COMP6247 REPORT: MAZE SOLVING PROJECT

Reshma Abraham
Department of Electronics and Computer Science,
University of Southampton.
ra2n21@soton.ac.uk

ABSTRACT

This report documents the work undertaken to solve the maze problem to compute the minimum time path. It describes the reinforcement learning techniques adopted and the modifications made to derive a solution. Additionally, the report attempts to capture the thought process in developing this custom algorithm. The results and the algorithm's performance have also been discussed.

1 PROBLEM

We are given a 199 x 199 maze environment surrounded by walls. The agent's goal is to compute the optimal path from the top left corner (1,1) to the bottom right corner (199,199). Each location has a binary value associated with it, which could be a wall or an empty spot. This problem is made further challenging by adding dynamic firing at random locations. However, I have only considered a static maze and left the handling of active firing for future improvements of this project.

The maze environment is given by the `read_maze.py` file provided with the coursework instructions. The function, *get_local_maze_information(x, y)* in the file, is used to retrieve the local position information of the agent after every action, and it is the only interface to retrieve any information regarding the maze.

I propose a dueling deep Q-network with a prioritised replay buffer to solve the optimal path maze problem. The components of the proposed algorithm are described in detail in section 2. The experimental values and results have been described in section 3. I believe the algorithm presented in this report can be extended easily for dynamic mazes with minor tweaks in the reward function and a few additional convolutional layers to the feature extraction process. This has been discussed in detail in section 5.

2 ALGORITHM

The traditional Q-learning algorithm [Watkins & Dayan (1992)] has shown remarkable results in maze solving problems [Osmanković & Konjicija (2011)]. Q-learning involves repeatedly visiting states and updating the expected cumulative reward for every possible action in each state to determine the best action for a state. The Q-value denotes the expected rewards for an action taken in a given state. Q-learning attempts to maximise the expected reward value of all the successive steps. However, they suffer from slow convergence and are inefficient in solving complex problems in large state-action spaces due to the large sparsity of Q-tables. The approach I've implemented uses dueling Deep Q-network with Prioritised Replay buffer. The dueling DQN is also coupled with ϵ-Greedy Decline Method. The approach draws inspiration from IPD3QN algorithm proposed in [Zhu et al. (2021).

Deep Q-networks allow extracting high level features from the sensed environment data. They offer a non-linear approximation of the value function. But often it is not necessary to estimate the value of every action in a state. Therefore in my solution, I have used dueling architecture[Wang et al. (2015)] to separately compute the state value and state action, thereby improving the accuracy of Q-value estimation. There are two streams of data in the dueling network, one corresponding to the state value $V(s)$, and the other to the action advantage $A(s, a)$ which is centralised to ensure optimisation stability. The output of the dueling DQN is given by:

$$Q(s, a) = V(s) + (A(s, a) - avgA(s, a')) \tag{1}$$

where a' denotes all possible actions.

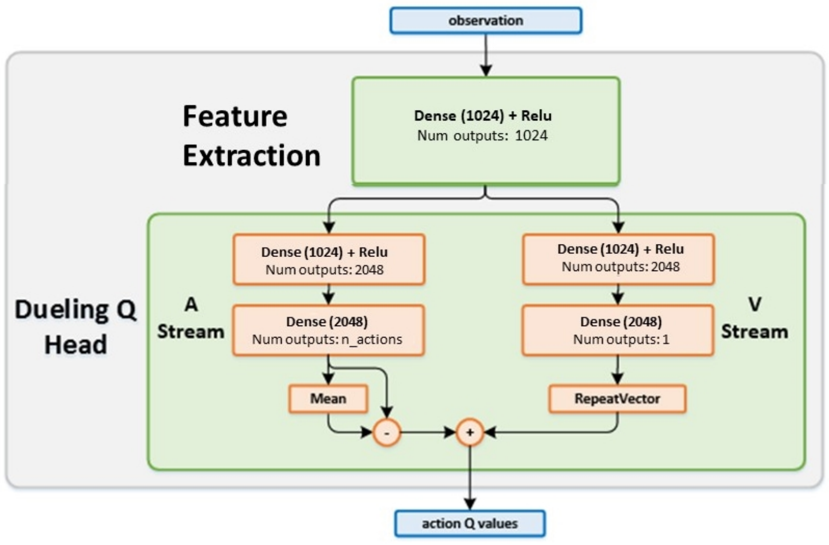

Figure 1: Proposed dueling DQN architecture

Deep Q-Networks [Roderick et al. (2017) and its variants utilise experience replay buffers to randomly sample prior actions instead of the most recent one. The key reason of doing this is to break the correlation in the observation sequences. But this does not take into account the importance of the experience samples and assigns equal probability to all samples. As a result, during uniform random sampling experiences of higher importance maybe missed out preventing the model from reaching its full efficiency. Therefore, I used prioritised experience replays [Schaul et al. (2015)], where weights are assigned to each experience. Every experience is associated with priority, probability and weight. The priority is updated according to the temporal-difference (TD) error obtained after the forward pass of the neural network given as:

$$TD_{error} = (Q_{pred} - Q_{target})^2 * w_i \tag{2}$$

To compute the TD-error and update the the network for successive steps, I have implemented a target network. The target Q-value is given as follows:

$$Q_{target} = R + \gamma . \max_{a'} Q(s', a') \tag{3}$$

The authors of prioritised experience replay used two hyper-parameters α and β to control the priority. The equations for sampling probability is as shown:

$$P(i) = \frac{p_i^\alpha}{\Sigma_k p_k^\alpha} \tag{4}$$

where $p_i > 0$ is the priority of transition i. In the original paper, $\alpha \in [0, 1]$ controls the extent of priority use. However, I have used $\alpha = 0.6$. It is to be noted that prioritised replay introduces bias due to the lack of diversity of experience and is prone to overfitting. To correct this bias, importance-sampling (IS) weights were introduced given by:

$$w_i = \left(\frac{1}{N} . \frac{1}{P(i)} \right)^\beta \tag{5}$$

where β denotes the amount of correction which is incremented in every episode during training.

To strike a balance in exploitation of prior knowledge and exploration of new options, the Dueling DQN was coupled with a ϵ-greedy decent method. In the training phase, the ϵ was decremented after every episode by a factor of 0.005 upto a min of 0.0003. This allowed the agent to explore the state-action space during the initial episodes and prevent it from getting stuck in a location for too long. The algorithm is described below :

Algorithm 1 Epsilon Greedy dueling Deep Q-network with Prioritised Experience replay

```
Input: $\beta$, $\gamma$, $\epsilon$, $\alpha$, no. of. episodes N, replay-memory M, batch-size B.
Initialise: Q-learning parameters, done = False, Observation map
for t = 1 to N:
 1: while not done: do
 2:     Observe $S_t$ and choose $A_t \sim \pi_{(S_t)}$ with $\epsilon-$ greedy.
 3:     Store transition $S_t, A_t, R_t, S_{t+1}, done$ in M with max priority $p_t = \max_{i<t} p_i$
 4:     Update state $S_t = S_{t+1}$
        If M mod B == 0
 5:     Compute $Q(s,a) = V(s) + (A(s,a) - avgA(s,a'))$
 6:     for j=1 to B do
 7:         Sample transition i ~ P(i) = $\frac{p_i^\alpha}{\Sigma_k p_k^\alpha}$
 8:         Compute importance-sampling weight $w_j = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$
 9:         Calculate the current target $Q_{target} = R + \gamma . \max_{a'} Q(s', a')$
10:         Use the mean square error loss function to update the Q-Network parameters.
11:         Recalculate TD-error, $\delta_j = Q_{target} Q(s_{t-1}, a_{t-1})$
12:         Update transition priority $p_j = |\delta_j|$
13:     end for
14: end while
```

3 Experiment Design

The experiment was carried out with PyTorch 1.8 and Python 3.7 under windows 11 OS. Occasionally, ECS GPU Compute Service was also used for training. The maze model provided with the coursework was visualised without the fire using Pygame and a snapshop of evaluation on (37x37) maze is shown in figure 2. It is to be noted that none of the visualised information was fetched by the agent during training and evaluation. The agent derived information only from *get_local_maze_information(x, y)*. The environment allows four actions defined as

$$A = [\textbf{'stay}, \textbf{'up'}, \textbf{'left'}, \textbf{'right'}, \textbf{'down'}] \tag{6}$$

The reward function designed for the problem is as follows:

$$R = \begin{cases} \textbf{'forward'} : +1, \\ \textbf{'away'} : +.8, \\ \textbf{'visited'} : -.05, \\ \textbf{'wall'} : -.1, \\ \textbf{'stay'} : -0.01 \end{cases} \tag{7}$$

'forward' provides the agent the incentive to move towards the goal and away from the origin to make forward progress. The attempt is to prevent the agent from staying at a location for too long. **'away'** denotes the case where the agent may have to retrace its steps due to walls, and move away from the goal. In this case, the reward is also weighted by

$$\left[\frac{No\_of\_steps\_retraced}{10}\right] \tag{8}$$

The case **visited** decrements the score if the agent had previously visited the location in that episode. This encourages the agent to explore alternative actions. **stay** penalises the agent if it stays at a location for no reason. In a static maze, the agent has no reason to stay and this action is irrelevant. However, this has been included in the solution to easily extend it for dynamic scenarios. Finally, the agents receives negative rewards when it arrives at a location with a wall. These reward values and associated logic has been arrived at through iterative trial and testing in the environment.

For feature extraction, I have implemented a sequential layer containing a dense layer of 1024 nodes and RELU (Rectified Linear Unit) activation layer. This is followed by the dueling head with the value and action stream. The dueling head structure implemented is described by figure 1.

I conducted two types of experiments to evaluate the performance of the solution and the associated agent. To decide the hyperparameters and determine the reward functions, I conducted the first set of experiments on a 5x5 and 7x7 maze whose goal state were (5,5) and (6,3) respectively. The training was conducted for 500 episodes and the hyperparameter setting are shown in table 2. ReLU is used as the activation function and ADAM is selected for gradient descent optimisation. Mean Squared Error (squared L2 norm) is used as the loss function for optimisation.

Hyperparameter	Value	Remarks
Episodes	500	For (5X5) and (7X7)
	10,000	For (199x199)
Learning rate	0.001	-
Epsilon	0.8	Decremented by 0.05
	0	For testing.
Min. Epsilon	0.0003	
Beta	0.4	Incremented by 0.01
Gamma	0.999	-
Batch size	64	-
Memory	100000	-
Optimiser	Adam	-
Loss	MSEloss	-

Table 1: Hyperparameter Values

The second set of training and experiments were conducted for the 200x200 matrix with 199x199 as the goal state. The network was trained for 10000 episodes, but the learning did not converge. The trained model was then evaluated on a smaller maze of size (37x37) and the actual (200x200) maze.

The implementation code has been published in my Github repository [Code]. To run the program, simply run the *main.py* file after making necessary configuration changes such as setting the network mode (train/test), number of episodes, etc. The *requirements.txt* file containing necessary package requirements are made available in the repo.

4 Performance and Results

The algorithm was evaluated for (5x5), (7x7), (37x37) and (200x200) mazes. The results are shown in table 2. It was observed that in smaller mazes, the agent took the optimal path and reached the goal state in less than a minute. As the size of the maze increased, due to the hard bound on the number of episodes during training, learning did not converge. This was observed in the cases of 200x200 maze. The logs of the training and the associated models are made available in the code repository. The output file submitted corresponds to the 37x37 maze. The results demonstrate the effectiveness of the proposed solution in finding the optimal path in minimum time.

Maze Size	**Goal state**	**Episodes Trained**	**Convergence status**	**Total Steps taken**	**Length of optimal path**
(5x5)	(5x5)	**100**	**Learning Converged**	**9 steps**	**7 steps**
(7x7)	(6x3)	**200**	**Learning Converged**	**12 steps**	**12 steps**
(37x37)	(36x13)	**5000**	**Learning Converged**	**60k steps**	**314 steps**
(200x200)	(199x199)	**10000**	**Learning did not converge**	**100K+ steps**	**Could travel till around 500 steps**

Table 2: Results on mazes of different sizes

Figure 3, 4 and 5 shows the loss, length of path taken and average score obtained by the agent over 5000 episodes in a (200x200) maze. It can be seen that the loss during the first 3000 episodes is fairly low. This can be attributed to setting the value high at 0.8. As the value hits the minimum of 0.003, the learning slows and the agent was seen to have a higher tendency of remaining in a location. However, the epsilon-greedy decent method prevented the agent from being stuck in one place for too long in the early stages of training. Decreasing the epsilon value was observed to reduce the agents reliance on the ϵ-greedy algorithm and choose optimal paths without getting stuck during evaluation.

It was observed that terminating the episode when the agent crossed the threshold of max visits to a location or wall during training, reduced the overall wall and visit counts during evaluation.

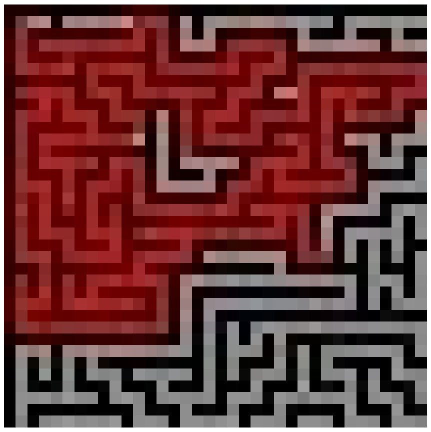

Figure 2: Snapshot of the trajectory of rat in 37x37 maze

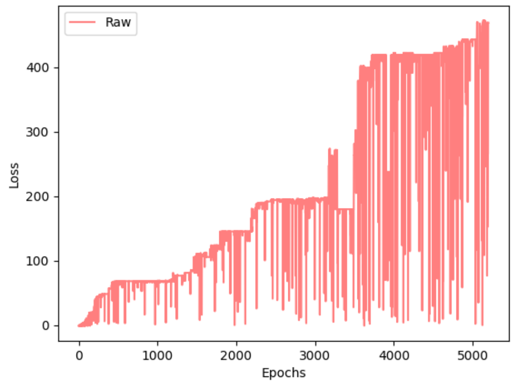

Figure 3: Loss Vs Episodes plot for training on 200x200 maze

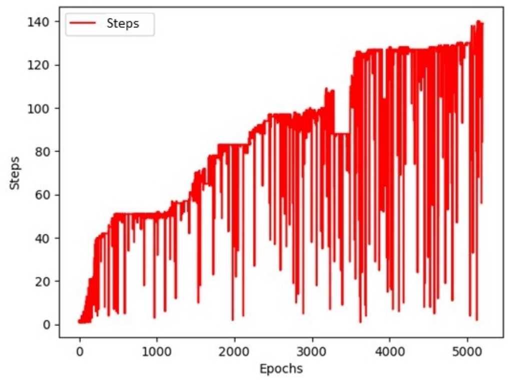

Figure 4: Steps Vs Episodes plot for training on 200x200 maze

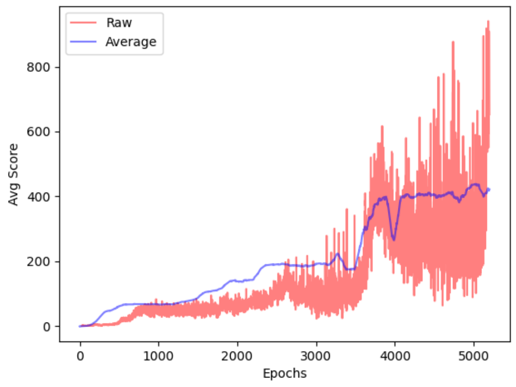

Figure 5: Average Score Vs Episodes plot for training on 200x200 maze

5 Future work and Improvements

As mentioned earlier the original problem statement included dynamic firing and was complex to solve given the duration of the coursework and the size of the maze. However, I have made consideration for active firing in my implementation so that the static maze problem can be easily extended for dynamic scenarios. The below paragraph summarises the improvements to be made to account active firing.

Since the fires are dynamic and location of firing changes after every time step, recurrent neural networks (CNN with LSTMs) have to be utilised in the feature extraction network. Additionally, the reward function has to be modified to decrement the score in case the agent lands on a fire. The number of training episodes will have to be increased to 100,000 for the maze of given size. Since the problem gets more complex, to reduce the sensitivity on outliers Huber's loss could be adopted. Mean is the estimator optimising L2 loss while median is the estimator optimising L1 loss. Huber's loss is a compromise and doesn't get easily influenced by outliers.

6 Conclusion

This report proposes an epsilon-greedy Dueling Deep Q-Network Based on Prioritised Experience Replay to compute the minimal time path for traversing a maze. The deep Q-Network selects and calculates the target Q value. The duelling network optimises the algorithm's stability by separating the state value and the state action. Prioritised experience replay is used to extract experience samples, increase the utilisation rate of important samples and accelerate the training speed of the neural network. Finally, the e-greedy descent method is used to find the optimal strategy and prevent the agent from getting stuck in locations, prompting it to explore. The experiments are first conducted on (5x5), (7x7), (37x 37) and (199x199) mazes to analyse the impact of algorithm hyperparameters on the model and set their values. Results demonstrate the effectiveness of the proposed algorithm. The loss and score were observed to saturate once the epsilon value reached a constant minimum. To improve the speed of traversal, learning has to run for longer epochs for the 200x200 maze.

References

D. Osmanković and S. Konjicija. Implementation of q — learning algorithm for solving maze problem. In *2011 Proceedings of the 34th International Convention MIPRO*, pp. 1619–1622, 2011.

Melrose Roderick, James MacGlashan, and Stefanie Tellex. Implementing the deep q-network. *arXiv preprint arXiv:1711.07478*, 2017.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015. URL `https://arxiv.org/abs/1511.06581`.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

Zhengwei Zhu, Can Hu, Chenyang Zhu, Yanping Zhu, and Yu Sheng. An improved dueling deep double-q network based on prioritized experience replay for path planning of unmanned surface vehicles. *Journal of Marine Science and Engineering*, 9(11):1267, 2021.