Structured Multimodal Reinforcement Learning for Playing NetHack

A Thesis Submitted to the Department of Computer Science and Communications Engineering, the Graduate School of Fundamental Science and Engineering of Waseda University in Partial Fulfillment of the Requirements for the Degree of Master of Engineering

Submission Date: January 23rd, 2023

Keisuke Izumiya
5121F009-6

Advisor: Assoc. Prof. Edgar Simo-Serra
Research guidance: Research on Computer Graphics

Abstract

Designing autonomous agents that play video games is important and valuable for game design, balancing, and testing. Video games are also preferable benchmark environments for developing robust Reinforcement Learning (RL) algorithms that can be applied to other real-world tasks. In this thesis, we address the modification of RL algorithms for NetHack, a type of video game. Specifically, we discuss three methods: appropriate inventory handling, appropriate in-game strings handling, and efficient learning with expert data. Inventory is an unordered set of items of variable size which is not straightforward to handle in the RL framework. To solve this problem, we propose a mechanism that uses attention and meta-action to test its effectiveness. Handling in-game strings in RL requires ingenuity because of the unique vocabulary and the need to consider the importance of strings. We examine different models for handling in-game strings and compare their performance. Expert data supports RL and can be provided naturally in game development, but using a large amount of expert data is impractical. Therefore, we propose a method to facilitate RL using a small amount of expert data and test its effectiveness.

概要

ビデオゲームをプレイする自律エージェントは，ゲームデザインやゲームバランスの調整，テストに役立つ重要なものである．また，ビデオゲームは他の実世界のタスクに適用できる頑健な強化学習アルゴリズムを開発するための好ましいベンチマーク環境である．そこで本論文では，ビデオゲームの一種である NetHack 環境に対して適用する強化学習アルゴリズムの改良を行う．具体的には，適切なインベントリ処理，適切なゲーム内文字列処理，エキスパートデータを用いた効率的な学習の三つの方法について議論する．インベントリは順序を持たないアイテムの集合であり，大きさも可変であることから，強化学習フレームワークで扱うのは容易でない．この問題を解決するために，注意機構とメタ行動を用いた手法を提案し，その有効性を検証する．強化学習でゲーム内の文字列を扱う場合，ゲーム特有の語彙や文字列の重要性を考慮する必要があるため，工夫が必要である．そこで，我々はゲーム内の文字列を扱うための様々なモデルを検証し，その性能を比較する．熟練者によるデータは強化学習を助けるものであり，ゲーム開発において自然に用意することが可能だが，大量のデータを用意するのは非現実的な設定である．そこで，少量の熟練者データを用いて強化学習を促進する方法を提案し，その有効性を検証する．

Acknowledgements

I would like to thank associate professor Simo-Serra and the lab member, especially Kotaro and Tsunehiko. I would also like to thank all of my friends and family who helped me outside of my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reinforcement Learning (RL), a branch of machine learning, has a general problem setting and the advantage of being able to learn without labeling data. Because of these characteristics, RL, especially Deep Reinforcement Learning (DRL), which combines deep learning and RL, is being studied in various fields, such as automated driving, advertising optimization, and searching for the optimal design of machine learning models and parameters. Among these, DRL methods have shown excellent performance, especially in games such as Atari 2600 [1], Go [2], and StarCraft II [3]. Designing autonomous agents to play games is important and beneficial for game design, balancing, and testing. Games are also desirable benchmark environments for developing robust RL algorithms that can be applied to real-world tasks other than video games.

- They can be fully computer-simulated and provide reliable experimental results.
- The task content, difficulty, and other settings can be changed relatively freely, allowing experiments to be conducted in various environments.
- The objective of the task is clear, and it is often easy to evaluate the intermediate stages.

However, RL has many problems, and one of the significant problems is its low performance. Compared to supervised learning, which has achieved a practical level of performance in many fields, there are only a limited number of fields in which RL can achieve state-of-the-art performance. Even in games, where RL excels, it has not shown sufficient performance for games with problems that even the most advanced RL methods have not completely overcome, such as sparse and delayed rewards.

In this thesis, we work on modifying RL algorithms for designing autonomous agents in the context of NetHack, a type of roguelike game. A roguelike is a role-playing game in which the player controls a character corresponding to themselves to achieve a goal while overcoming challenges. Although the definition of roguelike has not been established with certainty, the

Berlin Interpretation [4] presented at the International Roguelike Development Conference 2008 listed the following characteristics common to existing roguelike games.

- Most of the game contents, including dungeon structure and item placement, is randomly generated each time the game starts.
- The player investigates the identity of dungeons and items through gameplay.
- Once you die or complete the game, you have to start over from the beginning.
- There is a wide variety of items, enemies, and multiple strategies to clear the game.
- The game is turn-based and does not require real-time input.
- The game requires the player to manage limited resources and think about utilizing them.
- The player collects resources by fighting a large number of enemies.

From an RL perspective, it is a highly challenging environment with difficulties such as a huge state space, sparse rewards, and the need for short-term and long-term strategies that current state-of-the-art deep reinforcement learning methods have yet to exploit fully. NetHack, the subject of this thesis, is one of the most popular open-source roguelikes and is still being updated even though it is the first roguelike with a history of over 30 years. NetHack is described in detail in Section 2.4.

1.1 Contributions

The following are contributions of this thesis.

Inventory Management We proposed a method to properly handle inventories (sets of items) with permutation-invariant features with neural networks and incorporate them into the RL framework. The agent performed better than existing methods, indicating that it can handle items appropriately.

In-Game String Handling We compared several string handling methods that consider the features of in-game strings. We show that the string methods do not significantly affect the learning results.

Reinforcement Learning with Expert Data We proposed an efficient method for RL with a small amount of expert data. We show that introducing expert data improves the performance of the proposed method, and the proposed method performs more human-like and appropriate behavior.

1.2 Publications

The following are the publications related to this thesis.

- **K. Izumiya** and E. Simo-Serra, "Inventory Management with Attention-Based Meta Actions," in *Proc. IEEE Conference on Games*, Aug. 2021.
- **K. Izumiya** and E. Simo-Serra, "Item Management Using Attention Mechanism and Meta Actions in Roguelike Games," in *Proc. Visual Computing*, Sep.–Oct. 2021.

1.3 Thesis Overview

The overview of this thesis is as follows:

Chapter 2 We describe the prerequisites and related research for this paper as a whole.
Chapter 3 We describe methods for handling inventory in roguelike games.
Chapter 4 We describe a method for handling strings in roguelike games.
Chapter 5 We describe a method for efficient learning with a small amount of expert data.
Chapter 6 We conclude the paper and discuss prospects.

Chapter 2

Background

In this chapter, we provide background knowledge on areas such as neural networks and reinforcement learning that will be used throughout this thesis.

2.1 Neural Network

Neural networks model the workings of the human brain and are often used as a model to represent the function $f(\boldsymbol{x}; \boldsymbol{w})$, which is determined by the weights $\boldsymbol{w}$.

2.1.1 Multilayer Perceptron

A MultiLayer Perceptron (MLP) is a neural network consisting of one or more fully-connected layers. A fully-connected layer is a layer with weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$, and the output $\boldsymbol{y}$ is calculated from the input $\boldsymbol{x}$ by

$$\boldsymbol{y} = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{2.1}$$

where sigma is the activation function. For the activation function $\sigma\colon \mathbb{R} \to \mathbb{R}$, the sigmoid function $\varsigma\colon \mathbb{R} \ni x \mapsto 1/\{1 + \exp(-x)\} \in (0, 1)$ and the ReLU function $\mathrm{relu}\colon \mathbb{R} \ni x \mapsto \max(0, x) \in [0, \infty)$ are often used. Note that when applying the $\mathbb{R} \to \mathbb{R}$ function to a vector or matrix, the function is applied element by element. Vectors input to MLP are input to all the MLP's fully-connected layers in a chain, and the output from the last layer is the output of the MLP.

2.1.2 Convolutional Neural Network

A Convolutional Neural Network (CNN) consists of one or more layers for performing convolutional operations. For example, a two-dimensional convolutional layer has a kernel $\boldsymbol{W}$ and

a bias b and computes the output $\boldsymbol{Y}$ from the input $\boldsymbol{X}$ by

$$\boldsymbol{Y}[i,j] = \sigma\left(\sum_{[i',j']\in S(i,j)} \boldsymbol{X}[i,j]\boldsymbol{W}[i,j] + b\right). \tag{2.2}$$

Note that $S(i,j)$ is a square area centered at the position $[i,j]$. As in MLP, the input is input to the convolutional layers in the CNN in a chain, and the output of the last layer is the output of the CNN.

2.1.3 Recurrent Neural Network

A Recurrent Neural Network (RNN) is a general term for networks that represent the dependence of input sequences. There are several types of RNNs, including simple RNNs and Long Short-Term Memory (LSTM) [5], but the one used in this paper is a Gated Recurrent Unit (GRU) [6]. The GRU has six fully-connected layers whose activation functions are identity functions and calculates the hidden state $\boldsymbol{h}_t$ at from the input $\boldsymbol{x}_t$ and the hidden state $\boldsymbol{h}_{t-1}$ using the following equation at time t:

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{n}_t + \boldsymbol{z}_t \odot \boldsymbol{h}_{t-1} \tag{2.3}$$

where

$$\boldsymbol{n}_t = \tanh\big(\mathrm{FC}_1(\boldsymbol{x}_t) + \boldsymbol{r}_t \odot \mathrm{FC}_2(\boldsymbol{h}_{t-1})\big), \tag{2.4}$$
$$\boldsymbol{z}_t = \varsigma\big(\mathrm{FC}_3(\boldsymbol{x}_t) + \mathrm{FC}_4(\boldsymbol{h}_{t-1})\big), \tag{2.5}$$
$$\boldsymbol{r}_t = \varsigma\big(\mathrm{FC}_5(\boldsymbol{x}_t) + \mathrm{FC}_6(\boldsymbol{h}_{t-1})\big). \tag{2.6}$$

Note that $\odot$ denotes an element-by-element product and FC means the fully-connected layers.

2.1.4 Embedding Layer

The embedding layer converts data x from a finite set X into a fixed-length vector. This layer has $|X|$ vectors and outputs a vector corresponding to the input x.

2.2 Optimization

Continuous optimization finds parameters $\boldsymbol{w}$ such that the objective function $f(\boldsymbol{w})\colon \mathbb{R}^n \to \mathbb{R}$ is minimized. In machine learning, it is often solved by updating the parameters of each layer using the error backpropagation method. The optimization methods in this paper all update the parameter $\boldsymbol{w}$ by

$$\boldsymbol{w}^{(i+1)} \leftarrow \boldsymbol{w}^{(i)} + \Delta\boldsymbol{w}^{(i)}. \tag{2.7}$$

The superscript (i) represents the parameter generation. $\Delta \boldsymbol{w}^{(i)}$ depends on the optimization method and is defined in the subsequent subsections.

2.2.1 RMSprop

$\Delta \boldsymbol{w}^{(i)}$ in RMSprop [7] is defined by the following equation.

$$\Delta \boldsymbol{w}^{(i)} = -\eta \boldsymbol{g}^{(i)} \oslash \sqrt{\boldsymbol{v}^{(i)} + \epsilon \mathbf{1}}, \tag{2.8}$$

$$\boldsymbol{g}^{(i)} = \frac{\partial f(\boldsymbol{w}^{(i)})}{\partial \boldsymbol{w}^{(i)}}, \tag{2.9}$$

$$\boldsymbol{v}^{(i)} = \rho \boldsymbol{v}^{(i-1)} + (1-\rho) \boldsymbol{g}^{(i)} \odot \boldsymbol{g}^{(i)} \tag{2.10}$$

where $\oslash$ is the per-element quotient and $\mathbf{1}$ is a vector whose all elements are 1. The square root of a vector is an operation that takes the square root of every element. In addition, η is the learning rate, and ρ, ϵ are hyperparameters. The smooth exponential average of the updates up to the present time is kept and multiplied by the inverse of the smooth exponential average, which suppresses updates for parameters that have been updated significantly.

2.2.2 ADADELTA

$\Delta \boldsymbol{w}^{(i)}$ in ADADELTA [8] is defined by the following equation.

$$\Delta \boldsymbol{w}^{(i)} = -\boldsymbol{g}^{(i)} \odot \sqrt{\boldsymbol{u}^{(i-1)} + \epsilon \mathbf{1}} \oslash \sqrt{\boldsymbol{v}^{(i)} + \epsilon \mathbf{1}}, \tag{2.11}$$

$$\boldsymbol{g}^{(i)} = \frac{\partial f(\boldsymbol{w}^{(i)})}{\partial \boldsymbol{w}^{(i)}}, \tag{2.12}$$

$$\boldsymbol{v}^{(i)} = \rho \boldsymbol{v}^{(i-1)} + (1-\rho) \boldsymbol{g}^{(i)} \odot \boldsymbol{g}^{(i)}, \tag{2.13}$$

$$\boldsymbol{u}^{(i)} = \rho \boldsymbol{u}^{(i-1)} + (1-\rho) \Delta \boldsymbol{w}^{(i)} \odot \Delta \boldsymbol{w}^{(i)} \tag{2.14}$$

where ρ and ϵ are hyperparameters. It has the same characteristics as RMSprop, but the significant improvement is that the learning rate is no longer needed because $\boldsymbol{w}^{(i)}$ and $\Delta \boldsymbol{w}^{(i)}$ have equal dimensions.

2.3 Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning that maximizes rewards from the environment. The interaction between the environment and the agent is often represented using a Markov Decision Process (MDP). The following characteristics of RL make it difficult.

- The feedback to the agent's actions is not correct or incorrect but a numerical reward.

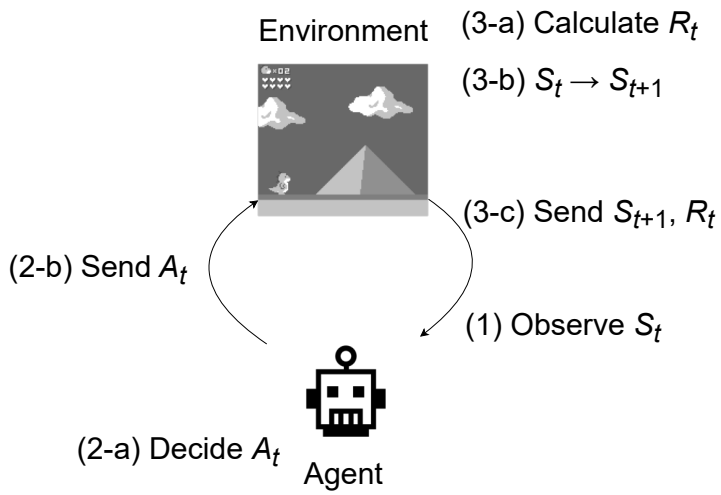


Figure 2.1 **Interaction between the environment and the RL agent.**

- In some cases, the agent can only observe one part of the environment.
- Rewards may be obtained some time after an appropriate action, or they may be obtained by multiple pairs of actions.

The interaction between the environment and the agent is as follows.

(1) The agent receives the state $S_t \in \mathcal{S}$ from the environment.
(2) The agent decides on the action $A_t \in \mathcal{A}$ to take in the state $S_t \in \mathcal{S}$ and sends it to the environment.
(3) The environment changes the state to S_{t+1} and sends the agent a reward $R_t \in \mathbb{R}$ and a new state S_{t+1}.

Note that $\mathcal{S}$ represents the state space, and $\mathcal{A}$ represents the action space. A schematic diagram of the interaction is shown in Figure 2.1.

2.3.1 Goals

An agent's policy $\pi\colon \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ represents the probability of taking each action in a given state. The goal of RL is to find the optimal policy π_* that maximizes expected G_0 under the policy π:

$$\pi_* = \arg\max_{\pi} \mathbb{E}_{\pi}[G_t]. \tag{2.15}$$

Note that $T \in \mathbb{N}_+ \cup \{\infty\}$ and $\gamma \in [0, 1)$ are the termination time and discount factor respectively. Also, G_t is the sum of discounted rewards after time t and is defined as

$$G_t := \sum_{\tau=t}^{T} \gamma^{\tau-t} R_\tau. \tag{2.16}$$

2.3.2 Bellman Equation

The state value function $v_\pi \colon \mathcal{S} \to \mathbb{R}$ is defined as the expected value of the sum of discounted rewards obtained by taking action from state s according to policy π:

$$v_\pi(s) := \mathbb{E}_\pi\big[G_t \mid S_t = s\big] \tag{2.17}$$

The action-value function $q_\pi \colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is defined as the expected value of the sum of the discounted rewards obtained when a person takes action a in state s and then acts according to policy π:

$$q_\pi(s, a) := \mathbb{E}_\pi\big[G_t \mid S_t = s, A_t = a\big] \tag{2.18}$$

These definitions show that v_π and q_π can be rewritten in the following recursive form. They are called the Bellman equation and are the source of parameter updates in RL algorithms. Note that the policy associated with the action-value function should be greedy, *i.e.*, it should always take the action that maximizes the action value.

$$v_\pi(s) = \mathbb{E}_\pi\big[R_t + \gamma v_\pi(S_{t+1}) \mid S_t = s\big], \tag{2.19}$$

$$q_\pi(s, a) = \mathbb{E}_\pi\big[R_t + \gamma \max_a q_\pi(S_{t+1}, a) \mid S_t = s, A_t = a\big]. \tag{2.20}$$

2.3.3 Reinforcement Learning Algorithms

(Deep) RL algorithms can be broadly classified into 2 types: methods that directly handle policies and those that do not. The methods that directly handle policies include REINFORCE [9], Actor-Critic, A3C [10], and IMPALA [11], and most of them improve policies based on the policy gradient theorem [12]. In addition to policies, most methods also deal with an estimation function V of state-values to stabilize the learning. For example, in the Actor-Critic method, the agent has a policy π determined by the parameter $\boldsymbol{\theta}$ and a state-value estimation function V determined by the parameter $\boldsymbol{\omega}$, and updates the parameters by repeating the following. Note that ∇ denotes the partial derivative in the subscript variable.

1. In state S_t, the agent performs action $A_t \sim \pi(\cdot \mid S_t)$ and receives new state S_{t+1} and reward R_t.

2. Calculates the Temporal-Distance (TD) error $\delta = R_t + \gamma V(S_{t+1}; \boldsymbol{\omega}) - V(S_t; \boldsymbol{\omega})$.
3. Updates $\boldsymbol{\theta}$ by $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_1 \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t; \boldsymbol{\theta})$.
4. Updates $\boldsymbol{\omega}$ by $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} + \eta_2 \delta \nabla_{\boldsymbol{\omega}} V(S_t; \boldsymbol{\omega})$.

Most of the methods that do not deal with policies directly have the action-value estimation function Q, that is updated based on the Bellman equation, and policies are represented through Q. Methods published to date include Q learning [13], DQN [1], Rainbow [14], R2D2 [15], R2D3 [16], NGU [17], and Agent57 [18]. For example, in Q-learning, the agent has an action-value estimation function Q determined by the parameter $\boldsymbol{\theta}$ and updates it by repeating the following.

1. In state S_t, the agent performs a random action A_t with probability ϵ and action A_t that maximizes the action-value with probability $1 - \epsilon$, and receives new state S_{t+1} and reward R_t.
2. Calculates the TD error $\delta = R_t + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\omega}) - Q(S_t, a; \boldsymbol{\omega})$.
3. Updates $\boldsymbol{\theta}$ by $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_1 \delta \nabla_{\boldsymbol{\theta}} Q(S_t, A_t; \boldsymbol{\theta})$.

The algorithm used in this thesis is IMPALA, which deals directly with policies and is described in detail in Section 2.3.3.

IMPALA

IMPortance-weighted Actor-Learner Architecture (IMPALA) [11] is a off-policy actor-critic method. That is, the "actor," an agent that operates in the environment and collects data, and the "learner," an agent that updates parameters, are separated, each with a (different) state-value estimation function and policies. Let μ denote the actor's policy, π the learner's policy, and V the state-value estimation function of the learner. Many actor-critic methods, including IMPALA, collect data in parallel by preparing multiple environments and actors. The purpose of this is to collect more diverse data more quickly. In particular, IMPALA is designed to use multiple learners and can scale to thousands of machines. A conceptual diagram of IMPALA is shown in Figure 2.2.

Although the learner synchronizes its parameters with the actor each time it updates them, strictly speaking, there is a discrepancy between the actor and learner parameters. IMPALA uses an algorithm called V-trace to compensate for this discrepancy. The gradient of the error function for the parameters that define the policy derived from this algorithm is as follows. Here,

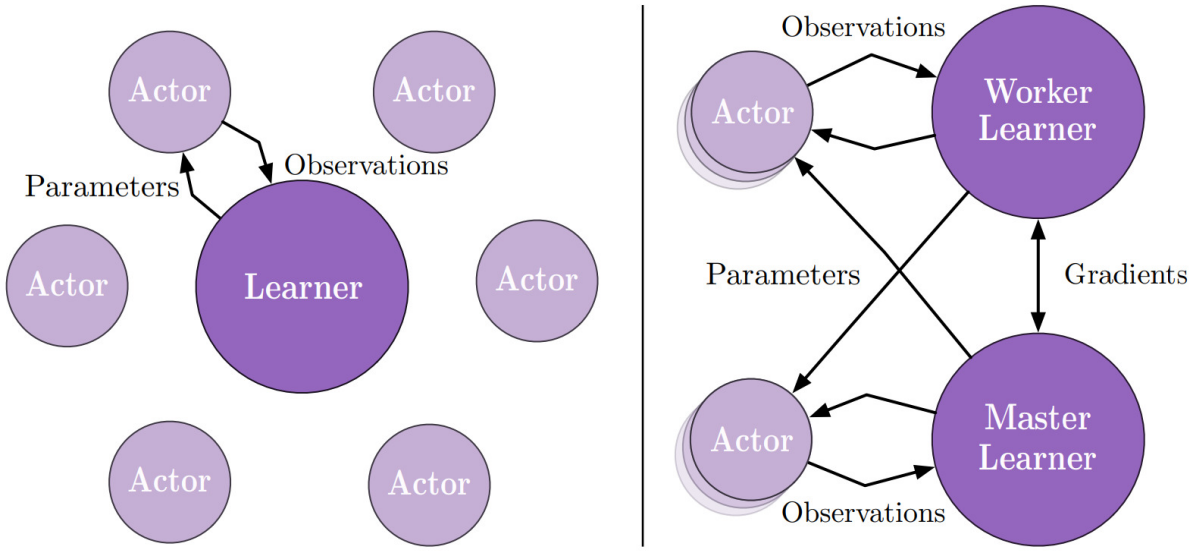


Figure 2.2 **Conceptual diagram of IMPALA**. The left figure shows a learning with single learner, and the right one shows a learning with multiple learners. This figure is taken from the paper [11].

$\beta, n, \overline{\rho}$, and $\overline{c}$ are hyperparameters.

$$\underbrace{\rho_t \nabla \log \pi(A_t \mid S_t)\{R_t + \gamma v_{t+1} - V(S_t)\}}_{\text{Policy gradient term}} + \beta \underbrace{\nabla \sum_{a \in \mathcal{A}} -\pi(a \mid S_t) \log \pi(a \mid S_t)}_{\text{Entropy term}} \tag{2.21}$$

where

$$v_t = V(S_t) + \sum_{s=t}^{t+n-1} \gamma^{s-t} \left(\prod_{i=t}^{s-1} c_i \right) \delta_s, \tag{2.22}$$

$$\delta_t = \rho_t \{R_t + \gamma V(S_{t+1}) - V(S_t)\}, \tag{2.23}$$

$$\rho_t = \min \left\{ \overline{\rho}, \frac{\pi(A_t \mid S_t)}{\mu(A_t \mid S_t)} \right\}, \tag{2.24}$$

$$c_t = \min \left\{ \overline{c}, \frac{\pi(A_t \mid S_t)}{\mu(A_t \mid S_t)} \right\}. \tag{2.25}$$

The gradient of the error function for the parameters that define the state-value estimation function is as follows.

$$\{v_t - V(S_t)\} \nabla V(S_t). \tag{2.26}$$

2.4 NetHack

NetHack is an early roguelike published in 1987. The game's objective (how to complete the game) is to explore a procedurally generated dungeon consisting of more than 50 levels using various items and to find and bring back "The Amulet of Yendor" that exists deep in the dungeon. In addition to the features common to roguelikes, NetHack has the following characteristics, making it an challenging game even for human players.

- The 50 or more level are not a straight path but has many branches, some of which have nearly 10 levels. The player must go back and forth between these branches to collect the items necessary to complete the game.
- There are more than 400 different enemies and items and more than 50 different actions.

A list of actions belonging to the action space used in this thesis is shown in Table 2.1.

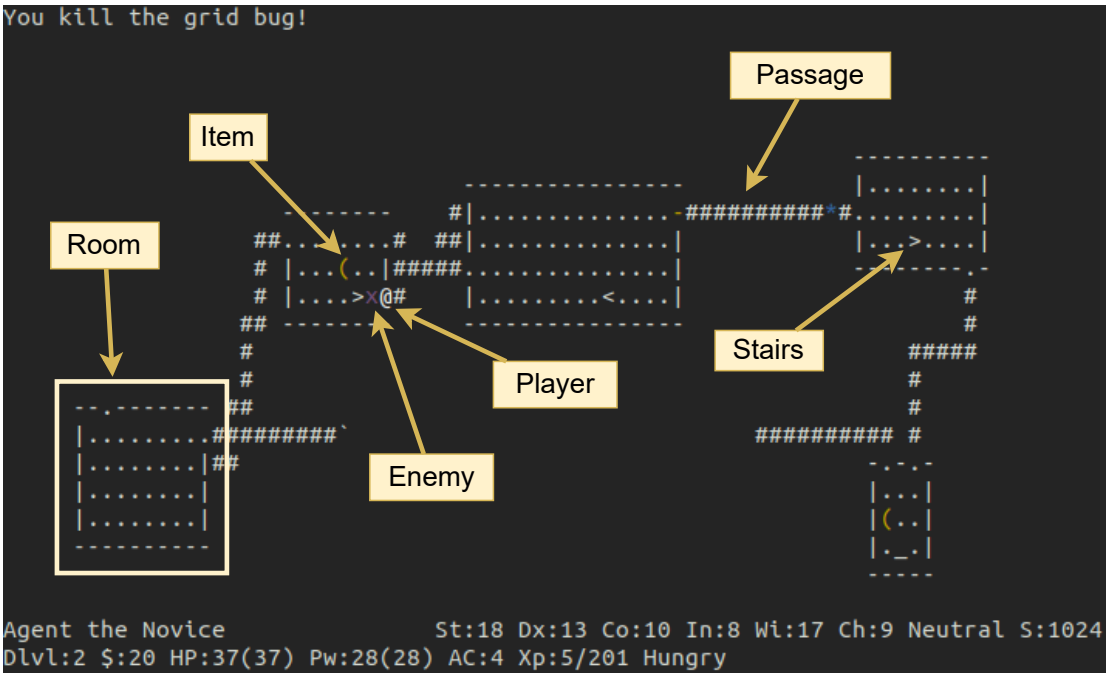
An example of a NetHack play screen is shown in Figure 2.3. The primary 4 states the player can observe are message, dungeon, status, and inventory.

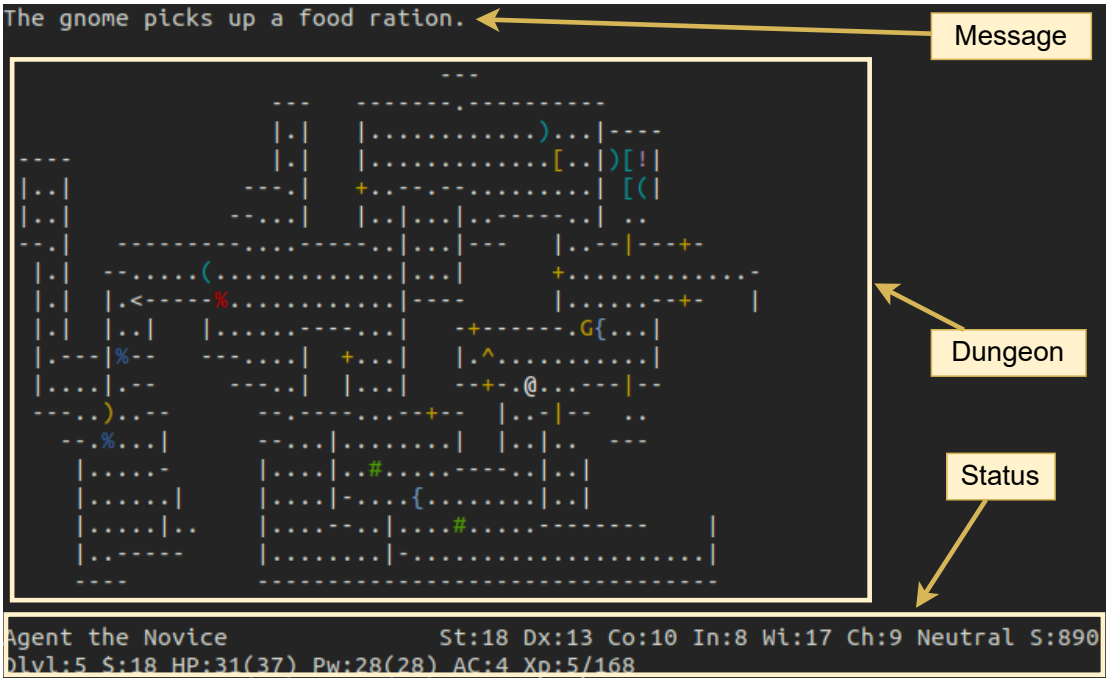
Message The events that have occurred, confirmation messages, and expected input are displayed. The first figure shows the state immediately after the player has killed the monster *grid bug*, so the message "You kill the grid bug!" is displayed. The middle figure shows the state immediately after the monster *gnome* picks up a *food ration* on the floor, so the message "The gnome picks up a food ration." is displayed.

Dungeon The level in which the player is located is displayed. Everything, including enemies and items, is represented by a single character. Examples of what each character represents are shown in Table 2.2. Most levels are composed of square rooms and corridors (top figure), but some are not (middle figure). A black area with nothing displayed indicates that the area has not been explored or that the area has been explored but nothing was found.

Status The player's current status, such as strength, HP and role, is displayed. Details of the status are shown in Table 2.3.

Inventory The letter on the left represents the symbol or alphabet used to specify the item.A list of items in the player's possession is displayed (bottom figure). The letter on the left represents the symbol or alphabet used to specify the item.






```
$ 7 gold pieces
a an uncursed +2 pair of leather gloves (being worn)
b an uncursed +1 robe (being worn)
c a blessed spellbook of healing
d an uncursed scroll of confuse monster
e 3 uncursed potions of healing
f 3 uncursed food rations
g 5 uncursed apples
h 5 uncursed oranges
i 3 uncursed fortune cookies
```

Figure 2.3 **Screenshot of NetHack.**

Table 2.1 **NetHack actions and their descriptions.** Actions used in this thesis are shown.

Action	Description
reading message	Reads a message. Other actions are ignored when it needs to be read.
move	Moves the player. If there is an enemy in the input direction, attacks it.
upstairs	Goes upstairs.
downstairs	Goes downstairs.
apply	Uses a tool item. Item specification is required.
drop	Drops an item. Item specification is required.
eat	Eats an item. Item specification is required.
kick	Kicks. Direction specification is required.
pickup	Picks up an item on the floor.
puton	Wears a ring or amulet. Item specification is required.
quaff	Drinks a potion. Item specification is required.
read	Reads a scroll of a spellbook. Item specification is required.
search	Searches for hidden passages, doors, and traps around the player.
takeoff	Takes off the player's armor. Item specification is required.
throw	Throws an item. Item and direction specification are required.
wear	Wears an armor. Item specification is required.
wield	Wields a weapon. Item specification is required.
zap	Zaps a wand. Item and direction specification are required.

2.5 Libraries

In this paper, we use PyTorch [19] as a framework for machine learning. We also use TorchBeast [20] as an implementation of RL algorithms (IMPALA). In addition, we use the NetHack Learning Environment (NLE) [21], which conforms to the OpenAI Gym [22] format that provides APIs for RL environments.

2.6 Related Work

2.6.1 Reinforcement Learning for Games

Modern RL algorithms are often evaluated in benchmark environments such as the Atari 2600 game provided by ALE [23] and various environments provided by the OpenAI Gym [22]. Ex-

Table 2.2 **Correspondence between a character and what it represents.**

Character	Description
`| -`	Wall. The player cannot pass through.
Colored `| -`	Opened door. The player can pass through and close it.
Colored `+`	Closed or locked door. Certain items are needed to open a locked door.
`.`	An empty floor. The player can walk.
`#`	Corridor. The player can walk.
`` ` ``	Boulder. The player cannot pass through, but can push it.
`@`	Player.
`> <`	Stairs. The player need to go down the stairs to complete the game.
`^`	Traps. It is not shown and revealed by triggering it or *search* action.
`_`	Altar. It has many useful features.
Alphabet (`x`)	Monster.
Other symbols (`! %`)	Items. The character is determined by its kind.

Table 2.3 **Details of the Status.**

`Agent`	Player name.
`Novice`	Rank title. Determined by a role and experience level.
`St`	Strength. It affectes the attack power.
`Dx`	Dexterity. It affectes the accuracy rate.
`Co`	Constitution. It affectes the weight that can be carried.
`In`	Intelligence. It affectes the attack power.
`Wi`	Wisdom. It affectes the success rate of spell learning and casting.
`Ch`	Charisma. It affectes the item price.
`Neutral`	Alignment. It affectes some strong items the player can get.
`S`	Score. Increased by defeating enemies or going to deeper levels.
`Dlvl`	Dunveon level. The deeper the player goes, the bigger it gets.
`$`	Gold coins the player carries.
`HP`	Hit points. If it reaches 0, the player dies. Number in () shows the maximum.
`Pw`	Power. Used for spell casting. Number in () shows the maximum.
`AC`	Armor class. It affectes the defense and avoidance. Lower is better.
`Xp`	Experience. Left shows the level and right shows the points.
`Hungry`	Warnings. Disadvantageous states the player is suffering are shown.

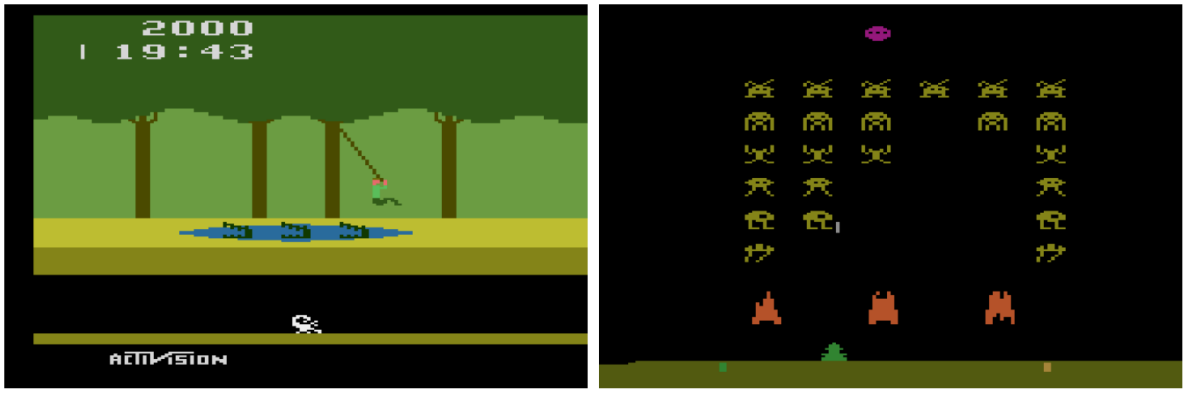


Figure 2.4 **Examples of Atari 2600 games provided in ALE.** The left is PITFALL! and the right is SPACE INVADERS. This figure is taken from the paper [23].

amples of Atari 2600 games are shown in Figure 2.4. Most of the games on the Atari 2600 are simple but include some challenging problems for RL, so it is a good benchmarking environment. For this reason, active RL research has been conducted, and the algorithm named Agent57 have been proposed to exceed human performance in all games included in ALE [18]. Agent57 is a distributed RL method in which the data collected by the actor is stored in a prioritized experience replay buffer [24], and the learner fetches data from the replay buffer to learn. A diagram of the learning process is shown in Figure 2.5. Prioritized experience replay [24] is an extension of experience replay [25]. It is intended to increase sample efficiency by storing data once in a buffer and fetching data with large loss in priority.

RL has also been applied to popular games such as Go [2], chess [26], and the multiplayer incomplete information games StarCraft II [3] and Dota 2 [27]. These studies use DRL in combination with Monte Carlo Tree Search (MCTS) [28]. There are also RL studies [29] in the Minecraft environment [30], [31], a game that uses items in the same way as NetHack.

2.6.2 Reinforcement Learning for Roguelikes

There is also much research on applying RL to roguelikes. Several highly customizable environments are available based on Rogue, the ancestor of roguelikes and a more straightforward game than NetHack [32], [33]. Many RL studies using these environments have targeted dungeon exploration (exploring the current level and moving to deeper levels) [32]–[36].

Several environments based on NetHack, which is the target of this thesis, are also provided [21], [37]. Studies using these environments include learning to combat enemies using abstracted state and action spaces [37] and using occupancy maps to efficiently discover hidden doors and passageways, which are essential for dungeon exploration [38]. Another study used

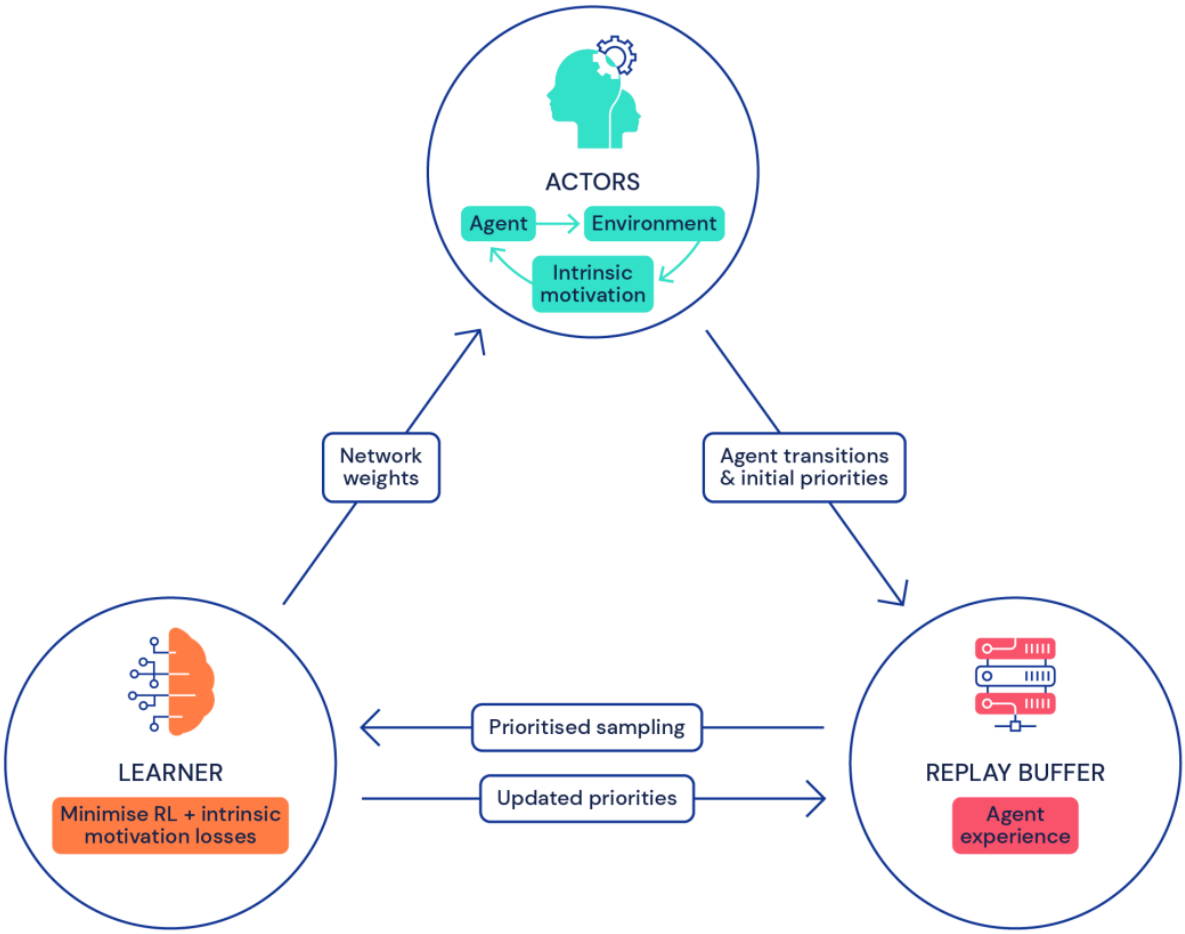


Figure 2.5 **Schematic diagram of RL process by Agent57.** This figure is taken from the paper [18].

Random Network Distillation [39], a explore facilitation method, to tackle tasks such as obtaining in-game scores and collecting money [21]. An example of the environment [37] not used in this thesis are shown in Figure 2.6. Because the study focused on combat with the enemy, the environment is simplified, with the player and the enemy placed in the corners of a single room instead of a real game.

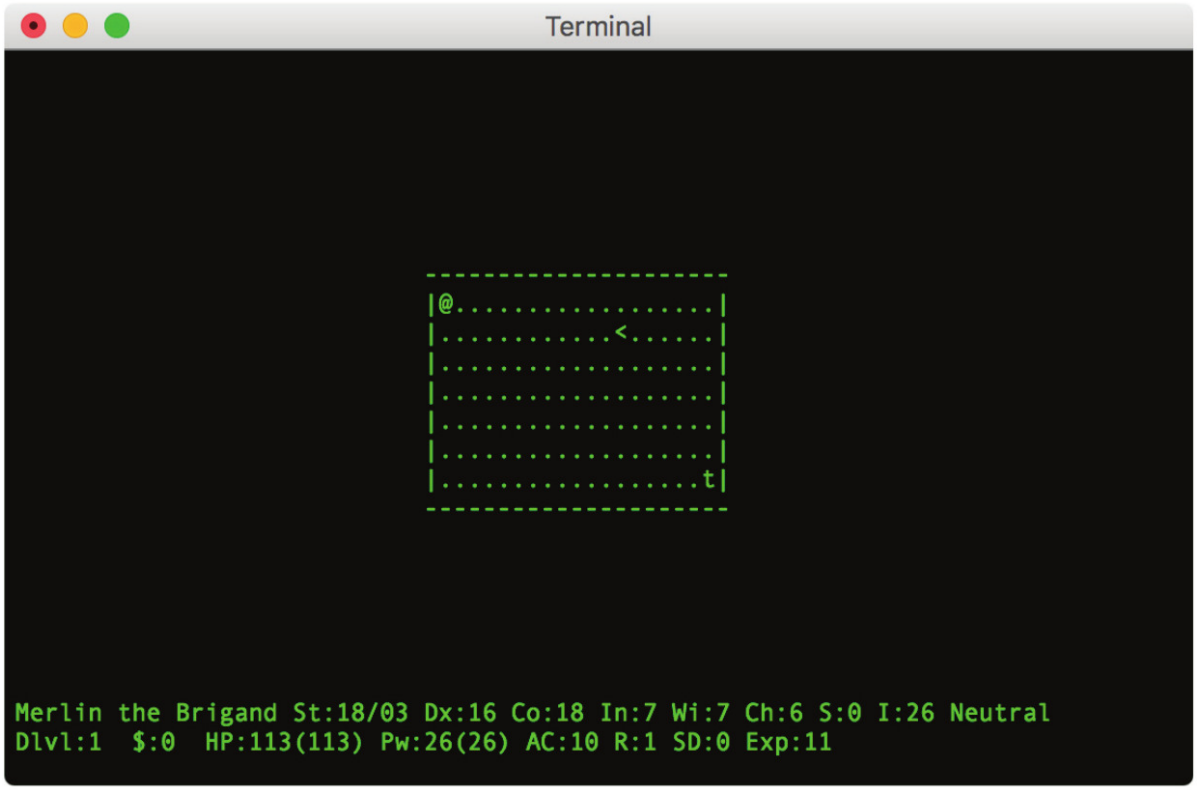


Figure 2.6 **Screenshot of the environment given by [37].** The player `@` and the enemy `t` are placed in the corner of a room, with no other enemies or items placed. This figure is taken from the paper [37].

Chapter 3

Inventory Management with Attention-Based Meta Actions

This chapter describes a method for handling the inventory, which are collections of items. Inventory has permutation-invariant features, and its size, *i.e.*, the number of items a player possesses, is variable, so applying an RL algorithm requires some ingenuity. Related publications are [40] and [41].

3.1 Background

As mentioned in Section 2.6.2, several studies utilize deep reinforcement learning for roguelike games. However, most of these studies focus on constructing the environment itself or learning in particular situations. In particular, items (inventory) exist in most role-playing games, including roguelikes, and their use is indispensable, but no research deals with them in general situations.

3.2 Method

3.2.1 Baseline

We use a modified version of the model used in the previous study [21] as a baseline. The overview of the model is shown in the upper part of the dashed line in Figure 3.1. The primary game information the model can handle is 3 types of message, dungeon, and status, excluding inventory, out of the 4 types described in Section 2.4.

Messages are 256 character strings, and features are extracted by embedding the ASCII code of each character into a fixed-length vector and then feeding it into a 1-dimensional CNN. Although

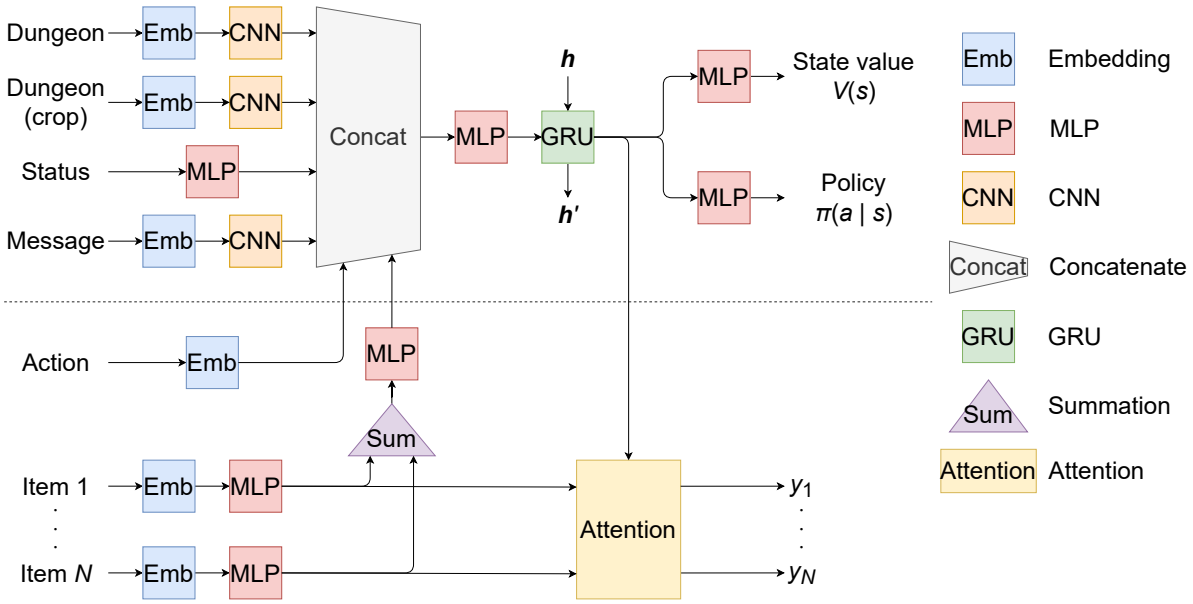


Figure 3.1 **Overview of the model.** The baseline model is the upper part of the dashed line, and the entire figure is the proposed model.

CNN is an outdated method in natural language processing, we use it because it is one of the simplest models, and message processing is not the focus of this experiment. The dungeon shape is 21×79, and each square has a fixed-length vector with embedded attributes such as color and type. Dungeon features are extracted using a 2-dimensional CNN. Furthermore, we extract features from the 9×9 region centered on the agent using a CNN. This has been shown to be effective in previous studies [42], [43]. The status is a vector consisting of numerical values such as the player's strength and attack power, and features are extracted using MLP.

These 3 features are combined and then input to MLP and GRU. The GRU outputs of the GRUs are regarded as the features of the current state, which are then input to the two MLPs to output a state-value estimation and a policy.

3.2.2 Action Recursion

We add an embedding layer to the model to incorporate information about the agent's previous actions. In this layer, the previous actions taken by the agent are converted into a fixed-length vector, which is concatenated with the 3 features described above as action features. In this case, only the previous action is input to this layer because the model incorporates a GRU to retain information on previous times.

This layer is added based on the observation that humans can determine their current actions based on the actions they have taken. Action recursion is particularly important in NetHack be-

cause specific actions are usually repeated multiple times, or a combination of multiple actions is used to perform a high-level action. An example of the former is the search for hidden passageways or doors. If there is a hidden passage or door in a square adjacent to an agent, the agent can find them with a certain probability by taking a *search* action. Therefore, it is common to repeat the *search* action a certain number of times when the agent arrives at a likely location. Examples of the latter include kicking and using items. The kicking action is triggered by inputting the kicking direction after the *kick* action is inputted. Using an item is triggered by specifying the item to be used after entering the type of action, such as *wear*, *read*. Because of these examples, it is important to input the previous actions taken by the agent into the model.

Theoretically, it should be possible to retain information about actions through the hidden state of GRUs. However, experimental results show that GRUs alone cannot retain action information and that direct input of actions to the model is practical. It is also possible to directly incorporate multiple high-level actions into the action space. However, this is not done because the action space becomes large, and learning becomes difficult. For example, NetHack has more than 10 types of actions to use items and more than 50 types of items that can be possessed. If these are handled directly, the action space size is only about 60, but if possible higher-level actions, *i.e.*, action/item pairs, are added to the action space, the action space size exceeds 500.

3.2.3 Meta Actions

In general, among all the actions an agent can take, there is a wide range of cases in which only specific actions are valid. For example, in NetHack, as described in Section 3.2.2, immediately after an action that uses an item (e.g.*read*), it is necessary to specify the item to be used. In such a state, the agent must select an action from a set of valid actions. Therefore, we introduce "meta actions" representing the entire set of valid actions. There are various possible meta actions, but we introduce "use item" as a meta action. When an agent selects the "use item" action, it selects an item according to the probability distribution $\exp(y_i)/\sum_i \exp(y_i)$. Note that y_i is the attention score of the i-th item, and the calculation method is described in Section 3.2.4.

Another advantage of introducing meta actions is that a variable number of actions can be handled. For example, the "use item" action is not easy to handle directly in RL framework because the number of items an agent possesses is not constant. By using meta actions, the size of the action space is fixed, making it easier to apply existing algorithms. It also has the advantage of preventing the action space from becoming too large for the same reasons described in Section 3.2.2 and separating standard and meta actions. As an example of treating actions separately, we propose modifying the design of the loss function. Details are given in Section 3.2.5.

3.2.4 Attention-Based Inventory Feature Extraction

Since an inventory is a set of items with no order, handling it properly in a neural network is not straightforward. We propose a feature extraction method for inventory and each item using an attention mechanism to express permutation invariance. First, each item is transformed into a vector $\boldsymbol{x}_i$ by the embedding layer and MLP. The embedding operation is performed in the same way as the embedding of each square in the field. Then, the sum of the vectors of all items, $\boldsymbol{x} = \sum_i \boldsymbol{x}_i$, is input to another MLP, and its output is the inventory feature. Since the summation is independent of the order of the items, this expresses permutation invariance. The inventory features are concatenated with the others described above and processed as in the baseline model.

Let $\boldsymbol{f}$ be the feature of the current state (GRU output), $\boldsymbol{W}_{\mathrm{Q}}, \boldsymbol{W}_{\mathrm{K}}$ and $\boldsymbol{w}_{\mathrm{V}}$ the 3 kinds of weights that the attention mechanism has, and $\boldsymbol{q}$ and $\boldsymbol{k}_i$ the dimension of $\boldsymbol{k}_i$. The score y_i of each item is then computed using the attention mechanism to specify the items to be used, using the following equation:

$$y_i = \frac{\boldsymbol{q}^\top \boldsymbol{k}_i}{\sqrt{d_{\mathrm{K}}}} v_i \tag{3.1}$$

where

$$\boldsymbol{q} = \boldsymbol{W}_{\mathrm{Q}} \boldsymbol{f}, \tag{3.2}$$
$$\boldsymbol{k}_i = \boldsymbol{W}_{\mathrm{K}} \boldsymbol{x}_i, \tag{3.3}$$
$$v_i = \boldsymbol{w}_{\mathrm{V}}^\top \boldsymbol{x}_i. \tag{3.4}$$

3.2.5 Loss Function

The introduction of item selection based on meta actions and attention mechanisms requires modifications to the loss function. First, we define the new action space and policies. Let b_0 be the "use item" action and b_i be the action to use the i-th item. We define the virtual action space as $\mathcal{A}_{\mathrm{v}} = \mathcal{A} \cup \{b_0\}$, the item action space as $\mathcal{A}_{\mathrm{i}} = \{b_1, b_2, \ldots\}$, and the raw action space as $\mathcal{A}' = \mathcal{A} \cup \mathcal{A}_{\mathrm{i}}$. Similarly, we define the virtual policy $\pi_{\mathrm{v}}\colon \mathcal{S} \times \mathcal{A}_{\mathrm{v}} \to [0, 1]$, the item action policy $\pi_{\mathrm{i}}\colon \mathcal{S} \times \mathcal{A}_{\mathrm{i}} \to [0, 1]$, and the raw policy $\pi'\colon \mathcal{S} \times \mathcal{A} \to [0, 1]$ are defined. These policies

satisfy the following for all $s \in \mathcal{S}$:

$$\pi'(a \mid s) = \begin{cases} \pi_{\mathrm{v}}(a \mid s) & \text{for } a \in \mathcal{A}, \\ \pi_{\mathrm{v}}(b_0 \mid s)\pi_{\mathrm{i}}(a \mid s) & \text{for } a \in \mathcal{A}_{\mathrm{i}}, \end{cases} \tag{3.5}$$

$$\sum_{a \in \mathcal{A}_{\mathrm{v}}} \pi_{\mathrm{v}}(a \mid s) = \sum_{a \in \mathcal{A}_{\mathrm{i}}} \pi_{\mathrm{i}}(a \mid s) = \sum_{a \in \mathcal{A}'} \pi'(a \mid s) = 1. \tag{3.6}$$

The RL algorithm used in this thesis is IMPALA [11]. To apply the proposed method to IMPALA, we replace all π with π' and the entropy term in Equation (2.21) with

$$\nabla \sum_{a \in \mathcal{A}_{\mathrm{v}}} -\pi_{\mathrm{v}}(a \mid S_t) \log \pi_{\mathrm{v}}(a \mid S_t) + \lambda \nabla \pi_{\mathrm{v}}(b_0 \mid S_t) \sum_{b \in \mathcal{A}_{\mathrm{i}}} -\pi_{\mathrm{i}}(b \mid S_t) \log \pi_{\mathrm{i}}(b \mid S_t). \tag{3.7}$$

Note that λ is a hyperparameter, and when $\lambda = 1$, all actions belonging to $\mathcal{A}'$ are treated equally in the entropy calculation, and Equation (3.7) is equal to

$$\nabla \sum_{a \in \mathcal{A}'} -\pi'(a \mid S_t) \log \pi'(a \mid S_t). \tag{3.8}$$

3.3 Experiments and Results

3.3.1 Experiment Settings

Reward

We use in-game scores as a reward. In-game scores can be earned through various events, such as raising the experience level of agents, defeating enemies, and acquiring gold coins. Since many of these events are essential, and the use of items, which is the focus of this study, contributes to the acquisition of scores, we set in-game scores as a reward.

Action Space

We use IMPALA [11] as a RL algorithm and compare the baseline with an extended version of a previous study conducted using a limited action space [21]. The previous action space was 14 in size, and its components were moving in 8 directions, upstairs and downstairs, reading messages, *eat*, *search*, and *kick*. In this study, we add 11 actions: *apply*, *drop*, *pickup*, *puton*, *quaff*, *read*, *takeoff*, *throw*, *wear*, *wield* and *zap*, so the action space size is 25. Details of each action are shown in Table 2.1.

Character

Most of the previous studies have been conducted with the character as `mon-hum-neu-mal` (representing role as Monk, race as human, attribute as neutral, and gender as male), but since

the game content of NetHack varies greatly depending on the character, especially the role, it is inappropriate to evaluate with only one type of character. Therefore, in this study, in addition to `mon-hum-neu-mal`, we conducted an experiment with 2 characters with very different characteristics. The additional characters are `val-dwa-law-fem` (Valkyrie, dwarf, lawful, female) and `tou-hum-neu-fem` (Tourist, human, neutral, female), and the characteristics of these roles are as follows.

- **Monk**: Powerful in the early stages due to its variety of items and high combat power, but becomes more challenging to conquer from the middle stage onward.
- **Valkyrie**: They have very few items at the beginning of the game but high combat power.
- **Tourist**: They have a variety of items, but their combat power is fragile and requires unique tactics.

Training and Evaluation

In all experiments, learning was performed until a total of 10 billion in actions were taken. An average of 10 episodes with different seeds was used in the evaluation tests.

3.3.2 Results

The proposed method and existing methods are compared in Figure 3.2 and Table 3.1. The proposed method outperforms the existing methods by a wide margin. Moreover, it significantly increases the score, especially for tourists whose fighting power is weak and whose use of items is essential. Comparison among the 4 types (baseline, no inventory mechanism, no action recursion, and the proposed model) shows that the components in the proposed model contribute to performance improvement. Furthermore, the in-game change difficulty due to the choice of role also significantly affects the score. The Monk, which is a powerful role due to its abundance of items and high combat power in the early stages of the game, has the highest score overall, followed by the Valkyrie, which is powerful in the early stages, and finally, the Tourist, which has low combat power.

3.4 Discussion

The agent trained by this method can use the appropriate items in various situations, regardless of the order of the items. To qualitatively confirm this, examples of agent output are shown in Figure 3.3 and Figure 3.4. The figure with the black background represents the game state, and

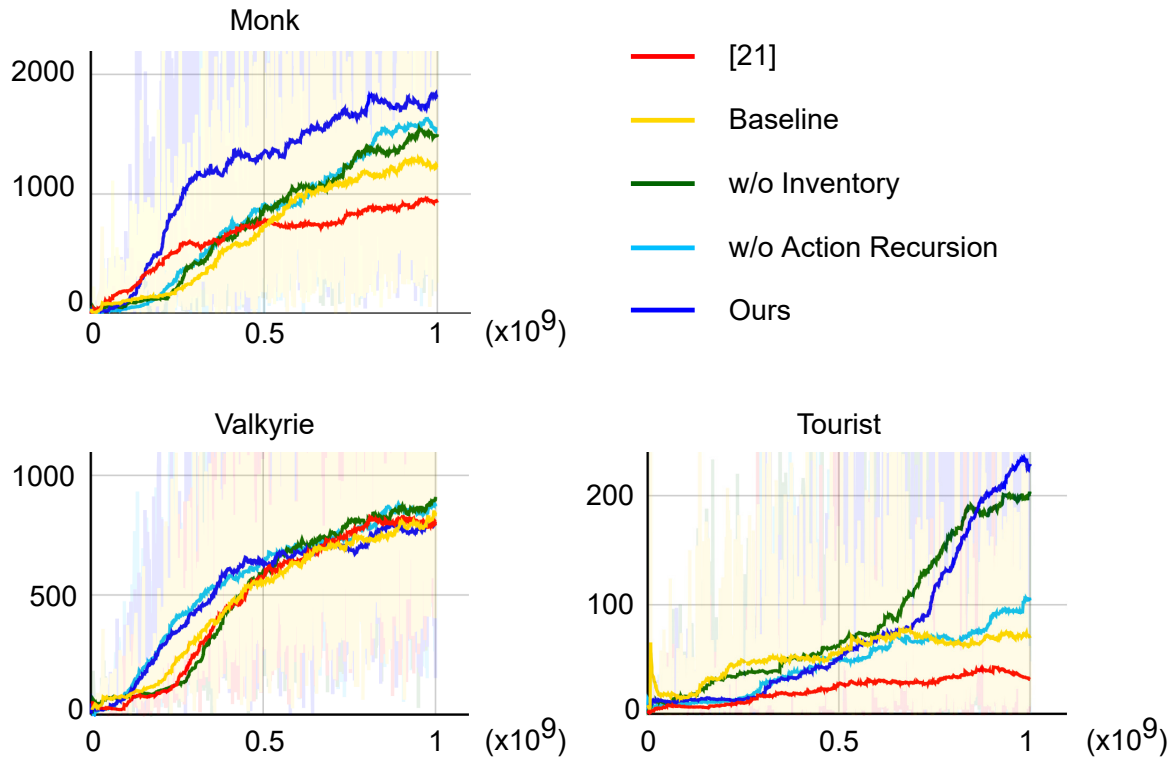


Figure 3.2 **Average return during training.** The horizontal axis represents the learning steps, and the vertical axis represents the average return.

Table 3.1 **Average return in 10 tests.** The best results are shown in bold.

	Monk	Valkyrie	Tourist
[21]	807.1	645.4	42.2
Baseline	1431.2	686.2	56.8
Ours w/o Inventory	1348.4	840.9	191.1
Ours w/o Action Recursion	1500.1	890.5	153.0
Ours	**2345.0**	**906.7**	**283.7**

the right side shows the actions taken by the agent in this state and their probabilities. Numbers in the figures show the actions with the highest probability of being taken in the action space $\mathcal{A}_{\text{v}}$. In the last figure in Figure 3.4, the probability transitions of the two most probable actions, *search* and *North* until the agent takes 5 consecutive *search* actions and finds the hidden door, are shown. The blue box represents the inventory, and the probability of using each item, *i.e.* π_{i}, is marked next to it.

In the first example in Figure 3.3, the message at the top of the screen asks the agent which

item to eat. In this state, the agent selects an edible item with a high probability, indicating that it can be used appropriately. In the second example in Figure 3.3, the agent is asked which item to drink immediately after taking the *quaff* action. The policy output shows that the latter is chosen. In the first example in Figure 3.4, the agent is asked in which direction to kick immediately after taking *kick* action. The agent specifies the appropriate kicking direction, indicating that it can "kick" that consists of two actions. The door below the agent is locked, and the agent who does not have the key can only open the door by kicking. Therefore, kicking is one of the appropriate actions in this state. The second example in Figure 3.4 is the state immediately after moving to the left due to the *West* action. Considering NetHack's field generation algorithm, there is still room on the left side of the unexplored field, but there seems to be no way to get to the left side of the field from the currently observable position. In such a case, it is appropriate to assume that there is a hidden door somewhere on the left edge of the explored area, and the agent is acting accordingly. In fact, the agent has found the hidden door by repeating the *search* action 5 times. Since the *search* can only search 8 squares around the agent, it is a natural strategy for the agent to move if it cannot find the hidden door after a certain amount of *search*. The probability transition between the *search* and *North* actions indicates that this strategy is taken.

Table 3.2 and Table 3.3 show the results of an additional experiment to determine the extent to which the items were used. The role used in this experiment is a Monk who always has food and three *potions of healing* at the beginning of the game. The experimental results indicate that inventory handling and action recursion mechanisms are necessary to use items properly. The most frequently taken actions are *eat* and *quaff*, which are directly related to an increase in the survival time of the agent. In NetHack, hunger progresses with actions, and high hunger decreases the agent's fighting ability, eventually leading to starvation. Therefore, agents need to eat food to survive longer and earn higher rewards. Drinking *potion of healing* helps agents to survive because it restores their HP. In addition, the proposed model also took the action of *read*. Monk starts the game with a random scroll, but if it is a *scroll of enchant armor*, he is trained to read it. This strengthens the agent's defensive and evasive capabilities and thus strengthens his fighting ability, which is a critical factor.

On the other hand, some actions that use an item are not learned. In NetHack, the agent does not know the identity of items not in the possession at the beginning of the game. These can only be revealed by limited actions, such as using the item or a specific type of item. However, some items can have disadvantageous effects when used or picked up, making their use dangerous. The tools to reveal themselves are outside the agent's possession at the beginning of the game. As a result, it is challenging for the agent to learn to use items outside its possession at the beginning of the game.

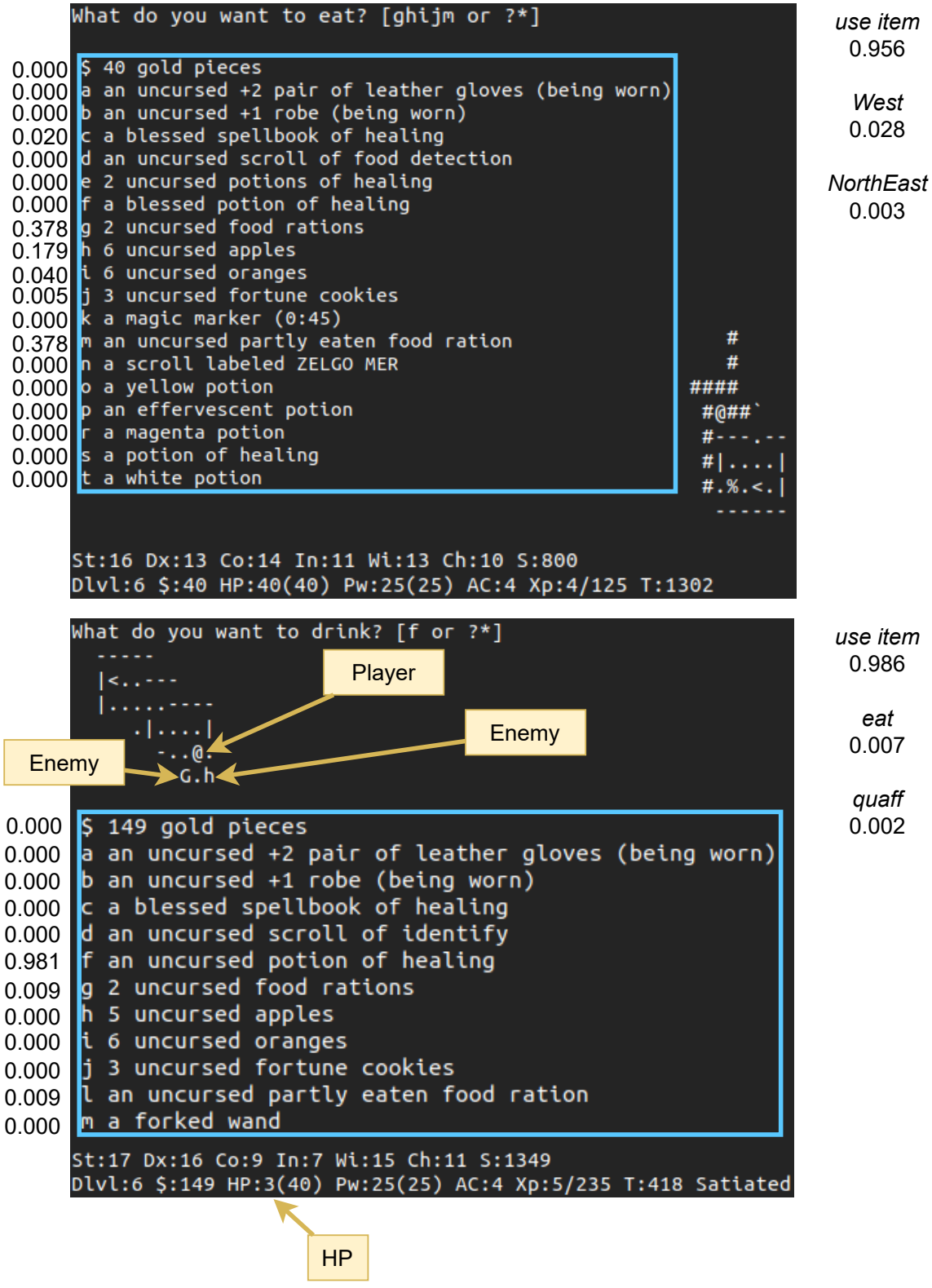
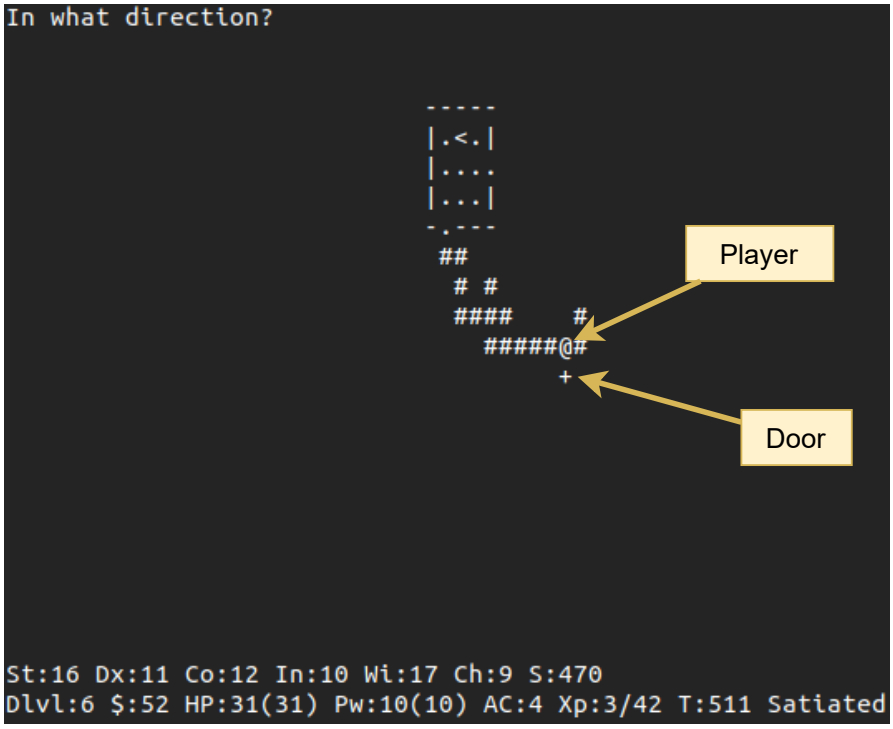


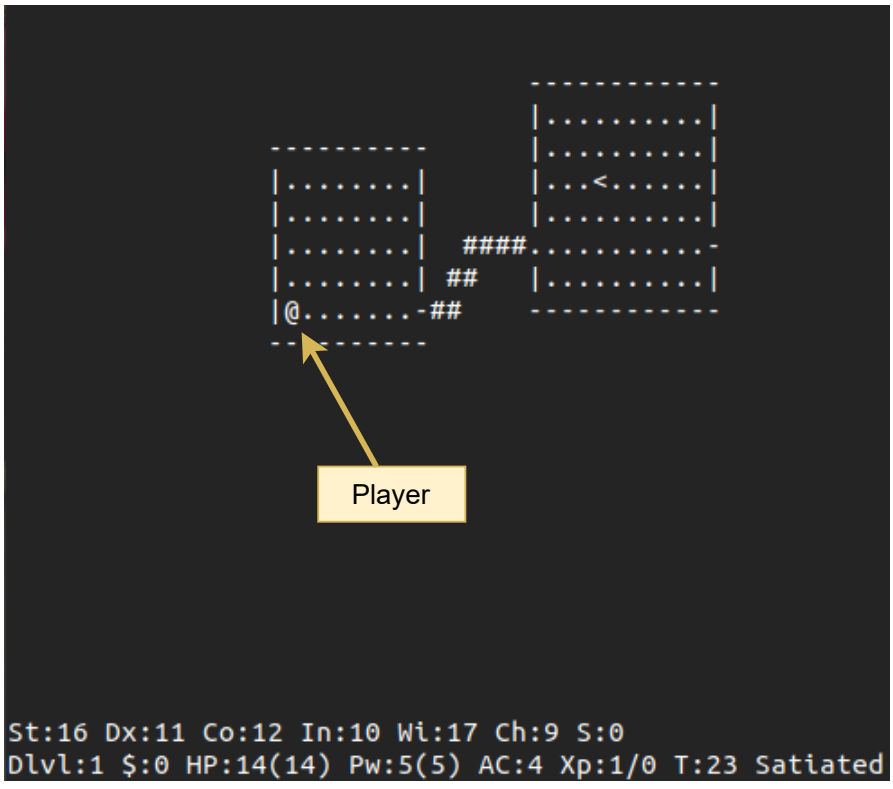
Figure 3.3 **A first example of the agent's policy.** The actions that have a high probability of being taken are shown. The blue box represents the inventory, with the corresponding π_i value shown next to it.



South 0.983

kick 0.011

East 0.004



search	*North*
0.957	0.030
0.717	0.214
0.717	0.219
0.679	0.250
0.616	0.304

Figure 3.4 **A second example of the agent's policy.** The actions that have a high probability of being taken are shown. In the last example, the transitions of π_v are shown.

Table 3.2 **The average number of times an agent used an item in an episode.** The role is Monk, and actions not used once are omitted.

	drop	*eat*	*quaff*	*read*
Baseline	0.0	4.0	1.0	0.2
Ours w/o Inventory	0.0	6.8	2.6	0.0
Ours w/o Action Recursion	0.0	4.0	2.7	0.0
Ours	0.6	6.2	2.7	1.7

Table 3.3 **Items specified in the "use item" meta action.** The role is Monk, and the maximum number of times the item has been used is three, in descending order of frequency of use. The numbers in parentheses indicate the average number of times the item was used in an episode.

Action	Item
drop	*unlabeled scroll* (0.5), Unknown *potion* (0.1)
eat	*food ration* (3.0), *apple* (1.5), *orange* (1.0)
quaff	*potion of healing* (2.7)
read	Unknown *spellbook* (1.3), Unknown *scroll* (0.2), *unlabeled scroll* (0.2)

Even models without action recursion or the inventory mechanism may successfully perform the action to use an item, although not as accurate as the proposed method. As mentioned above, the action to use an item usually consists of multiple actions, so it seems only possible for a model with both action recursion and the inventory mechanism to learn such actions. However, as shown in Figure 3.3, the message often indicate a second required action. Therefore, learning natural language messages is not easy, but it is possible to learn actions that require multiple inputs without using action recursion. It is also unnatural for a model without the inventory mechanism to succeed in using items, but this is due to a property of NetHack. In NetHack, the items possessed and the alphabet used to specify them at the beginning of the game are roughly fixed. Therefore, only items the agent has at the beginning of the game can be learned by "memorizing" the alphabet assigned to them.

Looking at the results by role in Table 3.1, it is shown that the effectiveness of the proposed method is significant for Monk and Tourist and small for Valkyrie. This is likely because Valkyrie has difficulty learning to use items, and most of the performance improvement is due to action recursion. As mentioned above, it is difficult to learn to use items that the agent does not have at the beginning of the game, so the proposed method is not very effective for Valkyrie, which has

only a few items compared to Monk and Tourist, which have many items at the beginning of the game.

Chapter 4

In-Game String Handling

This chapter describes a method for handling in-game strings. Strings that appear in the game have a difference vocabulary than natural language. They are also used as an adjunct for RL, so it is required a different approach than ordinary natural language processing methods.

4.1 Background

Character strings displayed in games are often selected from a predefined set of characters or character strings in the source code, so they generally have a vocabulary specific to the game. Furthermore, the need to accurately handle character strings is not that great in games, and image information processing is often more important. For these reasons, state-of-the-art string processing models such as BERT [44] are not optimal for games because they are much larger than necessary and retain unnecessary lexical knowledge even when trained models are used. Therefore, in this study, the model for processing strings is trained simultaneously as RL.

The neural network used to handle messages in Chapter 3 is a simple 1-dimensional CNN, an outdated method for handling strings. RNNs, which have hidden states and are more natural models for handling time-series data such as language, and Transformer [45], which uses self-attention without RNNs, are commonly used.

4.2 Method

4.2.1 Use of Non-CNN Models

Other than CNN, GRU and Transformer are used as lightweight models for training.

4.2.2 Online Bag-of-Words

Referring to Online Bag-of-Visual-Words [46], a method used in image recognition tasks, the following procedure is used for training. First, a set $V = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_K\}$ of random vectors $\boldsymbol{v}_k$ representing vocabularies is prepared. Note that K is the number of vocabularies. Next, we prepare two teacher and student models for self-supervised learning. They have a natural language processing model such as RNN and Transformer to compute features of strings, and the features computed by teacher are used in RL. The student also has another model G to compute weights $\boldsymbol{g}_k$ from the vocabulary vector $\boldsymbol{v}_k$. This is because the vocabulary is constantly updated in this method, and using dynamically generated weights is more stable than using the vocabulary as is. Another advantage is that G consists of a layer normalization [47] and an fully-connected layers to obtain permutation invariance.

The procedure for self-supervised learning, which is performed in parallel with RL, is as follows. First, the teacher computes the feature $\boldsymbol{u}$ from the input string $\boldsymbol{s}$. Then, the correspondence p_k between $\boldsymbol{u}$ and each vocabulary $\boldsymbol{v}_k$ is calculated as

$$p_k = \frac{\exp\left(-\|\boldsymbol{u} - \boldsymbol{v}_k\|^2/\delta\right)}{\sum_{k'=1}^{K} \exp\left(-\|\boldsymbol{u} - \boldsymbol{v}_{k'}\|^2/\delta\right)}. \tag{4.1}$$

Note that $\delta = \delta_0 m$, δ_0 is a hyperparameter, and m is the exponential smooth average of the minimum squared error $\min_k \|\boldsymbol{u} - \boldsymbol{v}_k\|^2$.

The student computes features $\tilde{\boldsymbol{u}}_i$ from several $\tilde{\boldsymbol{s}}_i$ cut from a random part of the input string $\boldsymbol{s}$. Then, the correspondence q_k between $\tilde{\boldsymbol{u}}$, which is the concatenation of all $\tilde{\boldsymbol{u}}_i$, and each dynamic weight $\boldsymbol{g}_k = G(\boldsymbol{v}_k)$ is calculated by

$$q_k = \frac{\exp\left(\kappa \tilde{\boldsymbol{u}}^\top \boldsymbol{g}_k\right)}{\sum_{k'=1}^{K} \exp\left(\kappa \tilde{\boldsymbol{u}}^\top \boldsymbol{g}_{k'}\right)} \tag{4.2}$$

with the hyperparameter κ.

We define the loss function as the cross entropy loss

$$\sum_{k=1}^{K} -p_k \log q_k \tag{4.3}$$

and minimize it. Also, replace the oldest vector in the vocabulary V with $\boldsymbol{u}$.

4.2.3 Variational Autoencoder

As mentioned in Section 4.1, fragments of strings appearing in the game can be obtained from the source code. Using the dummy strings generated from the source code as the dataset, a Vari-

ational Autoencoder (VAE) is pre-trained and used as the model for extracting string features. The specific training method is as follows. First, a list of strings displayed as messages during gameplay is obtained from the NetHack source code. At this point, the strings in NetHack, which is written in C language, may contain format specifiers. Since it is difficult to keep track of the actual strings included here, the format specifiers are replaced with random strings as an alternative. The resulting set of strings is used as a dataset and VAE is trained as follows:

1. A string $\boldsymbol{x}$ in the dataset is input to the VAE, and gets $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ that represents the latent space.
2. Calculates the reconstruction data $\hat{\boldsymbol{x}} = \boldsymbol{\mu} + \boldsymbol{e} \odot \sqrt{\boldsymbol{\sigma}^2}$ with the random vector $\boldsymbol{e} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$. Note that $\boldsymbol{I}$ means the identity matrix.
3. Updates the parameter in the direction of minimizing the loss function given by the following equation. Note that β is a hyperparameter and d is a dimension of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$. We can use any function as the reconstruction loss function loss, so we use the cross entropy error in this study.

$$\mathrm{loss}(\boldsymbol{x}, \hat{\boldsymbol{x}}) - \frac{\beta}{2} \sum_{i=1}^{d} \left(1 + \log \sigma_i^2 - \mu_i^2 - \sigma_i^2\right). \tag{4.4}$$

4.3 Experiments and Results

4.3.1 Experiment Settings

Experiments were conducted with the same settings as in Section 3.3.1. The model other than the string model was the same used in Chapter 3. We used only Monk for roles.

4.3.2 Results

The comparison of the performance between different string models is shown in Figure 4.1 and Table 4.1. Unfortunately, the online bag-of-words model could not be used due to machine specs, so the results are omitted.

4.4 Discussion

The performance of the 4 models tested in this study was almost the same. As mentioned in Section 4.1, string processing is not that important in games, and NetHack, the subject of this study, is no exception. The stage in NetHack when string processing is necessary (more advan-

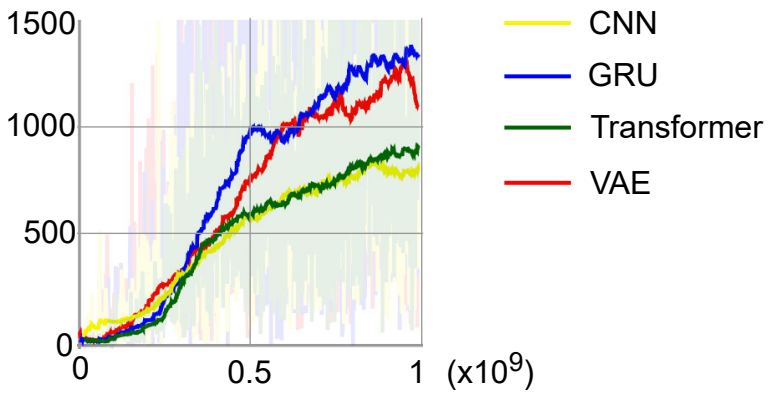


Figure 4.1 **Average return during training.** The horizontal axis represents the learning steps, and the vertical axis represents the average return. The role is Monk.

Table 4.1 **Average return in 10 tests.** The role is Monk.

Model	Return
CNN	1041.2
GRU	1215.6
Transformer	1084.1
VAE	1294.0

tageous to the game) is the middle stage of the game after the early stages have been overcome. The agents trained with our method had not yet reached that stage, and thus the string processing method did not produce a performance difference.

To demonstrate the correct progress of VAE training, we show in Table 4.2 the reconstruction results when a message that appeared during the expert's gameplay was input. From this example, we can see that VAE correctly learns key strings such as the enemy's name *goblin* and role *Barbarian*.

Table 4.2 **Reconstructed message by VAE.** The results are shown for 5 times for each input example. The ”Source” line shows the string in the source code corresponding to the input example. Bolded parts in the output are the parts where the reconstruction results are wrong.

Input	Hello izumiya, welcome to NetHack! You are a chaotic female orcish Barbarian.
Source	`"%s %s, welcome to NetHack! You are a%s %s %s."`
Output	Hello izumiya, welcome to NetHack! You are a chaotic female orcish Barbarian.
	Hello izumiya, welcome to NetHack! You are a chaotic female orcish Barbarian.
	Hello izumiya, welcome to NetHack! You are a chaotic female orcish Barbarian.
	Hello izumiya, welcome to NetHack! You are a chaotic female orcish Barbarian.
	Hello izumiya, welcome to NetHack! You are a chaotic female orcish Barbarian.
Input	The massive hammer hits! Lightning strikes the black unicorn!
Source	`"massive hammer hits%s %s%c"`
Output	The massive hammer hits! Lightning strikes the black unicorn**v**
	The massive hammer hits! Lightning strikes the black unicorn**e**
	The massive hammer hits! Lightning strikes the black unicorn**e**
	The massive hammer hits! Lightning strikes the black unicorn**e**
	The massive hammer hits! Lightning strikes the black unicorn**e**
Input	The goblin throws an orcish dagger!
Source	`"%s %s %s%s%s!"`
Output	The goblin throws an orcish dagger**e**
	The goblin throws an orcish dagger**]**
	The goblin throws an orcish dagger**e**
	The goblin throws an orcish dagge**mz**
	The goblin throws an orcish dagger**k**

Chapter 5

Reinforcement Learning with Expert Data

This chapter describes an efficient RL method using expert data. Although expert data supports RL, it is often not possible to prepare a large amount of data for tasks that require RL, so it is necessary to use the expert data efficiently.

5.1 Background

There are several advantages to using expert data from skilled players in RL. First, it facilitates learning, especially in the early stages of learning. An RL agent starts learning from a state in which it does not know, so it does not have even the "natural" knowledge that humans naturally acquire in life. Therefore, they have to learn content that is obvious to humans, which increases the learning time. The second point is reward-independent action learning. Current RL frameworks are theoretically designed to capture the relationship between actions and rewards, even if there is a sufficiently long delay between the action and the reward. However, this is difficult in reality. Actions that are not immediately rewarded but are important in the long run can be quickly learned by simply imitating the behavior of a skilled player. These advantages make RL with expert data important, especially for challenging tasks. There is a limit to the number of expert data that can be used in game development, which is one of the expected applications of RL, so it is necessary to use a small amount of data efficiently.

5.2 Method

5.2.1 Imitation Learning

Using the expert dataset $\mathcal{D}_{\mathrm{E}*}$, we create an imitation policy π_{E} that imitates the expert policy $\pi_{\mathrm{E}*}$ using supervised learning. However, because the expert policy is probabilistic, we do not use the general multiclass classification problem setup (corresponding to deterministic policy). Instead, we use the difference between the policies as the loss function. Specifically, we define the error function as

$$\mathbb{E}_{(s,a)\sim\mathcal{D}_{\mathrm{E}*}}\left[D_{\mathrm{KL}}\big(\pi_{\mathrm{E}*}(a \mid s), \pi_{\mathrm{E}}(a \mid s)\big)\right] \tag{5.1}$$

and minimize it. Note that D_{KL} represents KL divergence.

Since one of the primary purposes of utilizing expert data is to learn "rare" actions, such data should be preferentially used for training. Therefore, data $d_i \in \mathcal{D}_{\mathrm{E}*}$ is sampled with probability p_i, which is determined by solving the following convex optimization problem so that all behaviors are used equally for learning.

$$\text{Maximize} \quad \mathrm{Entropy}(p_1, p_2, \ldots, p_{|\mathcal{D}_{\mathrm{E}*}|}) \tag{5.2}$$

s.t.

$$\sum_{i=1}^{|\mathcal{D}_{\mathrm{E}*}|} p_i = 1, \tag{5.3}$$

$$\begin{aligned}\sum_{i=1}^{|\mathcal{D}_{\mathrm{E}*}|} p_i N_{ia_1} &= \sum_{i=1}^{|\mathcal{D}_{\mathrm{E}*}|} p_i N_{ia_2} \\ &= \sum_{i=1}^{|\mathcal{D}_{\mathrm{E}*}|} p_i N_{ia_3} \\ &= \cdots \\ &= \sum_{i=1}^{|\mathcal{D}_{\mathrm{E}*}|} p_i N_{ia_{|\mathcal{A}|}}.\end{aligned} \tag{5.4}$$

Note that $\mathcal{A} = \{a_1, a_2, ...\}$ and N_{ia} is the number of times action a is taken in the i-th data.

5.2.2 Reinforcement Learning with Imitation Policy

The imitation policy π_{E} got by the method shown in Section 5.2.1 is used for RL. Learning method is almost same as in Chapter 3, but we add the KL divergence between π_{E} and the agent's

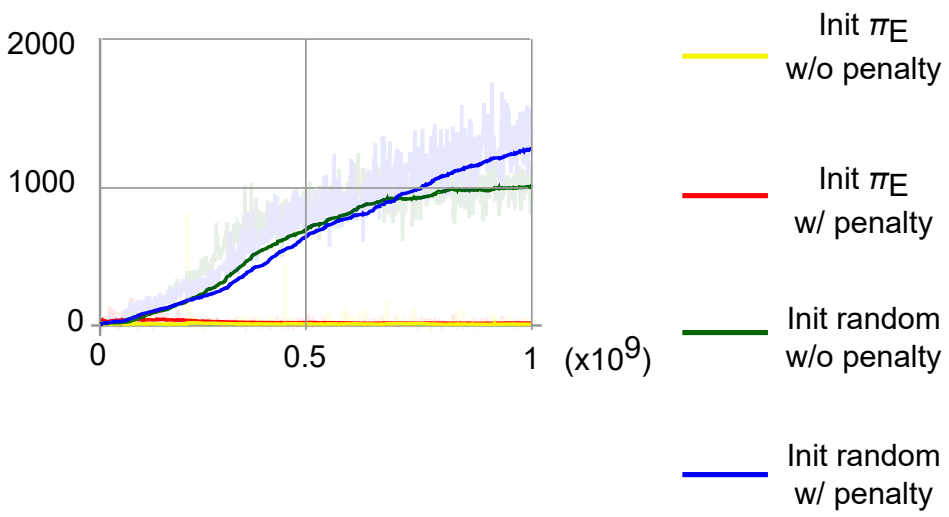


Figure 5.1 **Average return during training.** The horizontal axis represents the learning steps, and the vertical axis represents the average return. The role is Monk.

policy $\pi_{\boldsymbol{\theta}}$

$$D_{\mathrm{KL}}\big(\pi_{\mathrm{E}}(\cdot \mid S_t), \pi_{\boldsymbol{\theta}}(\cdot \mid S_t)\big) \tag{5.5}$$

to the reward penalty [48]. That is, we replace R_t in Equation (2.21) and Equation (2.23) with

$$R_t - \alpha D_{\mathrm{KL}}(\pi_{\mathrm{E}}(\cdot \mid S_t), \pi_{\boldsymbol{\theta}}(\cdot \mid S_t)) \tag{5.6}$$

where α is a hyperparameter.

5.3 Experiments and Results

5.3.1 Experiment Settings

Experiments were conducted with the same settings as in Section 3.3.1. We used only Monk for roles. Expert data was collected for approximately 44,000 steps, of which 20% was used as the validation dataset and the rest as the training dataset.

5.3.2 Results

Experimental results are shown in Figure 5.1 and Table 5.1 for the following variables: whether the agent's policy is initialized randomly or with π_{E}, and whether the reward penalty described in Section 5.2.2 is imposed during training.

Table 5.1 **Average return in 10 tests.** The role is Monk.

	Initialize randomly	Initialize with π_{E}
w/o reward penalty	1215.6	16.2
w/ reward penalty	1534.5	18.7

Table 5.2 **The average number of times an agent took invalid actions in an episode.** The agent's policy is initialized randomly. The role is Monk.

	Number of invalid actions
w/o reward penalty	113.1
w/ reward penalty	13.5

5.4 Discussion

First, we find that the learning only progresses when the agent's policy is initialized randomly, regardless of whether there is a reward penalty. The RL algorithm used in this method is IMPALA [11], which deals directly with policy, and the agent acts according to it. Although the expert's policy is probabilistic, there are many states in which he takes decisive action. Furthermore, due to the small size of the dataset $\mathcal{D}_{\mathrm{E}*}$, some states in which an expert would actually take a probabilistic action may be definitive in the dataset. Therefore, the learned policy π_{E} may be more decisive than the original one. If the agent's policy is initialized with π_{E}, the exploration will not be sufficiently performed, which is thought to be the reason why the learning does not proceed.

Next, we see that the reward penalty is effective when the policy are initialized randomly. As an example, we analyze invalid actions. Invalid actions are actions such as walking toward a wall or specifying an item that the agent does not have, and these actions are designed to be ignored. Therefore, while the agent does not receive any reward for taking an invalid action, it also avoids the agent's death. For example, in a state where an enemy attack will kill the agent, no matter what action he takes next, if he takes some action, the value of that action (the expected value of the future discounted reward sum) will be approximately 0. However, since taking an invalid action prevents the value of the action from becoming 0 upon the agent's death, the behavior was observed in which the agent postpones death by taking a series of invalid actions. Since experts do not engage in this behavior, we expect they will refrain from taking invalid actions

if a reward penalty is introduced to bring them closer to the expert policy. Table 5.2 shows the number of invalid actions the agent took with and without the reward penalty, with the agent's policy initialized randomly. As shown in the table, introducing a reward penalty significantly reduces the number of invalid actions.

Chapter 6

Conclusion

In this thesis, we study the modification of RL in NetHack, a kind of roguelike game. The specific results are as follows.

- We proposed a method for handling the inventory. It is not straightforward to incorporate the inventory handling into existing RL frameworks because of their permutation invariant and variable size features. We address these issues by using attention and meta actions, making it possible to incorporate the inventory into RL frameworks. Experimental results show that the proposed method improves performance and allows the appropriate use of items.
- We compared several models for handling in-game strings. In-game strings have a game-specific vocabulary and are used as auxiliary information in RL, so a different approach from general natural language processing is needed. We compared results using CNN, GRU, Transformer, and VAE and found that the performance was almost identical.
- We proposed a method to utilize expert data for RL efficiently. Expert data supports RL, but it is often difficult to prepare a large amount of data in practice, so it is necessary to use valid data efficiently. In order to ensure that rare actions can be learned equally well as other actions, we calculated the proportion of data to be used for learning by solving a convex optimization problem and then performed RL. Experimental results show that the proposed method improves performance and significantly reduces the number of invalid actions the expert does not perform.

Through this thesis, we examine how to utilize features common to games in general, such as strings and items, as well as the small amount of expert data that can be naturally provided in game development. As a result, we have laid the foundation for research on RL methods for very complex environments, such as NetHack, which are challenging to conquer even with the state-

of-the-art RL algorithms. The major remaining research issues are speeding up and stabilizing the learning process. In NetHack, the methods proposed in this thesis and improvements in RL algorithms have enabled agents to survive the early stages of the game, but there are still significant obstacles to overcome. To solve this problem, the use of expert data is essential. If we can extract the maximum amount of prior human knowledge from expert data and provide it to agents, learning will be faster, and they will be able to learn unusual actions and rewarding actions with extremely long delays. Thus, more useful utilization of expert data is a primary goal of future research.

Bibliography

[1] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, 2013.

[2] D. Silver, A. Huang, C. J. Maddison, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] O. Vinyals, I. Babuschkin, W. M. Czarnecki, *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[4] International Roguelike Development Conference, *Berlin interpretation*, Accessed: January 22nd, 2023, 2008. [Online]. Available: `http://www.roguebasin.com/index.php?title=Berlin_Interpretation`.

[5] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, 1997.

[6] K. Cho, B. van Merriënboer, C. Gulcehre, *et al.*, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing)*, Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734.

[7] T. Tieleman and G. Hinton, *Neural networks for machine learning*, Accessed: January 22nd, 2023, 2012. [Online]. Available: `http://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf`.

[8] M. D. Zeiler, "Adadelta: An adaptive learning rate method," *arXiv preprint*, 2012.

[9] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3–4, pp. 229–256, May 1992.

[10] V. Mnih, A. P. Badia, M. Mirza, *et al.*, "Asynchronous methods for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, Jun. 2016, pp. 1928–1937.

[11] L. Espeholt, H. Soyer, R. Munos, *et al.*, "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures," *35th International Conference on Machine Learning, ICML 2018*, vol. 4, pp. 2263–2284, 2018.

[12] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12, MIT Press, 1999.

[13] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, 1989.

[14] M. Hessel, J. Modayil, H. van Hasselt, *et al.*, "Rainbow: Combining improvements in deep reinforcement learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, Apr. 2018.

[15] S. Kapturowski, G. Ostrovski, W. Dabney, J. Quan, and R. Munos, "Recurrent experience replay in distributed reinforcement learning," in *International Conference on Learning Representations*, 2019.

[16] C. Gulcehre, T. L. Paine, B. Shahriari, *et al.*, "Making efficient use of demonstrations to solve hard exploration problems," in *International Conference on Learning Representations*, 2020.

[17] A. P. Badia, P. Sprechmann, A. Vitvitskyi, *et al.*, "Never give up: Learning directed exploration strategies," in *International Conference on Learning Representations*, 2020.

[18] A. P. Badia, B. Piot, S. Kapturowski, *et al.*, "Agent57: Outperforming the atari human benchmark," *arXiv preprint*, 2020.

[19] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.

[20] H. Küttler, N. Nardelli, T. Lavril, *et al.*, "TorchBeast: A PyTorch Platform for Distributed RL," *arXiv preprint*, 2019.

[21] H. Küttler, N. Nardelli, A. H. Miller, *et al.*, "The NetHack Learning Environment," in *Proceedings of the Conference on Neural Information Processing Systems*, 2020.

[22] G. Brockman, V. Cheung, L. Pettersson, *et al.*, "Openai gym," *arXiv preprint*, 2016.

[23] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, 2013.

[24] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

[25] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Mach. Learn.*, vol. 8, no. 3–4, pp. 293–321, May 1992.

[26] J. Schrittwieser, I. Antonoglou, T. Hubert, *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[27] C. Berner, G. Brockman, B. Chan, *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint*, 2019.

[28] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*, Springer, 2006, pp. 72–83.

[29] S. Frazier and M. Riedl, "Improving deep reinforcement learning in minecraft with action advice," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 15, 2019, pp. 146–152.

[30] W. H. Guss, C. Codel, K. Hofmann, *et al.*, "The MineRL competition on sample efficient reinforcement learning using human priors," *NeurIPS Competition Track*, 2019.

[31] W. H. Guss, M. Y. Castro, S. Devlin, *et al.*, "Neurips 2020 competition: The MineRL competition on sample efficient reinforcement learning using human priors," *NeurIPS Competition Track*, 2020.

[32] A. Asperti, C. De Pieri, and G. Pedrini, "Rogueinabox: An environment for roguelike learning," *International Journal of Computers*, vol. 2, 2017.

[33] Y. Kanagawa and T. Kaneko, "Rogue-gym: A new challenge for generalization in reinforcement learning," in *2019 IEEE Conference on Games*, 2019, pp. 1–8.

[34] A. Asperti, C. De Pieri, M. Maldini, G. Pedrini, and F. Sovrano, "A modular deep-learning environment for rogue," *WSEAS Trans. Syst. Control*, vol. 12, pp. 362–373, 2017.

[35] A. Asperti, D. Cortesi, and F. Sovrano, "Crawling in rogue's dungeons with (partitioned) a3c," in *Machine Learning, Optimization, and Data Science*, G. Nicosia, P. Pardalos, G. Giuffrida, R. Umeton, and V. Sciacca, Eds., Cham: Springer International Publishing, 2019, pp. 264–275, ISBN: 978-3-030-13709-0.

[36] A. Asperti, D. Cortesi, C. De Pieri, G. Pedrini, and F. Sovrano, "Crawling in rogue's dungeons with deep reinforcement techniques," *IEEE Transactions on Games*, vol. 12, no. 2, pp. 177–186, 2020.

[37] J. Campbell and C. Verbrugge, "Learning combat in NetHack," in *Thirteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Oct. 2017, pp. 16–22.

[38] J. Campbell and C. Verbrugge, "Exploration in NetHack with secret discovery," *IEEE Transactions on Games*, 2018.

[39] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, "Exploration by random network distillation," in *International Conference on Learning Representations*, 2019.

[40] K. Izumiya and E. Simo-Serra, "Inventory management with attention-based meta actions," in *IEEE Conference on Games*, 2021.

[41] K. Izumiya and E. Simo-Serra, "Item management using attention mechanism and meta actions in roguelike games," in *Visual Computing*, 2021.

[42] F. Hill, A. Lampinen, R. Schneider, *et al.*, "Environmental drivers of systematicity and generalization in a situated agent," in *International Conference on Learning Representations*, 2020.

[43] C. Ye, A. Khalifa, P. Bontrager, and J. Togelius, "Rotation, translation, and cropping for zero-shot generalization," in *2020 IEEE Conference on Games*, 2020, pp. 57–64.

[44] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186.

[45] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017.

[46] S. Gidaris, A. Bursuc, G. Puy, N. Komodakis, M. Cord, and P. Pérez, "Learning representations by predicting bags of visual words," in *Computer Vision and Pattern Recognition Conference*, 2021.

[47] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint*, 2016.

[48] Z. Huang, J. Wu, and C. Lv, "Efficient deep reinforcement learning with imitative expert priors for autonomous driving," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–13, 2022.