# Learning Combat in NetHack

**Jonathan Campbell** and **Clark Verbrugge**
School of Computer Science
McGill University, Montréal
jcampb35@cs.mcgill.ca
clump@cs.mcgill.ca

## Abstract

Combat in *roguelikes* involves careful strategy to best match a large variety of items and abilities to a given opponent, and the significant scripting effort involved can be a major barrier to automation. This paper presents a machine learning approach for a subset of combat in the game of *NetHack*. We describe a custom learning approach intended to deal with the large action space typical of this genre, and show that it is able to develop and apply reasonable strategies, outperforming a simpler baseline approach. These results point towards better automation of such complex game environments, facilitating automated testing and design exploration.

## Introduction

In many game genres combat can require non-trivial planning, selecting attack and defense strategies appropriate to the situation, while also managing resources to ensure future combat capability. This arrangement is particularly and notoriously true of *roguelikes*, which often feature a wide range of weaponry, items, and abilities that have to be well-matched to an also widely varied range of opponents. For game AI this becomes an interesting and complex problem, requiring the system to choose among an extremely large set of actions, which likewise can be heavily dependent on context. As combat behaviours rest on basic movement control and state recognition, the combined problem poses a particularly difficult challenge for learning approaches where learning costs are a factor.

In this work we describe a machine learning approach addressing one-on-one, player vs. monster combat in the paradigmatic roguelike *NetHack*. We focus on the core combat problem, applying a deep learning technique to the basic problem of best selecting weapons, armour, and items for optimizing combat success. To reduce the learning complexity and accelerate the learning process, we build on a novel, abstracted representation of the action set and game state. This representation limits generality of the AI, but allows it to focus on learning relevant combat strategy, applying the right behaviour in the right context, and relying on well-understood algorithmic solutions to lower-level behaviours such as pathing.



We evaluate our learning-based results on two scenarios, one that isolates the weapon selection problem, and one that is extended to consider a full inventory context. Our approach shows improvement over a simple (scripted) baseline combat strategy, and is able to learn to choose appropriate weaponry and take advantage of inventory contents.

A learned model for combat eliminates the tedium and difficulty of hard-coding responses for each monster and player inventory arrangement. For roguelikes in general, this is a major source of complexity, and the ability to learn good responses opens up potential for AI bots to act as useful design agents, enabling better game tuning and balance control. Further automation of player actions also has advantage in allowing players to confidently delegate highly repetitive or routine tasks to game automation, and facilitates players operating with reduced interfaces (Sutherland 2017).

Specific contributions of this work include:

- We describe a deep learning approach to a subset of NetHack combat. Our design abstracts higher-level actions to accelerate learning in the face of an otherwise very large low-level action set and state space.
- We apply our design to two NetHack combat contexts, considering both a basic weapon selection problem and an extended context with a full inventory of combat-related items.
- Experimental work shows the learned result is effective, generally improving over a simple, scripted baseline. Detailed examination of learned behaviour indicates appropriate strategies are selected.

## Related Work

There has been much recent interest in applying reinforcement learning to video games. Mnih et al. notably proposed the Deep Q-Network (DQN) and showed better than human-level performance on a large set of Atari games, including Breakout, Pong, and Q*bert (2013; 2015). Their model uses raw pixels (frames) from the game screen for the state space, or in a formulation proposed by (Sygnowski and Michalewski 2017), game RAM. We use the same DQN approach here, but with a hand-crafted game state and action set to speed up learning, given the much larger basic action set and complex state space typical of roguelikes. Kempka et al. demonstrated the use of deep Q-learning

on the 3D video game Doom, similarly using visual input (2016), with applications by (Lample and Chaplot 2017). Narasimhan et al. use a deep learning approach to capture game state semantics in text-based Multi-User Dungeon (MUD) games (2015).

Abstracted game states have been previously studied for RTS games (Uriarte and Ontañón 2014) as well as board games like Dots-and-Boxes (Zhuang et al. 2015), in order to improve efficiency on otherwise intractable state spaces. We use an abstracted state space here for the same purpose.

RL approaches have also been applied to other, more traditional types of games. The use of separate value and policy networks combined with Monte-Carlo simulation has been shown to do very well on the game of Go (Silver et al. 2016), while using RL to learn Nash equilibriums in games like poker has also been studied (Heinrich and Silver 2016). Competitions also exist for general game AI, such as the General Video Game AI Competition (Liebana et al. 2016).

A small handful of heuristic-based automated players exist for NetHack. The 'BotHack' player was in 2015 the first bot (possibly only so far) to finish the entire game. This bot relies on hard-coded strategies for all NetHack mechanics. For combat, it uses movement-based strategies like luring monsters into narrow corridors or keeping close to the exit staircase to have an option to retreat (Krajicek 2015a; 2015b). It also prefers to use ranged weapons against monsters with special close-range attacks and abilities. Although the BotHack player has beaten the entire game once (no small feat), its average success rate is unclear. An automated player also exists for *Rogue* (a 1980 predecessor to NetHack) called Rog-O-Matic. It uses an expert-system approach, although simple learning is enabled through a persistent store (Mauldin et al. 1984).

## Environment

We use NetHack for our combat environment, an archetypal roguelike made in the late '80s that is still popular and getting updates to this day. A turn-based game, it takes place on a 2D ASCII-rendered grid, with a player avatar able to move around, pick up items, fight monsters, and travel deeper into the dungeon by issuing different keyboard commands.

In every level of the dungeon lurk devious monsters – some having brute strength, others possessing nightmarish abilities such as causing the player to turn to stone, splitting in two every time the player tries to attack, or shooting death rays. A player must develop strategies for each particular monster in order to survive. These strategies typically involve the choice of a specific weapon to attack with, armor to equip, and/or the use of special scrolls, potions, wands, spells, and other miscellaneous items. For example, when fighting a pudding-type monster, weapons made of iron should not be used, since attacking with them would cause the pudding to divide in two (NetHack Wiki 2017).

NetHack is a difficult game, with over 375 monsters that a player must familiarize themselves with. Although detailed statistics for the game are unavailable, on at least one server where NetHack can be played, total ascension (win) rate was only 0.605% of over one million completed games by mid-2017, with the vast majority of failures arising from monster combat (NAO public NetHack server 2017).

Items are also randomly placed throughout the dungeon. As mentioned above, there are many different item types: weapons (melee and ranged), armor (helms, gloves, etc.), scrolls, potions, wands, and more. Each item can be one of blessed, uncursed or cursed (referred to as BUC status); blessed items are more powerful than their uncursed counterparts while cursed items may cause deleterious effects. Further, each item is made of a certain material (iron, silver, wood, etc.); materials interact with the properties of certain monsters (e.g., many demons are particularly susceptible to silver weapons). Weapons and armor can also have an enchantment level (positive or negative) which relates to their damage output, as well as a condition based on their material (wood items can be burnt, iron items corroded, etc.).

Another important mechanic of NetHack is resource management. Player turns are limited by food availability, so efficient exploration and shorter combats are ideal; items are also scarce, so their conservation for stronger monsters is also advised. We do not deal with these issues in this paper.

### Modifications

We use a modified version of NetHack to allow for experiments that focus singularly on combat and item selection. Game features that might confound experiment results were disabled, including starvation, weight restrictions, and item identification; we leave these issues for future work.

The *ZMQ* socket library is used as a basic interface between our code and the NetHack game (Hintjens 2011). Through ZMQ we send the keyboard character(s) corresponding to a chosen action, and NetHack sends back the currently-visible game map and player attributes/statistics (i.e., all information that is otherwise visible to a player via the typical NetHack console interface). After each action is taken, the player's inventory is also queried to ensure an accurate state. Using ZMQ sockets allows for much faster gameplay than the typical approach of terminal emulation.

## Learning Approach

We use a Deep Q-Network for our learning algorithm, detailed below, as well as two trivial baseline algorithms that approximate a beginner player's actions.

### Baseline algorithms

To compare our deep Q-network approach, we present two simple algorithms for combat in our NetHack environment.

The first baseline equips a random weapon from its inventory at the start of each combat episode, then moves towards the monster and attacks when in range; if a ranged weapon is chosen, the agent will line up with the monster and then fire projectiles until supply is exhausted. If the monster is invisible, a random move in any direction will be attempted (moving into a monster is considered an attack, so randomly moving has a chance to attack an invisible monster).

The second baseline has access to a wider variety of items. It equips a random weapon and tries to equip all armor it may have. It then follows the behavior of the former baseline

(approaching/attacking) 75% of the time, with the other 25% dedicated to using a random item from its inventory (a scroll, potion, or wand). This behavior better replicates that of a typical player who will occasionally use an item but spend the majority of their time on movement.

## Deep Q-Network

A *dueling double deep Q-network* (Wang et al. 2016) with experience replay is used for the learning agent. Although a tabular form may be successful, a deep network allows for generalizability and far more compact representation.

Rewards are given as follows: a small negative reward at each timestep to inspire shorter combat, while at episode end, a reward of 10 on monster death and -1 on player death. Other reward weights are of course possible; our parameters were selected based on preliminary experimentation.

Our design depends heavily on a hand-crafted state space and action set, as described in the text below.

**States** The state space for video games is most generically defined over the raw pixels from the game screen, and actions defined as the basic controller inputs, such as done in the Atari game experiments (Mnih et al. 2013). This approach is very easy to set up and can be ported to diverse environments, but depends on a fairly simple notion of targeting and combat strategy.

In our approach we provide a more high level representation of state. Doing so allows us to more easily address effects with long temporal dependencies and better handle the diversity of game elements and command options available in NetHack, i.e., to focus more directly on learning combat strategy instead of the more general context in which we also would need to learn the basics of pathing, maze navigation, and interpreting inventory and character status, all of which already have well-known and efficient algorithmic solutions.

Game state is parsed from the typical game information screen visible to a player. We encode as basic information,

- the player's normalized experience level, health, power, strength, and other attributes,
- the player's current status effects (confused, stunned, blinded, hallucinating, etc.),
- the player's current inventory with each ⟨item, enchantment, BUC-status, condition⟩ item tuple separately represented, and with normalized 0..1 values for projectile quantity and wand charges,
- what the player is currently wielding and wearing,
- the normalized distance between the player and monster.

Additionally, we include one-hot vectors to represent,

- the current monster being fought,
- the player's character type,
- the player's alignment,

and finally simple booleans for,

- whether the player has lost health this game,
- if the player is currently invisible,
- whether the player is lined up with the monster,
- if there are projectiles currently on the ground and if the player is standing on them,
- and whether both the player and monster have recently moved or approached the other.

Note that the actual game map is not included in the state. This choice was made to slim down the state space and decrease learning times. Since combat is one-on-one, and we already include monster-player distance and other combat-relevant information abstractly, it is unlikely that having the detailed game map would significantly improve success rate. In a multi-combat scenario, however, adding the game map to the state in conjunction with CNNs could potentially lead to more clever movement tactics.

**Actions** In many games the set of actions corresponds directly to controller inputs. In NetHack, however, keyboard commands can map to different actions depending on game context; keyboard characters used to access inventory items, for example, are determined by the game based on the order in which the items are acquired, more or less randomizing the command-action association in every playthrough. In order to allow a learning agent to select an appropriate item without the expense and complexity of incorporating a full game history into the learning process, we also abstract the set of actions.

Our game controls are divided into basic primitives, and actions that map onto inventory items. For the latter, the action performed depends on the item type. If an equippable item (weapon/armor) is selected as the action, then that item will be equipped. If the item is usable, it will be used instead (scrolls are read, wands zapped at the monster, and potions have two actions — either quaffing or throwing at the monster). Each item is represented many times in the action set, one time for each of the different item enchantment, condition, and BUC status combinations, as the decision to use an item strongly depends on these properties. Enchantments are capped in the [-1, +1] range to reduce complexity.

Item actions are complemented by nine primitive actions related to basic movement or other non-item strategies as follows. (Note that while the associated movement keys never change, abstracted forms are still required here since the game map is not passed in.)

- Move one unit towards the monster.
- Move one unit to line up with the monster (to allow for projectiles/wands/potions to be thrown), breaking ties in position by choosing the position closer to the monster.
- Move one unit to line up with the monster, breaking ties in position by choosing the position farther from the monster.
- Attack the monster with the currently-equipped melee weapon.
- Unequip the current weapon (switch to bare hands).
- Move one unit towards the closest projectile on the ground.
- Pick up a projectile on the ground under the player.
- Move one unit in a random direction.
- Wait a turn.

Many actions notably have prerequisites: you can melee attack a monster only when in melee range, use an item only

if you possess it, etc. At any time, the vast majority of actions will be missing a prerequisite; while in some cases their absence is benign (e.g., a potion can still be thrown at empty space, if the player is not lined up with the monster), attempts at actions that lack the corresponding keypress(es) are functionally impossible. This large number of impossible actions must be dealt with to avoid very long training times and unnecessary computation that could otherwise occur if impossible actions were given negative rewards.

Two changes to the regular Q-learning algorithm are made to address this issue. First, we define a prerequisite function for each action; this function takes in the current game state and outputs a boolean indicating if the action is possible to take in that state or not. With this set of functions, we can at all times narrow the action set to that of the possible actions. In the behaviour policy (epsilon-greedy), we then assign zero probability to all impossible actions. Secondly, in the Bellman update equation, when the max over the next state is taken ($\max_a Q(s_{t+1}, a)$), we limit the max to be over the set of possible actions instead of all actions. These two modifications eliminate the problem of impossible actions.

When using experience replay, the (state, action, reward, next state) tuple must also be augmented by the list of possible actions, so that the action-values of batch updates can be similarly affected.

## Experiments

Experiments were conducted with the player in arena-style, one-on-one combat against a slice of twelve monsters from levels 14 to 17 (with some exclusions). The twelve monsters at these levels have unique abilities and require a diverse list of strategies to defeat. Some monsters curse items in the player's inventory or cause their weapon to deteriorate while others have powerful melee attacks. A subset of monsters was chosen in place of the entire monster set in order to lower the computation time needed to learn a correct model, although it is likely that the agent would perform on average equally well on the entire set if trained on them (we do not report it here, but other (lower-level) ranges were also tried, with similar or better results).

Some monsters were excluded, even though they technically can appear in our level range: unique and non-randomly generated ones (like shopkeepers or the Wizard of Yendor) which require specialized strategies (10 possible), as well as shape-changers (2 possible); the latter since monster type is expected to stay constant in an episode.

In each episode, a random monster is chosen and placed together with the player in opposing corners of a large (8x16) rectangular room, as seen in figure 1. The large initial distance between player and monster allows for the player to have several turns to get ready before encountering the monster (e.g., equipping weapons and/or armor); it also gives the player a few turns to perform ranged attacks before the monster enters melee range. An episode terminates on player or monster death, or after 200 actions (whichever occurs first). 200 is an excessive number of actions, but was shown to perform better than a lower number in experiments.

The player is given a random sample of items to mimic the inventory of a real player facing against a monster midway

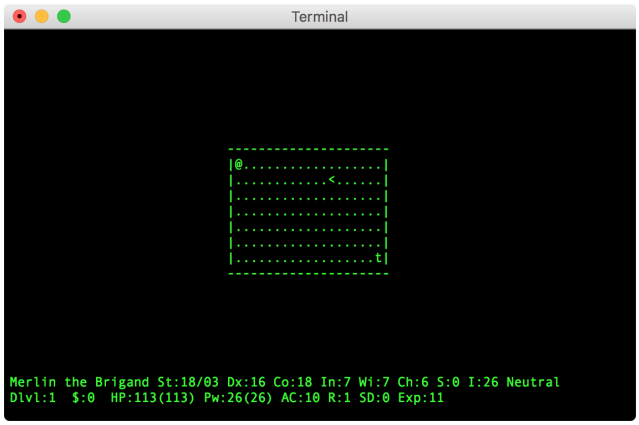


Figure 1: The room used for the learning environment. The player ('@' character) and monster ('t' character) start in opposing corners. The bottom two lines describe the player's current attributes and statistics.

through the game. This sample includes one melee weapon each per most material types (wood, iron, silver and metal), one ranged weapon with ammunition, one random scroll, one random potion, one random wand, and five random pieces of armor; in total, the player is given 13 items out of a total 186. All items are generated as being uncursed and having no enchantment (+0). Items which require special input behaviour (e.g., wand of wishing) are also excluded, as well as unique 'artifact' items which are typically over-powered.

The player's character/role is always set to 'barbarian' (out of the 13 roles available) – role determines the initial attributes of the player (e.g., strength or intelligence), as well as some special initial properties. The barbarian was chosen for its ability to handle both melee and ranged weapons reasonably well. Spellcasting is not used in our experiments so the barbarian's low intelligence is not a contraindication.

## Experimental Results

We present results of two models, one only given weapons and the other outfitted with the full gamut of items, and compare them with the corresponding baseline algorithms.

Each model is first trained on the range of selected monsters from levels 14 to 17, then tested on the same range. In each episode, a random inventory (according to the above rules) is generated, so no two episodes are likely to be the same. Further, randomness exists in attack chance, damage, and outcomes of some item effects.

### Model hyperparameters

In each experiment, player level is determined by the monster's difficulty as described below. Player level determines starting health and chance to hit in combat, as well as other effects that do not influence our combat environment.

For the weapons-only scenario, the player level is two levels higher than the monster difficulty, which allows some player survivability and reflection of correct weapon choice in results. For the full items case, it is set to three lower than

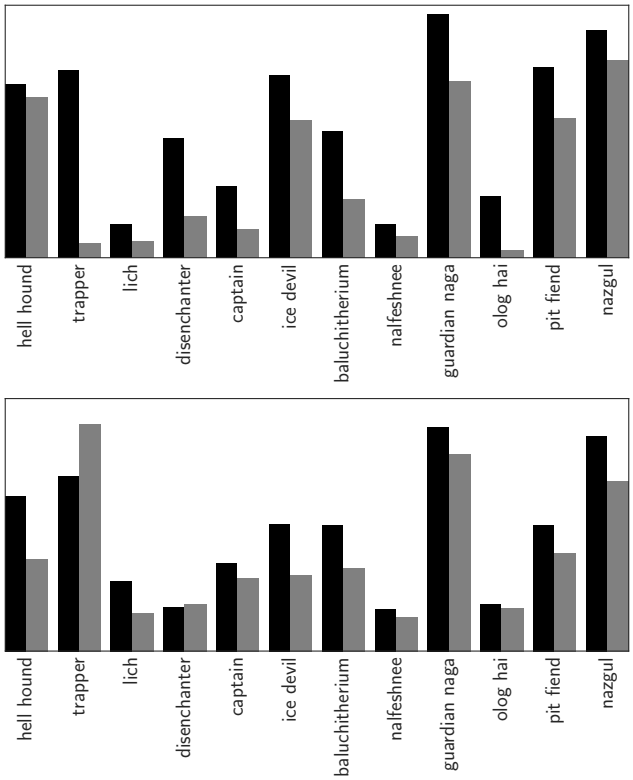


Figure 2: Success rates of the DQN (solid black) vs. baseline (grey) on selected monsters from levels 14 to 17. Vertical scale shows percentage of successful combat encounters from 0-100%. Models with weapons only are on top; full items on bottom.

the monster difficulty, since the armor increases survivability in general. Other parameterizations are of course possible.

Each of the two DQN models use an epsilon-greedy behaviour policy with epsilon linearly annealed from 1 to 0 through the course of training (with a value of 0 used for evaluation). The weapons-only model is trained over 1 million actions while the full items model is trained over 2 million. Each use an experience replay buffer of the most recent 1 million actions. Discount rate was 0.9.

The architecture of the neural network is as follows: an input layer of size equal to state length, followed by a dense layer of 128 units, a dense layer of 64 units, and an output layer of size equal to the number of actions. Adam was used for the optimizer with a learning rate of 0.001. The *keras-rl* (Plappert 2016) library was used for the implementation.

## Weapons only

The results for the models that only consider weapons are presented in figure 2 (top). As seen in the figure, the DQN does better against all monsters than the baseline, with the difference on some monsters (like the trapper or disenchanter) being more pronounced than others. The model does worst on the lich and nalfeshnee, monsters that can curse items in the player's inventory rendering them ineffec-

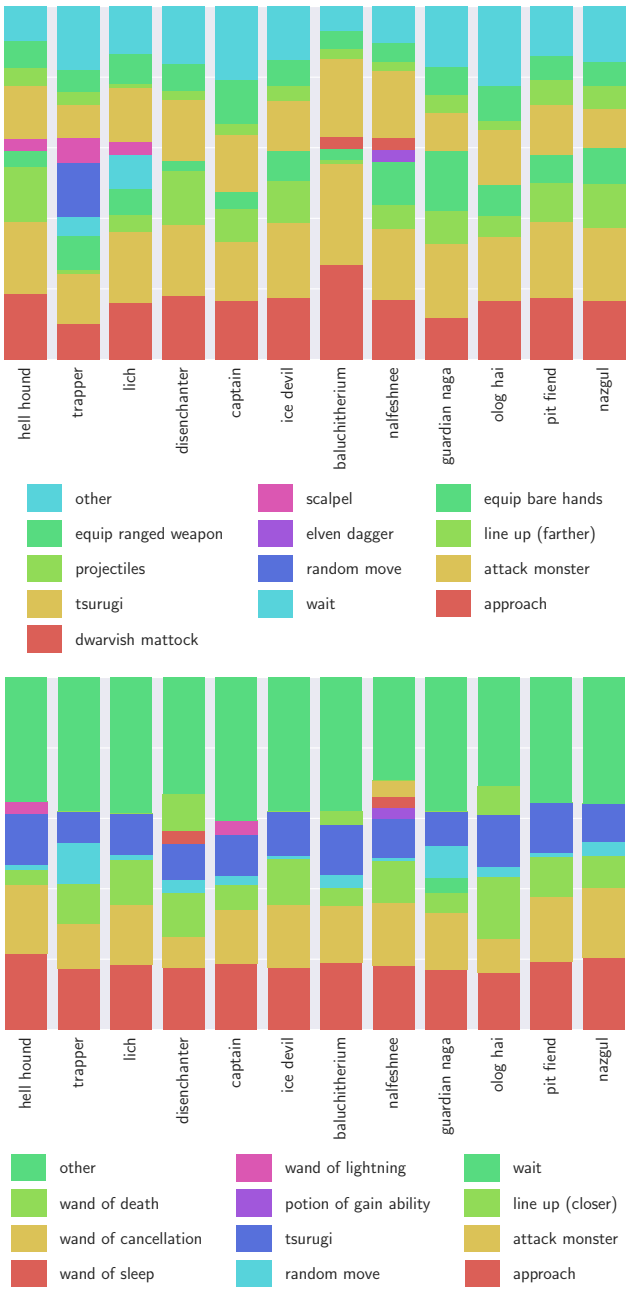


Figure 3: Actions taken by the DQN models in successful episodes on monsters from levels 14 to 17 (weapons-only model on top, full items on bottom). Vertical scale shows percentage of taken actions from 0-100%. Each action is counted once per episode; actions that in total make up less than 3% of all chosen actions and less than 3% of actions taken against an individual monster are grouped together in the 'other' category. Throwing any type of projectile was grouped into one category as well as equipping any type of ranged weapon. All the items present here are +0 and uncursed, omitted for brevity.

tive. Meanwhile, monsters like the guardian naga or Nazgul have powerful paralyzing attacks (whose effects would only show up after the combat has ended) but are otherwise easily defeated by any attack.

To get a better sense of what items the model prefers to use for each monster, the agent's actions per monster are summarized in figure 3 (top). Each action is counted once per episode (de-emphasizing repeated attacks, but allowing us to verify that singular actions like equipping are performed). The tsurugi is equipped in a disproportionate number of episodes simply because it is always present in half of all episodes (since there are only two metal weapons in the game). Further, it is preferred over the other metal weapon (scalpel) as well as all other weapons due to its high damage output and its use of the two-handed sword skill, which is one of the two random starting skills for the barbarian.

We see that the approach action was used in nearly every episode, while the lining up action was also taken—this action is in many instances the same as approach, but in some cases also lines up with the monster to allow for a ranged attack or wand zap/potion throw. The agent does not seem to distinguish much between approaching or lining up, with both present in the majority of all monster episodes.

Random moves are used often against the trapper, which has a special attack that envelops the player. When enveloped, a move in any direction is considered an attack.

Regarding number of actions taken per game, the weapons-only baseline takes about 40 actions on average to resolve a combat (i.e., either player or monster dying), while the DQN model takes about 55 actions. The DQN model sometimes takes wasteful actions (e.g., cycling through a few weapons) before approaching the monster, possibly caused by insufficient learning time.

### All items

Results for the models that can use the full set of items are presented in figure 2 (bottom). The difference between the DQN and baseline is less pronounced here, and generally results are the same or lower than the weapons-only DQN model. Even though we doubled learning time (to 10 hours), this did not fully compensate for the greater number of actions. Baseline performance was also worse than the weapons-only baseline, which is easily explained since a certain number of the items used by the baseline can have adverse effects (e.g., a scroll that destroys a piece of the player's armor). It is also possible that, for harder monsters like the lich, the models are hitting a barrier determined by their inventory, and the best strategy may simply be avoidance or retreat. One outlier in the difference between the two baselines is the trapper, on which the all-items baseline excels; this difference could be attributed to the armor that the baseline equips, which allows the player to survive long enough when engulfed by the trapper to escape.

Actions taken by the DQN model (with each action counted once per episode) are shown in figure 3 (bottom). Here again the tsurugi was the most favored weapon. Only one potion, that of gain ability, is used with any frequency (it has a chance to increase the player's success in combat). Scrolls are not used often since the majority are ineffective in combat or have limited effects. A few wands are used often: lightning (deals damage), sleep (causes the monster to stop moving), cancellation (removes a monster's special abilities), and death (kills a monster immediately). The wands were used most on the nalfeshnee and disenchanter, two monsters with powerful special abilities (item-cursing and weapon-disenchanting). We would expect the same wands to be used against the lich, but they are not. The lich in fact has a higher success rate than either the nalfeshnee or disenchanter, suggesting that different, more varied strategies (in the 'other' category) are used to defeat it.

No ranged weapon/projectile usage breaks through the 3% barrier against any monster here, which may be attributable to the availability of wands, of which many are more powerful than ranged weapons.

The larger 'other' section here compared to the weapons-only model is due to the much higher number of total possible items present in the agent's inventory (186 vs. 46); in many episodes few of the most useful items will be available, so suboptimal items would be used instead.

## Conclusions and Future Work

Combat in games like roguelikes is a complex task and a roadblock in developing successful automated players. Machine learning can alleviate the need for hard-coded heuristics or rules for a bot to follow. In this paper we described such an approach to a one-on-one combat environment in the roguelike NetHack. Our design, which uses a Deep Q-Network, employs a high level of abstraction over the NetHack game state and action set to reduce the complexity involved in the game. Our model outperforms easier to implement baseline combat strategies, trading long, but automated learning costs for better performance.

An ideal next step for this work would be to integrate it with an automated player for the entire game of NetHack, possibly in conjunction with other, non-learning approaches for other game issues (such as exploration). In terms of improving the model itself, running experiments on a more exhaustive NetHack environment would further improve generality, and it would be interesting to consider more complex room configurations (including more dungeon features like fountains and stair-access) as well as investigating multi-combat scenarios, where resource management and prolonged status effects come into play. Addressing greater complexity of course requires larger training times.

Generality can also be explored by applying our approach to other roguelike games. We are also interested in direct comparison with the BotHack player. This may provide a more focused upper bound for evaluating performance, although its hard-coded combat strategy is not well tuned to our limited, single-room test environment, and it would also require significant effort to integrate our customizations and environment with BotHack's Clojure-based code.

## Acknowledgements

This work was supported by the Natural Science and Engineering Research Council of Canada.

## References


Heinrich, J., and Silver, D. 2016. Deep reinforcement learning from self-play in imperfect-information games. In *NIPS Deep Reinforcement Learning Workshop*.

Hintjens, P. 2011. 0MQ - the guide. `http://zguide.zeromq.org/page:all`.

Kempka, M.; Wydmuch, M.; Runc, G.; Toczek, J.; and Jaskowski, W. 2016. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games, CIG 2016, Santorini, Greece, September 20-23, 2016*, 1–8.

Krajicek, J. 2015a. BotHack - a Nethack bot framework. `https://github.com/krajj7/BotHack`.

Krajicek, J. 2015b. Framework for the implementation of bots for the game NetHack. Master's thesis, Charles University in Prague.

Lample, G., and Chaplot, D. S. 2017. Playing FPS games with deep reinforcement learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 2140–2146.

Liebana, D. P.; Samothrakis, S.; Togelius, J.; Schaul, T.; and Lucas, S. M. 2016. General video game AI: Competition, challenges and opportunities. In Schuurmans, D., and Wellman, M. P., eds., *AAAI Conference on Artificial Intelligence*, 4335–4337. AAAI Press.

Mauldin, M. K.; Jacobson, G.; Appel, A.; and Hamey, L. 1984. ROG-O-MATIC: A belligerent expert system. In *Proceedings of the Fifth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, volume 5.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

NAO public NetHack server. 2017. NetHack – top types of death. `https://alt.org/nethack/topdeaths.html`.

Narasimhan, K.; Kulkarni, T. D.; and Barzilay, R. 2015. Language understanding for text-based games using deep reinforcement learning. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, 1–11.

NetHack Wiki. 2017. Black pudding — NetHack wiki. `https://nethackwiki.com/wiki/Black_pudding`.

Plappert, M. 2016. keras-rl. `https://github.com/matthiasplappert/keras-rl`.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529:484–503.

Sutherland, K. 2017. Playing roguelikes when you can't see — Rock, Paper, Shotgun. `https://www.rockpapershotgun.com/2017/04/05/playing-roguelikes-when-you-cant-see`.

Sygnowski, J., and Michalewski, H. 2017. Learning from the memory of Atari 2600. In Cazenave, T.; Winands, M. H.; Edelkamp, S.; Schiffel, S.; Thielscher, M.; and Togelius, J., eds., *Computer Games: 5th Workshop on Computer Games, CGW 2016, and 5th Workshop on General Intelligence in Game-Playing Agents, GIGA 2016, Held in Conjunction with the 25th International Conference on Artificial Intelligence, IJCAI 2016, New York, USA, July 9-10, 2016, Revised Selected Papers*, 71–85. Cham: Springer International Publishing.

Uriarte, A., and Ontañón, S. 2014. Game-tree search over high-level game states in RTS games. In *Artificial Intelligence and Interactive Digital Entertainment*.

Wang, Z.; Schaul, T.; Hessel, M.; Van Hasselt, H.; Lanctot, M.; and De Freitas, N. 2016. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, 1995–2003. JMLR.

Zhuang, Y.; Li, S.; Peters, T. V.; and Zhang, C. 2015. Improving Monte-Carlo tree search for dots-and-boxes with a novel board representation and artificial neural networks. In *IEEE Conference on Computational Intelligence and Games, CIG 2015*, 314–321.