CS230

Hacking NetHack: Novel Reinforcement Learning Architectures for Multi-Objective Optimization

Ayush Pandit

Stanford University
apandit@stanford.edu

Eduardo Esteban

Stanford University
eduardoe@stanford.edu

Abstract

Despite significant progress in the field, designing reinforcement learning algorithms for stochastic, high dimensional, and reward sparse environments remains difficult and computationally expensive. The roguelike game NetHack provides an opportunity to explore these open problems with minimal resources for simulation, and has recently been packaged as a model system for reinforcement learning tasks. In our project, we worked to extend the capabilities of the NetHack Learning Environment and investigated the learning efficiency of deep reinforcement learning algorithms. Although we found some strategies for improving learning to be promising, high variability across trials prevented us from reaching conclusive results. We have released our code and extensions to the NetHack Learning Environment at https://github.com/apandit42/cs230-deep-hack

1 Introduction

Over the past decade, the field of reinforcement learning has seen incredible breakthroughs. Deep learning systems have defeated the best human players in everything from Go to DOTA 2 and Starcraft II (Shao et al.). Driven by the exponential growth of computational power and improved neural network architecture design, these advancements have highlighted reinforcement learning's massive potential.

However, despite these specific successes, some key challenges in this field remain unaddressed. General purpose architectures and agents that are capable of navigating multi-objective, stochastic, and sparse-reward environments remain difficult to construct, and often even the best methods struggle to achieve results that can match let alone surpass human performance (Kiran et al.). Furthermore, model environments with these constraints are often computationally expensive to run, making exploration and research progress difficult.

The video game NetHack offers a surprising opportunity to investigate some of these challenges. First released in 1987, NetHack is a roguelike game where players journey through a multi-layer procedurally generated dungeon, discovering items and engaging in battles with various enemies, to find and retrieve the Amulet of Yendor (Küttler et al.). A complex, stochastic game with many different subsystems, NetHack also offers the advantage of requiring little computational power to run (Grefenstette et al.). A game that is difficult for even skilled human players to master, it provides a unique opportunity to push the boundaries of state-of-the-art deep reinforcement learning. Because of these advantages, NetHack has recently been explored as a model environment for reinforcement learning. For an algorithm to succeed in NetHack, it must successfully select chains of actions for tens of thousands of steps, while navigating with other challenges throughout the dungeons, only receiving

CS230: Deep Learning, Fall 2020, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

state and score information from the game. We focused on investigating the learning efficiency of algorithms early in the game, within the first several thousand steps a player takes, which can greatly impact the rest of the training process for an algorithm.

2 Related work

For our project, we rely heavily on the work completed by Küttler et. al. developing and distributing the NetHack Learning Environment (NLE) (2021). A Python library that serves as a direct interface to an underlying NetHack game instance, NLE offers an Open-AI Gym API and several predefined built-in tasks for users to attempt. In addition to the direct work completed by the authors themselves, we have also drawn inspiration from others who have used NLE, such as Zhang et al (2020). who used NetHack as one of their model systems for developing BeBold, an improved reinforcement learning algorithm for efficient exploration. These previous approaches helped us identify potential areas for improvement, including early learning efficiency, and addressing high variability in training regimes.

Beyond work that focused on NLE, we also found several closely related projects investigating roguelike and other procedurally generated games as platforms for reinforcement learning research. Also motivated by the same needs in reinforcement learning research, Samvelyan et. al. took a very different approach to building a NetHack based model system (2021). Designing their system to be highly modular, they focused on developing and enabling subtasks for game systems, rather than building clear objectives into the environment as with the NetHack team. Likewise, there have been several examples of researchers advocating for the general application of procedurally generated systems to reinforcement learning research and benchmarking reinforcement learning algorithms (Cobbe et al., 2020), and others have adapted other, similar roguelike systems as reinforcement learning systems such as the popular roguelike Dungeon Crawl Stone Soup (Dannenhauer et al.). Overall, though we decided to proceed with NLE, seeing the different approaches each group took to their efforts helped us identify the areas we were most interested in investigating, and gave us our starting algorithmic designs.

3 Dataset, Features, and Environment

Though reinforcement learning algorithms do not depend on the supervised datasets traditionally used for machine learning tasks, they do need to be trained over thousands of iterations of simulated interaction with their environment, and have their own data management requirements. Before beginning experiments, we randomly sampled 20,000 seed values, 19,000 of which were used for training algorithms, and 1,000 of which were held out to test the generalizability of our learned parameters, a protocol taken from the literature. Additionally, instead of training on an episode based regime, our system was designed to run for 1 million steps before quitting, reflecting our goal of optimizing early game exploration, strategies, and behaviors, as well as the practical limitations of computational power we face, especially compared to the 1.2 billion iterations the developers of NLE at Facebook ran their system for.

Unlike the original tasks included with NLE, which randomly generate characters for each simulation unless told not to do so, we restricted our exploration to two character builds, motivated by both the success and obstacles they had shown in the NetHack Learning Environment developer team's benchmarking. Focusing on lawful dwarven Valkyries and chaotic elven wizards, we conducted each experiment and training process separately, which should have allowed our algorithms to better specialize in the nuances of each character and thus optimize more quickly.

Through the NLE environment, we had access to both raw and encoded features representing the characters and glyphs on the ASCII interface edition of NetHack, padded byte-arrays holding the message strings displayed during certain action sequences, the bottom line character statistics visible during human playthroughs, and finally an array of glyphs representing the player's inventory. We also developed and tested three distinct action spaces for use in our project. These included the base set of 79 actions and keystrokes included by the NLE team as a default set, a reduced size minimal set of 45 interactions we believed were critical to progression through the game, and a larger 101 key actionspace composed of both the 45 minimal interactions and an additional set of all possible menu labels, with a filtering system to only allow valid actions to be entered at any step.

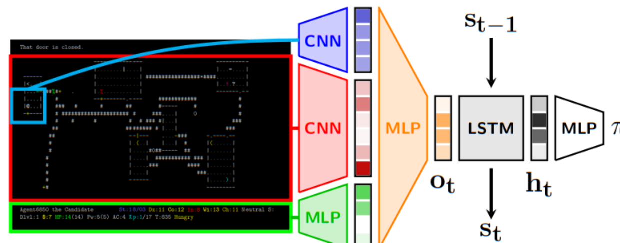
Finally, we also extended the built in NLE reward functions, developing a custom extension to the environment that combined rewards for multiple player statistics, with the goal of increasing the richness of the reward signal to optimize and speed up training. We also still tested and worked with the original reward function, which is based on NetHack's internal scoring metric and very sparse, providing little feedback for most turns and actions of the game.

4 Methods

For our experimental investigation of learning efficiency in NLE, we decided to explore both a simpler Deep Q Network and a complex state-of-the-art IMPALA architecture designed by Facebook for NetHack. Deep Q Networks are a reinforcement learning technique where neural networks are used to learn a function mapping a state action pair to an expected reward, and its difference from the optimal value. In addition to the components necessary for making it a Deep Q Network, we designed a simple 2 Layer Conv2d neural network followed by two Linear layers. For the first Conv2d layer, we selected an out channel of 16 and filter of 3 with no padding. We ReLU the outputs and MaxPool2d with a filter of 3, and padding of (0, 1). Then, we pass through a second Conv2d Layer, with an out channel of 64, filter of 3 and no padding. Once again, we ReLU, and MaxPool2d with a filter of 3 and padding of (1,0). We finally flatten the last layer, pass it through two hidden layers, with a ReLU in between and with the last layer having a node for each action in the environment. The inputs for this neural network consist of four arrays gathered by an NLE environment via an observation: "Glyphs", "Chars", "Colors" and "Specials." Each of these statistics represents an item on a screen, which should give our policy the appropriate information needed to make a good choice of action. By typical DQN fashion, we update our target model every 10 iterations, and stored each transition onto our memory.

The IMPALA algorithm is a decoupled learning system where batched updates are transmitted and received by a centralized learner. In the architecture from Facebook that we used as a template, it features a combination of multiple convolutional neural networks, each being given data from a different set of the features available: One big picture view of the terminal, one zoomed in to the agent (9x9) and one final one for the statistics provided at the bottom of the screen. Once these outcomes are calculated, they are stacked on top of each other and represent the input layer for the final LSTM network, which of course returns the policy estimate. As per Facebook's suggestion, we started with the same default hyperparameters.

Facebook's IMPALA Policy Architecture (Küttler, Heinrich et al.)



We worked to analyze the differences in learning efficiency and training time by testing each of these models with the two characters for investigation we discussed earlier, each of the two reward functions we designed, the base and enhanced, and the three different conditions for the action space. As we tested each of these combinations, we also tightened our hyperparameter selection, and eliminated combinations that performed poorly. We would also attempt to adjust these algorithms directly by modifying their layer design and activations to speed training.

5 Experiments, Results, and Discussion

With these takeaways, we then decided to explore baseline performance through the application of standard deep reinforcement learning algorithms. We investigated a random action system, implemented our own Deep-Q-Network (DQN), and then explored a publicly available baseline implementation of the IMPALA algorithm for the NLE.

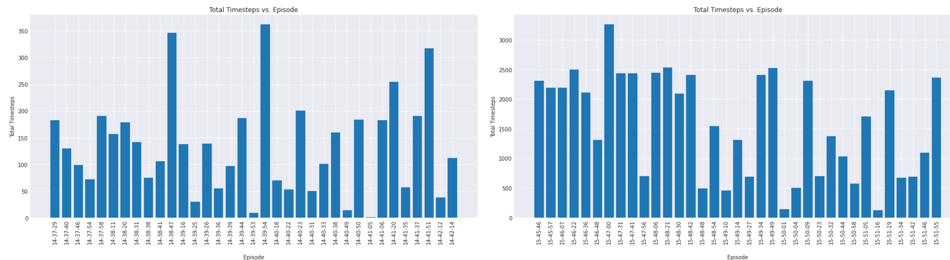
Our random action system verified that our testing setup and packages were all working correctly, and predictably did not perform well, with no agents passing the first level of the NetHack dungeon. We then moved onto building our DQN, using the PyTorch framework (Paszke). Our model would receive as input a batch of 4 different data views of the environment around our agent, provided by NLE. Once stored properly onto a PyTorch tensor, we then feed it to our Conv2d network. What comes out, is a vector, who's max entry represents the action we should take.

For optimization we used the Huber loss to protect against outlier runs due to stochastic events in the game. Though we ran into several problems with enabling GPU acceleration for our training, we were eventually able to get successful baseline performance estimates from our first attempt network. After training for 100,000 episodes, we identified that though several of our best performing episodes had runs that had managed to reach into the second or third level dungeon, out of approximately 50, even these agents had low overall reward scores, with the vast majority of runs(>95

We reasoned this was due to the sparseness of our rewards, and the low likelihood of our model initially spontaneously observing or completing the multi-step actions needed to navigate between dungeon levels. We then investigated the NLE IMPALA implementation, which achieved significantly better results. This suggested to us that we might potentially need to change model architectures, and that the DQN was unable to learn an accurate approximator for the Q-table due to its high dimensionality. Before abandoning the DQN however, we attempted several increasingly deep architectures, but continued to see the same challenges we had initially observed with rewards during training.

To further showcase DQN's limitations, let's take a look at our run with DQN for a Dwarven Valkyrie. After a million iterations, we attained an average timestep of 300, meaning that our agent would die simply after 300 turns that changed the state they were in (such as walking, opening a door, etc.). After training the model for another million steps, we saw the average timestep increase to 3000. While this is a good improvement, running the model for yet another million steps resulted in minimal decrease to our loss function. While the maximum timestep for this iteration seemed to maximize at 4000, it is nowhere near the tenfold increase we saw between the first million and the second (see figures below).

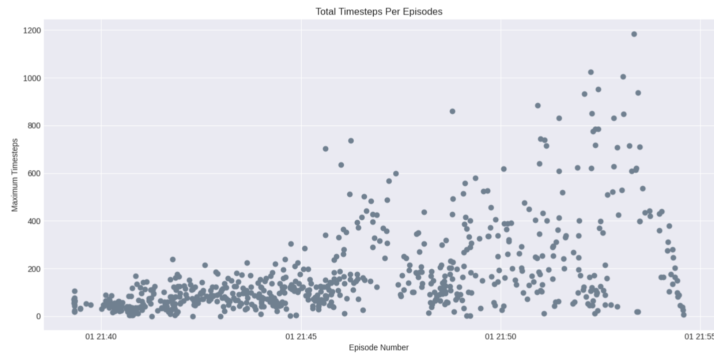
1M vs 2M DQN itertaions with a Dwarven Valkyrie



We also ran a DQN for an Elven Wizard, however the results were not as telling as with the Dwarven Valkyrie. In fact, the timestep did not significantly change for all 3 million iterations. In the wizard's defense, many people consider these two characters different play styles. A valkyrie is far more superior at melee combat than a wizard, but a wizard surpasses a valkyrie in magic.

IMPALA, on the other hand, quickly overtook our DQN. Take, for example, our run with a Elven Wizard (see below).

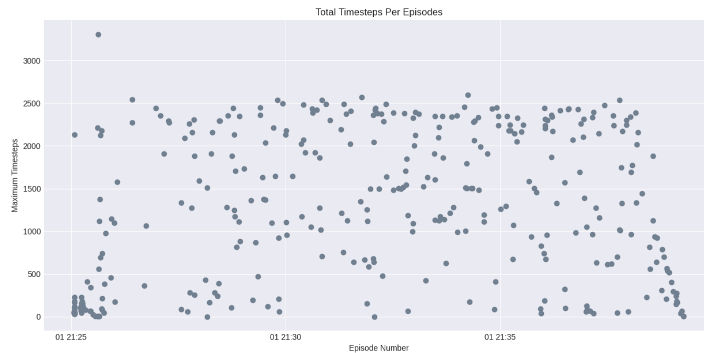
1M IMPALA Iterations with Elven Wizard



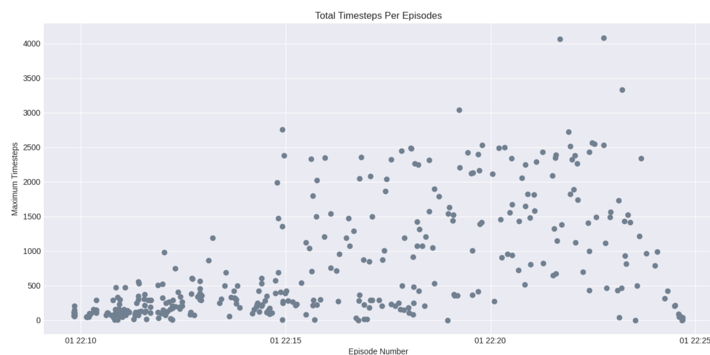
As we see from this image, our Wizard stays alive for more timesteps as the number of episodes decreases. The sharp downward trend at the end is likely caused by the fact that we have reached the end of our timesteps. Furthermore, this was run with 1,000,000 iterations. With IMPALA, that gave us on average a wait time of 13 minutes. On the other hand, our DQN would take up to an hour to run 1,000,000 iterations.

The Dwarven Valkyrie had a general trend of timestep increase. In comparison to our Wizard, who clearly remained the same throughout training. This could be due to a number of factors. For one, it could be due to human error. While we tried our best to get rid of keys we felt were not necessary, or keys that we thought our particular agent would not use, that key may have some other hidden feature that prevents the wizard agent from getting any better. We did our most extensive research as possible, however it is very likely we missed something due to the complexity of NetHack. The other reason could be due to their different playstyles. Our function punishes the agent for remaining in the same timestep at each step, often usually caused by navigating menus. Since a valkyrie is a melee character, usually you get points by moving into monsters, hitting them and earning points for killing them. A wizard, on the other hand, uses magic, spells and other projectiles which often extensive menu navigation is required to access. Furthermore, you often need to read books or scrolls to learn spells, which also require menu navigation. As a wizard needs a menu to be effective, our reward function is preventing them from getting any better as it essentially punishes them for using one of their core abilities.

1M IMPALA Iterations with Elven Wizard (Reeduced Action Space to 45)



1M IMPALA Iterations with Dwarven Valkyrie (Reeduced Action Space to 45)



The results created from our third, and final actions space with menu navigation resulted in similar results to above. While we at first were shocked to see no change, we also understand that there needs to be a lot of training in order to determine what objects you should keep, use right away etc. One million timesteps is relatively small and thus to further explore this method would require more training time.

6 Conclusion/Future Work

The IMPALA algorithm is far more sophisticated than a DQN. It shows in its superior and steady increase in timesteps over episodes, and even more stunning is its speed at running 1 million steps. Furthermore, we learned that Reinforcement learning requires a lot of training, especially for tasks that are as complex at NetHack.

Furthermore, we noticed the importance of a reward function. We clearly saw that our wizard could not learn, from either the DQN implementation or the IMPALA. While we believe it is due to the extensive menu navigation that wizards need to do, we will not know until we implement a custom reward function or perhaps a new iteration of the current one to be more lenient towards menus. We would like to investigate this phenomenon.

Because of NetHack's seemingly endless state space, ideally we would like to train our models for more iterations. According to Küttler, they trained their IMPALA algorithm with 1 Billion steps (Küttler, Heinrich et al.). As we tuned some of the models, we unfortunately could not run our models for that long. With more robust hyperparameter tuning and longer training sessions, we should see and increase in performance.

While Espeholt, Lasse, et al. hold the IMPALA algorithm with high regard, we were unable to compare it thoroughly with our A3C Implementation. After reading some of the source code, we realized that NLE is not very thread safe. A future project we would like to pursue is allowing NLE to utilize multiprocessing so that we are able to run independent environments. Not only would we benefit from such a project, we could make it open source so that other people could benefit from it as well.

7 Contributions

Eduardo and Ayush both contributed equally to this project. Ayush focused on the development of metrics and tools for analyzing run data, including building the terminal data recording utility, graphing utility, summary statistic collecting utility, and custom additions to NLE's action space, reward functions, and character selections. Eduardo focused on the development of the DQN architecture and the IMPALA architecture. Eduardo and Ayush collaborated throughout the entire report.

References

B. R. Kiran et al., "Deep Reinforcement Learning for Autonomous Driving: A Survey," in IEEE Transactions on Intelligent Transportation Systems, doi:10.1109/TITS.2021.3054625.

Espeholt, Lasse, et al. “Impala: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures.” ArXiv.org, 28 June 2018, https://arxiv.org/abs/1802.01561.

Grefenstette, Edward, et al. “Launching the Nethack Challenge at Neurips 2021.” Facebook AI, https://ai.facebook.com/blog/launching-the-nethack-challenge-at-neurips-2021/.

Kaplan, Russell, et al. “Beating Atari with Natural Language Guided Reinforcement Learning.” ArXiv.org, 18 Apr. 2017, https://arxiv.org/abs/1704.05539v1.

“Multicore Deep Reinforcement Learning | Asynchronous Advantage Actor Critic (A3C) Tutorial (PYTORCH).” YouTube, uploaded by Machine Learning with Phil, 15 March 2021, https://www.youtube.com/watch?v=OcIx$_{T}Bu90Q$.

Neftci, E.O., Averbeck, B.B. Reinforcement learning in artificial and biological systems. Nat Mach Intell 1, 133–143 (2019). https://doi.org/10.1038/s42256-019-0025-4

Küttler, Heinrich et al. "The Nethack Learning Environment". Arxiv.Org, 2021, https://arxiv.org/abs/2006.13760.

Shao, Kun et al. "A Survey Of Deep Reinforcement Learning In Video Games". Arxiv.Org, 2021, https://arxiv.org/abs/1912.10944.

Paszke, Adam. "Reinforcement Learning (DQN) Tutorial — Pytorch Tutorials 1.10.0+Cu102 Documentation". Pytorch.Org, 2021, https://pytorch.org/tutorials/intermediate/reinforcement$_{q}learning.html$.

Zhang, Tianjun, et al. "BeBold: Exploration Beyond the Boundary of Explored Regions." arXiv preprint arXiv:2012.08621 (2020).

Cobbe, Karl, et al. "Leveraging procedural generation to benchmark reinforcement learning." International conference on machine learning. PMLR, 2020.

Dannenhauer, Dustin, et al. "dcss-ai-wrapper: An API for Dungeon Crawl Stone Soup providing both Vector and Symbolic State Representations." (2021).