QUIXOTE: A NETHACK REINFORCEMENT LEARNING FRAMEWORK AND AGENT

Chandler M. Watson
Department of Mathematics
Stanford University
watsonc@stanford.edu

June 13, 2019

1 Introduction

In the 1980s, Michael Toy and Glenn Wichman released a game for Unix that would change the ecosystem of Unix games (and later many others) forever. *Rogue*–their invention–became the quintessential *dungeon crawler*: a game in which the objective is to explore a dungeon, often in hunt of some hidden treasure or artifact. The release of *Rogue* captured the imaginations of many college students who subsequently created a new genre of games, appropriately known as *roguelikes*, which tried to recreate the magic of the original *Rogue* by closely reimplementing battle, and exploration, and UI elements.

While *Rogue* remained proprietary software for a time, many rougelikes became open source software. 1982 and 1983 saw the birth of *Hack* and then *Moria*, who carried on the legacy of *Rogue* and eventually yielded many other spinoffs of their own. One of note which remains in active development today is *NetHack*, a cross-platform rougelike with a Dungeons and Dragons-esque player stat scheme which holds true to the original story and absurd difficulty of the original *Rogue*.

With such difficulty comes an interesting side-effect, however: communities of programmers joined together to build and test bots on public NetHack servers, exploring every nook and cranny of the game at massive parallel speed. Yet it was only recently, in 2015 (almost 30 years after the first release of NetHack!) that a bot was designed which could complete the game in its entirety. The bot, known as BotHack, is written entirely in Clojure and uses a large deal of rule-based AI to win the game.

In this project, we seek to, in the light of BotHack's victory, apply methods requiring less expert knowledge in order to succeed in the game. To do this, we approach NetHack from a reinforcement learning (RL) perspective. Unlike many other typical games RL is applied to, NetHack has significantly more complex items and actions (using nearly the entire keyboard and then additional text commands!) and varies greatly from playthrough to playthrough by using procedural map generation, making it an exciting target for machine learning. While this project won't make full use of NetHack's library of nuances, the aim of this project is to create a Python test harness for NetHack and test variations on Q-learning on it in an attempt to characterize the game from a (shallow) RL perspective.

Specifically, we implement a framework, Quixote, as a harness for the NetHack executable, implement a simple Q-learning algorithm choosing the discount factor by validating over 20 episodes (which are fairly long, taking on the order of thousands of actions and rate-limited by NetHack's screen refresh rate), implement linear function approximation for Q-learning as well, and then test a gradual ϵ-annealing strategy.

2 Related Work

The earliest related work is Mauldin et al's Rog-O-Matic: an AI system devoted to winning the original Rogue. This was an expert system published in 1984, which managed a complex internal state and contained ten separate internal experts for subtasks like performing magic, exploring the map, managing armor, etc. It was written in 12,000 lines of C code which encoded an unmatched amount of expert knowledge, and is the earliest example of an AI achieving expert status on a rougelike (namely Rogue). While the primary mechanism Rog-O-Matic worked on was simple expert rules, there was some simple learning that needed to be performed, namely for identifying and characterizing monsters in combat.

Much later, as interest in NetHack grew, a number of smaller projects shot up to try to conquer NetHack in the same way. In 2005, a C# bot known as nHackBot which was able to perform fairly advanced behaviors (and survive to the 10th floor, a remarkable achievement) inspired many other bots. One of the earlier ones was the Tactical Amulet Extraction Bot (TAEB). It was a joint project between members of the #interhack channel on Freenode's Internet Relay Chat servers, and quickly became

preeminent among NetHack bots. In 2015, a landmark event occurred: a bot known as BotHack "ascended," or completed the game in its entirety without dying (NetHack resets the game when the player dies and generates a new map). Following TAEB, BotHack also focused on creating a framework in which bots could be created, and embedded a massive amount of explicit expert knowledge to do so.

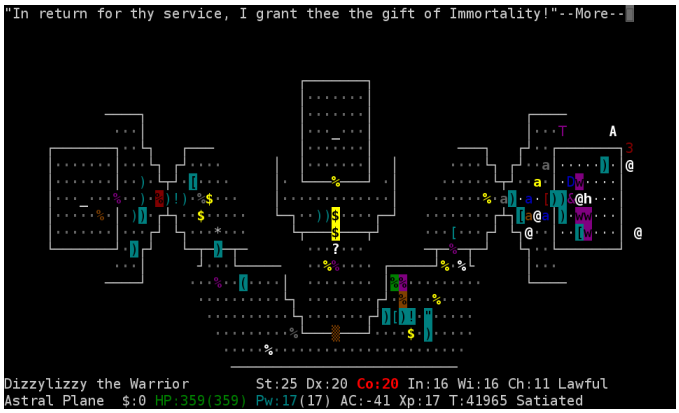


Figure 1: The holy grail of *NetHack*: "ascension" at the end of the game

While elements of traditional AI–such as knowledge bases, expert systems, path planning algorithms, etc.–are evident in mainstream NetHack bot creation, much of machine learning seemed to be missing from the scene entirely. Work specifically on NetHack for machine learning is incredibly sparse. The only paper this author could find was a Jonathan Campbell and Clark Verbrugge's 2017 work on learning NetHack combat. In it, the authors use a Deep Q-Network (specifically a dueling double deep Q-Network) with a strong positive reward given on monster death and small negative rewards for player death and "cost of living." The state is carefully constructed from data extracted from the screen, including what the player is wearing/wielding, what monster is being fought, the inventory of the player, etc., and a wide range of potential actions is allowed.

In terms of general work, the majority has focused on Rogue, and specifically deep reinforcement learning in Rogue. Work by Asperti et al. in 2017 focuses on the creation of a new framework and the use of deep Q-learning (partitioned) for a general-purpose Rogue AI. The same team later adapted their code to use (partitioned) A3C for general Rogue and ACER as a deep RL baseline.

3 Dataset and Features

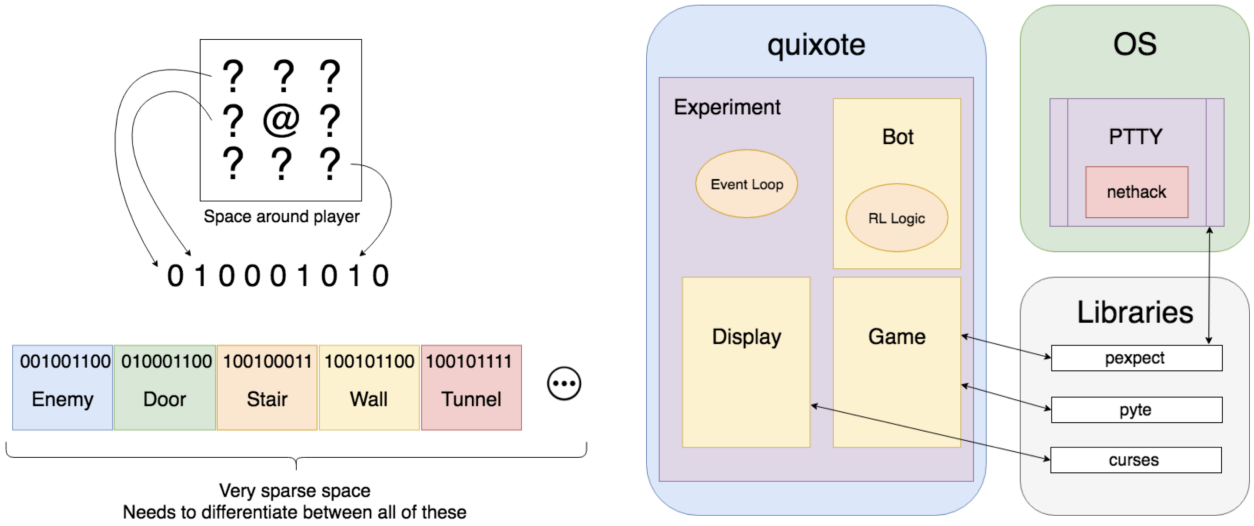
In this section, we discuss in lieu of a "dataset" our use of NetHack environment and, more specifically, how we harness it for reinforcement learning. NetHack itself has far too complex of a state and action space to be handled in its entirety for a single term project, so instead of focusing on every game mechanic, we follow Asperti et al. in their original paper and focus entirely on exploration of the map. As such, we limit our agent to ten actions: movement in each of the cardinal directions, all four diagonals, and then an up and down action to ascend/descend staircases.

The state itself was chosen to be simple in principle while allowing the agent to know the details of whats in it's immediate surroundings. For each potential object of interest–in our case, enemies, doors, stairs, walls, tunnels, etc.–we use a binary vector with one entry for each cell in the 3x3 grid surrounding the player to represent the presence of that object. Next, we have another binary vector representing which of those cells have been visited by the player before. These binary vectors are concatenated into a final 72 bit state.

For extracting these states, we develop a new framework named Quixote, which runs an instance of NetHack and allows full interactivity from a Python interface. It does this by using a library called `pexpect`, which allows for the running and control of executables in *psuedoterminals*, a Unix construct that acts like a terminal but only exists in kernel memory. The heart of Quixote, the `Game` object, uses `pexpect` to run NetHack, send commands to it, and get back terminal output. The terminal output itself contains ANSI escape codes for moving the cursor, however, so an in-memory terminal emulator supplied by `pyte` is used to parse these and return the screen of the terminal.

The `Display` object provides an abstraction for a `curses` interface, which allows one to draw on the terminal screen like a grid and rewrite elements of that grid. The various `Bot` classes implement various reinforcement learning strategies, taking in representations of the state output by a `Game` and outputting chosen moves. `Experiments` simply wrap all of these objects together in an event loop, getting a bot action, passing it to the game, getting the new game state, passing it to the display and the bot, and repeating. `Games` represent the NetHack state at the lowest level, interpreting the heads-up display at the bottom of the screen and capturing the map but leaving all interpretation to the relevant `Bot`.

Lastly, we give brief, important properties about NetHack itself. First, NetHack's map and initial state is randomly generated for each run, making it difficult for traditional RL algorithms which work well on fixed environments. Additionally, once the player



(a) A representation of the current NetHack state.

(b) The Quixote framework as a whole.

Figure 2: Illustrations of Quixote internals.

dies once, they have to start over with a new random map, meaning that one game the player could be extremely lucky and the next they could be extremely unlucky. At the end of each game, the player is given a score, which contains a great number of factors. Fortunately, since our bot only can explore, this simplifies to

$$S = 50 \cdot (\text{deepest level reached}) + \max\{1000 \cdot (\#\text{ of levels reached below 20}), 10000\} + \sum_{M \in \text{monsters defeated}} 4 \cdot (\text{experience of M})$$

meaning that rewards should be given for descent and for defeating monsters.

4 Methods

For this project, we focus strictly on shallow reinforcement learning techniques to characterize the difficulty of exploration in NetHack. To do this, we perform three experiments: one with training a simple vanilla Q-learning model for score maximization, one with a Q-learning model which uses linear function approximation, and one which uses the previous two models, but with a scheduling mechanism for epsilon, where epsilon is the probability that a random action will be chosen instead of the action which follows the greedy policy.

For the vanilla Q-learning experiment, we simply construct a Q-learning agent for the above state which uses the standard Q-learning update equation for a state-action-reward-state tuple (s, a, r, s'):

$$Q(s,a) := (1-\alpha) \cdot Q(s,a) + \alpha \cdot \left(r + \gamma \cdot \max_{a' \in \mathcal{A}} Q(s', a') \right) \tag{1}$$

where α is the learning rate and γ is the discount factor. We choose $\alpha = 0.2$, and use cross-validation by mean NetHack score over 20 episodes to determine the value of $\gamma \in \{0.2, 0.4, 0.6, 0.8\}$. The learning curves empirically appeared to have no "ramping up" phase, as the episodes are sufficiently long, so we use the score for all 20 episodes in the mean. Initially, some α_0 was to be chosen and then (s, a)-specific action rates were to be chosen as

$$\alpha_{s,a} = \frac{\alpha_0}{\#\text{ of times } (s, a, r, s') \text{ observed for any } r, s'} \tag{2}$$

as recommended in the original Q-learning paper and following Yizheng Liao, Kun Yi, and Zhe Yang's 2012 CS229 projection on RL for Mario to guarantee convergence, but this led to learning ceasing too quickly. Rather than tweak the decay rate, we opt here for a constant rate chosen by what empirically caused learning by observation.

For rewards, at each timestep we give a -0.1 "cost of living" reward to ensure the model keeps moving toward the next positive reward, a -0.5 reward every time a previously visited position is revisited, a $+5$ reward for making the map (not including the player) change, a reward equal to the number of points gained, and $+50$ points for each level descended.[1]

For the second experiment, we use linear function approximation in conjunction with Q-learning to ensure that our model is able to generalize well between states. For this, we instead calculate Q as

$$Q(s,a) = \theta^\top f(s,a) \tag{3}$$

for a vector-valued state function $f(s,a)$, and use an update rule for (s,a,r,s') of

$$\theta := \theta + \alpha \left(R(s,a) + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a) \right) \cdot f(s,a) \tag{4}$$

as specified by Alan Fern in his RL for Large State Spaces slides. For a state function, we partition state vector entries according to action as such, following the advice of Daniel Seita in his blog post on LFA for Q-learning:

$$\begin{aligned} f(s,a) = 1\{a = a_0\} \cdot [s_1, \cdots s_n, 0, \cdots 0, 0, \cdots 0, \cdots]^\top \\ + 1\{a = a_1\} \cdot [0, \cdots 0, s_1, \cdots s_n, 0, \cdots 0, \cdots]^\top \\ + 1\{a = a_2\} \cdot [0, \cdots 0, 0, \cdots 0, s_1, \cdots s_n, \cdots]^\top \cdots \end{aligned}$$

By doing so, we allow generalization between similar states, while giving the extra dimensionality necessary for the model to handle each action disjointly. We use the same hyperparameters as the previous experiment, and continue using an epsilon-greedy policy.

Lastly, for the third experiment, we use our domain knowledge that random agents tend to perform relatively well in this area, and are able to explore the map quickly. By initially setting $\varepsilon = 1$ and then slowly decreasing ϵ over the subsequent episodes, we give our agent the necessary "burn-in" time to explore randomly and learn the start to a good policy, then slowly begin exploiting as time goes on. The scheduling is performed by lowering ε by 0.2 every 10 episodes, and we apply this schedule to both of the previous models.

5 Results

Each of the above models was run for 50 episodes, and then the mean score was reported. The results are as follows:

Model	Random	QL	QL + schedule	LFA QL	LFA QL + schedule
Mean score	45.60	44.92	51.96	32.53*	49.52

Figure 3: Mean scores over 50 episodes for each model. QL is Q-learning, and LFA is linear function approximation.
*LFA QL was run for 30 iterations by time constraint.

Plots are shown below in Figure 4 depicting the score for each algorithm over time in a 50 episode run, taking on the order of 1-2 hours to complete. The metric used is the mean NetHack end-of-game score over 50 episodes.

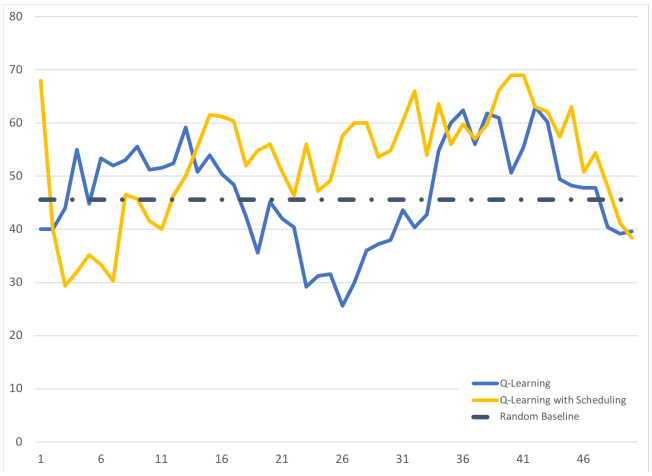
6 Discussion

While it may look like Q-learning with scheduling dominates Q-learning and the random baseline, multiple runs of the 50 episodes (as long as they are–for some algorithms this takes well north of an hour to complete running at full speed) qualitatively suggests that the variance in scores is sufficiently high as to not warrant a statistical test for significance (such as a simple Student's t-test). Indeed, a previous partial result for LFA Q-learning had a mean score higher than that of Q-learning with scheduling–simply rerunning the 50 episodes yielded different luck, both possibly on the part of NetHack's random initialization but also very possibly on how the model learned from a different initialization.

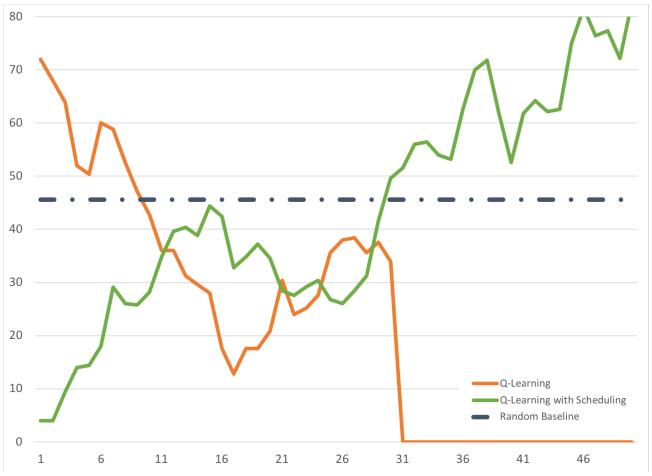
Thus, the first thing that should happen (with additional time) would be running these algorithms for as many episodes as would be feasible for testing. Experimenting with slow annealing of $(s,a)-$specific learning rates and monitoring Q-learning/LFA Q-learning updates could allow for automatic determination of convergence, rather than simply running the model for as long as time would allow.

After that, exploration of different state representations would be worthwhile. One major issue–observed with the viewing mode built into Quixote–is that the current policy formulation relies on penalizing the model for visiting the same space multiple times. While this is useful to force the model to explore other areas, it often paints the agent into a corner, discouraging the exploration

[1]Technical note: for determining whether the map has changed, and for determining the state vector at each timestep, we keep track of what is beneath the player by caching a previous version of the map and combining maps after each action.



(a) QL with (yellow) and without (blue) scheduling.



(b) LFA QL with (green) and without (red) scheduling.

Figure 4: Plots of smoothed (running average over 10) model performance over episodes.

of other areas as well as any form of backtracking, and is otherwise fairly fragile, often causing the agent to get stuck oscillating between states while stuck. Nonetheless, at other times it can be visually verified that it encourages fast linear exploration of the map.

Another issue is that of how Q is modeled. It's hard to determine (especially with the extreme variance in initial states of the environment) whether or not the model was able to properly generalize between states from linear function approximation or not. Most of the literature (specifically from Asperti et al.) focuses on deep architectures for reinforcement learning in Rogue, indicating that a larger state space with a more complex representation of Q is very possibly necessary.

While no statistically significant conclusions can be made from the data, the plots above seem to indicate that while both Q-learning and LFA Q-Learning seemed to waver about the baseline while learning, there is an upward trend to both ε-scheduled models, suggesting the efficacy of slowly annealing from a random strategy in this domain for exploration. Further tests (i.e. for more epsiodes) in this direction would be useful.

The primary issue each model had, from visual inspection, appeared to be that it couldn't properly handle states where the choice of action was non-obvious from immediate surroundings. If an enemy was near the agent, it often attacked it and won, yet when simply exploring it often wound in circles and explored very inefficiently. Thus, one potential fix could be a "meta-action" model in which higher-level actions (attack monster, move to door) are performed and executed by "traditional AI" like path-planning algorithms, which collect rewards and feed them back to the RL algorithm.

7 Conclusion

While it isn't definitive how the models in this paper performed due to the highly variable starting conditions of a NetHack game, a few things are clear. First: the Quixote framework for NetHack RL provides a viable framework for building RL models for the game NetHack. Second, while the RL strategies given here do appear to struggle at first, they do appear to at times explore the level well qualitatively, and annealing ϵ from 1 downward seems to be an effective strategy for achieving sufficient exploration in early episodes.

The code for this project is available at `https://github.com/watsoncm/quixote`.

8 Contribution

I (the sole author) developed the full Quixote framework, implemented and tested all the models involved, and wrote the paper and designed the poster.

9 References

Asperti, A., Cortesi, D., De Pieri, C., Pedrini, G. & Sovrano, F. Deep reinforcement learning & rogue. Retrieved from http://www.cs.unibo.it/ asperti/rogue.html

Asperti, A., De Pieri, C., & Pedrini, G. (2017). Rogueinabox: An environment for roguelike learning. International Journal of Computers, Retrieved from https://www.iaras.org/iaras/filedownloads/ijc/2017/006-0022(2017).pdf

Campbell, J., & Verbrugge, C. (2017). Learning combat in NetHack. AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 13. Retrieved from http://gram.cs.mcgill.ca/papers/campbell-17-learning.pdf

Fern, A. RL for large state spaces: Value function approximation. Retrieved from https://oregonstate.instructure.com/files/67025084/download

Krajj7/BotHack: BotHack - A nethack bot framework. Retrieved from https://github.com/krajj7/BotHack

Liao, Y., Yi, K. & Yang, Z. (2012). CS229 final report: Reinforcement learning to play mario. Retrieved from http://cs229.stanford.edu/proj2012/LiaoYiYang-RLtoPlayMario.pdf

Mauldin, M. L., Jacobson, G., Appel, A. & Hamey, L. (1984). ROG-O-MATIC: A belligerent expert system. Retrieved from https://www.cs.princeton.edu/ appel/papers/rogomatic.html

Moore, S. M.The tactical amulet extraction bot (blog). Retrieved from http://taeb-nethack.blogspot.com/

Pexpect/pexpect: A python module for controlling interactive programs in a psuedo-terminal. Retrieved from https://github.com/pexpect/pexpect

Rogue (video game). Retrieved from https://en.wikipedia.org/wiki/Rogue_(video_game)

Score - NetHack wiki. Retrieved from https://nethackwiki.com/wiki/Score

Seita, D. (2016). Going deeper into reinforcement learning: Understanding Q-learning and linear function approximation. Retrieved from https://danieltakeshi.github.io/2016/10/31/going-deeper-into-reinforcement-learning-understanding-q-learning-and-linear-function-approximation/

Selectel/pyte: simple VTXXX-compatible linux terminal emulator. Retrieved from https://github.com/selectel/pyte

The tactical amulet extraction bot. (2007). Retrieved from http://taeb.github.io/index.html

TAEB/TAEB: The tactical amulet extraction bot (for NetHack). Retrieved from https://github.com/TAEB/TAEB

Watkins, Christopher J. C. H, & Dayan, P. (1992). Q-learning. Machine Learning, 8(3-4), 279-292. doi:10.1007/BF00992698