Learning what to memorize: Using intrinsic motivation to form useful memory in partially observable reinforcement learning

Alper Demir

Accepted: 5 November 2022 / Published online: 20 February 2023
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract
Reinforcement Learning faces an important challenge in partially observable environments with long-term dependencies. In order to learn in an ambiguous environment, an agent has to keep previous perceptions in a memory. Earlier memory-based approaches use a fixed method to determine what to keep in the memory, which limits them to certain problems. In this study, we follow the idea of giving the control of the memory to the agent by allowing it to take memory-changing actions. Thus, the agent becomes more adaptive to the dynamics of an environment. Further, we formalize an intrinsic motivation to support this learning mechanism, which guides the agent to memorize distinctive events and enable it to disambiguate its state in the environment. Our overall approach is tested and analyzed on several partial observable tasks with long-term dependencies. The experiments show a clear improvement in terms of learning performance compared to other memory based methods.

Keywords Memory · Intrinsic motivation · Partial observability · Reinforcement learning

1 Introduction
Reinforcement Learning (RL) has proved itself as a feasible machine learning method in problems that involve sequential decision making [32]. Many algorithms guarantee convergence to an optimal policy when the state of the agent is clearly apparent. However, such optimality guarantee vanishes under partial observability where the agent is no longer capable of receiving necessary information about the task to be learned. In such environments, the observation space may be smaller than the state space and may be ambiguous so that an observation may correspond to different states, each of which calls for different optimal actions. In order to overcome this issue, the agent may benefit from an additional memory, storing previous observations. This way, it can clarify its state to learn a policy solving the task.

Besides finding a good policy for a problem, the speed of learning is another focus of Reinforcement Learning. A relatively alternative approach to improve RL is intrinsic motivation (IM) where the reward function is supported by an additional intrinsic reward [29]. This reward is determined by a motivation that is persistent throughout the agent’s life and is driven by aspects such as curiosity, novelty, and surprise. Applying intrinsic motivation increases the agent’s adaptability to various problems and encourages exploration of these, which is more consistent with the idea of lifelong learning. Intrinsic motivation is coupled by many RL studies, proving its advantages [4]. However, this concept is yet to be introduced to memory management under partial observability.

Many algorithms addressing RL under partial observability devised a fixed method for managing the memory. Such methods include keeping a fixed or variable length window of information gathered previously in the explicit memory [16, 20], learning a finite state machine [22], predicting state representations [18] and learning through long short-term memory [23, 33]. However, because of their strict assumptions, these methods are limited to certain problems. Our argument is that the control of the memory can be given to the agent and the agent can learn what to store in it based on the task [27]. Such an idea allows the agent to be adaptable to different tasks, especially those that possess long-term dependency between an observation and the corresponding action. In these problems, the observation that affects the best action at a particular time step is made...
some unfixed time steps ago, therefore it is important to change the content of the explicit memory based on the dynamics of the problem. Rather than directly following a fixed method to determine the memory content, the agent can be intrinsically rewarded for keeping in the memory those observations which will be useful in the future. This approach can motivate the agent to form a memory with rare perceptions without ignoring the dynamics of the problem.

In this paper, we focus on memory-based model-free Reinforcement Learning under partial observability. Our contributions are as follows.

- By following a previous study [27], we extend the action space with memory-modifying actions so that the agent can take control of its own memory,
- We formulate the aim of forming useful memory as a drive by providing intrinsic rewards to keep distinctive events in memory,
- We propose an lightweight end-to-end approach that can adapt to different partial observable tasks with long-term dependency.

In comparison to similar methods, experiments on several benchmark problems demonstrate the improvement in learning performance provided by our proposed framework. We provide extensive analysis of our approach on various aspects.

The rest of the paper is organized as follows. Section 2 includes the background on RL with partial observability and intrinsic motivation. Section 3 discusses an example to motivate our approach. Section 4 defines our overall method and Section 5 includes many experiments and analyses.

2 Background and related work

In this section, we provide a brief summary of the topics forming the basis of our study. The section defines the environment model for partial observability, describes several reinforcement learning approaches to address the problem, and discusses a concept which can improve learning performance, namely intrinsic motivation, together with its implementations in RL.

2.1 POMDPs

A general model for realistic problems with limited information is given by the Partially Observable Markov Decision Process (POMDP). A POMDP is defined by a tuple \((S, A, T, R, \Omega, O)\) where \(S\) is a finite set of states, \(A\) is a finite set of actions, \(T : S \times A \times S \rightarrow [0, 1]\) is a transition function, \(R : S \times A \times S \rightarrow \Re\) is a reward function, \(\Omega\) is a finite set of observations, and \(O : S \times A \rightarrow \Pi(\Omega)\) is the observation function. In this study, we assume that the model contains discrete states and actions.

The POMDP model has been addressed in two ways in the literature. The first of these aims to keep a belief state [1], which is a probability distribution over the set of states, by assuming that the agent has access to the state space of the problem and the transition function, and can update this belief state with observations from the environment. The other is the hidden state POMDP approach [20], which assumes that the agent has no access to the problem structure and only takes observations from the environment. Under this setting, the agent needs to keep an estimated state on which it bases its actions. In this study, we follow the latter approach.

2.2 Reinforcement learning in POMDPs with hidden states

Many reinforcement learning methods have proven successful in fully observable environments but fail in partially observable ones with the loss of the guarantees of reaching the optimal policy [6]. The main reason for this is the perceptual aliasing problem that occurs in such environments [34]. Perceptual aliasing happens when more than one state corresponds to the same observation. In this case, the agent cannot distinguish between different states, which may have different best actions, and therefore cannot find the optimal policy. This situation makes the task of reinforcement learning in POMDPs with hidden states a difficult and unexplored field.

Methods aimed at RL in such environments are divided into two; memory-based and memoryless. Memoryless methods aim to converge to the best policy without additional storage of the information gathered from the environment. In such problems, finding the best memory-free policy is classified as NP-Hard [17]. On the other hand, the eligibility trace approach has been shown to be useful in practice. One of the methods using eligibility traces, Sarsa(\(\lambda\)), is an on-policy learning algorithm that updates all of the entries in the value function at a step based on their eligibility [19].

Memory-based approaches, on the other hand, depend on the addition of memory to overcome the problem of partial observability. In an environment with limited sensory information, the agent needs to predict the state that it is in, based on the observations received for the goal of finding the right course of action. To avoid perceptual aliasing, the agent must use a memory that can hold past observations, actions, or rewards. Using this memory, the agent forms an estimated state for its place in the state space.

The simplest method of generating state estimation using memory is to keep a fixed length window [16]. In this
method, the current state estimation is formed by adding the current observation to the memory, which stores a fixed number of past observations. The agent acts based on this state estimation to solve the given task. A more sophisticated approach is to increase the memory as needed and allow it to have different lengths. Examples of such variable length memory methods are Utile Suffix Memory, Nearest Sequence Memory, and U-Tree methods [20, 35]. For example; USM and U-Tree form a tree of observations and actions where each leaf node contains a set of values for each action. Each branch in this tree can be considered as an estimated state on which the agent forms its policy.

Unlike window based methods, Long Short-Term Memory approach, which uses deep neural networks, has proven itself in partial observable problems based on visual data [2, 14]. In this approach, there are forget gates in the recurrent neural network, in addition to input and output gates; thus this internal memory is not constrained by a time interval and can hold useful information in the long-term.

One study that involves holding memory retains it as a finite state machine, and generates it online by using stochastic gradient descent [22]. Such a finite state machine is capable of representing every state of an environment by storing all gathered information. Another study uses the agent’s past observations to generate a predictive state representations based on future observations and action predictions [11, 18, 30]. The study of [10] proposes an approach where an external reward machine is learned with the help of a labelling function in order to represent the reward function and the problem is decomposed into smaller solvable tasks.

The study inspired this paper has the aim of keeping an explicit memory as a binary string and letting the agent learn its management. In this way, the memory is not controlled by a fixed method but can be managed by the agent according to the needs of the problem [13, 27]. VAPS(1) algorithm is used to learn a policy by stochastic gradient descent, where the agent has additional actions to set the memory to any value at a particular time. VAPS(1) is shown to learn a good policy in POMDPs with long-term dependencies. Another study, inspired by [27], similar to our study, allows the agent to control a memory formed as an event buffer yet this approach lacks a supporting mechanism through intrinsic motivation [9].

Methods that use the window-based memory are based on the assumption that the required distinctive information is temporally close to the current moment, but this assumption is not always realistic. Deep Reinforcement Learning methods with LSTM units are independent of the time interval but they face the problem of the neural network being excessively customized for each environment [7] and they possess an implicit memory that is not open for analysis. Then, the predictive state representation approach requires a comprehensive exploration of the problem’s infrastructure to achieve good results and the reward machines require an additional labelling function to be formed during learning. Finally, in the approach where memory is held as a binary string, it is unrelated to the observations and the order in which they are gathered. Thus, there is a room for improvement for the task of learning memory management in RL, which is a relatively unexplored field.

### 2.3 Intrinsic motivation

The concept of Intrinsic motivation (IM), inspired by the field of psychology, has been applied in Reinforcement Learning. According to this concept, the agent receives intrinsic rewards from the satisfaction of the action itself, independent of any external goal. This motivation never disappears, but continues for as long as the internal reward is active. Intrinsic motivation can be triggered by factors such as curiosity, novelty, and surprise [29].

In the traditional reinforcement learning structure, the rewards are determined by the model and given outside of the learning mechanism. Recent studies show that, in addition to external reward, intrinsic rewards contribute to learning performance of the agent [3, 29]. The new reward function, which determines the agent’s rewards, is defined as \( R(s, a, s') = R(s, a, s') + R^{int}(s, a, s') \) where \( R^{int}(s, a, s') \) represents the intrinsic motivation that corresponds to reaching state \( s' \) by taking action \( a \) at the state \( s \).

The use of intrinsic motivation in reinforcement learning can be addressed under two main headings. The first of these is gaining knowledge. In this approach, solutions are sought for the problems of exploration and state representation in RL by giving intrinsic rewards to the actions bringing greater amounts of information from the problem. Studies show that the intrinsic rewards given through perception novelty enable the agent to explore and learn the problem better [4, 25]. To give an example, the count-based discovery approach [4], in addition to the reward \( R(s, a, s') \) in state \( s \), gives \( R^{int}(s, a, s') = \beta(\hat{N}(s) + 0.01)^{-2} \) as an intrinsic reward where \( \hat{N}(s) \) is the number of times that the state \( s \) is visited and \( \beta \) is a method specific parameter. With this approach, the additional intrinsic reward received in a state decreases as it is encountered, and thus this additional reward encourages the agent to explore relatively rare states.

Another use of intrinsic motivation is the problem of skill learning. The aim here is to learn different and generalizable skills by using intrinsic motivation or to infer which skills should be learned by following a curriculum. With this approach, predefined intrinsic rewards have been shown to improve learning performance in problems requiring
Learning what to memorize...

Although the concept of intrinsic motivation has been studied extensively in fully observable reinforcement learning problems, there are far fewer studies in partially observable problems. This situation arises from the difficulty of learning through the lack of necessary information and the effect of intrinsic motivation in partially observable environments remains unexplored.

3 Motivating example

The necessity of keeping memory has led to the use of the agent’s history during an episode of interaction with the environment, yet the problem of how to utilize it remains. The safest choice is to use the complete history in agent’s decision making to make the learning task Markovian [24]. However, this approach causes scalability issues due to the dramatic increase in the estimated state space $X$ on which the agent forms its policy. In reality, many parts of the agent’s history have no effect on the agent’s action selection. Even though a recent part of history is stored in memory, it may still contain redundant information.

As a motivating example, consider the Load/Unload problem [27], given in Fig. 1. In this problem, the agent is capable of moving with deterministic actions west and east, and loading at the green location and unloading at the red location. An episode starts with the agent unloaded at the red cell and ends when the agent returns to the red cell after getting loaded at the green cell. The problem is partially observable since the agent only gets observations based on the presence of a wall in four directions, but it does not know whether it is loaded or not. The agent therefore must keep a memory of being loaded, on the states in the corridor where each of them provides the same observation.

Figure 2 shows the states of the problem as circles in which the unloaded states are shown in dashed lines. The observation function $O$ groups these states into 3 observations given as bounding boxes; $o_u, o_l, o_c$ for the unloading and loading station, and the corridor observation.

Note that the states where the agent is in the corridor are all grouped into one observation, causing perceptual aliasing. While taking action $a_{east}$ maximizes the value of unloaded corridor states, taking $a_{west}$ maximizes the value of loaded corridor states. This situation makes it impossible to form an observation-based policy that maximizes the value of all underlying states.

Consider an episode as shown in Fig. 3. The agent starts at the red location, and moves to east for 3 time steps. At $t = 3$, the decision to take an east or west action depends on whether the agent is loaded ($o_l$) or unloaded ($o_u$), but is not affected by the corridor observations ($o_c$). Here, the agent needs to keep the temporarily distant observation ($o_u$) in its memory while ignoring the corridor observations ($o_c$). This way, it can take the best action $a_{east}$ while avoiding large spaces of estimated states.

The agent can keep a single observation in memory and form a state estimation by concatenating the memory with its current observation. In order to choose the best action in the corridor states, the agent only requires to know whether or not it is loaded, while the exact position in the corridor is unnecessary for the optimal decision. An agent that keeps
Fig. 3 An example episode for 3 time steps in the Load/Unload problem. For each time step, the states and the observations are given where the location of the agent is marked with A in the sketch. The observations are formed based on the presence of a wall in four navigational directions.

| t  | 0       | →  | 1       | →  | 2       | →  | 3       |
|----|---------|----|---------|----|---------|----|---------|
|    | $s_{0,u}$ | $a_{east}$ | $s_{1,u}$ | $a_{east}$ | $s_{2,u}$ | $a_{east}$ | $s_{3,u}$ |
|    | $o_u$   | $a_{east}$ | $o_c$   | $a_{east}$ | $o_c$   | $a_{east}$ | $o_c$   |

The observations of being loaded $o_l$ and unloaded $o_u$ in the memory may group the states, as shown in Fig. 4. An optimal trajectory of the agent includes being unloaded at the initial state $s_{0,u}$, going east as unloaded $(s_{1,u}, s_{2,u}, s_{3,u}$ or $(o_u, o_c))$, getting loaded on loading station $(s_{4,l}$ or $(o_u, o_l))$, going west as loaded $(s_{3,u}, s_{2,u}, s_{1,u}$ or $(o_l, o_c))$ and reaching to the unloading station $(s_{0,u}$ or $(o_l, o_u))$. Note that, for the state $s_{0,u}$ where the agent is in the unloading station, there are two estimated states, $(o_u)$ and $(o_l, o_u)$. Although this is a redundant representation, it does not affect the final policy.

With these estimated states, the agent ignores its exact position in the corridor by only keeping the information on being loaded or not. But it distinguishes the perceptually aliasing states just sufficiently to form the optimal policy shown in the figure. This reveals that a clever selection of

Fig. 4 Transition graph between estimated states in the Load/Unload problem where an estimated state $x$ is formed by concatenating the memory and the current observation. Only the observations $o_u$ and $o_l$ are kept in memory and $(o_i, o_k)$ represents a state estimation $x$ with observation $o_i$ in memory and observation $o_k$ as the current observation.
Learning what to memorize...

the information in memory may be adequate to perform optimally in a POMDP. Such an optimal performance may not require Markovian space, contradicting the good state representation conditions given in [5].

**Observation 1** A state estimation space $X$ that allows optimal policy may not be Markovian.

For Load/Unload problem, a deterministic policy, $\pi : X \times A \rightarrow [0, 1]$ can be given as:

$$
\pi(a^{east} | x_t = (o_u)) = 1.0 \\
\pi(a^{west} | x_t = (o_c, o_l)) = 1.0 \\
\pi(a^{east} | x_t = (o_u, o_c)) = 1.0 \\
\pi(a^{west} | x_t = (o_l, o_c)) = 1.0 \\
\pi(a^{east} | x_t = (o_c, o_u)) = 1.0 
$$

Such a policy allows the agent to complete the task in optimal performance without visiting the rest of the estimated state space, and maximizes the values of all states onto which its estimated states map. However, for some estimated states Markov property does not hold. That is:

$$P(x_{t+1} | x_t, a_t) \neq P(x_{t+1} | x_0, a_0, ..., x_{t−1}, a_{t−1}, x_t, a_t)$$

For $t = 3$ in the example episode;

$$P((o_u, o_l) | (o_u, o_c), a^{east}) \neq P((o_u, o_l) | (o_u), a^{east}, (o_u, o_c), a^{east}, (o_u, o_c), a^{east}, (o_u, o_c), a^{east})$$

Being in the estimated state $(o_u, o_c)$ means that the agent was in the unloading station, and now is in the corridor, but without pinpointing the exact location in the corridor. Taking action $a^{east}$ may lead to stay in the corridor or to the loading station. Knowing that the agent has a history of being in the corridor for 3 consecutive time steps, the probability of reaching the loading station after an $a^{east}$ action is 1.0, since it means that the agent is in the position immediately before the loading station. As having the history changes the probability of the next estimated state, the Markov property does not hold for this case, even though the learned policy performs optimally.

### 4 Self memory management with intrinsic motivation

Partial observability creates a significant challenge for Reinforcement Learning; having limited and aliased information about the environment, the agent is unable to devise an effective policy. Overcoming this issue requires keeping a memory of the history, besides the current observation, in order to distinguish the different states that map to the same observation. Then, the issue becomes deciding which events in history should be kept in memory to allow the agent to learn.

Our study follows the concept of keeping a memory under partial observability, and is inspired by the approach that gives memory control to the agent [27]. Unlike several studies which use a fixed method to control the memory, our approach, named Self Memory Management (SMM), gives responsibility to the agent and allows it to modify the memory by augmenting the action space with memory-changing actions. This learning process is supported by an intrinsic motivation to keep key information in memory. In an ambiguous environment, relatively rare events become distinctive and can help the agent to clarify its state. An intrinsic reward can be consistently provided based on the memory content, thus supporting memories with uncommon events.

The motivation behind incentivizing the agent on distinctive memories is that memory management is not a separate task from finding a good policy in an environment but an integral step. It may not correspond to simply collecting unique experiences in all types of environments. A distinctive memory allows the agent to navigate through partial observability and it is especially useful on the time when it proposes disambiguity. That is, the definition of a good memory is dynamic such that it depends on the current situation of the agent. Hence, the objective in forming one must be a driving force rather than a fixed process. This way, the agent is guided with an unsatisifiable urge to keep distinctive experiences in memory at all times while aiming to form a good policy. It is up to the agent to establish the balance between the rewards from the environment and the need to keep a useful memory.

Our approach possesses several advantages in these terms. First, it enables the agent to be more flexible according to the dynamics of the problem. Each environment may require a different memory content based on the task to be learned. In one problem, an agent may need to remember an earlier observation to make the best decision in future in one problem, whereas in another, it may need to remember only the previous one for a sense of direction. With SMM, the agent can learn the best memory content, which allows it to solve the task, while interacting with the environment. Second, the idea of keeping rare events in memory is not forced but supported. Thus, the task requirements are not ignored, but added to the agent’s decision making. The agent can learn to store a rare event in memory when useful for the task. Finally, our approach does not require extensive computation, so it is a lightweight addition to a Reinforcement Learning agent.
4.1 Defining memory and actions over memory

In this section, we define the form of the memory, which is different from that in the study of [27], and propose the set of memory-modifying actions.

4.1.1 Explicit memory

Our definition of a memory \( m_t \) at time \( t \) is a sequence of events that is previously perceived. In this study, we focus on two types of events. An event \( e_i \) may contain (1) only the observation \( o_t \) or (2) observation - environment action pair \((o_t, a_t)\) at time \( t \). A memory is notated as;

\[
m_t = \langle e_i, e_j, ..., e_k \rangle
\]

where \( e_i, e_j, ..., e_k \) represent the events at times \( i, j, k \) respectively and \( i < j < k < t \). That is, the events are ordered from left to right according to the time that they are gathered. Note that \( i, j, k \) and \( t \) are not necessarily consecutive numbers. An empty memory \( m = \emptyset \) has the size of zero, that is \( |m| = 0 \).

4.1.2 Actions over memory

Our method composes the action space with a set of actions modifying the memory.

The set of memory actions \( \mathcal{A} \), that we use in this study, consists of two actions; \( \mathcal{A} = \{a_{push}, a_{skip}\} \) and the effects of these actions are defined by a function \( \mathcal{T} : M \times E \times \mathcal{A} \to M \) where \( M \) defines the set of all possible memories and \( E = \Omega \) and \( E = \Omega \times A \) for \( e = o \) and \( e = oa \), respectively.

At time \( t \), for a memory \( m_t = \langle e_i, e_j, ..., e_k \rangle \) and the current event \( e_t \), a memory action \( a_t \) determines the next memory \( m_{t+1} \) according to

\[
m_{t+1} = \mathcal{T}(m_t, e_t, a_t)
\]

\[
= \begin{cases} 
\langle e_i, e_j, ..., e_k \rangle, & \text{if } a_t = a_{skip} \\
\langle e_j, e_k, ..., e_k \rangle, & \text{if } a_t = a_{push} \text{ and } |m_t| < c, \\
\langle e_j, ..., e_k \rangle, & \text{if } a_t = a_{push} \text{ and } |m_t| = c,
\end{cases}
\]

where \( c \geq 0 \) is the capacity of the memory, a parameter to bound it. With this definition, taking the action \( a_{push} \) pushes \( e_t \) to the end, and if the memory is at its full capacity, pops the oldest event \( e_i \) from it. On the other hand, the action \( a_{skip} \) skips \( e_t \) without changing the memory.

The advantages of defining the memory in this way is two-fold. First, keeping the events in a sequence enables the memory to keep them ordered, giving the agent a sense of direction. By this definition, a memory \( \langle e', e'' \rangle \) is a distinct memory than \( \langle e'', e' \rangle \), hence it represents a different experience of the problem. Second, \( \mathcal{T} \) restricts the transitions between memories. The agent cannot jump between completely distinct memories, therefore the set of visited memories is kept bounded according to the events experienced in the environment.

4.2 Adding intrinsic motivation

The memory management method proposed in this study is based on learning what should be kept in the memory. This learning process can be supported by giving intrinsic motivation to changes in memory. Note that as the size of the memory increases, the space that the agent forms its policy upon also enlarges. With good guidance, the agent may learn a good policy without exploring redundant regions of this space.

Our method attacks the problems where there is long-term dependency between the information and the corresponding action. That is, the required event that allows taking the best action at a moment is temporally distant. These problems require the agent to keep the event in memory for the time that it is going to be important.

We argue such events are rare in an ambiguous environment. Therefore, an agent motivated to keep rare events in memory can improve its learning performance. In this study, we propose two intrinsic reward functions. The first one defines the intrinsic reward to keep a memory \( m_t \) at time \( t \) as;

\[
R_{1}^{int}(m_t) = \beta \cdot \left( \frac{\sum_{e \in m_t} (1 - f_t(e))}{c} - c \right)
\]

where \( c \geq 0 \) is the memory capacity (the number of events that can be kept in the memory), \( \beta \in [0, 1] \) is a parameter to control the intrinsic reward and \( f_t(e) \in [0, 1] \) is the relative frequency of the event \( e \) at time \( t \). This motivation is inversely proportional to the probability of getting an event, similar to the distributional model in [26]. The frequency \( f_t(e) \) can be calculated as \( f_t(e) = n_t(e)/\sum_{e' \in E} n_t(e') \) where \( n_t(e) \) represents the number of times the event \( e \) is get and \( E \) is the set of experienced events at time \( t \).

Note that \( R_{1}^{int}(m) \) is bounded to \([-c, 0]\) but the magnitude of the reward decreases with the addition of infrequent events to the memory. Such a function motivates the agent to utilize the memory to its full capacity by retaining rare events. Also, by being a negative value, it does not diverge the agent away from a goal state, leaving its main aim unchanged.

The second IM function defines the intrinsic reward for keeping a memory \( m_t \) at time \( t \) as;

\[
R_{2}^{int}(m_t) = \beta \cdot \left( \frac{\sum_{e \in m_t} (1 - f_t(e)) h_t(e)^{\frac{h_t(e)}{|m_t|}}}{c} - 1 \right)
\]

where \( h_t(e) \) is the number of times that the event \( e \) is stored in memory at time \( t \). Note that this count is reset at the end of each episode.
$R_{2}^{\text{int}}(m)$ is bounded by $[-1, 0]$ and decays the potential of keeping an event in memory, to ensure that the agent is motivated to keep recent events when it is beneficial to do so. However, this decay is directly linked to how rare the event is in general. The potential of keeping a rare event decays more slowly compared to a common one. While $R_{2}^{\text{int}}(m)$ is independent of the memory capacity $c$, the term $-1$ acts as an incentive to use the memory. Using an average term encourages the agent to extend its memory when, on average, it improves the memory’s potential.

### 4.3 Overall method

The agent, having a memory, needs to form a state estimation by using it in order to make better decisions.

At a time $t$, the agent’s state estimation $x_t$ is formed by concatenating the memory $m_t = \langle e_i, e_j, ..., e_k \rangle$ with the current observation $o_t$ as below;

\[
x_t = m_t \parallel o_t = \langle e_i, e_j, ..., e_k, o_t \rangle
\]

where $e_i$, $e_j$ and $e_k$ are previously seen events at times $i$, $j$ and $k$ in the memory. The set of estimated states, formed by this approach, is noted as $X$.

A composed set of actions $\hat{A} = A \times A$ is formed by using the environment actions $A$ and the memory actions $A$. Using these definitions, the agent aims to learn a policy $\pi : X \times \hat{A} \rightarrow [0, 1]$.

At a time $t$, with the estimated state $x_t$ formed by the memory $m_t$ and the current observation $o_t$, the agent takes a composed action $(a_t, a_t)$ according to its current policy $\pi_t$, where $a_t$ is the action to pass to the environment and $a_t$ is the action over the memory. Taking action $a_t$ in the environment results in the next observation $o_{t+1}$ and the agent’s next memory $m_{t+1}$ is determined according to $\hat{T}(m_t, e_t, a_t)$

The next estimated state $x_{t+1}$ is formed by concatenating $m_{t+1}$ with $o_{t+1}$. The agent gets a reward $\hat{r}_t$ according to the new reward function;

\[
\hat{r}_t = \hat{R}(x_t, (a_t, a_t), x_{t+1}) = R(x_t, (a_t, a_t), x_{t+1}) + R_{\text{int}}(x_t, (a_t, a_t), x_{t+1})
\]

where $R(x_t, (a_t, a_t), x_{t+1})$ reduces to the original reward function $R(s_t, a_t, s_{t+1})$ and $R_{\text{int}}(x_t, (a_t, a_t), x_{t+1}) = R_{\text{int}}(m_{t+1})$ as the intrinsic motivation to keep the memory $m_{t+1}$.

These modifications alter the setting on which the learning mechanism operates, while the interaction between the agent and the environment is unchanged. The agent provides an action from $A$ and obtains observations and rewards from $\mathcal{O}$ and $R$. The proposed method is independent of the reinforcement learning algorithm applied. Different learning methods can be used to learn the agent’s policy.

We implemented our approach on top of Sarsa($\lambda$) algorithm [19], which shows faster convergence under partial observability. In this setting, we have a Q value for each $(x, (a, a))$ pair and at a time $t$, the agent experiences $(x_t, (a_t, a_t), \hat{r}_t, x_{t+1})$ and follows the update rules;

\[
\eta_t(x_t, (a_t, a_t)) = 1
\]

\[
\eta_t(x, (a, a)) = \gamma \cdot \lambda \cdot \eta_{t-1}(x, (a, a))
\]

\[
\eta_t(x, (a, a)) = \gamma \cdot \lambda \cdot \eta_{t-1}(x, (a, a)) + \alpha \cdot \eta_t(x, (a, a)) \cdot \delta_t
\]

where $\eta_t(x, (a, a))$ is the eligibility trace of the estimated state - composed action pair $(x, (a, a))$, $\alpha$ is the learning rate, $\lambda$ is the trace decay constant, $\gamma$ is the discount factor and

\[
\delta_t = \hat{r}_t + \gamma \cdot Q_t(x_{t+1}, (a_{t+1}, a_{t+1})) - Q_t(x_t, (a_t, a_t))
\]

is the TD-error at time step $t$.

### 5 Experiments

We experimented on four partially observable environments with potential to demonstrate the effectiveness of our approach. Each of these have a long-term dependency, requiring the agent to keep a temporally distant event in memory to where it is useful. We selected Sarsa($\lambda$) as the underlying learning algorithm for our approach. Sarsa($\lambda$) is coupled with (1) only self memory management (abbreviated as Sarsa($\lambda$) w/ SMM) and (2) self memory management supported by intrinsic motivation (abbreviated as Sarsa($\lambda$) w/ SMM + IM). These are compared with methods using explicit memory; Sarsa($\lambda$) with a memory of fixed window (abbreviated as Sarsa($\lambda$) w/ FW), and the incremental implementation of VAPS(1) algorithm [27] since this is the original study proposing the concept of agent controlled memory.

VAPS(1) algorithm is implemented differently in terms of the action space: In the original study, the authors add a new action for setting each bit of the memory, but taking this action wastes a time step, making no changes in the environment. In order to have a fair comparison, we implemented their second approach, which allows the agent to take composite actions that can directly set the entire memory to a certain value while taking an action in the environment.

In addition to methods with explicit memory, the experiments consist of two baselines that use LSTM units to summarize the history of the agent; A2C [23] and PPO [28].

The experiments are organized in 5 sections. Section 5.3 tests how the type of the events in memory affects the learning...
performance of SMM without the support of IM. Section 5.4 examines the effect of different intrinsic motivation functions on the learning performance by coupling SMM with IM. Section 5.5 further analyzes the overall method with respect to event replacement schemes, changes in the time that IM is introduced to learning, and the parameters of the algorithm. Finally, Section 5.6 compares our method’s the best performance with the related work of others.

5.1 Environments

This section describes the environments that are included in the experiments.

5.1.1 Load/Unload

Load/Unload (Fig. 1, Section 3), is the environment [27] which VAPS(1) is experimented on, with a 1 bit of memory in the original study. The agent has to keep a memory of being loaded or unloaded, on the states in the corridor where each of them provides the same observation. Overall, the domain has $|S| = 8$, $|A| = 2$, $|\Omega| = 3$.

5.1.2 Meuleau’s maze

Meuleau’s Maze is a labyrinth like domain [21], shown in Fig. 5. In this problem, the agent has four navigational actions as north, east, south, west, and it starts from the state with $S$ and aims to reach the goal state with $G$. The actions are stochastic in this version, where the agent is capable of moving to the intended direction with 0.8 probability, and can end up in a random direction with 0.2 probability. The observations are partial in terms of the presence of a wall in four directions, and the agent is rewarded for reaching $G$ with 5.0 where other actions get -0.01. In this problem, the states in the corridors provide aliased observations, and may require opposite actions (as in the vertical corridors) and the agent has to update its memory whenever it reaches a corner or a T-junction, rather than keeping a fixed memory content. The environment has $|S| = 147$, $|A| = 4$, $|\Omega| = 9$.

5.1.3 Tree maze

Tree Maze, shown in Fig. 6, is a partial observable problem that contains long-term dependency [31]. The agent starts from the location $S$ and needs to reach a state at the end of a leaf. It has three deterministic actions to go north, east, south. The target goal state is determined at the beginning of each episode, making the problem more complex. An episode ends with a reward of 10 when the agent reaches the goal state (the end of the correct leaf), and -1 for reaching other leaf states. All regular steps of the agent are rewarded by -0.1. An observation is limited to the location in the corridor (classified as left-most, middle or right-most), the number of turns taken and an additional bit to represent the required turns to take in each T-junction, which are only available in the first two steps of the episode. For example, to reach the leaf second from the top, the corresponding bit of the first two observations are given for going north and south in this order. In this environment, the agent has to keep these initial observations in memory until they are useful, in order to reach the desired goal state. As a result, Tree Maze has $|S| = 140$, $|A| = 3$, $|\Omega| = 13$.

5.1.4 Basic scheduler

As a different type of environment, we propose Basic Scheduler (Fig. 7), which consists of $q$ number of jobs

![Basic Scheduler](image)

---

Fig. 5 Sketch of Meuleau’s Maze where the start and the goal states are marked with $S$ and $G$, respectively. The observations are limited to the presence of a wall in four navigational directions.

Fig. 6 Sketch of Tree Maze where the agent starts from the state with $S$ and aims to reach a leaf which is determined at the beginning of each episode. The agent observes the turns it needs to take, only at the first and the second time steps of the episode.
Learning what to memorize...

of the parameters are set as $\lambda$ [10]. Both methods used a network of 5 fully connected
hidden layers with 64 neurons per layer. During learning, the methods used 128 sampled experiences from a buffer of size 100,000 with the learning rate of 0.00001. We selected 256 as the number of hidden neurons for the LSTM, where the rest of the parameters are left as default.

Because of the stochastic process in each algorithm, we present the average results over 50 experiments where each episode either ended when the agent reached the goal state or to the step limit of 500.

5.3 The influence of event content

In this section, we focus solely on SMM, namely, on the effect of event type on the learning performance, the number of changes in memory and the size of the space on which the agent forms its policy. We experimented with two types of memory content; the agent forms its memory by pushing (1) events of only observations ($e_t = o_t$) and (2) events of observation - environmental action pairs ($e_t = o_t a_t$) as defined in Section 4.1. In the experiments of this section, the learning mechanism is not supported by intrinsic motivation, i.e. the experiments show the results of Sarsa($\lambda$) w/ SMM. We let the agent to have memory capacity $c$ as 1, 1, 2 and 1 in Load/Unload, Meuleau’s Maze, Tree Maze and Basic Scheduler, respectively. As an example, a memory with $c = 2$, $e_t = o_t a_t$ can contain at most 2 pairs of observation - environment action pairs.

As Table 1 shows, including actions in memory results in a larger set of possible estimated states, as expected. When the agent also memorizes the action taken on an observation ($e_t = o_t a_t$), its experiences become more distinct. This way, waiting to run, and an agent learning how to operate them. At the beginning of each episode, each job’s remaining execution time is initialized randomly between 1 and $p$ time steps where executing a job decreases this value until it reaches 0. At a time step, the agent may assign the CPU to any job, or may not assign at all, leaving the CPU idle. Thus, the agent has $q + 1$ number of actions. The agent is rewarded by 10 for completing a job (remaining execution time reaching to 0), -1 for assigning the CPU to an already completed job and -0.5 for leaving the CPU idle. An episode ends when all the jobs are completed or the CPU is reassigned to a finished job. The agent observes the states of the jobs as active or completed only when the CPU becomes idle. During the execution of a job, the agent gets a null observation. Due to this limited observability, the agent has to memorize the observation received when a job is complete to avoid assigning the CPU to an already completed task.

In our experiments, we choose $q = 8$ and $p = 25$. In overall, the environment has $|S| = 25^8$, $|A| = 9$, $|O| = 257$.

5.2 Setup

The results are presented in terms of three aspects; the learning performance, the number of times the explicit memory is changed, that is, $m_t \neq m_{t+1}$, and the empirical size of the estimated state space on which the agent learns. These selected metrics demonstrate that our approach improves the learning speed by learning how to manage the memory with minimal changes.

All Sarsa($\lambda$) versions are tested with $\epsilon$-greedy action selection with $\epsilon$ decaying from 0.2 to 0.001 and the rest of the parameters are set as $\lambda = 0.9$, $\alpha = 0.01$, $\gamma = 0.9$ whereas Sarsa($\lambda$) w/ SMM + IM used $\beta = 1.0$ unless stated otherwise. We selected various values for the memory capacity $c$ that can enable learning on different environments. VAPS(1) algorithm follows Boltzmann law to select actions, having a stochastic policy. The Boltzmann temperature value is decayed over episodes, as the original study suggests.

For A2C and PPO, we used Stable Baselines implementations [8]. We followed a setting similar to the study of [10]. Both methods used a network of 5 fully connected

---

**Table 1** The effect of the event type ($e_t = o_t$ or $e_t = o_t a_t$), on the size of the estimated state space that the agent explores with SMM where the values represents the average number of unique visited estimated states at the end of an experiment. SMM has memory capacity $c$ as 1, 1, 2 and 1 in Load/Unload, Meuleau’s Maze, Tree Maze and Basic Scheduler, respectively.

| Problem               | $e_t = o_t$ | $e_t = o_t a_t$ |
|-----------------------|-------------|-----------------|
| Load/Unload           | 9.52        | 15.72           |
| Meuleau’s Maze        | 72.12       | 239.10          |
| Tree Maze             | 151.72      | 593.06          |
| Basic Scheduler       | 1472.92     | 6855.32         |
the space of unique estimated states expands, and the agent is required to spend more time exploring in this space.

Figure 8 demonstrates this effect of a larger space in terms of learning performance and the number of memory updates. Storing actions in memory decreases the number of time steps needed for learning in Load-Unload environment, but either delays learning or has no dominant advantage in the other domains. This suggests that actions

**Fig. 8** Learning performance and number of changes on explicit memory during an episode in (a) Load/Unload (b) Meuleau’s Maze (c) Tree Maze (d) Basic Scheduler for SMM agent with event types $e_t = o_t$ and $e_t = o_t a_t$. The lines represent the average of 50 experiments and shaded areas are the 95% bootstrapped confidence intervals. Note that, in Load/Unload, the number of steps to reach the goal state is given as the domain gives reward only at the goal state.
help the agent disambiguate the environment further in Load-Unload but it is unnecessary to have them in memory in other ones.

In Meuleau’s Maze, Tree Maze and Basic Scheduler domains, the agent with \( e_t = o_t a_t \) has to explore a larger space to converge on a good policy and update its memory more frequently, yet the same performance can be achieved with \( e_t = o_t \) with more stable exploration. As a result, there is a trade-off on the content of the memory. A more detailed memory diminishes perceptual aliasing but also enlarges the space of estimated states on which the agent operates. Therefore, the actions of the agent may be missed in the memory unless they are absolutely necessary for convergence on a good policy, i.e., they contain distinctive information that the observations lack.

5.4 The influence of intrinsic motivation

This section includes experiments on SMM supported by IM. We tested three types of intrinsic motivation functions; \( R_{1}^{int} \) and \( R_{2}^{int} \) as proposed in Section 4.2 and \( R_{3}^{int} \) as a function that supports count-based exploration on estimated states [4]. It is used as

\[
R_{3}^{int}(x, (a, a)) = \frac{1}{\sqrt{n_1(x, (a, a))}}
\]

where \( n_1(x, (a, a)) \) is the number of times that estimated state - composed action pair \( (x, (a, a)) \) is observed at the time \( t \).

For the underlying memory content, we selected the event types that allow faster convergence with IM for each environment. SMM used \( e_t = o_t a_t \) in Load/Unload where it used \( e_t = o_t \) in the other domains.

Figures 9 and 10 show how intrinsic motivation modifies the learning performance of the agent. In Fig. 10, the effect of count-based exploration \( R_{3}^{int} \) is somewhat similar to having no intrinsic motivation in all environments. However, intrinsic motivation functions \( R_{1}^{int} \) and \( R_{2}^{int} \) lead to improvements. With these functions, the agent is rewarded when it keeps rare events in its memory. This allows it to keep more distinct experiences and solve the problems more quickly. Especially, SMM with \( R_{2}^{int} \) shows faster convergence in all environments, which makes it the most efficient motivation function to support memorizing rare events, among the others.

The effect of concentrating on rare events with \( R_{2}^{int} \) can be seen in Fig. 9. It shows in which locations the agent pushed the event in memory in the last episodes of Meuleau’s Maze. Comparing Fig. 9b to a, it is clear that the distribution of memorization is more focused on rare events for the agent supported by IM. With \( R_{2}^{int} \), the agent memorizes the observations such as corners and T-junctions but ignores the observations in the corridors because they are less common than the ones in the corridors. Memorizing rare events disambiguates well, hence the agent is converging faster to a solution. Figure 9 also shows that the agent dynamically updates its memory rather than keeping the most infrequent observations constantly in it. This supports the argument that adaptive memory management is a good fit for partially observable environments.

5.5 Further analysis of SMM + IM

In this section, we further analyzed the coupling of SMM and IM and employed a parameter analysis on the overall method.

First, we examined different event replacement schemes when the memory is full. Our initial definition of the memory transition function \( T \) follows a first in first out (FIFO) mechanism, i.e., the oldest event is removed when the memory is full and a new event is to be pushed. Although pushing the event to only one end of the memory is necessary to keep it time ordered, alternative replacement approaches are available.

We define three more alternatives to pushing a new event action, each one differing on the case when the memory is full. In total, we have four pushing actions defined as
Fig. 10 Learning performance and number of changes on explicit memory during an episode in (a) Load/Unload (b) Meuleau’s Maze (c) Tree Maze (d) Basic Scheduler for SMM agent with IM types $R_1^{int}$, $R_2^{int}$, $R_3^{int}$ and no IM. The lines represent the average of 50 experiments and shaded areas are the 95% bootstrapped confidence intervals. Note that, in Load/Unload, the number of steps to reach the goal state is given as the domain gives reward only at the goal state.
Learning what to memorize...

follows. At time $t$, for a memory $m_t = \langle e_1, e_2, ..., e_k \rangle$ with $|m_t| = c$ and the current event $e_t$,

- $a_{\text{pushFIFO}}$ discards $e_i \in m_t$ and pushes $e_t$ to the end of the memory,
- $a_{\text{pushMFO}}$ discards event $e_x \in m_t$ such that $e_x = \arg \max_{e \in m_t} (f_t(e))$ and pushes $e_t$ to the end of the memory,
- $a_{\text{pushMFK}}$ discards event $e_x \in m_t$ such that $e_x = \arg \max_{e \in m_t} (h_t(e))$ and pushes $e_t$ to the end of the memory,
- $a_{\text{pushLI}}$ discards event $e_x \in m_t$ such that $e_x = \arg \min_{e \in m_t} (1 - f_t(e)h_t(e))$ and pushes $e_t$ to the end of the memory.

That is, while adding $e_t$ to the rightmost end, the pushing actions $a_{\text{pushFIFO}}, a_{\text{pushMFO}}, a_{\text{pushMFK}}$ and $a_{\text{pushLI}}$, remove the oldest, the most frequently observed, the most frequently kept in memory during the episode and the least important event from the memory, respectively.

To test these alternative definitions, we experimented on Tree Maze since the other environments require a memory with one capacity and the definitions are indifferent on such a case.

Figure 11 demonstrates the effect of different event replacement methods on the case of full memory. Although all methods allow the agent to converge on the best policy, at the beginning of learning, there is a clear advantage for the methods that discards the least important and the most frequently observed events from the memory. The reason here is that such schemes allow the agent to keep the most distinctive events in memory and are more aligned with the agent’s intrinsic motivation. When compared in terms of number of memory changes, $a_{\text{pushLI}}$ action results in more stationary memory management. Thus, we selected LI as the event replacement mechanism on a full memory for the remaining experiments.

Second, we demonstrate how a delay in IM during the learning process affects the learning performance, even though our overall approach is devised with constant IM.

We experimented with a case where the agent still keeps count of each event, but does not include an intrinsic reward in its learning updates until a certain episode threshold is met. We used two thresholds to start IM, namely at 20% and 60% of the learning process, comparing these cases to SMM with constant IM. The method is formed with the best memory capacity, the event type and the intrinsic motivation function, based on the results of the previous experiments. To avoid overcrowding, we limited the analysis on Tree Maze in this section.

Figure 12 shows that motivating the agent to keep rare events in memory improves the learning performance, even if it starts later in the learning in Tree Maze domain. The Q-table is introduced with a new reward function when the intrinsic reward is added to the environmental reward, and therefore the learning performance declines for a short time. However, the intrinsic motivation quickly supports the agent.
Fig. 13  Total reward and number of memory changes for Sarsa(\(\lambda\)) with SMM (\(e_i = o_i\)) and IM (\(R^{int}_2\)) on alternative (a) \(\lambda\) values, (b) \(\beta\) values and (c) \(c\) values in Tree Maze problem. The lines represent the average of 50 experiments and the confidence intervals are omitted for better view.

In forming a better state estimation, and leads to the goal state, as demonstrated by the sudden increase in the learning performance after the motivation begins.

We further tested Sarsa(\(\lambda\)) with SMM (\(e_i = o_i\)) + IM (\(R^{int}_2\)) on several values of its parameters in Tree Maze domain while keeping the other parameters unchanged. To avoid overcrowding in the figures, we kept this set of values small. Figure 13a shows the effect of various \(\lambda\) values on the learning performance. As expected, lower \(\lambda\) translates to a slower convergence to a good policy since higher \(\lambda\) allows Sarsa(\(\lambda\)) algorithm to propagate the TD-error to a larger set of eligible estimated state - composed action pairs.

In Fig. 13b, the effect of \(\beta\), therefore the amplitude of intrinsic rewards on learning can be seen. While learning with \(\beta = 0\) is slow, even a small value as \(\beta = 0.2\) improves the learning speed of SMM + IM. Here, the intrinsic reward is smaller, yet it can modify the policy so that the agent

Table 2  The effect of the memory capacity, \(c\), on the theoretical size of the estimated state space \(|X|\) and the SMM+IM coverage of this space in Tree Maze with \(e_i = o_i\) and \(R^{int}_2\) where \(|\hat{X}|\) represents the average number of unique visited estimated states at the end of an experiment.

| \(c\) | \(|X| = \sum_{i=1}^{c+1} |\Omega|^i\) | \(|\hat{X}|\) | Coverage |
|------|------------------|------------|--------|
| 2    | 2379             | 308.08     | 12.9%  |
| 3    | 30940            | 992.12     | 3.2%   |
| 4    | 402223           | 2710.28    | 0.6%   |
| 5    | 5229042          | 6306.58    | 0.1%   |
Learning what to memorize...

Fig. 14 (a) Number of steps to complete an episode, (b) number of changes on explicit memory during an episode in Load/Unload problem. VAPS(1) algorithm has a 1 bit memory where both Sarsa(\(\lambda\)) with FW and Sarsa(\(\lambda\)) with SMM + IM has a memory with a capacity of \(c = 1\) observation - environment action pair \((e_t = o_t a_t)\). The lines represent the average of 50 experiments and shaded areas are the 95% bootstrapped confidence intervals.

Table 2 demonstrates how the memory capacity \(c\) affects the space of estimated states. As the memory expands, the set of possible estimated states dramatically enlarges. However, with the support of the IM, the size of the visited estimated states \(|\hat{X}|\) grows much more slowly. In fact, at the end of the experiments, the agent’s coverage of this space keeps getting smaller compared to the theoretical size of the entire space. This suggests that IM provides guidance to the agent in the large space of estimated states and enables a safe convergence.

The effect of \(c\) on learning performance of SMM can be seen in Fig. 13c. For Tree Maze, having \(c = 2\) is sufficient to learn as there are two directions to remember on two T-junctions. Increasing \(c\) causes the agent to increase the frequency of memory updates. However, the effect on the learning performance is insignificant, since \(R_{int}^2\) does not depend on the memory capacity \(c\). The agent extends its memory when beneficial. Even with a redundant larger memory, the useful events in the memory allow the agent to form its policy to solve the task.

5.6 Comparison to other related works

In this section, we compared our overall method (Sarsa(\(\lambda\)) w/ SMM + IM) with its best setting to corresponding approaches of others. Based on the results of Section 5.4, we used \(R_{int}^2\) as the intrinsic motivation function for Sarsa(\(\lambda\)) w/ SMM + IM in this section.

In Load/Unload, we let VAPS(1) algorithm to have 1 bit memory where both Sarsa(\(\lambda\)) with FW and Sarsa(\(\lambda\)) with SMM + IM has a memory with a capacity of \(c = 1\) observation - environment action pair \((e_t = o_t a_t)\).

Fig. 15 (a) Total reward collected throughout an episode, (b) number of changes on explicit memory during an episode in Meuleau’s Maze problem. VAPS(1) algorithm has a 3 bit memory where Sarsa(\(\lambda\)) with FW and Sarsa(\(\lambda\)) with SMM + IM has a memory with a capacity of \(c = 3\) and \(c = 1\) observations \((e_t = o_t)\), respectively. The lines represent the average of 50 experiments and shaded areas are the 95% bootstrapped confidence intervals.
In order to demonstrate the speed of learning, Fig. 14a shows the number of steps to complete an episode in Load/Unload problem, rather than total collected reward, since the environment provides a reward only when the agent reaches the goal state. It shows that a fixed window memory including actions allows for a rapid convergence on the optimal policy. Also Sarsa(\(\lambda\)) with SMM + IM, starts strong on reaching to the goal state in fewer steps. The observations at the loading and unloading stations are relatively rare, thus, the agent with SMM + IM is motivated to keep them in memory, leading to a better learning performance. VAPS(1) agent needs to learn when to set or clear its memory, while SMM, supported by intrinsic motivation, is able to learn much faster what to store in the memory. Both of the LSTM-based methods show convergence, but they are less sample-efficient compared to SMM + IM. It can be seen in Fig. 14b, this learning performance of SMM + IM is achieved with far fewer changes over the memory compared with the other explicit memory based methods. In fact, SMM + IM updates the memory only when the agent reaches the end of the corridor, and leaves it unchanged in the corridor itself.

For Meuleau’s Maze, we experimented with different memory capacities for VAPS(1) and Sarsa(\(\lambda\)) with FW, yet the methods were unable to reach the goal state. To compare, we let VAPS(1) algorithm and Sarsa(\(\lambda\)) with FW have 3 bit and \(c = 3\) observations \((e_t = o_t)\) memory, respectively. Figure 15a shows that the labyrinth-like structure of the problem prevents the other methods reaching the goal state. Due to challenging perceptual aliasing, other methods become trapped in the maze throughout the learning process. In fact, SMM + IM is the only approach that shows learning and converges to a near optimal performance. The agent with SMM + IM is motivated to remember observations such as corners and T-junctions because of their rarity, and keeps them in memory to guide itself through the maze. This way, corridors requiring opposite actions can be disambiguated and the perceptual aliasing is overcome. This performance is achieved with a minimum number of memory changes as seen in Fig. 15b.

In Tree Maze, we allowed VAPS(1) to have 2 bits of memory while FW and SMM + IM had a memory capacity of \(c = 2\) observations \((e_t = o_t)\) so that all explicit memory based methods have sufficient capacity to learn the task.

As shown in Fig. 16a, only VAPS(1) and Sarsa(\(\lambda\)) with SMM + IM learns in Tree Maze, while the others show only random achievements. Sarsa(\(\lambda\)) with SMM + IM outperforms all others by reaching a near optimal performance with a total reward of 9.86 in the last episodes. The methods using LSTM, A2C and PPO are unable to devise a good policy, supporting the argument that they require specific configurations for a problem to learn it. Similar to the other results, the number of changes that SMM + IM applies to the memory converges to a minimum value, as Fig. 16b shows. Note that SMM + IM does not fix the content of the memory at the beginning of an episode in this setting, but updates it when the corresponding observation is no longer required to solve the task. For example, the first observation in the episode contains the direction that the agent needs to take in the first T-junction. SMM + IM removes this observation from the memory by storing a new one when the agent takes that turn. This shows that SMM + IM learns a suitable behaviour for the task.

In our experiments in Basic Scheduler environment, we allowed FW and SMM to have a memory with a capacity of \(c = 1\) observations \((e_t = o_t)\). However, our experiments with VAPS(1) with different memory sizes showed no success. VAPS(1) makes an update on the whole table of values after each step, and therefore the bigger size of this environment caused the algorithm to run very slowly.

Figure 17a shows the total collected reward in the Basic Scheduler environment. In parallel with the other results, Sarsa(\(\lambda\)) with SMM + IM outperforms its competitors by learning to assign the CPU to an unfinished job. Keeping the rare observation that gives which jobs are finished in memory, SMM + IM directs the agents to a policy that
Learning what to memorize...

Fig. 17 (a) Total reward collected throughout an episode, (b) number of changes on explicit memory during an episode in Basic Scheduler problem. Both Sarsa(λ) with FW and Sarsa(λ) with SMM + IM has a memory with a capacity of c = 1 observations

avoids assigning the CPU to an already completed job and therefore avoids punishment. The fixed window memory approach and LSTM based methods are not capable of distinguishing this observation, thus their episodes end early. Sarsa(λ) with SMM + IM updates its memory 7 times on average as given in Fig. 17b. This directly shows that our method keeps the new state of the jobs whenever one finishes since 7 jobs finish during a successful episode.

6 Conclusion

This study proposes a main layout in which a reinforcement learning agent learns how to manage its own memory, with an intrinsic motivation to keep useful events. The proposed method, Self Memory Management with Intrinsic Motivation, defines the form of the memory, the actions changing the memory, and the intrinsic reward functions to be provided to the learning updates. It can be combined with many reinforcement learning algorithms and is shown to work in partially observable environments that have long-term dependency. In the experiments, we compared our method with the original study proposing the idea of allowing the agent to control the memory, and showed that, using intrinsic motivation, our method outperforms several memory based approaches in various POMDPs, with minimal changes on the memory.

To focus specifically on the analysis of our proposed method as a concept, we limit the scope of this study to discrete tabular Reinforcement Learning. However, our method can be employed on large and continuous domains provided that a proper discretization process is applied.

This study can be extended in three ways. First, alternative sets of memory actions $A$ can be employed to allow the agent to take extended modifications on the memory and the capacity bounding parameter $c$ can be disregarded, giving the agent full control over the memory size. Second, the definitions of intrinsic reward can be broadened, as some problems may require further heuristics on what to keep in the memory in addition to the rare events. Finally, SMM and IM can be combined with Deep Reinforcement Learning approaches, providing them with explicit and analyzable memory.

Acknowledgements This work is supported by the Scientific and Technological Research Council of Turkey under Grant No. 120E427. Authors would also like to thank Hüseyin Aydin, Erkin Çilden and Faruk Polat for their support.

References

1. Astrom KJ (1965) Optimal control of markov processes with incomplete state information. J Math Anal Appl 10(1):174–205
2. Bakker B (2001) Reinforcement learning with LSTM in non-Markovian tasks with longterm dependencies. Memory 1–18
3. Barto AG (2013) Intrinsic motivation and reinforcement learning. In: Intrinsically motivated learning in natural and artificial systems. Springer, pp 17–47
4. Bellemare MG, Srinivasan S, Ostrovski G, Schaul T, Saxton D, Munos R (2016) Unifying Count-Based exploration and intrinsic motivation. NIPS
5. Böhmer W., Springenberg JT, Boedecker J, Riedmiller MA, Obermayer K (2015) Autonomous learning of state representations for control: an emerging field aims to autonomously learn state representations for reinforcement learning agents from their real-world sensor observations. Künstliche Intell 29(4):353–362. https://doi.org/10.1007/s13218-015-0356-1
6. Chrisman L (1992) Reinforcement learning with perceptual aliasing: the perceptual distinctions approach. In: AAAI'92. AAAI Press, pp 183–188
7. Henderson P, Islam R, Bachman P, Pineau J, Precup D, Meier D (2018) Deep reinforcement learning that matters. In: AAAI 2018, pp 3207–3214
8. Hill A, Raffin A, Ernestus M, Glaave A, Kanervisto A, Traore R, Dhariwal P, Hesse C, Klimov O, Nichol A, Plappert M, Radford A, Schulman J, Sidor S, Wu Y (2018) Stable baselines. https://github.com/hill-a/stable-baselines

9. Icarte RT, Valenzano RA, Klassen TQ, Christoffersen P, Farahmand A, McIlraith SA (2020) The act of remembering: a study in partially observable reinforcement learning. arXiv:2010.01753

10. Icarte RT, Walddie E, Klassen TQ, Valenzano RA, Castro MP, McIlraith SA (2019) Learning reward machines for partially observable reinforcement learning. In: NeurIPS 2019, pp 15497–15508

11. James MR, Singh S (2004) Learning and discovery of predictive state representations in dynamical systems with reset. In: Proceedings of the twenty-first international conference on Machine learning, p 53

12. Kulkarni TD (2016) Deep reinforcement learning with temporal abstraction and intrinsic motivation. In: Advances in neural information processing systems, vol 48, pp 3675–3683

13. Lanzi PL (2000) Adaptive agents with reinforcement learning and internal memory. From Animals to Animats 6

14. Le TP, Vien NA, Chung T (2018) A deep hierarchical reinforcement learning algorithm in partially observable markov decision processes. IEEE Access 6:49089–49102

15. Li R, Cai Z, Huang T, Zhu W (2021) Anchor: The achieved goal to replace the subgoal for hierarchical reinforcement learning. Knowl-Based Syst 225:107128. https://doi.org/10.1016/j.knosys.2021.107128

16. Lin LJ, Mitchell TM (1992) Memory approaches to reinforcement learning in non-Markovian domains. Citeseer

17. Littman ML (1994) Memoryless policies: theoretical limitations and practical results. In: From animals to animats 3, pp 238–245

18. Littman ML, Sutton RS (2002) Predictive representations of state. In: Advances in neural information processing systems, pp 1555–1561

19. Loch J, Singh SP (1998) Using eligibility traces to find the best memoryless policy in partially observable markov decision processes. In: ICML’98. Morgan Kaufmann Publishers Inc, pp 323–331

20. McCallum A (1996) Reinforcement Learning with Selective Perception and Hidden State. Ph.D. thesis, University of Rochester NY

21. Meuleau N, Kim KE, Kaelbling LP, Cassandra AR (1999) Solving POMDPs by searching the Space of Finite Policies. UAI ’99 pp 417–426

22. Meuleau N, Peshkin L, Kim K, Kaelbling LP (1999) Learning finite-state controllers for partially observable environments. In: Laskey KB, Prade H (eds) UAI ’99. Morgan Kaufmann, pp 427–436

23. Mnih V, Badia AP, Mirza M, Graves A, Lillicrap T, Harley T, Silver D, Kavukcuoglu K (2016) Asynchronous methods for deep reinforcement learning. In: ICML 2016. PMLR, pp 1928–1937

24. Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, Riedmiller M (2013) Playing Atari with deep reinforcement learning. arXiv:1312.5602

25. Ostrovski G, Bellemare MG, Van Den Oord A, Munos R (2017) Count-based exploration with neural density models. In: ICML 2017, vol 6, pp 4161–4175. JMLR.org

Publisher’s note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Alper Demir is an Assoc. Professor at Computer Engineering Department in izmir University of Economics, izmir. He received the B.S., M.Sc. and Ph.D. degrees in Computer Engineering at the Department of Computer Engineering, Middle East Technical University, Ankara, in 2014, 2016 and 2019, respectively. His research interests include machine learning and data science.