Reinforcement Learning on Minihack the Planet

Sheslin Naidoo
Nico Jared Govindsamy
Mohammed Gathoo

11 November 2022

Contents

1 Abstract

In this paper, we will investigate the performance of two popular RL models on the MiniHack Sandbox Framework environment. Specifically, we will be looking at Deep Q-Networks (DQN) and Advantage Actor Critic (A2C). In this paper, we provide an in-depth overview of our agents evaluated on the MiniHack Quest-Hard environment. The github repository for the code can be found here: github repository

2 MiniHack Environment

2.1 Observation Space

The MiniHack environment has several observation options. We have chosen to use three of the provided options: glyphs, pixel and message.

A glyph is described as a 21×79 matrix of glyphs (IDs of entities) on the map. Each glyph represents an entirely unique entity. We have chosen this option as the fundamental observation space for our models, meaning that we use glyphs to represent our state space.

The pixel option provides a representation of the current screen in image form. This observation is used to view our agent on our environment by generating videos using this pixel option.

Message is the utf-8 encoding of the on-screen message displayed at the top of the screen. We decided to add this observation to our models, as it may contain useful information to help our agents learn the environment, as well as provide insight on certain states where our agent is asked for confirmation.

2.2 Action Space

As with the observation space, the MiniHack action space has a wide variety of actions, most of which would not be needed by our agent to navigate the Quest-Hard environment. We thus only chose actions that would be needed by our agent in order to decrease training time as well as the complexity of our model (more actions leads to a more complex model which is difficult to train).

Action	Discussion
N	Movement action - North
E	Movement action - East
S	Movement action - South
W	Movement action - West
NW	Movement action - North-West
NE	Movement action - North-East
SW	Movement action - South-West
SE	Movement action - South-East
PICKUP	Pickup an item - needed to pick up food and item to cross lava
EAT	Eat item - restores HP
APPLY	Apply an items special ability - needed to kill monster at the end of quest-hard
ZAP	Zap your wand - needed to use the wand of cold to cross the lava river
PUTON	Put on armour
QUAFF	Drink - Needed to drink potions - used to cross the lava river
WIELD	Use your weapon - needed to fight monsters
OPEN	Open a door - needed to open the various doors in quest-hard
RUSH	Used upon message/request to make selection - not meant to be used as an in-game action

The table above lists the actions that we considered as well as a short discussion on why this action is needed.

2.3 Rewards and Reward Shaping

For the quest hard environment, our agent receives a reward of 1 for reaching the staircase at the end of the game, and is rewarded a small negative reward for frozen steps. We thus decided to add several rewards to help the agent reach the end of the maze in Quest-Hard-v0.

Reward	Description
-2	Entering the corridors at the top and bottom of the screen
10	Reaching the door at the end of the maze and the end of floor 2
0.1	Exploring the maze - change in glyphs state
-1	Entering the lava and dying
1	Eating an apple - used to prevent reward glitch

3 DQN

DQN combines the off-policy temporal difference control algorithm with neural networks in order to estimate the state-value function.

Actions are chosen either randomly or based on a policy. In our implementation, we use an ε-greedy policy where we calculate the threshold from the eps-start and eps-end values. If our random value is within the threshold, then we use that random action, otherwise, we take the action chosen by our policy. We then sample the environment based on the action chosen. We record the results in the replay memory and also run optimization step on every iteration. Optimization picks a random batch from the replay memory to do training of the new policy. Older target network is also used in optimization to compute the expected Q values; it is updated occasionally to keep it current.

Our DQN implementation is a mixture of the RAIL Lab [rai] implementation and the DQN implementation provided in the Github repository *qfettes/DeepRL-Tutorials* [tut]. We have modified this implementation to incorporate Prioritised Experience Replay (PER) as well as reward shaping.

3.1 Prioritised Experience Replay

In the conventional DQN replay memory, we randomly sample experiences with a linear distribution which means we only need one container to store these experiences. On the other hand PER needs to associate every experience with additional information(priority and probability in our case).

Our priority is updated after a forward pass of the Neural Network using the loss obtained from this step. The probabilities is then computed from the priorities. We use SumTrees to store the priority values in order to make sampling more efficient Schaul *et al.* [2015].

3.2 DQN Network Architecture

The agent is initialized with two DQNs - the online model and the target model. They are both exactly the same but are used differently. The input of the network is the state observation, which is in the glyph format explained before. We then apply 2D convolution at 3 layers before passing the results to two fully-connected layers. Each output from the convolution layers and the first linear layer is passed to the Tanh activation function except the final layer which is passed through a Softmax layer. That result represents the probability of taking any one of the available actions.

Below is a detailed breakdown of our DQN architecture:

- 2D Convolutional Layer 1 (in_channels=1, out_channels=16, kernel_size=4, stride=1)
- 2D Convolutional Layer 2 (in_channels=16, out_channels=32, kernel_size=3, stride=2)
- 2D Convolutional Layer 3 (in_channels=32, out_channels=32, kernel_size=3, stride=2)
- Fully-connected Layer 1 (in_features=1728, out_features=100)
- Fully-connected Layer 2 (in_features=100, out_features=num_actions)

3.3 Hyperparameters

1. Gamma (discount factor) - 0.99
2. Learning rate (used by network optimizer) - 0.001
3. Number of steps - 1e6
4. Replay buffer size - 1e6 - This is the total number of transitions that can be stored in our replay buffer.
5. Batch size - 64 - the number of transitions sampled from the replay buffer every time we update the online network.
6. ε–greedy parameters - These parameters define our ε-greedy approach. eps-start = 1.0 is the starting epsilon value and eps-end = 0.1 is when we stop annealing epsilon. eps-fraction = 0.6 is the rate at which epsilon is annealed (decreased). This is how we get the model to explore less as time passes and how we get to choose at what rate it explores less.
7. Target update frequency - 1000 time steps - the rate at which we update the target network from the online network.

Instead of stipulating the maximum number of episodes, this model instead worked with the maximum number of steps. Episode counting was implicit because we let an episode run its full course (or until the environment built-in maximum episode length was reached), and this took a variable number of steps. Since we can learn from every transition/step, specifying the number of steps was thought to be more useful.

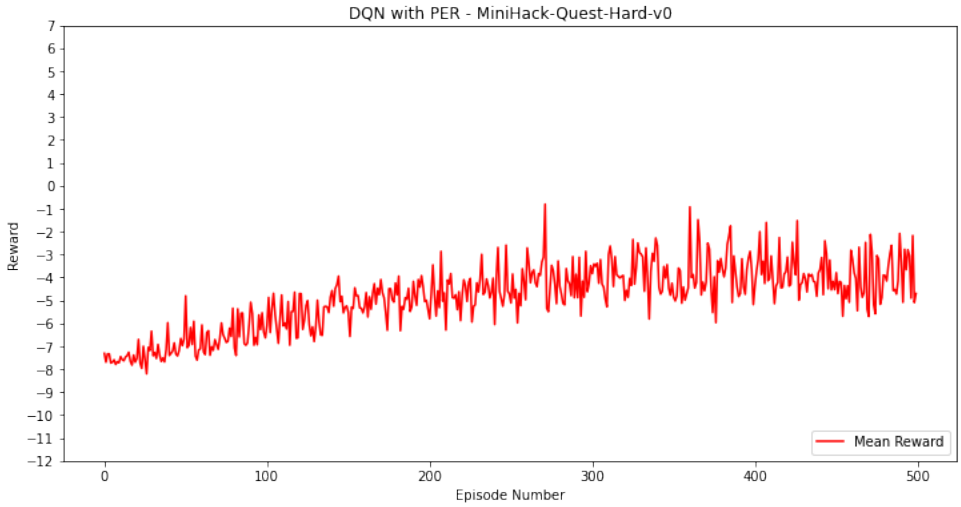
3.4 Training

Our DQN agent with prioritised experience replay was initially trained on a two other environments - *MiniHack-ExploreMaze-Easy-v0* and *MiniHack-Quest-Easy-v0* - before we trained it on *MiniHack-Quest-Hard-v0*. We also trained it on *MiniHack-Quest-Hard-v0* alone. Experimentally, we found that training it only on *MiniHack-Quest-Hard-v0* proved no difference to training it on those other environments initially. Various of the hyperparameters were varied, and the hyperparameters stated in the previous section proved to be the best from all the variations we explored. Training sometimes involved running our model for 500000 steps then saving the model, and using the same model as the initial model for another iteration of 500000 steps. The process of training is outlined below to a partial degree of detail. For each time step:

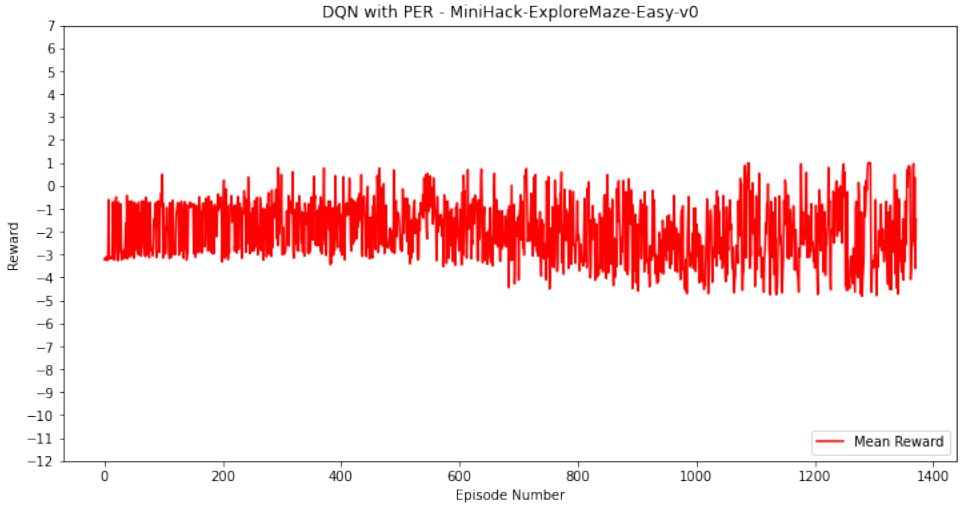
- Calculate the annealed value of epsilon. Sample a random probability value between 0 and 1, then use the ε–greedy approach to pick the next action. If the next action is picked from the model, we perform a forward pass for the current state (in glyph format) in the online network.
- After obtaining our next action, we take a step in the environment, obtain the reward and next state, and then store that entire transition (state, action, reward, next_state, float(done)) into the replay buffer.
- We then update our networks. We begin by preparing a mini-batch of transitions sampled from the replay buffer. After doing so, we compute the loss from our current q-values obtained from the sampled mini-batch and the expected q-values from our target network. Mean Squared Error is used. At this point, we also update the priorities of our replay buffer. We then update our online network by updating our parameters using the Adam optimizer.
- The target network is updated based off the target update frequency hyperparameter, and it just involves loading the parameters of the online model into the target model when the update occurs.
- If the episode ends, indicated by the done variable returned by the environment, then the environment is reset.

3.5 Results

A plot showing the average reward received by our agent is shown below.



From this run, which is for 500 episodes, approximately 500000 steps, it is evident that the DQN agent begins to learn for the first 300 episodes, but then converges to a reward just below -2. This agent isn't capable of solving the maze at the start. With regards to sub-goals, we trained the model on the *MiniHack-ExploreMaze-Easy-v0* environment. Below is the plot for that.



The same convergent behaviour is demonstrated in this environment as well. It does not learn to explore the maze.

3.6 DQN with PER Evaluation

In summary, the DQN agent modified with Prioritised Experience Replay (PER), is unable to get past the maze at the beginning of the Quest-Hard environment. The agent is unable to learn the required actions for navigating the maze. It would seem that the agent is unable to differentiate between observations that are similar because it has a problem of repeating actions for similar or the same states, which results in poor exploration, especially when stuck against a wall - it repeats the action that got it stuck in the first place.

This sort of behaviour is evident despite changing our DQN architecture in different ways, like adding more or less convolutional and fully-connected layers, or changing activation functions and parameters of these layers, etc. The problems may lie with the decisions made in the training process or the length of training. Perhaps this model requires a much longer time to train, which requires resources we don't have access to. We have tried training the model in iterations - training the model for 500,000 steps, saving the model, loading the network parameters during the next iteration of training and running for 500,000 steps again, but this did not prove useful for some reason as the model still started from low rewards again and again converged to the similar peek rewards achieved in the previous iteration of training.

4 A2C

Policy gradient methods are a subclass of policy-based methods which aims to optimize the policy directly without need for a value function. It does this by estimating the direction of the steepest increase of the returns. This is called the Policy Gradient.

$$\nabla_\theta J(\theta) = \sum_{t=0} \nabla_\theta log \pi_\theta(a_t|s_t) R(\tau)$$

Direction of the steepest increase of the (log) probability of selecting action at from state st

Cumulative reward

The main advantages of Policy Gradient methods are that they can estimate the policy directly and are able to learn a stochastic policy, whereas value functions cannot. This means that since we have a probability distribution over our actions, our agent would thus not exploit an action, and would rather explore the state space. We also dismiss the problem of perceptual aliasing, which occurs when two states look the same but have some variations which our model cannot identify.

Advantage Actor Critic (A2C) is a hybrid architecture that combines value-based and policy-based methods. Specifically, we have an actor which controls how our agent behaves, and a critic which measures how good our chosen action is. Our A2C method uses two function approximations, namely, a policy function parameterized by theta for the actor, and a value function parameterized by w for our critic. The main difference between A2C and conventional AC is that we use an Advantage function as the critic instead of the action value function. The main idea here is that our advantage function calculates how much better it is to take the action chosen, compared to the average value of that state.

$$A(s,a) = Q(s,a) - V(s)$$

$$r + \gamma V(s')$$

$$A(s,a) = r + \gamma V(s') - V(s)$$

TD Error

This method requires both the state value function V(s) and the state-action value function Q(s,a), therefore we would instead use the TD Error as an estimator for this advantage function. The A2C algorithm used is shown below:

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

Our A2C implementation is loosely based off that of the Rail Lab implementation [rai].

4.1 A2C Network Architecture

We use a single neural network for both the actor and the critic in our model, with the difference only in the output layer where the actor neural network estimates the value function of the state, and thus has one output neuron, whereas the critic would have output dimension equal to the number of actions, as the critic network computes the probability of taking each action.

We use a CNN architecture as it takes in a 2D representation as input and is a popular choice for training Reinforcement Learning agents on video games. Our actual network architecture was based on the also popular LeNet 5 architecture. A detailed breakdown of our Network architecture is listed below:

- 2D Convolutional Layer 1 (in_channels=1, out_channels=20, kernel_size=5)
- 2D Max Pooling (kernel_size=2, stride=2)
- 2D Convolutional Layer 2 (in_channels=16, out_channels=32, kernel_size=3, stride=2)
- 2D Max Pooling (kernel_size=2, stride=2)
- Fully Connected Layer 1
- Fully Connected Layer 2
- Fully Connected Layer 3 - Used for message input
- Fully Connected Layer 4 - Used to take an a combined input of both message and glyph
- Actor Output Layer - Fully connected Layer
- Critic Output Layer - Fully Connected Layer

We use the ReLu activation function between all layers except the final layer where we use the Softmax activation function. We use ReLu as we need real values output between layers. The system was also tested using a Tanh activation function which showed no improvement. We use the Softmax function on the critic output in order to get the action probabilities.

4.2 Hyperparameters

1. Gamma - This is the Discount factor used to calculate the returns of an episode. We use a value 0.99.

2. Learning Rate - The learning rate used by our Adam Optimizer. We use a value 0.02.
3. Number of Episodes - Number of episodes that we train our agent for.
4. Max Episode Length - In the conventional A2C algorithm, we would run our agent on an episode until it reaches the terminal state, however this is not feasible for training purposes, and thus we limit this to a finite number - for the quest hard environment, we use a value of 1000.

For ease of training, we run our model on batches of 500 epochs and save the model. We then reload the saved model for our next batch run.

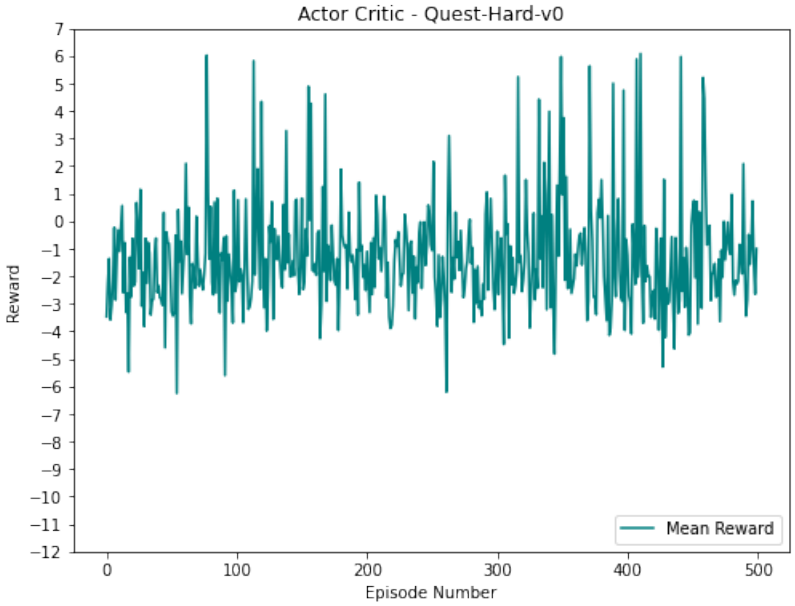
4.3 Training

For the training process of our A2C model, we initially trained our model using the quest-hard-v0 environment, however we noticed that our model struggled to find meaningful actions. We thus decided to train our model on simpler environments such as *MiniHack-MazeWalk-15x15-v0*; *MiniHack-Eat-v0*; *MiniHack-LavaCross-v0* and *MiniHack-WoD-Easy-v0*. We note however, that the model performed better from being trained on quest-hard-v0 first. We notice that the model trained on quest hard only figures out how to navigate the maze on floor 1.

4.4 Results

From our experiments, we see that our agent is able to navigate the maze on floor 1 and open the door to the next room occasionally, however, it is not able to cross the lava river in the next room. Videos of our agent on the Quest-Hard environment can be found at Git

A plot showing the average reward achieved by our agent is shown below.

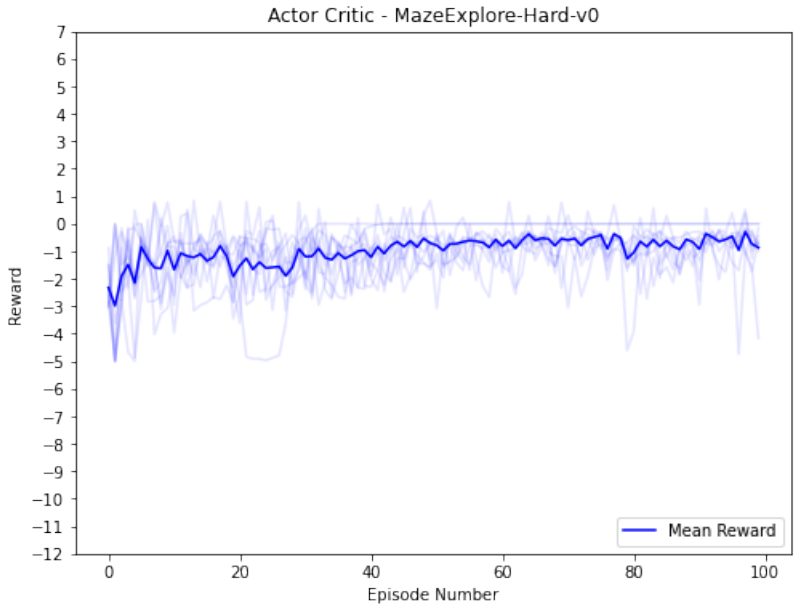


We see from the plot above that our agent occasionally received a high positive reward. This is the reward associated with exploring the maze. A positive reward here is shown to be achieved in the episodes where the agent has almost completely explored the maze

4.4.1 Sub-Goals

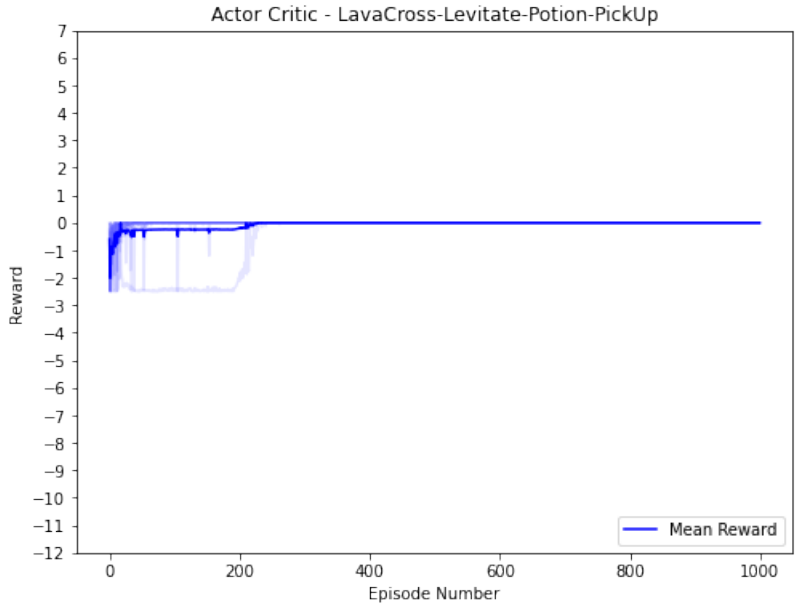
To see how well we achieves sub-goals, we tested our agent on a different environment for each sub-goal.

4.4.1.1 Reaching maze exit To test whether our agent reaches the exit of the maze, we test our agent on the *MiniHack-MazeWalk-15x15-v0* environment. A plot of the rewards achieved by our agent averaged over several runs is shown below.



From the plot above, we see that our agent does learn to navigate the the maze, however it does not do this effectively. We do occasionally see positive rewards, however it is not able to complete the maze.

4.4.1.2 Crossing the Lava River We trained our agent on the *LavaCross-Levitate-Potion-PickUp* environment, where the agent would have to pickup a levitation potion, apply it, confirm use and walk over the lava river. We note that this is quite a complicated task. To help our agent learn faster, we provide a reward of -1 for dying in the lava and $+1$ for crossing the lava and not dying(non-terminal). A plot of the rewards achieved by our agent on this environment is shown below.



From the plot, we see that our agent quickly learns that it needs to cross the lava, but cannot figure out the commands needed to do so, which causes our agent to continually die in the lava.

4.5 A2C Evaluation

Overall, even though our A2C agent performs better than the DQN agent, it is still not able to get past the maze on the floor 1. We conclude that our agent is not able to learn complicated strings of actions to complete a task, e.g., in order to cast a spell, the agent would have to execute the ZAP action to which the system would response with a confirmation message asking ”what would you like to zap?”, however the glyphs would remain the same, as there is no change in the layout of the environment. The agent would then have to select the RUSH command (corresponds to a confirmation command), which would already have a low probability of being selected by our agent as it is an otherwise useless action.

We also note that the environment has several glitches, examples the reward glitch, where using the Nethack RewardManager causes the environment to terminate after executing one action. A random reward event must be added to prevent this glitch. Another issue is that sometimes, our agent is given one of the coordinate rewards at the beginning of the episode instead of when our agent actually reaches that position, which is effecting training.

Due the these reward glitches, as well as the complexity of the environment, our agent is not able to learn effectively and thus is not able to reach the end of the maze in the *Quest-Hard-V0* environment.

5 Comparison of Agents

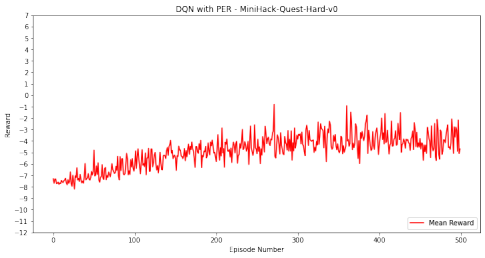
Looking at the results produced for both agents, we can see that while the DQN agent begins achieving lower rewards per episode at around -8, we see a gradual improvement of rewards as the episodes go on, increasing to an average reward just below -2 with a maximum of 0 in several episodes. This indicates the learning of the environment by the DQN agent.

A2C, however, displays a more erratic reward pattern throughout the 500 episodes of training compared to the DQN agent. Starting at a higher average episode reward, we see the A2C agent obtaining higher rewards

more frequently with an average episode reward of -1, an average higher than what we see the DQN agent achieving near the end of its training.

From this, we can see that while DQN shows to improve it's rewards at the end of training, its average reward thereafter is still considerably lower than that of the A2C agent at its worst performance, and never achieves a reward higher.

Training was also done on a separate maze exploration environment to see how effective each agent is. We again see a difference in the performance of the two agents where the A2C agent learns to navigate the maze but ineffectively and doesn't reach the goal. This results in an initial improvement in training but quickly plateauing to an average reward of -1. On the other hand, the DQN agent does not learn to explore the maze at all.

[DQN Agent] 

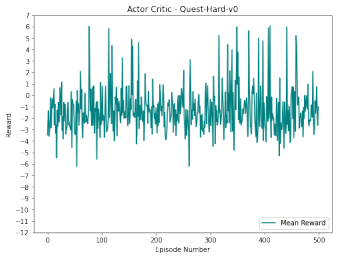
[A2C Agent] 

Figure 1: Plots of each agent rewards trained on Quest Hard

6 Future work

For future work in this environment, we suggest an hierachical method that involves options in order to acquire the necessary skills to complete tasks such as lava crossing or killing the monster. We also suggest a deeper look into the environments rewards and transition function.

7 Conclusion

This report provides a detailed comparison of two popular Reinforcement methods on the Minihack environment. Overall we note that neither of our agents were able to complete the Quest-Hard environments, however we do note that the A2C agent performed significantly better than the DQN agent, with our A2C agent being able to successfully navigate the maze in the Quest-Hard environment. We note that the agents poor performance is partly due to the instability of the MiniHack Environment, as even modifications such as reward shaping and restricting the number of actions has little effect on the rewards received by the agent.

References

[Git] *A link to the Github Repository containing our code and videos*. https://github.com/Scotts-Bots/Reinforcement-Learning-Project-2022.

[rai] *A link to the Rail Lab Repository containing*. https://github.com/raillab/.

[Schaul *et al.* 2015] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. arXiv, 2015.

[tut] *A link to the DeepRL-Tutorials Github Repository containing*. https://github.com/qfettes/DeepRL-Tutorials.