Exploring Memory in Deep Reinforcement Learning for Partially Observable Tasks

George Watkins
Mathsys, Centre for Complexity Science
University of Warwick
`george.watkins@warwick.ac.uk`

With thanks to

Giovanni Montana
WMG
University of Warwick
`g.montana@warwick.ac.uk`

Malcolm Strens
WMG
University of Warwick
`malcolm.strens@warwick.ac.uk`

Abstract

Although Reinforcement Learning has enjoyed well-publicised success in a number of areas, adapting its methods for use in partially observable environments remains a challenge. The lack of memory inherent in the traditional RL formulation compounds these problems in contexts where past experiences might usefully supplement current observations.

Various approaches have been developed for such environments, extending the traditional methods by giving them the ability to 'remember' past experiences.

When Reinforcement Learning is paired with Artificial Neural Networks (known as Deep Reinforcement Learning), memory can be baked into the architecture of the network. A second approach involves maintaining a belief state that makes inferences about the unseen parts of the environment based on prior experiences, and providing the belief state to the agent as part of its observations.

The relative performance of these methods has not been fully investigated. In this paper we use a toy pursuer-evader environment to compare a belief-based approach to the traditional RL algorithms, and identify the most effective way to equip an agent with memory.

1 Introduction

Reinforcement learning is, in essence, the formalisation of 'learning by doing'. This is how we (as humans) often learn: the child that burns her hand touching a hot stove will quickly learn not to do it again. Indeed, many of the concepts in reinforcement learning were originally inspired by biological learning systems [1].

We are all familiar with the ideas behind reinforcement learning: the 'carrot-and-stick' approach, used to promote (or discourage) certain behaviours, is essentially real-world RL. In the mathematical formalisation, we attempt to quantify the carrot and the stick.

Reinforcement learning has been used successfully to train computers to perform highly complex tasks, and is seen by some as the most likely source of Artificial General Intelligence (AGI). Many of the best-known achievements involve developing AIs that can attain super-human levels of performance in games: Backgammon [2], Atari video games [3] and, perhaps most famously, Go [4].

These are (in general) perfect information games: the agent can observe the complete state of the environment to inform its choice of action. (Note that for some Atari games it is necessary to include the last few game screen frames as part of the current state to achieve this [5].) Such environments are called *fully observable*.

There are many environments in which the agent's observations do not provide perfect information. These are known as *partially observable*. Training autonomous vehicles is a good example of a reinforcement learning problem in a partially observable environment. There is a lot of (potentially useful) information that the agent (in this case, the car) cannot see; anyone who has been stuck behind a lorry on a narrow road will be familiar with this!

In classic reinforcement learning methods, the agent determines which action to take based solely on its observation of the current state. When the environment is partially observable, this decision must therefore be made with imperfect information, with the unseen parts effectively ignored. This naturally leads to sub-optimal behaviour.

Often, however, the state of the unseen parts is dependent on what has been observed previously; as such if the agent were to maintain a memory it might be possible to better model the parts it can't see. Returning to the previous example, if a lorry pulls out of a side street, we don't immediately forget about the cyclist 100 metres up the road just because we can no longer see it - we can make a pretty good prediction about where it's likely to be based on where we last saw it and how fast it was travelling.

There are two approaches to equiping an agent a memory. The first is to give the agent the ability to maintain its memory itself and combine this with its (partial) observations to choose actions more effectively. This approach requires a departure from traditional reinforcement learning methods for which memoryless-ness is a fundamental property.

The second option is to use an external memory (i.e. we maintain it on the agent's behalf). We then combine it with the current observation to infer information about the unseen components that we then feed to the agent in place of the state. This approach is predicated on the assumption that the information we infer and give to the agent is correct, which is not necessarily certain.

For example, in real world contexts we often don't know anything (with complete certainty) about the dynamics of unseen elements: for all we know the cyclist could have been intending to turn off the road just after the lorry obscured it from view. Making inferences ourselves means they are susceptible to human biases.

Furthermore, the achievements of AlphaGo and similar algorithms demonstrate the potential of RL to achieve super-human performance. So by providing the agent with information we have derived ourselves, we force the agent to base its choice of action on 'human-limited' information; left to its own devices, it is possible the agent might come up with its own, better model of the unseen elements of the environment.

The question of how best to equip learning agents with a memory in partially observable environments is an open problem. In this paper we compare, for the first time, the training time and ultimate performance of traditional RL approaches with various alternative methods designed to equip the agent with a memory.

2 Reinforcement Learning: Theory

2.1 RL: Overview

The following exposition of reinforcement learning is inspired by an example in Sutton and Barto's Reinforcement Learning: An Introduction [1].

The traditional game of noughts-and-crosses is played by two players on a 3×3 grid. Players take turns to place their symbol ($\circ$ or $\times$) on the grid, aiming to make a line (vertically, horizontally or diagonally) of 3 of their own symbols.

The following example uses an adapted version of the game played on a 3×4 grid. The goal is still to get 3-in-a-row, but this version of the game is more interesting as optimal play from each player does not result in a draw; rather, there is a winning strategy for the first player [6] that we could use RL to discover. For the purpose of this example noughts will go first, and this will be the player we train.

The key components of the reinforcement learning method are the *environment*, the *agent* and the *objective*. In the noughts-and-crosses example:

- *Environment*: The noughts-and-crosses game itself.
- *Agent*: The noughts player.
- *Objective*: To win the game.

Reinforcement learning is effectively learning from experience, and each complete experience is called an *episode*. An episode is made up of timesteps, and at each timestep the agent observes the *state* of the environment and chooses an *action* to perform.

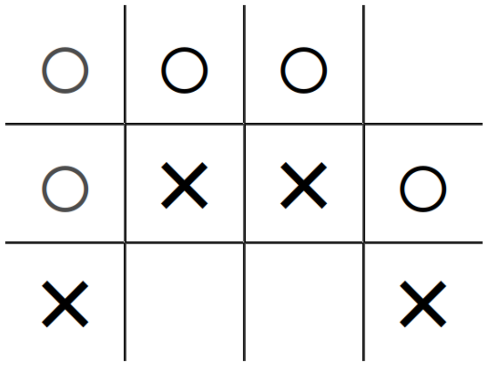


Figure 1: Noughts and crosses grid in which noughts played first and has just won.

The mapping from states to actions is called the *policy* - this defines the agent's strategy, and specifies what to do from each state. (Note that in this work we focus on deterministic policies in which each state is mapped to a single action that we choose with probability 1.) Once a *terminal state* is reached, the episode ends.

When the agent takes an action it may receive a *reward*. This is a scalar that provides an immediate indication as to whether the action was 'good' or 'bad', and should be constructed in such a way that it relates directly to the objective. The *return* is the sum of all future rewards up to the end of the episode, which we usually *discount* to reflect a preference for rewards in the near future over those further away (and also to ensure that the return in episodes that never terminate is finite). In the noughts-and-crosses example:

- *Episode*: A full game of nought-and-crosses.
- *Timestep*: One move in the game. Because players take turns, the agent plays a $\circ$ at every other timestep.
- *State*: The current grid configuration, populated with all the $\circ$'s and $\times$'s that have been played so far.
- *Action*: Placing a $\circ$ on the board (recall our agent is playing as noughts).
- *Terminal state*: Any grid configuration on which there are 3 $\circ$'s or 3 $\times$'s in row. (Note that not all terminal states are obtainable; if both $\circ$ and $\times$ have 3-in-a-row, the game would have already passed through a terminal state and therefore ended.)
- *Reward*: Given that our goal is to train the agent to win, a natural reward function might give a reward of $+1$ for playing a move that completes a 3-in-a-row, -1 for a move that allows the opponent to complete a 3-in-a-row on their next turn, and 0 for all other actions. One disadvantage of this kind of 'delayed gratification' reward function (in which non-zero rewards are only received at the end of the episode) is that this information can take a long time to propagate back to states that arise early on in the game. One way to overcome this is to define a reward function that rewards actions that get the agent 'close' to winning. For example, we could offer a small reward (of, say, 0.1) for playing actions that result in a state in which the agent has 2-in-a-row, but it is important to be cautious when adapting the reward function to ensure it does not give rise to unexpected and suboptimal behaviour.

Finally, we define the *value function*. Value functions can take different forms; in this work we will focus on state-action value functions $Q(s, a)$.

In the case of noughts-and-crosses, the value function tells us how 'good' it is to perform any move from any current board configuration. Specifically, it gives the expected return for playing action a in state s and following a certain policy π thereafter (recall the return is the discounted sum of future rewards).

Natually, then, the optimal value function $Q^*(s, a)$ gives the expected return for playing action a in state s and following the optimal policy π^* thereafter.

2.2 RL: Formalisation

Many reinforcement learning problems can be accurately modelled as a Markov Decision Process (or MDP). This formulation provides access to a suite of mathematical methods that can be used to train an agent based on its experience.

An MDP is defined as a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$ where:

- $\mathcal{S}$ is the (finite) set of states in the environment
- $\mathcal{A}$ is the (finite) set of actions available to the agent
- $\mathcal{P}$ is the state transition probability function, often written $\mathcal{P}_{ss'}^{a}$ (see below for full definition).
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the expected reward function, often written $\mathcal{R}_s^a$ (see below for full definition).
- $\gamma \in [0, 1]$ is the discount factor.

These are the key elements of an MDP. We supplement them with the following definitions:

$s \in \mathcal{S}$	A state of the environment.
$S_t \in \mathcal{S}$	The state at time t.
$\mathcal{A}(s)$	The set of possible actions from state s.
$a \in \mathcal{A}(s)$	An action.
$A_t \in \mathcal{A}(s)$	The action taken at time t.
$\pi : \mathcal{S} \mapsto \mathcal{A}(s)$	A policy that maps a state s to the action a to be taken from s. (Note that this definition is for deterministic policies; as noted previously, all policies in this work are of this type.)
$\mathcal{P}_{ss'}^{a}$	The probability of transitioning to state s' given that we took action a from state s, i.e. $\mathcal{P}_{ss'}^{a} = P[S_{t+1} = s' \| S_t = s, A_t = a]$
r_t	The reward at time t. Usually depends on s_{t-1}, s_t and a_{t-1}.
R_t	The expected (discounted) return from time t, i.e. $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$
$\mathcal{R}_s^a$	The expected (immediate) reward for performing action a from state s, i.e. $\mathcal{R}_s^a = E[r_{t+1} \| S_t = s, A_t = a]$
$q_\pi(s, a)$	The value of performing action a from state s under policy π (i.e. assuming we subsequently act according to policy π).
$Q(s, a)$	An estimate of $q_\pi(s, a)$.

The classic reinforcement learning diagram shown in Figure 2 illustrates how an MDP models the interaction between agent and environment. The feedback loop can be utilised to train the agent to improve its behaviour.

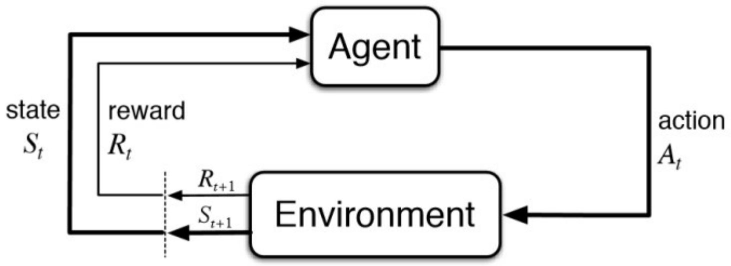


Figure 2: Dynamics of the agent-environment interaction in reinforcement learning. Given the current state, the agent submits an action to the environment and the environment returns a reward together with information about the new state. The agent chooses the next action and the process repeats.

Ultimately the goal is to train the agent to behave optimally, which means choosing the 'best' action at every decision point.

We have already defined a way to determine the 'good-ness' of selecting an action in a given state: the state-action value function. Assuming the agent knows it, the optimal action from each state can be determined by acting greedily with respect to the function (i.e. choosing the action that gives the highest output).

As such, the reinforcement learning problem reduces to finding the value function.

(Note that this is for model-free RL in which the agent purely learns the optimal policy. Model-based RL, in which the agent learns $\mathcal{P}$ and $\mathcal{R}$, is slightly different.)

2.3 Finding the value function: Q-learning

Q-learning [7] is a simple algorithm for finding the value function. The following update rule describes how to assimilate new experiences into the current estimate of the value function.

$$Q(S_t, A_t) \leftarrow (1-\alpha)Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a)) \tag{1}$$

α is the *learning rate* and it determines how much weighting to give new experiences relative to our current estimate of the value function.

Ultimately the optimal policy will be generated by choosing actions greedily with respect to the value function, but for convergence we require all state-action pairs to continue to be visited during training and a greedy policy does not achieve this. Instead, we need to follow a policy that balances exploration and exploitation. Q-learning is therefore an example of an *off-policy* algorithm (which is more accurately described as 'learning by watching').

In the Q-learning algorithm, typically an ϵ-greedy policy is followed: with probability ϵ $(0 < \epsilon \ll 1)$ the agent chooses a random action, and with probability $1-\epsilon$ it chooses the best available action according to the current estimate of the value function.

Note that the Q-learning algorithm only works if both the state space $\mathcal{S}$ and action space $\mathcal{A}$ are discrete (otherwise the probability of revisiting the same state-action pair would be 0) and not too large.

2.4 Finding the value function: Deep Q-learning

The Deep Q-learning [5] algorithm can approximate the value function when the state space is very large and even continuous (in certain cases).

In this method, rather than storing a value for every state-action pair, the value function $Q(s, a)$ is approximated using a neural network (known as a *Deep Q-Network* or DQN). The state is provided as input, and there is an output node for each action: if the state s is used as input, the output corresponding to action a gives $Q(s, a)$. These values can be used to generate a greedy (or ϵ-greedy) policy.

The neural network is trained using supervised learning. The training data are the experiences; the 'new experience' part of the Q-learning update rule (highlighted below) is used as the target.

$$Q(S_t, A_t) \leftarrow (1-\alpha)Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a))$$

The loss function is simply the squared difference between the current estimate of $Q(s, a)$ and the target:

$$\begin{aligned} \text{loss} &= (\text{prediction} - \text{target})^2 \\ &= ([Q(S_t, A_t)] - [R_{t+1} + \gamma \max_a Q(S_{t+1}, a)])^2 \end{aligned}$$

A potential problem arises here: in traditional supervised learning the targets remain stationary. However in deep Q-learning the target changes as the network is trained. Two strategies are used to combat this:

- *Two networks*: Rather than use the same network to generate both the prediction $Q(S_t, A_t)$ and the $(\max_a Q(S_{t+1}, a))$ part of the target (which may lead to divergence), two parallel networks (with the same architecture) are maintained. One is used to generate the predictions, and its parameters are updated by training the network. The second is used to generate the targets. The target network parameters are not updated by training; instead, they are periodically updated *towards* the weights in the prediction network (similar to how the learning rate is used in the Q-learning update rule).
- *Experience replay*: The data provided as input to a neural network should be independent and identically distributed (iid). But if the prediction network is updated *online* (i.e. each experience is used immediately to train the network), successive input states will be just one action apart, and therefore highly correlated. To get input data that is close to iid, a buffer of past experiences is maintained and the prediction network is updated using a random sample taken from the buffer.

2.5 Partially Observable Environments

The MDP formulation assumes that the environment is fully observable: when the agent chooses actions it is equipped with perfect information about the current state. Partially observable environments are instead modelled by a Partially Observable Markov Decision Process (POMDP) with the augmented tuple definition $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \Omega, \mathcal{O}, \gamma >$ where:

- Ω is the set of possible observations
- $\mathcal{O}$ is the set of conditional observation probabilities, i.e.
  $\mathcal{O}(o|a, s')$ = *probability of making observation* o *given that we have just take action* a *and arrived in state* s'

Note that an MDP can be reformulated as a POMDP simply by setting $\Omega = \mathcal{S}$ and making $\mathcal{O}$ deterministic, i.e.

$$\mathcal{O}(o|a, s') = \begin{cases} 1 & \text{if } o = s' \\ 0 & \text{otherwise} \end{cases}$$

Conversely a POMDP can't be turned directly into an MDP. However by maintaining a *belief state* - defined as a probability distribution over the set of states - and providing this to the agent instead of a partial observation of the state, a POMDP can be converted to a Belief MDP.

2.6 Belief MDPs

In a Belief MDP the agent receives a belief state rather than an observation of the environment state. In effect we introduce a filter between the environment and the agent (see Figure 3). The filter receives the observations, and stores and updates the belief state. The belief state is then passed to the agent analogously to the environment state in an MDP. Given that the agent always knows the belief state, a Belief MDP is fully observable.

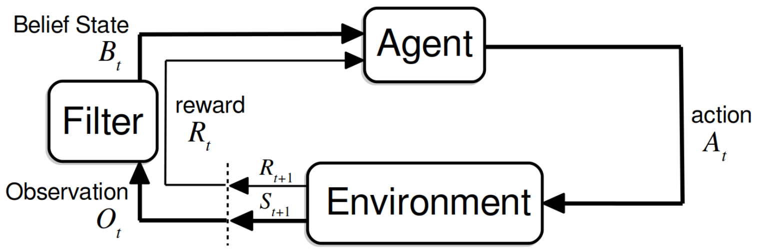


Figure 3: Dynamics of the agent-environment-filter interaction in a Belief MDP. Given the current belief state, the agent submits an action to the environment and the environment returns a reward together with a (partial) observation of the new state. The filter uses the observation to update the belief state which is passed to the agent and the process repeats.

A Belief MDP is defined as a tuple $< \mathcal{B}, \mathcal{A}, p, r, \gamma >$ where:

- $\mathcal{B}$ is the set of belief states (probability distributions over the states in the environment).
- $\mathcal{A}$ is the (finite) set of actions available to the agent (the same as the underlying POMDP).
- p is the belief state transition probability function.
- $r : \mathcal{B} \times \mathcal{A} \to \mathbb{R}$ is the expected reward function on belief states.
- $\gamma \in [0, 1]$ is the discount factor (the same as the underlying POMDP).

For a belief state $b \in \mathcal{B}$, $b(s)$ is the probability that s is the true state. Whenever the agent takes an action a and moves to state s', the filter uses a Bayesian update to incorporate the extra information into the new belief state b'.

$$b'(s') = \eta \left[\mathcal{O}(o|a, s') \sum_{s \in \mathcal{S}} \mathcal{P}(s'|s, a) b(s) \right]$$

where η is just a normalizing constant,

$$\eta = \frac{1}{\sum_{s' \in \mathcal{S}} \mathcal{O}(o|a, s') \sum_{s \in \mathcal{S}} \mathcal{P}(s'|s, a) b(s)}$$

Maintaining a belief state effectively equips the agent with a memory. Although the process itself is still Markovian - the agent only has access to the current belief state - the belief state holds information from the entire history of the episode. As such the agent is able to make decisions based on observations at timesteps much earlier in the episode.

Note however that because they are probability distributions, belief state spaces are continuous, even when the state space of the underlying POMDP is discrete and finite. As such, Belief MDPs are not compatible with Q-learning.

However the value function $Q(b, a)$ can still be approximated using Deep Q-learning. The belief state is a probability distribution over the states, so we can simply use an architecture with an input (in $[0, 1]$) for each state.

Clearly neural networks are powerful tools, and this is not the only way they can be utilised to equip agents with memory in POMDPs.

2.7 Finding the value function: Deep Recurrent Q-learning with LSTM

There are many tasks in the field of artificial intelligence that require an understanding of context (which is another way of saying memory).

The canonical example is text generation. Given a sequence of words, for example *Jane used to live in France. She speaks fluent* we can make a pretty confident guess that the next word will be *French*. However a Markovian agent that can only see the current word (or even the current sentence) would not be able to identify this.

Recurrent Neural Networks (RNNs) can solve problems such as these. When we have sequential data (like the words in a coherent sentence, a time series of stock prices, or successive states in a POMDP), the RNN maintains a hidden state that captures contextual information.

Effectively the hidden state is just the output of the network. But rather than being immediately discarded, it is re-used as part of the input when the next datapoint is fed into the network. Maintaining the hidden state in this way ensures it retains information from previous timesteps' observations.

However as time passes the information pertaining to observations long in the past fades. So, using our example above, if the two sentences *Jane used to live in France* and *She speaks fluent* were separated by a lot of unrelated information, the fact that Jane used to live in France may have been forgotten.

This issue can be resolved by a particular form of enhanced RNN. An LSTM (Long Short Term Memory) maintains two states: the hidden state (or working memory) and the cell state (or long-term memory) that retain information about the recent and distant past repectively.

Figure 4 [8] highlights the difference between the RNN and LSTM architectures and summarises what is going on 'under the hood' of an LSTM.

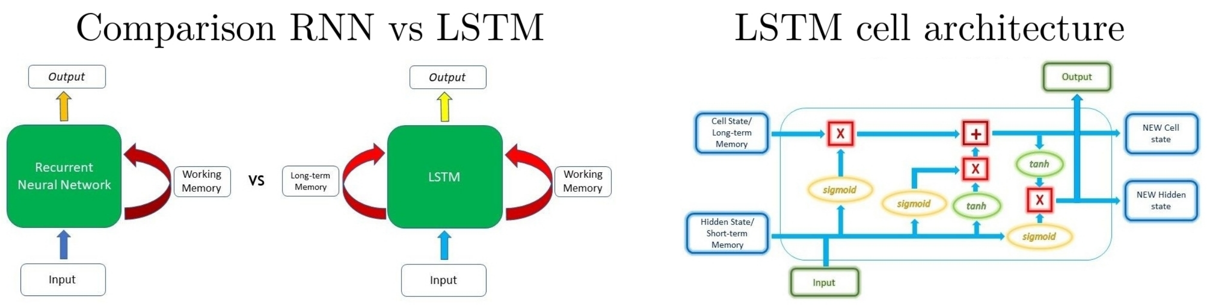


Figure 4: *Left*: Comparison of RNN and LSTM architectures. The RNN has one hidden state, while LSTM maintains two hidden states representing both long- and short-term memory.
Right: Architecture of an LSTM cell showing 'Input', 'Forget' and 'Output' gates. These gates (the paths through the cell) enable it to store both the recent and historic context.

3 Related Work

Reinforcement Learning The Q-learning algorithm as described above was formalised in [7]. It is a relatively simple (but powerful) algorithm that can form the basis of far more sophisticated methods [9]. One such method is Deep Q-learning [3], which uses a Deep Q-Network (DQN) to train agents, and has achieved human-level and often super-human performance at various Atari games [5]. However these methods are known to fail in certain partially observable environments in which memory is required.

There have been a number of efforts made to equip reinforcement learning agents with a memory.

Memory: Multiple-state observations One simple approach is to include previous states alongside the current state as part of an observation. For example [3] included the last four game screen frames in their algorithm for playing Atari games, but this has drawbacks in environments in which a longer memory is needed, or the required memory length is variable.

Memory: Architecture-based methods Deep Recurrent Q-Networks (DRQN) use a neural network architecture that incorporates an LSTM layer in order to maintain a memory [10]. Using this architecture, the authors found performance was at least as good as DQN, and better when trained with perfect information but evaluated with partial observability (in the form of Atari games with flickering screens). However their tests suggested that 'recurrency confers no systematic advantage when learning to play the game'.

[11] took a deeper look at the performance of DRQN, with a focus on partially observable environments. Using the customisable game Minecraft the authors showed that DRQN was harder to train and ultimately yielded inferior performance to the 'simpler' DQN. These results were surprising, and they state that 'further investigations to better understand this behaviour [is required]'.

[12] also used Minecraft as the platform for their environment, and compared three new architectures with DQN and DRQN. Their context-dependent memory retrieval method, employing an architecture with both feedback and recurrency, performed better than DQN and DRQN in tasks in which memory is required, and generalizes more effectively to unseen environments.

Memory: Other methods [13] notes that 'recurrent neural networks are known to have difficulty in performing memorization', and proposed a new method they called Memory Networks (MemNN). Using this approach the neural network is coupled with a (separate) memory component, and the network is trained to operate effectively with the memory. While the investigations were limited to a narrow set of question answering tasks, the performance on the most difficult of these was significantly better than DRQN; indeed DRQN often failed where MemNN completed the task successfully.

A similar approach was taken by [14]. They used a Neural Turing Machine [15] whereby the neural network interfaces with an external memory. This method had limited success: the main finding was how difficult it is to train an agent to effectively utilise the memory.

Memory: Belief-based methods Using an agent's observations to infer a belief state is not a new idea. Since the early work in this area [16] a large number of sophisticated methods for complex contexts have been put forward.

Grid-based approaches [17] [18], point-based value iteration [19] and Monte Carlo methods [20] have all been employed, but there has been very little work comparing belief-based Reinforcement Learning approaches with the traditional techniques.

We end on a cautionary note: we must be careful when making 'manual' inferences and feeding them to learning algorithms. By way of comparison, feature engineering - whereby the representations of the data are dependent on the problem itself and constructed by hand - has been completely supplanted because 'deep learning completely automates this step' [21]. It is possible that belief-based methods may suffer the same fate.

That said, there is little evidence in the literature that the benefits of using a belief state have been fully investigated. This paper compares a belief-based approach to the more traditional RL algorithms, and highlights the most effective approach for equipping an agent with memory.

4 Methodology

This work uses a simple pursuer-evader gridworld task to investigate the impact of equiping an agent with a memory in partially observable environments. The agent to be trained is the pursuer (the 'cat'), which will optimise its strategy for catching the evader (the 'mouse'). The mouse will not learn from experience, and will choose actions uniformly at random.

At the start of each episode both agents (the cat and mouse) are placed randomly on a board. The positions that they can occupy are discrete: when the board is viewed as a coordinate system the cat and mouse occupy positions that can be specified by 2D integer coordinates. Note that by convention we will specify an agent's position as (row, column) which is the reverse of a typical x-y coordinate system. Furthermore row and column indexing starts at 0, with row 0 at the top and column 0 on the left.

Figure 5 shows a 4×6 board with several interior walls (shown in black). This is the board that will be used for all the investigations presented in this paper. In the Figure, the mouse is in position $(0, 0)$ and the cat is in position $(1, 2)$.

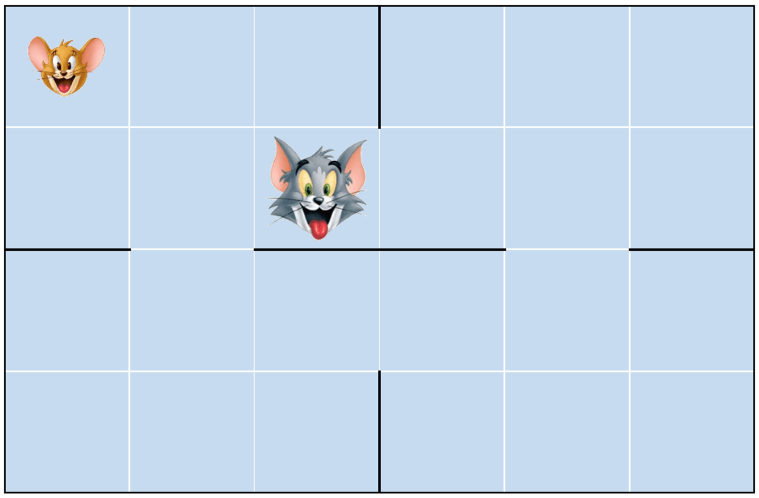


Figure 5: Example of 4×6 board with mouse in position $(0, 0)$ and cat in position $(1, 2)$. Note that positions are of the form (row, column) with indexing starting at 0. Row 0 is at the top and column 0 is on the left.

The cat and mouse navigate around the board until the episode ends. At each timestep both the cat and mouse (simultaneously) choose and execute a move; these are the actions. The set of actions is comprised of moves to adjacent squares (including diagonally) plus the 'stay still' action. As such there are 9 actions to choose from which can be thought of as compass directions together with the stay still action, X. So the complete set of actions available to each agent is {NW, N, NE, W, X, E, SW, S, SE}.

Note that the actions are deterministic (the action that is chosen is always the one that is executed), and the agents do not know what action the other has chosen.

If either agent attempts an invalid action (by trying to leave the board or walk through a wall), they remain where they are. For example in the board configuration shown in Figure 5, there are actually six actions the mouse could choose that would result in it remaining in position $(0, 0)$: {NW, N, NE, W, X, SW}.

Partial observability is achieved by using opaque walls and giving the cat limited sight range. Figure 6 shows the squares the cat can see from square $(1, 2)$ with different sight ranges.

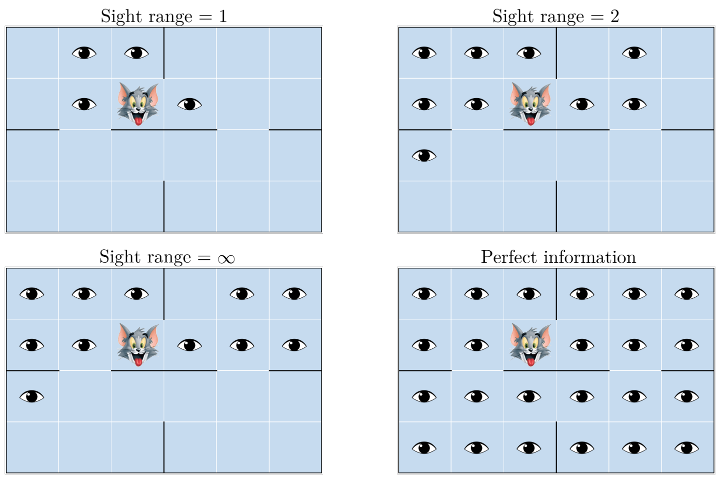


Figure 6: Demonstration of what the cat can see from $(1, 2)$ with different sight ranges. Note that walls are opaque, and the 'sight line' between two squares is a straight line between their centres. If the sight line intersects a grid vertex, the sight is blocked by any wall with an endpoint at that vertex (which is why the cat can't see square $(2, 1)$).

An episode ends when the cat has caught the mouse, so the terminal states are board configurations in which the cat and mouse occupy the same square. The most natural reward function is to give the cat a reward of 1 for reaching a terminal state and 0 otherwise. However empirically this results in longer training times (as the single reward takes time to propagate away from the terminal states) so instead we use the following reward function that includes a 'proximity penalty':

$$r_t = \begin{cases} b_h \times b_w & \text{if } S_t \text{ is terminal} \\ -1 \times \text{separation}(pos_c, pos_m) & \text{otherwise} \end{cases}$$

where b_h = board height, b_w = board width, and pos_c and pos_m are the cat and mouse positions respectively. The separation function is defined as

$$\text{separation}(pos_c, pos_m) = \text{the minimum number of moves in which the cat could reach the mouse assuming the mouse doesn't move and ignoring walls}$$

This is obviously a somewhat contrived reward function, and it is important to be mindful that such functions could result in unusual behaviour (if, for example, the cat prioritises getting 'close' to the mouse over actually trying to catch it). However the reward for catching the mouse is an order of magnitude greater than the proximity penalty, and the discount factor we use will be sufficiently close to 1 to mitigate this risk.

Four algorithms for training the cat will be tested and compared using this environment. The first and last algorithms don't have any sort of memory. The second and third incorporate memory in the ways described previously:

- Deep RL using a DQN without belief state.
- Deep RL using a DQN and belief state.
- Deep recurrent RL using a DRQN.
- Q-learning (which doesn't have a memory but is included to provide a baseline).

The hyperparameters for each algorithm (including neural network architectures) are presented in Appendix A.

Deep Q-learning is the only algorithm that will be run with a belief state. As discussed previously, providing this extra information converts the problem to a Belief MDP making it incompatible with Q-learning (which requires a discrete state space). And Deep Recurrent Q-learning already has its own form of memory so doesn't require a belief state.

Figure 7 demonstrates how the cat's knowledge of the board varies depending on whether a belief state has been provided (in this example we are using a sight range of 2). At timestep t the cat can see the mouse (so the probability distribution can be modelled as a Kronecker delta function). But its next action moves the cat to a square from which the mouse is out of sight.

If no belief state is provided, the cat 'forgets' what it has observed previously and assigns equal probability to every square that it cannot see (note that the squares it *can* see are naturally given a probability of 0).

On the other hand, if a belief state is provided, the cat 'remembers' that the mouse was in $(0, 0)$ at timestep t, so it could only have moved to one of four positions at timestep $t + 1$:

$(0, 0)$	Moves {NW, N, NE, W, X, SW} would cause the mouse to end up in this position.
$(0, 1)$	Mouse could not be in this position as the square is within the cat's sight range and the cat can see it is empty.
$(1, 0)$	Move {S} would cause the mouse to end up in this position.
$(1, 1)$	Mouse could not be in this position as the square is within the cat's sight range and the cat can see it is empty.

So the belief state is as follows, which is shown graphically in the bottom right image of Figure 7.

$$P(S_{t+1}) = \begin{cases} 6/7 & \text{if } S_{t+1} = (0, 0) \\ 1/7 & \text{if } S_{t+1} = (1, 0) \\ 0 & \text{otherwise} \end{cases}$$

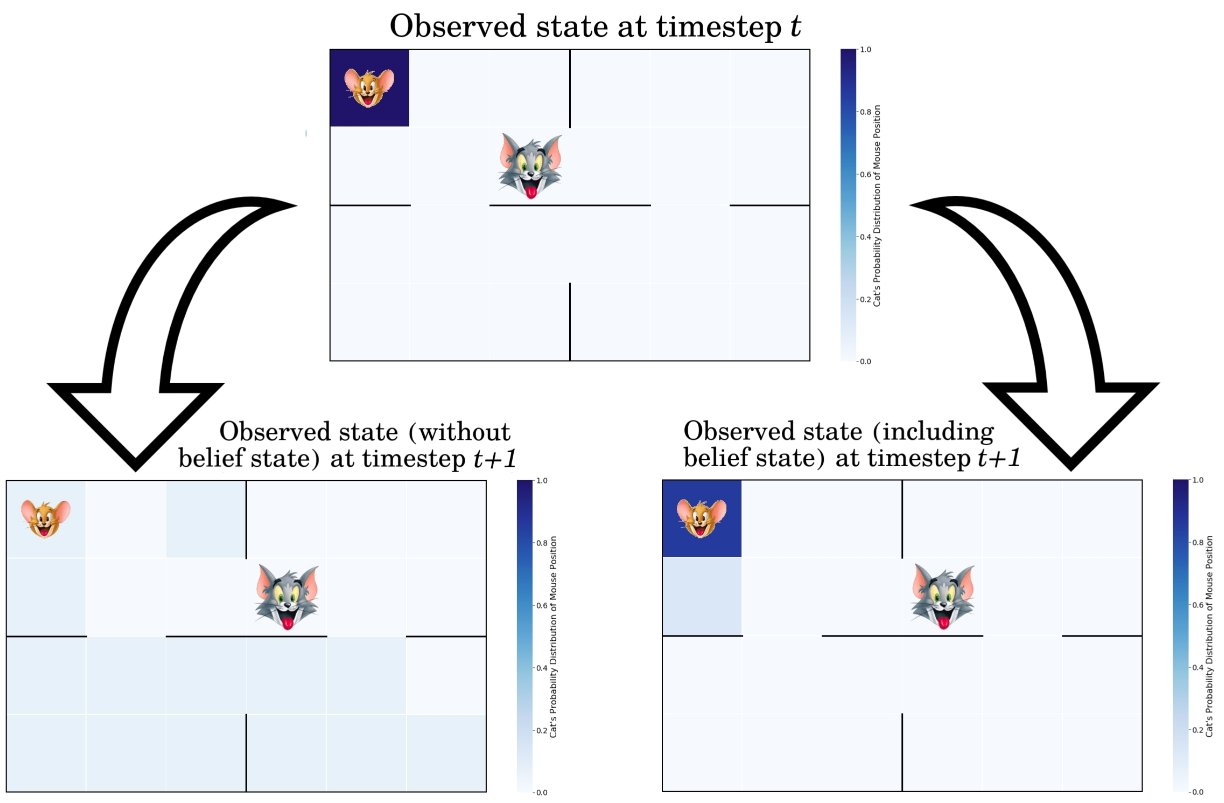

Observed state at timestep t
Cat's Probability Distribution of Mouse Position
Observed state (without belief state) at timestep $t+1$
Observed state (including belief state) at timestep $t+1$
1.0
0.8
0.6
0.4
0.2
0.0

Figure 7: Demonstration of how the belief state influences the cat's model of the environment. By providing a belief state, the cat can narrow down the possible positions of the mouse. If a belief state is not provided, the cat forgets any previous sightings of the mouse and assigns a uniform probability distribution over the squares it cannot see.

To determine which squares are obscured from the cat's view by walls, it is assumed that the cat 'looks' from the centre of its square, s_c. Another square *s'* can be seen if and only if a straight line can be drawn between the centres of s_c and *s'* without intersecting any walls.

Note that on the board in the bottom left of Figure 7, the square $(0, 2)$ (northwest of the cat's position) cannot be seen. By convention, if the sight line between s_c and *s'* passes through a vertex on the board at which a wall starts/ends, the square *s'* is considered to be out of sight.

All the code used to generate the environment, train the agent and analyse the results can be accessed via the Github link in Appendix B.

5 Results

In this section we compare the training speed and ultimate policy performance of the four algorithms described above:

- Deep Q-learning (without belief state)
- Deep Q-learning (using belief state)
- Deep Recurrent Q-learning
- Traditional Q-learning

As noted previously, all investigations will be conducted using the same basic game board (see Figure 5)

The two questions we seek to answer are:

- To what extent does using a belief state improve performance in memory-dependent tasks?
- How do methods that incorporate memory compare to each other, and to the traditional memoryless approaches?

5.1 Effects of belief state

In this section we compare the performance of the Deep Reinforcement Learning algorithm in the following situations:

- Imperfect information, without belief state.
- Imperfect information, using belief state.
- Perfect information.

In the first two cases, imperfect information is achieved by using a partially observable environment with opaque walls and a sight range of 2.

5.1.1 Training

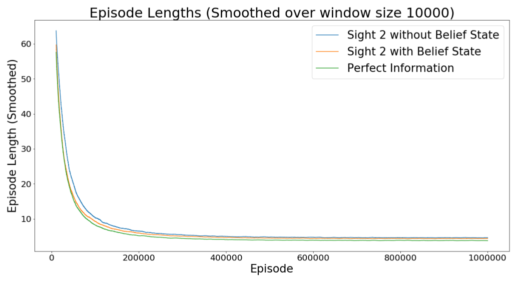


Figure 8: Comparison of episode lengths during training using Deep RL algorithm.

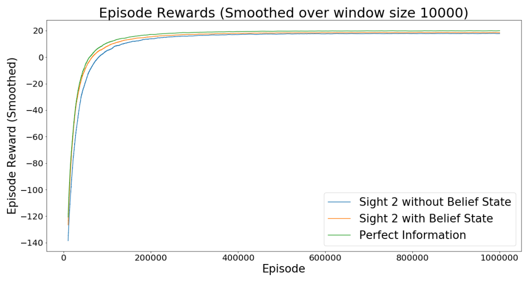


Figure 9: Comparison of episode rewards during training using Deep RL algorithm.

Figures 8 and 9 compare the lengths and rewards over the 1 million training episodes for the Deep RL algorithm. Note that while we would expect the episode lengths to reduce and episode rewards to rise, they will not be perfect inverses of one another because of the proximity penalty that was included in the reward function.

Focusing on the first 100,000 training episodes makes the differences clearer.

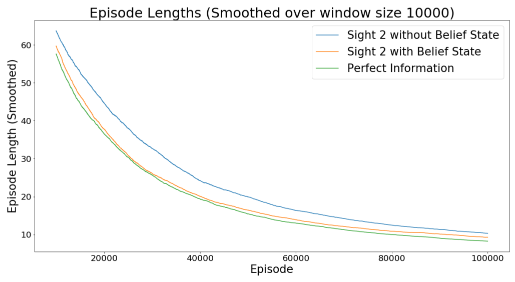


Figure 10: Comparison of episode lengths during first 100,000 training episodes using Deep RL algorithm.

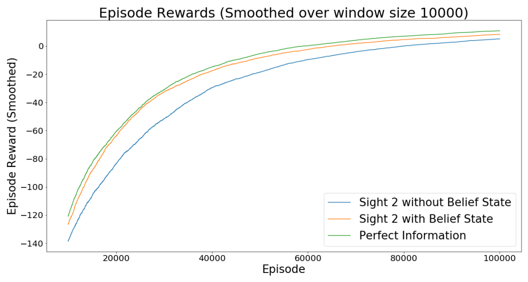


Figure 11: Comparison of episode rewards during first 100,000 training episodes using Deep RL algorithm.

Naturally, training with perfect information is the most efficient, and using a belief state in the partially observable environment is better than not. Interestingly, however, training in the partially observable environment using a belief state is not much less efficient than in the fully observable case. This suggests that as long as the inferences made by the belief state accurately model the environment dynamics, the belief state can almost compensate for the lack of information.

5.1.2 Policy Performance

In each case the episode lengths and rewards both appeared to be settling down to some limit. But these limits would still be under an ϵ-greedy policy. To properly compare ultimate performance we must follow a policy that acts greedily with respect to the Q-function approximated by the network.

Figure 12 shows the same pattern we observed during training: the best policy is generated in the fully observable environment, and using a belief state yields a better policy than not. Note that the more interesting comparison is between episode rewards as this is what the loss function relates to. Because we chose a sensible reward function, the episode lengths follow a similar pattern.

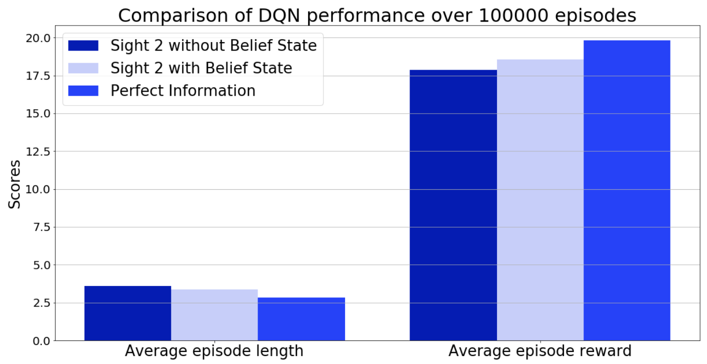


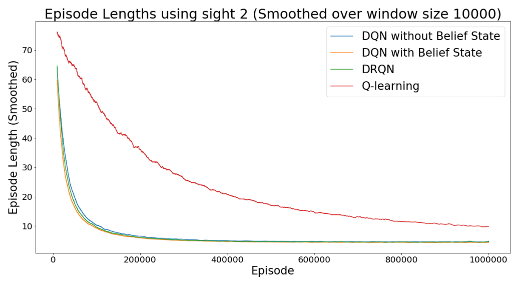
Figure 12: Comparison of ultimate policy performance using Deep RL.

5.2 Comparison of algorithms with memory (using sight range 2)

In this section we compare the performance of the following algorithms using the same board as previously, and an agent with sight range 2.

- Deep RL using a DQN without belief state.
- Deep RL using a DQN and belief state.
- Deep recurrent RL using a DRQN.
- Q-learning (included to provide a baseline).

5.2.1 Training



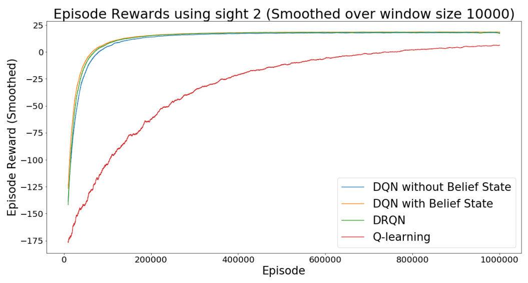


Figure 13: Comparison of episode lengths during training using various algorithms.

Figure 14: Comparison of episode rewards during training using various algorithms.

Figures 13 and 14 show that training takes significantly longer using the Q-learning algorithm than the others (which all use neural networks to approximate the Q-function).

This is because the number of possible inputs to the Q-function is huge and, unlike in the network-based aproaches, they must all be trained individually. For the 4×6 board we are using for our investigations:

$$\begin{aligned} \textit{no of possible inputs} &= \textit{no of states} \times \textit{no of actions at each state} \\ &= (\textit{board height}^2 \times \textit{board width}^2 + \textit{board height} \times \textit{board width}) \times 9 \\ &= 83160 \end{aligned}$$

Each state-action pair must occur a sufficient number of times for the randomness in the system to average out. Furthermore, the positive rewards need to propagate back from the terminal states. These factors result in very long training times for the Q-learning algorithm.

Removing this algorithm from our training analysis and focusing on the first 100,000 episodes, Figures 15 and 16 show that the algorithms that incorporate a memory (DQN with belief state and DRQN) can be trained more quickly than DQN without belief state.

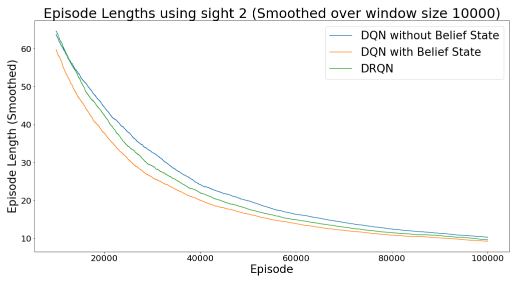


Figure 15: Comparison of episode lengths during first 100,000 training episodes using various algorithms.

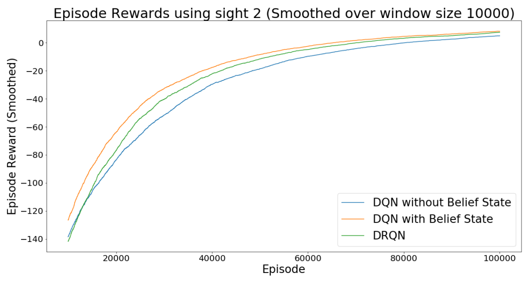


Figure 16: Comparison of episode rewards during first 100,000 training episodes using various algorithms.

The graph of episode rewards reveals some particularly interesting behaviour. Initially DRQN performs the worst, perhaps because it has the most parameters to train. But quickly the memory it acquires through the LSTM layer starts to yield higher rewards than DQN without belief state, eventually seeming to approach the level of performance achieved by DQN with belief state.

5.2.2 Policy Performance

However Figure 17 shows that the ultimate policy performance of DRQN doesn't quite match that of DQN with belief state.

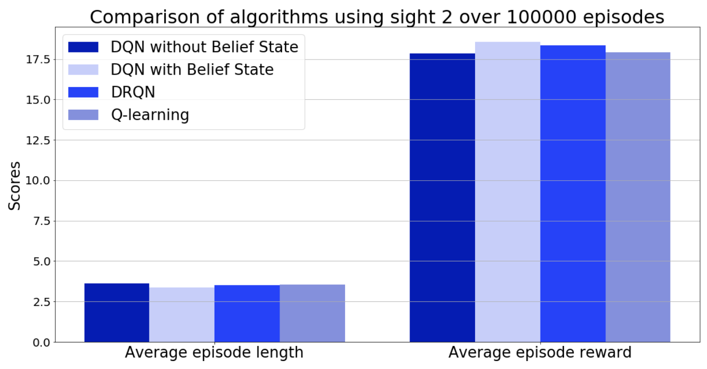


Figure 17: Comparison of ultimate policy performance using different RL algorithms and sight 2.

Although DQRN is equipped with a memory (and therefore outperforms DQN without belief state), the LSTM hidden elements do not provide perfect recall. On the other hand, the belief state that is provided is based on our knowledge of the exact dynamics of the mouse (i.e. that it moves uniformly at random). If the belief state was not able to perfectly model the environment's unseen elements it seems likely that the performance of DQN with belief state would drop below that of DRQN.

Note also that the performance of DQN without belief state and Q-learning are almost identical. This is to be expected as both converge to the optimal policy (given the observability of the environment).

Similar patterns are observed when the agent is given infinite sight range (note that this is not the same as full observability because the agent still can't see through walls). In this case DRQN is still better than the memory-free algorithms, but by a much smaller margin (see Figure 18)

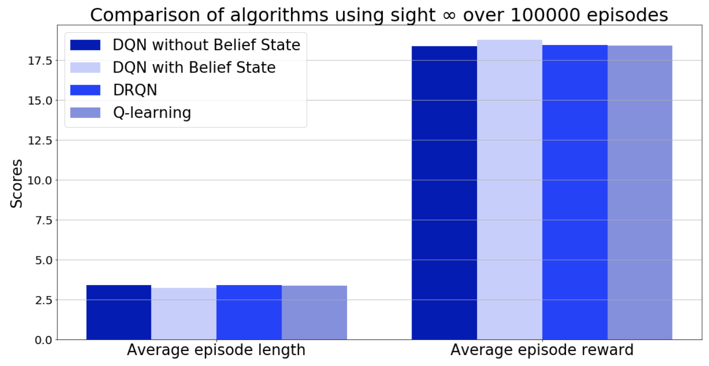


Figure 18: Comparison of ultimate policy performance using different RL algorithms and infinite sight.

We might expect the asymptotic performance of DRQN to be the same as DQN with belief state, so further investigation is required to understand the differences in performance.

6 Conclusions

As expected, when perfect information is not available the Deep RL algorithm performs better if a belief state is used (Figure 12). This highlights the need for memory to be integrated into the RL method to achieve maximum performance in partially observable environments.

Figures 17 and 18 confirm that in such environments, better performance is achieved by using an algorithm equipped with a memory. Of the two algorithms with memory that were investigated, using a DQN with belief state achieved faster learning and better ultimate performance than using a DRQN.

That said, when the belief state is hard to infer (and therefore may not perfectly model the environment), it seems likely that the DRQN approach, which would not suffer in such an environment, would achieve better asymptotic performance.

6.1 Future work

There are several avenues for further investigation in this area.

- It would be interesting to compare the Deep RL and Deep Recurrent RL algorithms using environments in which memory has a greater influence on performance. In the pursuer-evader task presented in this paper, it is relatively rare that the agent's memory actually gets invoked (especially once the cat has been trained), so the impact on performance is small. One way to do this would be to use transparent walls in such a way that the cat might see the mouse but have to move away from it to ultimately catch it.
- An investigation into the impact of getting the belief state 'wrong' would be valuable. It is not hard to imagine an agent being given a belief state that does not perfectly model the environment; the real world is messy and by definition models capture only a subset of the available information. As such, understanding the limitations of using a belief state by looking at how deficiencies in the model degrade performance would be wortwhile.
- Given the benefits (and limitations) of the belief state, it would be interesting to look at the effects of providing a DRQN with a belief state. Assuming the belief state is good but not perfect, the agent might start off relying heavily on the belief state but eventually realise that its own internal memory (i.e. the LSTM hidden/cell states) is more accurate, ultimately ignoring the belief state. This would give a good balance of efficient training and accurate performance that could potentially improve upon all current state-of-the-art algorithms for partially observable and hard-to-model environments.
- Similarly, it would be interesing to investigate the effects of using a belief state in fully observable but highly complex environments. There are many real-world contexts that are fully observable in theory but in practice the information is very dificult to capture. A belief state could be used as a shortcut to make these situations more manageable. By way of analogy, imagine catching a ball. It is theoretically possible to watch the ball right into your hands, but in reality we stop watching it just before it reaches us and predict where it will

be a moment later. An investigation into whether equipping robots with a belief state in fully observable environments where some of the information is hard to capture might yield fruitful results.

- Finally, it would be worth taking a closer look at the hyperparameters of the algorithms that have been tested. Comparisons of the algorithms have been done based on performance (both in training and in tests using the final policy). But performance is also dependent on the choice of hyperparameters (particularly in relation to the architecture of the neural networks). In order to obtain truly robust comparisons it would be worth optimising the hyperparameters to ensure we are extracting the best possible performance from each of the algorithms.

Acknowledgments

I would like to acknowledge MRC and EPSRC for providing the funding for this work, and thank University of Warwick Centre for Complexity Science for their support and Giovanni Montana & Malcolm Strens for their guidance throughout the project.

References

[1] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2011.

[2] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[4] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

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

[6] Wei Ji Ma. Generalized tic-tac-toe, Nov 2017.

[7] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[8] Gabriel Loye. Long short-term memory: From zero to hero with pytorch, Aug 2019.

[9] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[10] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*, 2015.

[11] Clément Romac and Vincent Béraud. Deep recurrent q-learning vs deep q-learning on a simple partially observable markov decision process with minecraft. *arXiv preprint arXiv:1903.04311*, 2019.

[12] Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, and Honglak Lee. Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128*, 2016.

[13] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

[14] Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines-revised. *arXiv preprint arXiv:1505.00521*, 2015.

[15] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[16] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[17] Ronen I Brafman. A heuristic variable grid solution method for pomdps. In *AAAI/IAAI*, pages 727–733. Citeseer, 1997.

[18] Blai Bonet. An e-optimal grid-based algorithm for partially observable markov decision processes. In *Proc. of the 19th Int. Conf. on Machine Learning (ICML-02)*, 2002.

[19] Joelle Pineau, Geoff Gordon, Sebastian Thrun, et al. Point-based value iteration: An anytime algorithm for pomdps. In *IJCAI*, volume 3, pages 1025–1032, 2003.

[20] David Silver and Joel Veness. Monte-carlo planning in large pomdps. In *Advances in neural information processing systems*, pages 2164–2172, 2010.

[21] F Chollet and JJ Allaire. Deep learning with r. manning publications, manning early access program. 2017.

Appendix A.

Q-learning hyperparameters

ϵ-greedy parameters:	
○ Start	$\epsilon_{start} = 1$
○ End	$\epsilon_{end} = 0.001$
○ Decay	Solution λ of $\epsilon_{start} \times \lambda^{no\_episodes} = \epsilon_{end}$
Discount factor	$\gamma = 0.99$
Learning rate	$\alpha = 5 \times 10^{-4}$
State encoding	Let b_h = board height and b_w = board width Each state is indexed as an integer in $\{0, 1, ..., (b_h^2 \times b_w^2 + b_h \times b_w) - 1\}$. The indices $\{0, ..., (b_h^2 \times b_w^2) - 1\}$ represent states in which the cat can see the mouse; the indices $\{(b_h^2 \times b_w^2), ..., (b_h^2 \times b_w^2 + b_h \times b_w) - 1\}$ represent states in which the cat cannot see the mouse.

DQN hyperparameters

ϵ-greedy parameters:	
○ Start	$\epsilon_{start} = 1$
○ End	$\epsilon_{end} = 0.001$
○ Decay	Solution λ of $\epsilon_{start} \times \lambda^{no\_episodes} = \epsilon_{end}$
Discount factor	$\gamma = 0.99$
Learning rate	$\alpha = 5 \times 10^{-4}$
Update rate for target network parameters	$\tau = 1 \times 10^{-3}$
Experience replay buffer size	1×10^5
Batch size	64
Network layers	3
Layer 1 type	Fully connected
Layer 1 input units	$2 \times$ board_height $\times$ board_width
Layer 1 output units	32
Layer 1 output activation	ReLU
Layer 2 type	Fully connected
Layer 2 input units	32
Layer 2 output units	16
Layer 2 output activation	ReLU
Layer 3 type	Fully connected
Layer 3 input units	16
Layer 3 output units	9
State encoding	Let b_h = board height and b_w = board width Board state encoded as a vector with dimension $2 \times b_h \times b_w$. The first $b_h \times b_w$ elements are a one-hot encoding of the cat's position. The last $b_h \times b_w$ elements encode the mouse position. If this is known, this is a one-hot encoding. If the cat can't see the mouse this is a probability distribution, incorporating the belief state if one is provided.

DRQN hyperparameters

ϵ-greedy parameters:	
○ Start	$\epsilon_{start} = 1$
○ End	$\epsilon_{end} = 0.001$
○ Decay	Solution λ of $\epsilon_{start} \times \lambda^{no\_episodes} = \epsilon_{end}$
Discount factor	$\gamma = 0.99$
Learning rate	$\alpha = 5 \times 10^{-4}$
Update rate for target network parameters	$\tau = 1 \times 10^{-3}$
Experience replay buffer size	1×10^5
Batch size	64
Network layers	4
Layer 1 type	LSTM
Layer 1 input units	$2 \times$ board_height $\times$ board_width
Layer 1 output units (also determines hidden/cell state dimensions)	32
Layer 2 type	Fully connected
Layer 2 input units	32
Layer 2 output units	32
Layer 2 output activation	ReLU
Layer 3 type	Fully connected
Layer 3 input units	32
Layer 3 output units	16
Layer 4 type	Fully connected
Layer 4 input units	16
Layer 4 output units	9
State encoding:	Let $b_h =$ board height and $b_w =$ board width Board state encoded as a vector with dimension $2 \times b_h \times b_w$. The first $b_h \times b_w$ elements are a one-hot encoding of the cat's position. The last $b_h \times b_w$ elements encode the mouse position. If this is known, this is a one-hot encoding. If the cat can't see the mouse this is a probability distribution, incorporating the belief state if one is provided.

Note that the LSTM layer is only used once for each input (in other contexts it is often used iteratively) and the hidden state is saved and fed back into the network alongside the next input.

Appendix B.

Github link for all code used in this project:

https://github.com/gpdwatkins/MSc_project