-means, as discussed in Section 5.1, is a widely employed clustering algorithm capable of partitioning data into a pre-specified number of clusters based on feature similarity. In this context, For each state $s$, we consider the Q-values of each action as a feature, resulting in a $d$-dimensional vector, where $d$ is the number of possible actions.

I then perform k-means clustering on these vectors to group similar states together. The optimal number of clusters, $k$, is an important parameter and is chosen empirically. An appropriate choice of $k$ should lead to clusters that represent different ***policies*** that the agent follows in different regions of the state space. For my problem, we found $k = 10$ to be an adequate balance between granularity and complexity.

### 5.2.3 Generating IF-THEN Rules

Once I obtain the clusters, the final step is to transform these clusters into a set of interpretable IF-THEN rules. The rationale behind this step is to provide a transparent and interpretable representation of the agent's learned policy. I achieve this by associating each cluster with an action that has the highest average Q-value within the cluster.

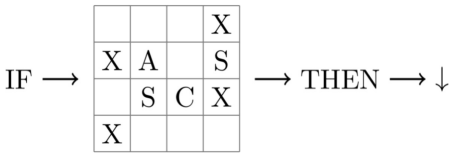


Figure 5.2: IF-THEN Rule [6]

For each cluster $c$, I calculate the average Q-values for all actions across all states in $c$. I then choose the action $a$ that has the highest average Q-value. The resulting rule is:

$$\text{IF state } s \text{ is in cluster } c, \text{ THEN take action } a \tag{5.4}$$

where $a = \text{argmax}_{\text{action}}(\text{average Q-values for all actions in } c)$

This process results in a simple, interpretable set of rules that represent the learned policy of the Q-learning agent. These rules provide insights into the agent's decision-making process and could be used for various purposes such as policy analysis, transfer learning, and more.

## 5.3 Implementation Details

My choice of Ron Sun's symbolic knowledge extraction algorithm is influenced by its capability to produce a transparent and easily interpretable representation of an agent's learned policy [11]. Unlike some other rule extraction algorithms, Sun's approach does not make strong assumptions about the underlying policy and can accurately handle complex situations with multiple overlapping rules. Moreover, the algorithm's simplicity and computational efficiency make it a practical choice for my task.

The implementation of my rule extraction algorithm is a Python-based solution that leverages the utilities provided by the NumPy library for numerical computations and the sklearn library for k-means clustering. The primary functions include normalization of Q-values, application of the k-means algorithm, and generation of IF-THEN rules. [1]

The generated rules are written into a text file for easy access and analysis. The rules are presented in a human-readable format, with each state of the environment represented as a grid. Each cell in the grid corresponds to a position in the environment and is represented by a specific symbol: an empty cell is `'.'`, a wall is `'X'`, a survivor is `'S'`, a charging station is `'C'`, and the agent is `'A'`. The code excerpts below detail the steps I have taken [6].

[1] *This implementation assumes a grid-based environment, where states are represented by a two-dimensional grid, and the agent's actions are Up, Down, Left, and Right.*

### Data Preparation

Before normalizing the Q-values, I extract them from the Q-table. I then transform these Q-values into a NumPy array for efficient numerical operations.

```
# Extract the Q-values from the Q-table
q_values = [vals for vals in agent.q_table.values()]
```

Listing 5.1: Data Preparation

### Normalization

I normalize the Q-values by dividing each value by the maximum Q-value in the Q-table. This operation scales the Q-values between 0 and 1.

```
# Normalize the Q-values
normalized_q_values = np.array(q_values) / np.max(q_values)
```

Listing 5.2: Normalization

### Clustering

I use the KMeans function provided by the sklearn library to perform k-means clustering on the normalized Q-values. I set the number of clusters to 20 and initialize the clusters' centroids 10 times to find the best clustering solution.

```
# Perform k-means clustering on the normalized Q-values
k = 20
kmeans = KMeans(n_clusters=k, n_init=10, random_state=0).fit(
    normalized_q_values)
```

Listing 5.3: Clustering

### Rule Generation

I generate IF-THEN rules for each cluster, associating states with the action that has the highest average Q-value in the cluster. The resulting rules are then stored in a Python list.

```
# Generate IF-THEN rules for each cluster
rules = []
for cluster_idx in range(k):
    cluster_states = [(state, agent_pos) for ((state, agent_pos), label
    )
        in zip(agent.q_table.keys(), kmeans.labels_) if label ==
    cluster_idx]
    cluster_actions = [np.argmax(q_vals) for q_vals in kmeans.
    cluster_centers_]

```

```
    rule = {
        'states': cluster_states,
        'action': cluster_actions[cluster_idx],
    }
    rules.append(rule)
```

Listing 5.4: Rule Generation

### Output Generation

Finally, I write the resulting rules to a text file. Each rule is represented by the states it covers and the associated action, which is encoded as a string representing a direction (`Up, Down, Left, Right`).

```
# Write the output to a text file
with open('output.txt', 'w') as file:
    for rule in rules:
        file.write('IF\n\n')
        for state, agent_pos in rule['states']:
            ...
        file.write(
            f"THEN {['Up', 'Down', 'Left', 'Right'][rule['action']]}\n\n")
        file.write("----------\n\n")

print("Output written to 'output.txt' successfully!")
```

Listing 5.5: Output Generation

For example, the output may look like this:

```
IF

. . . C
A S . C
. C C .
X . . C

. . S C
A C C .
. S S C
X C . S

THEN Right
```

Figure 5.3: Symbolic Knowledge Output [6]

Where this rule states that if the agent `'A'` is in either of the two states described (in this cluster), the optimal action to take is to move `'Right'`.

# Chapter 6

# Results and Discussion

This chapter discusses the results obtained from the training and testing phases of the proposed approach. It provides an analysis of the training performance and the quality of the symbolic knowledge extracted from the Q-table. It also highlights the implications and insights derived from these results.

## 6.1 Evaluation Criteria

The performance of the Q-Learning agent in the *DisasterZone* environment and the quality of the symbolic knowledge are evaluated based on two key metrics:

- **Cumulative Reward:** The agent receives a reward for each survivor successfully rescued and returned to the safe zone. The cumulative reward is calculated as the sum of the rewards received throughout an episode, which reflects the agent's ability to maximize its reward and hence achieve its objectives.
- **Valid Rules:** The percentage of valid rules versus rule violations in the testing phase, which provides an indication of the quality of the symbolic knowledge extracted from the Q-table.

## 6.2 Training Performance and Cumulative Reward

The Q-Learning agent was trained on our customized *DisasterZone* environment for a total of 1 million and 500 episodes. This environment was characterized by a grid size of 4, with 5 obstacles, 4 survivors, 2 charging stations, and a maximum battery life of 20. Despite the complexity of the environment, the agent managed to complete the training phase on a CPU in approximately ***18*** minutes.

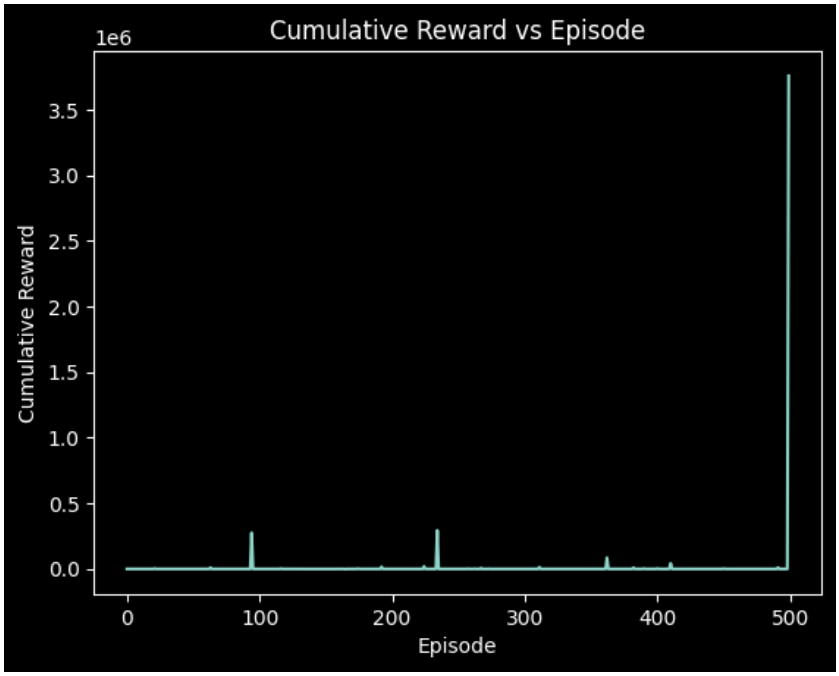


Figure 6.1: Cumulative Reward vs 500 Episodes [6]

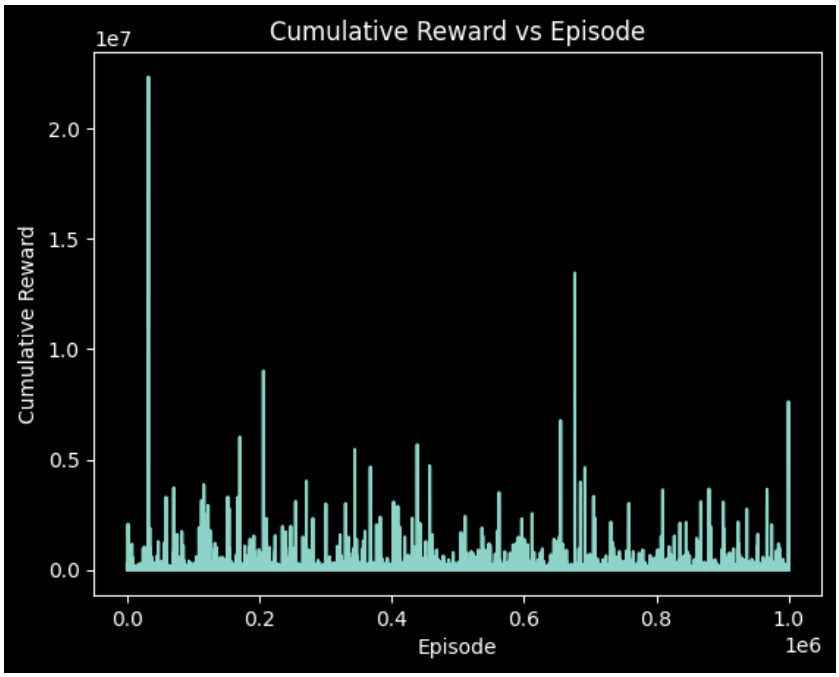


Figure 6.2: Cumulative Reward vs 10 million Episodes [6]

The cumulative reward, plotted against the number of episodes, displayed the learning trajectory and performance of the agent. This plot served as a crucial tool in observing the agent's learning process and its increasing proficiency in maximizing rewards.

## 6.3 Quality of Extracted Symbolic Knowledge

Upon completion of the training phase, the agent's Q-table was converted into a set of IF-THEN rules using k-means clustering. Despite taking around ***7*** minutes to complete, this process resulted in approximately ***30*** million lines of IF-THEN rules.

### 6.3.1 Rule Validation Script

To assess the quality of these rules, a custom `RuleValidator` class was implemented from scratch specially for our extracted rules [6]. This class takes an environment and a file containing rules as input, and reads the rules from the file. It then validates each rule by simulating the action suggested by the rule on the corresponding state in the environment. If the action results in a state change or a successful rescue operation, without hitting an obstacle, the rule is considered valid. However, if the action does not cause a state change, or if the agent hits an obstacle, the rule is considered a violation.

The `RuleValidator` class consists of several methods:

1. **Constructor:** The `RuleValidator` constructor initializes the instance with an environment and a list of rules read from a file using the `read_rules_from_file` method.
2. **read_rules_from_file:** This method reads the symbolic rules from a given file. It parses the IF-THEN rule format, translating the symbols into numerical values that the environment can understand, and packs the resulting rules into a dictionary. It also keeps track of the agent's position in the environment, which is critical for validating the rules.
3. **print_rules:** The `print_rules` method provides a utility function to print all the rules. This is especially useful for debugging purposes and to visually inspect the rules.
4. **validate_rules:** This method is the crux of the class, where each rule is validated by simulating its action on the corresponding state in the environment. A rule is considered valid if the action causes a change in state or a successful rescue operation, without hitting an obstacle. The method returns the percentage of valid rules and rule violations.
5. **visualize_validation_results:** This method visualizes the results of the validation process. It plots a bar chart to depict the percentages of valid rules and rule violations, providing an intuitive and visual perspective of the rule validation results.

#### Constructor and Reading Rules from a File

The `RuleValidator` class starts with a constructor that accepts an environment and a rules file. It initializes the instance variables and reads the rules from the provided file using the `read_rules_from_file` method.

```
1 class RuleValidator:
2     def __init__(self, env, rules_file):
3         self.env = env
4         self.rules = self.read_rules_from_file(rules_file)
```

Listing 6.1: Constructor

The `read_rules_from_file` method reads the symbolic rules from the file. It parses the file line by line, interprets the rules in the IF-THEN format, translates the symbols into numerical values that the environment can understand, and packs the resulting rules into a dictionary.

```
    def read_rules_from_file(self, filename):
        rules = []
        with open(filename, 'r') as file:
            lines = file.readlines()

        rule = {'states': [], 'action': None}
        state = []
        agent_pos = (None, None)
        for line in lines:
            stripped_line = line.strip()

            if stripped_line.startswith("IF"):
                if rule['states']:  # If it's not the first rule
                    rules.append(rule)
                    rule = {'states': [], 'action': None}
            elif stripped_line.startswith("THEN"):
                action_str = stripped_line.split()[1]
                rule['action'] = {"Up": 0, "Down": 1, "Left": 2, "Right
": 3}[action_str]
            elif stripped_line and not stripped_line.startswith("-"):
                state_line = list(map(lambda cell: {"A": 4, ".": 0, "X"
: 1, "S": 2, "C": 3}[cell], stripped_line.split()))
                state.append(state_line)
            elif state:  # empty line after a state
                for i in range(len(state)):
                    for j in range(len(state[i])):
                        if state[i][j] == 4:  # If it's the agent's
cell
                            agent_pos = (i, j)
                            state[i][j] = 0  # Set the cell to empty as
 the agent's position is already stored
                rule['states'].append((np.array(state), agent_pos))  #
add a copy of the state to the rule
                state = []
                agent_pos = (None, None)

        # Add the last rule
        if rule['states']:
            rules.append(rule)
        return rules
```

Listing 6.2: Reading Rules from a File

### Printing Rules

The `print_rules` method provides a utility function to print all the rules, which is useful for debugging purposes and visually inspecting the rules.

```
def print_rules(self, rules):
    for rule in rules:
        print("IF\n")
        for state, agent_pos in rule['states']:
            for i, row in enumerate(state):
                row_str = []
                for j, cell in enumerate(row):
                    if (i, j) == agent_pos:
                        row_str.append("A")
                    else:
                        row_str.append("." if cell == 0 else "X" if
    cell == 1 else "S" if cell == 2 else "C" if cell == 3 else ".")
                print(" ".join(row_str))
            print()
        print(f"THEN {['Up', 'Down', 'Left', 'Right'][rule['action']]}\
    n")
        print("----------\n")
```

Listing 6.3: Printing Rules

### Validating Rules

The `validate_rules` method is where each rule is validated by simulating its action on the corresponding state in the environment. A rule is considered valid *if-and-only-if* the action causes a change in state or a successful rescue operation, without hitting an obstacle. The method returns the percentage of valid rules and rule violations. The method is further explored below.

This section defines a function called `validate_rules` which will be a method of a class (given the self parameter). It then initializes three variables: `valid_count`, `obstacle_violation_count`, and `static_violation_count`. These will be used later to keep track of the validity of the rules applied to the environment.

```
def validate_rules(self):
    valid_count = 0
    obstacle_violation_count = 0
    static_violation_count = 0
```

Listing 6.4: Defining Function and Initializing Variables

This part of the code iterates over each rule in `self.rules`. For each rule, it also iterates over each state and corresponding agent position specified in the rule's cluster of states. It then updates the environment's grid and agent's position to match the current state and agent position.

```
for rule in self.rules:
    for state, agent_pos in rule['states']:
        self.env.grid = state.copy()
        self.env.agent_pos = agent_pos
```

Listing 6.5: Iterating Over Rules and Setting Up Environment

The initial state of the environment is stored, including the grid configuration, the agent's position, and the list of survivors. This is needed to compare the changes that occur in the environment after the rule is applied.

```
initial_state = self.env.grid.copy()
initial_agent_pos = self.env.agent_pos
initial_survivors = self.env.survivors.copy()
```

Listing 6.6: Storing Initial Environment

This part of the code applies the action specified by the current rule to the environment using the step method. It returns a tuple where the second value is the reward from the action and the fourth value is additional information about the step (e.g., if an obstacle was hit).

```
_, reward, _, info = self.env.step(rule['action'])
```

Listing 6.7: Applying Rule

Here, the changes made by applying the rule are examined to determine if the rule is valid or not. It first checks if the state of the environment or the agent's position has changed. Then it checks if any survivors were rescued and if an obstacle was hit. Based on these checks, it increments the count of valid actions, actions that hit obstacles, or actions that didn't change the state.

```
state_changed = not np.array_equal(initial_state, self.env.grid) or
    initial_agent_pos != self.env.agent_pos
survivors_rescued = len(initial_survivors) > len(self.env.survivors)
hit_obstacle = info['hit_obstacle']

if state_changed:
    if survivors_rescued:
        valid_count += 1
    else:
        if hit_obstacle:
            obstacle_violation_count += 1
        else:
            valid_count += 1
else:
    static_violation_count += 1
```

Listing 6.8: Checking Rule Validation

After each rule application, the environment is reset to its initial state, so the next rule can be evaluated independently.

```
self.env.grid = initial_state
self.env.agent_pos = initial_agent_pos
self.env.survivors = initial_survivors
```

Listing 6.9: Resetting the Environment

The total count of actions is calculated by summing the valid actions, actions that hit obstacles, and actions that didn't change the state. The percentages of each are then calculated and passed to a function to visualize these results. The function finally returns the percentages of valid actions, obstacle violations, and static violations.

```
total_count = valid_count + obstacle_violation_count +
    static_violation_count

valid_percentage = valid_count / (total_count) * 100
obstacle_violation_percentage = obstacle_violation_count / (total_count
    ) * 100
static_violation_percentage = static_violation_count / (total_count) *
    100

self.visualize_validation_results(valid_percentage,
    obstacle_violation_percentage, static_violation_percentage)

return valid_percentage, obstacle_violation_percentage,
    static_violation_percentage
```

Listing 6.10: Calculating and Printing Results

### Visualizing Validation Results

Finally, the `visualize_validation_results` method provides a visual representation of the rule validation results. It plots a bar chart to depict the percentages of valid rules and rule violations.

```
def visualize_validation_results(self, valid_percentage,
    obstacle_violation_percentage, static_violation_percentage):
    categories = ['Valid', 'Invalid']
    valid_percentages = [valid_percentage]
    invalid_percentages = [obstacle_violation_percentage,
    static_violation_percentage]

    # Set custom colors for each category
    valid_color = 'green'
    obstacle_color = 'red'
    movement_color = 'gray'

    # Plotting valid bars with green color
```

```
12      plt.bar(categories[0], valid_percentages[0], color=valid_color,
        label='Valid')
13
14      # Plotting invalid bars with different colors
15      plt.bar(categories[1], invalid_percentages[0], color=obstacle_color
        , label='Obstacle Hit')
16      plt.bar(categories[1], invalid_percentages[1], bottom=
        invalid_percentages[0], color=movement_color, label='No Movement')
17
18      plt.xlabel('Categories')
19      plt.ylabel('Percentage')
20      plt.title('Validation Results')
21      plt.legend()
22      plt.show()
```

Listing 6.11: Visualizing Validation Results

This function helps to intuitively understand the effectiveness and quality of the generated rules. This overview should provide a comprehensive understanding of the rule validation script and its different parts.

Running the rule validation script on the generated rules yielded a valid rule percentage of approximately ***31.92%*** and a violation percentage of around ***68.08%***, where ***11.38%*** of these were hit obstacles violations and ***56.69%*** were no movement ones. These figures point to the fact that while some of the symbolic knowledge extracted was useful, a significant proportion of the rules resulted in either ineffective or undesirable outcomes.

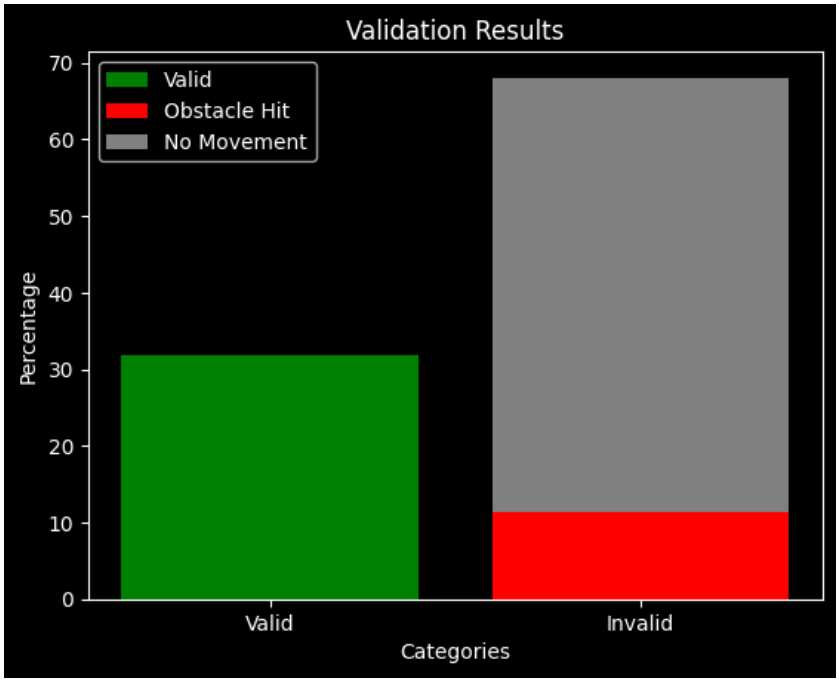


Figure 6.3: Valid Rules vs Rule Violations [6]

In summary, the `RuleValidator` class serves as an effective tool for validating the symbolic rules extracted from the Q-table. The results of this validation process provide valuable insights into the quality of the generated rules and highlight areas for improvement in the learning and knowledge extraction process.

## 6.4 Implications and Insights

The relatively high violation percentage in the extracted rules can be attributed to a number of potential factors. First, the complexity of the *DisasterZone* environment may make it challenging for the agent to learn effective strategies within the given number of training episodes. Second, the process of transforming continuous Q-values into discrete rules via clustering may inherently introduce some inaccuracies or loss of information. Third, the use of a deterministic rule validation approach might be overly strict in an environment where stochasticity is present. Finally, the reward function might require some modifications to motivate the agent to take an action instead of staying still.

Despite these challenges, the fact that approximately ***one-third*** of the rules were deemed valid indicates that the approach can indeed generate meaningful symbolic knowledge. This provides a valuable proof of concept and a foundation for further refinement and improvement.

# Chapter 7

# Conclusion

As this thesis reaches its conclusion, it is imperative to retrospectively examine the research journey traversed. The exploration of knowledge extraction from trained Reinforcement Learning (RL) agents, although computationally intricate, provides profound insights and presents significant opportunities for artificial intelligence advancement. The developed capacity to translate the opaque mechanisms of these RL agents into transparent, interpretable rules promises to enhance the intelligibility and trust in AI systems. This final chapter is devoted to summarizing the principal contributions of this study, addressing its inherent limitations, and outlining potential avenues for future investigations.

## 7.1 Summary of Contributions

In this thesis, I presented a comprehensive study on the extraction of symbolic knowledge from trained Reinforcement Learning (RL) agents. I reviewed the fundamental concepts of RL, its methodologies, and its applications in complex environments. I then discussed the gap between the black-box nature of these learning algorithms and the desire for explainable, transparent models that can be easily understood by humans.

I provided an in-depth explanation of a novel approach to extract symbolic knowledge in the form of IF-THEN rules from trained Q-Learning agents. This approach consists of four major steps: Data Preparation, Normalization, Clustering, and Rule Generation. This methodology was demonstrated in the context of a grid-world rescue game, a complex environment where an agent navigates a grid, rescuing survivors while avoiding obstacles.

A validation script was implemented from scratch to evaluate the quality and validity of the extracted rules. I tested the rules in the same environment and presented a quantitative measure of their performance, confirming the effectiveness of the rule extraction methodology.

## 7.2 Limitations and Future Work

I limit my investigation to the specific challenges and constraints present in the *DisasterZone*, which may not fully capture the complexity of real-world disaster response scenarios. Additionally, the agent's performance is restricted by the computational resources available, as training

takes place on a CPU rather than a GPU. Consequently, the agent's training time may be impacted by the complexity of the environment. Despite these limitations, my work aims to provide valuable insights into the potential applications of reinforcement learning and symbolic knowledge extraction in disaster response and search and rescue operations.

To enhance the quality of the extracted rules, future work could explore alternative methods for clustering or rule generation, incorporate more sophisticated validation criteria, or allow for a longer training period. Furthermore, it would be interesting to assess the interpretability of the generated rules and their potential usefulness in different application contexts.

The validation method is dependent on the complexity of the environment, and while it worked well for the grid-world rescue game, it may not be general enough for more complex environments with more varied and nuanced states.

Additionally, while Q-Learning was used as the RL algorithm in this study, there are many other RL algorithms that may yield different results or require slight modifications to the methodology. For example, if an agent is trained using a deep RL algorithm such as Deep Q-Networks (DQNs), the rules extracted might be more complex and could potentially be of higher quality.

Lastly, different clustering algorithms could be experimented with during the rule extraction process. The choice of clustering algorithm and the number of clusters could potentially affect the quality and applicability of the extracted rules.

These enhancements could potentially reduce the percentage of rule violations, thereby improving the quality of the symbolic knowledge extracted. An extensive validation scheme accounting for the stochastic nature of the environment might also yield more realistic results. The improved set of rules would not only offer better performance, but it would also provide more intuitive and interpretable knowledge for humans to understand and learn from.

## 7.3 Final Remarks

The quest for transparency and interpretability in Machine Learning models, and particularly in RL agents, is a challenging but crucial endeavour. This thesis has contributed towards this objective by developing a methodology for extracting human-readable symbolic knowledge from RL agents. Despite its limitations, the methodology provides a solid foundation for future research and improvements in this field. By continuing the exploration of this intersection between RL and symbolic knowledge extraction, I can work towards creating RL agents that are not only effective in their tasks but also understandable and transparent to their human users.

# Appendix

# Appendix A

# Lists

# List of Figures

# List of Tables

# Listings

# List of Algorithms

# Bibliography


[1] Daan Apeldoorn and Gabriele Kern-Isberner. An agent-based learning approach for finding and exploiting heuristics in unknown environments. In *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 298–305. IEEE, 2019.

[2] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[4] Filippo Cavallo, Valeria Semeraro, Laura Fiorini, Daniele Magistro, Andrea Monteriú, and Raffaele Limosani. Robotic technologies and fundamental open issues in robotics. *Biosystems & Biorobotics*, 21:7–19, 2018.

[5] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[6] Ahmed Khaled. Learning of symbolic representations for rescue scenarios in disaster zones (https://github.com/ahmillect/Learning-of-Symbolic-Representations-for-Rescue-Scenarios-in-Disaster-Zones), 2023.

[7] Stefan Kramer. A brief history of learning symbolic higher-level representations from data (and a curious look forward). *KI-Künstliche Intelligenz*, 30(1):89–95, 2016.

[8] James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, pages 281–297. Oakland, CA, USA., 1967.

[9] Robin R Murphy. Disaster robotics. In *MIT Press*, 2014.

[10] A. Smolka. Natural disasters and the challenge of extreme events: Risk management from an insurance perspective. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 364:2147–65, 09 2006.

[11] Ron Sun. Knowledge extraction from reinforcement learning. *Machine learning*, 35(1-2):79–111, 1999.

[12] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[13] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[14] Qi Zhang, Jie Shi, Jiawei Cheng, and Bo Liu. Learning symbolic representations from unlabeled data using hybrid neural networks. *Neurocomputing*, 384:155–167, 2020.