Applying Deep Reinforcement Learning to MiniHack the Planet

Brenton Budler - 1827655
University of the Witwatersrand
School of Computer Science and Applied Mathematics

Kavilan Nair - 1076342
University of the Witwatersrand
School of Computer Science and Applied Mathematics

Thishen Packirisamy - 1839434
University of the Witwatersrand
School of Computer Science and Applied Mathematics

Nicholas Raal - 793528
University of the Witwatersrand
School of Computer Science and Applied Mathematics

***Abstract*—This report investigates two different approaches, namely Deep Q-Network (DQN) and Advantage Actor Critic (A2C) for solving environments in the MiniHack sandbox framework. The goal was to train an agent to perform the sequence of tasks required to complete the *Quest-Hard* environment. A detailed overview of each implementation is described and the results for each agent performing smaller sub tasks and attempting the *Quest-Hard* environment are presented. The DQN was able to complete simple navigation tasks but failed to learn more complex skill acquisition tasks. The A2C agent was able to complete both navigation and skill acquisition tasks but failed to complete *Quest-Hard* . Reward shaping, action space reduction and increasing the maximum episodes were all introduced to facilitate learning but even with these modifications, neither of the agents could complete *Quest-Hard*. The incorporation of prioritized experience replay and dueling DQNs as well as the Asynchronous Advantage Actor Critic (A3C) algorithm with Recurrent Neural Networks are suggested as possible avenues for future research.**

I. Introduction

Reinforcement Learning (RL) serves as a method to allow machines to understand and interact in an environment without relying on human knowledge. On many occasions, games have provided suitable environments to research and explore the possibilities of RL. MiniHack [1] is a framework that allows the design of easily testable and efficient RL environments based on the game NetHack. Generalization is still a challenge in the field of RL and MiniHack [1] provides environments with a high level of randomness to train more generalizable models. In this report, a Deep Q-Network (DQN) and Advantage Actor-Critic algorithm (A2C) are used to tackle various pre-built MiniHack environments and a final evaluation of these approaches on the incredibly challenging *Quest-Hard* environment is performed. The implementation of these models is be located at *deep-rl-minihack-the-planet*. Section I gives a brief overview of the contents of the report, Section II introduces the MiniHack Framework and explores some of the research that has been done using the framework. The DQN algorithm, the network architecture used for the implementation of the DQN algorithm and the hyperparameters used for training the DQN agent are included in Section III while Section IV provides a similar outline of the A2C algorithm, network architecture and hyperparameters. Section V contains the results of training the DQN and A2C agent on navigation and skill acquisition tasks in simpler MiniHack environments while the modifications made to assist learning in the *Quest-Hard* environment and the performance of the agents in this more challenging environment are presented in Section VI. Section VII contains a discussion of the MiniHack Framework and its shortcomings, Section VIII provides recommendations for future work and section IX concludes the paper with a summary of the work and the results attained.

II. Background and Related Work

A. The MiniHack Framework

MiniHack [1] allows for the creation of environments using components of the game NetHack. NetHack is an open-source rogue-like game that is commonly played in the terminal of a computer. The game contains many sequential tasks and a level of randomness. MiniHack leverages the NetHack learning environment (NLE) to facilitate the creation of environments and allow for the easy and efficient training and testing of RL algorithms. The environments in the framework return a large number of different observation spaces including pixel observations, symbolic observations, agent statistics and many more. By default, MiniHack environments give a positive reward of 1 for reaching the end of the environment and a negative reward of 0.01 whenever an action does not cause a change in the game state. There is, however a reward manager which allows for custom rewards to be added to any environment. The framework includes many different skills and environments that either require a single skill or a combination of many skills to be learned. The environment Quest-Hard combines Navigation, PickUp, Inventory and Direction skills.

B. Related Work

[2], [3] apply deep learning techniques to reinforcement learning and introduce the DQN. This network was used to play Atari games given raw pixel data from the games. They were able to achieve human-level control in a range of Atari games and even surpass it in a few. These games, however, do not have the complexity of sequential actions

that are found in some of the MiniHack environments. As a way to demonstrate how RL agents interact and perform with the complexity of MiniHack's environments, Samvelyan *et al.* conducted various tests using existing RL methods based on the TorchBeast framework. These existing RL agents managed to achieve convergence on the majority of navigation tasks after millions of episodes. However, in environments such as *River-MonsterLava*, these methods weren't able to learn, even after 150 million episodes. In terms of skill tasks, environments such as *WoD-Medium* and *LavaCross* were not able to learn after 100 million episodes. This shows that even though the environments are supposed to be used as a way to train RL agents, some of the tasks are still too complicated, even for existing RL methods [1].

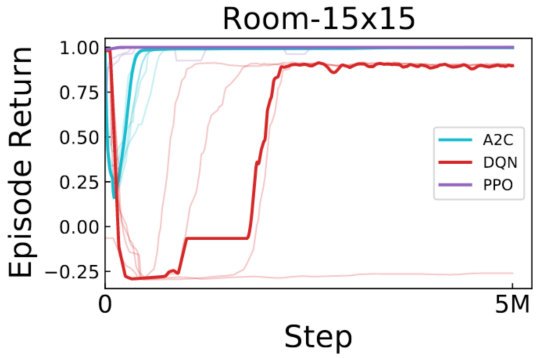

Fig. 1. A comparison between DQN, A2C and PPO from RLlib in the *MiniHack-Room-15x15* task [1]

RLlib is an RL framework that Samvelyan et al. have integrated with MiniHack as an alternative to TorchBeast. To test the methods provided by the library, Samvelyan et al. compared a DQN, A2C and PPO on the *MiniHack-Room-15x15* task. These results can be seen as figure 1 and serve as a good foundation for the work contained in this report.

III. DQN Overview

A DQN is a method in which a neural network is used in conjunction with a Q-learning framework in order to approximate a state-value function. Q-learning is an off policy temporal difference (TD) control algorithm which is used to calculate the learned action-value function Q, to directly approximate the optimal action-value function q_* independent from the policy being followed [4]. Q-learning is defined by equation 1.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (1)$$

We decided to use DQN due to its success in the Atari 2600 video games environment. DQN's have managed to perform at a comparable level to professional human games testers for the majority of Atari games, and in some cases even outperform them [2]. With literature showing that DQN's are a viable reinforcement learning solution to Atari video games, we believed it could be modified to the MiniHack environment in which decent results could be achieved.

A. DQN Algorithm

The training loop for the DQN is described in figure 2. Our implementation made use of an ϵ-greedy policy where we used a threshold that was calculated from the *eps-start* and *eps-end* hyperparameters. If the random value was found to be below the threshold, our agent would choose a random action, otherwise the action will be chosen by the policy.

The replay buffer is used to store the agents experiences at each time step, which is then pooled over multiple episodes into *experience replay*. By having a replay buffer, the learning phase of the algorithm takes random samples from the experience replay which helps to improve the policy by learning from actions closer to the optimal action. Our implementation used a standard replay buffer which was adapted from *raillab/dqn* [5].

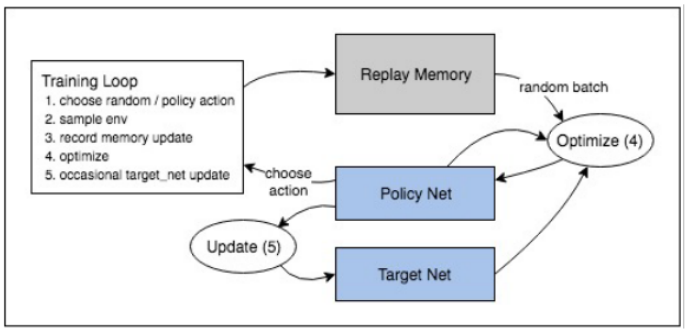

Fig. 2. The training loop architecture used in the DQN algorithm.

The optimization step is run on every iteration where a random batch is sampled from the replay memory in order to perform the training of the new policy. For the optimizing steps, an ADAM optimizer was used as it was found to perform better in DQN's compared to an RMSprop optimizer, which can be seen in figure 3. The RMSprop optimizers have been used throughout literature for reinforcement learning implementations [2], mostly likely due to their increased stability over the ADAM optimizer, but we found that an increased performance would be favoured over stability.

Finally, the target network computes the expected Q values based on the optimization. The target networks weights are occasionally updated, based on the number of steps, by the policy networks weights as a way to keep the target network relevant.

B. DQN Architecture

A double-DQN with a replay-memory was implemented using the PyTorch framework. The MiniHack environment provides multiple observation spaces that a model can use as its input. The glyphs observation space is used for the DQN models and is a 21 x 79 matrix where each glyphs is an ID that represents an entity on the map. This matrix is normalized and used as the input. A convolutional neural network (CNN) was implemented and used for both the policy and target network. The CNN architecture is comprised of two 2D convolutional layers that use the ReLU activation function and make use

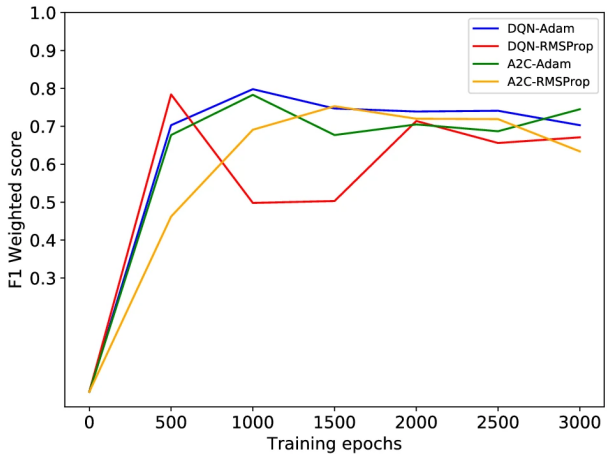

Fig. 3. A comparison between an ADAM optimizer and an RMSprop optimizer, based on their F1 weighted score [6]

of max pooling layers. Two fully connected layers follow the convolutional layers and output an array the size of the action space. The architecture details are shown in table I.

TABLE I
DQN NETWORK ARCHITECTURE

Layer	Details
conv1	Conv2d(1, 20, kernel_size=5, stride=1)
maxpool1	MaxPool2d(kernel_size=2, stride=2)
conv2	Conv2d(20, 50, kernel_size=5, stride=1)
maxpool2	MaxPool2d(kernel_size=2, stride=2)
fc1	Linear(in_features=1600, out_features=500)
fc2	Linear(in_features=500, out_features=action_space)

C. DQN Hyperparameters

The hyperparameters used to train the various DQNs across a variety of environments are shown in table II. These values were chosen after multiple training iterations and the best performing values were chosen. The number of episodes that the model was trained on varied depending on the environment as training a DQN can be resource intense.

- **replay-buffer-size:** Replay buffer size
- **learning-rate:** The learning rate for ADAM optimizer
- **discount-factor:** The discount factor used when calculating the discounted return of an episode
- **batch-size:** Number of transitions to optimize at the same time
- **target-update-freq:** Number of iterations between every target network update
- **eps-start:** ϵ-greedy start threshold
- **eps-end:** ϵ-greedy end threshold
- **eps-fraction:** Percentage of time the epsilon is annealed

IV. ACTOR-CRITIC OVERVIEW

Policy Gradient methods are a class of Reinforcement Learning approaches which seek to learn a parameterized policy $\pi(a \mid s, \boldsymbol{\theta}) = \Pr\{A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\}$. The objective is to learn the policy parameters, $\boldsymbol{\theta}$, which maximizes

TABLE II
HYPERPARAMETER CONFIGURATION FOR THE DQN

Hyperparameter	Value
replay-buffer-size	1e6
learning-rate	0.01
discount-factor	0.9
batch-size	32
target-update-freq	1000
eps-start	1.0
eps-end	0.1
eps-fraction	0.4

the expected discounted return of the policy. This can be stated as the objective function:

$$J(\theta) = \mathbb{E}_{\tau \sim p(\tau|\theta)} \left[\sum_t \gamma^t r_t \mid \pi_\theta \right]$$

where $p(\tau \mid \theta)$ is the probability of a trajectory τ under the policy parameterized by θ. As the objective is to maximize the cost function $J(\boldsymbol{\theta})$, updates made to the parameters during learning approximate gradient ascent. These updates are dependent on the gradient of the objective function with respect to the policy parameters, $\nabla_\theta J(\theta)$, however the Policy Gradient thereom allows us to relate this quantity to the gradient of the policy with respect to the policy parameters as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a) \right] \tag{2}$$

where $Q^{\pi_\theta}(s,a)$ is the action value function of being in state s and taking action a and thereafter following policy π. However, using this expression of $J(\boldsymbol{\theta})$ in policy gradient algorithms such as REINFORCE [7] has shown to have noisy gradients and high variability. One way to reduce the variance and increase stability is the incorporation of a baseline, a quantity that is subtracted from the cumulative discounted rewards (used when determining $Q^{\pi_\theta}(s,a)$), hereby resulting in smaller gradients and more stable updates. A baseline which has shown good results in practice , is the state value function $V^\pi(s)$ [8]. As the actual state value function for a state is not known, it is approximated as $V^\pi(s,\omega)$ where ω represents another set of learnable parameters. Using this as a baseline means the updates used in gradient ascent now become:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(s,a) A(s,a) \right] \tag{3}$$

where $A(s,a) = Q^\pi(s,a) - V^\pi(s,\omega)$. $A(s,a)$ is referred to as the Advantage Function and measures how much better it is to take action a in state s than it would have been if the agent just followed policy π. Using the relationship between the action value function and state value function from the Bellman Optimality equation,

$$Q(s_t, a_t) = \mathbb{E}\left[r_{t+1} + \gamma V(s_{t+1}) \right] \tag{4}$$

we can rewrite the Advantage Function as

$$A(s_t, a_t) = r_{t+1} + \gamma V(s_{t+1}, \omega) - V(s_t, \omega) \tag{5}$$

Using the update rule given in equation (3) and the Advantage Function definition in equation (5) to update parameters during training is referred to as the Advantage Actor Critic (A2C) approach.

The A2C implementation consists of two parts, the"Actor" which approximates the policy $\pi_\theta(s, a)$ and the "Critic" which estimates the state value function, $V^\pi(s, \omega)$. In this research, the Advantage Actor Critic is implemented using a single neural network with common lower layers and separate output layers for the Actor and Critic. This means that only one set of parameters needs to be updated during training (the weights of the network) and these parameters are used to parameterize both the policy and the value function estimates. This network is used to select actions in the environment (the"Actor"), the state is then evaluated using the same network (the "Critic") and then the Advantage Function calculated using the difference between the network's estimates and the actual reward obtained in the environment are used to update the network parameters. This process is outlined in the A2C algorithm outlined below.

A. Actor Critic Algorithm

Algorithm 1 Advantage Actor Critic Algorithm

Initialize neural network parameters θ and learning rate α
Reset the environment and terminal flag
for $t = 1....T$ **do**
  Sample action a_t from $\pi_\theta(s, a)$ based on current state
  Take action a_t receive reward r_{t+1} and next state s_{t+1}
  Calculate the advantage function $A(s_t, a_t)$:
  $A(s_t, a_t) = r_{t+1} + \gamma V(s_{t+1}, \theta) - V(s_t, \theta)$
  Calculate the gradient update $\nabla_\theta J(\theta)$:
  $\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(a_t \mid s_t) A(s_t, a_t)$:
  Update the network parameters:
  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
end for
Repeat for a large number of episodes

B. Actor Critic Architecture

As noted above, a single neural network was used for the implementation of the A2C algorithm in this research. The primary requirements for developing the model architecture were that the model needed to accept a state representation as input and output both a probability distribution over the set of actions (the policy) as well as an estimate of the value function of that state. The MiniHack environment supports a variety of observation spaces including pixel, symbolic and textual representations of the screen, the inventory, player statistics and the onscreen messages. The"glyphs" observation space, a 21×79 matrix representing the map where each glyph is represented by a unique integer between 0 and 5991 was initially selected as the primary input to the neural network. The matrix was flattened into a one-dimensional array and fed into a fully connected Deep Neural Network (DNN) with three hidden layers consisting of $1024, 512$ and 256 neurons respectively. The final hidden layer of the network supplied inputs to two separate fully connected layers, one with a single output neuron (for the value function estimate) and the other with multiple output neurons (one for each of the possible actions). The ReLU activation function was used for each of the hidden layers, as well as the value function output layer (as it is estimating a real-valued continuous variable). The SoftMax activation function was used for the output layer of the policy estimates as it is a probability distribution over a discrete action space. This categorical distribution was then used to sample actions during training.

Preliminary experiments utilizing this model in the A2C algorithm on a number of MiniHack environments indicated that while it was able to learn how to navigate in simple environment layouts, such as in the environment *Room-5x5*, it was unable to navigate through more complex environments such as *Corridor-R2* or interact with objects as required by environments such as in *Eat*. This was largely due to the fact that flattening the state representation resulted in the loss of any spatial information that could have been gained from the two-dimensional glyph representation. In order to preserve this information, the DNN was substituted for a Convolutional Neural Network (CNN), a neural network structure commonly used for image analysis and classification which takes a 2D representation as input, hereby preserving spatial information. The CNN architecture used was based on the popular LeNet-5 architecture which consists of two sets of convolutional and max pooling layers followed by a two fully connected layers [9]. Furthermore, to incorporate additional feedback from the environment in the model training process, the onscreen messages were added as seperate input to the model. The"messages" observation representation from the MiniHack Environment is a 256-dimensional vector representing the utf-8 encodings of the onscreen message prompts. This vector was fed through a single fully connected layer and the output was concatenated with the output of the CNN before being fed to the two separate output layers. Both the glyphs and message inputs are normalized before being fed as input to the network to accelerate model training.

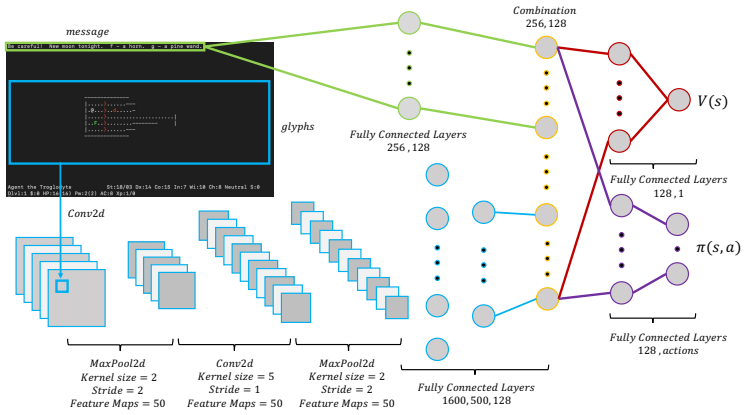

Fig. 4. Advantage Actor Critic Model Architecture

C. Actor Critic Hyperparameters

Below is a brief explanation of the various parameters used when training the A2C network, Table III indicates the values used for each hyperparameter.

- **max-episode-length**:According to the A2C algorithm above, the process of selecting actions, and using the observed reward to update network parameters should continue until the environment reaches a terminal state, T. However, in practice this is not always feasible as often an agent can take millions of steps before reaching the terminal state. In order to combat exorbitant training times, the number of steps an agent can take in an environment before the episode terminates is restricted using this hyperparameter. When creating an environment using MiniHack this restriction defaults to $250-300$ steps depending on the environment so it needs to be manually increased as needed.
- **num-episodes**: The number of episodes to use for training. Experimental results indicated that 1000 episodes is sufficient for most of the environments evaluated in MiniHack
- **learning-rate**: The learning rate used for the ADAM optimizer during model training. Once again, the ADAM optimizer is selected as it has been shown to outperform RMSProp in A2C implementations as outlined by figure 2.
- **gamma**: The discount factor used when calculating the discounted returns of an episode. Returns are also standardized by dividing by the standard deviation to stabilize training.

TABLE III
HYPERPARAMETER CONFIGURATION FOR MODELS

Hyperparameter	Value
max-episode-length	1000 - Simple Environments 10 000 - Complicated Environments
num-episodes	1000
learning-rate	0.02
gamma	0.99

V. RESULTS

MiniHack provides a large variety of different environments for training and evaluating agent performance as well as the opportunity for users to create their own custom environments. The pre-built environments that come with the MiniHack package are divided into two main categories according to the primary task an agent is required to perform in that environment. The first of these categories is the Navigation tasks which range from the basic learning required to navigate from a fixed starting position to a fixed goal position, like in the environment *Room-5x5*, to the more complicated exploration and memorization required to navigate out of a procedurally generated maze in environments like *Maze-Walk*. The second category of environments, namely Skills Acquisition tasks, requires the agent to develop a certain skill to succeed. These skills consist of a sequence of actions that need to be performed such as navigating to a certain location, choosing to interact with an item at that location and confirming your choice, this is referred to as the *Confirmation* skill. Other skills include learning to pickup an object (the *PickUp* skill) as well ass selecting an object from the inventory and choosing to utilize that item (the *InventorySelect* skill). More challenging environments which require an agent to develop a combination of these skills are also included in the package.
Both the DQN model and A2C model were trained on various MiniHack environments. Five training iterations across different random seeded environments were run and averaged to ensure that the results obtained are not specific to the environment seed. This also ensures that the results can be reproduced. All plots in this section contain the five seeded runs shown with high opacity and the averaged results plotted with a solid line. All DQN plots are shown in maroon, and all A2C plots are shown in teal.

A. MiniHack Navigation Tasks

In order to demonstrate the DQN and the A2C agent's ability to learn basic navigation, these agents were both trained and evaluated using the *Room-5x5* environment. It is important to note that the action space of this environment consists solely of the cardinal directions North, South, East, West, North West, North East, South West and South East for a total of 8 possible actions. As noted above, the agent's objective in the *Room-5x5* environment is to find the staircase down. The agent receives a reward of -0.01 for hitting the boundary walls of the environment and a reward of $+1$ for reaching the staircase down (the goal position). Both the DQN and A2C models were able to learn how to solve the *Room 5x5* navigation task.

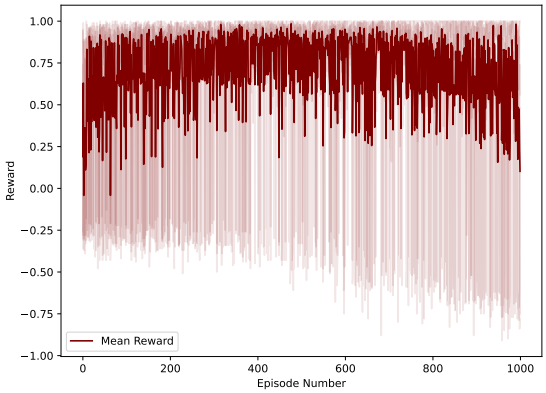

Fig. 5. DQN *Room 5x5* Episode Rewards

The average returns per episode across 1000 episodes on this task for the DQN are shown in figure 5. The DQN started poorly in the first 100 episodes but then managed to maintain a higher average reward thereafter. The DQN does however still makes mistakes and there is still a large amount of variance in the rewards as the model learns.

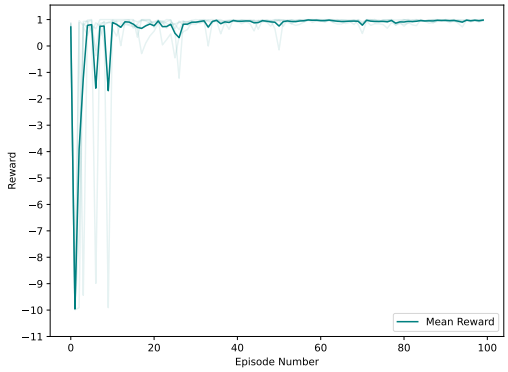

Fig. 6. A2C *Room 5x5* Episode Rewards

The A2C model was trained for only 100 episodes as it was able to converge faster. The episode reward plot for the A2C model solving the *Room 5x5* navigation task is shown in figure 6. The model struggles to solve the task in the first 10 episodes but quickly learns to solve the problem as the average reward is starts converging to 1 after only 20 episodes.

B. MiniHack Skill Acquisition Tasks

In the *Eat* environment, an agent is required to discover the interaction between an object and its actions. The objective is for the agent to locate an apple, choose to eat it and confirm its choice. Both the starting position of the agent and the location of the apple are randomized in every episode. Once again, the agent receives a reward of -0.01 for hitting the boundary walls of the environment but is only awarded a $+1$ reward if it successfully performs the sequence of actions required to eat the apple. By default, Skill Acquisition tasks consist of all 78 actions available in MiniHack environments, majority of these are not required for solving the *Eat* task and therefore the action space is restricted to only include the cardinal directions as well as the EAT command.

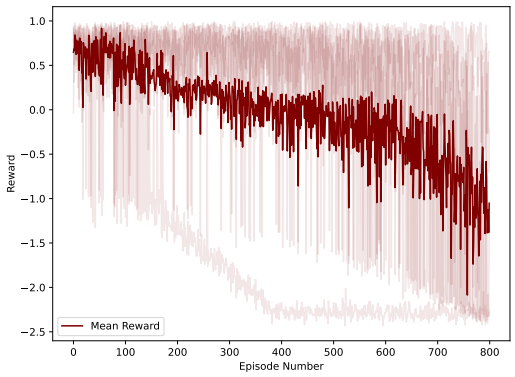

Fig. 7. DQN *Eat* Episode Rewards

The DQN model was unable to solve the *Eat* task. The episode return plot is shown in figure 7. The general trend of the mean reward is downwards, showing that it failed to learn. The model is able to do better in the beginning as it explores more choosing random actions when the training first starts.

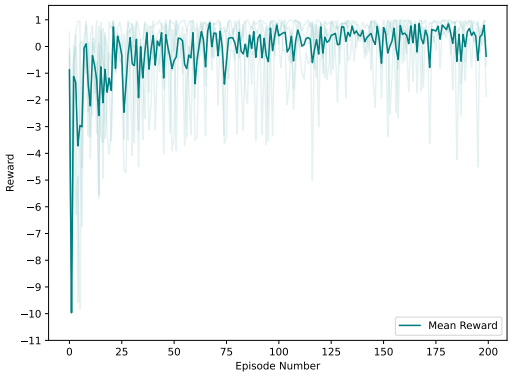

Fig. 8. A2C *Eat* Episode Rewards

The A2C model was able to solve this task as shown by its episode reward plot shown in figure 8. There is more variance in the rewards for this task when compared to the navigation task showing that it was harder for the model to learn. It does however still converge to an average reward close to 1.

VI. The Quest Hard Environment

Quest-Hard requires the agent to perform a large number of sequential actions before it returns a single sparse reward for succeeding at the end of the episode. This includes navigating out of a procedurally generated maze, picking up an item, using the item to cross a river of lava, killing a minotaur and reaching the staircase down. In order to decrease the difficulty of learning such a complicated task, reward shaping and actions space limitation were introduced.

A. Reward Shaping

In *Quest-Hard* the agent spawns in a procedurally generated maze and can only see the cells around it. As the agent moves through unexplored areas of the maze more blocks are revealed. To encourage exploration of the maze a custom reward was created in which compared the number of revealed cells in the current state with the number of revealed cells in the previous state. If the number of revealed cells in the current state was more than the previous, the agent receives a positive reward and this encourages the agent to uncover as much of the map as possible. Another useful method of reward shaping is the co-ordinate reward in which reaching a specific co-ordinate gives a reward. A positive reward was placed at the end of the maze, as this position remains constant in every episode of *Quest-Hard*. A positive reward was also introduced for killing the minotaur, another subtask which the agent needs to complete in order to complete *Quest-Hard*.

B. Action Space Limitation

Once again, the majority of the actions in the default action space of the *Quest-Hard* environment are not required to complete the tasks. The action space was limited to only include the following actions:

- Cardinal Directions
- NorthWest
- PICKUP
- APPLY
- FIRE
- RUSH
- ZAP
- PUTON
- READ
- WEAR
- QUAFF
- PRAY

C. Increasing Max Iterations

As noted previously, the default maximum number of steps in the environment is not sufficient for the agent to find its way out of the maze so the maximum number of steps per episode was manually increased to 10000.

D. MiniHack Quest Hard Results

The DQN was unable to learn how to solve the *Quest-Hard* environment. The episode returns continue to get worse as the model trains as shown in figure 10. The potential cause for this could be that at the start of training, the DQN explores and chooses random actions more frequently. The model is unable to learn and performs worse as less random actions are chosen.

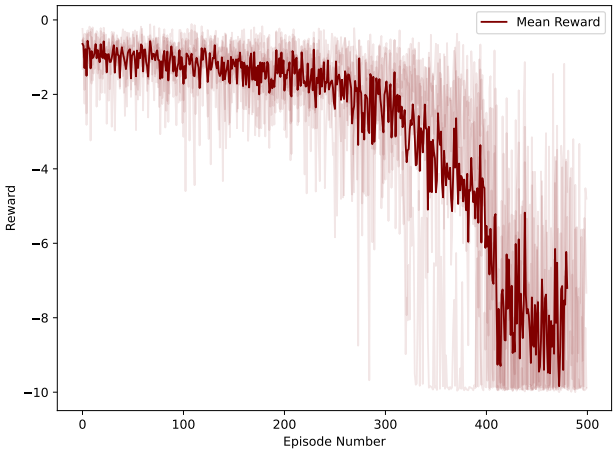

Fig. 9. DQN *Quest-Hard* Episode Rewards

Utilizing the reward shaping outlined above, the A2C model was able to learn how to explore more of the procedurally generated maze in the beginning of *Quest-Hard* but was unable to escape the maze and therefore could not complete the subsequent tasks required to beat the environment.

VII. Discussion

MiniHack is a powerful sandbox environment for RL based tasks, however, it does have it's downfalls. Firstly, the documentation provided for the MiniHack framework is riddled with errors, incorrect implementation details and lacks detailed examples of how to use the various features. Secondly, the *RewardManager()* function which is provided to incorporate custom rewards for reward shaping still has a number of implementation issues. For example, a location reward which should only reward an agent for reaching a specific location, will automatically reward the agent as the episode begins if

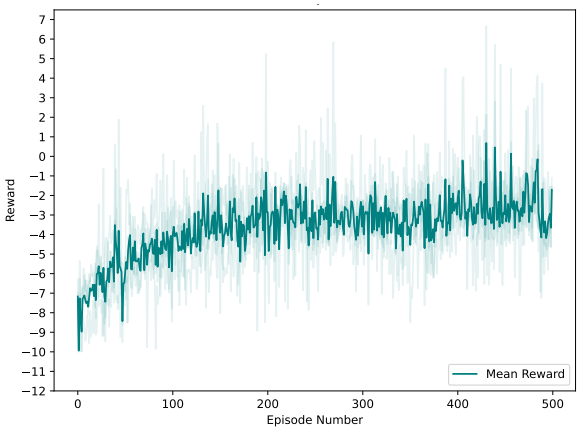

Fig. 10. A2C *Quest-Hard* Episode Rewards with Custom Rewards

the location is not currently visible on the screen. Furthermore, certain actions in environments have multiple uses, for instance when prompted to answer "y/n" by an in game message, the agent has to use the North-West command to select "y". Having a single key assigned to multiple actions make learning certain environments significantly more difficult for RL agents. These issues combined with the large amount of processing power and time required to train RL agents to complete the more complicated environments, suggest why the proposed models were not able to attain the expected results in the *Quest Hard* environment.

VIII. Future Recommendations

A. DQN

In order to potentially improve the results and reduce the computation time required to train the DQN, we would recommend investigating prioritized experience replay. A prioritised experience replay buffer, as described by Schaul et al. [6], was able to outperform a uniform replay in 41 out of 49 Atari games. By implementing the prioritized experience replay, we believe that the DQN agent would learn more efficiently as it would be able to prioritize experiences, based on their significance, from its past more frequently than the uniform buffer used in our implementation.

The inclusion of a long short-term memory (LSTM) cell into our neural network is another approach we recommend taking in future implementations. We found that the DQN would sometimes figure out the environment relatively quickly, but then over more iterations, it would begin to explore more and 'forget' how to get to the terminal state. By using an LSTM, the agent will have better memory of how to achieve the terminal state which could potentially improve the DQN results significantly.

The final recommendation is to change the network architecture to that of a duelling DQN. A duelling DQN explicitly separates the estimators in a state value function and the state-dependent action advantage function [10]. In doing so, the network can learn which states are valuable without the need to learn what happens when each action is taken for each state,

thus allowing for a more generalized learning approach. Wang *et al.* managed to outperform state-of-the-art implementations on the Atari 2600 games environment, and as such, we believe would be a beneficial change to the DQN we implemented for MiniHack.

B. A2C

Recently, a modified version of the A2C algorithm, the Asynchoronous Advantage Actor Crtic (A3C), which allows for the parallelization of training, has shown promising results in the RL field. A3C involves the incorporation of multiple agents, each instantiated with its own neural network, which interact with the environment and use this experience to update both their own local network parameters as well as the network parameters of a global neural network shared across agents. The A3C algorithm was first presented by Mnih *et al.* in 2016 where they demonstrated the ability of A3C to significantly outperform a variety of RL algorithms including a DQN on the Atari 2600 games *Beamrider, Breakout, Pong, Q*bert* and *Space Invaders* [11]. Due to the training parallelization, the A3C approach also boasts a dramatic reduction in training time which could be beneficial considering the complexity of the MiniHack environments which is why the A3C is recommended as a future direction for the training and evaluation of RL agents in the MiniHack environment. Incorporating Long Short-Term Memory (LSTM) Cells into the neural network of the agents in the A3C implementation could also be beneficial as this would allow for the temporal aspect of game-playing to be incorporated into the learning process of the agents.

IX. Conclusion

This report details the implementation and results obtained on MiniHack environments by two different RL agents. The policy based method, A2C, outperformed the value function based method, DQN, in both the navigation and skill acquisitions tasks. The DQN was found to have a high variance during learning, even in simple environments such as *Room-5x5*, whilst the A2C was able to converge to the maximimum reward in the same environments in a fraction of the episodes. Some future recommendations have been discussed as possible solutions to improving the two agents. Unfortunately, both agents were unable to perform on the *Quest-Hard* environment even when adaptations to the environment including reward shaping, action space reduction and increasing the maximum number of training iterations were implemented. The MiniHack environment is a complicated RL sandbox environment that does have some downfalls which combined with its vast complexity and large action space results in existing RL methods struggling to solve many of the more complicated environments, including *Quest-Hard*.

References

[1] M. Samvelyan, R. Kirk, V. Kurin, J. Parker-Holder, M. Jiang, E. Hambro, F. Petroni, H. Kuttler, E. Grefenstette, and T. Rocktäschel, "Minihack the planet: A sandbox for open-ended reinforcement learning research," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. [Online]. Available: https://openreview.net/forum?id=skFwlyefkWJ

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–33, 02 2015.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[5] B. Shall and S. James, "Double dqn," https://github.com/raillab/dqn, 2018.

[6] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[7] R. Williams, "A class of gradient-estimation algorithms for reinforcement learning in neural networks," in *Proceedings of the International Conference on Neural Networks*, 1987, pp. II–601.

[8] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[9] Y. LeCun, L. Jackel, L. Bottou, A. Brunot, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger *et al.*, "Comparison of learning algorithms for handwritten digit recognition," in *International conference on artificial neural networks*, vol. 60. Perth, Australia, 1995, pp. 53–60.

[10] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015. [Online]. Available: http://arxiv.org/abs/1511.06581

[11] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.