Minihack Project Report

Thabo Rachidi (1632496) Bongumusa Mavuso (1682836)

November 10, 2021

1 Introduction

In this project, we have created two agents using two types of reinforcement learning algorithms to navigate the Quest-Hard task from the NetHack[1] game using the MiniHack[2] environment.

The Quest-Hard task has various sub-asks which the agent must perform in order to complete the game. The first sub-task is for the agent to cross the river of lava, using any object it can find such as something that it can use to freeze the lava or levitate over the lava. The second task, which is reachable after it successfully finishes the first task, is for the agent to find a key and use it to open a hidden chest which will assist it to locate the WoD. The WoD is required to will kill the powerful monster who is the last agent's task before reaching the goal. The Quest-Hard is not trivial and requires the agent to navigate complicated mazes, interact with the environment and make decisions.

We have chosen to make the two reinforcement learning agents for completing the task using Deep-Q Network(DQN) and REINFORCE respectively.

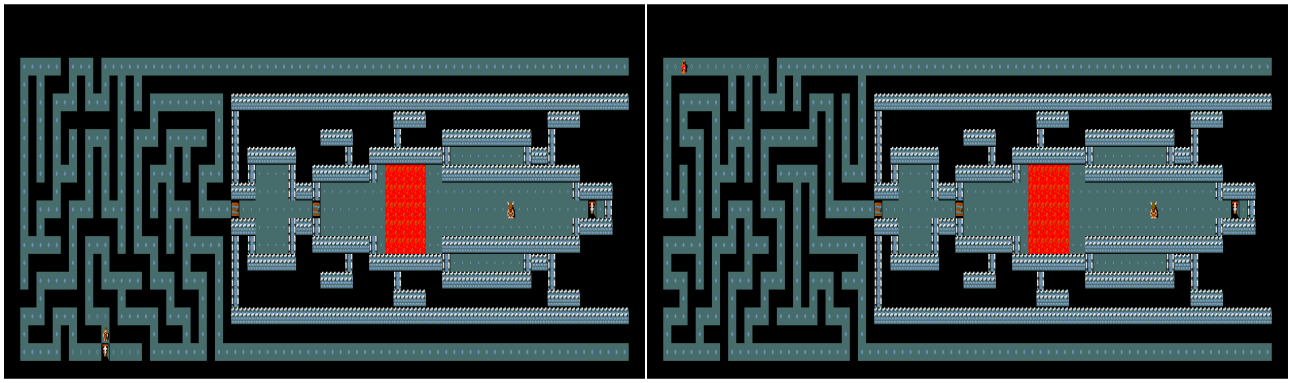

Figure 1: Examples of Quest-Hard task [2]

2 Algorithms

2.1 Deep-Q Network(DQN)

2.1.1 Overview

The Deep-Q Network(DQN) is a value function based method used train an agent by learning policies using only pixel data. This algorithm combines Q-Learning with deep neural networks[3].

The agent only is given a partial view of the task as it will see the current state of the game. The agent learns strategies using actions and observations $s_t = x_1, a_1, x_2, \ldots, a_{t-1}, x_t$ as inputs to the algorithm.

The agent is focused on choosing actions that maximise future rewards. Thus the aim of DQN algorithm is to maximise the discounted cumulative reward(return)

$$R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t, \quad 0 < \gamma \leq 1 \tag{1}$$

The constraint on γ(discount) ensures that the sum converges. This ensures that the rewards from the far future that the agent is uncertain about are less important than the rewards in the near future that the agent is somewhat certain about.

The optimal action-value function $Q^*(s, a)$ is the maximum expected return achievable by following any policy, after seeing some state s and taking some action a

$$Q^*(s, a) = \max_{\pi} \mathbb{E}[R_t | s_t = s, a_t = a, \pi] \tag{2}$$

where π is policy that maps from states to actions.

Since this is partially a reinforcement learning task we still need to estimate an action-value function using the Bellman equation as an iterative update.

$$Q_{i+1}(s, a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q_i(s', a')[s, a]] \tag{3}$$

By using value iteration algorithms this will converge to the optimal action-value function Q^*. It is usually more common to estimate the action-value function by using a function approximator.

A nonlinear approximator such as a neural network approximator with weights θ as a Q-network. We can then train the Q-network by adjusting the parameters at each iteration to reduce the mean-squared error in the Bellman equation. The optimal target values are substituted with approximate target values using parameters from some previous iteration. Thus we have the following sequence of loss functions that change at each iteration

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}[(y - Q(s, a; \theta_i))^2] + \mathbb{E}_{s,a,r}[\mathbb{V}_s[y]] \tag{4}$$

When optimising we keep the parameters stationary from the previous iteration when optimising the ith loss function. This results in a sequence of well defined optimisation problems.

The variance of the targets does not depend on the parameters that we are currently optimising and are therefore ignored. We can get the gradient by differentiating the loss function with respect to the weights

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}[(y - Q(s,a;\theta_i))^2] + \mathbb{E}_{s,a,r}[\mathbb{V}_s[y]] \tag{5}$$

We can optimise the loss function using stochastic gradient descent.

Note: This algorithm is model-free and off-policy. It should also be noted that the behaviour distribution is often selected by an ϵ-greedy policy.

2.1.2 Data Pre-processing

We used the pixel representation of the environment to train the agent. We decided to not resize the input to 84 by 84 as it was done in the paper *Human-level control through deep reinforcement learning* by Mnih, *et al* from DeepMind.

The environment observation space is given as a dictionary, with pixel and other formats used for state representation. Since we are interested in the pixel form, we make use of gym *PixelObservationWrapper* to obtain pixel observation space and discards other representations.

2.1.3 Training process

At the beginning of the training, the algorithm initializes the replay memory D to capacity N, the action value function Q with random weights θ and the target action value function $\hat{\theta}$. For each episode, the agent first selects and executes actions according to an ϵ-greedy policy and observes the reward r_t and image x_{t+1} and store this transition $S_{t+1} = s_t, a_t, x_{t+1}$ into the experience relay memory. Experience relay memory is used to store the agent's transitions at each time step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset $D_t = \{e_1, ..., e_t\}$ pooled over many already learned episodes. The Q-learning updates are updated during the inner loop through mini batch updates to samples of experience. The batches in the memory are randomized in order to decorrelate them by averaging the behaviour distribution over many previous states. This is to deal with the instability found in the neural network nonlinear approximator of a Q-function and increase learning efficiency. The algorithm then performs gradient descent on the random batch with respect to the parameters θ. For this task, we used Adam optimizer[3].

2.1.4 Q-Network Architecture

The Q-Network architecture is a deep CNN that takes in raw input from four past states stacked together. The structure of the network layers is made up of four convolution layers with linear layers in between and two final linear layer. The first hiddden layer convolves 32 filters of 8×8 with stride 4 with the input image to which it then applies a rectified linear unit(relu). The second hidden layer

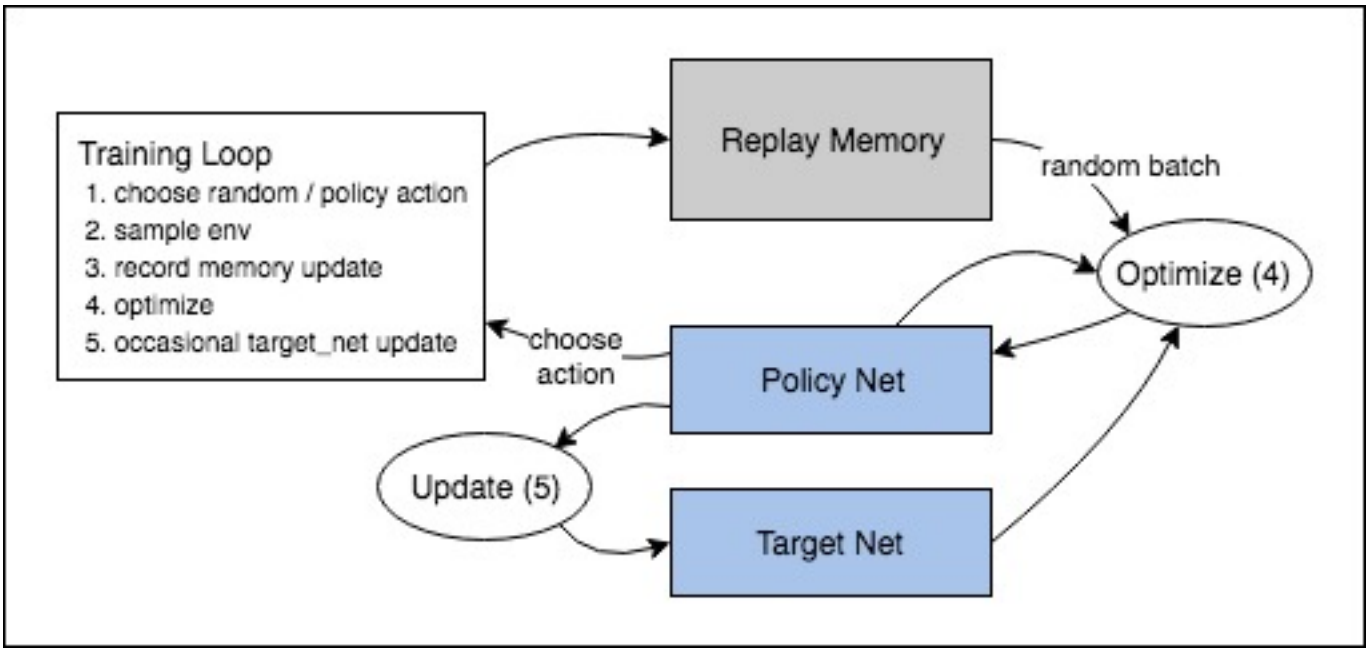

Figure 2: Training Loop

convolves 64 filters of 4×4 with stride 2 followed by a relu. The next layer convolves 64 filters of 3×3 with stride 1 followed by a relu. Then the final hidden layer which is a fully connected linear layer with 512 relu units. Then the output layer is also a fully connected linear layer that produces a single output from the action space of Quest game which is 58.

The summary of this network is provided below, as implemented in pytorch:

- (conv1): Conv2d(4, 256, kernel size=(8, 8), stride=(8, 8))
- (conv2): Conv2d(256, 64, kernel size=(4, 4), stride=(4, 4))
- (conv3): Conv2d(64, 32, kernel size=(2, 2), stride=(2, 2))
- (conv4)): Conv2d(32, 64, kernel size = 1, stride = 1)
- (linear1): Linear(in features=6080, out features=512)
- (linear2): Linear(in features=512, out features=58)

2.1.5 Hyper-parameters

- Learning Rate: $\alpha = 0.0001$ The learning rate used in the Adam Optimizer.
- Discount: $\gamma = 0.99$ Ensures that the algorithm converges
- Number of steps: 1×10^6 If the algorithm does not automatically converge, it will be run over this number of steps

- Replay-buffer-size: 1×10^3 The amount of memory to be used for relay memory.
- Batch-size: 256 Determines the number of samples used per training.
- Number of steps before learning starts: 10000
- Learning-Frequency: 5 This is the number of iterations between every target network update.
- target-update-frequency: 1000 This is the number of iterations between every target network update)
- eps-start: 1.0, Starting ε-greedy start threshold before decaying.
- eps-end: 0.01, ϵ-greedy end threshold or lower bound.

2.1.6 DQN Results

After training our DQN agent we managed to create an agent using the parameters stated above in section 2.1.5.

2.1.7 Motivation

The DQN makes use of memory buffer, to randomly store batches to remove correlation, the DQN agent will be able to generalize well in the Quest environment.

2.2 REINFORCE

2.2.1 Overview

REINFORCE(Monte-Carlo policy gradient)[4] is a type of on-policy gradient learning algorithm. It follows some policy π_θ to gain some experience function. Unlike the DQN, REINFORCE does not reuse old experiences as it discards them once it has learned new policy.

Policy gradient methods seek to maximise the expected return of a trajectory. These are represented by the objective function:

$$J(\theta) = E_{\tau\ \theta\pi}\left[R(\tau)\right] \tag{6}$$

The gradient of the objective function ∇J maximizes the objective function:

$$\begin{aligned}\nabla J &= E_{\tau\ \theta\pi}[\nabla_\theta log\pi_\theta(a|s)Q_{\pi\theta}(s,a)] \\ &= E_{\tau\ \theta\pi}[\nabla_\theta\pi_\theta(a_t|s_t)G_t],\end{aligned}$$

where G_t is the reward computed from a complete trajectory. This can only be done after some $T-t$ time steps, where T is the number of trajectories. The parameter Zθ update is done using gradient ascent, so that they will move to the same direction as the gradient[5].

2.2.2 Data Pre-processing

We used the cropped pixel representation of the environment to train the agent. The cropped pixel observations are RGB images of size $144 \times 144 \times 3$. To make things easier for the Neural Network we used a environment wrapper that reshapes the images to have dimension $3 \times 144 \times 144$.

2.2.3 Training Process

The REINFORCE algorithm training process is pretty straightforward. It begins by creating an episode $\{S_0, A_0, R_1, S_1, A_1, \dots\}$ by sampling the policy. We than save the episodes in a trajectory(τ). We then begin the training by first computing the estimated returns G_t, for every step of the episode. This is when we subtract a baseline from the return, so that we can reduce the variance of gradient estimation. We can then update the policy parameters by computing the policy loss. We use back-propagation with gradient ascent to calculate the policy loss.We can now reset the trajectory and start all over again. We also used the Adam optimizer for this algorithm.

2.2.4 Policy Architecture

The policy architecture is a Convolutional Neural Network (CNN) that takes in the RGB pixel observations one at a time. The network is made up of 3 convolutional layers, a max pooling layer and two final linear layers. Then finally we apply a softmax to obtain the probabilities.

The summary of the network is provided below, as implemented in PyTorch:

- (conv1): Conv2d(3, 32, kernel size=(8, 8), stride=4)
- (conv2): Conv2d(32, 64, kernel size=(5, 5), stride=2)
- (conv3): Conv2d(64, 64, kernel size=(3, 3), stride=1)
- (pool): MaxPool2d(2,2)
- (linear1): Linear(in-features=3136, out-features=128)
- (linear2): Linear(in-features=128, out-features=78)

2.2.5 Hyper-parameters

- Learning Rate: $\alpha = 0.001$ The learning rate used in the Adam Optimizer.
- Discount: $\gamma = 0.99$ Ensures that the algorithm converges
- Number of episodes: 5×10^4 If the algorithm does not automatically converge, it will be run over this number of steps
- Number of steps: 1×10^5 The max number of steps to take during a single episode.

2.2.6 REINFORCE Results

After training our REINFORCE agent we managed to create an agent using the parameters stated above in section 2.2.5.

2.2.7 Motivation

We used the REINFORCE algorithm because of simplicity and because it does not need us to model the environment as modelling the Quest-Hard environment is non-trivial. Policy gradient methods in general are also known to naturally explore due to the stochastic policy representation.

References

[1] H. Küttler, N. Nardelli, A. H. Miller, R. Raileanu, M. Selvatici, E. Grefenstette, and T. Rocktäschel, "The nethack learning environment," 2020.

[2] M. Samvelyan, R. Kirk, V. Kurin, J. Parker-Holder, M. Jianga, E. Hambro, F. Petroni, H. Kuttlera, E. Grefenstette, and T. Rocktäschel, "Minihack the planet: A sandbox for open-ended reinforcement learning research," in *Thirty-fifth Conference on Neural In-formation Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[4] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229—256, 1992.

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