# COMS4061A Report: MiniHack

Ncume, Vuyo 2095458
Mahlangu, Fezile 2089676
Raphiri, Mmasehume 2089198

October 2022

## 1 Introduction

Reinforcement Learning is method that allows agents to interact with the environment optimally without relying on human knowledge. The MiniHack environment is an RL environment that allows us to test our agents. In this project we trained several agents in the MiniHack environment which is built on top of Nethack. The two algorithms that were utilized are DQN, a value based method, and Actor Critic(A2C), a policy based method.

The objective was to train these agents to complete the Quest-Hard environment. The MiniHack environment is hard to learn because there is a significant of amount of randomness which makes generalization difficult. We introduced several techniques in an attempt to better our agents performance. These techniques include reward shaping, choosing between optimizers and reduction of actions that can be selected. The link to our GitHub repository is provided here.
In this report, section 2 provides an overview of the DQN which includes the description of the algorithm, design decisions and architecture and the hyperparameters used in the subsections. Section 3 gives an overview of the Actor Critic which follows the same structure. Section 4 details the MiniHack-Quest-Hard-v0 environment and other environments we used to observe how our agents perform in subtasks. Section 5 discusses the results of our agents and performance analysis. Section 6 provides a comparison of DQN and A2C agents in the *quest hard* environment.

## 2 DQN

### 2.1 Description of algorithm

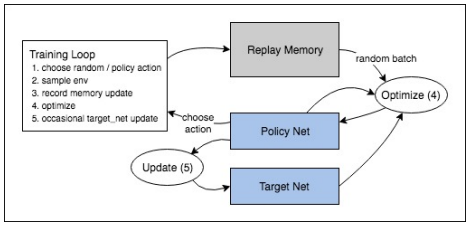


Figure 1: DQN

This algorithm combines Q-learning and neural networks (in this case convolutional neural networks) to map states to their action value. The agent uses experience replay to basically learn from states it has already encountered. The algorithm plays randomly for the first 10000 steps in order to populate the replay buffer then starts learning after those 10000 steps.

The agent first chooses a random/policy action. A random action is chosen with probability epsilon and exploits with probability 1 - epsilon. A random action is any action in the action space of the environment. A policy action is the action given by the policy network (given a state, it gives the action that has the highest q value). Using this action, it takes a step in the environment. The environment returns the reward, next state and a indicator variable "done" for taking the action.

"Done" indicates whether the state is terminal or not It then records(saves) the state, action, reward, next state and "done" to the memory replay buffer. The network learns using "experience replay" by sampling from the replay memory buffer. Learning means the policy network's weights are optimized This updates the network weights by backpropagating using the sampled batch of data (the network trains using the batch from memory).

The TD target is calculated using the target network and the bellman equation Using the TD target, the policy network weights are fitted to the target values The target network is occasionally (e.g. every n steps) updated. The target network weights are set to be the policy network weights. The policy and target networks take the form illustrated in the above figure. The network takes a state and passes it through a convolution neural network and then through a fully connected network. For each action, it produces a q-value given the state. The network basically maps input states with actions.The agent takes the action that has the highest q-value.

In our implementation, we utilize several fully connected layers with ReLU activation functions. The specifics of the architecture will be subsequently discussed.

## 2.2 Design decisions and architecture

There were several decisions made during the implementation of the DQN agent. This ranged from tweaking the optimizer, limiting the action space, and shaping the rewards for adequate learning. Thus in this subsection, we unpack these design decisions and give motivations for our actions.

### 2.2.1 Optimizers

We mainly experimented with 2 optimizers, Adam and RMSprop. Adam is an extension of stochastic gradient descent and is well suited for problems with large amounts of data and parameters ([1]). Like Adam, RMSprop is an extension of the AdaGrad version of gradient descent and has the potential of enabling faster convergence than gradient descent by restricting oscillations towards the minimum ([2], [3]).

It is for these reasons we chose these optimizers, and in the end, we didn't settle for one over the other but instead picked one based on what we empirically discovered was the best for a given task or subtask.

### 2.2.2 Reward Shaping

Reward shaping is a method of "teaching" whereby we give supplemental rewards to our agent to make the task given easier to solve, [7]. It is important to note that we only used this method on the MiniHack-Quest-Hard-v0 environment.

We are aware of the fact that using reward shaping can lead to suboptimal behaviors and can, in some cases, include unwanted biases from humans. The decision therefore to use this method was only due to (a) the sparsity of the rewards in this environment, and (b) the number of sequential subtasks that need to be completed in order to reach the final goal.

We added a negative reward of $-0.01$ to the "It's solid stone" event to make the agent follow the path, a reward of 1 was given to the agent on the "The door opens" event, a reward of 1 for eating an apple, $-0.01$ reward for the "What a strange direction! Never mind." event, a reward of 1 for reaching the end of the maze, and lastly, we gave the agent a reward of 1 for killing the minotaur which is the last task before you can reach the goal.

### 2.2.3 Action space

Through experimentation, we discovered that due to the large action and observation spaces - the random actions we took during the exploration phase did not give the agent enough information to adequately learn from its actions.

For this reason, we took the decision to reduce the randomness as much as we could. This involved reducing the action space to only the actions required for the agent to be able to solve the task.

The actions we allowed the agent to make were limited to INVOKE, OPEN, PICKUP, QUAFF, SWAP, WEAR, WIELD, and ZAP. These actions allow the agent to invoke an object's special powers, open a door, pick up things at the current location, drink something, swap wielded and secondary weapons, wear a piece of armor, use a weapon and zap a wand respectively ([4]).

### 2.2.4 Architecture

We settled on this architecture for our Q-function. Through empirical analysis, we discovered that this design was both efficient in the number of trainable parameters and compute time required to produce meaningful results. The network takes in 81 input features because we utilized the *glyphs_crop* observation and the number of outputs is 8 for the number of actions we required.

| **Layers** | **Value(s)** |
|---|---|
| fc1 | Linear(in_features=81, out_features=32) |
| act1 | ReLU(fc1) |
| fc2 | Linear(in_features=32, out_features=32) |
| act2 | ReLU(fc2) |
| fc3 | Linear(in_features=32, out_features=16) |
| act3 | ReLU(fc3) |
| fc4 | Linear(in_features=16, out_features=8) |

Table 1: Q network design

## 2.3 Hyperparameters

After multiple iterations through different environments, we empirically found that the best hyperparameters are the ones listed in 2. Below we also give a description of what each hyperparameter is and its effects.

| Hyper-parameter | Value(s) |
|---|---|
| learning rate | $1e-3$ |
| discount factor | 0.99 |
| batch size | 128 |
| epsilon | 0.01 |
| learning frequency | 2 (after 10000 steps) |
| update frequency | 1000 |
| replay buffer size | $5e3$ |
| number of steps | $1e6$ |

Table 2: List of hyperparameters for DQN

- **learning rate**: This hyperparameter is the step size we take towards the minimum of the cost function. It is used by the Adam or RMSprop optimizer.
- **discount factor**: Determines how much we care about the future reward as opposed to the immediate reward we might get.

- **batch size**: This is the number of transitions to optimize in one optimization step. Helps in smoothing out the descent towards the minimum.
- **epsilon**: Determines the fraction of times we choose exploratory actions versus greedy ones. We start this value at 1 then slowly decay it to 0.01 as we train.
- **learning frequency**: This is the number of iterations between every optimization step. Updating after every iteration can be computationally expensive and unnecessary.
- **update frequency**: This hyperparameter is the number of iterations between every target network update. Updating the target network after every iteration can lead to unstable training.
- **replay buffer size**: This is the size of the "memory" obtained from our past actions. We use the buffer to keep the data we use to train independent and identically distributed.
- **number of steps**: Determines the total number of steps we run the environment in question for.

# 3 Actor-critic

## 3.1 Description of algorithm

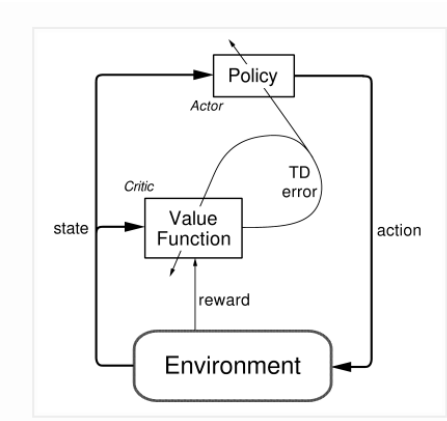


Figure 2: Actor Critic

Actor critic is a Policy Gradient method that approximates the optimal policy by parameterization. The optimal policy yields the best value function for all states in the state space. The actor decides which actions should be taken and the critic tells the actor how good the chosen action is and what adjustments should be made. A probability distribution is used since the agent does not know how actions affect the environment. The value function is approximated using neural networks. The value functions tells us the value of the current state and value of any other state the agent may encounter.

Actor-critic combines ideas from policy and value function methods where we will keep track of two sets of parameters:

- Actor improvement :Policy parameterised by $\theta$
- Critic evaluation :Value function parameterised by $\omega$

The *actor* approximates the policy $\pi_\theta(s,a)$ and the *critic* estimates the state value function $v_\pi(s,\omega)$. The actor samples actions from the environment following policy $\pi_\theta(s,a)$.

We optimize the parameter $\theta$ where the objective function is given by $J(\theta) = E_{\tau\ p(\tau|\theta)} = [\sum_t \gamma^t r_t|\pi_\theta]$. The objective is to optimize parameters such that we maximize return.

$J(\theta)$ has shown to have high variance, as seen in REINFORCE. The variance is reduced by using a baseline, usually $V^\pi(s,\omega)$. This results in the gradient:
$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log(\pi_\theta(s,a)A(s,a)]$.

$A(s,a)$ is the advantage function $A(s,a) = Q^\pi(s,a) - V^\pi(s,\omega)$. The advantage function evaluates how much better it is to take an action than the action given by the policy $\pi_\theta$.

## 3.2 Design decisions and architecture

In our implementations we are using a single neural network for our Actor and Critic, thus 1 set of parameters are updated. The procedure of actor critic is given in the A2C algorithm below. This is adapted from the vanilla actor critic provided in the class notes.

**Algorithm** A2C Algorithm

Initialize $\theta$ and $\alpha > 0$
**for** each episode **do**
  **Initialize** s
  **for** each timestep **do**
    choose $a$ from $\pi_\theta(s,a)$
    take action $a$, observe $s'$, $r$
    $A(s,a) = r_{t+1} + \gamma V(s',\theta) - V(s,\theta)$
    $\nabla_\theta J(\theta) = [\nabla_\theta log(\pi_\theta(a|s)A(s,a)]$
    $\theta = \theta + \alpha\nabla_\theta J(\theta)$
    $s = s'$

Figure 3: A2C algorithm

The MiniHack environment has a number of observations spaces primarily pixel, symbolic, and textual observations of the screen. In addition, observations can include in-game messages, inventory information and player statistics. In our work, we utilized "glyphs", "message" and "blstats" obsverations. "Glyphs" represent the map on a 21 x 79 matrix where each entity is a unique integer $\in [0, 5991]$. "Messages" are utf-8 256-dimensional

vector encoding of the message displayed at the top of the screen. "Blstats" is a dimension vector representation of the status on the bottom of the screen. They hold information about the player's current position, health, attributes and statuses.

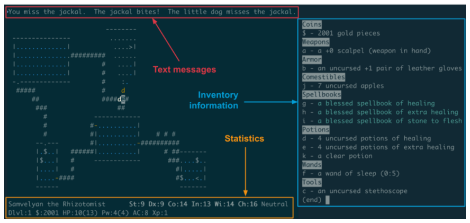


Figure 4: Screen observations

In our work, we first used "glyphs" together with "messages" as input to our architecture then later included "blstats". Reward shaping and limitation of action space were both implemented with our A2C; this will be further discussed in subsequent sections. The vector representations of the "glyphs" and "messages" were normalized, concatenated and passed to our architecture. The output of the network is a probability distribution over the action space and the estimated value function by the "critic".

| **Layers** | **Value(s)** |
|---|---|
| conv1 | Conv2d(1, 16, kernel=3) |
| maxpool1 | MaxPool2d(kernel=2, stride=2) |
| conv2 | Conv2d(16, 32, kernel=3) |
| maxpool2 | MaxPool2d(kernel=2, stride=2) |
| fc1 | Linear(in =1728, out =512) |
| fc2 | Linear(in =512, out =128) |
| message_fc | Linear(in =256, out =128) |
| blstats_fc | Linear(in=26,out=128) |
| action_fc | Linear(in =128, out =action_space) |

Table 3: A2C architecture design

The "glyphs" go through 2 convolutional layers with max pooling followed by 2 fully connected layers after being flattened. The "messages" go through 1 fully connected layer. The 2 representations are concatenated and sent through another fully connected layer. The combined features are sent through a final fully connected layer that has an output size of the action space. For each layer, ReLU activation function was used except for the output layer. To obtain the probability distribution, the Softmax activation function is used on the output layer. Due to suboptimal results, the "blstats" were also later included. "Blstats" go through 1 fully connected layer and then concatenated with the "glyphs" and "messages". Everything else remained the same.

### 3.2.1 Reward shaping and action space

The same design decisions mentioned in section 2.2.2 and 2.2.3 were implemented in our actor critic design. This resulted in sub optimal performance in our agent thus we decided to improve our reward shaping based on how much the agent explores. We concatenated the "blstats" vector with "glyphs" and "message" observation we were already using. We used "blstats" to keep track of the trajectory of the agent in the environment using a 21x79 matrix we created. This matrix is updated based on the information given by the "blstats" observation. We gave the agent a positive reward of 1 for visiting unvisited states and a negative reward of -0.2 for not exploring.

## 3.3 Hyperparameters

| Hyper-parameter | Value(s) |
|---|---|
| learning rate | 0.02 |
| num-episodes | 100 |
| max-episode-length | 1000 |
| discount factor | 0 |

Table 4: List of hyperparameters for Actor Critic

- **max-episode-length**: A2C utilizes episodic environments meaning the network weights should be updated until the environment reaches the terminal state (i.e game ends). This is not unpractical in complicated environments such as MiniHack as it may take millions of steps before the game ends. We thus reduce training time by having a maximum length for each episode.
- **discount factor**: Determines how much we care about the future reward as opposed to the immediate reward we might get.
- **num-episodes**: The number of episodes we train the A2C agent.
- **learning rate**: The ADAM optimizer is used in our A2C network.

# 4 Environments

MiniHack is a framework for creating simulated environments and is formulated from the game NetHack ([5]). To provide an interface for custom-developed RL agents and to provide communication between the agents and the game, NetHack uses the NetHack Learning Environment (NLE).

In this section, we outline the different environments from MiniHack that we used for our experiments, and we also build an understanding of the various skills each environment requires for completion.

### 4.1 MiniHack-Room-5x5-v0

This environment is set in a single square room of size $5 \times 5$ and the goal is to get to the staircase down. The are other variations of this room provided by MiniHack such as the MiniHack-Room-Random-5x5-v0, where the start and end goals are randomized, and MiniHack-Room-Monster-5x5-v0 which adds complexity by placing monsters into the environment. The environment offers a reward of +1 for reaching the goal and tests basic learning capability.

### 4.2 MiniHack-WoD-Easy-v0

The MiniHack-WoD-Easy-v0 environment is about utilizing the wand of death to zap a monster. The agent begins the task with the weapon in its inventory, which is not the case in the other variants of this environment (MiniHack-WoD-Medium-v0, MiniHack-WoD-Hard-v0, etc). This environment offers a reward of +1 for killing the minotaur (monster), it requires inventory and direction skills to complete it successfully.

### 4.3 MiniHack-LavaCross-Levitate-Potion-Inv-v0

The MiniHack-LavaCross-Levitate-Potion-Inv-v0 environment is part of the family of skills for Lava Crossing which requires the agent to get on the other side of the river by means of freezing it or levitating over it. The agent can use a wand or potion to do this, and in our environment specifically, the agent starts off with these objects in its inventory. The environment offers a reward of +1 for reaching the other side of the river and requires inventory skills to complete it successfully.

### 4.4 MiniHack-Quest-Hard-v0

This environment is a collection of most of the MiniHack environments. It requires the agent to cross a lava river, fight monsters, navigate rooms or mazes and utilize the wand of death to kill a monster before reaching the goal. The map in MiniHack-Quest-Hard-v0 is not simple (relatively) and it doesn't remain fixed. The environment offers a reward of +1 for reaching the goal and requires navigation, pick-up, inventory, and direction skills to complete it successfully.

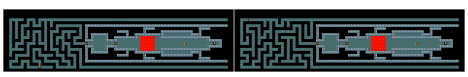

Figure 5: Quest environment [6]

## 5 Results

### 5.1 DQN

#### 5.1.1 Adam versus RMSprop

To determine which optimizing technique worked best we utilized the MiniHack-LavaCross-Levitate-Potion-Inv-v0 environment. We used reward shaping to give as much feedback as possible so that we could see which method moved fastest to convergence within 100 episodes. The results were averaged out on 10 runs.

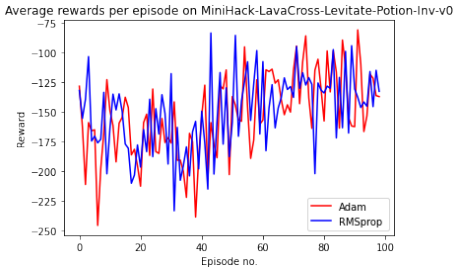


Figure 6:

The results indicate that there is no significant difference between the 2 optimizers. RMSprop does seem to have a slight advantage but it may not make such a difference as to make you consider ditching Adam completely.

#### 5.1.2 Comparisons with variants

To test if limiting the action space had a significant enough impact on our agent, we compared it against an agent that always picks random actions and a normal DQN agent that was using the whole action space. We chose the MiniHack-Room-5x5-v0 environment for this experiment.

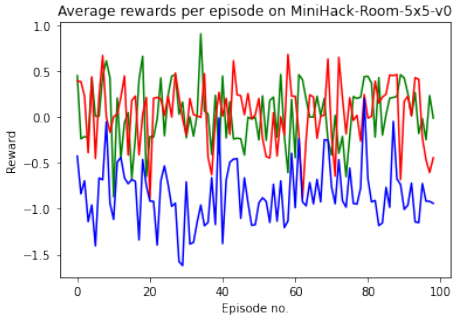


Figure 7: DQN 5x5 room

The results from the MiniHack-Room-5x5-v0 environment show that the DQN agent with limited actions (in red) outperformed the unrestricted DQN agent (in blue) consistently for the first 100 episodes of a run. We have

to note however that this is still as bad as random actions (in green) but it does hint towards the fact that limiting the amount of randomness has its merits.

### 5.1.3 Results of DQN on subtasks

We tested our DQN agent on the MiniHack-WoD-Easy-v0 and MiniHack-Room-5x5-v0 environments to see if they could accomplish the task. We ran the environments for a total 3800 episodes without any reward shaping. The agent had one run on each task and the plots show its mean reward over a 100-episode interval.

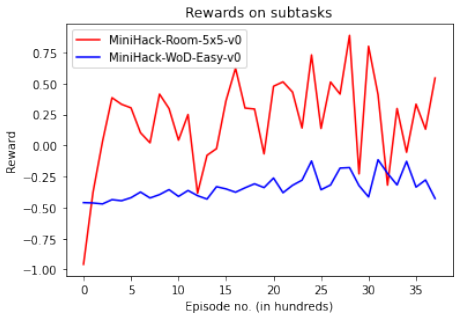


Figure 8: DQN 5x5 room

The results show that our agent did better in the MiniHack-Room-5x5-v0 environment and was able to reach the goal on a number of occasions consistently. However, the agent struggled in the MiniHack-WoD-Easy-v0 environment. The agent was not able to reach the goal at any point during the run which suggests that this environment's reward might be too sparse for the agent to learn.

### 5.1.4 Results on the MiniHack-Quest-Hard-v0 environment

We finally deployed our DQN agent with limited actions and reward shaping (as specified) into the MiniHack-Quest-Hard-v0 environment for a total of 500 episodes. In this subsection, we evaluate its performance over multiple intervals with the rewards in the plots averaged over 20 episodes.

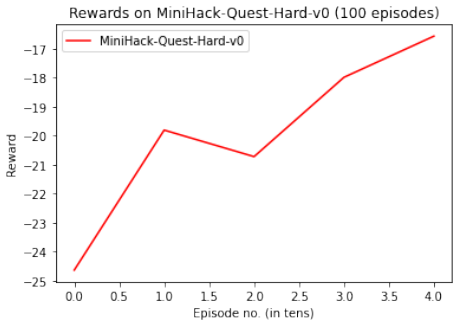


Figure 9: First 100 episodes

Within the first 100 episodes, we observed an increase in performance as the agent gradually moved from random exploratory moves to more informed moves learned through experience. By the end of this interval, the agent could get reasonably far in the maze but was still attempting to invoke weapons even though it didn't have any.

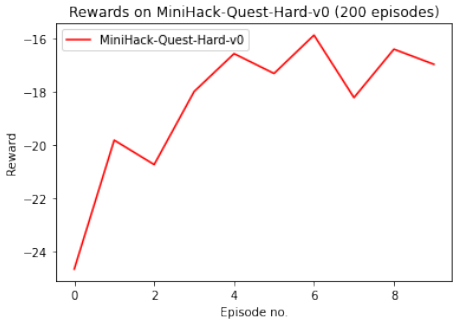


Figure 10: First 200 episodes

The next 100-episode interval saw a decline in the rate of increase of rewards per episode. The replays show that the agent started moving and exploring less, which led it to struggle to find the end of the maze.

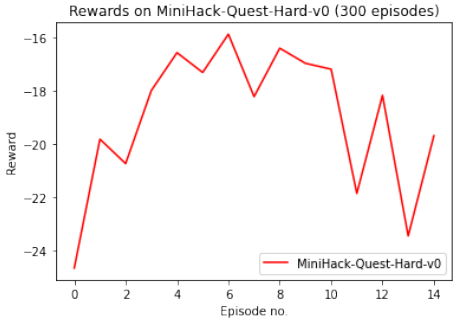


Figure 11: First 300 episodes

We then saw a major decrease in performance within the next 100 episodes as the agent at times went below

"random" performance. We define "random" as the phase in which it made random moves (in the beginning). The replays show that the agent was taking even less directional actions at this point.

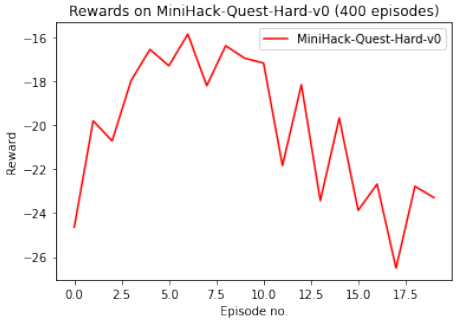


Figure 12: First 400 episodes

By episode 400 nothing much had really changed and performance was still decreasing. The trend of taking less directional actions continued.

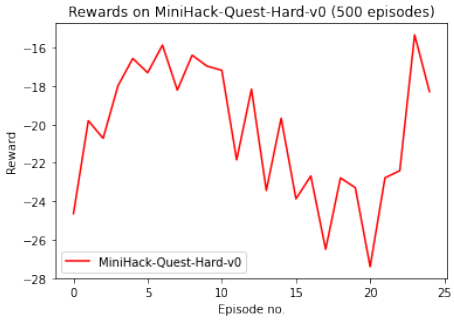


Figure 13: First 500 episodes

The interval from 400 episodes to 500 episodes saw a comeback from a decreasing trend with the rewards per episode seeing a positive uptick. We also saw the best performance in terms of reward in this interval (episode 474). The agent had learned how to pick up weapons and manage its inventory. The agent picked up a sling and club (in inventory) and received positive rewards for this. We also observed an increase in directional actions but the agent was still unable to make it out of the maze.

#### 5.1.5 Reflection

In the end, we noted that our agent struggled particularly with tasks that had sparse rewards, it also struggled when a task involved solving multiple sequential subtasks.

We observed that the agent made its biggest advance into the maze in the MiniHack-Quest-Hard-v0 environment in episode 47 when it was still exploring more but instead it received a larger reward for picking weapons and advancing less into the maze as seen in episode 474.

This suggests that the rewards given to our agent should be reviewed and adjusted. More emphasis should be put into exploring in order to solve the more complex problems optimally. To improve upon this we could enable more structured exploration as opposed to relying on random actions.

### 5.2 Actor-critic

#### 5.2.1 Sub-task

We ran the Actor-critic agent on the 5x5 room sub-task over 5 iterations using 5 different seeds. This shows that the agent does not over generalize on one specific seed. The agent fluctuates in the first 40 episodes but learns fairly well and averages with a reward of 0.6 in the later episodes. With this experiment, there is no rewards shaping or action limitations yet.

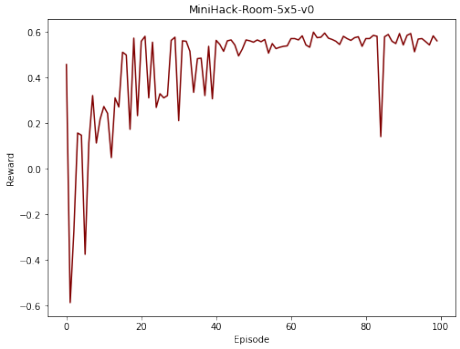


Figure 14: A2C 5x5 room

Our A2C agent seems to struggle more with the *Eat* environment than the *Room* environment. Figure 15 illustrates the rewards over 100 episodes averaged over 5 iterations with different seeds. The agent rewards are very stochastic but the agent does manage to receive positive rewards over all iterations in some episodes.

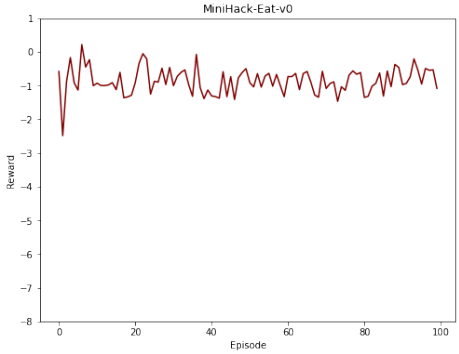


Figure 15: A2C Eat

We ran our A2C agent in the *WoD-Easy* environment over 3 iterations. The agent rarely achieves positive re-

wards and converges towards 0 withing the very last episodes.

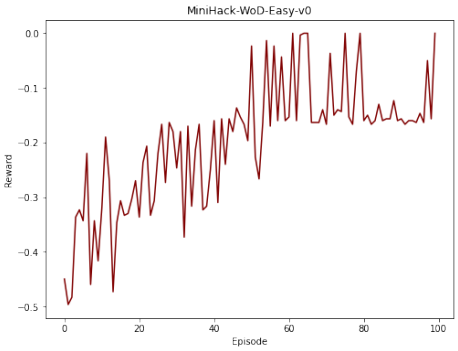


Figure 16: A2C WoD

### 5.2.2 MiniHack Quest Hard environment

We experimented with our A2C agent using multiple techniques. We utilized reward shaping and compared it to when we do not use reward shaping. Surprisingly reward shaping did not perform as well as not using reward shaping. Both these variants did not result in the agent receiving any positive rewards. In a further attempt to optimize our agent, we expanded our reward shaping by penalizing the agent for not exploring and giving it a small positive reward for exploring. This is when we included the "blstats" in our network. This made the agent perform significantly better and achieve positive rewards.

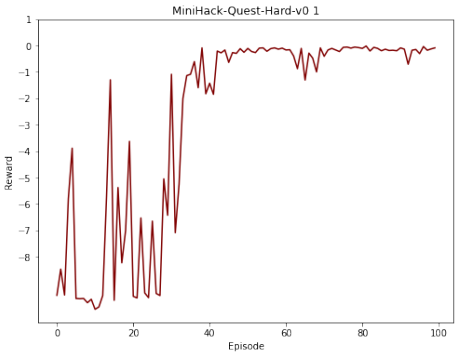


Figure 17: Quest Hard no reward shaping

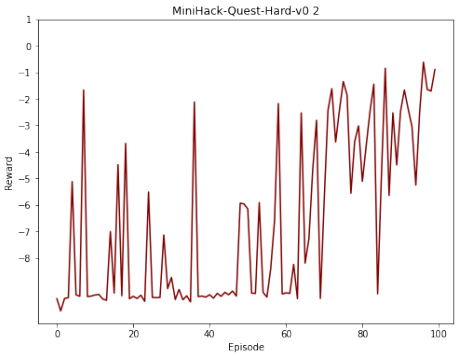


Figure 18: Quest Hard with reward shaping

We explored with the hyperparameter gamma. Figure 19 illustrates the agent with the exploration reward with gamma = 0.99. Figure 20 illustrates the agent with the exploration reward with gamma = 0. This change in the hyperparameter increased the rewards drastically.

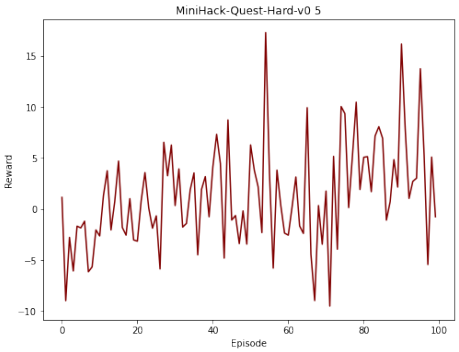


Figure 19: Quest Hard with exploration reward and gamma = 0.99

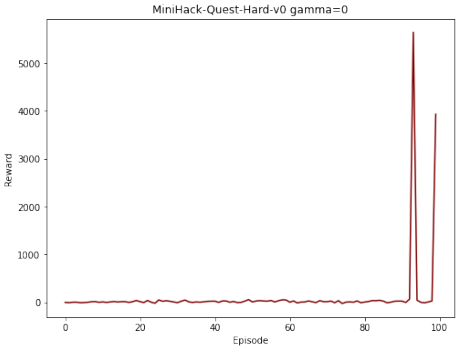


Figure 20: Quest Hard with exploration reward and gamma = 0

Our final experimentation involved action limitation. We ran our agent using the previously mentioned reward shaping implementation with *Compass direction* actions

plus certain *Command* actions. Using *Compass direction* with limited actions resulted in suboptimal performance in most of the episodes, even though the agent is able to receive high rewards in some episodes. The agent produced a really high reward on episode 33 however resulted in negative rewards in most of the other episodes.

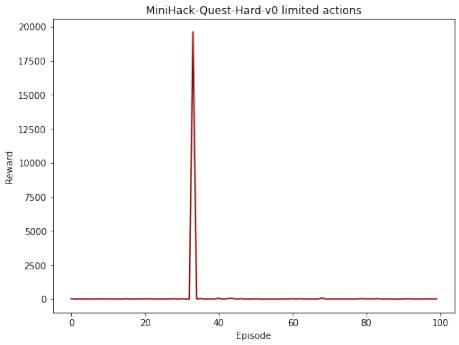


Figure 21: Quest Hard with limited actions

Our final A2C agent uses gamma = 0 with reward shaping that includes exploration reward. The action space for the agent included all the actions since limiting the actions resulted in sub-optimal results.

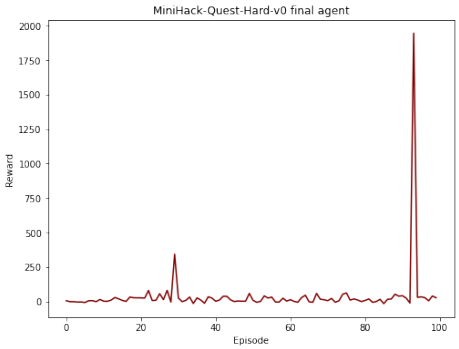


Figure 22: Final run

### 5.2.3 Reflection

Looking at the agents best runs (refer to provided videos)the agent is able to explore the maze a lot in some episodes. Based on the rewards the agent receives over the episodes, learning is not stable as the rewards are stochastic. Perhaps using a *lstm* network would help or using subtasks(options) to solve sub-goals in the Quest hard environment.

## 6 Comparison plots

As the results show, the A2C agent consistently outperformed the DQN agent, and it was mainly due to the reward for exploration given to the A2C agent.

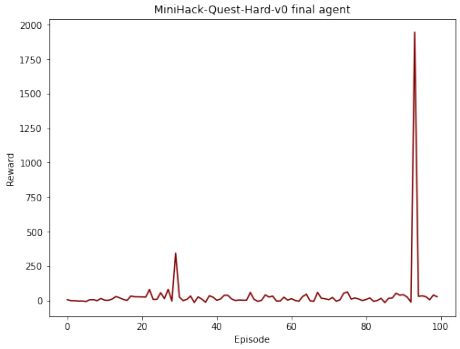


Figure 23: Final run A2C

We also noted that the A2C model was faster and easier to train. We thus believe it is the most computationally efficient.

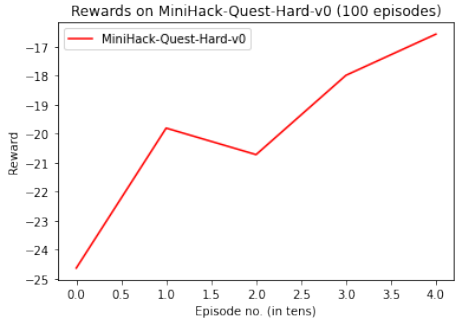


Figure 24: Final run DQN

## References

[1] Brownlee, J.: Gentle introduction to the adam optimization algorithm for deep learning (2021), URL https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[2] Brownlee, J.: Gradient descent with rmsprop from scratch (2021), URL https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[3] Gómez, Y.: Pros and cons of some machine learning algorithms (2020), URL https://www.linkedin.com/pulse/pros-cons-some-machine-learning-algorithms-yulieth-zulC3%B3mez/?trk=portfolio_article-card_title

[4] Samvelyan, M., Kirk, R., Kurin, V., Parker-Holder, J., Jiang, M., Hambro, E., Petroni, F., Kuttler, H., Grefenstette, E., Rocktäschel, T.: Action spaces (2021), URL https://minihack.readthedocs.io/en/latest/getting-started/action_spaces.html

[5] Samvelyan, M., Kirk, R., Kurin, V., Parker-Holder, J., Jiang, M., Hambro, E., Petroni, F., Kuttler, H., Grefenstette, E., Rocktäschel, T.: 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), URL https://openreview.net/forum?id=skFwlyefkWJ

[6] Samvelyan, M., Kirk, R., Kurin, V., Parker-Holder, J., Jiang, M., Hambro, E., Petroni, F., Kuttler, H., Grefenstette, E., Rocktäschel, T.: Quest (2021), URL https://minihack.readthedocs.io/en/latest/envs/skills/quest.html

[7] Wiewiora, E.: Reward shaping. (2010)