Appendix A. AI Crowd Evaluation

The evaluation service on AI Crowd worked as follows: each participant was required to submit their inference code and trained models, using AIcrowd's Gitlab repositories corresponding to their username. The participant's submitted code was packaged into a Docker image for every submission, and run on AWS "g4dn.xlarge" instances with 4 CPUs, 8 GB of RAM, and 1 NVidia T4 GPU with 16 GB video memory. Simultaneously, AIcrowd also started separate Docker images, which ran the NetHack environment with the fixed settings decided by the organizers, as well as code for tracking the scores of the rollouts. These ran on AWS "t3a.medium" instances with 2 CPUs and 4 GB of RAM. These two Docker images communicated observations and actions to each other respectively and provides the security guarantee that the environment or the scores cannot be hacked in any way by the participants.

Appendix B. Trajectory Analysis

The following plots show the investigation made in the trajectories of the Top 8 finalists in the NetHack Challenge. In particular, it consists of the three sets of evaluations submitted in Phase 2, consisting of a maximum of 12,288 trajectories. In some cases, this may have been reduced if the agent was unable to complete all 4,096 episodes during an evaluation.

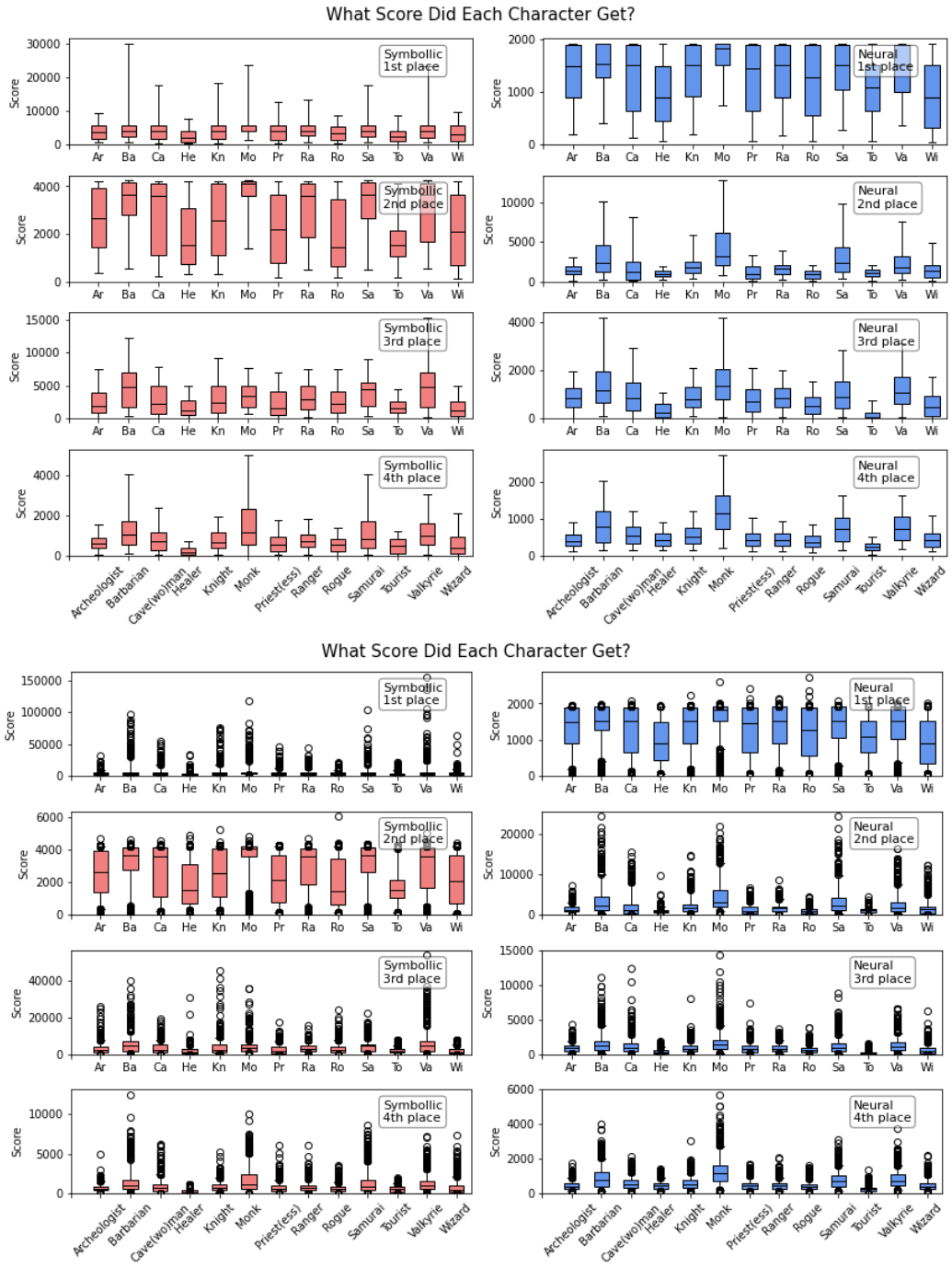

Figure 5: Box plots of the score broken down by starting role. The black line indicates the median, the box-plot is the interquartile range, and whiskers are the 5-95th percentiles. Outliers are shown in the second plot. Note how Symbolic 2nd and Neural 1st show evidence of early episode termination.

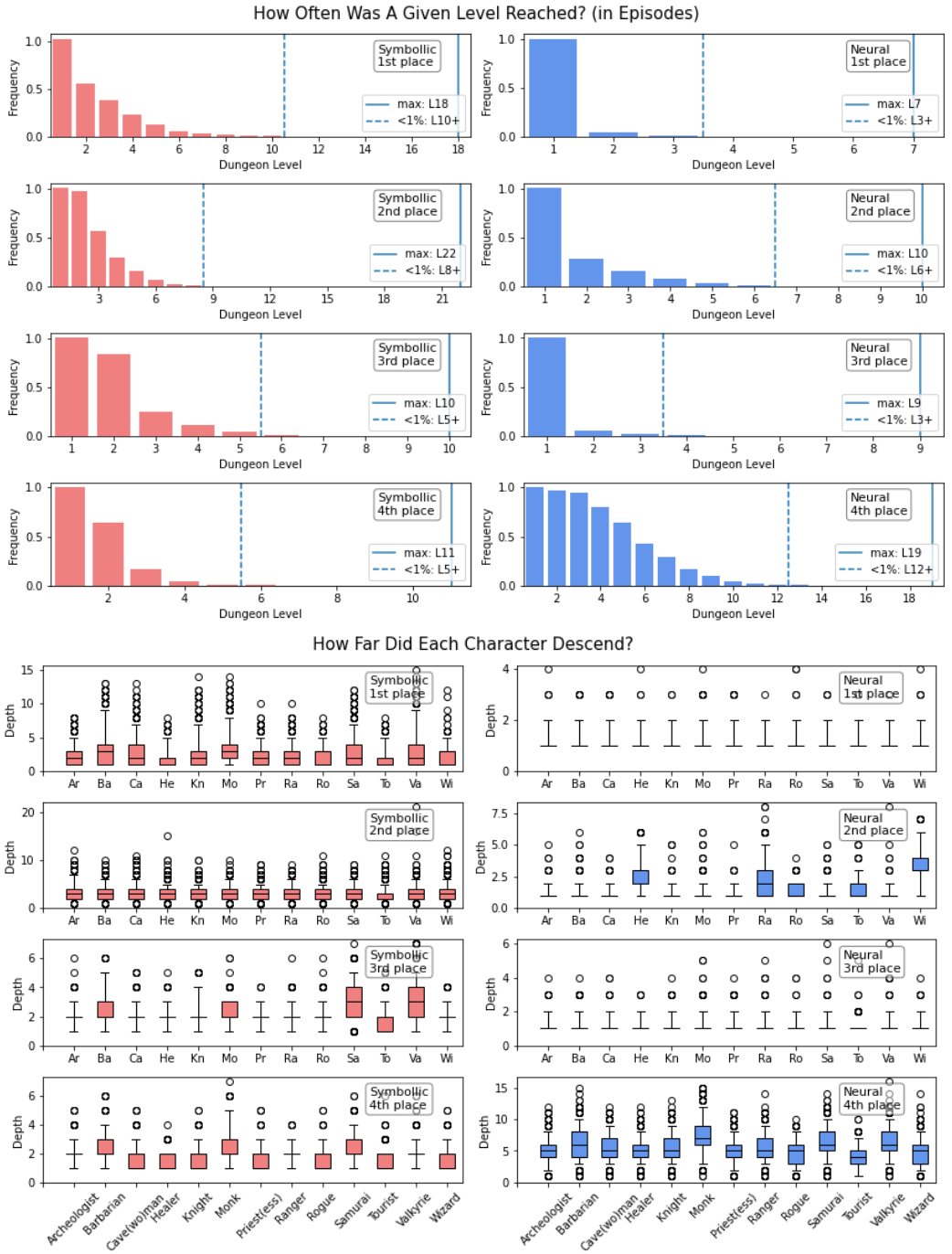

Figure 6: [Top] A plot of dungeon level exploration frequencies per episode. In what fraction of episodes was that dungeon reached? [Bottom] Top plot broken down by role, plotted as a box plot. The black line indicates the median, the box-plot is the interquartile range, and whiskers are the 5-95th percentiles. Outliers are shown in the second plot. Note how several entries (Neural 1, Neural 3) indicate restriction to the first level. Characters can still teleport or fall through holes to new levels by accident.

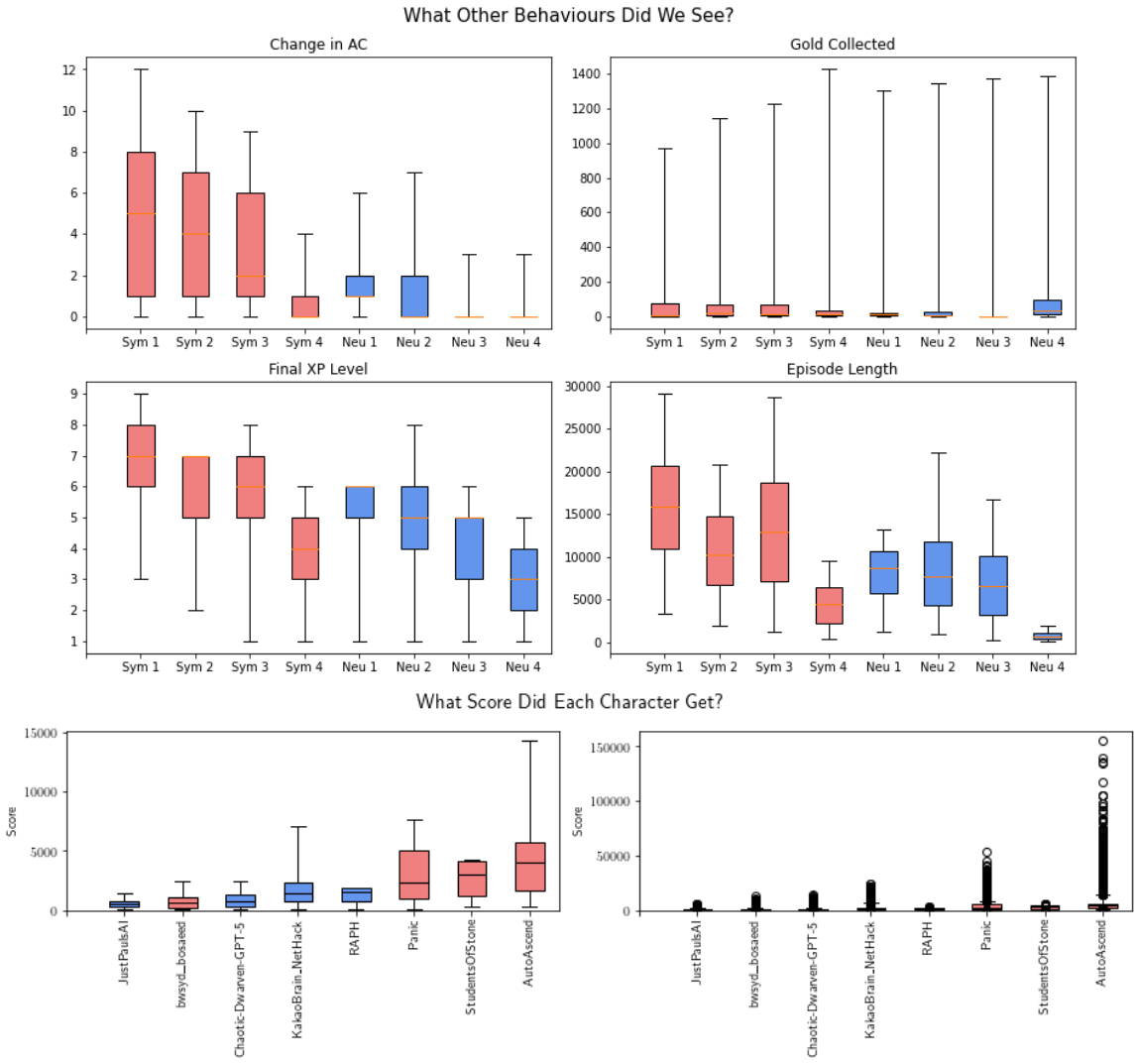

Figure 7: [Top] Boxplots of (clockwise from top left): max change in AC over an episode (NB this can happen by accident by damaging armour or polymorphing); gold accumulated; episode length; experience level at the time of death. [Bottom] The box plots for agent score, aggregated. Left without outliers, right with outliers. The black line indicates the median, the box-plot is the interquartile range, and whiskers are the 5-95th percentiles. Outliers are shown in the second plot. Note how Symbolic 2nd and Neural 1st show evidence of early episode termination.

Symbolic No. 1	Top Killed By	%
1	quit	34
2	died of starvation	5
3	killed by a giant bat	3
4	killed by a jackal	3

Neural No. 1	Top Killed By	%
1	quit	46
2	killed by a jackal	4
3	died of starvation	4
4	killed by a sewer rat	3

Symbolic No. 2	Top Killed By	%
1	quit	44
2	died of starvation	5
3	killed by a rothe	2
4	killed by a werejackal	2

Neural No. 2	Top Killed By	%
1	quit	11
2	killed by a jackal	5
3	killed by a giant bat	4
4	died of starvation	3

Symbolic No. 3	Top Killed By	%
1	quit	16
2	died of starvation	7
3	killed by a rothe	5
4	killed by a jackal	4

Neural No. 3	Top Killed By	%
1	killed by a jackal	7
2	died of starvation	6
3	poisoned by rotten food	6
4	killed by a sewer rat	5

Symbolic No. 4	Top Killed By	%
1	killed by a jackal	6
2	killed by a wand	5
3	killed by a guard	5
4	poisoned by rotten food	5

Neural No. 4	Top Killed By	%
1	killed by a dwarf	10
2	killed by a gnome lord	8
3	killed by a gnome	5
4	killed by a giant bat	3

Symbolic No. 1	Top Cause	%
1	while fainted from lack of food	32
2	while fainted	4
3	while frozen by a monster	2
4	while praying	1

Neural No. 1	Top Cause	%
1	while fainted from lack of food	20
2	while frozen by a monster	11
3	while fainted	3
4	while praying	2

Symbolic No. 2	Top Cause	%
1	while fainted from lack of food	18
2	while fainted	3
3	while praying	3
4	while frozen by a monster	2

Neural No. 2	Top Cause	%
1	while fainted from lack of food	26
2	while praying	19
3	while frozen by a monster	10
4	while sleeping	2

Symbolic No. 3	Top Cause	%
1	while fainted from lack of food	29
2	while fainted	6
3	while frozen by a monster	2
4	while helpless	1

Neural No. 3	Top Cause	%
1	while fainted from lack of food	28
2	while frozen by a monster	17
3	while fainted	4
4	while helpless	3

Symbolic No. 4	Top Cause	%
1	while fainted from lack of food	14
2	while sleeping	4
3	while frozen by a monster	3
4	while fainted	3

Neural No. 4	Top Cause	%
1	while fainted from lack of food	7
2	while praying	4
3	while sleeping	2
4	while frozen by a monster	1

Figure 8: Agents' deaths are described as *"Killed by [Death] while [Cause]"*. In this case [Top] A chart of most common *[Death]*. [Bottom] A chart of most common *[Cause]*. Note that in many cases, death is aggravated by fainting from a lack of food, or occuring whilst praying.

Stat	Symbol No. 1	Symbol No. 2	Symbol No. 3	Symbol No. 4
Angered God	204	421	197	615
Food Poisoning	60	24	215	630
FoodPois	61	54	526	633
Starved	567	572	808	467
Choked	0	0	0	2
Killed Pet	530	183	388	0
Killed Pet Hallu	60	19	7	0

Stat	Neural No. 1	Neural No. 2	Neural No. 3	Neural No. 4
Angered God	144	266	506	234
Food Poisoning	14	239	672	48
FoodPois	38	353	706	48
Starved	416	289	721	42
Choked	0	1	127	0
Killed Pet	1441	6128	0	155
Killed Pet Hallu	4	3	0	3

Stat	% Eps - Symbolic No. 1	% Eps - Symbolic No. 2	% Eps - Symbolic No. 3	% Eps - Symbolic No. 4
Burdened	28.50	13.16	31.64	17.64
Weak	95.09	97.16	88.11	60.35
Fainting	94.68	29.12	75.94	58.93
Fainted	93.25	25.98	71.32	31.93
Conf	81.12	69.42	66.18	43.67
Hallu	4.84	0.80	0.43	1.91
Stun	70.82	40.06	13.69	30.69
Blind	70.04	67.59	61.24	34.07
Deaf	94.99	39.57	76.65	54.07

Stat	% Eps - Neural No. 1	% Eps - Neural No. 2	% Eps - Neural No. 3	% Eps - Neural No. 4
Burdened	40.89	17.50	0.03	0.30
Weak	94.26	94.45	87.20	18.17
Fainting	30.64	43.10	47.94	8.36
Fainted	28.81	34.27	40.35	8.64
Conf	54.79	38.81	20.15	5.66
Hallu	0.05	1.49	0.32	0.24
Stun	32.16	52.69	41.03	15.66
Blind	39.76	28.98	19.75	13.00
Deaf	35.27	42.03	46.10	13.26

Figure 9: [Top] Number of episodes where certain events took place: death from angered god; death from food poisoning; contracting food poisoned status; starving to death; choking to death; killing a pet; killing a pet while hallucinating. [Bottom] Frequency of episodes where status was encountered.

Appendix C. AutoAscend

We implemented numerous behaviours and features targeting different aspects of the game. We list the most important ones to show the comprehensiveness of our solution.

Exploration related

- Untrapping traps and chests, looting containers
- Keeping track of all dungeon level states including glyphs, items (also stacked), search count per every tile, altars with their alignment, corpse ages, shop positions and types, stairs with the information where they lead
- Detecting vault entrances to avoid stepping into them, but if the agent happens to fall into the vault by an accident, drops gold and follows the guard to the exit
- Handling map specific behaviors, such as changed diagonal movement in Sokoban, no doors kicking in Minetown, or dungeon level finding helpers, e.g. Sokoban is always one level below the Oracle
- Basic wish handling
- Graceful handling of blindness, confusion, stun, hallucination, polymorph, etc.
- Curing lycanthropy by using a sprig of wolfsbane, holy water, or praying
- Sokoban solving including error-proof behaviours like checking for monsters before pushing the boulder to a place that cuts the player component in the movement graph, checking for boulder mimics, and destroying boulders

Item management related

- Smart item identification, e.g. using buy shop prices, possible glyph-object association (e.g. gem color), engrave-identification for wands, bijective properties of glyph-object association (if an item is identified, it cannot be under an unidentified glyph), combining results from these methods for drawing better conclusions
- Naming items (`#call`) for easy instance identification, e.g. too old corpses for sacrification, bag identifiers to keep track of items in bags
- Using bags for carrying items
- Dipping long swords in fountains to get the Excalibur
- Identifying BUC status of items using altars
- Sacrificing corpses on altars to get an artifact

Combat strategy related

- Damage calculation is implemented using NetHack code and the data from the wiki
- Enhancing skill proficiencies
- Avoiding melee attacking some enemies like "floating eye" or "gas spore"
- Not using ranged weapons when a pet or a peaceful monster is in the way
- Simulating ray wands reflection trajectory (with probabilities for all possible paths)
- Healing by using healing potions, using basic healing spells, and praying

We also developed a visualisation tool to help us debug the policy of our agent, displayed in Figure 10

Figure 10: An example visualization of game state and current agent behaviour. The state is visualized before every action (in raw `NLE` action space). On the top, there are four log panes (one line per step): raw `NLE` actions, messages, pop-ups (e.g. opened inventory), simplified strategy stacks (from left to right). In the centre, the current level is visualized using a graphical tileset. On the bottom, there is a raw TTY data visualization and miscellaneous stats. The left bar is used for inventory and item knowledge visualization.

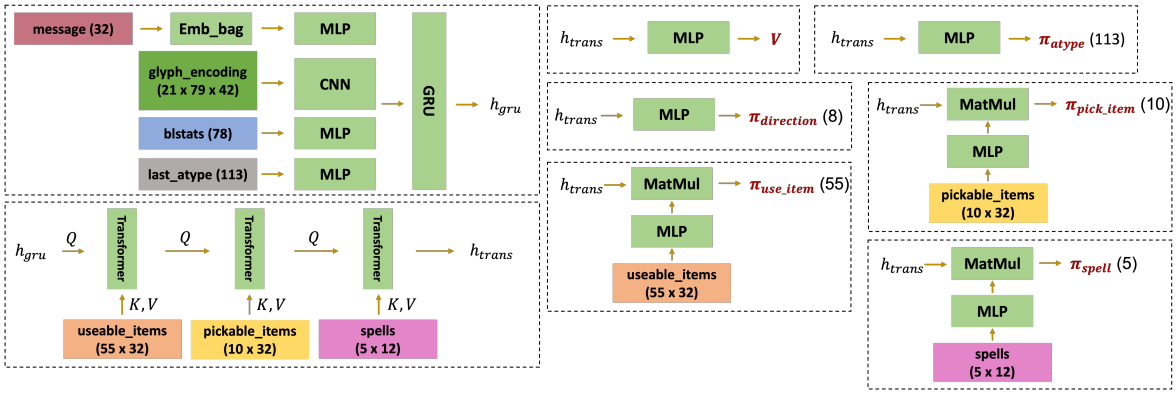

Figure 11: Model Structure of Team KakaoBrain.

Appendix D. RAPH

Algorithm 1: RAPH agent

Data: view_distance, agent, hard_coded_skills
state, *done* ← *env.reset*(), *False*;
while *not done* **do**
  action_queue = parse_message(state);
  if *action_queue* **then**
    state, reward, done, info = env.step(action_queue); `/* We have a prompt to response */`
    continue
  end
  monster_distance, preprocessed_state = parse_dungeon(state);
  if *monster_distance* <*view_distance* **then**
    action_queue = agent.act(preprocessed_state);
  else
    action_queue = first_fit(hard_coded_skills, preprocessed_state); `/* Select non-rl action on first-fit basis */`
  end
  state, reward, done, info = env.step(action_queue);
end

Appendix E. KakaoBrain

Here we explain details of our approach. An in-depth model structure is illustrated in Figure 11.

- **Observation Encoding**: For *in-game message tokenization*, we tokenize in-game messages by words rather than characters since it will be easier to understand a sentence in a limited vocabulary setting. We encode the first 32 words at most (tokens) for a sentence. For *extended blstats*, we use the blstats with extra information (race,

gender, alignment, and condition mask) which helps select optimal behaviors. For *glyph encoding*, we encode a glyph value (an integer between 0 and 5,976) of each pixel in the screen with its glyph group, object class, id, and is-agent. We expect that directly providing game-specific information rather than converting to an RGB image is more helpful for the agent to learn a general behavior. For *usable items*, we encode each item in the inventory with its glyph group, object class, cursed, worn, enchant, and count. For *pickable items*, we encoded pickable items similarly to usable items by parsing their information from the screen (it is not directly given in the original observation). For *spells*, we encode each spell information with its id, level, failure, and retention by parsing their information from the screen (it is not directly given in the original observation). We encode the information of at most 5 spells for simplicity.

- **Separated Action Spaces**: As we explained in the main text, our separated action spaces are composed of *action-type*, *direction*, *use-item*, *pick-item*, and *use-spell*. *action-type* is the same as the original action space, *direction* is composed of 8 direction choices for choosing a direction, *use-item* consists of item choices for choosing an item to be used, *pick-item* and *use-spell* are comprised of item choices for picking up items and spell choices for casting a spell, respectively. Generally, the agent chooses an action in action-type, and the other actions are used when required. This encourages better item farming, item utilization, and spell utilization by separating and clarifying action spaces for them.

- **Network Structure**: *Tokenized message* is fed to EmbeddingBag which embeds token id into a vector of 80 size and averages these vectors of multiple (32) tokens. We use 3-layer CNNs with 128, 64, and 32 channels, stride size of 3, and 2×2 average pooling for the first and second layers. We use a 2-layer MLP with the out sizes of 256 for `blstats`. We use a 1-layer GRU with the hidden size of 1024. We use a 1-layer MLP with the out size of 64 for *last atype*. We use TrXL-I structure without memory (Parisotto et al., 2020) to encode information of usable items, pickable items, and spells. We use 1-layer TrXL-I with the hidden size of 1024, the head size of 256, and 4 heads for each of them.

- **Role-specific Training**: To improve the score by encouraging a role-specific strategy, we train models that are dedicated to a specific role. In specific, we apply the role-specific reward shaping for Healer, Ranger, Rogue, Tourist, and Wizard. We set the HP difference as a reward to encourage healing itself for Healer. We give an incentive in killing monsters with firing for Ranger, killing monsters by throwing for Rogue, killing monsters by throwing for Tourist, and killing monsters with spells for Wizard.