Reinforcement Learning Assignment MiniHack the Planet

1st Matthew Dacre*
Student number: 2091295
2091295@students.wits.ac.za

2nd Joshua Wacks*
Student Number: 2143116
2143116@students.wits.ac.za

3rd Alex Vogt*
Student Number: 2152320
2152320@students.wits.ac.za

I. Introduction

Reinforcement learning enables machines, known as agents, to learn, interact and optimize an environment without any or minimal human intervention or knowledge. These agents learn to optimize the outcomes or the return from a given environment through a variety of techniques which will be expanded upon in this paper.
In this project we will be using the Minihack framework which is a framework specifically designed for reinforcement learning research. We will be putting two reinforcement learning models to the test in the Minihack-Quest-Hard-v0 environment, these being DQN and REINFORCE.
The implementations can be found at: https://github.com/AlexVogt1/RL-MiniHack/

A. The MiniHack environment

The Minihack environment was built on top of the nethack framework and the Gym framework, utilizing and building upon both to create a large, diverse and challenging testbed for Reinforcement learning. The most challenging being the Minihack-Quest-Hard-v0.

1) Minihack-Quest-Hard-v0: What makes this the most challenging testbed is the fact that there are multiple extremely difficult sub-challenges that build up this single quest. The first being the maze, it has been shown that RL agents are able to learn to solve mazes quite effectively. However, in this quest the maze is procedurally generated to form a very complex random maze. For an RL agent to learn in random mazes is notoriously difficult due to the large variability and to make the maze even more difficult the agent's starting position is also randomized and the view distance adds another layer of variability to the learning. The second challenge is the lava river-crossing once it has completed the maze. Here the agent is tasked with finding and utilizing objects in-order to cross the river. The third and final sub-task, after crossing the lava-river, is to battle a very powerful monster to reach the end of this difficult task.

2) The Reward System: The Minihack framework allows for the utilization of custom rewards to build on top of or replace the default rewards listed in table 1.

Parameter	Default Value	Description
reward_win	1	the reward received upon successfully completing an episode
reward_lose	0	the reward received upon death or aborting
penalty_mode	"constant"	name of the mode for calculating the step penalty. Can be constant, exp, square, linear, or always
penalty_step	-0.01	constant applied to amount of frozen steps.
penalty_time	0	constant applied to amount of non-frozen steps

TABLE I: Default Rewards

II. DQN

A. The DQN Algorithm

1) The RL Problem and Background: An agent in an reinforcement learning environment has the overall goal to maximise the total return over time. To maximise this return it is sensible to find the optimal actions which lead to optimal immediate rewards as part of this overall return. The optimal action at a given state is part of a policy, rules on how the agent should choose actions, specifically the optimal policy. Finding this optimal policy is known as the RL problem.

Often in problems with small state and action spaces finding the optimal policy can be done by iterating through improved versions of the policy in an algorithm known as policy iteration. We can use different update rules in this algorithm ranging from full-backup update steps such as Monte Carlo, 1-step look aheads such as TD(O), or somewhere in between with the use of TD(λ).

However, in many applications such as robotics, or end to end learning environments such as video games, both the action and state space are incredibly large(Go has 10^{170} states) and in some cases even infinite, directly affecting the use of the aforementioned update steps for the following reasons. These update steps require that we visit every state at least a few times, but this will not be possible in environments with very large state spaces. Thus, it may be better to approximate the value function instead, where the aim is now to learn a parameterised function to approximate the true value function. The next problem that is commonly faced with the use of linear value function approximators is that they do not scale well with the number of variables, to solve this problem we utilise deep architectures to allow the network to learn the features as part of the training process. This is where Deep Q-Networks come in, providing a means for a function approximate to perform end-to-end learning while scaling well with the number of features [1].

2) Architecture: The DQN agent combines Q-learning with a deep convolutional ANN(Artificial Neural Network) specialized for processing spatial arrays of data such as images. We describe the architecture of the ANN and the purpose of each of each components below.

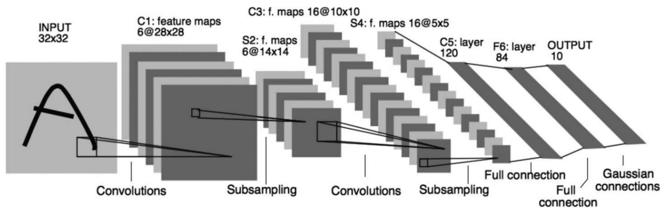

Fig. 1: Deep convolutional network.

In figure 1 above, we can see that the image is first passed through a convolutional layer which produces 6 feature maps, of size 28*28 units. Each unit has 5*5 overlapping receptive fields. The next layer is a subsampling layer which reduces the spatial resolution of the feature maps. The subsampling layer works by averaging over a receptive field of units in the feature maps of the preceding convolutional layer. This subsampling adds the property of spatial invariance to the network. This process is repeated again through the next two layers. These two layers of convolutions then subsampling result in meaningful features being extracted from the images. These features are then passed onto 2 fully connected layers where learning is performed and finally onto a Gaussian connection layer which will produce output. This output is the instructions which are to be directly played into the game [1].

3) Problems and solutions: Before we can delve into the steps of the algorithm there are other components of the architecture which need to be understood. These components were introduced to provide solutions to the following high-level problems. The first problem is that the data is not i.i.d (Independent and identically distributed) the input consists of the pixels from a game and thus each image, each state, is generated from the past state, therefore dependent on the previous input. The second issue is that the problem the agent is trying to solve is non-stationary this is because the target is continuously changing depending on the environment.

To solve the first problem a replay buffer is introduced, where the agent can reuse batches of old data to update the current network. Sampling mini-batches of transitions from the replay buffer decorrelates samples, enforcing the data to be more i.i.d. By using experience replay the behavior distribution is also averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. Note that when learning by experience replay, it is necessary to learn off-policy (because our current parameters are different to those used to generate the sample), which motivates the choice of Q-learning.Because the agent is learning and improving very old data is generally overwritten after time, we do not implement it in our architecture, but we could expand further on this idea with methods such as prioritised experience replay [2].

To overcome the problem of changing targets, an identical second copy of the network is used. This second copy is known as the target network and helps to create a more stationary problem. We specifically freeze the parameters of this network while performing updates, which helps to avoid oscillations while training. We now can compute the Q-learning target with respect to these frozen parameters. To incorporate the training and updates that have been made we periodically reset this target network to the current network used during updates and training [1].

4) Algorithm Loss Function: The DQN algorithm is centered around minimising the following loss function:

$$L_i(W_i) = E_{s,a,r,s'\backsim D_i}[(r+\gamma max_{a'}Q(s',a',\theta_i^-)-Q(s,a,\theta_i))^2] \tag{1}$$

Where $r + \gamma max_{a'}Q(s', a', \theta_i^-)$ is our target, the value that we would like to update towards and $Q(s', a', \theta_i^-)$ is our current estimate of the q-value.
Differentiating this loss function with respect to the weights we arrive at the following gradient:

$$\nabla_{\theta_i} L_i(\theta_i) = E_{s,a\backsim p();s'\backsim\mathcal{E}}[y_i - Q(s,a,\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)] \tag{2}$$

Where y_i is our target and is given as:

$$y_i = (r + \gamma max_{a'}Q(s', a', \theta_i^-)) \tag{3}$$

[2]

5) Algorithm Steps: The DQN algorithm executes the following high-level steps outlined in figure 2 below. The

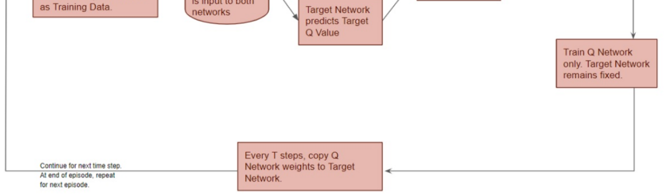
Experience Replay gathers a training sample by interacting with the environment, and saves it as Training Data.
Random batch of Training data is input to both networks
Q Network predicts Q Value
Target Network predicts Target Q Value
Compute Loss
Train Q Network only. Target Network remains fixed.
Every T steps, copy Q Network weights to Target Network.
Continue for next time step. At end of episode, repeat for next episode.

Fig. 2: DQN Algorithm Steps. [3]

DQN algorithm that we used is composed of these steps:

```
1) Initialize our replay memory D, with capacity N
2) Initialize action-value function Q (The convolutional ANN) with random weights
3) For each episode do the following:
    a) Initialize the sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    b) For each time-step(t) of the episode do:
        i) Select an action,$a_t$ from our Q action-value function using epsilon-greedy.
        ii) Execute action $a_t$ in emulator and observe reward $r_t$ and subsequent image $x_{t+1}$
        iii) Set the next-state $s_{t+1} = s_t, a_t, x_{t+1}$ and pre-process $\phi_{t+1} = \phi(s_{t+1})$
        iv) Store this transition $\phi_t, a_t, r_t, \phi_{t+1}$ in our replay buffer D.
        v) Sample a mini-batch of transitions $\phi_j, a_j, r_j, \phi_{j+1}$ from D.
        vi) Set our target $y_j$ such that:
            $y_j = r_j$ when $\phi_{j+1}$ is terminal
            $y_j = r_j + \gamma max_{a\prime} Q(\phi_{j+1}, a\prime; \theta)$ when $\phi_{j+1}$ is non-terminal
        vii) Execute a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation (2) above.
```

6) Conclusion: The DQN algorithm showed promising results through the the use of the of a deep neural network which allowed it to perform end-to-end learning and to scale well with the number of variables. It also overcame the problem of non-stationary targets and non- i.i.d data by utilising a target network and replay buffer respectively. However, the DQN algorithm also implements three techniques that form the 'deadly triad'; function approximation, bootstrapping and off-policy training which is known to lead to instability and divergence while training. We were aware of these aspects while training and tested an extensive range of hyper-parameters and assessed the model on a substantial range of environments to check for stability and any possible bugs.

B. *DQN design choices*

- Epsilon start value: ϵ = 1, we initially want the agent to do as much exploration in the environment as possible, thus we start off by only doing exploration.
- Epsilon decay value: We decay epsilon at a rate of 0.4 of the current epsilon per epoch. We decay epsilon as a fraction of the current value which means we slow down the rate of epsilon decay as the episode continues.
- Epsilon end = 0.3 was used because in the environments as complex as the ones explored in this paper we always want to have some exploration.
- Input Space: We take in the glyphs, pixels and message as our input space. The pixels were required as the DQN is designed to perform full end-to-end learning.
- Architecture Choices: We chose to implement 2 sets of convolution layers followed by 2 pooling layers, which ensures adequate feature detection and adds some scale invariance to the network. Subsequently we have 2 fully connected layers which enables the network to learn possibly non-linear combinations between these features.
- Action Choice: We tried to choose the actions that were available to the agent as minimally as possible so that it could complete each stage effectively, without over complicating learning.
- Choice of Gamma: We specifically chose a large gamma as we knew that a lot of the environments had long event horizons as positive rewards were only ever received very late in the episodes.

C. *Achieving Sub-Goals*

When the same agent is tested on environments for each sub-task, it can perform reasonably well. When the agent is required to perform all of these tasks in a single environment however, there is no measurable performance. This indicates that the agent is learning how to act in the environment, but is not able to complete them all in sequence.

Below, are plots of rewards per episode for our trained agent for each sub-problem.

1) MazeWalk: Using the 'MiniHack-MazeWalk-15x15-v0' environment, which is a 15 by 15 size room with randomly generated mazes and measures the agents ability to navigate a random maze environment, the fully trained DQN agent was training for 600 epochs with the rewards plotted in Figure 3

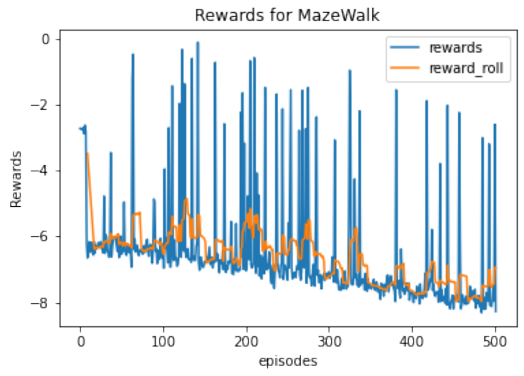

Fig. 3: Reward For Maze

2) Random Room: Using the 'MiniHack-Room-Random-15x15-v0', which is a room of size 15 by 15 with a random exit and entrance location and measure the ability of the agent to find the exit of the environment, the fully trained agent was testing for 600 epochs with the rewards plotted in Figure 4

3) Lava Crossing: Using the 'MiniHack-Lava-Crossing-v0' which has five random instances where the agent needs to cross the lava using (i) potion of levitation, (ii) ring of levitation, (iii) levitation boots, (iv) frost horn, or (v) wand of cold. The agent was trained for 500 epochs which is plotted in Figure 5.

D. *Quest-Hard*

Using the 'MiniHack-Quest-Hard-v0' which is an environment that requires the agent to work its way though a randomly

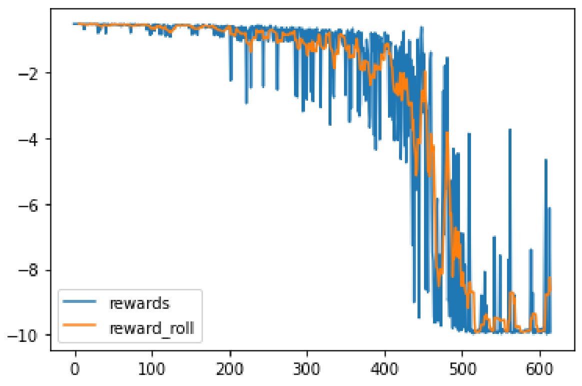

Fig. 4: Reward for Room Random

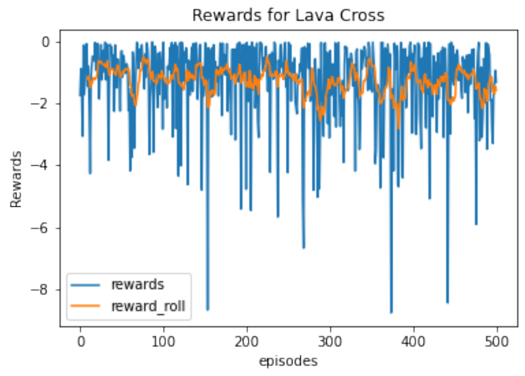

Fig. 5: Reward For Reward for Lava Cross

procedurally generated maze, then crossing a river of lava and lastly has to fight monster to reach the end of the level. The Agent was trained for 500 and the rewards are plotted in Figure 6.

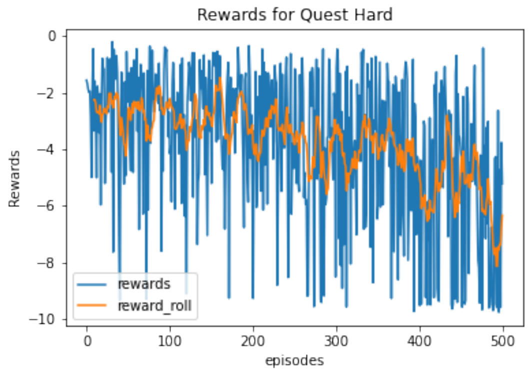

Fig. 6: Reward For Quest Hard

E. DQN rewards

Looking at all the reward plots of the DQN it is clear that the DQN does not perform well in these environments. In the Quest-hard environment there was a lot of variance and the agent was unable to learn in the level this is likely to the the high level of random variability and long horizons in the environment compared to the sub-taks.

F. DQN hyperparameters

The hyperparameters used for the DQN can be found on Table II

Parameter	Description	Value
Replay buffer size	Size of the replay buffer	1000
Learning Rate	The Learning rate for the optimizer	0.02
Gamma	The discount factor	0.99
num steps	The maximum number of steps can be taken	2e6
batch size	number of transitions to optimize at the same time	32
Learning start	The step at which we start to learn	1000
Learning frequency	Number of transitions between every learning step	1
Use double DQN	Use Double Deep Q-Learning	True
Target update Frequency	The Frequency at which we update the target network	1000
Epsilon start	Starting e-greedy policy probability	1.0
Epsilon end	final e-greedy policy probability	0.3
Epsilon fraction	The decay fraction of e-greedy probability	0.4

TABLE II: DQN Hyperparameters

G. Video of DQN agent

Gifs of the DQN agent can be found at https://github.com/AlexVogt1/RL-MiniHack/tree/main/dqn/video.

III. REINFORCE

A. The REINFORCE Algorithm

1) Background: Finding the optimal policy to solve the RL problem is often done through action-value methods as has been seen in DQN. Where the agent learns the values of actions and then selects actions based on their estimated action values(the policy). The REINFORCE algorithm approaches the RL problem different, instead it aims to learn a parameterized policy directly that can select actions without consulting a value function [1].

2) Advantages of Policy Approximation: We often learn a parameterised policy $\pi_\theta(s,a) = \pi(s,a;\theta)$, as it is sometimes more suitable in certain environments to learn the policy directly instead of a complex Q-value function. This is because it easier to inject prior information into a policy than it is to do so with a value-function. The second advantage of using policy approximation is that it allows for the selection of actions with arbitrary probabilities. In certain games or environments the optimal policy may be stochastic in nature, but action-value methods have no natural way of finding stochastic optimal policies, whereas we are able to do so by using a parameterized policy.

3) Policy Gradient Theorem: The fundamental idea or theory that allows policy approximation to happen, is the Policy Gradient Theorem. This theorem relates the gradient of performance with respect to the policy parameter to the gradient of the policy. The analytical expression of the Policy Gradient Theorem is:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log_{\pi_\theta}(s,a) Q^{\pi_\theta}(s,a)] \tag{4}$$

This theorem also enables an important theoretical advantage; with continuous policy parameterization the action probabilities change smoothly as a function of the learned parameter, whereas in epsilon-greedy selection the action probabilities may change dramatically for an arbitrarily small change in the estimated action values, if that change results in a different action having the maximal value.

4) Deriving REINFORCE: We define a trajectory under our current policy as $\tau = (s_0, a_0, r_0, s_1, a_1)$. From this trajectory we can define our cost function:

$$J(\theta) = E_{\tau \backsim p(\tau;\theta)}[r(\tau)] \tag{5}$$

Changing the expectation into an integral we get:

$$J(\theta) = \int_\tau r(\tau) p(\tau;\theta) d\tau \tag{6}$$

Taking the derivative of our cost fucntion results in:

$$\nabla_\theta J(\theta) = \int_\tau \nabla_\theta p(\tau;\theta) d\tau \tag{7}$$

Multiplying and dividing through by $p(\tau;\theta)$, then applying the log derivative trick gives us the following expression:

$$\nabla_\theta J(\theta) = \int_\tau (r(\tau) \nabla_\theta log p(\tau;\theta)) p(\tau;\theta) d\tau \tag{8}$$

The probability of a trajectory(τ) given parameters of the model θ is:

$$p(\tau;\theta) = \prod_{t \geq 0} p(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t) \tag{9}$$

Taken the log of this trajectory results in a sum, which is easier to deal with:

$$log p(\tau;\theta) = \sum_{t \geq 0} log p(s_{t+1}|s_t, a_t) + log \pi_\theta(a_t|s_t) \tag{10}$$

Taking the derivative of this log results in:

$$\nabla_\theta log p(\tau;\theta) = \sum_{t \geq 0} \nabla_\theta log \pi_\theta(a_t|s_t) \tag{11}$$

Which allows us to lose dependence on dynamics and finally our gradient estimate is given as:

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta log \pi_\theta(a_t|s_t) \tag{12}$$

5) REINFORCE Algorithm: The REINFORCE algorithm that we followed throughout training is as follows:

1) Initialize the training network, θ with a very high learning rate which helped us to minimize our vanishing gradient problems.
2) For each episode played out:
    a) In state s, choose actions according to policy π given the parameters θ as follows:
    $\pi_\theta : a \sim \pi_\theta(a|s)$
    b) Gather the samples that we have observed $s_1, a_1, r_1, ..., s_T, a_T, r_T$
    c) For each time-step, t = 1 to T perform the update as follows:
        - $\theta \leftarrow \theta + \alpha R_t \nabla_\theta log \pi_\theta(a_t|s_t)$

6) Interpretation Of Our Algorithm: Looking at the estimate of our gradient $\nabla_\theta J(\theta) \approx \Sigma_{t \geq 0} r(\tau) \nabla_\theta log \pi_\theta(a_t|s_t)$ we can see the following interpretation of our update. If the term $r(\tau)$ is high, the rewards we observed for a given trajectory, then push up the probabilities of those seen actions in our policy. However, if $r(\tau)$ is low, push down the probabilities of the actions we saw, as this did not lead us to a high reward. Overall this update step can be seen as a simple version of credit assignment.

7) Improvements: One observation that was made during the training of the REINFORCE agent is that it trained with a high variance, it oscillated between rewards, thus resulting in slow convergence. A possible improvement that can be implemented here in future studies is the use of a baseline in our update step. Where the new estimation of our gradient would be as follows:

$$\nabla_\theta J(\theta) \approx \Sigma_{t \geq 0}(r(\tau) - b(s_t)) \nabla_\theta log \pi_\theta(a_t|s_t) \tag{13}$$

This new baseline term, $b(s_t)$ does not bias the algorithm as it simply adds or subtracts from the observed reward. The most suitable choice of baseline would actually be the value function. The intuition behind choosing the value function is that we would now be changing the policy not directly on the return but instead on whether or not reward was better than expected. This new term resembles an advantage function. Including this baseline, changes the algorithm and now we would be learning both the policy π and the value function. Thus, entering a completely new class of algorithms known as actor-critic algorithms.

8) Conclusion: The REINFORCE algorithm did provide a new approach to how we tackled the environment, and learnt consistently on the smaller tasks but due to the complexity of the larger tasks it did not perform well on them. There are possible improvements that we can try implement in further studies, such as using a baseline in our algorithm.

B. Design Choices

These are following design choices that we chose to implement when training the REINFORCE agent and their respective motivations.

- Glyphs and Chars Input Space: This helped to reduce the dimensionality of the problem as well as the state space, which helped to make learning easier and thus the agent converged on the solution quicker
- Consistent Random Seed: We used the same random seed throughout the experiment to ensure the reproducibility of our results.
- Policy Initialisation: Early in training it was noted that we were running into NaN results and thus to overcome this problem we added a policy initialisation step to run for 50 episodes with a high learning rate and low gamma to initialise the parameters of the network better than random would.
- Network Architecture: We chose a simple neural network architecture of 1 hidden layer to represent our policy function approximation. This enabled the network to add non-linearity to the problem as well as being able to quickly learn the policy.
- Action Choice; We tried to choose the actions that were available to the agent as minimally as possible so that it could complete each stage effectively, without over complicating learning.
- Choice of Gamma: We specifically chose a large gamma as we knew that a lot of the environments had long event horizons as positive rewards were only ever received very late in the episodes.

C. Hyperparameters

The hyperparameters used in training are shown in Table III

D. Achieving Sub-Goals

When the same agent is tested on environments for each sub-task, it can perform reasonably well. When the agent is required to perform all of these tasks in a single environment however, there is no measurable performance. This indicates that the agent is learning how to act in the environment, but is not able to complete them all in sequence.

Parameter	Description	Value
Initialisation Learning Rate	Learning rate for initialisation step	1
Initialisation Length	Number of episodes for parameter initialisation	10
Initialisation Gamma	Discount factor for parameter initialisation	0.3
Learning Rate	Learning rate for policy optimisation	0.1
Learning Rate Decay	Decay rate for Learning Rate	1e-5
Number of Episodes	Number of episodes for learning	10000
Max Episode Length	Max length of each episode	1000
Gamma	Discount factor for learning	0.99

TABLE III: REINFORCE Hyperparameters

Below, A plot of rewards per episode for our trained agent for each sub-problem is shown.

1) MazeWalk: Using the ‘MiniHack-MazeWalk-15x15-v0‘ environment, which is a 15 by 15 size room with randomly generated mazes and measures the agents ability to navigate a random maze environment, the fully trained agent was testing for 500 epochs with the rewards plotted in Figure 7

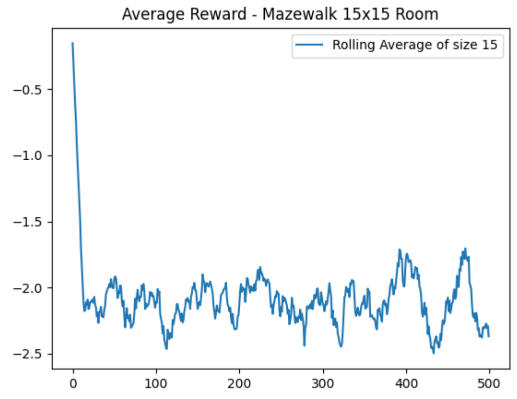

Fig. 7: Reward For Maze

2) Room-Random: Using the ’MiniHack-Room-Random-15x15-v0’, which is a room of size 15 by 15 with a random exit and entrance location and measure the ability of the agent to find the exit of the environment, the fully trained agent was testing for 500 epochs with the rewards plotted in Figure 8

3) LavaCross: Using the ’MiniHack-LavaCross-Levitate-Potion-Inv-v0’, which measures the ability for the agent to use an item to traverse a lava river, the fully trained agent was testing for 500 epochs with the rewards plotted in Figure 9

E. Rewards During training

The average rewards achieved during training are shown in Figure 10. The final model was tested and an average over 5

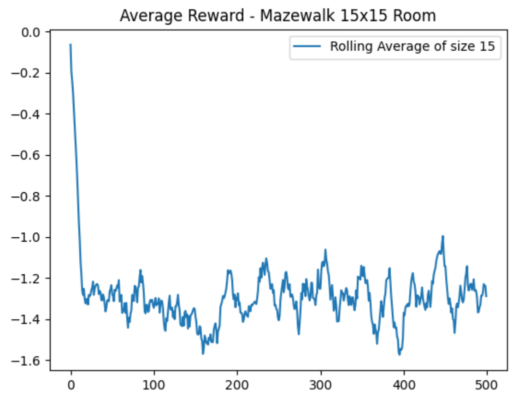

Fig. 8: Rewards for room

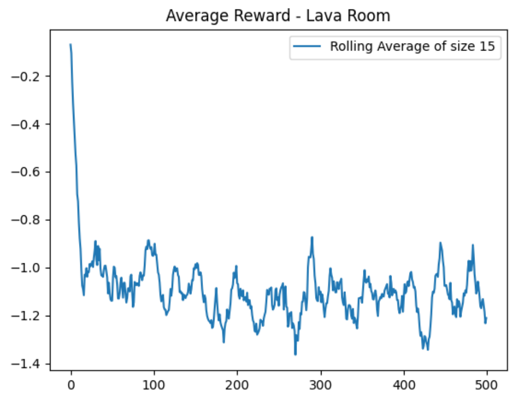

Fig. 9: Rewards for Lava

runs was calculated over a total of 500 episodes each, shown in Figure 11

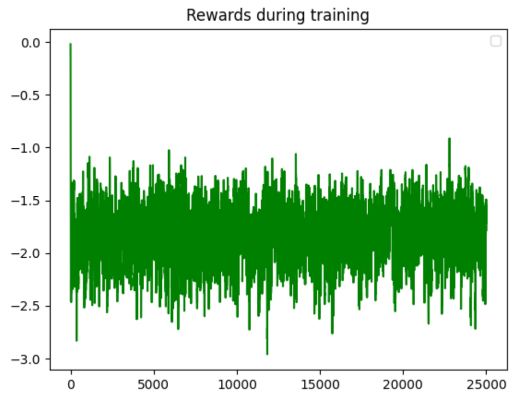

Fig. 10: Rewards During Training

1) Combined Plot of each sub-goal during training: Figure 12 shows the rewards achieved for each sub-goal at each step during training. This is only the first 1000 episodes of training,

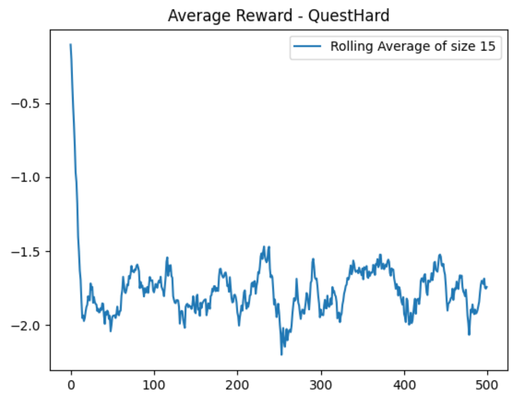

Fig. 11: Rewards After Training - QuestHard

with a sample taken every 10 episodes. The final agent was trianed for a total of 25000 episodes. The agent appears to occasional achieve positive rewards for each individual sub-goal, but never for Quest-Hard. This is likely due to the relatively longer horizons required for the agent to receive a positive reward in Quest-Hard when compared with each sub-task. The overall performance of the agent could likely be improved by providing smaller, intermediate rewards that gives it a better indication of its progress. This shows that the agent is able to achieve sub-goals during training.

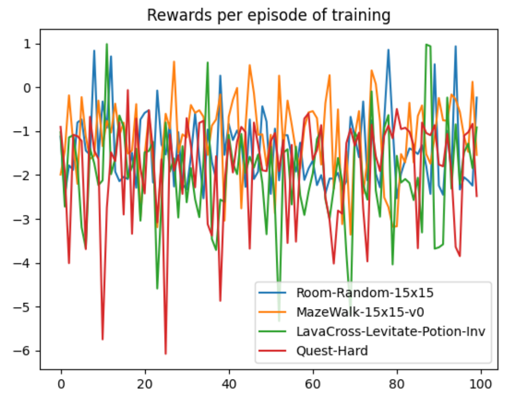

Fig. 12: Rewards for each sub-goals during training - REINFORCE

F. Video of Agent

A gif of our REINFORCE agent can be found at: https://github.com/AlexVogt1/RL-MiniHack/blob/main/reinforce/gifs/reinforce_agent.gif. In the video, the agent can be seen exploring the maze and attempting to find the exit, before the episode ends.

IV. Conclusion

In conclusion, due to high level of complexity, long reward horizons and the random environment MiniHack-Quest-Hard-V0 is very difficult for reinforcement learning agents to perform in and learn effectively. Neither of the models considered in this report were able to sufficiently complete tasks in the environment. In order to increase performance, improvements to the reward structure to make rewards more dense as well as implementing techniques that allow the agent to effectively traverse a random environment, would likely be effective. Improvements such as adding a baseline to the REINFORCE algorithm, or a more effective neural network architecture for the DQN model would likely also contribute to increased performance. We were not able to allow our models significant training time beyond a few days worth of continuous computation due to lack of a consistent supply of power. Longer training schedules with higher performance hardware might yield better results than what was achieved in this report.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.

[3] K. Doshi, “Reinforcement learning explained visually (part 5): Deep q networks ...” Dec 2020. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b