
The NetHack Learning Environment

Heinrich Küttler⁺ Nantas Nardelli⁼ Alexander H. Miller⁺
Roberta Raileanu^{*} Marco Selvatici[#] Edward Grefenstette^{+!} Tim Rocktäschel^{+!}

⁺Facebook AI Research ⁼University of Oxford ^{*}New York University

[#]Imperial College London [!]University College London

{hnr, rockt}@fb.com

Abstract

Progress in Reinforcement Learning (RL) algorithms goes hand-in-hand with the development of challenging environments that test the limits of current methods. While existing RL environments are either sufficiently complex or based on fast simulation, they are rarely both. Here, we present the NetHack Learning Environment (NLE), a scalable, procedurally generated, stochastic, rich, and challenging environment for RL research based on the popular single-player terminal-based roguelike game, NetHack. We argue that NetHack is sufficiently complex to drive long-term research on problems such as exploration, planning, skill acquisition, and language-conditioned RL, while dramatically reducing the computational resources required to gather a large amount of experience. We compare NLE and its task suite to existing alternatives, and discuss why it is an ideal medium for testing the robustness and systematic generalization of RL agents. We demonstrate empirical success for early stages of the game using a distributed Deep RL baseline and Random Network Distillation exploration, alongside qualitative analysis of various agents trained in the environment. NLE is open source and available at <https://github.com/facebookresearch/nle>.

1 Introduction

Recent advances in (Deep) Reinforcement Learning (RL) have been driven by the development of novel simulation environments, such as the Arcade Learning Environment (ALE) [9], StarCraft [64, 69], BabyAI [16], Obstacle Tower [38], Minecraft [37, 29, 35], and Procgen Benchmark [18]. These environments introduced new challenges for state-of-the-art methods and demonstrated failure modes of existing RL approaches. For example, *Montezuma’s Revenge* highlighted that methods performing well on other ALE tasks were not able to successfully learn in this sparse-reward environment. This sparked a long line of research on novel methods for exploration [e.g., 8, 66, 53] and learning from demonstrations [e.g., 31, 62, 6]. However, this progress has limits: the current best approach on this environment, Go-Explore [22, 23], overfits to specific properties of ALE and Montezuma’s Revenge. While Go-Explore is an impressive solution for Montezuma’s Revenge, it exploits the determinism of environment transitions, allowing it to memorize sequences of actions that lead to previously visited states from which the agent can continue to explore.

We are interested in surpassing the limits of deterministic or repetitive settings and seek a simulation environment that is complex and modular enough to test various open research challenges such as exploration, planning, skill acquisition, memory, and transfer. However, since state-of-the-art RL approaches still require millions or even billions of samples, simulation environments need to be fast to allow RL agents to perform many interactions per second. Among attempts to surpass the limits of deterministic or repetitive settings, *procedurally generated environments* are a promising path

towards testing systematic generalization of RL methods [e.g., 39, 38, 60, 18]. Here, the game state is generated programmatically in every episode, making it extremely unlikely for an agent to visit the exact state more than once during its lifetime. Existing procedurally generated RL environments are either costly to run [e.g., 69, 37, 38] or are, as we argue, of limited complexity [e.g., 17, 19, 7].

To address these issues, we present the *NetHack Learning Environment* (NLE), a procedurally generated environment that strikes a balance between complexity and speed. It is a fully-featured *Gym* environment [11] around the popular open-source terminal-based single-player turn-based “dungeon-crawler” game, NetHack [43]. Aside from procedurally generated content, NetHack is an attractive research platform as it contains hundreds of enemy and object types, it has complex and stochastic environment dynamics, and there is a clearly defined goal (descend the dungeon, retrieve an amulet, and ascend). Furthermore, NetHack is difficult to master for human players, who often rely on external knowledge to learn about strategies and NetHack’s complex dynamics and secrets.¹ Thus, in addition to a guide book [58, 59] released with NetHack itself, many extensive community-created documents exist, outlining various strategies for the game [e.g., 50, 25].

In summary, we make the following core contributions: (i) we present NLE, a fast but complex and feature-rich Gym environment for RL research built around the popular terminal-based game, NetHack, (ii) we release an initial suite of tasks in the environment and demonstrate that novel tasks can be added easily, (iii) we introduce baseline models trained using IMPALA [24] and Random Network Distillation (RND) [13], a popular exploration bonus, resulting in agents that learn diverse policies for early stages of NetHack, and (iv) we demonstrate the benefit of NetHack’s symbolic observation space by presenting in-depth qualitative analyses of trained agents.

2 NetHack: a Frontier for Reinforcement Learning Research

In traditional so-called *roguelike* games (e.g., Rogue, Hack, NetHack, and Dungeon Crawl Stone Soup) the player acts turn-by-turn in a procedurally generated grid-world environment, with game dynamics strongly focused on exploration, resource management, and continuous discovery of entities and game mechanics [IRDC, 2008]. These games are designed to provide a steep learning curve and a constant level of challenge and surprise to the player. They are generally extremely difficult to win even once, let alone to master, i.e., win regularly and multiple times in a row.

As advocated by [39, 38, 18], procedurally generated environments are a promising direction for testing systematic generalization of RL agents. We argue that such environments need to be both sufficiently complex and fast to run to serve as a challenging long-term research testbed. In Section 2.1, we illustrate that NetHack contains many desirable properties, making it an excellent candidate for driving long-term research in RL. We introduce NLE in Section 2.2, an initial suite of tasks in Section 2.3, an evaluation protocol for measuring progress towards *solving* NetHack in Section 2.4, as well as baseline models in Section 2.5.

2.1 NetHack

NetHack is one of the oldest and most popular roguelikes, originally released in 1987 as a successor to *Hack*, an open-source implementation of the original *Rogue* game. At the beginning of the game, the player takes the role of a hero who is placed into a dungeon and tasked with finding the *Amulet of Yendor* to offer it to an in-game deity. To do so, the player has to descend to the bottom of over 50 procedurally generated levels to retrieve the amulet and then subsequently escape the dungeon, unlocking five extremely challenging final levels (the four Elemental Planes and the Astral Plane).

Many aspects of the game are procedurally generated and follow stochastic dynamics. For example, the overall structure of the dungeon is somewhat linear, but the exact location of places of interest (e.g., the *Oracle*) and the structure of branching sub-dungeons (e.g., the *Gnomish Mines*) are determined randomly. The procedurally generated content of each level makes it highly unlikely that a player will ever experience the exact same situation more than once. This provides a fundamental challenge to learning systems and a degree of complexity that enables us to more effectively evaluate an agent’s ability to generalize. It also disqualifies current state-of-the-art exploration methods such as Go-Explore [22, 23] that are based on a goal-conditioned policy to navigate to previously visited

¹“NetHack is largely based on discovering secrets and tricks during gameplay. It can take years for one to become well-versed in them, and even experienced players routinely discover new ones.” [26]



Figure 1: Annotated example of an agent at two different stages in NetHack (Left: a procedurally generated first level of the Dungeons of Doom, right: Gnomish Mines). A larger version of this figure is displayed in Figure 11 in the appendix.

states. Moreover, states in NetHack are composed of hundreds of possible symbols, resulting in an enormous combinatorial observation space.² It is an open question how to best project this symbolic space to a low-dimensional representation appropriate for methods like Go-Explore. For example, Ecoffet et al.’s heuristic of downsampling images of states to measure their similarity to be used as an exploration bonus will likely not work for large symbolic and procedurally generated environments. NetHack provides further variation by different hero roles (e.g., monk, valkyrie, wizard, tourist), races (human, elf, dwarf, gnome, orc) and random starting inventories (see Appendix A for details). Consequently, NetHack poses unique challenges to the research community and requires novel ways to determine state similarity and, likely, entirely new exploration frameworks.

To provide a glimpse into the complexity of NetHack’s environment dynamics, we closely follow the educational example given by “Mr Wendal” on YouTube.³ At a specific point in the game, the hero has to get past *Medusa’s Island* (see Figure 2 for an example). Medusa’s Island is surrounded by water **W** that the agent has to cross. Water can rust and corrode the hero’s metallic weapons **J** and armor **L**. Applying a can of grease **C** prevents rusting and corrosion. Furthermore, going into water will make a hero’s inventory wet, erasing scrolls **?!** and spellbooks **P** that they carry. Applying a can of grease to a bag or sack **B** will make it a waterproof container for items. But the sea can also contain a kraken **K** that can grab and drown the hero, leading to instant death. Applying a can of grease to a hero’s armor prevents the kraken from grabbing the hero. However, a cursed can of grease will grease the hero’s hands instead and they will drop their weapon and rings. One can use a towel **T** to wipe off grease. To reach Medusa **G**, the hero can alternatively use magic to freeze the water and turn it into walkable ice **I**. Wearing snow boots **S** will help the hero not to slip. When Medusa is in the hero’s line of sight, her gaze will petrify and instantly kill—the hero should use a towel to cover their eyes to fight Medusa, or even apply a mirror **M** to petrify her with her own gaze.

There are many other entities a hero must learn to face, many of which appear rarely even across multiple games, especially the most powerful monsters. These entities are often compositional, for example a monster might be a wolf **d**, which shares some characteristics with other in-game canines such as coyotes **D** or hell hounds **C**. To help a player learn, NetHack provides in-game messages

²Information about the over 450 items and 580 monster types, as well as environment dynamics involving these entities can be found in the NetHack Wiki [50] and to some extent in the NetHack Guidebook [59].

³youtube.com/watch?v=SjuTyJ1gLJ8

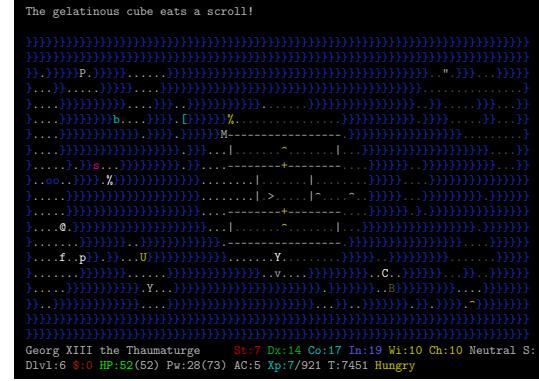


Figure 2: The hero (**G**) has to cross water (**W**) to get past Medusa (**G**, out of the hero’s line of sight) down the staircase (**>**) to the next level.

describing many of the hero’s interactions (see the top of Figure 1).⁴ Learning to capture these interesting and somewhat realistic albeit abstract dynamics poses challenges for multi-modal and language-conditioned RL [46].

NetHack is an extremely long game. Successful expert episodes usually last tens of thousands of turns, while average successful runs can easily last hundreds of thousands of turns, spawning multiple days of play-time. Compared to testbeds with long episode horizons such as StarCraft and Dota 2, NetHack’s “episodes” are one or two orders of magnitude longer, and they wildly vary depending on the policy. Moreover, several official *conducts* exist in NetHack that make the game even more challenging, e.g., by not wearing any armor throughout the game (see Appendix A for more).

Finally, in comparison to other classic roguelike games, NetHack’s popularity has attracted a larger number of contributors to its community. Consequently, there exists a comprehensive game wiki [50] and many so-called spoilers [25] that provide advice to players. Due to the randomized nature of NetHack, this advice is general in nature (e.g., explaining the behavior of various entities) and not a step-by-step guide. These texts could be used for language-assisted RL along the lines of [72]. Lastly, there is also a large public repository of human replay data (over five million games) hosted on the NetHack Alt.org (NAO) servers, with hundreds of finished games per day on average [47]. This extensive dataset could spur research advances in imitation learning, inverse RL, and learning from demonstrations [1, 3].

2.2 The NetHack Learning Environment

The NetHack Learning Environment (NLE) is built on NetHack 3.6.6, the 36th public release of NetHack, which was released on March 8th, 2020 and is the latest available version of the game at the time of publication of this paper. NLE is designed to provide a common, turn-based (i.e., synchronous) RL interface around the standard terminal interface of NetHack. We use the game *as-is* as the backend for our NLE environment, leaving the game dynamics unchanged. We added to the source code more control over the random number generator for seeding the environment, as well as various modifications to expose the game’s internal state to our Python frontend.

By default, the observation space consists of the elements *glyphs*, *chars*, *colors*, *specials*, *blstats*, *message*, *inv_glyphs*, *inv_strs*, *inv_letters*, as well as *inv_oclasses*. The elements *glyphs*, *chars*, *colors*, and *specials* are tensors representing the (batched) 2D symbolic observation of the dungeon; *blstats* is a vector of agent coordinates and other character attributes (“bottom-line stats”, e.g., health points, strength, dexterity, hunger level; normally displayed in the bottom area of the GUI), *message* is a tensor representing the current message shown to the player (normally displayed in the top area of the GUI), and the *inv_** elements are padded tensors representing the hero’s inventory items. More details about the default observation space and possible extensions can be found in Appendix B.

The environment has 93 available actions, corresponding to all the actions a human player can take in NetHack. More precisely, the action space is composed of 77 command actions and 16 movement actions. The movement actions are split into eight “one-step” compass directions (i.e., the agent moves a single step in a given direction) and eight “move far” compass directions (i.e., the agent moves in the specified direction until it runs into some entity). The 77 command actions include *eating*, *opening*, *kicking*, *reading*, *praying* as well as many others. We refer the reader to Appendix C as well as to the NetHack Guidebook [59] for the full table of actions and NetHack commands.

NLE comes with a Gym interface [11] and includes multiple pre-defined tasks with different reward functions and action spaces (see next section and Appendix E for details). We designed the interface to be lightweight, achieving competitive speeds with Gym-based ALE (see Appendix D for a rough comparison). Finally, NLE also includes a dashboard to analyze NetHack runs recorded as terminal tty recordings. This allows NLE users to analyze replays of the agent’s behavior at an arbitrary speed and provides an interface to visualize action distributions and game events (see Appendix H for details). NLE is available under an open source license at <https://github.com/facebookresearch/nle>.

⁴An example interaction after applying a figurine of an Archon: “*You set the figurine on the ground and it transforms. You get a bad feeling about this. The Archon hits! You are blinded by the Archon’s radiance! You stagger... It hits! You die... But wait... Your medallion feels warm! You feel much better! The medallion crumbles to dust! You survived that attempt on your life.*”

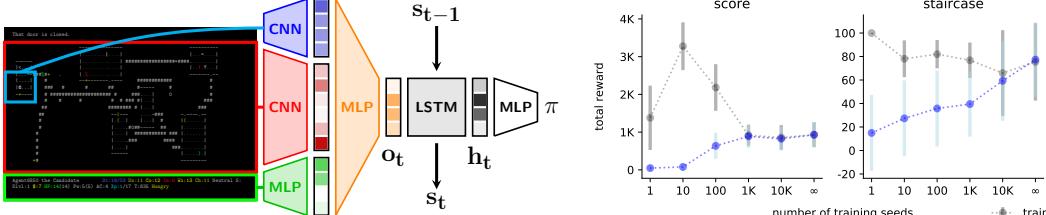


Figure 3: Overview of the core architecture of the baseline models released with NLE. A larger version of this figure is displayed in Figure 12 in the appendix.

Figure 4: Training and test performance when training on restricted sets of seeds.

2.3 Tasks

NLE aims to make it easy for researchers to probe the behavior of their agents by defining new tasks with only a few lines of code, enabled by NetHack’s symbolic observation space as well as its rich entities and environment dynamics. To demonstrate that NetHack is a suitable testbed for advancing RL, we release a set of initial tasks for tractable subgoals in the game: navigating to a **staircase** down to the next level, navigating to a staircase while being accompanied by a **pet**, locating and **eating** edibles, collecting **gold**, maximizing in-game **score**, **scouting** to discover unseen parts of the dungeon, and finding the **oracle**. These tasks are described in detail in Appendix E, and, as we demonstrate in our experiments, lead to unique challenges and diverse behaviors of trained agents.

2.4 Evaluation Protocol

We lay out a protocol and provide guidance for evaluating future work on NLE in a reproducible manner. The overall goal of NLE is to train agents that can solve NetHack. An episode in the full game of NetHack is considered solved if the agent retrieves the *Amulet of Yendor* and offers it to its co-aligned deity in the Astral Plane, thereby ascending to demigodhood. We declare NLE to be solved once agents can be trained to consecutively ascend (ten episodes without retry) to demigodhood on unseen seeds given a random role, race, alignment, and gender combination. Since the environment is procedurally generated and stochastic, evaluating on held-out unseen seeds ensures we test systematic generalization of agents. As of October 2020, NAO reports the longest *streak* of human ascensions on NetHack 3.6.x to be 61; the role, race, etc. are not necessarily randomized for these ascension streaks. Since we believe that this goal is out of reach for machine learning approaches in the foreseeable future, we recommend comparing models on the score task in the meantime. Using NetHack’s in-game score as the measure for progress has caveats. For example, expert human players can solve NetHack while minimizing the score [see 50, “Score” entry, for details]. NAO reports ascension scores for NetHack 3.6.x ranging from the low hundreds of thousands to tens of millions. Although we believe training agents to maximize the in-game score is likely insufficient for solving the game, the in-game score is still a sensible proxy for incremental progress on NLE as it is a function of, among other things, the dungeon depth that the agent reached, the number of enemies it killed, the amount of gold it collected, as well as the knowledge it gathered about potions, scrolls, and wands.

When reporting results on NLE, we require future work to state the full character specification (e.g., mon-hum-neu-mal), all NetHack options that were used (e.g., whether or not *autopickup* was used), which actions were allowed (see Table 1), which actions or action-sequences were hard-coded (e.g., engraving [see 50, “Elbereth” as an example]) and how many different seeds were used during training. We ask to report the average score obtained on 1000 episodes of randomly sampled and previously unseen seeds. We do not impose any restrictions during training, but at test time any save scumming (i.e., saving and loading previous checkpoints of the episode) or manipulation of the random number generator [e.g., 2] is forbidden.

2.5 Baseline Models

For our baseline models, we encode the multi-modal observation o_t as follows. Let the observation o_t at time step t be a tuple (g_t, z_t) consisting of the 21×79 matrix of glyph identifiers and a 21-dimensional vector containing agent stats such as its (x, y) -coordinate, health points, experience level, and so on. We produce three dense representations based on the observation (see Figure 3). For

every of the 5991 possible glyphs in NetHack (monsters, items, dungeon features, etc.), we learn a k -dimensional vector embedding. We apply a ConvNet (red) to all visible glyph embeddings as well as another ConvNet (blue) to the 9×9 crop of glyphs around the agent to create a dedicated egocentric representation for improved generalization [32, 71]. We found this egocentric representation to be an important component during preliminary experiments. Furthermore, we use an MLP to encode the hero’s stats (green). These vectors are concatenated and processed by another MLP to produce a low-dimensional latent representation \mathbf{o}_t of the observation. Finally, we employ a recurrent policy parameterized by an LSTM [33] to obtain the action distribution. For baseline results on the tasks above, we use a reduced action space that includes the movement, search, kick, and eat actions.

For the main experiments, we train the agent’s policy for 1B steps in the environment using IMPALA [24] as implemented in TorchBeast [44]. Throughout training, we change NetHack’s seed for procedurally generating the environment after every episode. To demonstrate NetHack’s variability based on the character configuration, we train with four different agent characters: a neutral human male monk (`mon-hum-neu-mal`), a lawful dwarf female valkyrie (`val-dwa-law-fem`), a chaotic elf male wizard (`wiz-elf-cha-mal`), and a neutral human female tourist (`tou-hum-neu-fem`). More implementation details can be found in Appendix F.

In addition, we present results using Random Network Distillation (RND) [13], a popular exploration technique for Deep RL. As previously discussed, exploration techniques which require returning to previously visited states such as Go-Explore are not suitable for use in NLE, but RND does not have this restriction. RND encourages agents to visit unfamiliar states by using the prediction error of a fixed random network as an intrinsic exploration reward, which has proven effective for hard exploration games such as Montezuma’s Revenge [12]. The intrinsic reward obtained from RND can create “reward bridges” between states which provide sparse extrinsic environmental rewards, thereby enabling the agent to discover new sources of extrinsic reward that it otherwise would not have reached. We replace the baseline network’s pixel-based feature extractor with the symbolic feature extractor described above for the baseline model, and use the best configuration of other RND hyperparameters documented by the authors (see Appendix G for full details).

3 Experiments and Results

We present quantitative results on the suite of tasks included in NLE using a standard distributed Deep RL baseline and a popular exploration method, before additionally analyzing agent behavior qualitatively. For each model and character combination, we present results of the mean episode return over the last 100 episodes averaged for five runs in Figure 5. We discuss results for individual tasks below (see Table 5 in the appendix for full details).

Staircase: Our agents learning to navigate the dungeon to the staircase  with a success rate of 77.26% for the monk, 50.42% for the tourist, 74.62% for the valkyrie, and 80.42% for the wizard. What surprised us is that agents learn to reliably kick in locked doors. This is a costly action to explore as the agent loses health points and might even die when accidentally kicking against walls. Similarly, the agent has to learn to reliably search for hidden passages and secret doors. Often, this involves using the search action many times in a row, sometimes even at many locations on the map (e.g., around all walls inside a room). Since NLE is procedurally generated, during training agents might encounter easier environment instances and use the acquired skills to accelerate learning on the harder ones [60, 18]. With a small probability, the staircase down might be generated near the agent’s starting position. Using RND exploration, we observe substantial gains in the success rate for the monk (+13.58pp), tourist (+6.52pp) and valkyrie (+16.34pp) roles, while lower results for wizard roles (-12.96pp).

Pet: Finding the staircase while taking care of the hero’s pet (e.g., the starting kitten  or little dog ) is a harder task as the pet might get killed or fall into a trap door, making it impossible for the agent to successfully complete the episode. Compared to the staircase task, the agent success rates are generally lower (62.02% for monk, 25.66% for tourist, 63.30% for valkyrie, and wizard 66.80%). Again, RND exploration provides consistent and substantial gains.

Eat: This tasks highlights the importance of testing with different character classes in NetHack. The monk and tourist start with a number edible items (e.g., food rations , apples  and oranges ). A

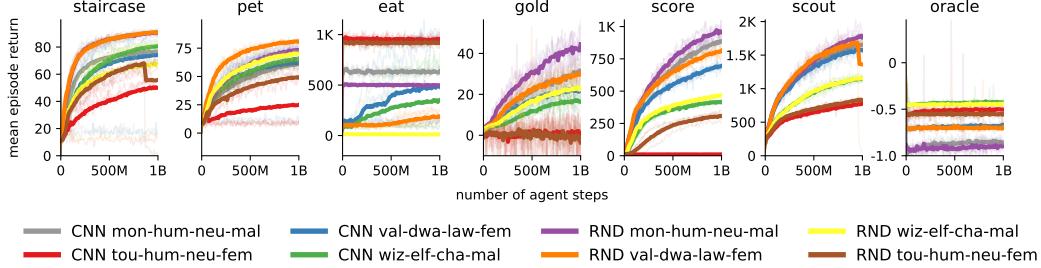


Figure 5: Mean return of the last 100 episodes averaged over five runs.

sub-optimal strategy is to consume all of these comestibles right at the start of the episode, potentially risking choking to death. In contrast, the other roles have to hunt for food, which our agents learn to do slowly over time for the valkyrie and wizard roles. By having more pressure to quickly learn a sustainable food strategy, the valkyrie learns to outlast other roles and survives the longest in the game (on average 1713 time steps). Interestingly, RND exploration leads to consistently worse results for this task.

Gold: Locating gold **\$** in NetHack provides a relatively sparse reward signal. Still, our agents learn to collect decent amounts during training and learn to descend to deeper dungeon levels in search for more. For example, monk agents reach dungeon level 4.2 on average for the CNN baseline and even 5.0 using RND exploration.

Score: As discussed in Section 2.4, we believe this task is the best candidate for comparing future methods regarding progress on NetHack. However, it is questionable whether a reward function based on NetHack’s in-game score is sufficient for training agents to solve the game. Our agents average at a score of 748 for monk, 11 for tourist, 573 for valkyrie, and 314 for wizard, with RND exploration again providing substantial gains (e.g. increasing the average score to 780 for monk). The resulting agents explore much of the early stages of the game, reaching dungeon level 5.4 on average for the monk with the deepest descent to level 11 achieving a high score of 4260 while leveling up to experience level 7 (see Table 6 in the appendix).

Scout: The scout task shows a trend that is similar to the score task. Interestingly, we observe a lower experience level and in-game score, but agents descend, on average, similarly deep into the dungeon (e.g. level 5.5 for monk). This is sensible, since a policy that avoids to fight monsters, thereby lowering the chances of premature death, will not increase the in-game score as fast or level up the character as quickly, thus keeping the difficulty of spawned monsters low. We note that delaying to level up in order to avoid encountering stronger enemies early in the game is a known strategy human players adopt in NetHack [e.g. 50, “Why do I keep dying?” entry, January 2019 version].

Oracle: None of our agents find the Oracle **o** (except for one lucky valkyrie episode). Locating the Oracle is a difficult exploration task. Even if the agent learns to make its way down the dungeon levels, it needs to search many, potentially branching, levels of the dungeon. Thus, we believe this task serves as a challenging benchmark for exploration methods in procedurally generated environments in the short term. Long term, many tasks harder than this (e.g., reaching *Minetown*, *Mines’ End*, *Medusa’s Island*, *The Castle*, *Vlad’s Tower*, *Moloch’s Sanctum* etc.) can be easily defined in NLE with very few lines of code.

3.1 Generalization Analysis

Akin to [18], we evaluate agents trained on a limited set of seeds while still testing on 100 held-out seeds. We find that test performance increases monotonically with the size of the set of seeds that the agent is trained on. Figure 4 shows this effect for the score and staircase tasks. Training only on a limited number of seeds leads to high training performance, but poor generalization. The gap between training and test performance becomes narrow when training with at least 1000 seeds, indicating

that at that point agents are exposed to sufficient variation during training to make memorization infeasible. We also investigate how model capacity affects performance by comparing agents with five different hidden sizes for the final layer (of the architecture described in Section 2.5). Figure 7 in the appendix shows that increasing the model capacity improves results on the score but not on the staircase task, indicating that it is an important hyperparameter to consider, as also noted by [18].

3.2 Qualitative Analysis

We analyse the cause for death of our agents during training and present results in Figure 9 in the appendix. We notice that starvation and traps become a less prominent cause of death over time, most likely because our agents, when starting to learn to descend dungeon levels and fight monsters, are more likely to die in combat before they starve or get killed by a trap. In the score and scout tasks, our agents quickly learn to avoid eating rotten corpses, but food poisoning becomes again prominent towards the end of training.

We can see that gnome lords  , gnome kings  , chameleons  , and even mind flayers  become a more prominent cause of death over time, which can be explained with our agents leveling up and descending deeper into the dungeon. Chameleons are a particularly interesting entity in NetHack as they regularly change their form to a random animal or monster, thereby adversarially confusing our agent with rarely seen symbols for which it has not yet learned a meaningful representation (similar to unknown words in natural language processing). We release a set of high-score recordings of our agents (see Appendix J on how to view them via a browser or terminal).

4 Related Work

Progress in RL has historically been achieved both by algorithmic innovations as well as development of novel environments to train and evaluate agents. Below, we review recent RL environments and delineate their strengths and weaknesses as testbeds for current methods and future research.

Recent Game-Based Environments: Retro video games have been a major catalyst for Deep RL research. ALE [9] provides a unified interface to Atari 2600 games, which enables testing of RL algorithms on high-dimensional visual observations quickly and cheaply, resulting in numerous Deep RL publications over the years [4]. The *Gym Retro* environment [51] expands the list of classic games, but focuses on evaluating visual generalization and transfer learning on a single game, *Sonic The Hedgehog*.

Both *StarCraft: BroodWar* and *StarCraft II* have been successfully employed as RL environments [64, 69] for research on, for example, planning [52, 49], multi-agent systems [27, 63], imitation learning [70], and model-free reinforcement learning [70]. However, the complexity of these games creates a high entry barrier both in terms of computational resources required as well as intricate baseline models that require a high degree of domain knowledge to be extended.

3D games have proven to be useful testbeds for tasks such as navigation and embodied reasoning. *Vizdoom* [42] modifies the classic first-person shooter game *Doom* to construct an API for visual control; *DeepMind Lab* [7] presents a game engine based on *Quake III Arena* to allow for the creation of tasks based on the dynamics of the original game; *Project Malmo* [37], MineRL [29] and *CraftAssist* [35] provide visual and symbolic interfaces to the popular *Minecraft* game. While *Minecraft* is also procedurally generated and has complex environment dynamics that an agent needs to learn about, it is much more computationally demanding than NetHack (see Table 4 in the appendix). As a consequence, the focus has been on learning from demonstrations [29].

More recent work has produced game-like environments with procedurally generated elements, such as the *Progen Benchmark* [18], *MazeExplorer* [30], and the *Obstacle Tower* environment [38]. However, we argue that, compared to NetHack or Minecraft, these environments do not provide the depth likely necessary to serve as long-term RL testbeds due to limited number of entities and environment interactions that agents have to learn to master. In contrast, NetHack agents have to acquire knowledge about complex environment dynamics of hundreds of entities (dungeon features, items, monsters etc.) to do well in a game that humans often take years of practice to solve.

In conclusion, none of the current benchmarks combine a fast simulator with a procedurally generated environment, a hard exploration problem, a wide variety of complex environment dynamics, and

numerous types of static and interactive entities. The unique combination of challenges present in NetHack makes NLE well-suited for driving research towards more general and robust RL algorithms.

Roguelikes as Reinforcement Learning Testbeds: We are not the first to argue for roguelike games to be used as testbeds for RL. Asperti et al. [5] present an interface to Rogue, the very first roguelike game and one of the simplest roguelikes in terms of game dynamics and difficulty. They show that policies trained with model-free RL algorithms can successfully learn rudimentary navigation. Similarly, Kanagawa and Kaneko [41] present an environment inspired by Rogue that provides a parameterizable generation of Rogue levels. Like us, Dannenhauer et al. [20] argue that roguelike games could be a useful RL testbed. They discuss the roguelike game *Dungeon Crawl Stone Soup*, but their position paper provides neither an RL environment nor experiments to validate their claims.

Most similar to our work is *gym_nethack* [14, 15], which offers a Gym environment based on NetHack 3.6.0. We commend the authors for introducing NetHack as an RL environment, and to the best of our knowledge they were the first to suggest the idea. However, there are several design choices that limit the impact and longevity of their version as a research testbed. First, they heavily modified NetHack to enable agent interaction. In the process, *gym_nethack* disables various crucial game mechanics to simplify the game, its environment dynamics, and the resulting optimal policies. This includes removing obstacles like boulders, traps, and locked doors as well as all item identification mechanics, making items much easier to employ and the overall environment much closer to its simpler predecessor, Rogue. Additionally, these modifications tie the environment to a particular version of the game. This is not ideal as (i) players tend to use new versions of the game as they are released, hence, publicly available human data becomes progressively incompatible, thereby limiting the amount of data that can be used for learning from demonstrations; (ii) older versions of NetHack tend to include well-documented exploits which may be discovered by agents (see Appendix I for exploits used in programmatic bots). In contrast, NLE is designed to make the interaction with NetHack as close as possible to the one experienced by humans playing the full game. NLE is the only environment exposing the entire game in all its complexity, allowing for larger-scale experimentation to push the boundaries of RL research.

5 Conclusion and Future Work

The NetHack Learning Environment is a fast, complex, procedurally generated environment for advancing research in RL. We demonstrate that current state-of-the-art model-free RL serves as a sensible baseline, and we provide an in-depth analysis of learned agent behaviors.

NetHack provides interesting challenges for exploration methods given the extremely large number of possible states and wide variety of environment dynamics to discover. Previously proposed formulations of intrinsic motivation based on seeking novelty [8, 53, 13] or maximizing surprise [56, 12, 57] are likely insufficient to make progress on NetHack given that an agent will constantly find itself in novel states or observe unexpected environment dynamics. NetHack poses further challenges since, in order to win, an agent needs to acquire a wide range of skills such as collecting resources, fighting monsters, eating, manipulating objects, casting spells, or taking care of their pet, to name just a few. The multilevel dependencies present in NetHack could inspire progress in hierarchical RL and long-term planning [21, 40, 55, 68]. Transfer to unseen game characters, environment dynamics, or level layouts can be evaluated [67]. Furthermore, its richness and constant challenge make NetHack an interesting benchmark for lifelong learning [45, 54, 61, 48]. In addition, the extensive documentation about NetHack can enable research on using prior (natural language) knowledge for learning, which could lead to improvements in generalization and sample efficiency [10, 46, 72, 36]. Lastly, NetHack can also drive research on learning from demonstrations [1, 3] since a large collection of replay data is available. In sum, we argue that the NetHack Learning Environment strikes an excellent balance between complexity and speed while encompassing a variety of challenges for the research community.

For future versions of the environment, we plan to support NetHack 3.7 once it is released, as it will further increase the variability of observations via *Themed Rooms*. This version will also introduce scripting in the Lua language, which we will leverage to enable users to create their custom sandbox tasks, directly tapping into NetHack and its rich universe of entities and their complex interactions to define custom RL tasks.

6 Broader Impact

To bridge the gap between the constrained world of video and board games, and the open and unpredictable real world, there is a need for environments and tasks which challenge the limits of current Reinforcement Learning (RL) approaches. Some excellent challenges have been put forth over the years, demanding increases in the complexity of policies needed to solve a problem or scale needed to deal with increasingly photorealistic, complex environments. In contrast, our work seeks to be extremely fast to run while still testing the generalization and exploration abilities of agents in an environment which is rich, procedurally generated, and in which reward is sparse. The impact of solving these problems with minimal environment-specific heuristics lies in the development of RL algorithms which produce sample efficient, robust, and general policies capable of more readily dealing with the uncertain and changing dynamics of “real world” environments. We do not solve these problems here, but rather provide the challenge and the testbed against such improvements can be produced and evaluated.

Auxiliary to this, and in line with growing concerns that progress in Deep RL is more the result of industrial labs having privileged access to the resources required to run environments and agents on a massive scale, the environment presented here is computationally cheap to run and to collect data in. This democratizes access for researchers in more resource-constrained labs, while not sacrificing the difficulty and richness of the environment. We hope that as a result of this, and of the more general need to develop sample-efficient agents with fewer data, the environmental impact of research using our environment will be reduced compared to more visually sophisticated ones.

Acknowledgements

We thank the NetHack DevTeam for creating and continuously extending this amazing game over the last decades. We thank Paul Winner, Bart House, M. Drew Streib, Mikko Juola, Florian Mayer, Philip H.S. Torr, Stephen Roller, Minqi Jiang, Vegard Mella, Eric Hambro, Fabio Petroni, Mikayel Samvelyan, Vitaly Kurin, Arthur Szlam, Sebastian Riedel, Antoine Bordes, Gabriel Synnaeve, Jeremy Reizenstein, as well as the NeurIPS 2020, ICML 2020, and BeTR-RL 2020 reviewers and area chairs for their valuable feedback. Nantas Nardelli is supported by EPSRC/MURI grant EP/N019474/1. Finally, we would like to pay tribute to the 863,918,816 simulated NetHack heroes who lost their lives in the name of science for this project (thus far).

References

- [1] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *ICML*, 2004.
- [2] Aransentin Breggan Hampe Pellsson. SWAGGINZZZ. <https://pellsson.github.io/>, 2019. Accessed: 2020-05-30.
- [3] Brenna Argall, Sonia Chernova, Manuela M. Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57:469–483, 2009.
- [4] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.
- [5] Andrea Aspertì, Daniele Cortesi, Carlo De Pieri, Gianmaria Pedrini, and Francesco Sovrano. Crawling in rogue’s dungeons with deep reinforcement techniques. *IEEE Transactions on Games*, 2019.
- [6] Yusuf Aytar, Tobias Pfaff, David Budden, Tom Le Paine, Ziyu Wang, and Nando de Freitas. Playing hard exploration games by watching youtube. In *NeurIPS*, 2018.
- [7] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab. *CoRR*, abs/1612.03801, 2016.

- [8] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *NeurIPS*, 2016.
- [9] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2013.
- [10] S. R. K. Branavan, David Silver, and Regina Barzilay. Learning to win by reading manuals in a monte-carlo framework. In *ACL*, 2011.
- [11] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.
- [12] Yuri Burda, Harrison Edwards, Deepak Pathak, Amos J. Storkey, Trevor Darrell, and Alexei A. Efros. Large-scale study of curiosity-driven learning. In *ICLR*, 2019.
- [13] Yuri Burda, Harrison Edwards, Amos J. Storkey, and Oleg Klimov. Exploration by random network distillation. In *ICLR*, 2019.
- [14] Jonathan Campbell and Clark Verbrugge. Learning combat in NetHack. In *AIIDE*, 2017.
- [15] Jonathan Campbell and Clark Verbrugge. Exploration in NetHack with secret discovery. *IEEE Transactions on Games*, 2018.
- [16] Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. Babyai: A platform to study the sample efficiency of grounded language learning. In *ICLR*, 2018.
- [17] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic Gridworld Environment for OpenAI Gym. <https://github.com/maximecb/gym-minigrid>, 2018.
- [18] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. *arXiv preprint arXiv:1912.01588*, 2019.
- [19] Karl Cobbe, Oleg Klimov, Christopher Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. In *ICML*, 2019.
- [20] Dustin Dannenhauer, Michael W Floyd, Jonathan Decker, and David W Aha. Dungeon crawl stone soup as an evaluation domain for artificial intelligence. *Workshop on Games and Simulations for Artificial Intelligence, AAAI*, 2019.
- [21] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In *NeurIPS*, 1992.
- [22] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Go-Explore: A New Approach for Hard-exploration Problems. *arXiv preprint arXiv:1901.10995*, 2019.
- [23] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. First return then explore. *CoRR*, abs/2004.12919, 2020.
- [24] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. In *ICML*, 2018.
- [25] Eva Myers. List of Nethack spoilers. <https://sites.google.com/view/evasroguelikegamessite/list-of-nethack-spoilers>, 2020. Accessed: 2020-06-03.
- [26] Juan Manuel Sanchez Fernandez. Reinforcement Learning for roguelike type games (eliteMod v0.9). https://kcir.pwr.edu.pl/~witold/aiarr/2009_projekty/elitmod/, 2009. Accessed: 2020-01-19.
- [27] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip H.S. Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. In *ICML*, 2017.

- [28] Javier García and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 2015.
- [29] William H. Guss, Cayden Codel, Katja Hofmann, Brandon Houghton, Noboru Kuno, Stephanie Milani, Sharada Mohanty, Diego Perez Liebana, Ruslan Salakhutdinov, Nicholay Topin, et al. The MineRL competition on sample efficient reinforcement learning using human priors. *NeurIPS Competition Track*, 2019.
- [30] Luke Harries, Sebastian Lee, Jaroslaw Rzepecki, Katja Hofmann, and Sam Devlin. Mazeexplorer: A customisable 3d benchmark for assessing generalisation in reinforcement learning. In *IEEE Conference on Games*, 2019.
- [31] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, Gabriel Dulac-Arnold, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep q-learning from demonstrations. In *AAAI*, 2017.
- [32] Felix Hill, Andrew K. Lampinen, Rosalia Schneider, Stephen Clark, Matthew Botvinick, James L. McClelland, and Adam Santoro. Emergent systematic generalization in a situated agent. In *ICLR*, 2020.
- [33] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.
- [34] International Roguelike Development Conference. Berlin Interpretation. http://www.roguebasin.com/index.php?title=Berlin_Interpretation, 2008. Accessed: 2020-01-08.
- [35] Yacine Jernite, Kavya Srinet, Jonathan Gray, and Arthur Szlam. Craftassist instruction parsing: Semantic parsing for a minecraft assistant. *CoRR*, abs/1905.01978, 2019.
- [36] Yiding Jiang, Shixiang Gu, Kevin Murphy, and Chelsea Finn. Language as an abstraction for hierarchical deep reinforcement learning. In *NeurIPS*, 2019.
- [37] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *IJCAI*, 2016.
- [38] Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Ervin Teng, Hunter Henry, Adam Crespi, Julian Togelius, and Danny Lange. Obstacle Tower: A Generalization Challenge in Vision, Control, and Planning. In *IJCAI*, 2019.
- [39] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Illuminating generalization in deep reinforcement learning through procedural level generation. *arXiv preprint arXiv:1806.10729*, 2018.
- [40] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 1996.
- [41] Yuji Kanagawa and Tomoyuki Kaneko. Rogue-gym: A new challenge for generalization in reinforcement learning. In *IEEE Conference on Games*, 2019.
- [42] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, 2016.
- [43] Kenneth Lorber. NetHack Home Page. <https://nethack.org>, 2020. Accessed: 2020-05-30.
- [44] Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. TorchBeast: A PyTorch Platform for Distributed RL. *arXiv*, abs/1910.03552, 2019.
- [45] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. In *NeurIPS*, 2017.

- [46] Jelena Luketina, Nantas Nardelli, Gregory Farquhar, Jakob Foerster, Jacob Andreas, Edward Grefenstette, Shimon Whiteson, and Tim Rocktäschel. A survey of reinforcement learning informed by natural language. In *IJCAI*, 2019.
- [47] M. Drew Streib. Public NetHack server at alt.org (NAO). <https://alt.org/nethack/>, 2020. Accessed: 2020-05-30.
- [48] Daniel J. Mankowitz, Augustin Zídek, André Barreto, Dan Horgan, Matteo Hessel, John Quan, Junhyuk Oh, Hado van Hasselt, David Silver, and Tom Schaul. Unicorn: Continual learning with a universal, off-policy agent. *arXiv*, abs/1802.08294, 2018.
- [49] Nantas Nardelli, Gabriel Synnaeve, Zeming Lin, Pushmeet Kohli, Philip H.S. Torr, and Nicolas Usunier. Value propagation networks. In *ICLR*, 2019.
- [50] NetHack Wiki. NetHackWiki. <https://nethackwiki.com/>, 2020. Accessed: 2020-02-01.
- [51] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta learn fast: A new benchmark for generalization in rl. *arXiv preprint arXiv:1804.03720*, 2018.
- [52] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 2013.
- [53] Georg Ostrovski, Marc G Bellemare, Aäron van den Oord, and Rémi Munos. Count-based exploration with neural density models. In *ICML*, 2017.
- [54] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 2019.
- [55] Ronald Parr and Stuart J. Russell. Reinforcement learning with hierarchies of machines. In *NeurIPS*, 1997.
- [56] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, 2017.
- [57] Roberta Raileanu and Tim Rocktäschel. RIDE: Rewarding impact-driven exploration for procedurally-generated environments. In *ICLR*, 2020.
- [58] Eric S. Raymond. *A Guide to the Mazes of Menace*, 1987.
- [59] Eric S. Raymond, Mike Stephenson, et al. *A Guide to the Mazes of Menace: Guidebook for NetHack*. NetHack DevTeam, 2020. URL <http://www.nethack.org/download/3.6.5/nethack-365-Guidebook.pdf>.
- [60] Sebastian Risi and Julian Togelius. Procedural content generation: From automatically generating game levels to increasing generality in machine learning. *CoRR*, abs/1911.13071, 2019.
- [61] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. In *NeurIPS*, 2019.
- [62] Tim Salimans and Richard Chen. Learning montezuma’s revenge from a single demonstration. *CoRR*, abs/1812.03381, 2018.
- [63] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G.J. Rudner, Chia-Man Hung, Philip H.S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft Multi-Agent Challenge. In *AAMAS*, 2019.
- [64] Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. TorchCraft: a Library for Machine Learning Research on Real-Time Strategy Games. *arXiv preprint arXiv:1611.00625*, 2016.
- [65] TAEB. TAEB Documentation: Other Bots. <https://taeb.github.io/bots.html>, 2015. Accessed: 2020-01-19.

- [66] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel. # Exploration: A study of count-based exploration for deep reinforcement learning. In *NeurIPS*, 2017.
- [67] Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal Machine Learning Research*, 2009.
- [68] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Manfred Otto Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *ICML*, 2017.
- [69] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittweis, et al. StarCraft II: A New Challenge for Reinforcement Learning. *arXiv preprint arXiv:1708.04782*, 2017.
- [70] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 2019.
- [71] Chang Ye, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. Rotation, translation, and cropping for zero-shot generalization. *CorR*, abs/2001.09908, 2020.
- [72] Victor Zhong, Tim Rocktäschel, and Edward Grefenstette. RTFM: Generalising to new environment dynamics via reading. In *ICLR*, 2020.

A Further Details on NetHack

Character options The player may choose (or pick randomly) the character from thirteen roles (archaeologist, barbarian, cave(wo)man, healer, knight, priest(ess), ranger, rogue, samurai, tourist, valkyrie, and wizard), five races (human, elf, dwarf, gnome, and orc), three moral alignments (neutral, lawful, chaotic), and two genders (male or female). Each choice determines some of the character's features, as well as how the character interacts with other entities (e.g., some species of monsters may not be hostile depending on the character race; priests of a particular deity may only help religiously aligned characters).

The hero's interaction with several game entities involves pre-defined stochastic dynamics (usually defined by virtual dice tosses), and the game is designed to heavily punish careless exploration policies.⁵ This makes NetHack an ideal environment for evaluating exploration methods such as curiosity-driven learning [56, 12] or safe reinforcement learning [28].

Learning and planning in NetHack involves dealing with partial observability. The game, by default, employs *Fog of War* to hide information based on a simple 2D light model (see for example the difference between white □ and gray □ room tiles in Figure 1 or Figure 11), requiring the player not only to discover the topology of the level (including searching for hidden doors and passages), but to also condition their policy on a world that might change, e.g., due to monsters spawning and interacting outside of the visible range.

On top of the standard ASCII interface, NetHack supports many official and unofficial graphical user interfaces. Figure 6 shows a screenshot of Lu Wang's BrowserHack⁶ as an example.



Figure 6: Screenshot of BrowserHack showing NetHack with a graphical user interface.

⁵Occasionally dying because of simple, avoidable mistakes is so common in the game that the online community has defined an acronym for it: *Yet Another Stupid Death* (YASD).

⁶Playable online at <https://coolwanglu.github.io/BrowserHack/>

Conducts While winning NetHack by retrieving and ascending with the Amulet of Yendor is already immensely challenging, experienced NetHack players like to challenge themselves even more by imposing additional restrictions on their play. The game tracks some of these challenges with the `#conduct` command [59]. These official challenges include eating only vegan or vegetarian food, or not eating at all, or playing the game in “pacifist” mode without killing a single monster. While very experienced players often try to adhere to several challenges at once, even moderately experienced players often limit their use of certain polymorph spells (e.g., “polypiling”—changing the form of several objects at once in the hope of getting better ones) or they try to beat the game while *minimizing* the in-game score. We believe this established set of conducts will supply the RL community with a steady stream of extended challenges once the standard NetHack Learning Environment is solved by future methods.

B Observation Space

The Gym environment is implemented by wrapping a more low-level NetHack Python object into a Python class responsible for the featurization, reward schedule and end-of-episode dynamics. While the low-level NetHack object gives access to a large number of NetHack game internals, the Gym wrapper exposes by default only a part of this data as numerical observation arrays, namely the observation tensors *glyphs*, *chars*, *colors*, *specials*, *blstats*, *message*, *inv_glyphs*, *inv_strs*, *inv_letters*, and *inv_oclasses*.

Glyphs, Char, Colors, Specials: NetHack supports non-ASCII graphical user interfaces, dubbed window-ports (see Figure 6 for an example). To support displaying different monsters, objects and floor types in the NetHack dungeon map as different tiles, NetHack internally defines *glyphs* as ids in the range $0, \dots, \text{MAX_GLYPH}$, where $\text{MAX_GLYPH} = 5991$ in our build⁷. The *glyph* observation is an integer array of shape $(21, 79)$ of these game glyph ids.⁸ In NetHack’s standard terminal-based user interface, these glyphs are mapped into ASCII characters of different colors which we return as the *chars*, *colors*, and *specials* observations, both all which are of shape $(21, 79)$; *chars* are ASCII bytes in the range $0, \dots, 127$ whereas *colors* are in range $0, \dots, 15$. For additional highlighting (e.g., flipping background and foreground colors for the hero’s pet), NetHack also computes xor’ed values which we return as the *specials* tensor.

Blstats: “Bottom line statistics”, a integer vector of length 25, containing the (x, y) coordinate of the hero and the following 23 character stats that typically appear in the bottom line of the ASCII interface: `strength_percentage`, `strength`, `dexterity`, `constitution`, `intelligence`, `wisdom`, `charisma`, `score`, `hitpoints`, `max_hitpoints`, `depth`, `gold`, `energy`, `max_energy`, `armor_class`, `monster_level`, `experience_level`, `experience_points`, `time`, `hunger_state`, `carrying_capacity`, `dungeon_number`, and `level_number`.

Message: A padded byte vector of length 256 representing the current message shown to the player, normally displayed in the top area of the GUI. We support different padding strategies and alphabet sizes, but by default we choose an alphabet size of 96, where the last character is used for padding.

Inventory: In NetHack’s default ASCII user interface, the hero’s inventory can be opened and closed during the game. Other user interfaces display a permanent inventory at all times. NLE follows that strategy. The inventory observations consist of the following four arrays: *inv_glyphs*: an integer vector of length 55 of glyph ids, padded with MAX_GLYPH ; *inv_strs*: A padded byte array of shape $(55, 80)$ describing the inventory items; *inv_letters*: A padded byte vector of length 55 with the corresponding ASCII character symbol; *inv_oclasses*: An integer vector of shape 55 with ids describing the type of inventory objects, padded with $\text{MAXOCLASSES} = 18$.

⁷The exact number of monsters in NetHack depends on compile-time options as well as the target operating system. For instance, the mail daemon `m` is only available on Unix-like operating systems, where it delivers email in the form of a NetHack scroll if the system is configured to host a Unix mailbox.

⁸NetHack’s set of glyph ids is not necessarily well suited for machine learning. For example, more than half of all glyph ids are of type “swallow”, most of which are guaranteed not to show up in any actual game of NetHack. We provide additional tooling to determine the type of a given glyph id to process this observation further.

The low-level NetHack Python object has some additional methods to query and modify NetHack’s game state, e.g. the current RNG seeds. We refer to the source code to describe these.⁹

C Action Space

The game of NetHack uses ASCII inputs, i.e., individual keyboard presses including modifiers like Ctrl and Meta. NLE pre-defines 98 actions, 16 of which are compass directions and 82 of which are command actions. Table 1 gives a list of command actions, including their ASCII value and the corresponding key binding in NetHack, while Table 3 lists the 16 compass directions. For a detailed description of these actions, as well as other NetHack commands, we refer the reader to the NetHack guide book [59]. Not all actions are sensible for standard RL training on NLE. E.g., the VERSION or QUIT actions are unlikely to be useful for direct input from the agent. NLE defines a list of USEFUL_ACTIONS that includes a subset of 76 actions; however, what is useful depends on the circumstances. In addition, even though an action like SAVE is unlikely to be useful in most game situations it corresponds to the letter S, which may be assigned to an inventory item or some other in-game menu entry such that it does become a useful action in that context.

By default, NLE will auto-apply the MORE action in situations where the game waits for input to display more messages.

Table 1: Command actions.¹⁰

Name	Value	Key	Description
EXTCMD	35	#	perform an extended command
EXTLIST	191	M-?	list all extended commands
ADJUST	225	M-a	adjust inventory letters
ANNOTATE	193	M-A	name current level
APPLY	97	a	apply (use) a tool (pick-axe, key, lamp...)
ATTRIBUTES	24	C-x	show your attributes
AUTOPICKUP	64	©	toggle the pickup option on/off
CALL	67	C	call (name) something
CAST	90	Z	zap (cast) a spell
CHAT	227	M-c	talk to someone
CLOSE	99	c	close a door
CONDUCT	195	M-C	list voluntary challenges you have maintained
DIP	228	M-d	dip an object into something
DOWN	62	>	go down (e.g., a staircase)
DROP	100	d	drop an item
DROPTYPE	68	D	drop specific item types
EAT	101	e	eat something
ESC	27	C-[escape from the current query/action
ENGRAVE	69	E	engrave writing on the floor
ENHANCE	229	M-e	advance or check weapon and spell skills
FIRE	102	f	fire ammunition from quiver
FIGHT	70	F	Prefix: force fight even if you don’t see a monster
FORCE	230	M-f	force a lock
GLANCE	59	;	show what type of thing a map symbol corresponds to
HELP	63	?	give a help message
HISTORY	86	V	show long version and game history
INVENTORY	105	i	show your inventory
INVENTTYPE	73	I	inventory specific item types
INVOKE	233	M-i	invoke an object’s special powers
JUMP	234	M-j	jump to another location
KICK	4	C-d	kick something
KNOWN	92	\	show what object types have been discovered
KNOWNCLASS	96	'	show discovered types for one class of objects
LOOK	58	:	look at what is here
LOOT	236	M-1	loot a box on the floor

⁹See, e.g., the `nethack.py` as well as `pynethack.cc` files in the NLE repository.

¹⁰The descriptions are mostly taken from the `cmd.c` file in the NetHack source code.

MONSTER	237	M-m	use monster's special ability
MORE	13	C-m	read the next message
MOVE	109	m	Prefix: move without picking up objects/fighting
MOVEFAR	77	M	Prefix: run without picking up objects/fighting
OFFER	239	M-o	offer a sacrifice to the gods
OPEN	111	o	open a door
OPTIONS	79	O	show option settings, possibly change them
OVERVIEW	15	C-o	show a summary of the explored dungeon
PAY	112	p	pay your shopping bill
PICKUP	44	,	pick up things at the current location
PRAY	240	M-p	pray to the gods for help
PREVMSG	16	C-p	view recent game messages
PUTON	80	P	put on an accessory (ring, amulet, etc)
QUAFF	113	q	quaff (drink) something
QUIT	241	M-q	exit without saving current game
QUIVER	81	Q	select ammunition for quiver
READ	114	r	read a scroll or spellbook
REDRAW	18	C-r	redraw screen
REMOVE	82	R	remove an accessory (ring, amulet, etc)
RIDE	210	M-R	mount or dismount a saddled steed
RUB	242	M-r	rub a lamp or a stone
RUSH	103	g	Prefix: rush until something interesting is seen
SAVE	83	S	save the game and exit
SEARCH	115	s	search for traps and secret doors
SEEALL	42	*	show all equipment in use
SEETRAP	94	~	show the type of adjacent trap
SIT	243	M-s	sit down
SWAP	120	x	swap wielded and secondary weapons
TAKEOFF	84	T	take off one piece of armor
TAKEOFFALL	65	A	remove all armor
TELEPORT	20	C-t	teleport around the level
THROW	116	t	throw something
TIP	212	M-T	empty a container
TRAVEL	95	-	travel to a specific location on the map
TURN	244	M-t	turn undead away
TWOWEAPON	88	X	toggle two-weapon combat
UNTRAP	245	M-u	untrap something
UP	60	<	go up (e.g., a staircase)
VERSION	246	M-v	list compile time options
VERSIONSHORT	118	v	show version
WAIT / SELF	46	.	rest one move while doing nothing / apply to self
WEAR	87	W	wear a piece of armor
WHATDOES	38	&	tell what a command does
WHATIS	47	/	show what type of thing a symbol corresponds to
WIELD	119	w	wield (put in use) a weapon
WIPE	247	M-w	wipe off your face
ZAP	112	z	zap a wand

D Environment Speed Comparison

Table 4 shows a comparison between popular Gym environments and NLE. All environments were controlled with a uniformly random policy using reset on terminal states. The tests were conducted on a MacBook Pro equipped with an Intel Core i7 2.9 GHz, 16GB of RAM, MacOS Mojave, Python 3.7, Conda 4.7.12, and latest available packages as of May 2020. *ObstacleTowerEnv* was instantiated with (`retro=False, real_time=False`). Note that this data does not necessarily reflect performance of these environments with better—or worse—policies, as each environment has computational dynamics that depend on its state. However, we expect the difference in terms of magnitude to remain mostly unchanged across these environments.

Table 3: Compass direction actions.

Direction	one-step		move far	
	Value	Key	Value	Key
North	107	k	75	K
East	108	l	76	L
South	106	j	74	J
West	104	h	72	H
North East	117	u	85	U
South East	110	n	78	N
South West	98	b	66	B
North West	121	y	89	Y

Table 4: Comparison between NLE and popular environments when using their respective Python Gym interface. SPS stands for “environment steps per second”. All environments but ObstacleTowerEnv were run via gym with standard settings (and headless when possible), for 60 seconds.

Environment	SPS	steps	episodes
NLE (score)	14.4K	868.75K	477
CartPole-v1	76.88K	4612.65K	207390
ALE (MontezumaRevengeNoFrameskip-v4)	0.90K	53.91K	611
Retro (Airstriker-Genesis)	1.31K	78.56K	52
ProcGen (procgen-coinrun-v0)	13.13K	787.98K	1283
ObstacleTowerEnv	0.06K	3.61K	6
MineRЛNavigateDense-v0	0.06K	3.39K	0

E Task Details

For all tasks described below, we add a penalty of -0.001 to the reward function if the agent’s action did not advance the in-game timer, which, for example, happens when the agent tries to move against a wall or navigates menus. For all tasks, except the *Gold* task, we disable NetHack’s *autopick* option [59]. Furthermore, we disable so-called *bones files* that would otherwise lead to agents occasionally discovering the remains and ghosts of previous agents, considerably increasing the variance across episodes.

Staircase The agent has to find the staircase down \blacktriangleright to the next dungeon level. This task is already challenging, as there is often no direct path to the staircase. Instead, the agent has to learn to reliably open doors \blacktriangleleft , kick-in locked doors, search for hidden doors and passages $\#$, avoid traps \blacksquare , or move

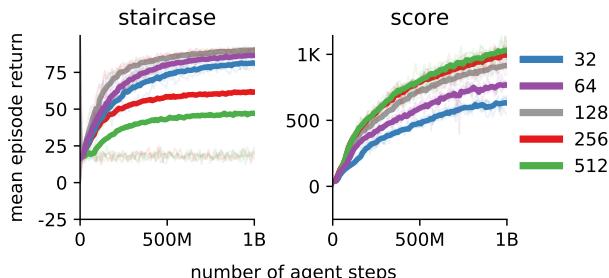


Figure 7: Mean episode return of the last 100 episodes for models with different hidden sizes averaged over five runs.

boulders **0** that obstruct a passage. The agent receives a reward of 100 once it reaches the staircase down and the episode terminates after 1000 agent steps.

Pet Many successful strategies for NetHack rely on taking good care of the hero’s pet (e.g., the little dog **d** or kitten **f** that the hero starts with). Pets are controlled by the game, but their behavior is influenced by the agent’s actions. In this task, the agent only receives a positive reward of 100 when it reaches the staircase while the pet is next to the agent.

Eat To survive in NetHack, players have to make sure their character does not starve to death. There are many edible objects in the game, for example food rations **%**, tins, and monster corpses. In this task, the agent receives the increase of nutrition as determined by the in-game “Hunger” status as reward [see 50, “Nutrition” entry for details]. A steady source of nutrition are monster corpses, but for that the agent has to learn to locate and to kill monsters while avoiding to consume rotten corpses, poisonous monster corpses such as Kobolds **k** or acidic monster corpses such as Acid Blobs **b**.

Gold Throughout the game, the player can collect gold **\$** to, for example, trade for useful items with shopkeepers. The agent receives the amount of gold it collects as reward. This incentivizes the agent to explore dungeon maps fully and to descend dungeon levels to discover new sources of gold. There are many advanced strategies for obtaining large amounts of gold such as finding, identifying and selling gems; stealing from or killing shopkeepers; or hunting for vaults or leprechaun halls. To make this task easier for the agent, we enable NetHack’s *autopickup* option for gold.

Scout An important part of the game is exploring dungeon levels. Here, we reward the agent (+1) for uncovering previously unknown tiles in the dungeon, for example by entering a new room or following a newly discovered passage. Like the previous task, this incentivizes the agent to explore dungeon levels and to descend.

Score In this task, the agent receives the increase of the in-game score between two time steps as reward. The in-game score is governed by a complex calculation, but in early stages of the game it is dominated by killing monsters and the number of dungeon levels that the agent descends [see 50, “Score” entry for details].

Oracle While levels are procedurally generated, there are a number of landmarks that appear in every game of NetHack. One such landmark is the Oracle **@**, which is randomly placed between levels five and nine of the dungeon. Reliably finding the Oracle is difficult, as it requires the agent to go down multiple staircases and often to exhaustively explore each level. In this task, the agent receives a reward of 1000 if it manages to reach the Oracle.

F Baseline CNN Details

As embedding dimension of the glyphs we use 32 and for the hidden dimension for the observation \mathbf{o}_t and the output of the LSTM \mathbf{h}_t , we use 128. For encoding the full map of glyphs as well as the 9×9 crop, we use a 5-layer ConvNet architecture with filter size 3×3 , padding 1 and stride 1. The input channel of the first layer of the ConvNet is the embedding size of the glyphs (32). Subsequent layers have an input and output channel dimension of 16. We employ a gradient norm clipping of 40 and clip rewards using $r_c = \tanh(r/100)$. We use RMSProp with a learning rate of 0.0002 without momentum and with $\varepsilon_{\text{RMSProp}} = 0.000001$. Our entropy cost is set to 0.0001.

G Random Network Distillation Details

For RND hyperparameters we mostly follow the recommendations by the authors [13]:

- we initialize the weights according to the original paper, using an orthogonal distribution with a gain of $\sqrt{2}$
- we use a two-headed value function rather than merely summing the intrinsic and extrinsic reward
- we use a discounting factor of 0.999 for the extrinsic reward and 0.99 for the intrinsic reward

- we use non-episodic intrinsic reward and episodic extrinsic reward
- we use reward normalization for the intrinsic reward, dividing it by a running estimate of its standard deviation

We modify a few of the parameters for use in our setting:

- we use exactly the same feature extraction architecture as the baseline model instead of the pixel-based convolutional feature extractor
- we do not use observation normalization, again due to the symbolic nature of our observation space
- before normalizing, we divide the intrinsic reward by ten so that it has less weight than the extrinsic reward
- we clip intrinsic rewards in the same way that we clip extrinsic rewards, i.e., using $r_c = \tanh(r/100)$, so that the intrinsic and extrinsic rewards are on a similar scale

We downscale the forward modeling loss by a factor of 0.01 to slow down the rate at which the model becomes familiar with a given state, since the intrinsic reward often collapsed quickly despite the reward normalization. We determined these settings during a set of small-scale experiments.

We also tried using subsets of the full feature set (only the embedding of the full display of glyphs, or only the embedding of the crop of glyphs around the agent) as well as the exact architecture used by the original authors, but with the pixel input replaced by a random 8-dimensional embedding of the symbolic observation space. However, we did not observe this improved results.

We tried using intrinsic reward only as the authors did in the original RND paper, but we found that agents trained in this way made no significant progress through the dungeon, even on a single fixed seed. This indicates that this form of intrinsic reward is not sufficient to make progress on NetHack. As noted in Section 3, the intrinsic reward did help in some tasks for some characters when combined with the extrinsic reward. Crucially, RND exploration is not sufficient for agents to learn to find the Oracle, which leaves this as a difficult challenge for future exploration techniques.

H Dashboard

We release a web dashboard built with NodeJS (see Figure 10) to visualize experiment runs and statistics for NLE, including replaying episodes that were recorded as `tty` files.

I NetHack Bots

Since the early stages of the development of NetHack, players have tried to build bots to play and solve the game. Notable examples are *TAEB*, *BotHack*, and *Saiph* [65, 50]. These bot frameworks largely rely on search heuristics and common planning methods, without generally making use of any statistical learning methods. An exception is *SWAGGINZZZ* [2] which uses lookups, exhaustive simulation and manipulation of the random number generator.

Successful bots have made use of exploits that are no longer present in recent versions of NetHack. For example, *BotHack* employs the “pudding farming” strategy [see 50, “Pudding farming” entry] to level up and to create items for the character by spawning and killing a large number of black puddings. This enabled the bot to become quite strong, which rendered late-game fights considerably easier. This strategy was disabled by the NetHack DevTeam with a patch that is incorporated into versions of NetHack above 3.6.0. Likewise, the random number generator manipulations employed in *SWAGGINZZZ* are no longer possible.

We believe that it is very unlikely that in the future we will see a hand-crafted bot solving NetHack in the way we defined it in Section 2.4. In fact, the creator of *SWAGGINZZZ* remarked that “[e]ven with RNG manipulation, writing a bot that 99% ascends NetHack is **extremely** complicated. So much stuff can go wrong, and there is no shortage of corner cases” [2].

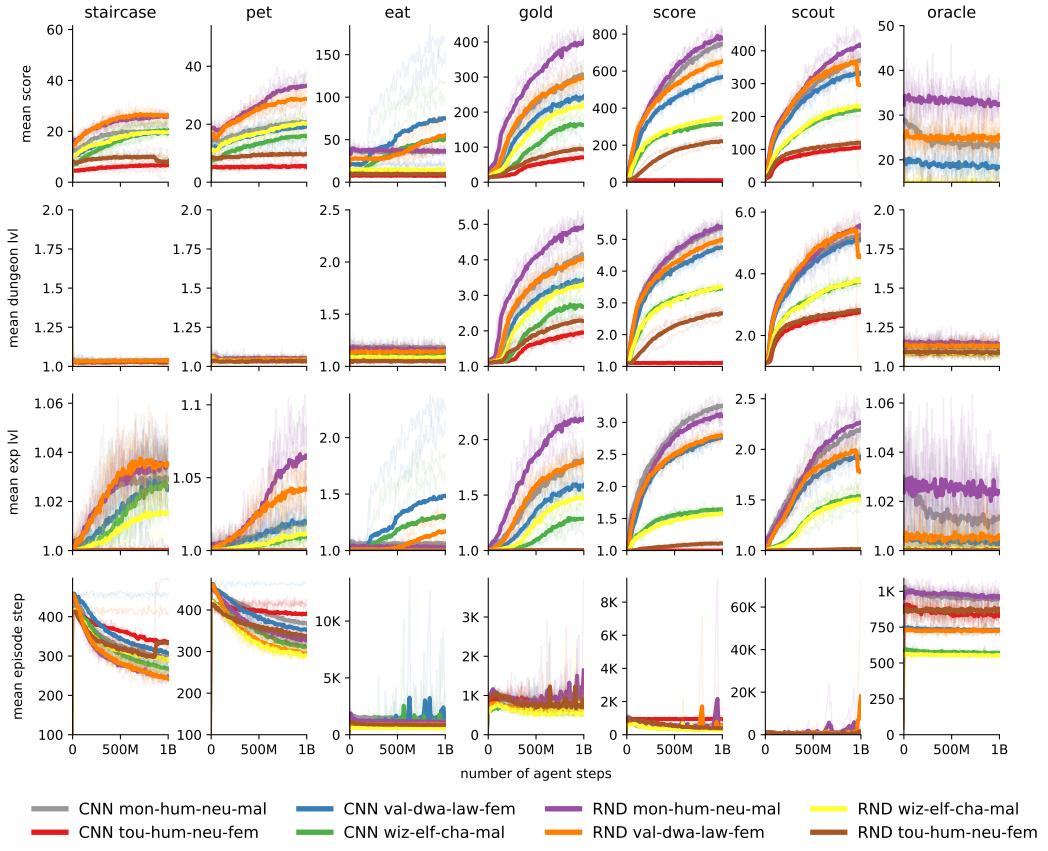


Figure 8: Mean score, dungeon level reached, experience level achieved, and steps performed in the environment in the last 100 episodes averaged over five runs.

J Viewing Agent Videos

We have uploaded some agent recordings to <https://asciinema.org/~nle>. These can be either watched on the Asciinema portal, or on a terminal by running `asciinema play -s 0.2 url` (asciinema itself is available as a pip package at <https://pypi.org/project/asciinema>). The `-s` flag regulates the speed of the recordings, which can also be modified on the web interface by pressing > (faster) or < (slower).

Table 5: Metrics averaged over last 1000 episodes for each task.

Task	Model	Character	Score	Time	Exp Lvl	Dungeon Lvl	Win
staircase	CNN	mon-hum-neu-mal	20	252	1.0	1.0	77.26
		tou-hum-neu-fem	6	288	1.0	1.0	50.42
		val-dwa-law-fem	19	329	1.0	1.0	74.62
		wiz-elf-cha-mal	20	253	1.0	1.0	80.42
	RND	mon-hum-neu-mal	26	199	1.0	1.0	90.84
		tou-hum-neu-fem	8	203	1.0	1.0	56.94
		val-dwa-law-fem	25	242	1.0	1.0	90.96
		wiz-elf-cha-mal	20	317	1.0	1.0	67.46
pet	CNN	mon-hum-neu-mal	20	297	1.0	1.1	62.02
		tou-hum-neu-fem	6	407	1.0	1.0	25.66
		val-dwa-law-fem	18	379	1.0	1.0	63.30
		wiz-elf-cha-mal	16	273	1.0	1.0	66.80
	RND	mon-hum-neu-mal	33	319	1.1	1.0	74.38
		tou-hum-neu-fem	10	336	1.0	1.0	49.38
		val-dwa-law-fem	28	311	1.0	1.0	81.56
		wiz-elf-cha-mal	20	278	1.0	1.0	70.48
eat	CNN	mon-hum-neu-mal	36	1254	1.1	1.2	–
		tou-hum-neu-fem	7	423	1.0	1.0	–
		val-dwa-law-fem	75	1713	1.5	1.1	–
		wiz-elf-cha-mal	50	1181	1.3	1.1	–
	RND	mon-hum-neu-mal	36	1102	1.0	1.2	–
		tou-hum-neu-fem	9	404	1.0	1.0	–
		val-dwa-law-fem	55	1421	1.2	1.1	–
		wiz-elf-cha-mal	14	808	1.0	1.1	–
gold	CNN	mon-hum-neu-mal	307	947	1.8	4.2	–
		tou-hum-neu-fem	71	788	1.0	2.0	–
		val-dwa-law-fem	245	1032	1.6	3.5	–
		wiz-elf-cha-mal	162	780	1.3	2.7	–
	RND	mon-hum-neu-mal	403	1006	2.2	5.0	–
		tou-hum-neu-fem	92	816	1.0	2.2	–
		val-dwa-law-fem	298	998	1.8	4.0	–
		wiz-elf-cha-mal	217	789	1.5	3.3	–
score	CNN	mon-hum-neu-mal	748	932	3.2	5.4	–
		tou-hum-neu-fem	11	795	1.0	1.1	–
		val-dwa-law-fem	573	908	2.8	4.8	–
		wiz-elf-cha-mal	314	615	1.6	3.5	–
	RND	mon-hum-neu-mal	780	863	3.1	5.4	–
		tou-hum-neu-fem	219	490	1.1	2.6	–
		val-dwa-law-fem	647	857	2.8	5.0	–
		wiz-elf-cha-mal	352	585	1.6	3.5	–
scout	CNN	mon-hum-neu-mal	372	838	2.2	5.3	–
		tou-hum-neu-fem	105	580	1.0	2.7	–
		val-dwa-law-fem	331	852	1.9	5.1	–
		wiz-elf-cha-mal	222	735	1.5	3.8	–
	RND	mon-hum-neu-mal	416	924	2.3	5.5	–
		tou-hum-neu-fem	119	599	1.0	2.8	–
		val-dwa-law-fem	304	1021	1.8	4.6	–
		wiz-elf-cha-mal	231	719	1.5	3.8	–
oracle	CNN	mon-hum-neu-mal	24	876	1.0	1.1	0.00
		tou-hum-neu-fem	9	674	1.0	1.1	0.00
		val-dwa-law-fem	18	1323	1.0	1.1	0.02
		wiz-elf-cha-mal	10	742	1.0	1.1	0.00
	RND	mon-hum-neu-mal	32	967	1.0	1.1	0.00
		tou-hum-neu-fem	13	811	1.0	1.1	0.00
		val-dwa-law-fem	26	1353	1.0	1.1	0.00
		wiz-elf-cha-mal	14	791	1.0	1.1	0.00

Table 6: Top five of the last 1000 episodes in the score task.

Model	Character	Killer Name	Score	Exp Lvl	Dungeon Lvl
CNN	mon-hum-neu-mal	warg	4408	7	9
		forest centaur	4260	7	11
		hill orc	2880	6	8
		gnome lord	2848	6	9
		crocodile	2806	6	8
	tou-hum-neu-fem	jackal	200	1	3
		hobgoblin	200	1	5
		hobbit	200	1	3
		giant rat	190	1	4
		large kobold	174	1	4
	val-dwa-law-fem	gnome lord	2176	5	12
		ape	1948	6	7
		gremlin	1924	5	11
		gnome king	1916	5	11
		vampire	1864	4	10
	wiz-elf-cha-mal	dingo	1104	3	9
		giant ant	1008	3	8
		gnome mummy	988	3	8
		coyote	988	3	9
		kicking a wall	972	3	8
RND	mon-hum-neu-mal	rothe	3664	5	7
		rotted dwarf corpse	3206	5	7
		leocrotta	2771	5	11
		winter wolf cub	2724	6	9
		starvation	2718	6	6
	tou-hum-neu-fem	grid bug	1432	1	7
		sewer rat	1253	1	4
		bolt of cold	1248	1	3
		goblin	1125	1	4
		goblin	1078	1	4
	val-dwa-law-fem	bugbear	2186	6	9
		starvation	2150	5	10
		ogre	2095	5	9
		rothe	2084	6	8
		Uruk-hai called Haiaigrisai of Aruka	2036	5	6
	wiz-elf-cha-mal	cave spider	1662	2	7
		iguana	1332	2	5
		starvation	1329	1	5
		starvation	1311	1	5
		gnome lord	1298	5	9

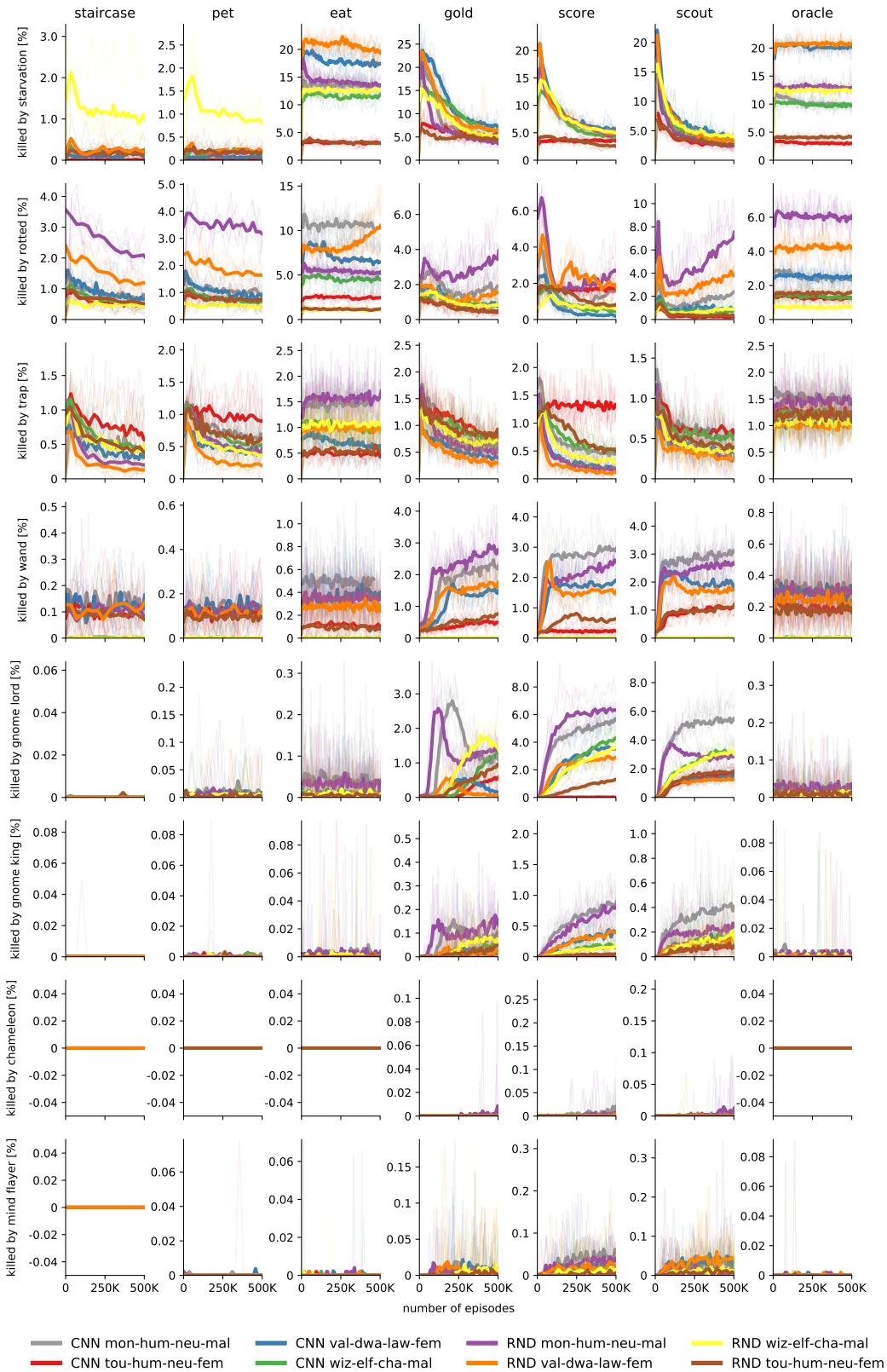


Figure 9: Analysis of different causes of death during training, averaged over the last 1000 episodes and over five runs.

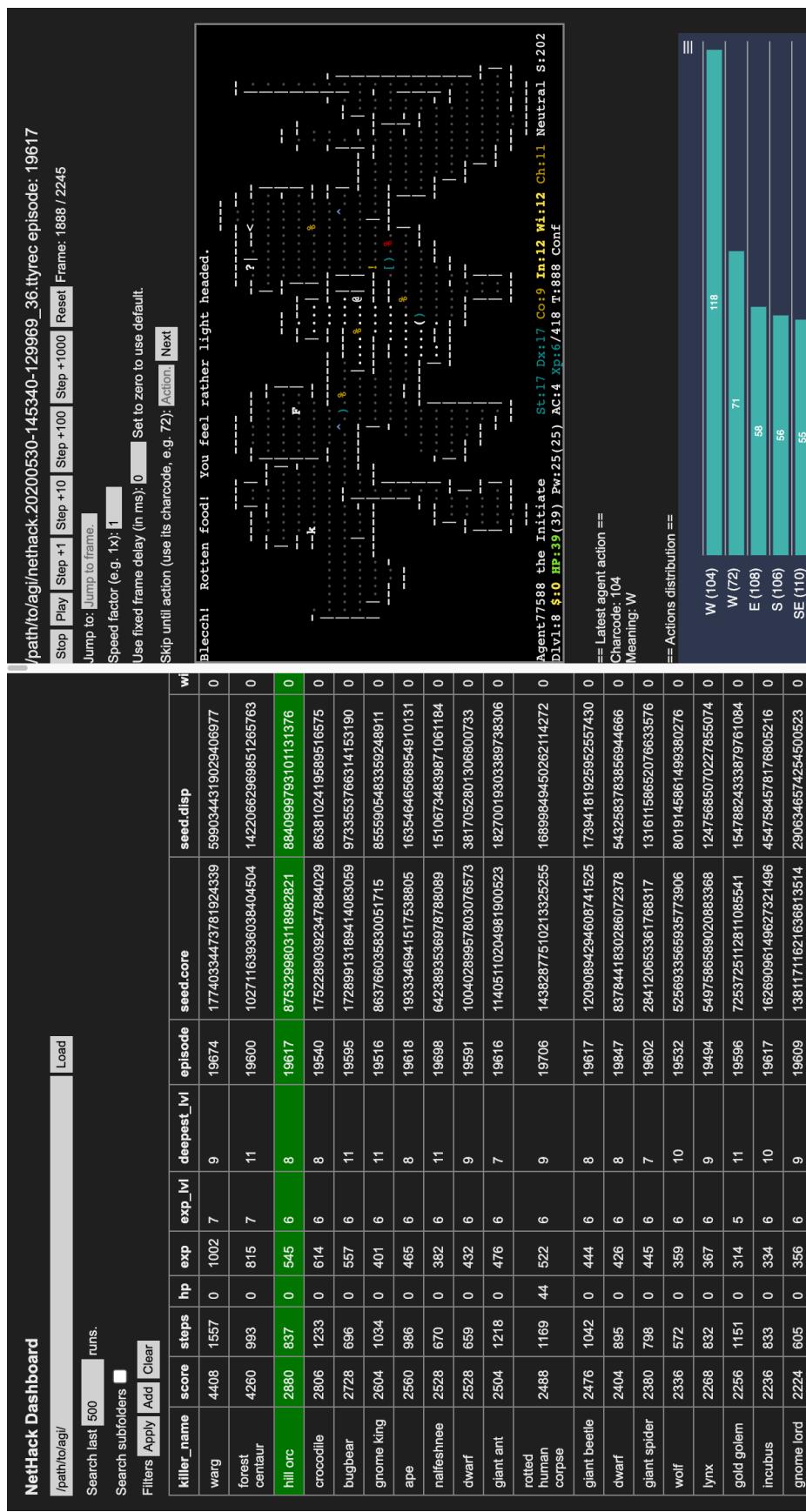


Figure 10: Screenshot of the web dashboard included in the NetHack Learning Environment.

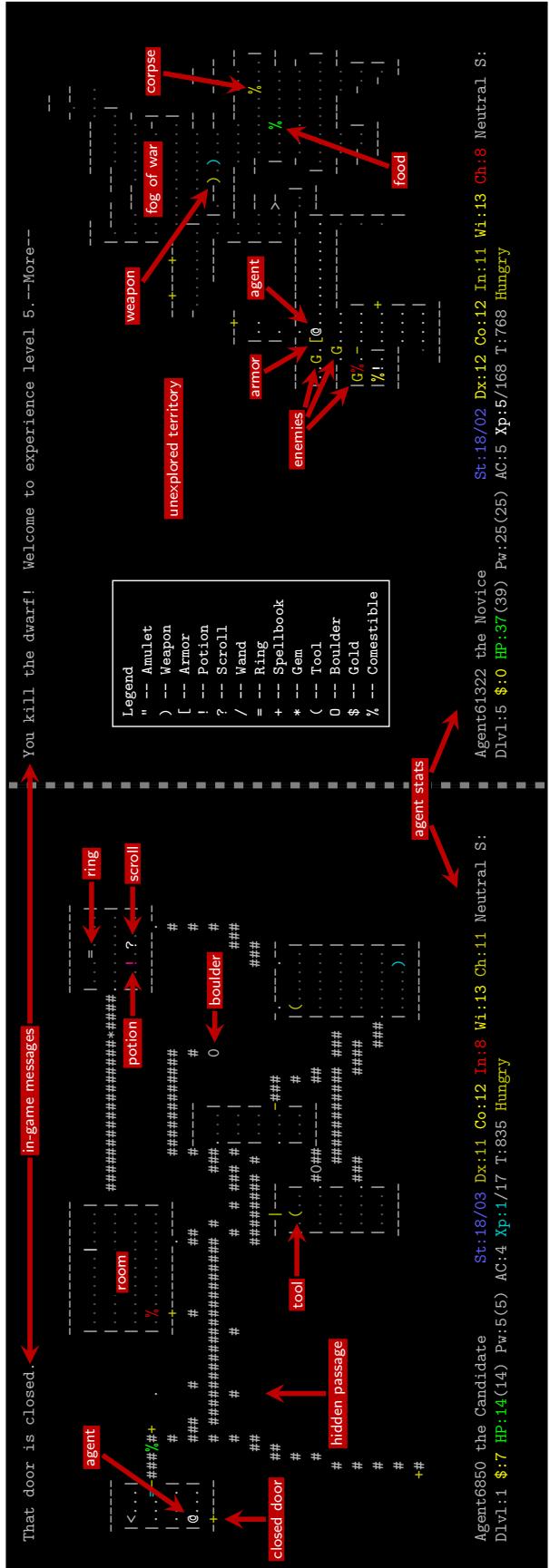


Figure 11: Annotated example of an agent at two different stages in NetHack (Left: a procedurally generated first level of the Dungeons of Doom, right: Gnomish Mines).

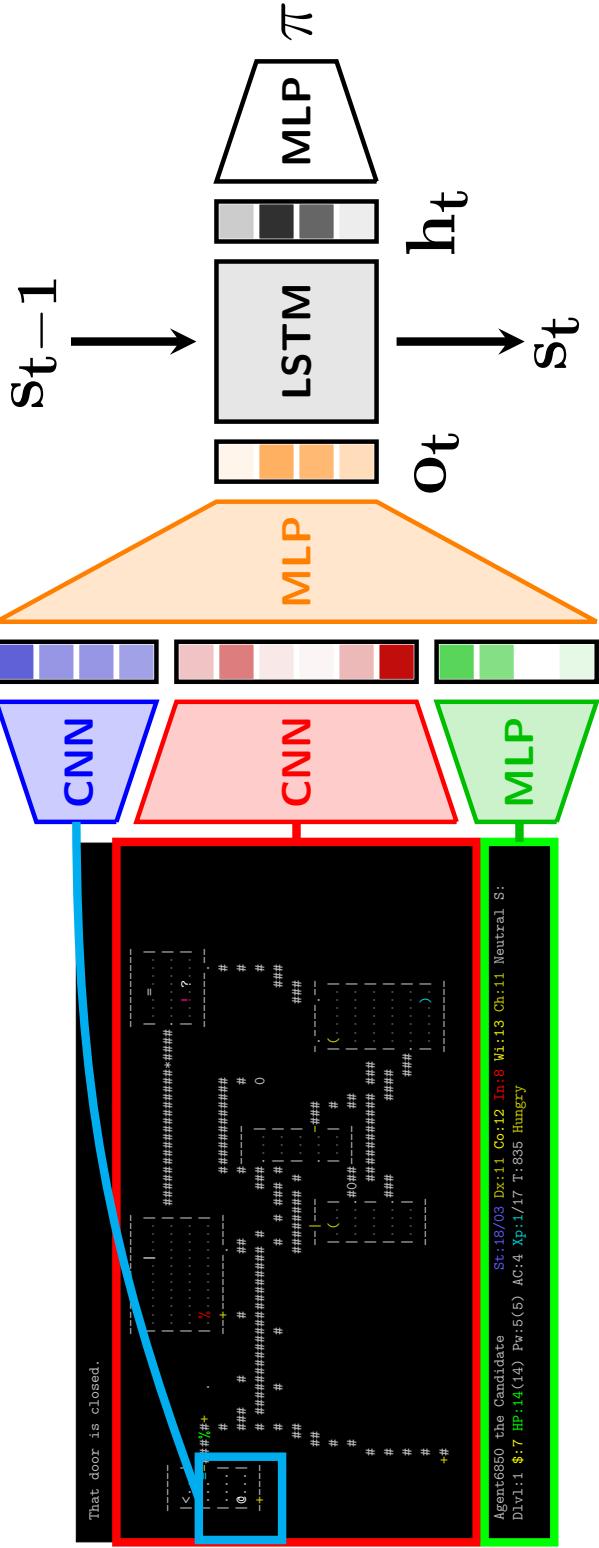


Figure 12: Overview of the core architecture of the baseline models released with NLE.