Learning What to Remember: Strategies for Selective External Memory in Online Reinforcement Learning Agents

by

Kenneth Young

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

In realistic environments, intelligent agents must learn to integrate information from their past to inform present decisions. An agent's immediate observations are often limited, and some degree of memory is necessary to complete many everyday tasks. However, an agent cannot remember everything it observes. The history of observations may be arbitrarily long, making it impractical to store and process. In this thesis, we will develop a novel method, called *online policy gradient over a reservoir* (OPGOR), for selecting what to remember from the stream of observation. We will also explore a number of alternative methods for handling this selective memory problem.

OPGOR contrasts with recurrent neural networks (RNNs), which handle the task of selective memory by propagating information forward one step at a time. RNNs are very practical to a point but suffer on longer time horizons due to the difficulty of maintaining information over many steps with a mechanism based on one step updates. Furthermore, RNNs are generally trained by backpropagation through time (BPTT) which requires, at each time-step, compute time proportional to the length of the history. To handle this in practice, backpropagation is only performed over a fixed time window, further limiting its utility over long time horizons.

OPGOR operates within the framework of *external memory mechanisms* for selective memory, which use explicit read and write operations instead of recurrent updates. An external memory consists of a fixed number of constant length vectors, or slots, in which relevant information can be written and later read. Such mechanisms give rise to three key questions: what to read from memory, what to write to memory, and what to drop from memory when something is written.

External memory mechanisms have some advantages over RNNs for learning long-term dependencies, but in many cases they are still trained using BPTT, in which case they require similarly restrictive computation time. We will focus on external memory mechanisms which avoid the use of BPTT. Such mechanisms generate a vector, which we call a *state variable*, at each time-step and then, perhaps stochastically, decide whether to write it to memory, replacing an existing state variable in the process. External memory mechanisms that forgo BPTT often use a heuristic to decide which item to replace. Such heuristic replacement mechanisms can work well if the memory is very large but may fail if there are dependencies over much larger timescales than the memory size.

This thesis will focus on the question of how to learn to prioritize which information is written to and retained in an external memory. We will focus on the online case, where a single agent acts and learns concurrently, with a limited amount of memory and compute time. In doing so, we hope to produce agents that can learn to perform well, while storing much less information. Our primary approach, OPGOR, will apply policy gradient to the process of selecting which state variables to store in memory from the entire trajectory.

Naively applying policy gradient to draw a subset of the full history of state variables would require us to store the full history of state variables and then draw a sample. This is not feasible for an online method. On the other hand we can easily make a stochastic decision at each time-step regarding whether to add the current state variable, and if so which one to replace. One could ask whether we can construct an incremental stochastic procedure such that the marginal probability of inclusion for any given state variable in an observed sequence is as if we had drawn a sample from the full history of state variables.

A variety of algorithms exist which maintain a fixed sized sample with particular statistical properties from a stream observed one item at a time. Such algorithms are called *reservoir sampling algorithms*, named for the fact that they maintain a fixed size sample, or *reservoir*, of items drawn from a stream. The goal is to ensure, through specific add and drop probabilities, that the n items in the reservoir at each time-step correspond to a sample with certain desired statistical properties over n-subsets of all observed items. This is done without ever storing more than n items from the stream.

In this thesis we will use a reservoir sampling algorithm to maintain an external memory where the inclusion probability for each state variable in the history is given by a differentiable, closed form expression. Reservoir sampling allows us to achieve this while storing only a small fixed number of state variables. Our main contribution will be to combined this technique with policy gradient to tune the inclusion probabilities of each state variable online, to improve expected return. The resulting procedure, OPGOR, allows us to efficiently train our memory to maintain useful state variables.

We test OPGOR, along with a number of alternative selective memory strategies, on a set of psychology inspired problems, simplified to focus on the specific aspects of the problem we aim to investigate. In doing so, we explore the challenges of deciding what to retain in memory and to what degree various methods handle them.

Acknowledgements

First and foremost, I thank my supervisor Richard Sutton for his patience and encouragement over the last couple of years. His commitment to clear understanding and precise thinking is something I admire deeply, and that I hope to carry into all my future research. He has opened my mind to many exciting research directions, and helped me immeasurably to frame my own ideas as clearly and accurately as possible. He continues to provide a comfortable and engaging environment for research and I greatly appreciate the community he has helped to establish.

I also wish to thank all the students and researchers in the RLAI lab for numerous engaging discussions, which never fail to remind me of how exciting and vibrant our field can be. In particular I wish to thank Roshan Shariff for the numerous times he has helped me to wrap my head around a difficult concept, or to turn a half baked idea into something tangible. I am continually impressed by his ability to soak in the essence of a concept and clearly instill his intuition and insight onto others.

Lastly I wish to thank NSERC for providing me with the funding which has allowed me to fully focus on the research I love.

Contents

List of Tables

List of Figures

Chapter 1

Learning What to Remember

In realistic environments, intelligent agents must learn to integrate information from their past to inform present decisions. The reinforcement learning (RL) framework, which we will utilize in this thesis, provides powerful general techniques, which allow an agent to learn to optimize arbitrary reward signals in arbitrary environments. One area where RL remains limited, however, is the oft applied assumption that given the current state, future states and rewards are independent of all past states. This is called the Markov assumption, and the practical implication is that an agent has all the information it needs to make correct decisions and predictions at each time-step, and need not remember its past.

In realistic problems, an agent's immediate observations are often limited, and some information about the past is necessary to make informed decisions. Since it is not practical to store, or to process, the entire sequence of observations, an agent must have some method for selectively storing information from its past. Various methods exist which attempt to handle this problem. One common approach is to maintain a *hidden state*, updated with each new observation. The update function is adjusted over time, learning to store information pertinent to future decision making. Other approaches utilize external memories which an agent can explicitly write to and read from using various mechanisms.

A major contribution of this thesis will be introducing a method, called *online policy gradient over a reservoir* (OPGOR), for handling the problem of selecting what to remember from the stream of observation. OPGOR operates on a *reservoir* of data drawn from the stream of observations, and attempts to optimize this limited storage such that what is remembered helps to facilitate good performance. OPGOR does this by applying *policy gradient*, which facilitates improvement through trial and error by increasing the probability of selections which lead to high return. OPGOR works *online*, which here means it operates in real time, on the stream of experience generated by a single agent interacting with an environment, without storing a large amount of extra data to learn from.

OPGOR contrasts with *recurrent neural networks* (RNNs), which handle the task of selective memory by propagating information forward one step at a time. More precisely, RNNs maintain a hidden state which is recurrently updated with each new observation. In the simplest case of an ordinary RNN, the state update consists of a matrix multiplication of the last hidden state concatenated with the current observation, followed by a nonlinear activation to produce the next hidden state. Explicitly, if h_t is the RNN state and x_t is the input at time t, an RNN updates it's hidden state as follows:

$$h_t = \sigma(Wx_t + Uh_{t-1} + b),$$

where W and U are learned matrices, b is a learned bias vector and σ is a nonlinear activation. Though it is in principle possible to propagate information over arbitrary lengths of time with such a mechanism, the form of the update can make long-term dependencies difficult to learn. One reason for this is the lack of any specific mechanism for retaining information over time. Since each update is defined by a matrix multiplication, in order to maintain some part of the hidden state over many time-steps the system must explicitly learn something like an identity transformation, and small deviations may cause the hidden state to decohere in unpredictable ways. This manifests in the well known vanishing (or exploding) gradient problem, where the gradients with respect to past inputs quickly become negligible (or inordinately large).

To fight the vanishing gradient problem more modern RNN variants generally make use of learnable gating mechanisms. Gates attenuate the updates to certain elements of the RNN state by multiplying by a number between zero and one. This gives the system the ability to explicitly choose to retain information over many time-steps by consistently gating its update with a value near zero. *Long short term memory* (LSTM) (Hochreiter & Schmidhuber, 1997) and *gated recurrent unit* (GRU) (Cho, Van Merriënboer, Bahdanau, & Bengio, 2014) networks are two commonly used examples of such gated systems. GRUs make use of two gates, a reset gate for clearing information currently stored in an element of the state, and an update gate for adding information to an element. Explicitly, the GRU updates its state as follows:

$$\begin{aligned}
z_t &= \sigma_g(W_z x_t + U_z h_{t-1} + b_z),\\
r_t &= \sigma_g(W_r x_t + U_r h_{t-1} + b_r),\\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \sigma(W x_t + U(r_t \odot h_{t-1}) + b),
\end{aligned}$$

where σ_g is a sigmoid activation constraining z_t and r_t to be between one and zero, W_z, W_r, W, U_z, U_r and U are all learned matrices and b_z, b_r and b are learned bias vectors. The operator $\odot$ denotes an element-wise product. Notice that, by setting elements of the update gate z_t to a value near one, this mechanism allows elements of the hidden state from the last time-step to be maintained as is. On the other hand, setting elements of the reset gate r_t near zero allows elements of the hidden state to be explicitly cleared. In addition to these two gates, LSTM also uses an output gate for gating information read from an element. These mechanisms help to greatly reduce the vanishing or exploding gradient problem by providing a unit gradient pathway between the hidden state at one time-step and the next.

Gating mechanisms help to extend the utility of recurrent systems to much larger timescales, however they still have limitations. Gates provide a convenient way to learn to retain information, however ultimately a sequence of one step updates still needs to be learned. As dependencies become longer, it becomes increasingly difficult to learn what information is worth propagating over many steps, even with a gating mechanism. Further difficulty arises because RNNs, LSTMs and GRUs are all usually trained with *backpropagation through time* (BPTT). BPTT unrolls the computation as if the series of recurrent updates were one large network with one layer for each observation seen in the history so far. It then backpropagates errors from the current time-step through all of these layers to determine the gradient of these errors with respect to each network parameter over all time-steps. This requires compute time proportional to the length of the history at each update step. To handle this in practice, backpropagation is generally only performed over a fixed time window, which further limits the systems ability to learn long-term dependencies.

Instead of recurrent updates, OPGOR builds on an alternative framework for selective memory, which utilizes an explicit memory that is external to the network. Like gating mechanisms, these *external memory mechanisms* also aim to provide a learning agent with the ability to store information over many time-steps, but take this idea further. While gating mechanisms provide a way to selectively maintain individual elements of the hidden state, an external memory provides a series of slots in which whole vectors can be stored and subsequently read. By using addressing mechanisms which explicitly select slots to write to and read from, such systems allow an agent to store drastically more information, with less learning, and perhaps less computation, than would be associated with an RNN of similar capacity. Unlike a gating mechanism, interaction with memory is constrained to be highly sparse, which biases the system towards maintaining information for a longer period of time. External memory based systems give rise to three key questions: what to read from memory, what to write to memory and what to drop from memory when something is written.

External memory mechanisms have some advantages over RNNs for learning long-term dependencies, but in many cases they are still trained using BPTT. Training an external memory with BPTT requires that the read and write mechanisms be differentiable. In practice this means that reading and writing are fractional operations. A write operation updates or partially overwrites the content of each slot, while a read operation returns a weighted average over the values in each slot.

External memory mechanisms trained with BPTT have the same computational issue inherent in RNNs trained similarly. In particular, training with BPTT loses much the computational benefit of maintaining a limited memory, as in order to train a system in this manner we must actually record the full history in order to backpropagate over it. If instead we truncate the backpropagation to a fixed number of steps, as is usually done when training RNNs, it is unclear how the system could learn dependencies over a longer timescale than the truncation length. Items stored for longer than the truncation length could not be traced back to when they were written.

We will focus on external memory mechanisms which avoid the use of BPTT. Such mechanisms generate a vector, which we call a *state variable*, at each time-step and then, perhaps stochastically, decide whether to write it to memory, replacing an existing state variable in the process. In this case the writing mechanism is not fractional, the state variable either entirely replaces another in memory, or is not written at all. The state variable may be the hidden state of an RNN, or generated through some other mechanism. Generally, external memory mechanisms that forgo backpropogation through time instead use a simple heuristic to decide which item to replace when the memory becomes full. Possible heuristics include dropping the least recently used, or even simply maintaining the last n state variables. Such heuristic replacement mechanisms can work well if the memory is very large but may fail if there are dependencies over much longer timescales than the memory size. Reading from memory may still be based on a differentiable mechanism similar to the one described above. Note however, that differentiable reading does not require BPTT, as we need only select from the current memory contents, which are immediately available. Alternatively, reading from memory may be stochastic, and trained by a trial and error procedure, such as policy gradient.

Deciding what to write to and retain in memory poses some interesting challenges. Once we have decided what is worth putting in memory it is relatively straightforward to figure out how to use it in a given time-step, for example by training a system to make queries, either content addressable or based on separate keys and values. Selecting what to remember, however, is in some sense a much more difficult problem. The contents of memory at a given time are in principle a function of the entire observation history. This is made more difficult because decisions about what is worth remembering must be made on the fly, once we decide not to remember some information it cannot be recovered at a later time.

This thesis will focus in particular on the problem of selecting what to remember in the context of an external memory based system. We aim to build a system that can learn online to use a limited external memory capacity effectively, without introducing a large amount of additional time and memory complexity. This precludes BPTT, since BPTT requires recording the entire history, at least up to a specified truncation length.

Our primary approach to selective memory will be to apply policy gradient to the process of choosing which state variables are stored in memory from the entire history. This will allow the system to learn over time to remember things which are found to facilitate better than expected return when they are recalled in the future. To do this naively we would have to maintain the entire history in order to sample from it.

Given we want our agent to operate online, storing the entire history will not be feasible. Consider instead an online procedure which maintains only a finite sized memory and, at each time-step, stochastically decides whether to add the new state variable, and if so which one to replace. Note that, for a fixed sequence of state variables, such a procedure necessarily gives rise to a certain marginal distribution over sets of state variables contained in memory after observing the full sequence. One could ask whether it is possible to tune the online add and drop probabilities such that the marginal inclusion probability for each state variable has a useful closed form.

A number of algorithms exist which maintain a fixed-size sample with particular statistical properties from a stream observed one item at a time. Such algorithms are called *reservoir sampling algorithms*, named for the fact that they maintain a fixed-size set, or *reservoir*, of items drawn from a stream. The goal is to ensure, through specific add and drop probabilities, that the n items in the reservoir at each time-step correspond to a sample with certain desired statistical properties over n-subsets of all observed items. This is done without ever storing more than n items from the stream.

In this thesis, we will use a reservoir sampling algorithm which maintains an external memory, in which the inclusion probability for each state variable in the history is given by a differentiable, closed form expression. Reservoir sampling allows us to achieve this while only ever storing a small fixed number of state variables. In particular, the reservoir sampling technique we apply gives closed form, differentiable inclusion probabilities for each state variable in terms of an associated weight. In our case, this weight is generated by an artificial neural network (ANN) for each observed state variable. This allows us to apply the techniques of policy gradient to improve the weights assigned to state variables in memory with respect to the resulting expected return. The resulting procedure is online policy gradient over a reservoir (OPGOR).

Our main contribution is to introduce the OPGOR method. In doing so we demonstrate how policy gradient can be applied to a reservoir sampling algorithm to manage an external memory in an online RL context.

A secondary contribution is to perform an empirical exploration of a number of different selective memory techniques for reinforcement learning, including OPGOR. This evaluation will consist of several episodic reinforcement learning problems. Each of these problems is designed to test an agent's ability to store and recall pertinent information from earlier in an episode to inform present decision making. In the process we validate the performance of OPGOR when the assumptions of our derivation hold.

We also expose a weakness of our first version of OPGOR, when a certain assumption does not hold, which it may not in many realistic problems. We demonstrate when this issue occurs, and derive another version of the OPGOR, OPGOR-DS, which mitigates it.

Aside from evaluating OPGOR we found in our experiments that the relative performance of various selective memory strategies is highly problem dependent. We provide insight into why certain technique may work better in certain settings and worse in others. In particular, we found that the least recently used heuristic, variants of which have been used in a number of prior works, was surprisingly robust despite its simplicity.

Chapter 2

Background

In this chapter, we will provide the relevant background necessary to understand the rest of the thesis. We begin with an introduction to the RL problem, along with the solution approach of actor-critic algorithms. We will then discuss how partial observability can be introduced into the RL framework. The final section provides an overview of external memory based approaches to solving partially-observable problems, on which OPGOR is built.

2.1 Reinforcement Learning

We consider the RL problem, where a learning agent interacts with an environment, while striving to maximize a reward signal. The problem is generally formalized as a Markov decision process described by a 5-tuple: $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$. At each time-step the agent observes the *state* $S_t \in \mathcal{S}$ and selects an *action* $A_t \in \mathcal{A}$. Based on S_t and A_t, the next state S_{t+1} is generated, according to a *transition probability* $p(S_{t+1}|S_t, A_t)$. The agent additionally observes a *reward* R_{t+1}, generated by $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. RL is concerned not with the immediate reward, but with a temporally discounted sum $G_t = \sum\limits_{k=t}^{\infty} \gamma^{k-t} R_{k+1}$, known as the *return*. The *discount factor* γ controls how much we care about long-term rewards versus short-term rewards. Algorithms for RL broadly fall into two categories, prediction and control; this thesis will focus primarily on control.

In the prediction task, an agent follows a fixed policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, which assigns probabilities to each action conditioned on the state. From the current state S_t, the agent aims to estimate the expectation value of the return

$G_t = \sum_{k=t}^{\infty} \gamma^{k-t} R_{k+1}$, with discount factor $\gamma \in [0,1]$. We refer to $E_\pi[G_t|S_t = s]$ as the value $v_\pi(s)$ of state s under policy π. We will write our estimate of the value as $\hat{v}(s,\theta)$, where θ is a set of parameters which define the current value estimate. We will write the policy as $\pi(a|s)$ to emphasize that it is a state conditional probability distribution over actions. If a problem is such that an agent eventually reaches an absorbing state where all further rewards are zero, it is said to be *episodic*, and the return need only be computed until the absorbing state is reached. Otherwise, if the agent continues interacting with the environment indefinitely we say the problem is *continuing.* In the continuing case it is necessary to have γ strictly less than one for G_t to be well defined.

In the control task, the goal is to learn, through interaction with the initially unknown environment, a policy $\pi(a|s,\theta)$ that maximizes the expected return G_t with respect to the parameters θ, with discount factor $\gamma \in [0,1]$. For this to be a meaningful objective it is necessary to choose the distribution of states over which we want to maximize expected return. In the episodic case, a reasonable choice is to maximize expected return from the start state. In the episodic case, discounting may be unnecessary but could still prove useful algorithmically for biasing learning towards short-term rewards, which generally have lower variance and are thus easier to learn. Our experiments will focus on the episodic case, where we hold the view that it is better to think of γ as merely a parameter of the solution method, while acknowledging that the objective we really wish to optimize is the undiscounted return (Schulman, Moritz, Levine, Jordan, & Abbeel, 2015). In the continuing case, we may instead choose to optimize over the steady state distribution (i.e., the expected long-term state occupancy under the current policy). The expected return objective over the steady state distribution is identical to the average reward objective, independent of the choice of γ (Sutton & Barto, 2017).

Action-value methods like Q-learning are often used for RL control, however, we will focus on an alternative class of algorithms for control, known as actor-critic (AC) methods. AC methods separately learn a state value function $\hat{v}(s,\theta)$, known as the *critic*, which approximates $v_\pi(s)$ for the current policy and a parameterized policy $\pi(a|s,\theta)$, known as the *actor*, which attempts to maximize that value function. The policy and value function are both parametrized by a set of parameters θ. The parameters controlling $\hat{v}(s,\theta)$ may be disjoint from those controlling $\pi(a|s,\theta)$ but in the general case some parameters may be shared. We will build our work on actor-critic with eligibility traces $AC(\lambda)$ (Degris, Pilarski, & Sutton, 2012; Schulman et al., 2015). While eligibility traces are often associated with prediction methods like $TD(\lambda)$ they are also applicable to AC.

AC methods have a number of advantages over action value methods for control (Sutton & Barto, 2017). Firstly, they allow the policy to approach a deterministic policy in a natural way. Action value methods on the other hand require a choice of how to map action values to an associated policy. Generally, this includes some stochasticity to facilitate exploration. While this stochasticity can be made arbitrarily small, it is difficult to strike a balance between exploration and exploitation of the current best action. AC allows this balance to be learned over time as the policy is only adjusted when it is found to be beneficial. On the other hand AC is capable of explicitly learning a stochastic policy when it is beneficial to do so. Stochastic policies can be beneficial in the presence of an adversary, to limit ones predictability. They can also be beneficial when the environment is *partially-observable*, meaning only some information about the state is available to the agent. This is because, in the presence of partial observability, an agent may have no way to know the best action. Choosing stochastically allows a chance of success, where a deterministic policy may get stuck indefinitely. This thesis focuses on partially-observable environments, which is one reason we choose to focus on AC.

To specify the objective of $TD(\lambda)$, and by extension $AC(\lambda)$, we first define the λ-return G_t^λ. Here we will define G_t^λ recursively:

$$G_t^\lambda = R_{t+1} + \gamma\left((1-\lambda)\hat{v}(S_{t+1},\theta) + \lambda G_{t+1}^\lambda\right).$$

G_t^λ bootstraps future evaluations to a degree controlled by λ. If $\lambda < 1$, then G_t^λ is a biased estimate of the return, G_t. If $\lambda = 1$, then G_t^λ reduces to G_t.

Defining the TD-error $\delta_t = R_t + \gamma \hat{v}(S_{t+1}, \theta) - \hat{v}(S_t, \theta)$, we can expand G_t^λ as the current state value estimate plus the sum of future discounted δ_t values:

$$G_t^\lambda = \hat{v}(S_t, \theta) + \sum_{k=t}^{\infty} (\gamma\lambda)^{k-t} \delta_k.$$

This form is useful in the derivation of $TD(\lambda)$ as well as $AC(\lambda)$. $TD(\lambda)$ can be understood as minimizing the mean squared error $\left(G_t^\lambda - \hat{v}(S_t, \theta)\right)^2$ between the value estimate $\hat{v}$ and the λ-return G_t^λ. In deriving $TD(\lambda)$, the target G_t^λ is taken as constant despite its dependence on the parameters of the value function. For this reason, $TD(\lambda)$ is often called a *semi-gradient* method. Intuitively, we want to modify our current estimate to match our future estimates and not the other way around. For $AC(\lambda)$, we will combine this mean squared error objective with a policy improvement term, such that the combined objective represents a trade-off between the quality of our value estimates and the performance of our policy:

$$\mathcal{J}_\lambda(\theta) = \frac{1}{2} \left(\sum_{t=0}^{\infty} \left(G_t^\lambda - \hat{v}(S_t, \theta)\right)^2 - \sum_{t=0}^{\infty} \log(\pi\left(A_t | S_t, \theta\right)) \left(G_t^\lambda - \hat{v}(S_t, \theta)\right) \right). \tag{2.1}$$

As in $TD(\lambda)$, we apply the notion of a semi-gradient to optimizing equation 2.1. In this case along with $G_{\theta,t}^\lambda$, the appearance of $\hat{v}(S_t, \theta)$ in the right sum is taken to be constant. Intuitively, we wish to improve our actor under the evaluation of our critic, not modify our critic to make our actor's performance look better. With this caveat in mind, by the policy gradient theorem (Sutton, McAllester, Singh, & Mansour, 2000), the expectation of the gradient of the right term in equation 2.1 is approximately equal to the (negated) gradient of the expected return. This approximation is accurate to the extent that the expectation of $\left(G_t^\lambda - \hat{v}(S_t, \theta)\right)$, which we will refer to as the *advantage estimate*, accurately approximates the *advantage* $E[G_t | S_t, A_t] - E[G_t | S_t]$. Descending the gradient of the right half of $\mathcal{J}_\lambda(\theta)$ is then ascending the gradient of an estimate of expected return. Taking the semi-gradient of equation 2.1 yields:

$$\begin{aligned}
\frac{\partial}{\partial\theta}\mathcal{J}_\lambda(\theta) &= -\sum_{t=0}^{\infty}\left(\frac{\partial\hat{v}(S_t,\theta)}{\partial\theta}+\frac{1}{2}\frac{\partial\log(\pi\left(A_t|S_t,\theta\right))}{\partial\theta}\right)\left(G_t^\lambda-\hat{v}(S_t,\theta)\right)\\
&= -\sum_{t=0}^{\infty}\left(\frac{\partial\hat{v}(S_t,\theta)}{\partial\theta}+\frac{1}{2}\frac{\partial\log(\pi\left(A_t|S_t,\theta\right))}{\partial\theta}\right)\sum_{k=t}^{\infty}(\gamma\lambda)^{k-t}\delta_k\\
&= -\sum_{t=0}^{\infty}\delta_t\sum_{k=0}^{t}(\gamma\lambda)^{t-k}\left(\frac{\partial\hat{v}(S_k,\theta)}{\partial\theta}+\frac{1}{2}\frac{\partial\log(\pi\left(A_k|S_k,\theta\right))}{\partial\theta}\right).
\end{aligned}$$

Define the eligibility trace at time t as:

$$z_t=\sum_{k=0}^{t}(\gamma\lambda)^{t-k}\left(\frac{\partial\hat{v}(S_k,\theta)}{\partial\theta}+\frac{1}{2}\frac{\partial\log(\pi\left(A_k|S_k,\theta\right))}{\partial\theta}\right),$$

such that:

$$\frac{\partial}{\partial\theta}\mathcal{J}_\lambda(\theta)=-\sum_{t=0}^{\infty}\delta_t z_t. \tag{2.2}$$

Offline $AC(\lambda)$ can be understood as performing a gradient descent step along equation 2.2. Online $AC(\lambda)$, analogous to online $TD(\lambda)$ can be seen as an approximation to this offline version that updates parameters after every time-step using a single term in the above sum. This approximation is exact in the limit where the *step-size* α is low enough to give quasistatic parameters. Advantages of the online version include making immediate use of new information, and being applicable to continuing problems. Online $AC(\lambda)$ is defined by the following set of equations:

$$\begin{aligned}
z_t &= \gamma\lambda z_{t-1}+\frac{\partial\hat{v}(S_t,\theta_t)}{\partial\theta}+\frac{1}{2}\frac{\partial\log(\pi\left(A_t|S_t,\theta_t\right))}{\partial\theta},\\
\theta_{t+1} &= \theta_t+\alpha z_t\delta_t.
\end{aligned}$$

Most of the theory around AC methods assumes that the actor is updated on a slower timescale than the critic. This is to ensure that the critic is actually able to adapt to the current policy before the policy significantly changes. In practice, however, updating the two simultaneously at a similar rate generally works well. One could speculate as to why this seems to work so well. One possible explanation is that the critic need not learn a value function for the current policy exactly, but something like a running average of recent policies.

If the policy updates tend toward improving on this running average it will tend to get better in the long run even if the critic is never properly adapted to any fixed policy. We will not explore this further in the present thesis and simply note that we will perform our actor and critic updates on a similar timescale.

2.2 Partial Observability

In Section 2.1 we assume the agent is given a state S_t at each time-step, such that the value function and policy are restricted to depend only on S_t. The agent's task is to optimize performance under this constraint. When the environment has the property that S_t really does provide all the relevant information to make optimal decisions and predictions, or more precisely when future states and rewards are conditionally independent of past states given the present state, the environment is said to have the Markov property. In realistic problems such a state is rarely available. Nevertheless, the framework of function approximation is sufficiently powerful to reason about RL when a Markov state S_t is not available. If some pertinent aspect of the environment is hidden from the agent, we can simply consider this to be a limitation of the parameterized form of $\hat{v}(s, \theta)$ and $\pi(a|s, \theta)$ (Sutton & Barto, 2017).

While sound, considering partial observability as merely a limitation of the function parameterization ignores the possibility of improving performance by integrating certain aspects of the observation history into the state representation. To be precise, assume that rather than state, the environment outputs at each time-step only an observation O_t. O_t gives some limited information about the agent's current situation while keeping other information hidden. In this case we can recover something like a Markov state in the form of a full history of observations and actions up to the current time $H_t = O_0, A_0, O_1, A_1, ..., O_t$. In some sense this must be considered a sufficient representation because it represents all the information the agent could possibly have at time t. Any information not available in the history can be considered part of the stochasticity of the environment. Note, however, that any function approximation we apply over the full history must be severely restricted. The full history may be unbounded in length, it may not be practical to even maintain it in memory, much less input it directly to something like an ANN. Once we drop the assumption that our agent is privileged with all the information it could need at each time-step, we can start to think about how we might propagate information received in the past to better inform our present value and policy.

In this thesis, information will be propagated via objects we refer to as state variables, a term introduced by Wayne et al. (2018). The state variable generated at time t, ϕ_t, is a low dimensional function of the history available at each time-step, on which an agent can condition its value estimate and policy. For simplicity, in this thesis, ϕ_t will be generated by the environment, making it equivalent to an observation. In general, the state variable may be something more elaborate, such as the hidden state of an RNN or, as in the MERLIN architecture of Wayne et al. (2018), a sample drawn from a learned belief distribution over predictive states. In addition to conditioning the current value estimate and policy, state variables will be stored in an external memory, which will be used to condition later action selection in a manner described in detail in Chapter 5.

2.3 External Memory Systems and Writing Mechanisms

Deep learning systems which make use of an external memory have received a lot of interest lately. Such systems are broadly motivated by the desire to expand the capabilities of neural network based systems to allow more general computation, and to operate over longer timescales than is possible with RNNs. Two prototypical examples are *neural Turing machines* (NTM) (Graves, Wayne, & Danihelka, 2014) and the follow-up, *differentiable neural computers* (DNC) (Graves et al., 2016). These systems use an LSTM controller attached to read and write heads of a fully differentiable external memory and train the combined system to perform algorithmic tasks.

The NTM architecture uses a writing mechanism inspired by the reset and update gates of systems like GRU and LSTM. At each time-step t each of some fixed number of write heads generates a weight vector $\mathbf{w}_t$ over available memory slots with elements summing to 1. Each write head also produces an erase vector $\mathbf{e}_t$ and add vector $\mathbf{v}_t$, each of size equal to the length of the vector stored at each memory location. An update to the memory $\mathcal{M}_t(i)$ in slot i is then computed as follows:

$$\mathcal{M}_t(i) = \mathcal{M}_{t-1}(i)(\mathbf{1} - \mathbf{w}_t(i)\mathbf{e}_t) + \mathbf{w}_t(i)\mathbf{v}_t,$$

where $\mathbf{w}_t(i)$ represents the i^{th} element of $\mathbf{w}_t$, $\mathbf{e}_t$ determines which elements of selected slots are cleared, while $\mathbf{v}_t$ determines which are updated. The write weight vector $\mathbf{w}_t$ determines which slots to update and is constrained to sum to one such that the update is sparse. The elements of $\mathbf{w}_t$ are generated by a combined content and relative location based addressing scheme. The content addressable part uses cosine similarity between a learned query and each item stored in memory. The location addressable part is implemented as a learned shift weight which acts to add write weight to the slots directly above and below the last written locations. Another learned gate is used to interpolate between the content addressable and location addressable parts of the write mechanism.

The follow up architecture to the NTM, the DNC also utilizes a content addressable writing mechanism, but removes the location addressable writing mechanism and adds a dynamic memory allocation mechanism. The dynamic allocation mechanism maintains a usage counter which is increased proportional to the write weight each time a location is written to. The usage counter can then be decreased by another mechanism called the free gate whenever a location is read from. More precisely, assuming a single read head for simplicity, a memory retention vector ψ_t is defined from the last read weights $\mathbf{q}_{t-1}$ and the free gate f_t as follows:

$$\psi_t = \mathbf{1} - f_t\mathbf{q}_{t-1},$$

the usage vector is then defined as:

$$\mathbf{u}_t = (\mathbf{u}_{t-1} + \mathbf{w}_{t-1} - \mathbf{u}_{t-1} \odot \mathbf{w}_{t-1}) \odot \psi_t.$$

Each write to a location increases its usage until it is explicitly decreased by the free gate after being read. This allows the system to learn when a value it has just read is no longer likely to be of further use and subsequently release it. An allocation weight vector $\mathbf{a}_t$ is then determined from the usage vector by sorting the elements in ascending order of usage. A given item's allocation is the product of one minus its own usage with the usage of all the items of larger usage. This gives high allocation to items with low usage, as well as rapidly dropping off the allocation of other items when items with very low usage are available. Similar to NTM, another learned gate is used to interpolate between the content addressable and allocation based components of the write mechanism. A location may be updated either because its allocation is high, or because the system explicitly chooses to update it based on its content. Contrary to OPGOR, both NTM and DNC are trained entirely by backpropagation through time, making them computationally intensive and not readily applicable to the online RL case.

More directly related to the present thesis is the work of Oh, Chockalingam, Singh, and Lee (2016). They experiment with architectures using a combination of key-value memory and a RNN. They test on a number of RL environments in the Minecraft domain. The memory saves learned keys and values corresponding to the last N observations for some integer N, thus it is inherently limited in temporal extent but does not require any mechanism for information triage.

More recently Wayne et al. (2018) introduced MERLIN, a more sophisticated external memory based architecture for RL, along with a detailed study of its performance and behavior over a wide variety of problems. In their approach, the items stored in memory, which they refer to as state variables, are trained to function as a world model. They utilize recalled state variables in two places, first to update a distribution from which the present state variable is drawn, and second to condition the policy. The latter is similar to how we use the recalled state variables in this thesis. To decide which state variables to maintain in memory, MERLIN uses a simple usage indicator, monitoring the cumulative magnitude of reads made to each state variable in memory, and always replacing the one with the smallest usage with the current state variable.

In MERLIN, the processes of learning useful state variables and conditioning the policy are decoupled by stopping gradients, such that the policy learning does not influence the learned state variables in any way. This is done to emphasize that training end to end to optimize return is unnecessary and often inefficient. Their approach treats the encoding and storage of sensory data as a separate problem from policy optimization, with the policy acting only as a consumer. This view fits quite well into the approach of the present thesis, where we optimize memory storage based on what is useful to the policy, but do not backpropagate policy gradient errors through the state representation.

In addition to an external memory of size 1350, MERLIN utilizes recurrent state updates with a relatively short truncation length of 20. Contrary to the present thesis, which uses a single online actor, MERLIN employs 192 parallel actors to train a single shared learner.

To conclude this section, we note our own prior work on applying reservoir sampling to manage an external memory for RL (Young, Sutton, & Yang, 2018). That work introduced a similar procedure to the one outlined here but used a novel, and somewhat complex reservoir sampling procedure. Here we simplify the approach by using an established reservoir sampling technique from the literature (note that the two procedures are identical in the single-state memory case). In addition we extend the method to be usable with eligibility traces, along with multiple read heads and soft queries. Finally we provide a solution to a limitation of that work, and our first version of OPGOR, which causes them to require a very accurate value estimate to perform well. We also perform much more extensive evaluation, testing OPGOR on three different problems against a number of alternative selective memory strategies.

There are many other examples of deep learning systems with integrated external memory (e.g., Gulcehre, Chandar, and Bengio, 2017; Gulcehre, Chandar, Cho, and Bengio, 2018; Joulin and Mikolov, 2015; Kaiser, Nachum, Roy, and Bengio, 2017; Sukhbaatar, Szlam, Weston, and Fergus, 2015; Zaremba and Sutskever, 2015), as well as many examples of work applying deep RL to non-Markov tasks (e.g., Bakker, Zhumatiy, Gruener, and Schmidhuber, 2003; Hausknecht and Stone, 2015; Wierstra, Förster, Peters, and Schmidhuber, 2010; Zhang, Levine, McCarthy, Finn, and Abbeel, 2016).

Chapter 3

Advantage Actor-Critic with External Memory

We have reviewed the reinforcement learning framework, the solution method of actor-critic, and the additional challenges posed by partial observability. In addition, we discussed the framework of external memory systems, which provide one way of addressing the challenges of partial observability, by giving an agent access to a fixed number of slots to write and read pertinent information. Our main contribution, OPGOR, will operate within the framework of external memory, and provides one answer to the question of how to decide what is worth storing in such a memory. We will now describe in detail the particular architecture in which we will frame our derivation of OPGOR in Chapter 5. This architecture is a variant of actor-critic, augmented with an external memory. In our experiments in Chapter 6, we will also investigate a number of alternative selective memory techniques, each of which will operate within a variant of the architecture presented in this chapter.

We build our primary architecture on the advantage actor-critic architecture introduced by Mnih et al. (2016), which consists of a *value network* and *policy network*. In addition to the value and policy networks of the advantage actor-critic architecture, we include an external memory consisting of a sequence of n pairs $((\mathcal{M}_t(0), \mathcal{W}_t(0)), ..., (\mathcal{M}_t(n-1), \mathcal{W}_t(n-1)))$ of vectors $\mathcal{M}_t(i)$ with associated scalar importance weight $\mathcal{W}_t(i)$. As an additional notational convenience we will use $\mathcal{M}_t$ to refer to the sequence of vectors $(\mathcal{M}_t(0), ..., \mathcal{M}_t(n-1))$ alone. Following Wayne et al. (2018), we refer to the vectors stored in memory as state variables, taking care to distinguish them from the Markov state S_t specified by the RL problem. The state variable generated at time t will be a vector of fixed size, denoted by ϕ_t. While, in this thesis, ϕ_t is generated directly by the environment, in general we would want to use some learned state representation. The fact that we use a ϕ_t generated directly by the environment is for simplicity in this initial proof of concept and not something we wish to impose as a restriction. For our purposes, ϕ_t represents the immediate information an agent has on hand at a particular time, whether constructed by a recurrent state update, provided by the environment directly, or otherwise.

The importance weights $\mathcal{W}_t(i)$ stored with each state variable in memory are generated by the *write network* $w(\phi_t, \theta)$, which takes the current state variable as input and outputs a single real value. The importance weights will be used in a reservoir sampling procedure, and adjusted via OPGOR. The details of the reservoir sampling algorithm in which these weights are used will be described in Chapter 4, for now simply note that these weights determine how likely a state variable is to be written to, and subsequently retained in, the external memory.

The *query network* $q(\phi_t, \theta)$ outputs a vector of size equal to the size of ϕ_t with tanh activation. At each time-step, a single state variable $m_t = \mathcal{M}_t(i)$ is drawn from the memory to condition the policy $\pi(A_t|\phi_t, m_t, \theta)$ according to the probability distribution:

$$Q(\mathcal{M}_t(i)|\phi_t, \mathcal{M}_t, \theta) = \exp\left(\langle q(\phi_t, \theta)|\mathcal{M}_t(i)\rangle \beta\right) \Bigg/ \sum_{j=0}^{n-1} \exp\left(\langle q(\phi_t, \theta)|\mathcal{M}_t(j)\rangle \beta\right), \tag{3.1}$$

where β is a positive learnable parameter representing the inverse temperature of the softmax. The parameter β is intended to allow the agent to make its queries more or less precise as appropriate as learning progresses. The state variable m_t selected from memory is given as input to the policy network along with the current state variable ϕ_t, both of which condition the resulting policy. An illustration of this architecture is shown in Figure 3.1.

Recall from Section 2.2 that partial observability can be considered a limitation of our function approximation, while the full history is taken to serve as the Markov state S_t. The particular architecture outlined here can be viewed as a particular choice about this limitation where there is a significant asymmetry between the information provided to the value function approximation and policy. In particular, the policy is conditioned on a past state variable, while the value is limited to only the present state variable ϕ_t.

In addition to the single read head architecture described above, we will experiment with using multiple read heads. In this case the query network will generate multiple queries of size matching the state variable and the memory will output one state variable for each query generated. The policy in this case will be conditioned on the concatenation of the current state variable with each of the recalled state variables. This could in principle allow each head to specialize to recall a particular type of relevant information as is observed in the navigation experiments of Wayne et al. (2018).

In all our experiments we will use single hidden layer ANNs for the value, query and write networks, and two hidden layers for the policy network. This choice is based on the intuition that the policy network must aggregate information from the current state variable as well as the state variable recalled from memory in order to select a good action. This is likely to be a more complex function than what is necessary for the other three networks. For simplicity, there will be no parameter sharing between networks. Unless otherwise specified, hidden layers will be of size 32 and use *sigmoid linear unit* (SiLU) activations. The SiLU activation is described by Elfwing, Uchibe, and Doya (2018), where it was found to be beneficial when training ANNs for online RL with eligibility traces, as we will do in this thesis.

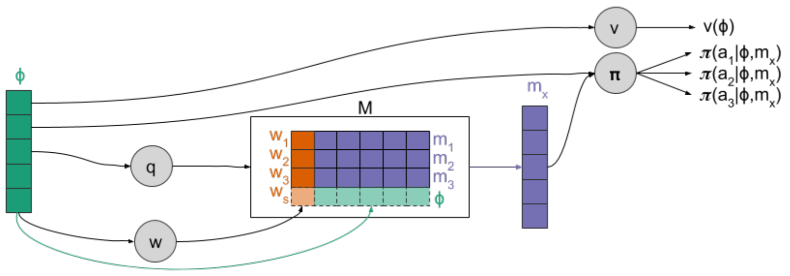


Figure 3.1: Advantage actor-critic with external memory architecture. Each grey circle represents an ANN module. The state variable (ϕ) is provided as input to the query (q), write (w), value (v) and policy (π) networks at each time-step. The query network outputs a vector, equal in length to the state variable, which is used (via equation 3.1) to choose a past state variable from the memory (m_1 ,m_2 or m_3 in the above diagram) to condition the policy. The write network assigns a weight to each new state variable, determining how likely it is to stay in memory. The policy network assigns probabilities to each action conditioned on current state variable and recalled state variable. The value network estimates expected return (value) from the current state variable.

Chapter 4

Reservoir Sampling

In this chapter, we provide the relevant background on reservoir sampling, which we will use, in association with OPGOR, to manage our agent's external memory. Specifically, the importance weight output by the write network, as discussed in Chapter 3, will be used in a reservoir sampling algorithm. Given the importance weight associated with each state variable in memory and the importance weight assigned to the new state variable ϕ_t, the reservoir sampling algorithm stochastically decides whether to add ϕ_t and if so which state variable to remove. In turn, our main contribution, OPGOR will be used to tune the weights generated by the write network to preferentially retain state variables that help to yield higher return.

Reservoir sampling refers to a class of algorithms for sampling from a distribution over n-subsets of items from a larger set streamed one item at a time. The goal is to ensure, through specific add and drop probabilities, that the n items in the reservoir at each time-step correspond to a sample with some desired statistical properties over n-subsets of all observed items. The particular reservoir sampling technique we apply gives closed form, differentiable inclusion probabilities for each state variable in terms of an associated weight. In our case, this weight will be generated by an artificial neural network (ANN) for each observed state variable. This allows us to apply the techniques of policy gradient to improve the weights assigned to states in memory with respect to the resulting expected return.

One of the simplest examples of a reservoir sampling algorithm maintains a reservoir of n items drawn uniformly at random from a stream observed one item at a time (Vitter, 1985). To achieve this, the first n items are added to the reservoir automatically, after which the algorithm proceeds as follows at each time-step t:

- Observe a new item ϕ_t
- Choose whether to add the item to the current reservoir with probability n/t
- If we do choose to add it, replace an item from the current reservoir uniformly at random

To see that this correctly produces uniform inclusion probabilities, first note that the most recently observed item is included with precisely probability n/t. Assume towards a proof by induction that all other items in the reservoir are included with probability $n/(t-1)$ prior to observing ϕ_t. Now, since ϕ_t is added with $P = n/t$, and if it is added each item has equal probability of being replaced, the inclusion probability of each item in the reservoir after the new observation is:

$$\begin{aligned} P &= \frac{n}{t-1} \cdot \left(1 - \frac{n}{t} + \frac{n}{t} \cdot \left(1 - \frac{1}{n}\right)\right) \\ &= \frac{n}{t-1} \cdot \left(\frac{t-n}{t} + \frac{n-1}{t}\right) \\ &= \frac{n}{t}, \end{aligned}$$

which is indeed equal to the inclusion probability of ϕ_t. As the base case for the inductive proof simply note that at time n the inclusion probability of each item is 1 which is indeed n/t at that time.

There are various ways to extend reservoir sampling to the more complex case of unequal probabilities, for example probabilities proportional to a weight w_t associated with each ϕ_t. In this thesis we will focus on one such method, first presented by Chao (1982), which we will now describe. In this case at each time-step we observe an item ϕ_t along with an associated weight w_t. We want the inclusion probability of each ϕ_t in the reservoir to be linearly proportional to the associated w_t. However, if we allow arbitrary positive weights this may necessitate some probabilities greater than one, so we have to refine this desiderata slightly. Precisely, define the set of inadmissible indices:

$$\Omega_t = \underset{\omega \subset [0,..,t-1]}{\arg\min} |\omega| \text{ s.t. } \forall i \in \{0,..,t-1\} \setminus \omega, \ \frac{(n-|\omega|)w_i}{\sum\limits_{j \in \{0,..,t\} \setminus \omega} w_j} < 1. \tag{4.1}$$

In other words, the smallest set of indices such that when we fix the associated item in the reservoir, the inclusion probabilities of the remaining items given the remaining reservoir space are all valid probabilities less than one. Note that such a set is easily constructed by recursively removing the largest weight item until the inequality holds for the next largest. Now define $\mathcal{M}_t$ as the set of items in the reservoir prior to observing item ϕ_t. We will assert the following probabilities for each item indexed from 0 to $t-1$ being present in the reservoir at time t:

$$P(\phi_i \in \mathcal{M}_t) = \begin{cases} 1 & \text{if } i \in \Omega_t \\ \frac{(n-|\Omega_t|)w_i}{\sum\limits_{j \notin \Omega_t} w_j} & \text{if } i \notin \Omega_t \end{cases}. \tag{4.2}$$

All items whose probability would be greater than one are included with certainty, while the remaining items are included with probability proportional to their weight. With the desired inclusion probabilities in place, it remains to describe how to formulate incremental replacement rules to give rise to these probabilities. As in the equal probability case, the first step will be to determine whether or not to add ϕ_t to the reservoir. There is little choice in this step, to achieve the desired inclusion probability for the new item we must add it with probability $P(\phi_t \in \mathcal{M}_{t+1})$ as defined in equation 4.2. After that, we separately handle the items which were included with certainty at time $t-1$ and those which were not. The full procedure goes as follows:

- Observe a new item ϕ_t
- Choose whether to add the item to the current reservoir with probability $P(\phi_t \in \mathcal{M}_{t+1})$

- If we choose not to add it, stop here and maintain the same reservoir, otherwise continue to the next step
- From all items ϕ_i with index in Ω_t choose one, or none, to drop with probability $\frac{1-P(\phi_i \in \mathcal{M}_{t+1})}{P(\phi_t \in \mathcal{M}_{t+1})}$, note that this may be 0 if w_i remains inadmissible
- If no item is dropped in the previous step, drop one of the other items uniformly at random

From now on we will refer to this procedure as *Chao sampling*, after the author who originally described it. We will now show how these rules give rise to the correct inclusion probabilities for all items. Note that though Chao gives a sketch of how these rules produce the desired probabilities, the detailed breakdown provided here is our own. We will break the reasoning down into four cases as follows:

- ϕ_t
- $\phi_i, i \in \Omega_{t+1}$
- $\phi_i, i \in \Omega_t \setminus \Omega_{t+1}$
- $\phi_i, i \in \{0, ..., t-1\} \setminus \Omega_t$

First note that we trivially assert $P(\phi_t \in \mathcal{M}_{t+1})$ obeys equation 4.2 by choosing to add it with the correct probability. Similarly, for items with index in Ω_{t+1} we simply omit them from consideration when choosing an item to swap, thus they are included with certainty as desired. For items with index in $\Omega_t \setminus \Omega_{t+1}$ we note that they were included with probability one in the last time-step. They will be included in the next time-step if either we choose not to add ϕ_t, or choose to add ϕ_t but swap a different item out. Thus the probability of inclusion for an item ϕ_i with $i \in \Omega_t \setminus \Omega_{t+1}$ can be computed as follows:

$$\begin{aligned} P'(\phi_i \in \mathcal{M}_{t+1}) &= 1 - P(\phi_t \in \mathcal{M}_{t+1}) + P(\phi_t \in \mathcal{M}_{t+1}) \left(1 - \frac{1-P(\phi_i \in \mathcal{M}_{t+1})}{P(\phi_t \in \mathcal{M}_{t+1})}\right) \\ &= P(\phi_i \in \mathcal{M}_{t+1}). \end{aligned}$$

Which is the probability we intended. Note that we have assumed that $\frac{1-P(\phi_i \in \mathcal{M}_{t+1})}{P(\phi_t \in \mathcal{M}_{t+1})}$ for each $i \in \Omega_t$ sum to a number less than one, such that they represent valid probabilities and we can actually run the procedure as indicated. This follows from the fact that that all $i \in \Omega_t \setminus \Omega_{t+1}$ are inadmissible at time t. If we assume w_t is admissible then since Ω_{t+1} is a subset of Ω_t we have by definition, for all $i \in \Omega_t \setminus \Omega_{t+1}$:

$$\begin{aligned}
&(n - |\Omega_{t+1}|)w_i > \sum_{j \in \{0,\ldots,t-1\} \setminus \Omega_{t+1}} w_j \\
\implies &\frac{(n - |\Omega_{t+1}|)}{|\Omega_t \setminus \Omega_{t+1}|} \sum_{i \in \Omega_t \setminus \Omega_{t+1}} w_i > \sum_{j \in \{0,\ldots,t-1\} \setminus \Omega_{t+1}} w_j \\
\implies &\frac{(n - |\Omega_{t+1}|)}{|\Omega_t \setminus \Omega_{t+1}|} \sum_{i \in \Omega_t \setminus \Omega_{t+1}} w_i + w_t > \sum_{j \in \{0,\ldots,t\} \setminus \Omega_{t+1}} w_j \\
\implies &\frac{(n - |\Omega_{t+1}|)}{|\Omega_t \setminus \Omega_{t+1}|} \sum_{i \in \Omega_t \setminus \Omega_{t+1}} w_i + \frac{(n - |\Omega_{t+1}|)}{|\Omega_t \setminus \Omega_{t+1}|} w_t > \sum_{j \in \{0,\ldots,t\} \setminus \Omega_{t+1}} w_j \\
\implies &(n - |\Omega_{t+1}|) \sum_{i \in \Omega_t \setminus \Omega_{t+1}} w_i + (n - |\Omega_{t+1}|)w_t > |\Omega_t \setminus \Omega_{t+1}| \sum_{j \in \{0,\ldots,t\} \setminus \Omega_{t+1}} w_j \\
\implies &\sum_{i \in \Omega_t \setminus \Omega_{t+1}} \left(\sum_{j \in \{0,\ldots,t\} \setminus \Omega_{t+1}} w_j - (n - |\Omega_{t+1}|)w_i \right) < (n - |\Omega_{t+1}|)w_t \\
\implies &\sum_{i \in \Omega_t \setminus \Omega_{t+1}} \left(1 - \frac{(n - |\Omega_{t+1}|)w_i}{\sum_{j \in \{0,\ldots,t\} \setminus \Omega_{t+1}} w_j} \right) < \frac{(n - |\Omega_{t+1}|)w_t}{\sum_{j \in \{0,\ldots,t\} \setminus \Omega_{t+1}} w_j} \\
\implies &\sum_{i \in \Omega_t \setminus \Omega_{t+1}} \frac{1 - P(\phi_i \in \mathcal{M}_{t+1})}{P(\phi_t \in \mathcal{M}_{t+1})} < 1.
\end{aligned}$$

In the forth line we have used $n - |\Omega_{t+1}| > |\Omega_t| - |\Omega_{t+1}| = |\Omega_t \setminus \Omega_{t+1}|$. If instead we assume w_t is inadmissible, note that we have $|\Omega_t \setminus \Omega_{t+1}| = |\Omega_t| - |\Omega_{t+1}| + 1$ as Ω_{t+1} contains t while Ω_t does not, but $\Omega_{t+1} \setminus \{t\} \subset \Omega_t$. In this case we can reach the same conclusion by slightly different reasoning:

$$
\begin{aligned}
&(n - |\Omega_{t+1}| + 1)w_i > \sum_{j \in \{0,...,t-1\} \setminus (\Omega_{t+1} \setminus \{t\})} w_j \\
&\implies (n - |\Omega_{t+1}| + 1)w_i > \sum_{j \in \{0,...,t\} \setminus \Omega_{t+1}} w_j \\
&\implies \frac{(n - |\Omega_{t+1}| + 1)}{|\Omega_t \setminus \Omega_{t+1}|} \sum_{i \in \Omega_t \setminus \Omega_{t+1}} w_i > \sum_{j \in \{0,...,t\} \setminus \Omega_{t+1}} w_j \\
&\implies (n - |\Omega_{t+1}| + 1) \sum_{i \in \Omega_t \setminus \Omega_{t+1}} w_i > |\Omega_t \setminus \Omega_{t+1}| \sum_{j \in \{0,...,t\} \setminus \Omega_{t+1}} w_j \\
&\implies (n - |\Omega_{t+1}|) \sum_{i \in \Omega_t \setminus \Omega_{t+1}} w_i > |\Omega_t \setminus \Omega_{t+1}| \sum_{j \in \{0,...,t\} \setminus \Omega_{t+1}} w_j - \sum_{i \in \Omega_t \setminus \Omega_{t+1}} w_i \\
&\implies (n - |\Omega_{t+1}|) \sum_{i \in \Omega_t \setminus \Omega_{t+1}} w_i > |\Omega_t \setminus \Omega_{t+1} - 1| \sum_{j \in \{0,...,t\} \setminus \Omega_{t+1}} w_j \\
&\implies \sum_{i \in \Omega_t \setminus \Omega_{t+1}} \left(\sum_{j \in \{0,...,t\} \setminus \Omega_{t+1}} w_j - (n - |\Omega_{t+1}|)w_i \right) < \sum_{j \in \{0,...,t\} \setminus \Omega_{t+1}} w_j \\
&\implies \sum_{i \in \Omega_t \setminus \Omega_{t+1}} \left(1 - \frac{(n - |\Omega_{t+1}|)w_i}{\sum_{j \in \{0,...,t\} \setminus \Omega_{t+1}} w_j} \right) < 1 \\
&\implies \sum_{i \in \Omega_t \setminus \Omega_{t+1}} \frac{1 - P(\phi_i \in \mathcal{M}_{t+1})}{P(\phi_t \in \mathcal{M}_{t+1})} < 1.
\end{aligned}
$$

Thus, in either case the probabilities are indeed valid and we can sample from $\Omega_t \setminus \Omega_{t+1}$ as previously specified. Lastly we have the items $\phi_i, i \in \{0, ..., t-1\} \setminus \Omega_t$. For this case assume toward a proof by induction that the inclusion probability at time t, $P'(\phi_i \in \mathcal{M}_t)$, is given by $P(\phi_i \in \mathcal{M}_t)$ as specified in equation 4.2. The probability that ϕ_i is included at the following time-step, $P'(\phi_i \in \mathcal{M}_{t+1})$, can then be computed as follows:

$$
\begin{aligned}
P'(\phi_i \in \mathcal{M}_{t+1}) = P(\phi_i \in \mathcal{M}_t) \Bigg(& 1 - P(\phi_t \in \mathcal{M}_{t+1}) \\
& + P(\phi_t \in \mathcal{M}_{t+1}) \left(\sum_{j \in \Omega_t \setminus \Omega_{t+1}} \frac{1 - P(\phi_j \in \mathcal{M}_{t+1})}{P(\phi_t \in \mathcal{M}_{t+1})} \right) \\
& + P(\phi_t \in \mathcal{M}_{t+1}) \left(1 - \sum_{j \in \Omega_t \setminus \Omega_{t+1}} \frac{1 - P(\phi_j \in \mathcal{M}_{t+1})}{P(\phi_t \in \mathcal{M}_{t+1})} \right) \\
& \qquad \frac{n - |\Omega_t| - 1}{n - |\Omega_t|} \Bigg).
\end{aligned}
$$

That is, the probability that ϕ_i was included at time t times the probability that it is not removed at that time. We have broken down the probability of i not being removed as the probability no swap occurs, the probability an element of $\Omega_t \setminus \Omega_{t+1}$ is swapped, and the probability an element of $\{0, ..., t-1\} \setminus \Omega_t$ other than i is removed. From this expression it is a matter of algebraic manipulation to obtain the desired probability:

$$\begin{aligned} P'(\phi_i \in \mathcal{M}_{t+1}) = {} & P(\phi_i \in \mathcal{M}_t)\Bigg(1 - P(\phi_t \in \mathcal{M}_{t+1}) + \sum_{j \in \Omega_t \setminus \Omega_{t+1}} (1 - P(\phi_j \in \mathcal{M}_{t+1})) \\ & + \Bigg(P(\phi_t \in \mathcal{M}_{t+1}) - \sum_{j \in \Omega_t \setminus \Omega_{t+1}} (1 - P(\phi_j \in \mathcal{M}_{t+1}))\Bigg) \\ & \frac{n - |\Omega_t| - 1}{n - |\Omega_t|}\Bigg) \\ = {} & P(\phi_i \in \mathcal{M}_t)\Bigg(\Bigg(\sum_{j \in \Omega_t \setminus \Omega_{t+1}} (1 - P(\phi_j \in \mathcal{M}_{t+1})) \\ & - P(\phi_t \in \mathcal{M}_{t+1})\Bigg)\frac{1}{n - |\Omega_t|} + 1\Bigg). \end{aligned}$$

From here we again consider two cases, either w_t is admissible, or it is not. If w_t is admissible we have:

$$\begin{aligned} P'(\phi_i \in \mathcal{M}_{t+1}) = {} & P(\phi_i \in \mathcal{M}_t)\Bigg(\frac{|\Omega_t| - |\Omega_{t+1}|}{n - |\Omega_t|} \\ & - \frac{n - |\Omega_{t+1}|}{n - |\Omega_t|}\frac{\sum\limits_{j \in \Omega_t \setminus \Omega_{t+1}} w_j + w_t}{\sum\limits_{j \in \{0,...,t\} \setminus \Omega_{t+1}} w_j} + 1\Bigg) \\ = {} & P(\phi_i \in \mathcal{M}_t)\frac{n - |\Omega_{t+1}|}{n - |\Omega_t|}\left(1 - \frac{\sum\limits_{j \in \Omega_t \setminus \Omega_{t+1}} w_j + w_t}{\sum\limits_{j \in \{0,...,t\} \setminus \Omega_{t+1}} w_j}\right), \end{aligned}$$

where we have used the fact that for admissible w_t, Ω_{t+1} is a subset of Ω_t, and thus $|\Omega_t \setminus \Omega_{t+1}| = |\Omega_t| - |\Omega_{t+1}|$. If on the other hand w_t is inadmissible we have $|\Omega_t \setminus \Omega_{t+1}| = |\Omega_t| - |\Omega_{t+1}| + 1$ as noted previously. In this case we instead get:

$$\begin{aligned}
P'(\phi_i \in \mathcal{M}_{t+1}) &= P(\phi_i \in \mathcal{M}_t)\left(\frac{|\Omega_t| - |\Omega_{t+1}|}{n - |\Omega_t|} - \frac{n - |\Omega_{t+1}|}{n - |\Omega_t|}\frac{\sum\limits_{j \in \Omega_t \setminus \Omega_{t+1}} w_j}{\sum\limits_{j \in \{0,\dots,t\} \setminus \Omega_{t+1}} w_j} + 1\right) \\
&= P(\phi_i \in \mathcal{M}_t)\frac{n - |\Omega_{t+1}|}{n - |\Omega_t|}\left(1 - \frac{\sum\limits_{j \in \Omega_t \setminus \Omega_{t+1}} w_j}{\sum\limits_{j \in \{0,\dots,t\} \setminus \Omega_{t+1}} w_j}\right).
\end{aligned}$$

Note that this differs by the exclusion of w_t from the numerator of the second term. In either case we can simplify to get:

$$\begin{aligned}
P'(\phi_i \in \mathcal{M}_{t+1}) &= \frac{(n - |\Omega_t|)w_i}{\sum\limits_{j \in \{0,\dots,t-1\} \setminus \Omega_t} w_j}\left(\frac{(n - |\Omega_{t+1}|)\sum\limits_{j \in \{0,\dots,t-1\} \setminus \Omega_t} w_j}{(n - |\Omega_t|)\sum\limits_{j \in \{0,\dots,t\} \setminus \Omega_{t+1}} w_j}\right) \\
&= \frac{(n - |\Omega_{t+1}|)w_i}{\sum\limits_{j \in \{0,\dots,t\} \setminus \Omega_t} w_j} \\
&= P(\phi_i \in \mathcal{M}_{t+1}).
\end{aligned}$$

Thus, again, we recover the desired probability. To complete the inductive proof simply note that we have already demonstrated that when an item is either first added, or first enters the admissible set it's probability will obey equation 4.2. We have demonstrated that the probability will continue to obey equation 4.2 on all subsequent time-steps.

Note that in the absence of inadmissible items this procedure takes $O(1)$ time each time-step; only requiring that we compute the inclusion probability of the new item, add it to the denominator, and randomly select its position if we decide to add it to memory. Inadmissible items require extra computation as it is necessary to check at each time-step whether each remains inadmissible. Assuming inadmissible items are maintained in ascending order of the associated weight, this can be done by checking for each previously inadmissible item i whether $(n - |\Omega_t|)w_i < \sum\limits_{j \notin |\Omega_t|} w_j$, and if so removing them. If inadmissible items are maintained in a linked list, newly admissible items can be removed and ϕ_t added if necessary in a single sweep, requiring at most $O(|\Omega_t|)$ time. Thus the total cost of writing to memory is at most $O(n)$ in the infrequent case where almost all items in memory transition from inadmissible to admissible. Generally inadmissible items will be rare, so on average the compute time should be closer to constant every time a new item is added.

In the special case of a single item memory, the Chao sampling algorithm becomes very simple. In this case we need only choose whether to add each newly observed item according to:

$$P(\mathcal{M}_{t+1} = \{\phi_t\}) = \frac{w_t}{\sum\limits_{i=0}^{t} w_i}.$$

In the single item case there is no need to worry about inadmissibility, if a new item is added, the stored item must be dropped. A simple calculation analogous to the unweighted case reviewed above will show the inclusion probabilities for all $j \in [0, t]$ will then obey:

$$P(\mathcal{M}_{t+1} = \{\phi_j\}) = \frac{w_j}{\sum\limits_{i=0}^{t} w_i}.$$

To conclude this chapter we briefly mention another weighted reservoir sampling algorithm first described by Efraimidis and Spirakis (2006). Instead of enforcing that inclusion probabilities be directly proportional to weights this algorithm enforces that the reservoir represents a sequential weighted random sample without replacement from the full stream. The manner in which it does this is quite interesting in its own right and it may also be useful in the context of RL with external memory. We do not explore this further here.

Chapter 5

Online Policy Gradient Over a Reservoir (OPGOR)

Having described our particular external memory architecture, and Chao sampling, which will manage the external memory, we are ready to describe our main contribution, online policy gradient over a reservoir (OPGOR). OPGOR is an online algorithm for training a *write network* which generates weights for use in Chao sampling, as described in Chapter 4. OPGOR trains the write network to emphasize state variables that lead to high expected return when later recalled. OPGOR does this by treating the state variables stored in memory like actions, where the action space is the entire history, and applying policy gradient to increase the probability of state variables that result in positive advantage. To make it possible to do this online, we employ a few mathematical tricks and approximations.

We will also describe, in Section 5.6, how an external memory, trained with OPGOR can be easily combined with the other components of an online RL agent, trained using standard online RL algorithms. In particular, we use the architecture presented in Chapter 3, and train using the $AC(\lambda)$ algorithm described in Section 2.1. All components of our proposed agent are trained online, performing one update per time-step with no experience replay or multiple parallel actors.

The goal of OPGOR is to apply policy gradient to the process of selecting which state variables from the history are stored in memory. Towards this, we associate a parameterized *importance weight* $w(\phi_i, \theta_w)$, as introduced in

Chapter 3, with each observed state variable ϕ_i. We refer to the function w that generates the importance weights as the write network. We wish to satisfy three main desiderata with respect to the importance weights $w(\phi_i, \theta_w)$:

1. We want the importance weights $w(\phi_i, \theta_w)$ to be such that the probability of a particular state variable ϕ_i being present in memory at any given future time $t > i$ is proportional to the exponential of the importance weight $\exp(w(\phi_i, \theta_w))$. These importance weights thus form something like the logits of a softmax distribution, as is often used to parameterize the policy of AC agents with discrete action spaces.

2. We want to obtain estimates of the gradient of the return with respect to the parameters θ_w of the write network, such that we can perform stochastic gradient ascent on the return with respect to those parameters.

3. We want to construct these gradient estimates online, without maintaining significantly more information than the fixed set of state variables stored in memory.

A naive approach to the first desiderata could involve storing the entire history of state variables up to time t and then drawing a subset from that history to fill our memory according to some weighted probability distribution. This would obviously be extremely memory intensive as the history grows, and is thus not appropriate for an online method, which violates the third desiderata. This is exactly what we avoid through the use of Chao sampling to manage a fixed-size memory online, according to the importance weight associated with each state variable.

Given that Chao sampling solves the problem of drawing a weighted sample from the full history online, there is an additional complication from the second desiderata, estimating the gradient of the return with respect to the parameters of the write network. The denominator of the inclusion probability of a particular state variable will be the sum of the importance weights assigned to all state variables. To naively compute the gradient of this denominator, for use in policy gradient, we would again have to maintain the full history to compute the gradient of the weight assigned to each state variable.

We will begin by showing that, for a single-state memory, if we have an accurate critic to act as a baseline, we can produce an unbiased estimate of the gradient of the expected return with respect to the parameters of the write network using only the numerator of the inclusion probability. OPGOR uses this estimate to perform approximate gradient ascent in the expected return. Here, by accurate critic, we mean we have a value function approximation which is a good estimate of the history conditional expected return. This is possible because the denominator is independent of the specific state variable stored in memory, hence we can pull it out of the sum, similar to how one can subtract an action independent baseline in the REINFORCE algorithm of Williams (1992). In Section 5.1 we will describe how this estimator is constructed in detail. In Section 5.2 we will also describe how we extend this estimator to the case of a multiple-state memory, for which we employ a semi-gradient approximation.

In deriving OPGOR we assume, as discussed above, that we have a value function and associated state variable on hand which provide an accurate estimate of $E_t[G_t]$. In Section 6.5 we will show in a simple case how serious issues can result when this assumption fails to hold. In more realistic problems, an approximation to such a value function could be obtained, for example, by using a recurrent network, such as a GRU. Nonetheless, since avoiding the use of backpropagation over long time horizons is one motivation for OPGOR, this is a somewhat limiting assumption and would be useful to relax. Motivated by this issue, in Section 5.7, we derive a modified version of OPGOR which mitigates the reliance on an accurate critic. This new version of OPGOR maintains an online sample of the contribution of the denominator of the inclusion probability to the policy gradient, hence we call this version *OPGOR with Denominator Sampling* (OPGOR-DS).

For brevity, in this chapter, we will use the notation $E_t[x]$ to denote $E[x|\phi_0, ..., \phi_t]$, the expectation conditioned on the entire history of state variables up to time t. Similarly, $P_t(x)$ will represent probability conditioned on the entire history of state variables. All expectations and probabilities are assumed to be with respect to the current policy, query and write network. We will use $\mathcal{A}$ to represent the set of available actions and A_t the action selected at time t.

Formally, we will consider the episodic case and take our optimization objective to be the undiscounted expected return from the start state:

$$\mathcal{J} = E_0[G_0] = E[G_0|\phi_0] = E[R_1 + R_2 + ...|\phi_0]. \tag{5.1}$$

We will nonetheless estimate advantages using a non-unit discount factor γ which has a dual role. First, it serves to reduce variance due to high uncertainty in distant future rewards, in exchange for biasing the value estimate towards short-term rewards. See, for example, the work of Schulman et al. (2015) for a discussion of the interpretation of γ as a parameter of the algorithm as opposed to a property of the problem. Additionally, γ will act as a surrogate for the time limits which will be imposed in many of the problems used in our experiments. Due to the discount factor the agent will be encouraged to obtain rewards quickly without needing to explicitly encode the remaining steps in the state representation.

OPGOR could be applied to the continuing case by geometrically discounting the weights associated with items in memory at each time-step, such that more recent items would be favored. This would prevent the probability of adding new items to memory from decaying uniformly to zero, which would be the case otherwise. The possibility of adding a recency bias to a reservoir sampling algorithm is discussed by Aggarwal (2006). This is, however, beyond the scope of this thesis.

5.1 OPGOR for a Single-State Memory

To introduce the idea behind OPGOR, we first derive the algorithm for the case when our external memory can store just one state variable, and thus there is no need to query. We will use m_t to represents the state variable in memory at time t and thus, in the single-state memory case, the state variable read from memory by the agent at time t. Assume the stored state variable is drawn from a distribution parameterized by a set of weights $\{w_i | i \in \{0, ..., t-1\}\}$. Each $w_i = w(\phi_i, \theta_w)$ is associated with a state variable ϕ_i by the write network w, parameterized by θ_w, when the associated state is visited. In the single-state memory case we parameterize the distribution as follows:

$$P_t(m_t = \phi_i) = \exp(w_i) \Bigg/ \sum_{j=0}^{t-1} \exp(w_j) \ , \tag{5.2}$$

which is the probability resulting from the single item case of Chao sampling outlined in Chapter 4, except with the weights exponentiated. Exponentiated importance weights are used because they give a simpler form for the gradient estimate, and enforce positive probabilities with arbitrary real weights. We can then write the expectation of the return $G_t = R_{t+1} + R_{t+2} + ...$ as follows:

$$E_t[G_t] = \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t | A_t = a]. \tag{5.3}$$

We wish to perform stochastic gradient ascent on this objective with respect to the parameters θ_w of the write network. OPGOR employs an estimate of this gradient using an AC method that, assuming no discounting, is unbiased if the critic can accurately estimate $E_t[G_t]$. Additionally, this estimate will involve computing the gradient only of the w_i associated with the index i such that $m_t = \phi_i$. This means we will only have to compute gradients for the weight associated with the stored state variable. This is crucial to allow OPGOR to run online, as otherwise we would need to store every generated state variable to compute the gradient estimate. We now compute the gradient of the expected return at some time t, as in equation 5.3, with respect to the parameters of the write network θ_w, given the distribution over state variables in memory is given by equation 5.2.

$$\begin{aligned}\frac{\partial}{\partial \theta_w} E_t[G_t] = \sum_{k=0}^{t-1} \Bigg(&\frac{\partial P_t(m_t = \phi_k)}{\partial \theta_w} \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t | A_t = a] \\ &+ P_t(m_t = \phi_k) \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) \frac{\partial E_t[G_t | A_t = a]}{\partial \theta_w} \Bigg). \end{aligned} \tag{5.4}$$

Note that π does not depend on the write network parameters θ_w. Working out the first term from the right hand side of equation 5.4:

$$\sum_{k=0}^{t-1} \frac{\partial P_t(m_t = \phi_k)}{\partial \theta_w} \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a]$$

$$= \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \frac{\partial \log(P_t(m_t = \phi_k))}{\partial \theta_w} \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a]$$

$$= \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \frac{\partial}{\partial \theta_w} \left(w_k - \log(\sum_{j=0}^{t-1} \exp(w_j)) \right) \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a]$$

$$= \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \left(\frac{\partial w_k}{\partial \theta_w} - \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w} \right) \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a]$$

$$= \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \frac{\partial w_k}{\partial \theta_w} \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a]$$
$$- \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w} \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a]$$

$$= \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \frac{\partial w_k}{\partial \theta_w} \left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a] - E_t[G_t] \right).$$

Working out the second term from the right hand side of equation 5.4:

$$\sum_{k=0}^{t-1} P_t(m_t = \phi_k) \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) \frac{\partial E_t[G_t|A_t = a]}{\partial \theta_w}$$

$$= \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) \left(\frac{\partial E_t[R_{t+1}|A_t = a]}{\partial \theta_w} + \frac{\partial E_t[G_{t+1}|A_t = a]}{\partial \theta_w} \right)$$

$$= E_t \left[\frac{\partial}{\partial \theta_w} E_{t+1}[G_{t+1}] \right].$$

Where we are able to drop $\frac{\partial E_t[R_{t+1}|A_t = a]}{\partial \theta_w}$ because the immediate reward is independent of the state variable in memory once conditioned on the action.

Thus we finally arrive at:

$$\frac{\partial}{\partial \theta_w} E_t[G_t] = \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w} \left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i, \theta) E_t[G_t | A_t = a] - E_t[G_t] \right) + E_t \left[\frac{\partial}{\partial \theta_w} E_{t+1}[G_{t+1}] \right]. \tag{5.5}$$

We will use a policy gradient approach, similar to REINFORCE (Williams, 1992), to estimate this gradient using an estimator ζ_t such that:

$$E_t[\zeta_t] \approx \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w} \left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i, \theta) E_t[G_t | A_t = a] - E_t[G_t] \right). \tag{5.6}$$

Expanding equation 5.5 recursively this will also give us:

$$E_0 \left[\sum_{t \geq 0} \zeta_t \right] \approx \frac{\partial E_0[G_0]}{\partial \theta_w}. \tag{5.7}$$

Consider the following gradient estimator:

$$\zeta_t = \delta_t \frac{\partial w(m_t, \theta_t)}{\partial \theta_w}, \tag{5.8}$$

which has expectation:

$$\begin{aligned}
E_t[\zeta_t] = & \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w} \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i, \theta) \\
& E_t[R_{t+1} + \gamma \hat{v}(\phi_{t+1}, \theta) - \hat{v}(\phi_t, \theta) | A_t = a] \\
= & \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w} \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i, \theta) \\
& \Big((E_t[R_{t+1} + \gamma G_{t+1} | A_t = a] - E_t[G_t]) \\
& + (\gamma E_t[\hat{v}(\phi_{t+1}, \theta) - G_{t+1} | A_t = a] + (E_t[G_t] - \hat{v}(\phi_t, \theta))) \Big) \\
\approx & \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w} \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i, \theta) \\
& (E_t[R_{t+1} + \gamma G_{t+1} | A_t = a] - E_t[G_t]) \\
= & \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w} \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i, \theta) \\
& \Big((E_t[G_t | A_t = a] - E_t[G_t]) \\
& + (\gamma - 1) E_t[G_{t+1} | A_t = a] \Big)
\end{aligned}$$

$$\approx \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w} \left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i, \theta) E_t[G_t|A_t = a] - E_t[G_t] \right).$$

Thus the specified estimator approximately satisfies equation 5.6 and by extension equation 5.7. The approximation is limited by the accuracy of our value function as well as biased towards short-term rewards by γ. In conventional policy gradient, subtracting the state value (e.g. using $\delta_t = R_{t+1} + \gamma\hat{v}(\phi_{t+1}, \theta) - \hat{v}(\phi_t, \theta)$ instead of $R_{t+1} + \gamma\hat{v}(\phi_{t+1}, \theta)$) is a means of variance reduction. Here it is critical to avoid computing gradients with respect to the denominator of equation 5.2, which allows OPGOR to run online while computing the gradient with respect to only the importance weight stored in memory. The reason this trick is possible is simply that the denominator of equation 5.2 does not depend on the remembered state variable m_t.

To apply OPGOR in a more realistic setting we would want ϕ_t to be a learned summary of the history, trained in part to make $\hat{v}(\phi_t, \theta)$ a good estimate of $E_t[G_t]$. See, for example, the predictive representation learning approach applied by Wayne et al. (2018). In this thesis, ϕ_t will be fixed for each problem and, if it is not possible to estimate $E_t[G_t]$ well from ϕ_t, the above approximation could be poor, and the performance of OPGOR may suffer. We will describe a way to mitigate this dependence in Section 5.7.

We are now ready to precisely describe OPGOR for the single-state memory case. OPGOR consists of using the gradient estimator ζ_t in a gradient ascent procedure to emphasize retention of state variables which improve the return. OPGOR uses online updating, meaning it updates the parameters of the write network at each time-step according to a single term in the above sum as follows:

$$\theta_{w,t+1} = \theta_{w,t} + \alpha\delta_t \frac{\partial w(m_t, \theta_t)}{\partial \theta_w}. \tag{5.9}$$

There are, however, a couple of subtle issues with this approach. First, the probability $P_t(m_t = \phi_i)$ used to sample ζ_t is sampled based on the values of w_i in memory, but we compute $\frac{\partial w_i}{\partial \theta_w}$ with respect to the parameters at time t. With online updating, this introduces a potential issue with stale gradients, which we suspect will vanish in the limit of sufficiently small learning rate, but leave further theoretical investigation to future work.

Another potential issue stems from the use of Chao sampling to manage the memory of our agent online, rather than sampling from the history according to equation 5.2 directly. This introduces a complication when the memory contents are used for control. In the single item case, Chao sampling ensures, for a fixed sequence of state variables and associated weights, that the marginal probability of a given state variable being present in memory at time t is given by equation 5.2. However, when we use the current item in memory to condition a RL control agent, it is no longer quite true that $P_t(m_t = \phi_i)$ will be given by equation 5.2. This is because the inclusion probabilities specified by Chao sampling implicitly assumes that the data stream is not influenced by the content of the memory at any particular time. If the agent is making decisions based on the contents of memory at each time this assumption is surely violated. In this case $P_t(m_t = \phi_i)$ will depend on not only the probability of sampling ϕ_i under a fixed history, but also the probability of certain items being in memory in past time-steps given the history. For example if a certain action a_τ in the trajectory is highly correlated with a certain item $m_\tau = \phi$ being in memory, then given that a_τ was selected at time τ it is much more likely that $m_\tau = \phi$. If $m_\tau = \phi$ it is also more likely that $m_t = \phi$, and less likely that m_t is any state variable observed before ϕ, due to the nature of the reservoir sampling approach. In turn the selected actions may influence latter observed state variables. The final result is that the state variable trajectory may contain information about memory contents at past times which is not captured by equation 5.2. We do not account for this interaction and simply treat the state variable trajectory as an arbitrary data stream for the purpose of managing the memory. The question of whether this can be theoretically justified or improved upon is left to future work.

5.2 OPGOR for a Multiple-State Memory

In the multiple-state case, the external memory will be managed by the full, multiple-item, version of Chao sampling, which we reviewed in Chapter 4.

Instead of just one state variable m_t stored in memory at each time we will now have a reservoir $\mathcal{M}_t$ of some fixed number of state variables n. We will continue to use m_t to refer to the particular state variable returned by the query from the network at time t. With this notation we can write the expected return as:

$$\begin{aligned}
E_t[G_t] &= \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \sum_{a\in\mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a] \\
&= \sum_{k=0}^{t-1} P_t(\phi_k \in \mathcal{M}_t) P_t(m_t = \phi_k|\phi_k \in \mathcal{M}_t) \\
&\qquad \sum_{a\in\mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a] \\
&= \sum_{k=0}^{t-1} P_t(\phi_k \in \mathcal{M}_t) E_t[Q(\phi_k|\phi_t, \mathcal{M}_t, \theta)|\phi_k \in \mathcal{M}_t] \\
&\qquad\qquad \sum_{a\in\mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a].
\end{aligned}$$

To compute a gradient estimate for the multiple-state case, we will make a simplifying approximation, using a form of semi-gradient, and take $\frac{\partial E_t[Q(\phi_k|\phi_t,\mathcal{M}_t,\theta)|\phi_k\in\mathcal{M}_t]}{\partial w_i} \approx 0$ for all k and i. Note that in reality since $Q(\phi_k|\phi_t, \mathcal{M}_t, \theta)$ is a function of not just ϕ_k but all the other items in memory at time t this derivative will actually be non-zero. Nonetheless, we suggest that propagating gradients through $P_t(m_t = \phi_k)$ while treating $E_t[Q(\phi_k|\phi_t, \mathcal{M}_t, \theta)|\phi_k \in \mathcal{M}_t]$ as constant with respect to the importance weights w_i will adequately capture the primary effect of the write process, while $E_t[Q(\phi_k|\phi_t, \mathcal{M}_t, \theta)|\phi_k \in \mathcal{M}_t]$ can be optimized with respect to the query process alone. Investigating the implications of this approximation is left to future work. With the above approximation in place we compute the gradient with respect to the write network parameters θ_w as follows:

$$\begin{aligned}
\frac{\partial}{\partial\theta_w} E_t[G_t] \approx \sum_{k=0}^{t-1} \Bigg(&\frac{\partial P_t(\phi_k \in \mathcal{M}_t)}{\partial\theta_w} E_t[Q(\phi_k|\phi_t, \mathcal{M}_t, \theta)|\phi_k \in \mathcal{M}_t] \\
&\left(\sum_{a\in\mathcal{A}} \pi(a|\phi_t, \phi_k) E_t[G_t|A_t = a] - \hat{v}(\phi_t, \theta)\right) \\
&+ P_t(m_t = \phi_k) \sum_{a\in\mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) \frac{\partial E_t[G_t|A_t = a]}{\partial\theta_w} \Bigg). \quad (5.10)
\end{aligned}$$

In addition to the approximation outlined above, we have subtracted a constant baseline of $\hat{v}(\phi_t, \theta)$, independent of m_t, as is common practice in policy gradient methods to reduce the variance of the gradient estimate. Note that this baseline does not effect the value of the expectation because $\frac{\partial P_t(\phi_k = \mathcal{M}_t)}{\partial \theta_w}$ must sum to zero to maintain the sum of probabilities at one and $\hat{v}(\phi_t, \theta)$ does not depend on k, meaning it can be pulled out of that sum. We could have done this in the single state variable case as well, however, in that case it would have given the same final answer, as is easily verified. Modifying equation 4.2 slightly to use exponentiated weights, the single item inclusion probabilities will be given by:

$$P_t(\phi_i \in \mathcal{M}_t) = \begin{cases} 1 & \text{if } i \in \Omega_t \\ \frac{n \exp(w_i)}{\sum\limits_{j \in \Omega_t} \exp(w_j)} & \text{if } i \notin \Omega_t, \end{cases} \tag{5.11}$$

where $\Omega_t = \underset{\omega \subset [0,..,t]}{\arg\min} |\omega|$ s.t. $\forall i \in [0,..,t] \setminus \omega$, $\frac{(n - |\omega|) \exp(w_i)}{\sum\limits_{j \in [0,..,t] \setminus \omega} \exp(w_j)} < 1$ is the inadmissible set. Recall from Chapter 4 that the inadmissible set refers to those indices where the inclusion probability of the associated state variable would be greater than one if computed naively. Applying this to work out the first term from the right hand side of equation 5.10 gives, for all $i \notin \Omega_t$:

$$\begin{aligned}
&\sum_{k=0}^{t-1} \frac{\partial P_t(\phi_k \in \mathcal{M}_t)}{\partial \theta_w} E_t[Q(\phi_k | \phi_t, \mathcal{M}_t, \theta) | \phi_k \in \mathcal{M}_t] \\
&\qquad \left(\sum_{a \in \mathcal{A}} \pi(a | \phi_t, \phi_k, \theta) E_t[G_t | A_t = a] - \hat{v}(\phi_t, \theta) \right) \\
&= \sum_{k \notin \Omega_t} \frac{\partial \log(P_t(\phi_k \in \mathcal{M}_t))}{\partial \theta_w} P_t(\phi_k \in \mathcal{M}_t) E_t[Q(\phi_k | \phi_t, \mathcal{M}_t, \theta) | \phi_k \in \mathcal{M}_t] \\
&\qquad \left(\sum_{a \in \mathcal{A}} \pi(a | \phi_t, \phi_k, \theta) E_t[G_t | A_t = a] - \hat{v}(\phi_t, \theta) \right) \\
&= \sum_{k \notin \Omega_t} P_t(m_t = \phi_k) \frac{\partial}{\partial \theta_w} \left(w_k - \log \left(\sum_{j \notin \Omega_t} \exp(w_j) \right) \right) \\
&\qquad \left(\sum_{a \in \mathcal{A}} \pi(a | \phi_t, \phi_k, \theta) E_t[G_t | A_t = a] - \hat{v}(\phi_t, \theta) \right) \\
&= \sum_{k \notin \Omega_t} P_t(m_t = \phi_k) \left(\frac{\partial w_k}{\partial \theta_w} - \sum_{i \notin \Omega_t} \frac{P_t(\phi_i \in \mathcal{M}_t)}{n - |\Omega_t|} \frac{\partial w_i}{\partial \theta_w} \right) \\
&\qquad \left(\sum_{a \in \mathcal{A}} \pi(a | \phi_t, \phi_k, \theta) E_t[G_t | A_t = a] - \hat{v}(\phi_t, \theta) \right)
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{k\notin\Omega_t} P_t(m_t = \phi_k)\frac{\partial w_k}{\partial \theta_w}\left(\sum_{a\in\mathcal{A}} \pi(a|\phi_t,\phi_k,\theta)E_t[G_t|A_t = a] - \hat{v}(\phi_t,\theta)\right)\\
&- \sum_{i\notin\Omega_t} \frac{P_t(\phi_i \in \mathcal{M}_t)P_t(m_t \notin \Omega_t)}{n - |\Omega_t|}\frac{\partial w_i}{\partial \theta_w}\left(E_t[G_t|m_t \notin \Omega_t] - \hat{v}(\phi_t,\theta)\right)\\
&\approx \sum_{k\notin\Omega_t} P_t(m_t = \phi_k)\frac{\partial w_k}{\partial \theta_w}\left(\sum_{a\in\mathcal{A}} \pi(a|\phi_t,\phi_k,\theta)E_t[G_t|A_t = a] - \hat{v}(\phi_t,\theta)\right).
\end{aligned}
$$

In the final line we have approximated $E_t[G_t|m_t \notin \Omega_t] \approx \hat{v}(\phi_t,\theta)$. This could be considered dubious since $\hat{v}(\phi_t,\theta)$ is trained not to approximation $E_t[G_t|m_t \notin \Omega_t]$ but the unconditioned expectation $E_t[G_t]$. The accuracy of this approximation in situations where inadmissible items are common will depend on the effectiveness of the query network. When inadmissible items are highly important in a given state, the gradient with respect to weights of admissible items should be negligible anyway. Since inadmissible items are guaranteed to be present, they should be selected with very high probability, meaning $P_t(m_t \notin \Omega_t)$ will be small. If, on the other hand, inadmissible items are not important in a state, $E_t[G_t|m_t \notin \Omega_t] \approx \hat{v}(\phi_t,\theta)$ should be a good approximation. One could in principle train a separate conditional value function to approximate $E_t[G_t|m_t \notin \Omega_t]$, however we deemed it unlikely to be worthwhile and did not do so in our experiments. In any case, OPGOR-DS, introduced in Section 5.7 makes no assumption on the accuracy of the subtracted baseline, and thus eliminates this issue.

With the approximations we applied, the multiple-state memory version of OPGOR works out very similarly to the single-state memory version discussed in Section 5.1. The only difference is the condition that we only propagate gradients through admissible items. Local modifications of the importance weights have no effect on items whose inclusion probability is saturated at one, hence we don't update those weights further. Likewise, the second term of equation 5.10 works out the same as the second term of equation 5.4 and no modification to its derivation is necessary, thus applying the same reasoning as in the single-state memory case we use the gradient estimator:

$$
\zeta_t = \begin{cases} \delta_t \frac{\partial w(m_t,\theta_t)}{\partial \theta} & m_t \notin \Omega_t \\ 0 & \text{otherwise} \end{cases}. \tag{5.12}
$$

The associated online update rule for OPGOR with a multiple-state memory is:

$$\theta_{w,t+1} = \theta_{w,t} + \alpha \mathbb{1}(m_t \notin \Omega_t)\delta_t \frac{\partial w(m_t, \theta_{w,t})}{\partial \theta_w}. \tag{5.13}$$

5.3 OPGOR with Multiple Read Heads

In practice it can be useful to use multiple read heads to allow the network to access multiple state variables at each time-step. In this case the heads can learn to specialize such that each provides a different kind of useful information to the policy network. To train the write network for this case we simply produce one gradient estimate for the item queried by each read head and average them to provide the training signal for the write network. In this case, define $m_t(j)$ to be the state variable returned by the query of the j^{th} read head and let N_Q be the total number of heads. Likewise each head j will be associated with its own query probability over the items in memory $Q_j(\mathcal{M}_t(i)|\phi_t, \mathcal{M}_t, \theta)$. Explicitly, we use the estimate

$$\frac{\partial E_0[G_0]}{\partial \theta_w} \approx E_0 \left[\sum_{t \geq 0} \delta_t \frac{1}{N_Q} \sum_{j=0}^{N_Q - 1} \mathbb{1}(m_t(j) \notin \Omega_t) \frac{\partial w(m_t(j), \theta_w)}{\partial \theta_w} \right], \tag{5.14}$$

for each parameter θ_w of the write network. Leading to the online update rule, for OPGOR with a multiple-state memory and multiple read heads:

$$\theta_{w,t+1} = \theta_{w,t} + \alpha \delta_t \frac{1}{N_Q} \sum_{j=0}^{N_Q - 1} \mathbb{1}(m_t(j) \notin \Omega_t) \frac{\partial w(m_t(j), \theta_w)}{\partial \theta_w}. \tag{5.15}$$

5.4 OPGOR(λ), OPGOR with Generalized Advantage Estimation

Instead of using only the one step TD error it is possible to introduce an eligibility trace to the training process for the write network to allow training using an advantage estimate based on the λ-return. The idea behind this more general use of eligibility traces is discussed by Schulman et al. (2015) under the name *generalized advantage estimation* (GAE). In addition to potentially facilitating faster learning, multi-step updates can help with partial observability by allowing information to propagate across aliased states. Since the focus of this thesis is partially-observable problems this could be particularly helpful. We will now illustrate how to derive a version of OPGOR which uses online GAE to construct an advantage estimate. We refer to this version of OPGOR as OPGOR(λ).

Consider the most general gradient estimate derived so far (for multiple state variables stored in memory, as well as multiple read heads) in equation 5.14. Recall the δ_t term used there serves as an estimate of $E_t[G_t|A_t = a] - E_t[G_t]$. In this case $R_{t+1} + \gamma\hat{v}(\phi_{t+1}, \theta)$ is used to estimate $E_t[G_t|A_t = a]$. We could equally well use any estimate of the action conditional expected return to estimate this value. One such estimate is the λ-return G_t^λ, for which we will use the expression given in equation 2.1 to derive a new advantage estimate as follows:

$$\begin{aligned} E_t[G_t|A_t = a] - E_t[G_t] &\approx E_t[G_t^\lambda - \hat{v}(\phi_t, \theta)] \\ &= \sum_{k=t}^{\infty} (\gamma\lambda)^{k-t}\delta_k. \end{aligned}$$

Substituting this into equation 5.14 in place of δ_t gives:

$$\begin{aligned} \frac{\partial E_0[G_0]}{\partial \theta_w} &\approx E_0\left[\sum_{t=0}^{\infty}\sum_{k=t}^{\infty}(\gamma\lambda)^{k-t}\delta_k \frac{1}{N_Q}\sum_{j=0}^{N_Q-1}\mathbb{1}(m_t(j) \notin \Omega_t)\frac{\partial w(m_t(j), \theta_w)}{\partial \theta_w}\right] \\ &= E_0\left[\sum_{t=0}^{\infty}\delta_t \sum_{k=0}^{t}(\gamma\lambda)^{t-k}\frac{1}{N_Q}\sum_{j=0}^{N_Q-1}\mathbb{1}(m_k(j) \notin \Omega_t)\frac{\partial w(m_k(j), \theta_w)}{\partial \theta_w}\right]. \end{aligned}$$

Now define an eligibility trace:

$$z_{w,t} = \sum_{k=0}^{t}(\gamma\lambda)^{t-k}\frac{1}{N_Q}\sum_{j=0}^{N_Q-1}\mathbb{1}(m_k(j) \notin \Omega_t)\frac{\partial w(m_k(j), \theta_w)}{\partial \theta_w}, \tag{5.16}$$

such that:

$$\frac{\partial E_0[G_0]}{\partial \theta_w} \approx E_0\left[\sum_{t=0}^{\infty} z_{w,t}\delta_t\right]. \tag{5.17}$$

We will compute $z_{w,t}$ as well as the update for θ_w online leading to the update equations for OPGOR(λ):

$$\begin{aligned} z_{w,t} &= \gamma\lambda z_{w,t-1} + \frac{1}{N_Q}\sum_{j=0}^{N_Q-1} \mathbb{1}(m_t(j) \notin \Omega_t)\frac{\partial w(m_t(j), \theta_{w,t})}{\partial \theta_w}, \\ \theta_{w,t+1} &= \theta_{w,t} + \alpha\delta_t z_{w,t}. \end{aligned}$$

5.5 Soft-OPGOR(λ), OPGOR with Soft Queries

In addition to the algorithm described so far, in which the query mechanism stochastically selects a single state variable per read head from memory, we experimented with an agent using soft queries. In this case rather than interpreting the query weights given by equation 3.1 as selection probabilities, each read head returns a sum over all state variables in memory, weighted by the associated query weight, as follows:

$$m_t(j) = \sum_{k=0}^{n} Q_j(\mathcal{M}_t(k)|\phi_t, \mathcal{M}_t, \theta_t)M_t(k). \tag{5.18}$$

This means that the query process is now differentiable and can be trained directly by backpropagation from the policy gradient signal.

Our derivation of OPGOR(λ) no longer applies directly in the soft query case. Instead we will adopt a heuristic update rule, motivated by the hard (stochastic) query case. In particular, where in the hard query case we would add only the gradients of the importance weights for the selected state variables to the trace, in the soft query case we will add gradients for each item in the memory weighted by its associated query weight. With this modification the update equations for soft-OPGOR(λ) can be summarized as follows:

$$\begin{aligned} z_{w,t} = {} & \gamma\lambda z_{w,t-1} + \frac{1}{N_Q}\sum_{k=0}^{n}\sum_{j=0}^{N_Q-1} \mathbb{1}(\mathcal{M}_t(k) \notin \Omega_t), \\ & Q_j(\mathcal{M}_t(k)|\phi_t, \mathcal{M}_t, \theta_{w,t})\frac{\partial w(\mathcal{M}_t(k), \theta_{w,t})}{\partial \theta_w} \\ \theta_{w,t+1} = {} & \theta_{w,t} + \alpha\delta_t z_{w,t}. \end{aligned}$$

Note that in the limit where the query is deterministically focused on one index, as well as in the single-state memory case, the hard and soft query versions of the algorithm become identical.

5.6 Integrating OPGOR with Actor-Critic

In our experiments we will test OPGOR(λ) as described in Section 5.4, which we will from now on refer to as the hard-OPGOR(λ). In addition, we will test the version using differentiable queries described in section 5.5, similarly referred to as soft-OPGOR(λ). For simplicity we will train the value, policy and query network with eligibility traces using the same λ as OPGOR. Also for simplicity, in this work the policy, value, query and write networks will use independent parameters. In the general case they may share some parameters. We will represent the complete vector of network parameters by θ. Parameter sharing can be accommodated by using a combined trace updated with the sum of the 4 different trace vectors as follows for hard-OPGOR(λ):

$$\begin{aligned}
z_t =& \gamma\lambda z_{t-1} + \frac{1}{N_Q}\sum_{j=0}^{N_Q-1} \mathbb{1}(m_t(j) \notin \Omega_t)\frac{\partial w(m_t(j),\theta_t)}{\partial\theta} \\
&+ \frac{\partial \hat{v}(\phi_t,\theta_t)}{\partial\theta} \\
&+ \frac{1}{2}\frac{\partial \log(\pi(a_t|\phi_t,m_t,\theta_t))}{\partial\theta} \\
&+ \frac{1}{2N_Q}\sum_{j=0}^{N_Q-1}\frac{\partial \log(Q_j(m_t(j)|\phi_t,\mathcal{M}_t,\theta_t))}{\partial\theta}, \\
\theta_{t+1} =& \theta_t + \alpha\delta_t z_t,
\end{aligned}$$

where now $m_t = m_t(0), ..., m_t(N_Q - 1)$. Aside from the write network update, which uses OPGOR, this algorithm is essentially actor-critic with eligibility traces as outlined in Section 2.1. Each head of the query network is treated as a policy with the items in memory as the space of possible actions. With soft-OPGOR(λ), we use the combined update rule:

$$\begin{aligned}
z_t =& \gamma\lambda z_{t-1} + \frac{1}{N_Q}\sum_{k=0}^{n}\sum_{j=0}^{N_Q-1}\mathbb{1}(\mathcal{M}_t(k) \notin \Omega_t)Q_j(\mathcal{M}_t(k)|\phi_t, \mathcal{M}_t, \theta_t)\frac{\partial w(\mathcal{M}_t(k), \theta_t)}{\partial\theta}, \\
&+ \frac{\partial\hat{v}(\phi_t, \theta_t)}{\partial\theta} \\
&+ \frac{1}{2}\frac{\partial\log(\pi(a_t|\phi_t, m_t, \theta_t))}{\partial\theta} \\
\theta_{t+1} =& \theta_t + \alpha\delta_t z_t,
\end{aligned}$$

where again $m_t = m_t(0), ..., m_t(N_Q - 1)$, but $m_t(j)$ is given by equation 5.18. Note that with soft queries, the trace update no longer explicitly incorporates a term for the query network gradient. Rather the query network parameters are included in the parameters for $\pi(a_t|\phi_t, m_t, \theta)$ and trained by the error backpropagated from the policy gradient directly.

5.7 Online Policy Gradient Over a Reservoir with Denominator Sampling (OPGOR-DS)

It's interesting that, under the assumption of a perfect critic, OPGOR can obtain unbiased gradients without storing any information about the denominator of the inclusion probabilities (aside from incrementally computing its value for use in Chao sampling). We will see in our experiments in Chapter 6, however, that when this assumption is violated, performance degrades quickly and OPGOR's behaviour can be unpredictable. In this section, we will show that it is possible to compute an online sample of the denominator as well, and by doing so we can remove the assumption that an unbiased critic is subtracted from the advantage estimate. We will refer to this alternative version of OPGOR as *OPGOR with denominator sampling* (OPGOR-DS).

We will again begin by deriving OPGOR-DS for the single-state memory case, with no eligibility traces. We begin our derivation of OPGOR-DS for a single-state memory from the following expression for the gradient of the expected return with respect to the write network parameters θ_w:

$$\frac{\partial}{\partial \theta_w} E_t[G_t] = \sum_{k=0}^{t-1} \Bigg(\frac{\partial P_t(m_t = \phi_k)}{\partial \theta_w} \sum_{a \in \mathcal{A}} (\pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a] - \hat{v}(\phi_t, \theta)) \\ + P_t(m_t = \phi_k) \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) \frac{\partial E_t[G_t|A_t = a]}{\partial \theta_w} \Bigg). \quad (5.19)$$

In this case we subtract a baseline $\hat{v}(\phi_t, \theta)$ purely for variance reduction. Working out the first term in equation 5.19:

$$\begin{aligned}
&\sum_{k=0}^{t-1} \frac{\partial P_t(m_t = \phi_k)}{\partial \theta_w} \left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a] - \hat{v}(\phi_t, \theta) \right) \\
&= \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \frac{\partial \log(P_t(m_t = \phi_k))}{\partial \theta_w} \\
&\qquad \left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a] - \hat{v}(\phi_t, \theta) \right) \\
&= \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \frac{\partial}{\partial \theta_w} \left(w_k - \log(\sum_{j=0}^{t-1} \exp(w_j)) \right) \\
&\qquad \left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a] - \hat{v}(\phi_t, \theta) \right) \\
&= \sum_{k=0}^{t-1} P_t(m_t = \phi_k) \left(\frac{\partial w_k}{\partial \theta_w} - \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w} \right) \\
&\qquad \left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_k, \theta) E_t[G_t|A_t = a] - \hat{v}(\phi_t, \theta) \right).
\end{aligned}$$

Note that this is essentially identical to the derivation of ordinary policy gradient (Sutton et al., 2000) except that we have explicitly expanded:

$$\frac{\partial \log(P_t(m_t = \phi_k))}{\partial \theta_w} = \left(\frac{\partial w_k}{\partial \theta_w} - \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w} \right),$$

in order to emphasize the term $\sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial w_i}{\partial \theta_w}$ which we note can be estimated by single-item Chao sampling. All that is necessary is to maintain a second reservoir of size one, which maintains an independently Chao sampled state variable $\tilde{m}_t$ using the same importance weights as the external memory. Each time we compute the gradient with respect to the importance weight of the state variable m_t we subtract the gradient of the importance weight of the state variable $\tilde{m}_t$. More precisely, we use the following online update rule for OPGOR-DS, analogous to the parameter update given in equation 5.9:

$$\theta_{w,t+1} = \theta_{w,t} + \alpha\delta_t \left(\frac{\partial w(m_t, \theta_{w,t})}{\partial \theta_w} - \frac{\partial w(\tilde{m}_t, \theta_{w,t})}{\partial \theta_w} \right). \tag{5.20}$$

By similar reasoning we can derive corrected algorithms for the hard-OPGOR(λ) and soft-OPGOR(λ), for the multiple-state memory case. Note that, in the multiple-state case, the gradient of the denominator we wish to sample remains almost the same. The main difference is that $\tilde{m}_t$ should be sampled only from admissible state variables. This can be achieved easily by only streaming items to the denominator reservoir when they become admissible in the external memory. Regardless of the size of the external memory, we use a single item $\tilde{m}_t$ for the denominator sample; one could use arbitrarily large reservoirs to obtain a lower variance estimate of the denominator gradient. The corresponding trace and parameter updates are straightforward modifications of the uncorrected algorithm. For hard-OPGOR-DS(λ) we get:

$$\begin{aligned} z_{w,t} &= \gamma\lambda z_{w,t-1} + \frac{1}{N_Q}\sum_{j=0}^{N_Q-1} \mathbb{1}(m_t(j) \notin \Omega_k) \left(\frac{\partial w(m_t(j), \theta_{w,t})}{\partial \theta_w} - \frac{\partial w(\tilde{m}_t, \theta_{w,t})}{\partial \theta_w} \right), \\ \theta_{w,t+1} &= \theta_{w,t} + \alpha\delta_t z_{w,t}, \end{aligned}$$

and for soft-OPGOR-DS(λ):

$$\begin{aligned} z_{w,t} = \gamma\lambda z_{w,t-1} + & \frac{1}{N_Q}\sum_{k=0}^{n}\sum_{j=0}^{N_Q-1} \mathbb{1}(\mathcal{M}_t(k) \notin \Omega_t) \\ & Q_j(\mathcal{M}_t(k)|\phi_t, \mathcal{M}_t, \theta_t) \left(\frac{\partial w(\mathcal{M}_t(k), \theta_{w,t})}{\partial \theta_w} - \frac{\partial w(\tilde{m}_t, \theta_{w,t})}{\partial \theta_w} \right), \\ \theta_{w,t+1} = \theta_{w,t} + & \alpha\delta_t z_{w,t}. \end{aligned}$$

The combination with the other components of our agent is a straightforward extension of the update rules outlined in Section 5.6.

To understand intuitively why subtracting the gradient of an independently sampled state variable negates the need for subtraction of an accurate baseline, note that this amounts to an estimate of the expected gradient of the importance weight of a state variable in memory at time t. By subtracting the expected gradient we no longer have to worry about emphasizing state variables whose presence is merely predictive of high expected return. The average gradient for state variables in the history is captured by $\tilde{m}_t$ and removed, meaning only state variables which facilitate better than average expected return relative to the current history will be emphasized.

5.8 Time Complexity

Limiting run-time to $O(k)$ per time-step, where k is the total number of parameters involved in the function approximation, is widely considered to be an important restriction for online RL algorithms. Generally, inference, the process of extracting an output for a given input, for a ANN based architecture requires $O(k)$ time, so limiting the learning to the same timescale implies the algorithm will scale well as the number of parameters increases, and learning will never become a major bottleneck. This requirement is met by the backpropagation algorithm for training ANN weights, which makes up the majority of the computation in our agent. Aside from this we should consider the computational complexity of the remaining parts of our agent.

The process of querying memory requires $O(nm)$ where n is the size of the memory and m the length of the state variables stored therein. This applies whether soft or hard queries are used, since in either case we have to compare every item in memory to the query to compute the query weights Q. In the soft query case, we additionally have to sum over each item weighted by their query weight and backpropagate through each, however this also requires $O(nm)$ and thus does not change the big O complexity, though it will somewhat increase the constant factor compared to the hard query version.

Computing the write network gradients for hard-OPGOR(λ) requires $O(k_w)$ where k_w is the number of parameters in the write network. soft-OPGOR(λ) on the other hand requires $O(nk_w)$ since we need to compute a gradient for each state variable in memory. hard-OPGOR-DS(λ) and soft-OPGOR-DS(λ) both add just one more write network gradient computation on top of the associated algorithm without denominator sampling.

The complexity of Chao sampling as a write mechanism is discussed in Chapter 4. In the common case where inadmissible items are relatively rare, this requires approximately constant time for the reservoir sampling process plus $O(m)$ to actually write the state variable to memory. Even in the worst case where each item is inadmissible when it is added it will require only at most $O(n)$ if implemented efficiently and thus does not affect the overall, per time-step, big O complexity of the agent given the complexity of querying memory.

Chapter 6

Experiments

We have established the architectural framework, and the theoretical foundation for OPGOR, and are ready to empirically investigate its performance and limitations. Recall that OPGOR is an online algorithm for training an ANN to generate weights for use in Chao sampling, which in turn is applied to manage which state variables are written to and retained in an external memory. It does so by applying a variant of policy gradient to emphasize retention of state variables which lead to high advantage when they are recalled later on. Our experiments here will investigate the ability of OPGOR, and a number of other selective memory strategies, to learn to selectively store and recall useful information. We will empirically evaluate how RL agents' augmented with OPGOR perform on a set of episodic, partially-observable problems designed to test an agent's capacity to inform present decisions based on past experiences within the same episode. We will also show how OPGOR can fail when a key assumption involved in our derivation does not hold. Specifically we illustrate the reliance of OPGOR on a state representation which is sufficient to accurately predict the history conditional expected return $E_t[G_t]$ at each time. We will also show that adding denominator sampling, as in OPGOR-DS, mitigates this issue while still running online.

We evaluated OPGOR on three psychology inspired environments designed to test the ability of an agent to learn to remember salient information from earlier in an episode and apply it to select good actions. The problems are inspired to varying degrees by problems explored by Wayne et al. (2018).

In each case, we simplified the problems to use a hand-coded binary feature vector and relatively small state space, as opposed to the more realistic high dimensional visual representation and virtual reality environments used in that work. Our main focus in the present thesis is on the mechanism for deciding what should be written to, and retained in, memory, hence we side-step the representation learning problem to more clearly isolate that component. A natural extension would be to apply OPGOR within a more sophisticated agent like the MERLIN architecture introduced by Wayne et al. (2018) to solve more realistic problems.

In addition to the three psychology inspired environments, in Section 6.5 we present a simple counterexample designed to illustrate the reliance of OPGOR on the availability of a state representation which is highly predictive of the history conditional expected return $E_t[G_t]$ at each time. The counterexample we present is a particularly pathological case, where because the state representation is insufficient, OPGOR learns importance weights that focuses on state variables that are predictive of high return, and neglects to remember those that help to condition good action selection. We will also see that OPGOR-DS has no such problem.

The primary architecture in which we applied OPGOR in all our experiments was the one shown in Figure 3.1, with the addition of multiple read heads as described in Section 5.3. In addition to the update rules described in Section 5.6, we applied entropy regularization to the policy, as well as L2 regularization to the generated importance weight $w(\phi_t, \theta_t)$. Entropy regularization helps to maintain exploration, and prevent premature convergence to suboptimal policies. L2 regularization was added to the importance weights to prevent weights from growing arbitrarily large, potentially leading to overflow. With these two additions our parameter update on each time-step (for the applicable eligibility trace z_t depending whether hard-OPGOR(λ) or soft-OPGOR(λ) was used) was:

$$\theta_{t+1} = \theta_t + \alpha\left(\delta_t z_t + \psi\frac{\partial\mathcal{H}_t}{\partial\theta} - \eta\frac{\partial w(\phi_t,\theta_t)^2}{\partial\theta}\right),$$

where $\mathcal{H}_t = \sum_{a\in\mathcal{A}}\pi(a|\phi_t, m_t, \theta_t)\log(\pi(a|\phi_t, m_t, \theta_t))$ is the single-step policy entropy. Following Mnih et al. (2016) we fixed $\psi = 0.01$ in all our experiments. We set η to a small value of 0.0001 to minimize its influence beyond avoiding numerical overflow and keeping the importance weights more-or-less centered about 0. We also fixed $\gamma = 0.99$ and $\lambda = 0.8$ unless otherwise specified, other hyperparameter values were selected in a manner described in each individual experiment.

6.1 Single Decision Keychain

We began with a straightforward test of the ability of an OPGOR augmented RL agent to learn to recall a single important state variable among a number of distractors. Recalling the important state variable at the right time would allow the agent to select the correct action to obtain a reward. Taking advantage of the simplicity of this environment we investigated how an RL agent, using OPGOR to manage its memory, performs with a variety of parameter settings.

The first environment we present is similar to the *secret informant problem* found in the work of Young et al. (2018). It is also vaguely similar to the *latent learning in a T-Maze environment* explored by Wayne et al. (2018). We call it *keychain* because the agent moved through a directed chain of cells which contained a single key observation. In each cell there were three available actions, which we label as *forward*, *up*, and *down*. Recalling the key observation would allow the agent to select the single rewarding action in the decision cell at the end of the chain. In every cell except for the decision cell, the *forward* action moved the agent to the next cell in the chain, while the other two actions caused it to remain in the same cell. If the correct action was selected in the decision cell at the end of the chain, a reward of one was given, otherwise no reward was given. The correct action varied each episode, and was specified by a one hot *action indicator* forming part of the state variable of the *informative cell*, which was placed at a random location in the chain in each episode.

The informative cell was distinguished from all the other cells in the chain,

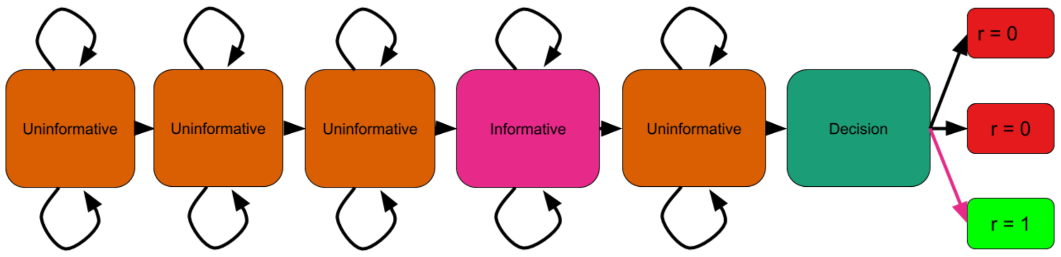


Figure 6.1: Visualization of single decision keychain environment. Here there are 5 cells before decision to keep the diagram small, in our experiments there will be 10. The informative cell (marked in pink) specifies what actions to take in the decision cell at the end of the chain in order to receive a reward. The agent must learn to distinguish informative from uninformative cells, through trial and error, in order to reliably receive a reward.

called *uninformative cells*, by two bits in the state variable. These bits were $(1,0)$ for the informative cell and $(0,1)$ for an uninformative cell. Thus, we refer to these bits as the *informative indicator* and *uninformative indicator* respectively. Other than that, the informative cell was indistinguishable from the uninformative cells, however for uninformative cells the associated action indicator was randomly generated with no correlation to the rewarding action in the decision cell. In total there were 10 cells before the decision cell in each episode, precisely one of which was informative.

An illustration of an instance of the single decision keychain problem is shown in Figure 6.1. Table 6.1 outlines the feature representation for the state variable presented to the agent in each cell.

Our first experiment used OPGOR(λ) with a single-state memory. We swept over α values in the set $\{2^{-i} : i \in \{3, 4, ..., 8\}\}$. Note that hard and soft OPGOR(λ) are identical in the single-state memory case. The results for this experiment are summarized in Figure 6.2.

Figure 6.2 (a) shows average return over training episodes for a variety of α values. Note that the maximum expected return is 1.0 if the agent reliably chooses the correct action in the decision cell. Figure 6.2 (b) shows the

Feature Name	Length	Meaning
Cell Index	Length of chain	Node in the chain the agent currently occupies.
Action Indicator	Number of actions	Suggested action indicated by current cell.
Informative Bit	1	Indicates the suggested action and decision cell are accurate.
Uninformative Bit	1	Indicates the suggested action and decision cell are random.
Decision Cell Identifier	1	Always 1 in this environment, different in multiple decision keychain, but included here for uniformity.
Decision Cell Indicator	1	Indicates the current cell is the decision cell.

Table 6.1: Feature representation for single decision keychain environment. Features are listed in the same order they appear in the state variable.

difference between average importance weight for informative state variables and uninformative state variables for $\alpha = 2^{-5}$, which was the optimal value in terms of average return over the last 100 episodes.

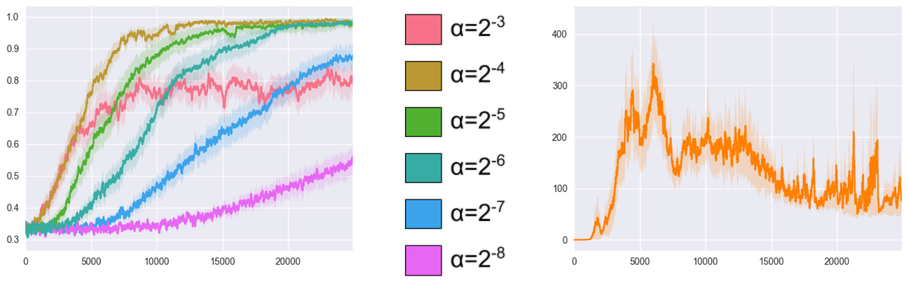
In this experiment, the agent reached essentially perfect performance with a range of α values. The write network was able to learn to significantly separate the informative cell from the uninformative cells with an average difference of roughly 100 by the end of training. Since in the single-state memory case there are no inadmissible weights, the inclusion probability is always proportional to the exponential of the importance weights. This means there was negligible probability of storing an uninformative state variable once an informative state variable had been observed.

In our next experiment we increased the memory to 3 states in order to test whether OPGOR(λ) is still able to manage the memory effectively in this slightly more complex case. Recall that in extending OPGOR to multiple state variables in Section 5.2 we made some additional approximations, hence this experiment will serve as an initial test of whether these approximations were reasonable. For this experiment we held the α value fixed at 2^{-5}, the optimal value from the single-state memory experiment. The results of this experiment are shown in Figure 6.3.

Figure 6.3 (a) shows returns for agents using hard-OPGOR(λ) and soft-OPGOR(λ). Figure 6.3 (b,d) shows the difference between average importance weight for informative state variables and uninformative state variables, for the hard and soft query case respectively. Figure 6.3 (c,e) shows the evolution of the average query vector element, generated in the decision cell associated with the informative indicator, uninformative indicator, and first decision cell identifier.

For both the agent using hard queries with hard-OPGOR(λ) and the agent using soft queries with soft-OPGOR(λ) we see similar performance to using a single-state memory, with the soft-query agent appearing to reach optimal performance somewhat faster. This makes some sense, since with soft queries, as long as the informative state variable occupies one of the three available memory slots the agent will have some signal indicating its presence. This could allow the agent to consistently benefit from having the informative state in memory, even with relatively imprecise queries. Both the importance weights and the query vector elements associated with the relevant bits were learned as expected. The query vector element associated with the decision cell identifier remained near zero in both cases, since the fact that there was only one decision cell made it irrelevant. In both hard and soft query cases there was a significant separation between the importance weights assigned to informative and uninformative cells, though somewhat less than in the single-state memory case. This makes sense because the agent had three chances to remember the single informative cell, making it more likely to appear in memory at a given importance weight difference. Also, in the multiple-state memory case, importance weights may become inadmissible if they are high enough, which guarantees a state variable's presence in memory and halts further updates.

Our final experiment in this section evaluated the effect of disabling eligibility traces, fixing $\lambda = 0$. Again, we held α fixed at 2^{-5}. Eligibility traces are not really necessary in this environment since the only decision which directly impacts received reward gives immediate feedback. In other environments,



(a) Return (b) Importance Weights

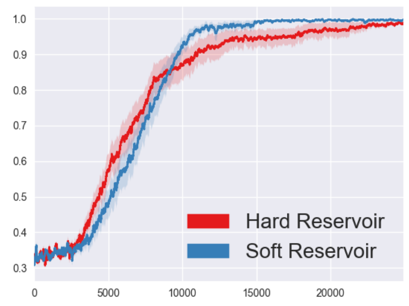
Figure 6.2: Results for single decision keychain with single-state memory OPGOR(λ) agent. In the single-state memory case the query module is unnecessary, and the soft and hard versions of the algorithm are identical. The x-axis shows training episodes. (a) shows the average return for a variety of settings of the step-size α. (b) shows the difference between the average importance weight assigned to informative and uninformative cells for the optimal α value of 2^{-5}. All curves are smoothed with a running mean over 100 episodes and show an average of 30 runs with error bars showing standard error in the mean.

they will be useful or, due to partial observability, even necessary to achieve good performance. Hence we wish to confirm they don't significantly hurt the performance of OPGOR(λ) in simpler cases. This experiment served as an initial check that the performance is not significantly impacted by applying eligibility traces when they are not needed. These results are shown in Figure 6.4.

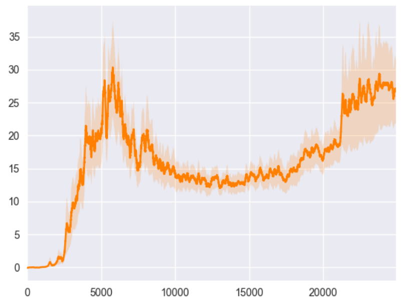
We observed that disabling eligibility traces had negligible impact in both the hard and soft query case. This is reassuring since we would not expect them to be necessary for the keychain problem. It is nonetheless informative to note that performance does not suffer by having them enabled. In the next section we will show an environment where eligibility traces are crucial to propagate information across aliased states.

6.2 Two Decision Keychain

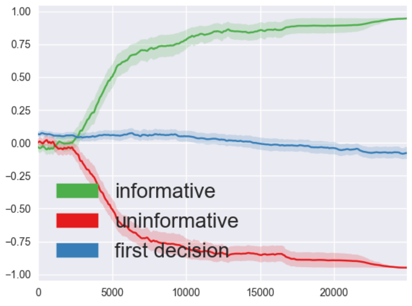
In the previous section we verified the feasibility of OPGOR in the simple case where only one crucial state variable must be remembered. We are now interested in how OPGOR scales as the problem becomes more complex. In



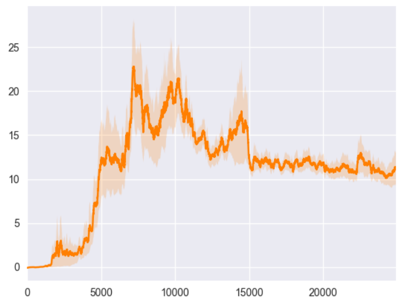
(a) Return



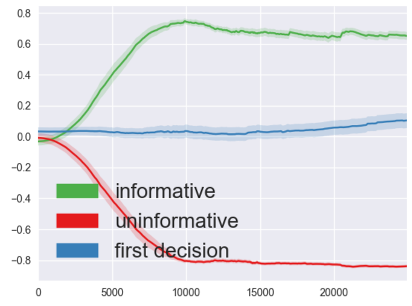
(b) Importance Weights (Hard Queries)



(c) Queries (Hard Queries)

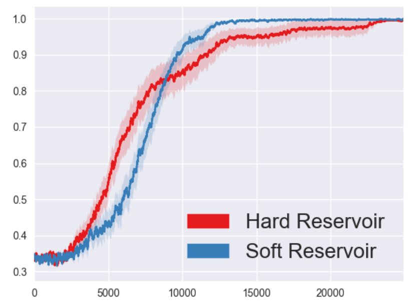


(d) Importance Weights (Soft Queries)

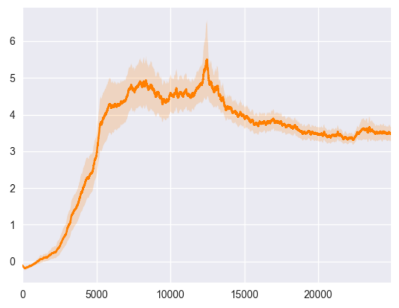


(e) Queries (Soft Queries)

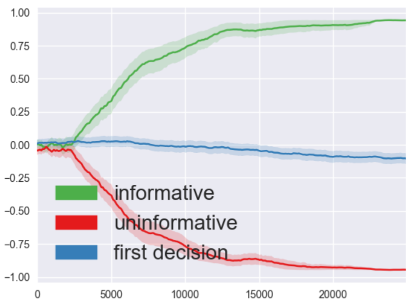
Figure 6.3: Results for single decision keychain with 3 state memory OPGOR(λ) agent. The x-axis shows training episodes. (a) shows returns for the hard and soft variant of the algorithm. (b,d) shows the difference between the average importance weight assigned to informative and uninformative cells. (c,e) shows three important elements of the query vector output by the query network in the single decision cell. All curves are smoothed with a running mean over 100 episodes and show an average of 30 runs with error bars showing standard error in the mean.



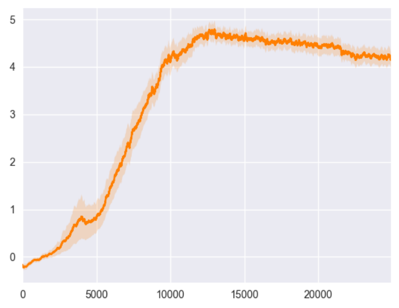
(a) Return



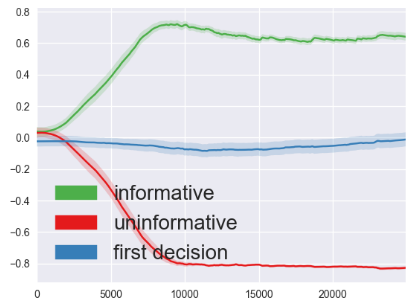
(b) Importance Weights (Hard Queries)



(c) Queries (Hard Queries)



(d) Importance Weights (Soft Queries)



(e) Queries (Soft Queries)

Figure 6.4: Results for single decision keychain with 3 state memory OPGOR agent, with eligibility traces disabled ($\lambda = 0$). The x-axis shows training episodes. (a) shows returns for the hard and soft variant of the algorithm. (b,d) shows the difference between the average importance weight assigned to informative and uninformative cells. (c,e) shows three important bits of queries output by query network in the single decision cell. All curves are smoothed with a running mean over 100 episodes and show an average of 30 runs with error bars showing standard error in the mean.

particular, in this section, we modified the environment from Section 6.1 by adding a second decision cell, as well as a second informative cell. In this case, the agent had to select the correct action in each of the two decision cells at the end of the chain in order to obtain a reward. Each decision cell corresponded to a separate informative cell located at a different random position in the chain. Informative cells additionally indicated which decision cell they provided information about, via a *decision cell identifier* in the state variable. For uninformative cells both the decision cell identifier and action indicator were set randomly.

The two decision version of keychain provided a substantial increase in difficulty compared to the single decision version. In this case, a random agent had only a 1/9 chance of receiving reward, and the agent had to learn to make queries conditional on which decision cell it occupied in order to select the correct action. Also, since the agent had no way to tell whether its first decision was correct until after making its second decision, there was a delay in credit assignment and eligibility traces were necessary to perform well on this problem. Without eligibility traces the agent could learn to correctly select the action in the second decision cell, but would have no way to receive positive reinforcement for its first choice.

An instance of this modified version of the problem is shown in Figure 6.5. Table 6.2 shows the associated feature representation.

Our first experiment with this environment used agents with hard queries using hard-OPGOR(λ) and agents with soft queries using soft-OPGOR(λ) all with a 3 state memory. Again, we swept alpha values from the set $\{2^{-i} : i \in \{3, 4, ..., 8\}\}$. The results are summarized in Figure 6.6.

Figure 6.6 (b,c) shows returns for the full range of tested α values for the hard and soft query case respectively. Figure 6.6 (a) compares returns for the optimal α value for each, in terms of average performance over the final 100 episodes. Incidentally the optimal α value was 2^{-7} for both versions. Figure 6.6 (d,g) shows the difference between the average importance weight assigned to informative and uninformative cells for the optimal α. Figure 6.6 (e,h) shows the evolution of the average query vector element generated in

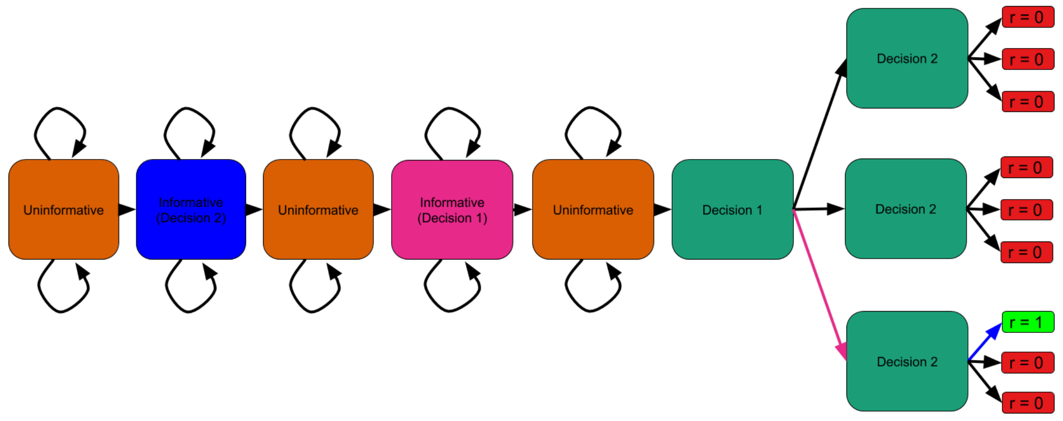


Figure 6.5: Visualization of two decision keychain environment. The two informative cells (marked in blue and pink) specify what actions to take in the corresponding decision cell at the end of the chain in order to receive a reward. The agent must learn to distinguish informative from uninformative cells, through trial and error, in order to reliably receive a reward.

Feature Name	Length	Meaning
Cell Index	Length of chain	Node in the chain the agent currently occupies.
Action Indicator	Number of actions	Suggested action indicated by current cell.
Informative Bit	1	Indicates the suggested action and decision cell are accurate.
Uninformative Bit	1	Indicates the suggested action and decision cell are random.
Decision Cell Identifier	Number of decisions	For a decision cell, identifies the current cell. Otherwise indicates which decision cell the current cell provides information about (if informative, otherwise it's random).
Decision Cell Indicator	1	Indicates the current cell is a decision cell.

Table 6.2: Feature representation for keychain environment. Features are listed in the same order they appear in the state variable.

the first decision cell, associated with the informative indicator, uninformative indicator, first decision cell identifier and second decision cell identifier. Figure 6.6 (f,i) shows the same thing but for the query vector generated in the second decision cell.

In this case, both versions of the agent performed reasonably well (far better than the 1/3 return that would be expected from only handling one decision correctly for example), with the soft-query agent significantly outperforming the hard-query agent. However neither reached perfect performance. The primary cause of this was likely the small memory of size 3, which had only one more slot than necessary to complete the task.

While both hard-OPGOR(λ) and soft-OPGOR(λ) learned to separate informative from uninformative, as evident from the importance weight curves, there was still a risk of lingering in a certain informative cell and adding three copies of one of the two informative cells rather than at least one of each. A quick calculation shows that if we assume all other aspects of the agent were perfect, but each of the three state variables in memory were occupied equiprobably by either informative cell, the expected return is given by $3/4 + 1/4 \cdot 1/3 = 0.83$ (i.e., the odds that both decision cells are contained in memory at least once, plus the odds that they aren't times the expected return for one random decision). This highlights a limitation of OPGOR in its current form which is that it does not account for the history when deciding inclusion probability for the current state variable. In particular, the algorithm is unaware when observing a certain informative state variable for the second time that it has already recorded it in memory. This issue could be partially addressed by using a recurrently updated state variable, which could perhaps learn to note that certain information has already been stored in memory. We also see that the query network is able to learn queries appropriately, increasing the weight assigned to the informative bit, decreasing the weight assigned to the uninformative bit, and increasing the weight assigned to the associated decision cell identifier in each decision cell.

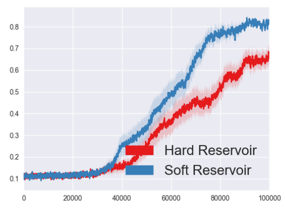
Our next experiment tested the effect of adding a second read head to the agent, as described in Section 5.3. We fix alpha to 2^{-7} the optimal value from the first experiment of this section, and maintain all other parameter. Note that two read heads are by no means necessary in this problem since only one informative state variable is needed to make the correct decision in each decision cell. Nonetheless, we may expect to see a benefit to early improvement resulting from providing the agent with more chances to select useful information. The results are shown in Figure 6.7.

The second read head did not appear to speed initial improvement significantly. However, importantly there was no significant detriment to final performance in either the hard or soft query case. In more complex problems, the idea of having multiple read heads is that each can specialize to a particular purpose. This specialization among individual read heads was observed for example in the work of Wayne et al. (2018).

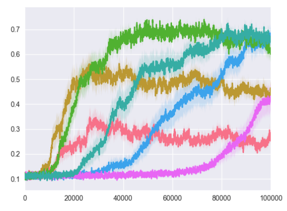
Next we observed the effect of increased length of temporal dependencies by increasing the number of cells prior to the decision cells from 10 to 20, again using 2 read heads, holding α and all agent parameters fixed. We refer to this new environment as *two decision, double length keychain.* Figure 6.8 shows the result. Note the increased number of training episodes in the x-axis of each plot.

The primary effect of this doubling of environment length was an increase in training time by roughly a factor of three. Final performance did not appear to be affected in the soft query case. In the hard query case learning does not appear to have plateaued, but the performance at 300000 episodes in the double length environment was comparable to that at 100000 episodes in the original two decision keychain environment. This experiment gives an early indication of how OPGOR scales with length of temporal dependencies.

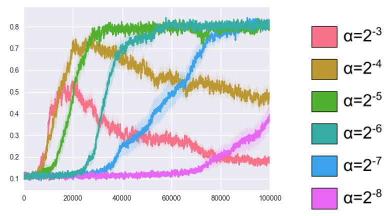
Next, again using the double length environment, we expanded the memory size from 3 to 5 to give the agent a better chance to store the two relevant state variables. Here, again, we use the version of the architecture with 2 read heads. With this version of the architecture and keychain environment we tested the hard-OPGOR(λ) and soft-OPGOR(λ) along with a number of alternative selective memory strategies and baselines. This allowed us to assess the difficult of the problem when approached with other methods, as well as to frame the performance of hard and soft OPGOR(λ) against possible alternatives. In what follows we describe each of the agents we compared against, and describe their role in our evaluation. Figure 6.9 shows a rough



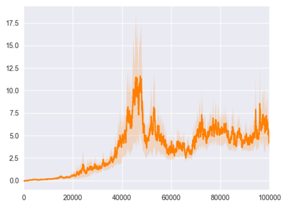
(a) Return
(Best α)



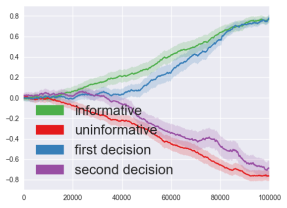
(b) Return
(Hard Queries)



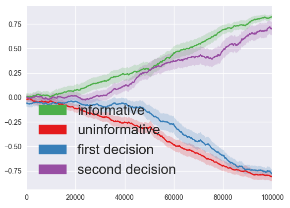
(c) Return
(Soft Queries)



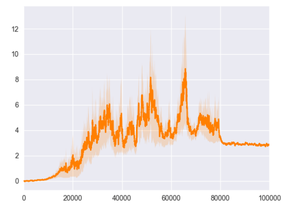
(d) Importance Weights
(Hard Queries)



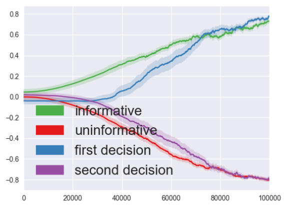
(e) Decision 1 Queries
(Hard Queries)



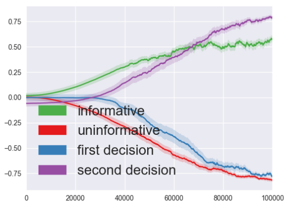
(f) Decision 2 Queries
(Hard Queries)



(g) Importance weights
(Soft Queries)



(h) Decision 1 Queries
(Soft Queries)

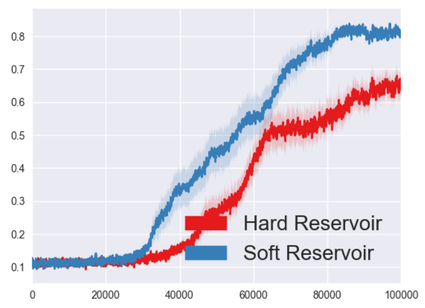


(i) Decision 2 Queries
(Soft Queries)

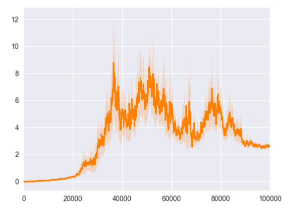
Figure 6.6: Results for two decision keychain with 3 state memory, single read head OPGOR(λ)(λ) agent. The x-axis shows training episodes. (b,c) shows the average return for a variety of settings of the step-size α for the hard and soft variants of the algorithm. (a) compares the results for optimal α, in terms of average return over the final 100 episodes. (d,g) shows the difference between the average importance weight assigned to informative and uninformative cells for the optimal α value of 2^{-7}. (e,h) shows four important bits of queries output by the query network in the first decision cell. (f,i) shows four important bits of queries output by the query network in the second decision cell. All curves are smoothed with a running mean over 100 episodes and show an average of 30 runs with error bars showing standard error in the mean.

taxonomy of the agents we include in our comparison, illustrating how the various approaches relate. We also compared hard and soft OPGOR(λ) against the same set of agents in the next two sections of this chapter.

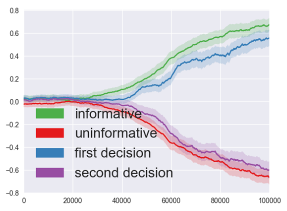
Memoryless: This agent uses no memory at all. Instead it consists of a



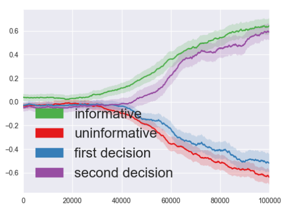
(a) Return



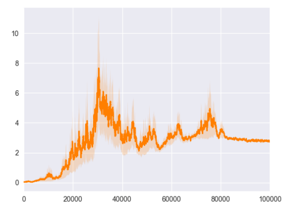
(b) Importance Weights (Hard Queries)



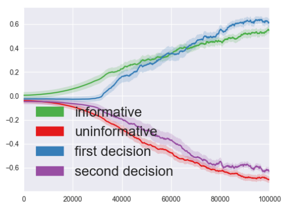
(c) Decision 1 Queries (Hard Queries)



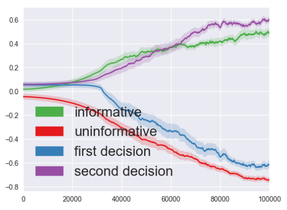
(d) Decision 2 Queries (Hard Queries)



(e) Importance Weights (Soft Queries)



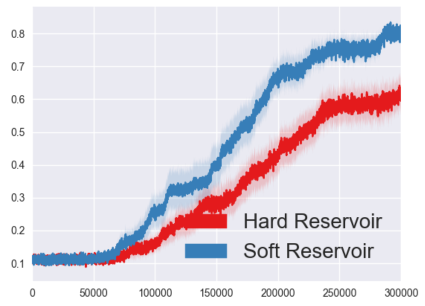
(f) Decision 1 Queries (Soft Queries)



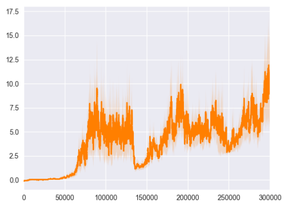
(g) Decision 2 Queries (Soft Queries)

Figure 6.7: Results for two decision keychain with 3 state memory, 2 read head OPGOR(λ) agent. The x-axis shows training episodes. (a) shows the average return for the hard and soft variants of the algorithm. (b,e) shows the difference between the average importance weight assigned to informative and uninformative cells. (c,f) shows four important bits of queries output by one of the two read heads in the first decision cell. (d,g) shows four important bits of queries output by one of the two read heads in the second decision cell. All curves are smoothed with a running mean over 100 episodes and show an average of 30 runs with error bars showing standard error in the mean.

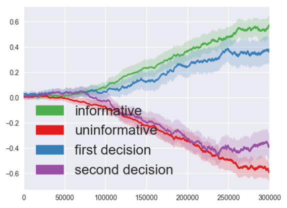
value network identical to our main architecture, along with a policy network with the same number of hidden units and layers as the main architecture but using only the current state variable as input. Since the problems here were designed to test memory we expected this agent to perform very poorly, but



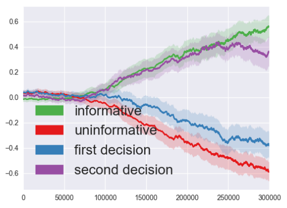
(a) Returns



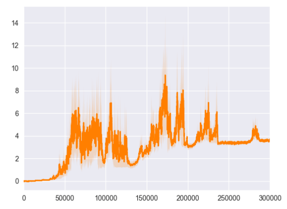
(b) Importance Weights (Hard Queries)



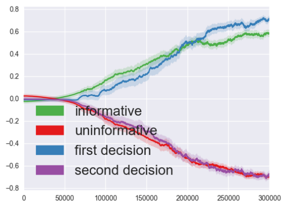
(c) Decision 1 Queries (Hard Queries)



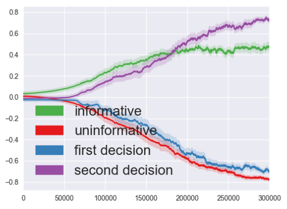
(d) Decision 2 Queries (Hard Queries)



(e) Importance Weights (Soft Queries)



(f) Decision 1 Queries (Soft Queries)



(g) Decision 2 Queries (Soft Queries)

Figure 6.8: Results for two decision, double length keychain with 3 state memory, 2 read head OPGOR(λ) agent. The x-axis shows training episodes. (a) shows the average return for the hard and soft variants of the algorithm. (b,e) shows the difference between the average importance weight assigned to informative and uninformative cells. (c,f) shows four important bits of queries output by one of the two read heads in the first decision cell. (d,g) shows four important bits of queries output by one of the two read heads in the second decision cell. All curves are smoothed with a running mean over 100 episodes and show an average of 30 runs with error bars showing standard error in the mean.

provide an idea of how important memory is in each problem.

Uniform Reservoir Sampling: This agent is similar to our soft-OPGOR(λ) agent with soft queries, but with importance weights fixed to one instead of

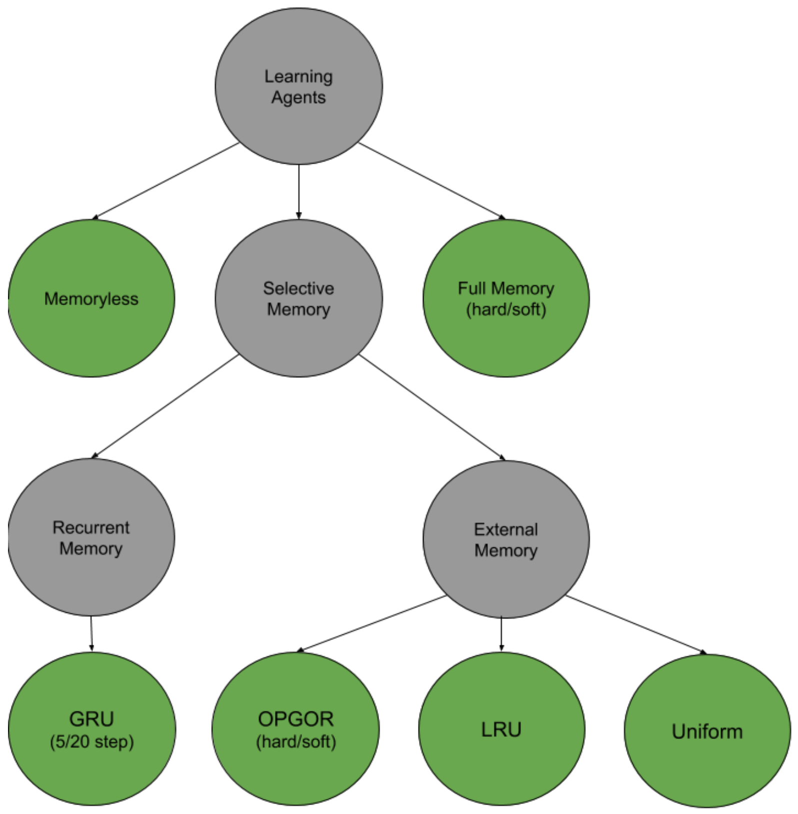


Figure 6.9: Taxonomy of different memory mechanisms we test in our experiments. Selective memory mechanisms span everything between the two extremes of memoryless agents, which rely on only the immediate observation, and full memory agents which remember the full history and need only learn how to use it. We divide selective memory into recurrent memory and external memory. Mechanisms we include in our experiments are marked in green, with parenthesis indicating multiple variations of a particular mechanism.

learned using soft-OPGOR(λ). This reservoir sampling procedure reduces to the simple procedure for sampling items uniformly introduced in Chapter 4. This agent gave us an idea of how effective OPGOR(λ) was at tuning importance weights, and how important this was in each problem. We only tested the soft query version of this agent because in general we found soft queries performed better.

Full Memory with Hard Queries: This agent performs queries over the state variables of all visited states, rather than a small bounded memory, hence no selective memory procedure is necessary. It is otherwise identical to our main architecture. In general we expected this agent to perform better than OPGOR(λ), since it always keeps around all the necessary information, and simply had to learn to correctly retrieve it. However, it required much more memory, and more computation to perform the query, which would scale poorly to larger problems. This agent gave a sense of how much our selective memory agents lost by restricting their memory to a small set of state variables.

Full Memory with Soft Queries: This agent is similar to the "full memory with hard queries" agent described above except using the soft query mechanism described in Section 5.5. The comparison between the hard and soft query variants was broadly interesting, as hard and soft query and write mechanisms are both used in various places in the literature and each appears to have advantages and disadvantages.

Least Recently Used (LRU): For each item in memory, this agent keeps track of a usage timer equal to the number of time-steps passed since each item in memory was either written or returned as a query result. It always replaces the item with the largest usage timer with the current state variable. This is based on the reasoning that items which the query network has determined would be useful to query in the recent past are also likely to be more useful in the future. This served as a simple, though potentially powerful, heuristic against which to test OPGOR(λ).

Gated Recurrent Neural Network (GRU): This agent replaces the entire memory module of our main agent with a GRU. Note however that as in the other agents the value function approximation is not recurrent, but rather uses the same feed forward architecture as the other agents in order to make the architectures as comparable as possible. Of all the agents we test the GRU is the most distinct from our main agent, replacing the entire read write semantics of the external memory with the differentiable single step updates of a GRU. We fix the number of GRU cells to 32, matching the number of units in each of the hidden layers of our architecture.

We tested this agent with 2 different truncation lengths for backpropagation through time. First, length 5 which requires similar memory and compute time to our external memory based agents with 5 memory slots, which is the setting we used in our comparison. Second, length 20 which more reasonably captures the length of temporal dependencies in the problems we tested on. In our implementation, a truncation length of 20 made the agent roughly 3.5 times slower per time-step than the reservoir based agent when running on a single CPU. Like all the other agents we trained the GRU agent online, computing gradients and updating at each time-step. More precisely, at each time-step we computed a forward pass starting from the earliest time in the truncation window, then propagated back the full truncation length. The GRU hidden state was propagated forward each time an observation was dropped from the truncation window.

We see external memory, and by extension OPGOR, as complementary to, not a replacement for, the incremental state updates used in architectures such as GRUs. Nonetheless, it was interesting to see how a GRU alone can perform on the tasks in our test suite, hence the inclusion of this mechanism.

We test agents using hard and soft OPGOR(λ) along with each of the above alternatives on two decision, double length keychain. The OPGOR(λ) agents, along with the uniform reservoir sampling agent, and LRU agent all use a 5 state memory. All of the external memory based agents used 2 read heads. The α values used by each agents was tuned from $\alpha \in \{2^{-i} : i \in \{3, ..., 8\}\}$ to maximize average return over the final 100 episodes. Figure 6.10 shows the evolution of the average return for each of these agents.

Both the hard and soft OPGOR(λ) based agents improved significantly over the memoryless agent and unweighted reservoir agents, indicating that OPGOR(λ) learned useful weights on this problem. The soft query agent slightly outperformed the hard query agent.

The soft full memory agent was, perhaps unsurprisingly, the best overall. This agent obtained essentially perfect performance in a very short time, setting the bar for what can be done with immediate access to the full history. The hard full memory agent also did quite well, though not nearly so well as the soft query version.

With truncation length 5, the GRU agent eventually began to improve, though significantly slower than any of the external memory based methods.

With truncation length 20, the GRU agent was quite competitive, obtaining final performance similar to that of the soft reservoir agent. However, as we noted earlier, its computation time per step was around 3.5 times longer in this case.

The LRU agent performed surprisingly well, similarly to the hard version of the full memory agent, with which it shares a query method, over the last half of training. It's interesting to note that the queried values, on which the LRU agent depends to decide which memories to maintain, are only relevant at the end of the episode. This means that the LRU agent most likely relies on harmless over-generalization, wherein the values that will be useful at the end are also queried throughout the episode. Another interesting thing to note is that in principle the LRU agent could learn to make queries intentionally to reset the usage timer of an item in order to keep it around. This is somewhat analogous to a human mentally repeating something to themselves so that they can remember it when needed. This is, however, only possible if the memory reading policy is trained using a trial and error mechanism directly, and not if it receives policy gradients directly from the policy as in the soft query case. This highlights an interesting potential advantage of the hard query mechanism, it has the ability to learn to exploit any potential secondary effects of remembering a particular state variable, an ability which is missing from a differentiable query mechanism trained on the policy gradient alone.

To get a better idea of what the write network of each OPGOR(λ) agent learned, we ran an additional 1000 episodes using the trained OPGOR(λ) agents from the previous experiment. During these 1000 episodes we recorded the importance weights associated with each observed state variable. The results of this experiment are summarized in Figure 6.11.

Figure 6.11 (a) shows the result for the hard query version of the algorithm, while Figure 6.11 (b) shows the result for the soft query version of the algorithm. To show as clearly as possible the effect of the importance weights we use the exponentiated importance weights, which are proportional to the selection probability. In addition, we applied two normalizations before plotting the average importance weights as a function of agent, and active state variable element. First, we divided the value of each importance weight by the maximum importance weight observed within the same episode, this ensured the plots were not affected by variability in importance weights between episodes. Since state variables only compete with those within the same episode for a spot in memory, this better highlights the relative magnitude in case there was large intra-episode variation. Second, we normalized by the maximum over state variable elements for each agent, such that the highest weighted element was always set to one.

The results for the importance weights in both the hard query case and soft query case were somewhat unexpected. In each case, the largest importance weights were usually associated with decision indicators. We hypothesize that this was a vestige from early learning which persisted because it was not significantly harmful. Before the value function was well trained, any positive reward received will result in a significant positive TD error. Since rewards are only received at the end of the episode, and it is only possible to have a decision cell in memory at the end of an episode, the system may learn to associate receiving a decision cell as the result of a query with a positive TD error. One might assume this bias should have been unlearned as the value function becomes more accurate, allowing the agent to learn that being in a decision cell, as opposed to querying the decision cell indicator, is what tends to result in a positive reward. However, if the high importance weight is already learned when the system learns not to query the decision cell indicator, the importance weight will rarely be updated further, as it is rarely received as a response to a query. Thus the importance weight may be stuck at a high value even when the state variable is learned not to be useful.

Note that the high importance weight attached to decision cells was nearly harmless, as the memory held five state variables and needed to store only two informative state variables to succeed. Since there were only two decision cells, and it was impossible to stay in a decision cell for more than one time-step, the worst case scenario was that the first decision cell was in memory when the agent reached the second. Replacing a single informative cell with a decision cell before the final decision was unlikely to matter much. Furthermore, we can

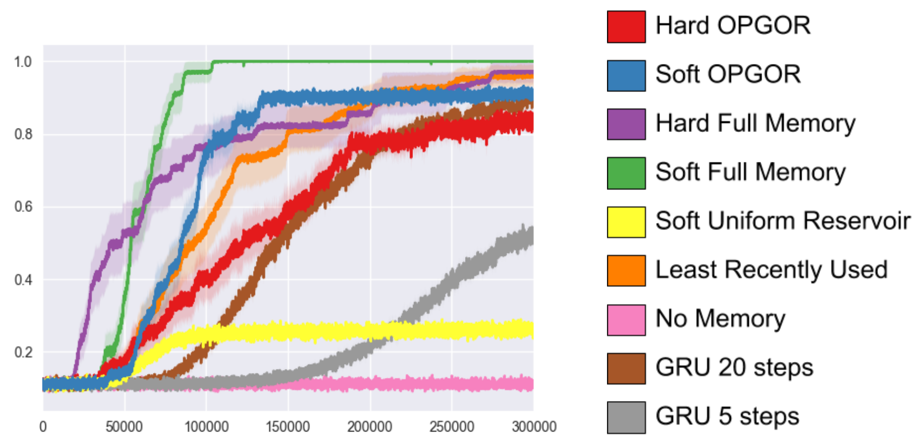
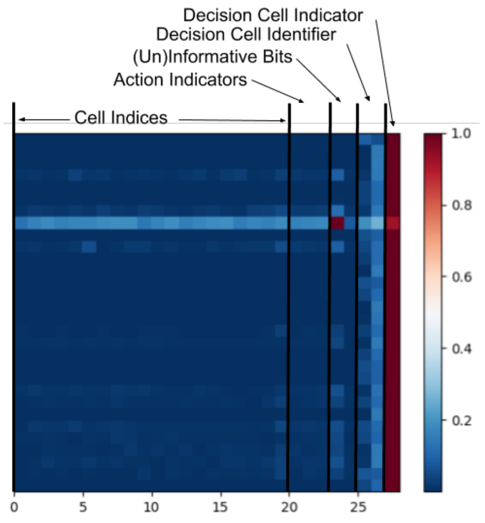


Figure 6.10: Comparison of returns v.s. training episodes for hard-OPGOR(λ) and soft-OPGOR(λ) agents along with various alternatives on two decision, double length keychain environment. The step-size parameter α is tuned separately for each agent by selecting for best average performance over the last 100 episodes over values from the set $\{2^{-i} : i \in \{3, 4, ..., 8\}\}$. All curves are smoothed with a running mean over 100 episodes and show an average of 30 runs with error bars showing standard error in the mean.

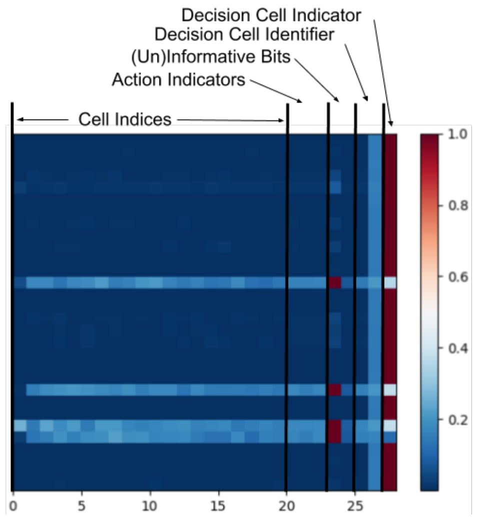
also see in both Figure 6.11 (a) and Figure 6.11 (b) that the importance weight associated with the second decision cell identifier tended to be significantly higher than the first. Since storing the second decision cell is entirely harmless, as the episode ends immediately after, this likely indicates that the agent partially learned to correct the minor bias. We can tell from the reasonably high performance, as well as the many plots of importance weight ratios for informative and uninformative cells in this section, that the agent was still able to learn to separate the informative and uninformative cells. This is, however, hardly visible in Figure 6.11 due to the high weight attached to the decision cell indicator.

6.3 Rapid Reward Valuation

The keychain problem served as a simple and direct instantiation of the kind of problem a memory system may be expected to handle. Keychain is, however, limited in the sense that recalled information is only required to condition action selection at the very end of an episode. In more realistic cases, history



(a) Importance Weights
(Hard Queries)



(b) Importance Weights
(Soft Queries)

Figure 6.11: Average, normalized, exponentiated importance weights associated with activation of various features for the hard-OPGOR(λ) and soft-OPGOR(λ) agent on the keychain environment. Average is taken over 1000 episodes generated after 300,000 episodes of training. The y-axis shows different independent training runs, the x-axis shows the different features of the state variable. Two max normalizations are applied. First, we divide the value for each feature observed in a given episode by the maximum importance weight value observed in the same episode, this is done because importance weights from different episodes will not affect the inclusion probability of a given state variable. Second, we divide the weights of all features by the max over all features for each training run separately, this is done for readability so the highest weighted feature over each run is set to one.

conditional decisions must be made at many points in time and a selective memory agent must learn to balance storing information required for each individual decision. In addition, determining what information is relevant to a particular decision may not be as simple as matching a single salient feature. Instead, relevant information may rely on a more subtle relationship between the decision to be made and the information which informs it. In this section, we evaluate hard-OPGOR(λ) and soft-OPGOR((λ), along with the alternative selective memory strategies discussed in the previous section, on a problem designed to highlight these more complex aspects of selective memory.

This environment is a simplified version of the *rapid reward valuation* task outlined by Wayne et al. (2018). The task consisted of a 5x5 grid world populated by an agent along with 15 *food items* of 3 distinct types. The agent could determine its position on the grid via a *cell index* which formed part of the state variable. The presence of each item in a cell was represented to the agent by a random binary *food type indicator* (of length 5 in our experiments), which was included in the state variable. Each food type was also classified as either good or bad, mandating that there be at least one good and one bad item. When occupying the same cell as a food item, the agent could choose to select the *eat* action or *discard* action. Eating a good item gave +1 reward while eating a bad item gave -1. Conversely discarding a good item gave -1 reward while discarding a bad item gave +1. When an item was either eaten or discarded it was removed from the map and the next state variable seen by the agent included the food type indicator along with a two bit *good/bad food indicator*, indicating whether it was good or bad. Selecting eat or discard in a cell which contained no food item had no effect. In addition to the eat and discard actions, the action space consisted of moving in the four cardinal directions (*up*, *down*, *left*, *right*). The agent also received, as part of the feature representation, a *remaining food map* indicating the location of remaining food items on the board. The remaining food map would allow the agent to quickly navigate between food items once it had learned to do so.

At the start of each episode, 3 new item types were generated with random quality and keys, and 15 items from this set were placed at random positions on the grid. An episode ended either when all food items were consumed or a fixed number of time-steps had elapsed (500 in our experiments). Ideally, to solve this task the agent would learn to store the quality indicator associated with an item after trying it once. On subsequent encounters with the same item, the agent could query the item in memory and use the quality indicator to decide whether to eat or discard it.

Note that associating a positive reward with each item regardless of its quality was necessary for the assumption, used in the derivation of OPGOR, that the state variable is highly predictive of $E_t[G_t]$ to hold. If instead, only eating a good item gave positive reward while eating a bad item gave negative reward, as in the task used by Wayne et al. (2018), the value of sitting on a good item and bad item would differ in a manner not observable from the immediate state variable alone. The central issue here is that if the state variable is not sufficient to provide a good estimate of $E_t[G_t]$, the agent may learn to remember state variables whose presence in the history is *predictive* of higher return, as opposed to state variables that *cause* higher return by conditioning the policy. In Section 6.5 we will explicitly examine how this can cause problems. Having a state variable which is in itself highly predictive of future return mitigates this distinction such that a greater than expected return is more likely due to the way the remembered state variable conditions the policy, not simply the fact that it occurred in the history. We emphasize that in a more realistic application, the state variable should be trained to be predictive of return, perhaps using a recurrent network, which would help to address this limitation. OPGOR-DS, introduced in Section 5.7 provides another possible solution by subtracting a sample of the denominator gradient from the importance weight gradient, which removes the need to subtract an accurate baseline from the return estimate.

An illustration of an instance of the rapid reward valuation problem is shown in Figure 6.12. Table 6.3 outlines the feature representation presented to the agent in each cell.

In the rapid reward valuation environment, we again tested agents based hard-OPGOR(λ) and soft-OPGOR(λ), along with each of the alternative selective memory strategies discussed in the previous section. Again, we used a 5 state memory for each OPGOR(λ) agent and each alternative in which a limited slot based memory was used. Parameters of the other agents were set in the same way discussed in the previous section with one exception; finding the useful step-size range to be drastically different from the keychain environment, we swept α from $\alpha \in \{2^{-i} : i \in \{6, ..., 11\}\}$. Once again, α was tuned to yield the highest average return over the final 100 episodes. Figure 6.13 show the results of this experiment.

Again, both hard-OPGOR(λ) and soft-OPGOR(λ) improved on the unweighted reservoir sampling agent, indicating both versions of OPGOR(λ) were

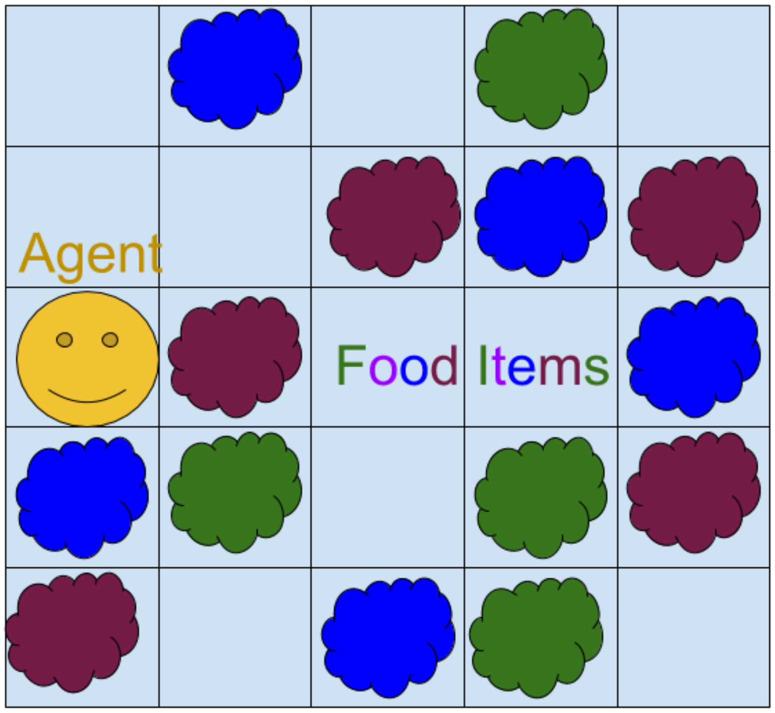


Figure 6.12: Visualization of rapid reward valuation environment.

Feature Name	Length	Meaning
Cell Index	grid width × grid height	Index of grid cell the agent currently occupies.
Food Type Indicator	5	Randomly generated key corresponding to food type in current cell, all zeros if no food in cell. Remains active one time-step after food is eaten. Normalized by dividing by half of the key length.
Remaining Food Map	grid width × grid height	Indicates grid cells in which food remains. Normalized by dividing by initial number of food items for stability.
Good Food Indicator	1	Indicates agent just ate or discarded good food.
Bad Food Indicator	1	Indicates agent just ate or discarded bad food.

Table 6.3: Feature representation for rapid reward valuation. Features are listed in the same order they appear in the state variable.

able to learn to generate useful importance weights. The hard-OPGOR(λ) agent and soft-OPGOR(λ) agnet performed similarly in terms of final performance. However, for the optimal α value, the soft query version showed some degradation after initial improvement. This behaviour was not observed in the hard query version.

From the performance of the hard and soft full memory agents, we can see that having access to the full history gives these agents a large advantage over the others, indicating the difficulty of the selective memory problem. Interestingly the hard query version of the full memory agent eventually exceeded the final performance of the soft memory agent in this case, though with much slower initial improvement. This perhaps indicates a trade off in which the precision of single item queries can be more stable in the long run after dealing with the initially noisier signal due to selecting state variables from memory stochastically. This could also explain the observed difference in behaviour observed between the hard-OPGOR(λ) and soft-OPGOR(λ) on this task.

The GRU agent was not very competitive in this case, with the truncation length of 20 showing fairly minor improvement over truncation length 5. We hypothesize that the main reason for the worse performance on this problem was due to the fairly precise comparison between state variables needed to determine the correct action associated with a food item. To determine the correct action for a given item it was necessary to compare the full five element key to that of another visited food item. If state variables are stored in full in memory they can be later recalled and compared to the current food item in a relatively straightforward manner. For a GRU to learn such an association from strictly recurrent updates is likely to be much more complex. This highlights another potential benefit of using external memory when a task requires precise relational reasoning between observations at different times.

The LRU agent was essentially indistinguishable from the hard version of the reservoir algorithm with which it shared a query method. Indicating that LRU, and by association other usage based strategies, may be a surprisingly viable selective memory strategy. Based on this result, and the considerations mentioned in the Section 6.1, it would be interesting to compare to a soft query version of LRU as well. This could perhaps be achieved by degrading the usage timer toward zero by a fraction equal to the associated query weight.

As with the keychain environment, we investigated the learned importance weights for rapid reward valuation by running an additional 1000 episodes with the trained hard and soft OPGOR(λ) agents and recording the importance weights generated for each observed state variable. The same importance weight normalization used in the keychain importance weight experiment was also used in this experiment. The results of this experiment are summarized in Figure 6.14.

The main thing to note in Figure 6.14 is that the largest importance weights are generally associated with the final two elements of the state variable which correspond to the good and bad indicator. Interestingly, for hard query OPGOR, the agents generally place significantly emphasis on one indicator over the other. This is actually a reasonable strategy here, as if a particular food is not remember the agent can simply assume it is associated with the less emphasized indicator. Either it will be correct and get +1 reward or it will be wrong and get -1 but gain an opportunity to add that item to memory to remember next time. Emphasizing both indicators is in some sense a waste of memory space.

6.4 Randomized Maze

So far we have evaluated OPGOR(λ), and various alternative selective memory strategies, on problems that were explicitly designed to test an agent's ability to recall details from specific past observations. A common real-world situation where memory is important is the task of navigation in a novel environment. Given a fixed environment and goal, a RL agent without memory could learn to navigate to the goal, assuming individual locations are distinct enough to be locally recognizable. However, when approaching a novel environment, the ability to selectively remember key route information could allow an agent to successfully navigate more efficiently than learning a memoryless policy from scratch. In this section, we will evaluate various memory strategies,

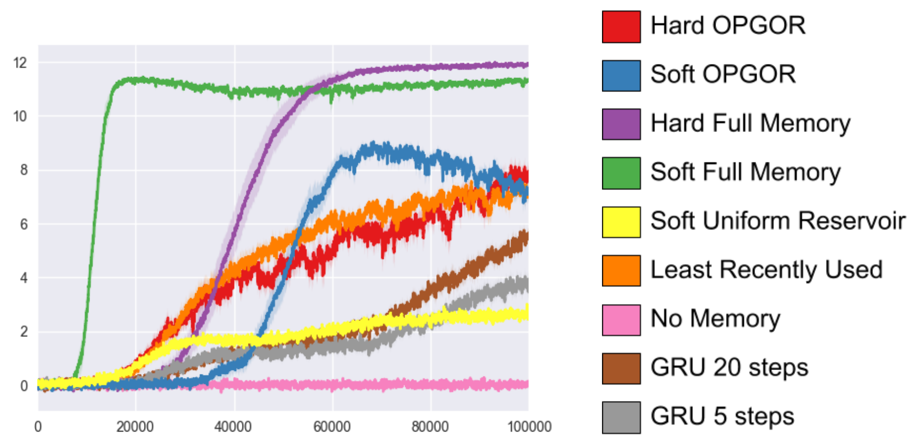
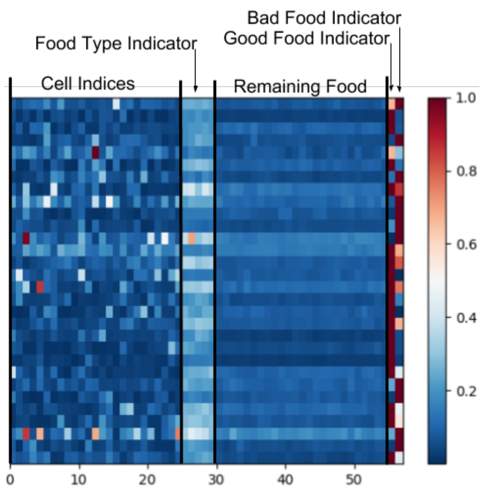


Figure 6.13: Comparison of returns v.s. training episodes for hard and soft OPGOR(λ) agents along with various alternatives on rapid reward valuation environment. The step-size parameter α is tuned separately for each agent by selecting for best average performance over the last 100 episodes over values from the set $\{2^{-i} : i \in \{6, 7, ..., 11\}\}$. All curves are smoothed with a running mean over 100 episodes and show an average of 30 runs with error bars showing standard error in the mean.

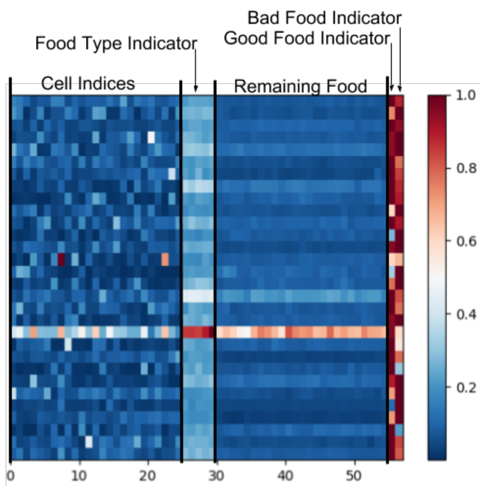
including hard and soft OPGOR(λ), on a simplified navigation task where the environment changes between episodes.

This environment was designed as a significantly simpler version of the goal-finding navigation task outlined by Wayne et al. (2018). The task consisted of an 8x5 grid world in which the agent received a reward of +1 every time it reached a goal cell, after which it was teleported to a new random location. This continued until a fixed number of steps (500 in our experiments) had elapsed at which point the episode terminated. The agent's path was impeded by two walls at fixed horizontal positions, each of which had a single hole, or *passage* the agent could pass through. The passage through each wall was located at a vertical position chosen randomly at the beginning of each episode. The action space consisted of moving in the four cardinal directions on the grid.

At the start of each episode the agent location, goal location and vertical position of the passage through each wall were randomly chosen. Passage location and goal location were then held fixed for the duration of the episode



(a) Importance Weights
(Hard Queries)



(b) Importance Weights
(Soft Queries)

Figure 6.14: Average, normalized, exponentiated importance weights associated with activation of various features for the hard and soft(λ) OPGOR(λ) agent on the rapid reward valuation environment. Average is taken over 1000 episodes generated after 100,000 episodes of training. The y-axis shows different independent training runs, the x-axis shows the different features of the state variable. Two max normalizations are applied. First, we divide the value for each feature observed in a given episode by the maximum importance weight value observed in the same episode, this is done because importance weights from different episodes will not affect the inclusion probability of a given state variable. Second, we divide the weights of all features by the max over all features for each training run separately, this is done for readability so the highest weighted feature over each run is set to one.

with only the agent location being randomized each time the goal was reached. If the agent could remember the location of the goal upon first reaching it, along with the locations of the passages in each wall, it could use this information to more quickly navigate back to the goal on subsequent trials within an episode. The state variable in this case consisted of a cell index unique to each location in the grid world, along with a *goal indicator* active when the goal was reached, *passage indicator* active when standing in a passage, and *goal near indicator*. The goal near indicator was active when the agent was in the same room as the goal, and was intended to approximate vision.

An illustration of an instance of the randomized maze problem is shown in Figure 6.15. Table 6.4 outlines the feature representation presented to the

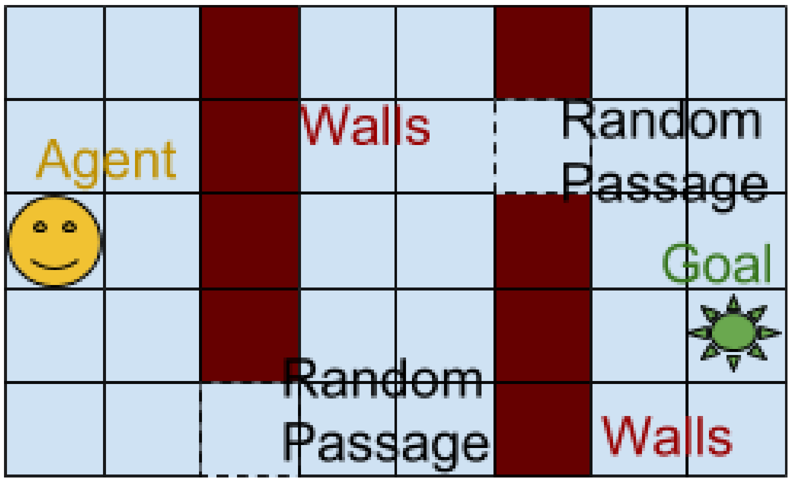


Figure 6.15: Visualization of randomized maze environment.

Feature Name	Length	Meaning
Cell Index	grid width × grid height	Index of grid cell the agent currently occupies.
Passage Indicator	1	Indicates that agent is currently standing in a passage (gap in a wall).
Goal Indicator	1	Indicates that agent has reached goal cell.
Goal Near	1	Indicates that agent is in same room as goal cell.

Table 6.4: Feature representation for randomized maze environment. Features are listed in the same order they appear in the state variable.

agent in each cell.

Note that this environment violates our assumption regarding the state variable providing enough information to estimate $E_t[G_t]$. The expected return in this case is highly dependent on the agent location relative to the goal, as well as the particular layout of passages and the goal, neither of which were available to the agent in the current state variable.

Once again, for the randomized maze environment, we tested agents using hard and soft OPGOR(λ) along with each of the alternative memory strategies discussed in Section 6.1. Again, we used a 5 state memory for the hard and soft

OPGOR(λ) agents, as well as each alternative in which a limited slot based memory was used. Parameters of the other agents were set as in Section 6.4 with α again tuned separately for each agent from $\alpha \in \{2^{-i} : i \in \{6, ..., 11\}\}$ to optimize performance over the final 100 training episodes. Figure 6.13 shows the results of this experiment.

In this case both hard and soft OPGOR(λ) failed to outperform the unweighted reservoir sampling agent. Initially, we hypothesized that the poor performance of OPGOR(λ) was due to the violation of the assumption that the state variable provides enough information to estimate $E_t[G_t]$. OPGOR-DS on the other hand does not require this assumption. We tested agents using hard-OPGOR-DS(λ) and soft-OPGOR-DS(λ) on this problem to see if they performed better.

Unfortunately, the hard and soft OPGOR-DS(λ) agents also failed to outperform the unweighted reservoir sampling agent. However, both hard and soft OPGOR-DS(λ) agents did appear to be stable at higher α than the other methods tested. In particular, the optimal α value in the set $\{2^{-i} : i \in \{6, ..., 11\}\}$ for both the hard and soft OPGOR-DS(λ) agent was the highest tested, 2^{-6}, and these algorithms still appeared to be showing improvement at the end of the training period. Motivated by this, we also tested OPGOR-DS(λ) with $\alpha = 2^{-5}$ which was found to yield similar final performance to $\alpha = 2^{-6}$ for both hard and soft OPGOR-DS(λ).

The failure of OPGOR-DS(λ) to outperform unwieghted reservoir sampling suggests that the weakness of OPGOR(λ) on this problem was not entirely due to the violation of our ideal critic assumption. Another hypothesis, to explain the relatively poor performance of OPGOR(λ) (and OPGOR-DS(λ)), is that the uniform reservoir sampling strategy may actually have some non-trivial advantages on this problem. With uniform sampling, each visit to the goal results in positive feedback, making it more likely that state variables in a uniform sample lie on a shortest path, which provides information that helps to reach the goal more quickly the next time. The simplicity and relative stability of this mechanism allows the query network to co-adapt over time. On the other hand, a mechanism like OPGOR(λ) quickly learns to greedily recall state variables with the goal and goal near indicator active. This is helpful at first, but provides less robust route information in the long run. For example, because of the inability of the the agents using OPGOR(λ) to account for when something has already been written to memory, the memory buffer can rapidly fill up with redundant copies of the goal location. This wastes space that could otherwise be used to store, for example, the location of passages. Nonetheless, it is possible OPGOR-DS(λ) would outperform uniform sampling given enough training episodes.

As expected the full memory agents significantly outperformed all the selective memory strategies. In this case we see a large advantage of the soft query version of the full memory agent over the hard query version.

The LRU agent outperformed the uniform reservoir agent, despite both versions of OPGOR(λ) failing to, giving further evidence of its relative robustness as a selective memory strategy.

The GRU agent also excelled at this problem, obtaining final performance even better than the hard full memory agent, and close to that of the soft full memory agent, albeit somewhat more slowly. This was true regardless of whether a truncation length of 5 or 20 was used. This indicates a significant ability to generalize over larger horizons than it was trained for, as it is common for there to be significantly more than 5 time-steps between subsequent visits to the goal (dividing the 500 time-steps in an episode by the final performance of around 50 gives an average of around 10, even at the end of training). It would be interesting to see how much this was due to the use of an eligibility trace, to do so one could also test a GRU with $\lambda = 0$ in future work. We also hypothesize that the GRU is particularly useful for the kind of short-term reasoning necessary for effective navigation. Specifically, we expect the GRU would be useful for things like remembering where you just came from so you don't go back immediately. For this reason it is likely that a combined system using external memory along with recurrent state updates, which is often how external memory systems are applied in practice, could do better on such tasks than either alone.

At this point it is worth briefly discussing the complementary role of eligibility traces and recurrent state updates in capturing temporal dependencies. Eligibility traces allow reinforcement of actions when the positive impact is only observable with some temporal delay, for example the choice to move toward the goal in the randomized maze task when the goal itself is only seen many time-steps later. They do not, however, facilitate conditioning that action on observations from several time-steps earlier, for example the last time the goal was observed, for this we require something like a recurrent update. The combination of the two means that as long as recalling information benefits action selection within the truncation length, the agent can learn to propagate it forward to subsequent time-steps outside the truncation length. In the randomized maze task in particular the agent can, for example, see the benefit of recalling the location of a goal observed within the truncation length, even if the goal is only revisited several steps later when the last observation of the goal is no longer within the truncation length. Remembering the location of the goal will continue to be useful at each step until it is reached, because remembering the goal allows the agent to continue to move toward it. Eligibility traces give credit to each action taken on the way, thus in principle, with a sufficiently large λ, the GRU agent can learn to propagate this information far past the truncation length

One could ask: what general feature allows the GRU with relatively short truncation length, but with eligibility traces, to perform so well on the random maze task? A plausible answer is that information which is important to remember for immediate decisions, such as the goal or passage location, is likely to remain important at latter steps. When this is true, optimizing recall for the recent past can easily allow the important information to be carried forward into the future over distances significantly larger than the truncation length. Note that this does not hold on the keychain task where we observe a significant negative impact on performance with reduced truncation length. In keychain, remembered information is only relevant near the end of the episode. Information must be propagated recurrently from arbitrary earlier time-steps without being useful for action selection at any intermediate step.

Despite OPGOR performing very poorly on the randomized maze problem,

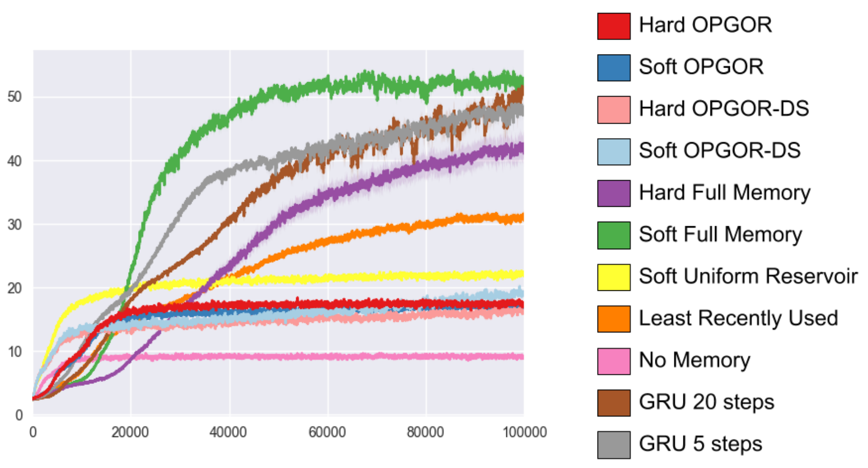
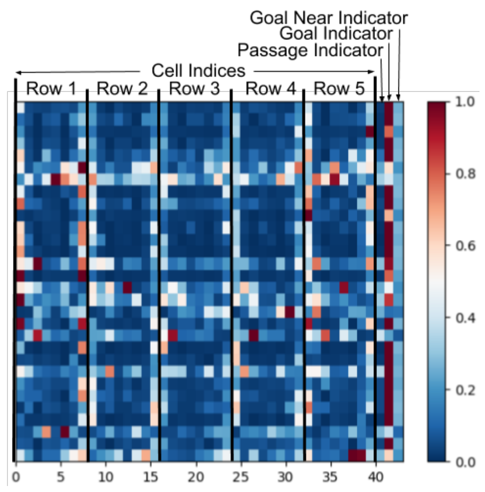


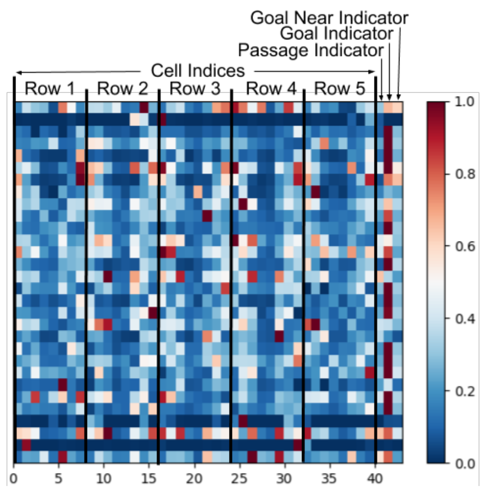
Figure 6.16: Comparison of returns v.s. training episodes for hard and soft OPGOR(λ) agents along with various alternatives on randomized maze environment. The step-size parameter α is tuned separately for each agent by selecting for best average performance over the last 100 episodes over values from the set $\{2^{-i} : i \in \{6, 7, ..., 11\}\}$. All curves are smoothed with a running mean over 100 episodes and show an average of 30 runs with error bars showing standard error in the mean.

we again investigated the learned importance weights to get a sense of what it learned. As before we ran an additional 1000 episodes with the trained hard and soft OPGOR(λ) agents and recorded the importance weights generated for each observed state variable. The results are summarized in Figure 6.15.

In both the hard and soft query case the goal indicator was often the most emphasized feature. This is more pronounced in the hard query case, which also shows a fairly clear secondary emphasis on the goal near indicator. This is intuitively reasonable as it is certainly useful to know where the goal is located in order to navigate toward it; failing that, it is also useful to know which room the goal is located in. In addition, in the hard query case, there appears to be a noticeable emphasis on cells on the far left and right side of the grid, for which we do not currently have a good hypothesis. Neither hard nor soft OPGOR(λ) seems to have learned to emphasize the passage indicator to any significant extent, despite its apparent utility in finding a path to the goal.



(a) Importance Weights
(Hard Queries)



(b) Importance Weights
(Soft Queries)

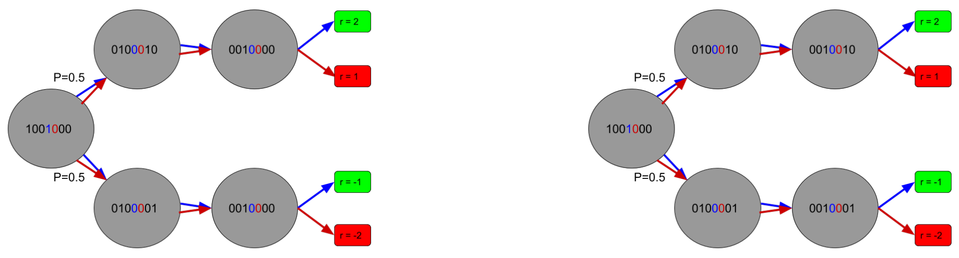
Figure 6.17: Average, normalized, exponentiated importance weights associated with activation of various features for the hard and soft OPGOR(λ) agent on the randomized maze environment. Average is taken over 1000 episodes generated after 100,000 episodes of training. The y-axis shows different independent training runs, the x-axis shows the different features of the state variable. Two max normalizations are applied. First, we divide the value for each feature observed in a given episode by the maximum weight value observed in the same episode, this is done because importance weights from different episodes will not affect the inclusion probability of a given state variable. Second, we divide the weights of all features by the max over all features for each training run separately, this is done for readability so the highest weighted feature over each run is set to one.

6.5 Simple Counterexample

In this section we present a very simple environment demonstrating an important limitation of OPGOR (without denominator sampling). Specifically, this example illustrates the reliance of OPGOR on a state representation which gives sufficient information to estimate the history conditional expected return $E_t[G_t]$ at each time. We tested one version of this environment where the state variable is always sufficient to predict $E_t[G_t]$. We then made a small change, making the state variable in the final cell insufficient to predict $E_t[G_t]$. We will see in the latter case OPGOR tended to emphasize retention of state variables which were predictive of high expected return but did not help to select the optimal action, which in this case led to catastrophic failure. On the other hand OPGOR-DS, the version of OPGOR augmented with denominator sampling, performs well even with the insufficient state representation. This *simple counterexample*, with and without sufficient state representation, is illustrated in Figure 6.18. Table 6.5 outlines the feature representation for the state variable presented to the agent in each cell.

An instance of the simple counterexample task always consisted of moving sequentially through three cells before terminating with a reward dependent on both the final action selected and which of two possible branches the agent was located on. The state variable of the starting cell contained an *action indicator*, indicating which action would give the highest reward in the final cell. Thus an optimal selective memory agent should remember the first state variable and use it to select its final action. After the first cell, the agent transitioned to either the top or bottom branch with uniform probability. The second cell contained a *branch identifier*, indicating which branch the agent had ended up on. The top branch contained higher rewards on termination, giving $+2$ for the optimal action and $+1$ for the suboptimal action. The bottom branch contained lower rewards, giving -1 for the optimal action and -2 for the suboptimal action. Finally, the agent transitioned deterministically to the third cell of whichever branch it was on. In the sufficient state variable case, the final cell also indicated which branch the agent was located on. In the insufficient state variable case the final cell contained no information about either the current branch or the optimal action. In addition, the state variable always included a *cell index* indicating which of the 3 steps in the problem the agent was currently on.

We tested a simple OPGOR(λ) agent with single-state memory on the simple counterexample. We will see that when the state variable was sufficient, OPGOR(λ) learned to remember the optimal action specified by the first cell. When the state variable was insufficient, however, the agent instead prioritized the second cell on the top branch, which was predictive of increased reward, but did not help the agent to attain higher reward than it would have otherwise. In the process, the retention probability of the first cell was dropped toward zero, causing OPGOR(λ) to perform no better than random. The results for



(a) Insufficient State Variable (b) Sufficient State Variable

Figure 6.18: Visualization of an instance of our simple counterexample with insufficient (a) and sufficient (b) state variables. In each example the first cell indicates the optimal action to take in the final cell. However, the magnitude reward the agent will receive will also depend on whether the first transitions moves it to the top or bottom branch. (a) is an example where the state variable is insufficient to predict history conditional expected return in the final cell. (b) is an example where the state variable is sufficient to predict expected return in all cells. Because of this OPGOR(λ) will work well in (b) but fail catastrophically in (a).

Feature Name	Length	Meaning
Cell Index	3	Node position the agent currently occupies (0, 1 or 2).
Action Indicator	2	Zero except in starting cell (position 0), where it indicates optimal final action.
Branch Identifier	2	Zero except for position 1 in insufficient state representation case and positions 1 and 2 in sufficient state representation case. When active it indicates whether the agent is on the top or bottom branch.

Table 6.5: Feature representation for simple counterexample environment. Features are listed in the same order they appear in the state variable.

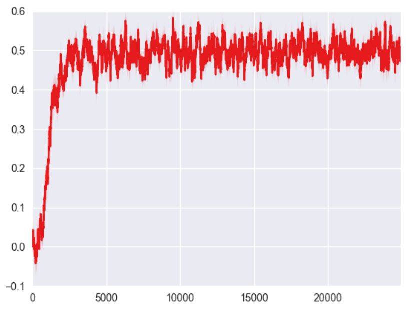
OPGOR(λ) in the sufficient and insufficient state variable cases are shown in Figure 6.19.

In Figure 6.19 (a), we see that when the state variable was sufficient, the OPGOR(λ) agent is able to reliably select the optimal action on both the top and bottom branch, reaching near the optimal expected return of 0.5.

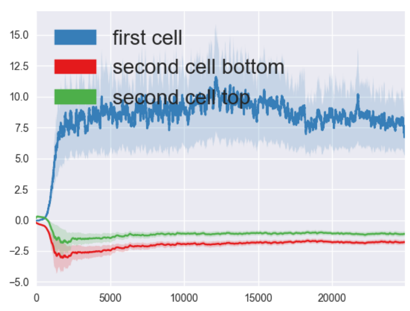
In Figure 6.19 (b), we see that this was a consequence of learning to retain the state variable of the first cell, which specified the optimal action, over the second cells which indicated which branch the agent was on. On the other hand, Figure 6.19 (c), shows the result when the state variable was insufficient in the last cell, meaning the last cell gave no indication of which branch the agent was located on. In this case the OPGOR(λ) agent never obtained significantly better than the expected return of 0.0 for a random agent. In Figure 6.19 (d), we see that this was a consequence of the first cell being deemphasized in favor of the second cell on the top branch which was predictive of high return, but useless for informing action selection. Despite the second cell on the bottom branch being predictive of low return it was not deemphasized to the extent of the first cell. This had the consequence that the emphasis on the second cell of the top branch destroyed the agents ability to select the correct action on both branches.

The failure of OPGOR when the state variable is insufficient stems from using $\hat{v}(\phi_t, \theta)$ as a baseline for OPGOR. When the obtained λ-return is higher than this baseline OPGOR will learn to emphasize the recalled state variable. In the example shown here the return tends to be higher when the second cell of the top branch is recalled simply because whenever the agent is on the top branch the return is guaranteed to be higher than the bottom branch. What we actually want is for the agent to remember things only if they lead to higher return by informing action selection. One solution to this is to use a baseline which is itself trained, perhaps using a RNN variant, to be highly predictive of return given the history. In that case only remembered state variables which facilitate better action selection than average will be able to improve on the baseline and be increasingly emphasized. Another solution is to use OPGOR-DS which corrects the policy gradient update by subtracting a sample from the denominator of the inclusion probability. In this case the gradient estimator does not require subtraction of a baseline at all, though we still subtract $\hat{v}(\phi_t, \theta)$ for variance reduction.

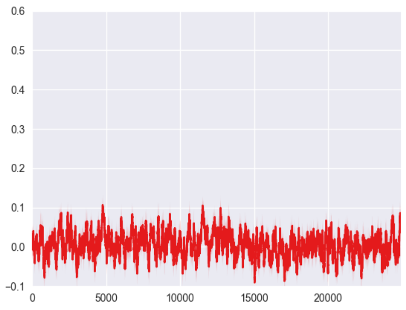
Keeping the same architecture and hyperparameters, we tested OPGOR-DS(λ) on the simple counterexample. We will see that with denominator



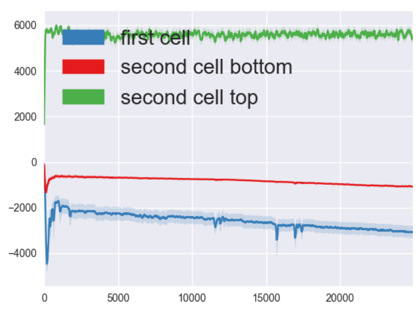
(a) Return
(Sufficient State Variable)



(b) Importance Weights
(Sufficient State Variable)



(c) Return
(Insufficient State Variable)



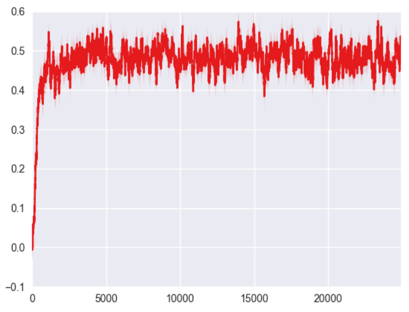
(d) Importance Weights
(Insufficient State Variable)

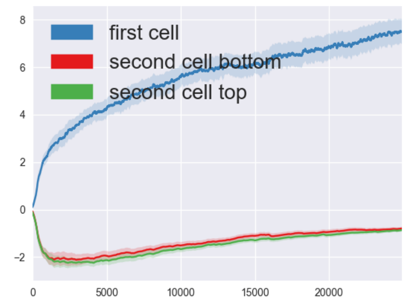
Figure 6.19: Results on simple counterexample with single-state memory OPGOR(λ) agent. The x-axis shows training episodes. (a,c) show return, (b,d) show average importance weight in the first cell, the second cell on the top branch, and the second cell on the bottom branch. The step-size parameter α is tuned separately for the sufficient and insufficient problems by selecting for best average performance over the last 100 episodes over values from the set $\{2^{-i} : i \in \{3, 4, ..., 8\}\}$. All curves are smoothed with a running mean over 100 episodes and show an average of 30 runs with error bars showing standard error in the mean.

sampling, the agent correctly learns to emphasize the first cell containing the action indicator, regardless of whether the state variable is sufficient for estimating $E_t[G_t]$. The results for OPGOR-DS(λ) in the sufficient and insufficient state variable cases are shown in Figure 6.20.

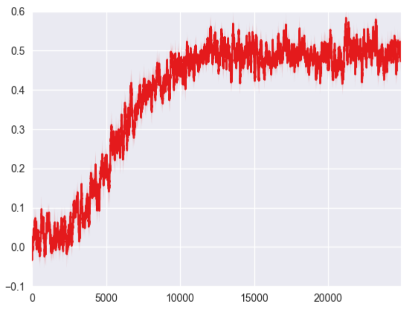
In Figure 6.20 (a), we see that when the state variable was sufficient, the OPGOR-DS(λ) agent is able to reliably select the optimal action on both the top and bottom branch, reaching near the optimal expected return of 0.5. In

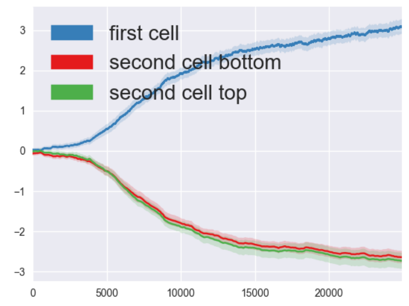
Figure 6.20 (b), we see that this was a consequence of learning to retain the state variable of the first cell, which specified the optimal action, over the second cells which informed which branch the agent was on. On the other hand, Figure 6.20 (c), shows the result when the state variable was insufficient in the last cell, meaning the last cell gave no indication of which branch the agent was located on. OPGOR-DS(λ) still reached near the optimal expected return of 0.5. In Figure 6.20 (d), we see that unlike OPGOR(λ), OPGOR-DS(λ) is able to correctly learn to emphasize the first cell while deemphasizing the other two, even with an insufficient state variable. The slower improvement in the insufficient state variable case is largely due to a lower optimal α value. However, even with the highest α value that achieved comparable performance, the insufficient state variable case still required somewhat longer to learn. Additionally with an insufficient state variable performance began to suffer significantly with $\alpha > 2^{-5}$, while with a sufficient state variable learning was stable up to $\alpha = 2^{-3}$.


(a) Return
(Sufficient State Variable)



(b) Importance Weights
(Sufficient State Variable)


(c) Return
(Insufficient State Variable)



(d) Importance Weights
(Insufficient State Variable)

Figure 6.20: Results on simple counterexample with single-state memory OPGOR-DS(λ) agent. The x-axis shows training episodes. (a,c) show return, (b,d) show average importance weight in the first cell, the second cell on the top branch, and the second cell on the bottom branch. The step-size parameter α is tuned separately for the sufficient and insufficient problems by selecting for best average performance over the last 100 episodes over values from the set $\{2^{-i} : i \in \{3, 4, ..., 8\}\}$. All curves are smoothed with a running mean over 100 episodes and show an average of 30 runs with error bars showing standard error in the mean.

Chapter 7

Conclusion

In this chapter, we summarize the contributions and insights presented in this thesis. We also provide directions for future work and improvements based on the insight gained.

7.1 Contributions and Insights

Our primary contribution was to present OPGOR, a new algorithm for selective memory in an online, partially observable, reinforcement learning setting with external memory. OPGOR makes use of the statistical technique of reservoir sampling, which enables us to select a weighted sample of n states from the full history, without ever storing more than n states. Using this capability, we are able to apply policy gradient to learn to assign importance weights that emphasize retaining states which are useful for future decision making, and dropping those which are not. We call this policy gradient procedure online policy gradient over a reservoir (OPGOR). Most of the prior work using external memory in a reinforcement learning setting uses either backpropagation through time, which is computationally intensive, or a heuristic mechanism for deciding what to retain in memory. This thesis represents a first step toward deriving mechanisms for this purpose which are both principled and computationally tractable for an online agent.

Beyond the specific mechanism presented here, we hope this thesis highlights reservoir sampling as an interesting general technique, which may be useful in various RL areas where it is necessary to sample a set of items from a large history (whether those items are state variables, transitions or something else). If one is not familiar with reservoir sampling it may be non-obvious that it is possible to draw weighted samples from a stream online in this manner. The fact that this is possible could allow for a variety of interesting algorithmic innovations, of which OPGOR is one example.

A secondary contribution was to evaluate several versions of OPGOR, along with a number of other selective memory strategies on a set of three simple environments designed to test an agent's ability to store and recall important information for future decisions. In the process we identify limitations of our proposed method which lead to directions for improvement. We also gain insight into the challenges involved in selective memory for online RL, and partial observability in general, and to what extend various techniques address these challenges.

We found that our original version of OPGOR, with both hard and soft queries, was able to learn useful importance weights in settings where the assumptions involved in its derivation held. In particular, when the state variable was adequate to produce a value function that was a good estimate of the expected return. In these cases, OPGOR was able to considerably improve on the unweighted reservoir sampling baseline. However, we demonstrate on a simple problem that, when this assumption is violated, the weight tuning may fail catastrophically. We identify the central issues to be that, if the state is not sufficient to provide a good estimate of history conditional expected return, the agent may learn to remember things that are merely *predictive* of higher return, as opposed to things that cause higher return by conditioning the policy.

To address this weakness, we derive another version of OPGOR, OPGOR-DS, which makes no assumption on the accuracy of the subtracted baseline. We show that as a consequence, OPGOR-DS succeeds on the simple counterexample where OPGOR fails.

Nonetheless, in the randomized maze problem, both OPGOR and OPGOR-DS performed worse than the uniform reservoir sampling baseline. We hypothesize that this is due in part to the simplicity of uniform sampling, which facilitates co-adaptation of the other components of the agent. Such co-adaptation is more difficult when the distribution of stored states is constantly changing due to learning of the importance weights.

During our evaluation we also found the least recently used heuristic to be surprisingly robust, even when the memory size is relatively small. This result warrants further evaluation to determine what allows this method, and by extension other usage-based heuristics, to perform well, where they might fail, and how they could be further improved. Once again, this suggests that there is some advantage to using a simple heuristic instead of an adaptive method, when a simple heuristic is sufficient. In particular, codependent learning processes can be difficult to synchronize, while a stable heuristic can be easily adapted to.

7.2 Future Work

Based on the insight gained from applying policy gradient to the selective memory problem, we suggest a number of avenues for future work. First we note that, in this work, we experiment only with toy examples, designed to illustrate certain aspects of the challenges of selective memory. The full complexity of the problem may be better illustrated with more realistic simulated environments, with high dimensional sensory spaces. In this case it would probably be necessary to use a learned state representation, which introduces its own set of challenges. On the theoretical side, we used a number of assumptions and approximations in our derivations, which would be good to clarify and better understand. Finally, we suggest a couple of tangential research directions, motivated by our interest in exploring applications of reservoir sampling to reinforcement learning, and by our desire to explore the space of selective memory strategies.

7.2.1 Learned State Representation

In realistic applications we would not want apply an external memory systems to store observation directly. This is limiting, and relies on individual observations providing sufficient context to condition later decisions. Individual observations are unlikely to be sufficient for this purpose in general, thus we would like to apply a mechanism like OPGOR with a learned state representation, perhaps generated by an RNN.

There may be challenges in applying OPGOR with learned state representations. We would need to learn query and write policies over states which are themselves changing over time, opening the possibility for difficult multiple timescale issues. This is made even more difficult if we continue to work within the online case, without utilizing experience replay or multiple parallel actors.

Assuming we can apply OPGOR with a learned state representation, using recalled sates only to condition the policy is still limiting. We would like to use these remember states in the state update itself, as done by Wayne et al. (2018). This would further help towards making the states stored in memory highly expressive, however again this presents some interesting challenges. In this case it would not make sense to optimize retention with only the advantage estimate, as we have done here. We also wish to utilizing remembered states to improve our value estimate, and perhaps other predictions. More work needs to be done to determine whether the framework can be modified to handle these additional objectives.

7.2.2 Assumptions and Approximations

Another avenue for improvement is in the derivation of OPGOR, where we made a number of assumptions and approximations which should be investigated theoretically and empirically, and perhaps improved upon. In the single state memory case, the primary approximation was to ignore the influence of past memory contents on the history, effectively treating the history as an arbitrary stream for the purposes of memory selection. This assumption is broken in RL control, because past decisions are conditioned on what is in memory at the time.

In the multiple state memory case we used this approximation as well as two others. Firstly, we ignore $\frac{\partial E_t[Q(S_k|S_t,\mathcal{M}_t,\theta)|S_k\in\mathcal{M}_t]}{\partial w_i}$ in the gradient estimate, optimizing importance weights only with respect to the term associated with whether a queried item is available in memory, and not whether it is actually queried. Secondly, we use $E_t[G_t|m_t \notin \Omega_t] \approx \hat{v}(S_t, \theta)$ in our gradient estimate for admissible items even though $\hat{v}(S_t, \theta)$ is trained instead to approximate $E_t[G_t]$. This second approximation could be easily relaxed by introducing a conditional value function approximation to predict $E_t[G_t|m_t \notin \Omega_t]$ directly. This approximation was made primarily for the sake of simplicity. Furthermore, this approximation is not limiting when OPGOR-DS is used. Unlike OPGOR, OPGOR-DS makes no assumption about the accuracy of the subtracted baseline. The first approximation however is more difficult to relax, as it is unclear how we could account for $\frac{\partial E_t[Q(S_k|S_t,\mathcal{M}_t,\theta)|S_k\in\mathcal{M}_t]}{\partial w_i}$ in our gradient estimates in an online manner.

7.2.3 Improved Understanding of Soft Query OPGOR

The soft query version of OPGOR is essentially a heuristic derived by analogy to the hard query version. It would be interesting to look into whether a more principled version could be derived for the case of soft queries, especially given that our results demonstrate they often perform better. One possibility is to produce a system which stores only a fixed set of states, but when queried produces an estimate of a differentiable query made over the full history. Aggarwal (2006) provides a possible direction to approach this, with a framework for using the contents of a reservoir to sample unbiased estimates of certain types of queries over a stream.

Also along these lines, one could ask whether it is possible to design a similar reservoir sampling based algorithm which uses soft writes as well as soft reads. Standard reservoir sampling techniques manage a set of discrete items, hence it is non-obvious how they could be applied with soft reads and writes. Note, however, that it may still be possible to apply an algorithm like OPGOR to stochastically select a subset of states for which to modify importance weights, without actually applying it to select a discrete set to store in memory.

7.2.4 Additive Networks with Stochastically Sampled Updates

The recurrent structure of RNNs means it is not possible to look at the contribution of a specific observation to the current hidden state without accounting for subsequent observations. If instead each observation was incorporated into the hidden state in a purely additive fashion (i.e., by summing embeddings that are each a learned function of the associated observations) we could look at the gradient of the hidden state with respect to each update in isolation. See the work of Garnelo et al. (2018) for an example of an architecture based on such additive integration of state information.

Within such an additive architecture, we could select only a subset of observations from the history and update their contribution to the hidden state independently of the others. Such a selection could for example be handled online with a reservoir sampling algorithm. By scaling the gradients by the inverse of the inclusion probability, we could then obtain unbiased gradient estimates while only ever storing a fixed-size subset of observations at a time. This could be an interesting mechanism to explore, and may provide another valuable application of reservoir sampling for handling partial observability in an online manner.

A similar learning algorithm could be implemented for an eligibility trace like structure, maintaining a weighted sum of recently viewed states, perhaps with learned, discounted weights. This would allow learning of a summary of only recent observations, as opposed to an order independent summary of the entire observation history.

7.2.5 Value Based Selective Memory

Having explored policy gradient methods for selective memory, it would be interesting to look at the analogy to action value methods. In particular one could assign a value to each state variable equal to its estimated future value added. That is, how much having that state variable in memory will improve expected return compared to not having it. This could be estimated, for example, by maintaining a prior value function before recalling a state variable from memory, along with a posterior value function given the recalled state variable. The difference between these two value would contribute to the value of the recalled state variable. For now, we leave open the question of exactly how this system would be trained. An interesting advantage of such a mechanism is that it could not only ranking the utility of state variables in memory, but could also credit the agent for actively seeking high value state variables. This latter usage is similar to the idea of *temporal value transport* from work by Hung et al. (2018).

7.3 Summary

We have demonstrated how reservoir sampling may be combined with policy gradient to yield OPGOR, a new approach to deciding what to write to and retain in an external memory in an online RL setting. External memory mechanisms are an area of growing interest for handling partial observability in RL. Our empirical results reveal that while OPGOR is capable of learning a good selective memory policy in ideal circumstances, it is currently limited in other problems of interest. In addition we found that a simple heuristic, dropping the least recently used, performs similarly to, or better than OPGOR in many cases. This is an intriguing result and opens questions of why such heuristics perform so well, as well as where they might fail.

We are encouraged by these early results to further explore the potential of reservoir sampling based algorithms to help deal with partial observability in an online fashion. More generally we are interested in exploring the space of potential strategies for selective memory in an online RL setting.

References

Aggarwal, C. C. (2006). On biased reservoir sampling in the presence of stream evolution. In *Proceedings of the thirty second international conference on very large data bases (VLDB)* (pp. 607–618).

Bakker, B., Zhumatiy, V., Gruener, G., & Schmidhuber, J. (2003). A robot that reinforcement-learns to identify and memorize important previous observations. In *Proceedings of the international conference on intelligent robots and systems (IROS).*

Chao, M.-T. (1982). A general purpose unequal probability sampling plan. *Biometrika*, *69*(3), 653–656.

Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: encoder-decoder approaches. In *Proceedings of the eighth workshop on syntax, semantics and structure in statistical translation.*

Degris, T., Pilarski, P. M., & Sutton, R. S. (2012). Model-free reinforcement learning with continuous action in practice. In *Proceedings of the american control conference (ACC)* (pp. 2177–2182). IEEE.

Efraimidis, P. S. & Spirakis, P. G. (2006). Weighted random sampling with a reservoir. *Information Processing Letters*, *97*(5), 181–185.

Elfwing, S., Uchibe, E., & Doya, K. (2018). Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks*, *107*, 3–11.

Garnelo, M., Rosenbaum, D., Maddison, C. J., Ramalho, T., Saxton, D., Shanahan, M., … Eslami, S. (2018). Conditional neural processes. *arXiv preprint arXiv:1807.01613.*

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401.*

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., … Agapiou, J., et al. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, *538*(7626), 471–476.

Gulcehre, C., Chandar, S., & Bengio, Y. (2017). Memory augmented neural networks with wormhole connections. *arXiv preprint arXiv:1701.08718.*

Gulcehre, C., Chandar, S., Cho, K., & Bengio, Y. (2018). Dynamic neural turing machine with continuous and discrete addressing schemes. *Neural computation*, *30*(4), 857–884.

Hausknecht, M. & Stone, P. (2015). Deep recurrent Q-learning for partially observable MDPS. In *Proceedings of the AAAI fall symposium on sequential decision making for intelligent agents*.

Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Hung, C.-C., Lillicrap, T., Abramson, J., Wu, Y., Mirza, M., Carnevale, F., … Wayne, G. (2018). Optimizing agent behavior over long time scales by transporting value. *arXiv preprint arXiv:1810.06721*.

Joulin, A. & Mikolov, T. (2015). Inferring algorithmic patterns with stack-augmented recurrent nets. *Advances in neural information processing systems 28 (NIPS)*.

Kaiser, Ł., Nachum, O., Roy, A., & Bengio, S. (2017). Learning to remember rare events. In *Proceedings of the fifth international conference on representation learning (ICLR)*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., … Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *Proceedings of the thirty third international conference on machine learning (ICML)*.

Oh, J., Chockalingam, V., Singh, S. P., & Lee, H. (2016). Control of memory, active perception, and action in minecraft. In *Proceedings of the thirty third international conference on machine learning (ICML)*.

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Sukhbaatar, S., Szlam, A., Weston, J., & Fergus, R. (2015). End-to-end memory networks. *Advances in neural information processing systems 28 (NIPS)*.

Sutton, R. S. & Barto, A. G. (2017). *Reinforcement learning: an introduction*. MIT press Cambridge (Draft).

Sutton, R. S., McAllester, D. A., Singh, S. P., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems 13 (NIPS)*.

Vitter, J. S. (1985). Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, *11*(1), 37–57.

Wayne, G., Hung, C.-C., Amos, D., Mirza, M., Ahuja, A., Grabska-Barwinska, A., … Lillicrap, T. (2018). Unsupervised predictive memory in a goal-directed agent. *arXiv preprint arXiv:1803.10760*.

Wierstra, D., Förster, A., Peters, J., & Schmidhuber, J. (2010). Recurrent policy gradients. *Logic Journal of IGPL*, *18*(5), 620–634.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, *8*(3-4), 229–256.

Young, K. J., Sutton, R. S., & Yang, S. (2018). Integrating episodic memory into a reinforcement learning agent using reservoir sampling. *arXiv preprint arXiv:1806.00540*.

Zaremba, W. & Sutskever, I. (2015). Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*.

Zhang, M., Levine, S., McCarthy, Z., Finn, C., & Abbeel, P. (2016). Policy learning with continuous memory states for partially observed robotic control. In *Proceedings of the international conference on robotics and automation (ICRA)*.