The A-maze-ing World of Q-learning.

Ashley Mason,
Frederick Miles,
City, University of London

April 11, 2018

Contents

1 Introduction

In this paper, we will demonstrate the agent model of artificial intelligence, showing how it may be applied with a simple Q-learning example. We will measure the performance of the model as we adjust the parameters and investigate how prior domain knowledge about the task may be used to improve model performance.

2 Defining the Q-Learning Model

2.1 Domain and Task

With careful calibration of the transition and reward functions, reinforcement learning can be a very effective way to solve Markov chain decision processes [1]. Moreover, Q-learning is a popular choice in reinforcement learning models as it is proven to eventually converge to an optimal solution.

In this demonstration of Q-learning, the domain we have chosen is a maze environment. A maze is a network of paths and walls designed as a puzzle through which one has to successfully navigate. This makes it a great candidate for demonstrating Q-learning, as a maze is a simple environment with few rules, which are easily programmable into our agent. In addition to successfully navigating the maze, we will require that our agent should do it in the fewest steps. This addition will equip our agent with the implicit human knowledge that a more direct solution with fewer steps is more desirable than one in which we take longer, repeat steps or travel in loops.

Throughout Section 2 of this paper, we will use a simple 6x6 maze to demonstrate how to set up the Q-learning model. In Section 3, we will graduate to a more complex 24x24 maze to carry out our investigations.

2.2 Defining a State Transition Function

In Q-learning, the state transition function describes how the agent gets to and from one state to the next.

Utilising a birds-eye view, with black and white pixels representing walls and positions in the maze respectively, we assign each position in the maze a node. We number these nodes starting with the top left (yellow), which will be the initial state for our agent, and ending with the bottom right (red), which will be the target state.

The actions available to our agent in a given position are moving to any adjacent positions or remaining in the current position. In our example, the available actions from position 0 are {0,1,5}, the available actions from position 1 are {0,1,2} and so on, Figure 1.

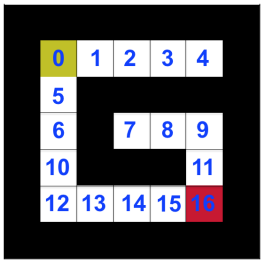

Figure 1: A representation of labelled nodes in the example maze.

In our maze environment, we define s_i to be the state in which the agent occupies the i-th position in the maze. In general, a state transition function maps a state-action pair to a resultant state; $f(s_{old}, a) = s_{new}$. In the context of our maze, each available action is equivalent to its resultant state so this can be simplified as follows.

$$f(s_i) = s_j \text{ where } s_j \in A_i \tag{1}$$

where the set of available actions, A_i, is defined as the set of states s_j such that i and j are adjacent (or equal).

2.3 Defining the Reward Function

In Q-learning, the reward function assigns a reward to each state-action pair and the agent's task is, typically, to maximise the total reward. The design of the reward function is the tool that allows the agent to find an optimal solution to a Markov chain decision process.

In other implementations of Q-learning on mazes [2], a common approach has been for transitioning to each state to carry zero reward except from the target state, which carries a positive reward. A problem with this approach is that if the maze has more than one solution path, the agent can converge to a sub-optimal solution; one with more steps. To account for this, we decided to penalise our agent for moving between non-target states with a reward of -1. We decided not to restrict our agent from continually moving, as we wanted it to have the same rules as a human, but penalised stationary actions with a reward of -5. Finally, reaching the target state would carry a reward of +100.

The reward function, r, which takes two adjacent (or equal) states[1] as inputs is defined as follows,

$$r(s_i, s_j) = \begin{cases} -1, & \text{if } s_i \neq s_j, s_j \text{ is not the target state} \\ -5 & \text{if } s_i = s_j, s_j \text{ is not the target state} \\ 100 & \text{if } s_j \text{ is the target state} \end{cases} \tag{2}$$

Note, r takes an ordered pair of states as an input. $r(s_i, s_j) = x$ should read as the reward for transitioning from the i-th state to the j-th state is equal to x.

[1] In this context, we say two states are adjacent if their corresponding positions in the maze are adjacent.

2.4 Choosing a Learning Policy

In Q-learning the learning policy, π, is the policy whereby the agent chooses an action to execute based on its current state in the environment. In other words, the learning policy says given a state, s_i, and set of possible actions, A_i, execute the action $a_j \in A_i$. The policy can be thought of as a function, $\pi(s_i) = a_j$.

Since we want to find the optimal solution, we choose a greedy policy which executes the action with the greatest Q-score. When there are multiple maximum Q-scores to choose from, the greedy policy selects one at random. Typically, the agent will have no prior knowledge of its environment and the Q-scores for each action are initiated as zero. This means that in the early stages, the agent chooses many random actions before learning which actions to take. It is therefore possible that a greedy policy will choose a series of actions leading to convergence on a sub-optimal solution.

To avoid this, we introduce an element of exploration in our agent. For some number $\epsilon \in [0, 1]$, our agent will select a random action with a probability of ϵ and select the action with the maximum Q-score with a probability of $1-\epsilon$. This is known as an epsilon-greedy policy with ϵ known as the exploration factor and a greater ϵ meaning a more exploratory agent. Our policy is formally defined as follows:

$$\pi(s_i) = \begin{cases} s_j, & \text{with probability } 1 - \epsilon \\ s_k, & \text{with probability } \quad \epsilon \end{cases} \tag{3}$$

$$\text{where } s_j, s_k \in A_i \text{ and } s_k = argmax_{s \in A_i}(Q(s_i, s)).$$

2.5 Graphical Representation

This problem can be represented graphically with nodes representing positions in the maze, and edges representing 1-step transitions between adjacent positions[2]. This has the advantage of allowing us to test whether our agent has reached an optimal solution by applying a shortest path algorithm for unweighted graphs, such as depth first search [3], and the two solutions can then be compared.

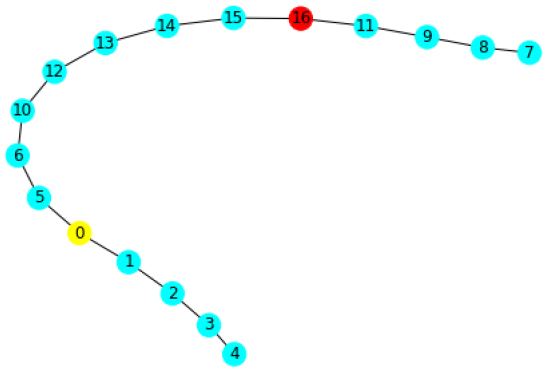

Figure 2: A graphical representation of a maze.

[2]An accurate graph of our maze would contain a loop at each node. We have omitted this to reduce visual clutter.

2.6 Setting the R-matrix

We store the rewards described in Section 2.3 in a square matrix with side length equal to the number of nodes in the maze. Equation (2) dictates the value of each entry in the R-matrix, $\mathbf{R}(i, j)$. The full R-matrix for our example is as follows.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	-5	-1	.	.	.	-1	.	.	.	.	.	.	.	.	.	.	.
1	-1	-5	-1	.	.	.	.	.	.	.	.	.	.	.	.	.	.
2	.	-1	-5	-1	.	.	.	.	.	.	.	.	.	.	.	.	.
3	.	.	-1	-5	-1	.	.	.	.	.	.	.	.	.	.	.	.
4	.	.	.	-1	-5	.	.	.	.	.	.	.	.	.	.	.	.
5	-1	.	.	.	.	-5	-1	.	.	.	.	.	.	.	.	.	.
6	.	.	.	.	.	-1	-5	.	.	.	-1	.	.	.	.	.	.
7	.	.	.	.	.	.	.	-5	-1	.	.	.	.	.	.	.	.
8	.	.	.	.	.	.	.	-1	-5	-1	.	.	.	.	.	.	.
9	.	.	.	.	.	.	.	.	-1	-5	.	-1	.	.	.	.	.
10	.	.	.	.	.	.	-1	.	.	.	-5	.	-1	.	.	.	.
11	.	.	.	.	.	.	.	.	.	-1	.	-5	.	.	.	.	100
12	.	.	.	.	.	.	.	.	.	.	-1	.	-5	-1	.	.	.
13	.	.	.	.	.	.	.	.	.	.	.	.	-1	-5	-1	.	.
14	.	.	.	.	.	.	.	.	.	.	.	.	.	-1	-5	-1	.
15	.	.	.	.	.	.	.	.	.	.	.	.	.	.	-1	-5	100
16	.	.	.	.	.	.	.	.	.	.	.	100	.	.	.	100	100

In the R-matrix $\mathbf{R}(i, j)$ represents the reward for transitioning from the i-th state to the j-th state; in our case the i-th to the j-th position in the maze and blank entries represent that those nodes are not adjacent (or equal).[3]

2.7 Setting the Q-learning Parameters

The Q-learning parameters are formally defined below in (Greek-)alphabetical order.

The Learning Rate, $\alpha \in [0, 1]$: In Q-learning, the agent makes estimates about the value of each state: the Q-scores. As the agent explores, these Q-scores are updated as new information is gained. The learning rate, α, determines the extent to which the difference between old and new estimates is accounted for on each update. When $\alpha = 0$, no learning will take place. When $\alpha = 1$, the Q-matrix will be updated with the full value of the new reward and estimated return [4]. We shall be using $\alpha = 0.25$ as the default learning rate for our agent.

The Discount Factor, $\gamma \in [0, 1]$: Dictates the extent to which the agent should take into account future rewards. When $\gamma = 0$, the agent should only consider the immediate reward. When $\gamma = 1$, the agent should consider all rewards: future and immediate. We shall be using $\gamma = 0.5$ as the default discount factor for our agent.

[3] In our python implementation of this code we use -10 in replace of the blank cells and allow the agent to choose only from cells with values greater than -10.

The Exploration Factor, $\epsilon \in [0, 1]$: As introduced in Section 2.4, ϵ is a measure of how exploratory the agent is and refers to the probability with which the agent either randomly explores or follows the learning policy. We shall be using $\epsilon = 0.5$ as the default exploration factor for our agent.

The Decay Factor, $\lambda \in [0, 1]$: As the agent learns, we want it to become less exploratory and follow the learnt policy more closely. Therefore, we update ϵ after each learning episode, where $\epsilon_{new} = \epsilon_{old}(1 - \lambda)$. We shall be using $\lambda = 0.001$ as the default decay factor for our agent.

The Learning Policy, π : The policy which governs the way in which the agent learns. As introduced in Section 2.4, we shall be using an epsilon-greedy policy for our agent.

2.8 Setting the Q-matrix and Updating

In Q-learning, the Q-scores of state-action pairs are updated according to the estimated value of the resulting state. We use a square matrix with side length equal to the number of nodes in our maze to store these Q-scores.

In some cases, prior domain knowledge is accounted for by pre-populating the Q-matrix with non-zero scores. In our case, we want our agent to have no prior knowledge of the maze and thus initiate our Q-matrix as a zero matrix. The following equation shows the general case of how each Q-score is updated for a state-action pair.

$$Q_{new}(s, a) = Q_{old}(s, a) + \alpha[r(s, a) + \gamma.argmax_{a \in A}\{Q(s_{new}, a)\} - Q_{old}(s, a)]$$

In the context of our maze, the state-action pair, (s, a), can be replaced with (s_i, s_j) which denotes moving from the i-th to the j-th state. Therefore, the following equation describes how to update a general entry $\mathbf{Q}(i, j)$ in the Q-matrix. Here, the term '$argmax_{a \in A}\{Q(s_{new}, a)\}$' can be thought of as the maximum value in the j-th row.

$$Q_{new}(s_i, s_j) = Q_{old}(s_i, s_j) + \alpha[r(s_i, s_j) + \gamma.argmax_{s \in A_j}\{Q(s_j, s)\} - Q_{old}(s_i, s_j)] \tag{4}$$

To demonstrate the updating process, we continue with our simple example:
The agent starts at node 0.
The available actions at node 0 are $A_0 = \{0, 1, 5\}$**.**
The agent chooses action 1 based on the epsilon-greedy policy.
Values are then substituted into equation (4) **to update** $\mathbf{Q}(0, 1)$**:**

$$\begin{aligned} Q_{new}(0, 1) &= Q_{old}(0, 1) + \alpha[r(0, 1) + \gamma.argmax_{s \in A_1}\{Q(1, s)\} - Q_{old}(0, 1)] \\ &= 0 + 0.25[-1 + 0.5 * 0 - 0] \\ &= -0.25 \end{aligned}$$

The Q-matrix is then updated.

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ \dots \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & \dots \end{array} \\ \begin{bmatrix} 0 & -0.25 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix} \end{array}$$

The agent is now at node 1.
The available actions at node 1 are $A_1 = \{0, 1, 2\}$.
The agent chooses an action based on the epsilon-greedy policy, for example action 1 again.
Equation (4) is applied to update $\mathbf{Q}(1, 1)$.

$$\begin{aligned} Q_{new}(1,1) &= Q_{old}(1,1) + \alpha[r(1,1) + \gamma.argmax_{s \in A_1}\{Q(1,s)\} - Q_{old}(1,1)] \\ &= 0 + 0.25[-5 + 0.5 * 0 - 0] \\ &= -1.25 \end{aligned}$$

The Q-matrix is updated again.

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ \dots \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & \dots \end{array} \\ \begin{bmatrix} 0 & -0.25 & 0 & 0 & \dots \\ 0 & -1.25 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix} \end{array}$$

This process continues for each step taken until the target state is reached. When this occurs, the training episode has finished and the epsilon value is updated. The agent then resets to the starting point of the maze and begins the next training episode with the Q-matrix continuing from the last training episode. When all training episodes are complete, the final Q-matrix is used to test the agent's performance.

3 Utilising the Q-learning Model

The 24x24 maze we will be using for our investigations was generated using a maze generator function supplied by Github user fcogama [5]. The starting point for our agent will be node 0 at the top left (yellow) and the target is node 291 at the bottom right (red).

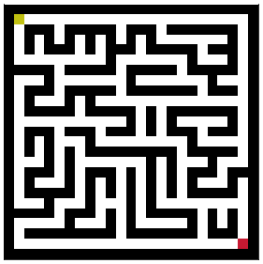

Figure 3: A pixel representation of the maze used to train our agent.

3.1 Performance vs Episodes

In this section we will measure the performance of the model both quantitatively and qualitatively for default parameter values.

As a quantitative measure of the agent's performance over learning episodes, we sum the total reward for each episode and calculate the difference between that and the total reward for the optimal path (found using a shortest path algorithm for graphs) through the maze. We call this value the reward cost.

We applied the model using the default parameters outlined in Section 2.7, applied a three point moving average for smoothing and plotted the results, Figure 4. The results show that initially the agent takes much longer paths to navigate the maze but learns from its experience over the number of training episodes and converges towards an optimal solution. This makes intuitive sense as the agent has no knowledge of its environment initially, so as it learns, it improves its performance. As the agent progresses through the training episodes, the epsilon decay dictates that the agent use its experience more and explore less, which causes an accelerated improvement. This can be seen in the non-linearity of the graph. After around 200 training episodes, the agent begins to converge to an optimal solution with some variance due to exploration.

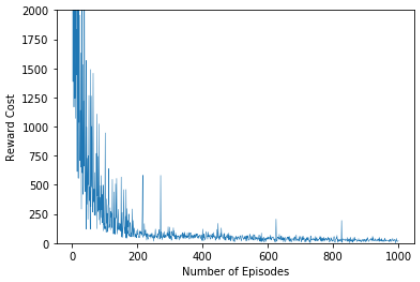

Figure 4: A plot of reward cost vs number of episodes, with 3 point moving average smoothing.

Our agent took 28.83 seconds to perform 1000 training episodes. We then tested the agent on the maze, and it found the solution seen in Figure 5. As a qualitative measure of performance, we tested our agents solution for optimality using a shortest path algorithm for graphs and found the solutions to be identical and therefore optimal.

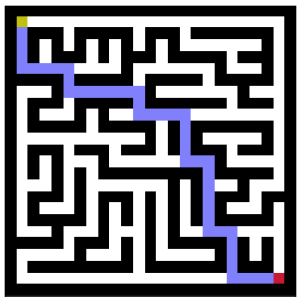

Figure 5: A pixel representation of the maze, with agent's solution .

3.2 Repetition with Different Parameters

Keeping all other parameters constant, we performed a grid search for α and γ over the values $\{0.25, 0.5, 0.75, 1\}$ to try and improve upon our performance. The results are plotted in a scatter plot matrix in Figure 6. From this scatter plot matrix, we see that for $\gamma = 0.25$, our model fails to converge to an optimal solution over 1000 training episodes. For greater values of γ and over each value of α, we see little difference in performance with all instances converging over similar numbers of episodes.

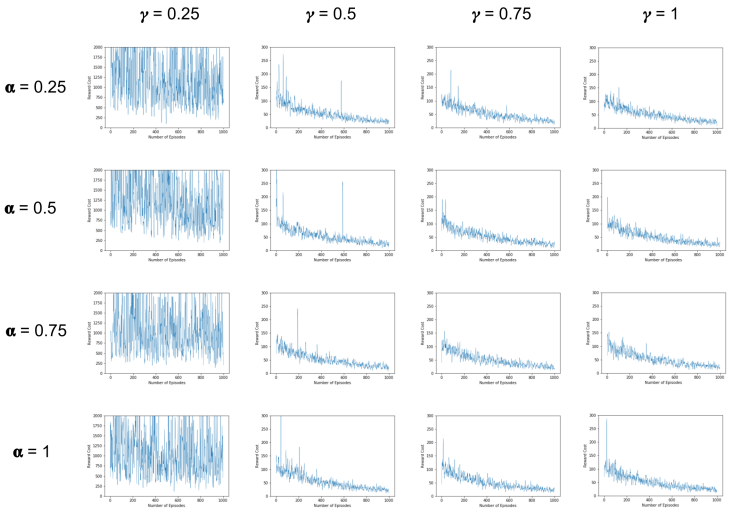

Figure 6: A matrix scatter plot of search for α and γ values.

Since we saw little difference between performance of the agent for $\alpha \in \{0.25, 0.5, 0.75, 1\}$ and $\gamma \in \{0.5, 0.75, 1\}$, we decided to choose the parameter values with the least computational cost. We evaluated the training time for each pair of parameters in the above range and found $\alpha = 0.75$ and $\gamma = 0.75$ to be the fastest with a training time of 18.72 seconds.

Next we decided to vary the exploratory factor to investigate its effects on model performance. We carried out a search for ϵ values over the range $\{0, 0.25, 0.5, 0.75, 1\}$, with $\alpha = 0.75$ and $\gamma = 0.75$ fixed.

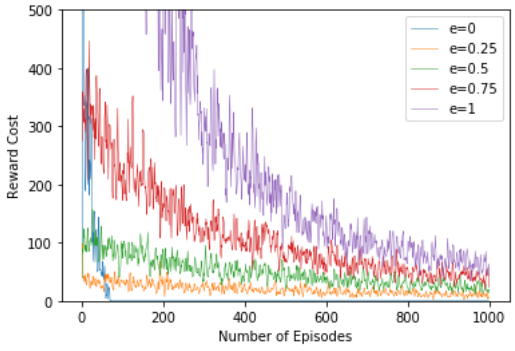

Figure 7: A scatter plot of search for ϵ with α and γ fixed.

When $\epsilon = 1$, the ϵ-greedy policy becomes a policy whereby the agent selects actions purely at random throughout the training process. After 1000 training episodes the agent managed to successfully converge to an optimal solution to our maze, albeit slower than smaller values of ϵ. The problem with a purely random policy is that the problem must be sufficiently small for the agent to explore all avenues numerous times through random choice. In our case, the

choice of maze was relatively small, so the agent was able to find an optimal solution. However, we expect this approach to be too computationally inefficient to merit its use as the problem is scaled up. When $\epsilon = 0$, the ϵ-greedy policy becomes a purely greedy policy. As discussed in Section 2.4, this can lead to sub-optimal solutions (although it did not in our maze) and so is undesirable for generalisation. For values of ϵ between 0 and 1, we found that $\epsilon = 0.25$ was the fastest to converge with minimal reward cost.

3.3 Randomising Starting Points

Once our agent was trained on this maze, we tested it over 10 randomly generated starting points within the maze. We found the agent had limited success, finding optimal paths from only 3 random starting points.

For a better picture of how the agent performed in this maze, we tested the agent over every starting position in the maze and mapped the results in the colourised map of the maze. In Figure 8, red pixels represent randomised starting points from which the agent failed to find an optimal solution. The remaining non-black pixels represent starting points from which the agent was able to find an optimal solution. We have differentiated the original optimal path by colouring it purple for clarity.

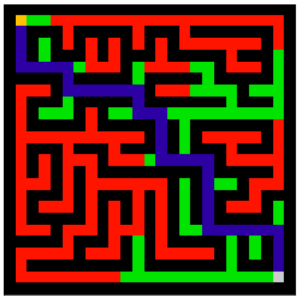

Figure 8: A colourised map of agent performance from different starting points in the maze.

From this colourised map, we see that the agent was successful from all positions immediately adjacent to the original optimal path. However, as the distance from the original optimal path increases, the likelihood of the agent being successful decreases. This relationship can be generalised as the probability of success is inversely proportional to the distance from the original optimal path.

3.4 Prior Domain Knowledge

In some cases, it might be that prior knowledge is known about a domain and the Q-matrix can be pre-populated to reflect that knowledge in the agent. We wanted to investigate the effects of pre-populating the Q-matrix to reflect prior domain knowledge. To do this we trained our model from two different starting points: one to reflect accurate and the other reflecting inaccurate prior domain knowledge. To represent accurate prior knowledge, we pre-populated the Q-

matrix with positive Q-scores for the first 15 steps along the known optimal solution, Figure 9 (left). To represent inaccurate prior knowledge, we pre-populated the Q-matrix with positive Q-scores along a non-optimal path, Figure 9 (right). In both instances we will use $\alpha = 0.75$, $\gamma = 0.75$ and $\epsilon = 0.25$ for our parameters. Both instances converged to an optimal solution, with training times of 14.94 seconds and 15.59 seconds respectively.

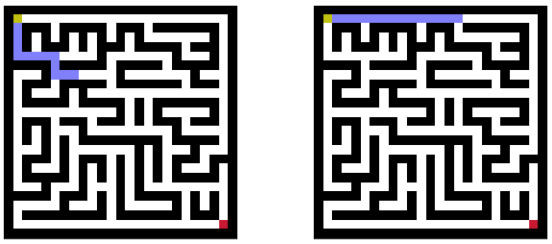

Figure 9: A representation of accurate and inaccurate prior domain knowledge reflected in our agent.

With no prior knowledge, the training time for the model is 14.97 seconds. This equates to a change in training time of -0.2% for pre-populating the Q-matrix with inaccurate prior domain knowledge and +4.1%. The difference is trivial in our case but, as the problem scales or in other domains, this could become very significant. This suggests that in order to reflect prior domain knowledge about the agent's environment, the user must be very confident that the knowledge is actually accurate to merit its inclusion. Inputting inaccurate information could cost you much more time than you stand to gain with accurate information, so starting from an assumption of no prior knowledge might be best.

4 Conclusions and Future Work

In this paper, we demonstrated that the Q-learning model is a good candidate to train an agent to perform a task in the simple environment of a maze. However, Q-learning has some limitations which should be discussed. Whilst Q-learning is proven to eventually converge to an optimal solution, this convergence is only guaranteed when training is extended to infinity. This is because the estimates for the quality of states-action pairs become more accurate as more information is gained. In our case, we could represent this problem graphically and apply a shortest path algorithm to check for convergence, but in other domains this may not be possible so some sort of threshold is usually required.

In Section 3.2, we analysed the performance of our model over different parameters during which we found $\{\alpha = 0.75, \gamma = 0.75, \epsilon = 0.25\}$ to have the best combination of convergence speed and computational cost for our task. An extension of this would be a more granular search, or to search over all parameters. We decided to perform a more limited parameter search in the interest of completing it in a timely manner.

In Section 3.4, we investigated how prior domain knowledge can affect performance when reflected in the model. We found that inaccurate information could

cost much more training time than accurate information could save. Whilst this gives an indication of the risks of instilling domain knowledge into an agent, it is only a simple example. For a more complete picture, a more robust investigation would need to be carried out with different maze sizes, different parameters and perhaps different domains.

In Section 3.3, we assessed our trained model's generalisability by testing it from randomised starting points within the maze. For the parameter values we tested, we found our model to have limited success. As an extension of our work, a grid search could be performed over parameter values, with final parameters selected with an appropriate balance between generalisability, speed of convergence and computational cost. Furthermore, a suggested improvement to the model would be to train the agent with a randomised starting point for each training episode. This would result in the agent repeatedly exploring a larger area of the maze from which we would expect a more generalisable trained model.

References

[1] 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, p. 529, 2015.

[2] D. Osmanković and S. Konjicija, "Implementation of q-learning algorithm for solving maze problem," in *MIPRO, 2011 Proceedings of the 34th International Convention*, pp. 1619–1622, IEEE, 2011.

[3] R. Tarjan, "Depth-first search and linear graph algorithms," in *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pp. 114–121, IEEE, 1971.

[4] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[5] fcogama, "mazegen.py." `https://gist.github.com/fcogama/3689650`, 2017.