Reinforcement Learning Assignment 2023 MiniHack the Planet : Using DQN and Actor Critic

Thato Mahlatji 473561 Shivaan Govender 712019
Alexandra Barry 1056862 Kayleize Govender 1821334

November 2, 2023

1 DQN

1.1 Algorithm Description

The Deep Q-Network (DQN) algorithm is a deep learning algorithm for reinforcement learning that combines Q-learning and deep neural networks to achieve impressive results in a variety of Atari 2600 games, using only raw pixels as input [Mnih *et al.* 2015]. The DQN model consists of various components that interact in order to learn an optimal policy that will maximize the total rewards. The Policy network is a neural network responsible for approximating the Q-function that provides an estimate of the returns for taking various actions in certain states. The Target Network is a essentially a copy of the Policy network with "frozen" weights. This is to maintain stability during training.

The replay buffer is used to store previous interactions or experiences as a tuple (state, action, reward, next state, done). These experiences are sampled randomly in small batches to train the Policy Network. The training process for a DQN Agent involves using the agent to iteratively interact with the environment by using the state and the epsilon greedy policy to choose an action. This is then passed to the environment which returns the next state, reward and whether the target was reached. The state, action, reward, next state and done values are then stored in the replay buffer to be sampled later on. The agent is encouraged to explore more during the first 1000 steps. After 1000 steps the network is optimized whereby the agent will collect a batch of 256 samples from the replay buffer and use those to minimize the TD error. After every five iterations and the target network is updated to match the policy network to ensure convergence.

1.2 Design Decisions

Given that the baseline implementation was developed in the DQN lab as well as the RAIL lab [RAIL-Lab 2018], the main design decisions involved constructing the neural network architecture, augmenting the input received from the environment and shaping the rewards as well as fine tuning the hyperparameters. The Neural Network processed a stacked tensor of two state items(glyphs and chars). The network consisted of convolutional layers and fully connected layers. The first convolutional layer consists of 16 filters with a kernel size of 3 followed by a ReLU activation. Then a max-pooling layer with 3x3 kernel size is used to reduce the spatial dimensions. The second convolutional layer then uses 64 filters and ReLU activation similar to the first. This is then passed to another max pooling layer. After the convolutional layers, the resulting data is flattened to be passed through the fully connected layers which uses ReLU activations. Using Batch normalization allowed for more stablized training. The agent first trained with the standard rewards as given by the environment. Since the reward for collisions is -0.01, the agent struggled to learn to avoid collisions. Thus the reward function was shaped to return a reward of -1 for collisions, 0.1 for any other action and 1.5 for reaching the target.

1.3 Hyperparameters

Hyperparameters are the parameters that regulate the agent's behaviour and the training process. The values and descriptions of the hyperparameters are provided in Table 1.

Table 1: DQN Hyperparameters

Hyperparameter	Description	Value
replay-buffer-size	The max number of samples the replay buffer can hold	5000
learning-rate	Learning rate for the Adam optimizer	0.0001
discount-factor	Discount factor for future returns	0.99
num_steps	Number of steps to run the environment for	100000
batch-size	Number of transitions to optimize at a time	256
learning-starts	Number of steps before learning starts	1000
learning-freq	Number of iterations between every optimization step	1
use-double-dqn	Use double deep Q-learning (boolean)	true
target-update-freq	Number of iterations between every target network update	5
eps-start	e-greedy start threshold	1.0
eps-end	e-greedy end threshold	0.01
eps-fraction	fraction of num-steps	0.1

The neural network's learning rate determines how quickly the neural network learns. A low learning rate might result in the network taking too long to learn, whereas a high learning rate can cause the network to overfit the training set. The discount factor regulates the relative importance of future benefits in comparison to present gains. It's crucial to understand that there isn't a universally applicable set of hyperparameters for DQN. The ideal hyperparameter values will differ based on the particular environment and task at hand.

1.4 Tasks

MiniHack is a sandbox framework designed to facilitate the creation of new reinforcement learning environments and enhance existing ones, making it easier to define a diverse range of challenging tasks and access a broad set of entities interacting in a complex environment [Samvelyan *et al.* 2021]. The foundation of MiniHack consists of description files that define worlds that are built procedurally using the domain-specific language (DSL) of the NetHack game. NetHack's complexity makes it a great RL research platform since it enables researchers to investigate a variety of issues, including planning, skill learning, exploration, and language-conditioned RL Küttler *et al.* [2020]. At the same time, NetHack is computationally efficient, reducing the computational resources needed to accumulate extensive experience. The NetHack Learning Environment (NLE) is a fully-featured Gym environment that balances complexity and speed.

MiniHack's navigation tasks test the agent's ability to get to the desired place while navigating through challenging environments, fighting monsters in corridors, crossing rivers, and more. The tasks investigated using the DQN method are:

- Navigation Room - 5x5
- Navigation Room - 15x15
- Skill Acquisition - Eat
- Navigation - Quest Easy

Figure 1 illustrates the tasks investigated using the DQN method.

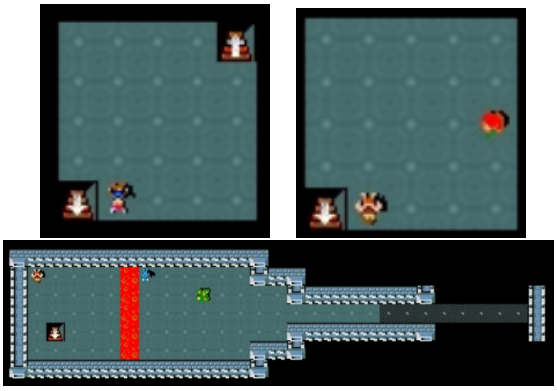

Figure 1: The top-left image illustrates the environment for the 5x5 navigation room, the top-right illustrates the image of the skill acquisition task and the bottom image illustrates the navigation quest-easy environment

1.5 Sub Task Results

Figure 2 shows the mean 100 episode reward for the Deep Q-Network (DQN) agents on MiniHack tasks. The DQN agents were trained for a 1000 episodes on each task, and the mean 100 reward is plotted.

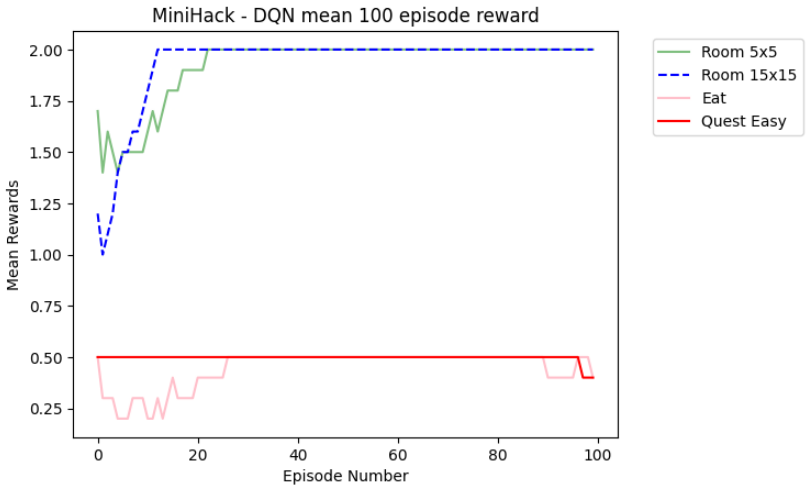

Figure 2: DQN mean 100 episode reward

The results show that DQN agents are able to learn to solve only the navigation room task of the MiniHack tasks. It failed to solve the eat and Quest easy tasks. The failed attempts to solving the Eat and Quest-Easy subtasks can be attributed to the available actions provided to the agent and the handling of the commands. Upon review of the trained agent, it was aparent that the agent would often find the apple but not know what to do next, since the EAT action was supplied it could mean that the agent had to confirm it completed the task which would result in it being rewarded. For Quest-Easy, the agent automatically dies when stepping into the lava, this could also correlate to the limited or overflow of actions provided to the agent. The Quest-Easy also consisted of a more complex environment structure in contrast to the room sub-tasks which made indicate that the DQN requires more complex layers to produce a better estimated action for more challenging environments. It may also be necessary to allow the agent to interact for a longer duration with the environment in the case of Quest-Easy. Increasing the current exploration time for more than 1000 steps may help the agent understand the dynamics of the environment better.

1.6 Potential Improvements

- Use atleast two recent frames in the DQN to provide the agent with a more complete picture of the environment, which can be helpful for skill acquisition tasks that require understanding the context of the current

state.

- Reduce over-fitting by sampling from a prioritized replay buffer instead of the standard replay buffer that samples from a normal distribution of past experiences.
- Encourage more action exploration using a more appropriate reward shaping function or optimistic initialization.
- Provide more interaction time with the environment depending on the complexity of the environment.

2 Actor Critic with Hierarchical Options

For the second algorithm in our assignment we have decided to implement an Actor-Critic method, to train the Agent on individual sub-tasks and then use the concept of Hierarchical Options Framework to combine these learnt skills ('options') to be used in more complex environments. We like this approach as the Quest-Hard environment in MiniHack can conceptually be seen as completing a lot of smaller less complex tasks, such as navigation, consuming items, fighting monster etc rather than one big complicated one. Hence we felt that training the model in a similar vein could obtain the best results. With this method you are also not limited you to only using one architecture for training the Agent. You can rather tailor your approach for each specific sub-task to generate the underlying policies that achieve the optimal results on each respective skill.

2.1 Options Framework

The Options Framework is one way of implementing Hierarchical Reinforcement Learning; which is a method of simplifying both the learning and planning process in complex RL problem by introducing a hierarchy of policies. I.e. decomposing a complicated problem into smaller sub-tasks.

The Options Framework consists of a high-level policy that selects options (skills) and a low-level policy that executes them. An option can be seen as an action that is executed over many time steps (like a mini-policy that can be run insides a bigger policy).

An option consists of 3 components:

- Initiation Set: Set of states where you can run the option from.
- Termination Condition: The option terminates once you get to that state (can be probabilistic).
- Option Policy: What to do at every step along the way.

When implementing the options framework instead of having an actions space you now have a set of options, some of which may be primitive actions (where primitive actions can just be considered as an options that runs for 1 step and that can be run from anywhere).

When you do this, the problem changes from an MDP (Markov Decision Process) to a Semi-MDP. This is very similar to MDPs with the main difference being that the transition model and reward function now become temporal.

The main benefits for us deciding to implement this framework for this assignment being:

- Modularity: Designing and training the Agent becomes easier by breaking down complex tasks into smaller, more manageable sub-tasks (options).
- Sparse Rewards: Options help address the challenge of sparse reward problems, as the agent can receive rewards when options are executed, making it easier to learn in scenarios where reward signals are infrequent.
- Transfer Learning: The policies learnt in the smaller environments can be transfer into the more complicated ones.

2.2 Actor Critic

We liked this method for training the Agent on the sub-task due to the fact that it combines the benefits of both policy-based and value-based approaches and felt that it could perform well in the MiniHack environment because of its ability to learn policies in complex environments whilst maintaining a good balance between exploration and exploitation.

The Actor-Critic architecture consists of two key components:

1. Actor: The actor approximates the policy, and hence is responsible for selecting the actions that the agent should take for a given state. It does this by learning to choose actions that maximize the expected cumulative reward over time.
2. Critic: The critic is a value (or state-action value) function estimator. It evaluates the actions taken by the actor and helps it learn by estimating the values of the respective states.

In essence; the actor and the critic work together in a constant loop, with the actor exploring different actions and the critic using those actions to update its estimate of the value functions.

Besides for being good at balancing exploration and exploitation, actor-critic methods do have other properties which we think make it a good candidate for handling the MiniHack environment:

1. Transfer Learning: You can pre-train the model on a related task, and the knowledge acquired can then be applied to a target task. Thus we can accelerate the learning in the more complex environments by first training the model in similar, less complex environments.
2. Sample efficient: Doesn't require many interactions with the environment to learn a good policy.
3. Generalization: They tend generalize from the state and action spaces they have experienced during training. Meaning that they can adapt well to similar but previous unseen situations. This can be beneficial for the randomness of the MiniHack environment.

These algorithms do however tend to have a few drawbacks. The main two being that; slow convergence and hyper-parameter sensitivity.

2.3 Algorithm Architecture

A single end-to-end Convolutional Neural Network was used to implement the A2C architecture. The network was designed to take in the state representation of the environment via the input layer and have a shared final layer to output a probability distributions over the action space (actor) as well as a single node to represent the value function estimation (critic). Combining the actor and critic into a single network provides the benefit of only having to optimize one set of parameters during the training process.

2.3.1 CNN Network

The design for the networks architecture was based of the code found in Budler [2021]. We decided to use this code as a starting point so that we could rather build upon what they built and focus on creating the hierarchical options framework to put it all together. We felt like this would be a better approach to tackling the Quest-Hard environment.

2.3.2 Input Layer

The whole input layer to the network can be broken down into 4 separate inputs, each providing the network with different contextual information. This should in turn provide the Agent with enough semantic data to allow it to make the best possible solutions. The 4 sources of information are:

1. Glpyhs: a 21 x 79 matrix of glyphs on the map. Each glyph represents an entirely unique entity, so these are integers between 0 and 5991. In the standard terminal-based view of NetHack, these glyphs are represented by characters, with colours and other possible visual features. This data is passed through two sets of convolutional and max pooling layers followed by a two fully connected layers. Budler [2021]

2. Messages: A utf-8 encoding of the on-screen message displayed at the top of the screen. This is used to provide the user with contextual information on what is happening in the environment.

3. Neighborhood Descriptions: This is a 3 x 3 matrix of what items are around the character. This was encoded to integers so that it could be passed through to the network.

4. Search for Item Direction: This returns the cardinal direction of an specified item if it is in the neighboring pixels on the character. This was passed through a one-hot encoder and fed through to the network as a 9x1 vector.

2.3.3 Coding Option Policies

To include the use of Options in our network training algorithms, several adjustments to the architecture were made. In the original training algorithm, the

set of available actions in the specified environment is used. To add the additional sub-policies as Options, a new set of environment options was defined and passed into the training algorithm. The structure of this follows the structure:

```
ENV_OPTIONS = [(action, None, 1) for action in env.actions] + [
(option_policy, termination_clause, max_steps)]
```

This list extends the actions within the environment to include all 'option_policy's required. Here, the 'option_policy' can be one that is loaded from a stored model or other class that contains a function to return an action item, given the environment and other required contextual information. Examples of the components used in this architecture can be seen in the Python notebook 'addition_of_hierarchical_concepts.ipynb' in the GitHub repository4. Another required component of Options, is that of a termination condition. We used environmental conditions, or a maximum number of steps, to specify these for our sub-policies. Examples of 'MessageTerminationEvent' and 'InventoryTerminationEvent' can be found in the sample code. These classes contain a 'check_complete' function which check for a specific message or item in the agent's inventory, respectively.

To implement Options and 'wrapped' primitive actions, the 'env.step()' function that is used in the ActorCritic training algorithm was replaced with a function that takes in an option number instead of an action number. The implementation of this can be seen in the 'actor_critic_v3_hierarchical_options.py' file, with the 'complete_option' function. The processing of this goes as follows:

- The option_policy, termination_clause and max_steps are extracted from the input Option selection.
- If the termination clause is defined as 'None', it is assumed to be a primitive action and the 'env.step()' function is used as normal.
- If it is an Option, the sub-policy is used to select primitive actions iteratively until the termination_clause or maximum number of steps is reached.
- A total reward and number of steps for the Option is returned along with the next state and relative contextual information.

2.4 Design Decisions

Small environments were used to train the agent to acquire certain skills that would be useful in the full QuestHard environment and to develop any further techniques that could be useful in solving the quest.

We began with a sample environment, with the goal of having the agent eat an apple. Following this, a Lava Crossing environment was used as this is one of the two first challenges for the agent in QuestHard.

2.4.1 Lava Crossing Skills

A core component of the QuestHard environment is the requirement of the agent to cross a field of lava. To do this, he must use either a levitation potion, ring or a frost horn to freeze the lava. A custom environment was built, based off the "MiniHack-LavaCross-Levitate-Potion-Pickup-Restricted-v0" environment, that contains a small room with a potion on the left half, a strip of lava through the middle and the destination staircase on the right half. An example of this can be seen in figure 3. The size of the environment was created to reduce the minimize the possible steps of the agent, due to the hardware limitations, yet still provide enough variation by having the potion and agent be placed at random locations for each iteration.

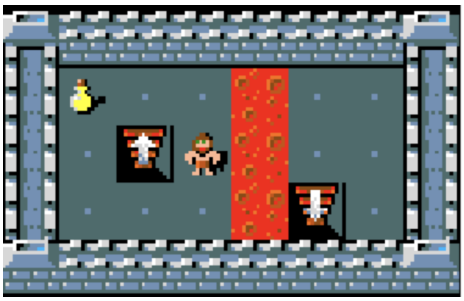

Figure 3: Training Environment for Options

This sample environment was broken into three main sub-tasks:

1. Locating (and picking up) the potion
2. Drinking the potion
3. Crossing the lava to the destination

Through this sample environment, the benefit of using Options could be assessed. We began by training the agent to pick up the potion, using the A2C algorithm discussed previously. This would be the first Option and sub-policy. Ideally, a second Option would be trained that would include the agent learning to drink the potion. This requires the agent to select the QUAFF action followed by a confirmation key which depends on the item's location in his inventory. Due to the time and computational restrictions, this policy was hard-coded by creating an Option containing the actions:

- QUAFF
- Check inventory for the "potion" and return the action corresponding to the inventory key

The third section, crossing the lava to the destination was left to be trained in the full Policy.

2.4.2 Reward Shaping

Reward shaping is an important aspect in achieving successful training of the agent and in speeding up the learning process. A careful balance between mitigating the difficulty of sparse rewards versus adding too much bias was maintained, however.

Adjustments to the MiniHack Reward Manager
There were several issues found with the built-in MiniHack reward functions that made our training objectives difficult to achieve. For example, the 'add_location_event()' function attempts to check whether the agent is standing on top of a specified object. The function checks for visibility of the object in the environment. This means that if an object does not exist in the environment or is not yet visible, it would be considered as achieved. To improve upon this, we created an adapted version of MiniHack's reward functions. A summary of these can be found in the Github repository, under 'reward_manager_improvements.ipynb' 4. Here, a set of the neighbouring cells for the previous time step is saved so, when the event check is made, it ensures that the relevant object was visible in the surrounding cells before the agent is assumed to be standing on top of it.

Addition of New Reward Events
Three new reward events were created. Each of these will be outlined, below, and can be found in the 'custom_reward_function.py' and 'reward_manager_improvements.ipynb'(summary of changes) files in the GitHub repository.

- **InventoryEvent:** This event was created to determine whether an item is found in the agent's inventory. It is based off the default Event types and uses the 'inv_strs' observation to assess the current inventory. This was particularly useful for improving the reward shaping when training the agent to pickup an object or ingest a consumable.

- **RelativeCoordEvent:** This event uses the agent's coordinates, stored in the 'blstats' observation key, and provides a reward if he has moved in a direction relative to the furthest he's been in that direction thus far. This was created to encourage the agent to move rightwards when exploring the maze and reaching the goal in the QuestHard (and most learning environments).

- **NeighbourEvent:** This event uses the MiniHack base function 'env.get_neighbor_descriptions()' to get the details of the squares surrounding the agent and output whether a specified object is within these squares. This was developed with the maze in mind, as it would be beneficial to produce a reward when the agent reaches the door.

2.4.3 Reward Shaping in the Lava Environment

While training the agent to collect and consume a levitation potion and cross the lava, rewards were given for:

- Finding the potion (auto-pickup activated) - Inventory Event
- Consuming the potion - Message Event
- Reaching the downwards staircase - Location Event

2.4.4 Reward Shaping in QuestHard

To encourage exploration in the QuestHard environment, we used a reward event that calculates the number of black squares visible and compares this to those from the frame before. If the number has been reduced, a reward is received. This encourages the agent to move along the map to uncover more squares. Another technique employed, was to use the 'RelativeCoordEvent' and 'Neighbour' event to encourage the agent to move rightwards and receive a reward when reaching the door.

2.5 Hyper-parameters

The following hyper-parameter were used when implementing our actor-critic algorithm to train the sub-tasks:

max-episode-length: Used to restrict the maximum number of steps that the agent can take in the environment before the episode is terminated. This was adjusted depending the size and complexity of the environment that was being run. We adjusted this value from 100 on the really simple environments, up till 1000 on the more complex ones.

num-episodes: Used to control the number of episodes run during the training process. We usually found 1000 to be sufficient, but some environments where train on up to 6000 episodes. We were also often limited due to time and hardware constraints.

learning-rate: Learning rate used for the optimizer. We decided to select a value of 0.01 as this seemed to work the best for the ranges of values that we found were typical used on actor-critic algorithms.

gamma: The discount factor used when calculating the discounted returns of an episode. A value of 0.99 was used as we wanted to also consider longer-term rewards.

2.6 Training Results

The results from training the various sub-tasks are provided below.

2.6.1 Eat Apple

The reward shaping used in our sample 'Eat Apple' environment are outlined in the table below.

Reward Manager		
Action	**Reward**	**Terminal**
Pick-up apple	2	No
Eat apple	3	No
Confirm Eat	5	Yes

We found the network did learn to eat an apple, however it did not always perform the 3 actions (pick-up, eat, and confirm message) sequentially. For the majority of the time the agent would pick-up the apple and then only eat and confirm after a couple of steps.

2.6.2 Lava Crossing Skills

Training the agent to complete the full lava crossing environment (without Options) did not have much success. With improved compute performance, this may have been achievable without any input. Our custom training technique of using learned skills as Options proved to be far faster to train and the agent was able to reach the goal.

Figure 4 shows a plot of the agent's episodes while training it to locate the potion.

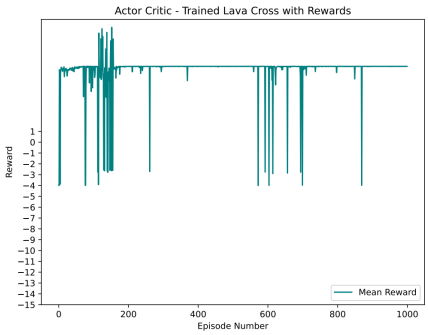

Figure 4: Trained Agent to Locate a Potion

Figure 5 shows a plot of the agent learning to cross the full lava field.

Large variability is visible but both plots roughly converge to an optimal return.

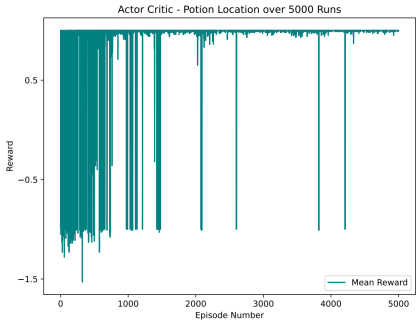

Figure 5: Trained Agent to Cross the Lava Field

2.6.3 Explore Maze

Due to our computation limitations, we were unable to perform the large number of episodes that would be required to achieve substantial learning for the agent to escape the maze at the start of the environment. Several observations were made, however, which included the effects of varying the maximum number of steps per episode. When increasing this parameter, the agent began maximizing the returns much faster. Unfortunately, this made training incredible slow and thus no full plots could be produced for this training session.

The agent did however, manage to make some progress through the maze with our training-limited policy. An example of this can be seen in the video in the GitHub repository.

2.7 Discussion of Results and Future Work

There are many ways in which our implementation of this algorithm and architecture could be improved upon. Those requiring large computation aside, several are discussed below.

- Extend the built in function 'env.get_neighbor_descriptions()' so that the agent has a bigger FOV of what is around it. This will help improve the agents ability to locate and navigate to objects.
- Add in other contextual information in the observations. The network currently only uses messages and glyphs as the main input source. Glyphs however, are unique identifiers for each item in the game. So, for example, potions each have a different identifier. Adding another source, such as the colours or chars, would assist in providing information that would link these object classes together.
- Extend the 'Reward Manager' functions to implement the sequential rewards. The existing function did not seem to work as intended. This could

be potentially be useful in training the agent to better perform when executing consecutive action (such as using items from inventory).

Further observations over training include the points:

- Policy initialization: We found that pre-training on smaller environments, help improved the speed of training on bigger/more complex environments. For example, the lava crossing policy was learned on the small environment shown in figure 3 yet applied to the larger "MiniHack-LavaCross-Levitate-Potion-Pickup" environment when recording the video.
- High variance: The network managed to learn all the sub-tasks, as the rewards received tend to converge to the maximum rewards available for their respective environment. However, as can be seen in the plots, it does this with a high degree of variance.

While training the agent in the lava environment, the effects of reward shaping were acknowledged. It was important not to add too much human bias and an interesting observation was made. The initial thought was to penalize the agent for entering the lava. It seems logical that this penalty should be made for him dying and terminating the episode. However, this meant he would later, after consuming the potion, be extremely hesitant to cross the lava. If there was no penalty, or a penalty too small however, we found that the policy tended to converge on him running straight into it, as seen in figure 6 in the Appendix. What ended up being most efficient, was not penalizing entering the lava but creating strong rewards for finding and consuming the potion. This meant that he would not be averse to crossing the lava but would always want to optimize the rewards by first consuming the necessary potion. The video 'lava_cross_complete' shows the effectiveness of this strategy, using the Option policy as described earlier.

Ideally, the the network architecture could be constructed to learn the skills on its own. However, as noted, the computational requirements for this would be very large and thus this formed a good demonstration of the effectiveness of incorporating skills into the learning process.

2.8 Best Runs Of Agent/Video Link

Please find the videos for the various agent episodes under the 'videos' directory in our GitHub4 repository.

3 Conclusion

Our implementation of a DQN algorithm managed to perform well in certain environments. However, we found our method of using Actor-Critic with Options Framework to be the better algorithm as it performed better overall, both within the Quest-Hard environment as well as learning a policy on the respective

sub-tasks. The policies trained using the Actor-Critic on the sub-task all converged to the optimal rewards, showing that they successfully learned to do the required task, but they did seem to have very high variance in their results. We found that not having access to GPUs in order to train our data to be a major hurdle. Overall we are satisfied with our results using the options framework as it was a complex implementation and worked very well on the test environments. Due to the time and computational requirements of training the agent to complete the maze, we were not able to effectively test the implementation in the full Quest-Hard environment, however.

4 Link To Github

GitHub Repository

5 Appendix

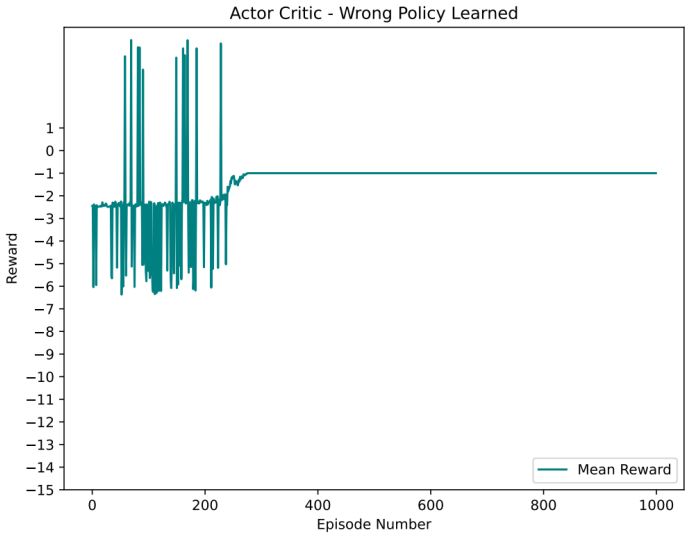

Figure 6: Example of an A2C Policy Converging Undesirably.

References

[Bacon *et al.* 2016] Pierre-Luc Bacon, Jean Harb, and Doina Precup. *The option-critic architecture*, Dec 2016.

[Brockman *et al.* 2016] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*, 2016.

[Budler 2021] Brenton Budler. *deep-rl-minihack-the-planet*. `https://github.com/BrentonBudler/deep-rl-minihack-the-planet`, 2021.

[Espeholt *et al.* 2018] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, and et al. *Impala: Scalable distributed deep-RL with importance weighted actor-learner architectures*, Jun 2018.

[Grillitsch 2023] Carina Grillitsch. Performance of generative adversarial imitation learning in various minihack environments. *Scientific Computing Conference 2023*, 2023.

[Küttler *et al.* 2019] Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. TorchBeast: A PyTorch Platform for Distributed RL. *arXiv preprint arXiv:1910.03552*, 2019.

[Küttler *et al.* 2020] Heinrich Küttler, Nantas Nardelli, Alexander Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The nethack learning environment. *Advances in Neural Information Processing Systems*, 33:7671–7684, 2020.

[Matthews *et al.* 2023] Michael Matthews, Mikayel Samvelyan, Jack Parker-Holder, Edward Grefenstette, and Tim Rocktäschel. *SkillHack: A benchmark for skill transfer in open-ended...*, May 2023.

[Mnih *et al.* 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, and et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[RAIL-Lab 2018] RAIL-Lab. *dqn*. `https://github.com/raillab/dqn`, 2018.

[Samvelyan *et al.* 2021] Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Kuttler, Edward Grefenstette, and Tim Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.