Learning What to Remember with Online Policy Gradient Over a Reservoir

Kenny Young, Richard S. Sutton
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
{kjyoung,rsutton}@ualberta.ca

External memory has become an important technique for handling partial observability in reinforcement learning. However, deciding what to store in memory is often approached with heuristic techniques. We introduce *online policy gradient over a reservoir* (OPGOR) to learn to favor retention of useful state variables, in a way that is tractable for an online agent. We utilize *reservoir sampling* (Chao, 1982) to maintain a weighted sample of state variables from the full history, while only storing a fixed-size buffer. Nevertheless, the policy gradient is still a function of the full history. In order to perform policy gradient without actually storing the full history, we introduce a novel sampling technique.

Consider an actor-critic agent whose policy has access to an external memory consisting of n stored *state variables* from previous time-steps. A state variable is a feature vector ϕ_t generated at each time-step which acts as the agent's state representation. For simplicity, consider the case where $n = 1$, i.e. the memory contains a single past state variable m_t. At each time-step the remembered state variable, $m_t = \phi_i$ for some $i < t$, will be used, alongside the current state variable ϕ_t, to condition the policy $\pi(A_t|\phi_t, m_t)$. This allows the agent to condition its decisions on past events. Assume m_t is drawn from a distribution parameterized as follows by a set of *importance weights* $\{w(\phi_i)|i \in \{0, ..., t-1\}\}$, where $w(\phi_i)$ is generated by a *write network* $w(\cdot)$ when the associated state is visited:

$$P_t(m_t = \phi_i) = \exp(w(\phi_i)) \Big/ \sum_{j=0}^{t-1} \exp(w(\phi_j)) \tag{1}$$

Naively sampling from this distribution at each time-step would require us to store the full history, and then draw a sample. This is not feasible for an online agent, and eliminates much of the computational advantage of using a limited memory. Reservoir sampling allows us to maintain such a sample online, while storing only a single state variable. Reservoir sampling works by carefully choosing the probability of adding each observed state variable, and the probability of dropping state variables from memory, such that the marginal inclusion probability for each state variable has a desired form. The reservoir sampling algorithm we use, including the more complex multiple item case, was originally described by Chao (1982), see Appendix A for a review of this algorithm.

The expected return conditioned on the history of state variables up to time t can be written:

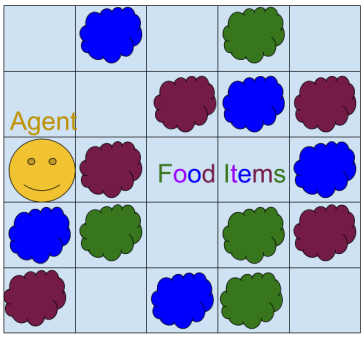
$$E_t[G_t] = \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i) E_t[G_t|A_t = a] \tag{2}$$

We can write the gradient of the return with respect to the parameters θ_w of the write network as a sum of terms of the following form at each time-step:

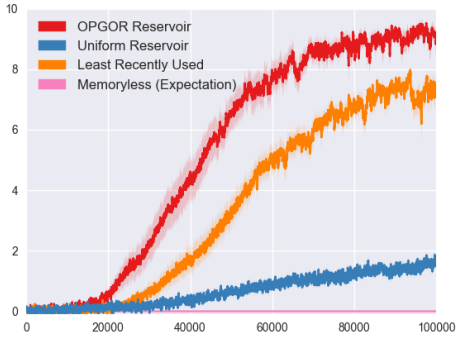
$$\sum_{i=0}^{t-1} P_t(m_t = \phi_i) \left(\frac{\partial w(\phi_i)}{\partial \theta_w} - \sum_{j=0}^{t-1} P_t(m_t = \phi_j) \frac{\partial w(\phi_j)}{\partial \theta_w} \right) \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i) E_t[G_t|A_t = a] \tag{3}$$

Note that this is essentially identical to the derivation of ordinary policy gradient (Sutton, McAllester, Singh, & Mansour, 2000), applied to state variable selection, except that we have explicitly expanded:

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.



(a) Rapid Action Association



(b) Return v.s. Episodes

Figure 1: (a) Diagram illustrating the rapid action association problem. (b) Returns over time for OPGOR and 2 baseline agents. Each curve is the average of 30 runs. The α value for each agent was tuned from $\{2^{-i} : i \in \{5, 6, ..., 10\}\}$

.

$$\frac{\partial \log(P_t(m_t = \phi_i))}{\partial \theta_w} = \frac{\partial w(\phi_i)}{\partial \theta_w} - \sum_{j=0}^{t-1} P_t(m_t = \phi_j) \frac{\partial w(\phi_j)}{\partial \theta_w} \tag{4}$$

as a difference of terms corresponding to the numerator and denominator of equation 1. While the numerator term is easily computed as the gradient of the write network for the state variable stored in memory, to naively compute the gradient of the denominator term would require storing the entire history of state variables, which is not feasible in an online setting. We can instead sample an unbiased estimate the denominator term online, using a second reservoir, sampled identically but independently to m_t. The state variable in the first reservoir m_t conditions the policy. The state variable in the second reservoir, say $\tilde{m}_t$, is an independent sample from the same distribution, which allows us to compute an unbiased sample of the denominator gradient.

Appendix B provides a detailed derivation of OPGOR, including how we extend it to the multiple-state memory case. In the simplest case, using a single-state memory and one step advantage estimation, OPGOR consists of the following update rule for the write network parameters $\theta_{w,t}$ at each time-step:

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

In Figure 1, we show the results of an experiment comparing this approach against two baselines on a *rapid action association* problem, designed to test an agent's ability to learn to recall salient past state variables in order to inform present decisions. In this problem, the agent wandered a grid world filled with food items. Each item had an associated type which was specified by a random 5 bit key, which formed part of the state variable when the agent was collocated with the item. For each type, one of two special actions gave reward $+1$ while the other gave reward -1. This information was only made available after committing to an action for a given item. To succeed the agent had to remember the outcome of acting on past items of matching type within an episode and use this information to select the positive reward action for the current item. Item types were randomized each episode. See Appendix D for a full description of this environment.

The uniform reservoir baseline fixed the importance weights used in reservoir sampling to one, thus sampling uniformly from the history. The least recently used baseline was a strong heuristic, which droped the state variable which had been least recently written or queried at each time-step. We also show the expected performance for a memoryless agent, which is fixed at zero. See Appendix E for further details of our experiment setup.

We suggest OPGOR represents the first step towards a principled, online approach to a problem which is often approached using various heuristics (e.g. Gulcehre, Chandar, and Bengio (2017), Gulcehre, Chandar, Cho, and Bengio (2018), Oh, Chockalingam, Singh, and Lee (2016), Wayne et al. (2018)), and/or backpropagation through time (e.g. Graves et al. (2016)), which is not tractable for an online agent.

References

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

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.

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

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.

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*.

Sutton, R. S., McAllester, D. A., Singh, S. P., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Neural information processing systems (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., ... Santoro, A., et al. (2018). Unsupervised predictive memory in a goal-directed agent. *arXiv preprint arXiv:1803.10760*.

A Reservoir Sampling Overview

Here, we provide the relevant background on reservoir sampling, which we apply to manage the external memory of a reinforcement learning agent.

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 neural network 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. ϕ_t

is added with $P = n/t$. If ϕ_t is added, each item is equally likely to be replaced. As a result, 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 work, we use of 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:

$$\Omega_t = \underset{\omega \subset [0,..,t-1]}{\arg\min} |\omega| \text{ s.t. } \forall i \in \{0,..,t-1\} \setminus \omega,\ (n - |\omega|)w_i < \sum_{j \in \{0,..,t\} \setminus \omega} w_j \tag{5}$$

i.e., those indices for which the weight is too large to represent a relative probability of inclusion. 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:

$$P(\phi_i \in \mathcal{M}_t) = \begin{cases} 1 & \text{if } i \in \Omega_t \\ \frac{(n - |\Omega_t|)w_i}{\sum_{j \notin \Omega_t} w_j} & \text{if } i \notin \Omega_t \end{cases} \tag{6}$$

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 6. 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

we will refer to this procedure as *Chao sampling*, after the author who originally described it. We omit the proof here for brevity, but note that this procedure gives rise to precisely the inclusion probability specified in equation 6 for each item at each time. In the single-item case this procedure is particularly simple, as there is no possibility of inadmissible items. In that case Chao sampling gives rise to the probability given by equation 1, except that the weights have been exponentiated. We use exponentiated weights because it simplifies the gradient calculation, and enforces positive probability for arbitrary real numbers.

B Details of Online Policy Gradient Over a Reservoir

Here we provide a detailed derivation of the OPGOR algorithm. We will first derive OPGOR for the simple case of a memory that can hold only a single state variable. We will then describe how we extend the algorithm to the more complex case of multiple state variables, for which we employ a *semi-gradient* approximation.

B.1 Single-State Memory

We will begin our derivation of OPGOR for the single-state memory case from the following expression for the gradient of equation 2, the expected return, with respect to the write network parameters θ_w:

$$\frac{\partial}{\partial \theta_w} E_t[G_t] = \sum_{i=0}^{t-1} \left(\frac{\partial P_t(m_t = \phi_i)}{\partial \theta_w} \sum_{a \in \mathcal{A}} (\pi(a|\phi_t, \phi_i) E_t[G_t|A_t = a] - \hat{v}(\phi_t)) \right.$$
$$\left. + P_t(m_t = \phi_i) \sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i) \frac{\partial E_t[G_t|A_t = a]}{\partial \theta_w} \right) \quad (7)$$

We subtract a baseline $\hat{v}(\phi_t)$ for variance reduction. Working out the first term from the right hand side of equation 7:

$$\sum_{i=0}^{t-1} \frac{\partial P_t(m_t = \phi_i)}{\partial \theta_w} \left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i) E_t[G_t|A_t = a] - \hat{v}(\phi_t) \right)$$
$$= \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial \log(P_t(m_t = \phi_i))}{\partial \theta_w}$$
$$\left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i) E_t[G_t|A_t = a] - \hat{v}(\phi_t) \right)$$
$$= \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \frac{\partial}{\partial \theta_w} \left(w_i - \log(\sum_{j=0}^{t-1} \exp(w_j)) \right)$$
$$\left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i) E_t[G_t|A_t = a] - \hat{v}(\phi_t) \right)$$
$$= \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \left(\frac{\partial w_i}{\partial \theta_w} - \sum_{j=0}^{t-1} P_t(m_t = \phi_j) \frac{\partial w_j}{\partial \theta_w} \right)$$
$$\left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i) E_t[G_t|A_t = a] - \hat{v}(\phi_t) \right)$$

Working out the second term from the right hand side of equation 7:

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

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:

$$\frac{\partial}{\partial \theta_w} E_t[G_t] = \sum_{i=0}^{t-1} P_t(m_t = \phi_i) \left(\frac{\partial w_i}{\partial \theta_w} - \sum_{j=0}^{t-1} P_t(m_t = \phi_j) \frac{\partial w_j}{\partial \theta_w} \right) \left(\sum_{a \in \mathcal{A}} \pi(a|\phi_t, \phi_i) E_t[G_t|A_t = a] - \hat{v}(\phi_t) \right) + E_t \left[\frac{\partial}{\partial \theta_w} E_{t+1}[G_{t+1}] \right] \quad (8)$$

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

$$\frac{\partial \log(P_t(m_t = \phi_i))}{\partial \theta_w} = \left(\frac{\partial w_i}{\partial \theta_w} - \sum_{j=0}^{t-1} P_t(m_t = \phi_j) \frac{\partial w_j}{\partial \theta_w} \right)$$

in order to emphasize the separate numerator term $\frac{\partial w_k}{\partial \theta_w}$, and denominator term $\sum_{j=0}^{t-1} P_t(m_t = \phi_j) \frac{\partial w_j}{\partial \theta_w}$. Each of these terms can be estimated by maintaining a single-item Chao sample of the state variable associated with each weight, and computing the gradient only with respect to the selected item. For the numerator term the sampled state variable m_t will also be used to condition the policy, while the denominator sample $\tilde{m}_t$ will be independently sampled from the same distribution. 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 sample based estimate of equation 8 at $t = 0$:

$$\frac{\partial}{\partial \theta_w} E_0[G_0] \approx \sum_{t=0}^{\infty} \left(\frac{\partial w(m_t)}{\partial \theta_w} - \frac{\partial w(\tilde{m}_t)}{\partial \theta_w} \right) \left(\hat{G}_t - \hat{v}(\phi_t) \right) \quad (9)$$

Where $\hat{G}_t$ is some estimate of the return. In this work we apply online *generalized advantage estimation* (Schulman, Moritz, Levine, Jordan, & Abbeel, 2015) for this purpose, taking $\hat{G}_t$ to be equal to the λ-return:

$$\begin{aligned} G_t^\lambda &= R_{t+1} + \gamma \left((1 - \lambda) \hat{v}(\phi_{t+1}) + \lambda G_{t+1}^\lambda \right) \\ &= \hat{v}(\phi_t) + \sum_{k=t}^{\infty} (\gamma \lambda)^{k-t} \delta_k \end{aligned}$$

where we have defined the TD-error:

$$\delta_t = R_t + \gamma \hat{v}(\phi_{t+1}) - \hat{v}(\phi_t)$$

Thus, our gradient estimate becomes:

$$\begin{aligned} \frac{\partial}{\partial \theta_w} E_0[G_0] &\approx \sum_{t=0}^{\infty} \left(\frac{\partial w(m_t)}{\partial \theta_w} - \frac{\partial w(\tilde{m}_t)}{\partial \theta_w} \right) \sum_{k=t}^{\infty} (\gamma \lambda)^{k-t} \delta_k \\ &= \sum_{t=0}^{\infty} \delta_t \sum_{k=0}^{t} (\gamma \lambda)^{t-k} \left(\frac{\partial w(m_k)}{\partial \theta_w} - \frac{\partial w(\tilde{m}_k)}{\partial \theta_w} \right) \end{aligned}$$

Now define an eligibility trace:

$$z_{w,t} = \sum_{k=0}^{t} (\gamma \lambda)^{t-k} \left(\frac{\partial w(m_k)}{\partial \theta_w} - \frac{\partial w(\tilde{m}_k)}{\partial \theta_w} \right)$$

such that:

$$\frac{\partial}{\partial \theta_w} E_0[G_0] \approx \sum_{t=0}^{\infty} \delta_t z_t$$

We use online updating, meaning we update the parameters of the write network at each time-step according to a single term in the above sum as follows:

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

B.2 Multiple-State Memory

In the multiple-state case, the memory will be managed by the full, multiple-item, version of Chao sampling, which we reviewed in Appendix A. Instead of just one state variable m_t stored in memory at each time we will now have a reservoir $\mathcal{M}_t = (\mathcal{M}_t(0), ..., \mathcal{M}_t(n-1))$ 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:

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

To compute a gradient estimate for the multiple-state case, we will make a simplifying approximation, using a form of semi-gradient, and take $\frac{\partial E_t[Q(\phi_k|\phi_t,\mathcal{M}_t)|\phi_k\in\mathcal{M}_t]}{\partial w_i} \approx 0$ for all k and i. Note that in reality since $Q(\phi_k|\phi_t, \mathcal{M}_t)$ 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 $E_t[Q(\phi_k|\phi_t, \mathcal{M}_t)|\phi_k \in \mathcal{M}_t]$ as constant with respect to the importance weights w_i will adequately capture the primary effect of the write process, while $E_t[Q(\phi_k|\phi_t, \mathcal{M}_t)|\phi_k \in \mathcal{M}_t]$ 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:

$$
\begin{aligned}
\frac{\partial}{\partial\theta_w} E_t[G_t] \approx \sum_{i=0}^{t-1} \Bigg(& \frac{\partial P_t(\phi_i \in \mathcal{M}_t)}{\partial\theta_w} E_t[Q(\phi_i|\phi_t, \mathcal{M}_t)|\phi_i \in \mathcal{M}_t] \\
& \left(\sum_{a\in\mathcal{A}} \pi(a|\phi_t, \phi_i) E_t[G_t|A_t = a] - \hat{v}(\phi_t)\right) \\
& + P_t(m_t = \phi_i) \sum_{a\in\mathcal{A}} \pi(a|\phi_t, \phi_i) \frac{\partial E_t[G_t|A_t = a]}{\partial\theta_w}\Bigg) \qquad (10)
\end{aligned}
$$

Again, we have subtracted a constant baseline of $\hat{v}(\phi_t)$. Modifying equation 6 slightly to use exponentiated weights, the inclusion probabilities for each state variable will be given by:

$$
P_t(\phi_i \in \mathcal{M}_t) = \begin{cases} 1 & \text{if } i \in \Omega_t \\ \frac{n\exp(w_i)}{\sum_{j\in\Omega_t}\exp(w_j)} & \text{if } i \notin \Omega_t \end{cases} \qquad (11)
$$

where:

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

is the inadmissible set. Recall from Appendix A 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 10 gives, for all $i \notin \Omega_t$:

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

Note that the term $\frac{\exp(w_j)}{\sum_{k \notin \Omega_t} \exp(w_k)}$ is precisely the single-item probability given in equation 1, except that the summation is now restricted to state variables not in Ω_t. Hence, even though we are using a multiple-state memory, we can still use a single-state reservoir with single-item Chao sampling to sample the denominator gradient, now sampling only from items which are admissible in $\mathcal{M}_t$. We can also use a larger denominator reservoir $\tilde{\mathcal{M}}_t = (\tilde{\mathcal{M}}_t(0), ..., \tilde{\mathcal{M}}_t(\tilde{n}-1))$ with multiple-item Chao sampling to get a lower variance estimate of the denominator gradient. Inclusion probabilities under mutliple-item Chao sampling, given by equation 11, are simply a multiple of the associated single-item inclusion probabilities as long as all items are admissible. Hence, as long as the denominator reservoir is smaller than, or the same size as the external memory ($\tilde{n} \leq n$), we can simply take the average gradient of the items in the denominator reservoir as our estimated denominator gradient. If the denominator reservoir is larger it will be necessary to re-weight any inadmissible items appropriately. We do, however, have to be careful to sample from only admissible state variables, which can be accomplished by only streaming state variables to $\tilde{\mathcal{M}}_t$ when they first become admissible in $\mathcal{M}_t$.

With the approximation we applied, the multiple-state memory version of OPGOR works out very similarly to the single-state memory version discussed in Section 7. 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 10 works out the same as the second term of equation 8 and no modification to its derivation is necessary. We also modify the update equations to account for a larger denominator reservoir, but limit it to the case where the denominator reservoir is no larger than the external memory for simplicity. The associated online

update rule becomes:

$$z_{w,0} = 0$$

$$z_{w,t} = \gamma\lambda z_{w,t-1} + \mathbb{1}(m_t \notin \Omega_t)\left(\frac{\partial w(m_t)}{\partial \theta_w} - \frac{1}{\tilde{n}}\sum_{i=0}^{\tilde{n}-1}\frac{\partial w(\tilde{\mathcal{M}}_t(i))}{\partial \theta_w}\right)$$

$$\theta_{w,t+1} = \theta_{w,t} + \alpha\delta_t z_t$$

C Algorithmic and Architectural details

We build our 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 $\mathcal{W}_t(i)$. Following Wayne et al. (2018), we refer to the vectors stored in memory as state variables. The state variable generated at time t will be a vector of fixed size, denoted by ϕ_t. In this work, for the sake of simplicity, ϕ_t is generated directly by the environment, in general we would want to use some learned state representation.

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 parameter θ represents the full parameter vector of the agent. In this work, each of the discussed network modules will be independently parameterized, however in general some of the parameters in θ may be shared. The importance weights will be used to manage the external memory via the Chao sampling algorithm described in Appendix A, and adjusted via OPGOR.

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:

$$Q(\mathcal{M}_t(i)|\phi_t, \mathcal{M}_t, \theta) = \exp\left(\langle q(\phi_t, \theta)|\mathcal{M}_t(i)\rangle\,\beta\right) \Big/ \sum_{j=0}^{n-1}\exp\left(\langle q(\phi_t, \theta)|\mathcal{M}_t(j)\rangle\,\beta\right) \tag{12}$$

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 2.

We used a single hidden layer neural network 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 neural networks for online RL with eligibility traces, as we will do in this work.

We train our agent using a variant of actor-critic with eligibility traces, AC(λ) Degris, Pilarski, and Sutton, 2012. An online variant of actor-critic which utilizes eligibility traces in both the actor and the critic. The basic AC(λ) algorithm can be written as follows:

$$z_0 = 0$$

$$z_t = \gamma\lambda z_{t-1} + \frac{\partial \hat{v}(S_t, \theta_t)}{\partial \theta} + \frac{1}{2}\frac{\partial \log(\pi\left(A_t|S_t, \theta_t\right))}{\partial \theta}$$

$$\theta_{t+1} = \theta_t + \alpha z_t \delta_t$$

Note that the trace z_t contains a term for the actor update and another for the critic. In addition to this basic algorithm we add in a term for the query network, similar to another actor term where the queried item is treated as an action. We also add a term for the write network which uses OPGOR, as described in Appendix B. We applied entropy regularization to the policy, as well as L2 regularization

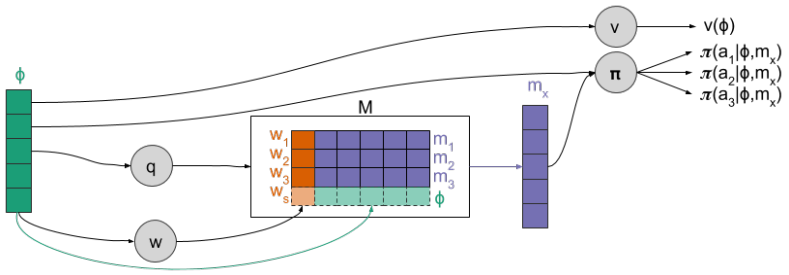


Figure 2: Advantage actor-critic with external memory conditioned policy architecture. Each grey circle represents an neural network module. The state variable (ϕ) is provided as input to the query (q), write (w), value (v) and policy (π) networks at each time-step. The query network outputs a vector, equal in length to the state variable, which is used (via equation 12) 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.

to the generated importance weight $w(\phi_t, \theta)$. 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. The full set of update equations used by our agent is:

$$
\begin{aligned}
z_0 =& 0 \\
z_t =& \gamma\lambda z_{t-1} + \mathbb{1}(m_t \notin \Omega_t) \left(\frac{\partial w(m_t)}{\partial \theta} - \frac{1}{\tilde{n}} \sum_{i=0}^{\tilde{n}-1} \frac{\partial w(\tilde{\mathcal{M}}_t(i))}{\partial \theta} \right) \\
&+ \frac{\partial \hat{v}(\phi_t, \theta_t)}{\partial \theta} \\
&+ \frac{1}{2} \frac{\partial \log(\pi(a_t|\phi_t, m_t, \theta_t))}{\partial \theta} \\
&+ \frac{1}{2} \frac{\partial \log(Q(\mathcal{M}_t(i)|\phi_t, \mathcal{M}_t, \theta_t)}{\partial \theta} \\
\theta_{t+1} =& \theta_t + \alpha \left(\delta_t z_t + \psi \frac{\partial \mathcal{H}_t}{\partial \theta} - \eta \frac{\partial w(\phi_t, \theta_t)^2}{\partial \theta} \right)
\end{aligned}
$$

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 $\psi = 0.01$. 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 fixed $\gamma = 0.99$ and $\lambda = 0.8$.

D Details of Rapid Action Association Problem

This environment was loosely based on 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 agents position on the grid was available to it 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 also 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 grid 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

Feature Name	Length	Meaning
Cell Index	grid width $\times$ 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 $\times$ 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 1: Feature representation for rapid action association.

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 state variable, 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 when 500 time-steps had elapsed. 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.

An illustration of an instance of the rapid action association problem is illustrated in Figure 1 (a). Table 1 outlines the (binary) state variable presented to the agent in each grid cell.

E Experiment Details

Our experiment tested OPGOR, along with 2 alternative memory selection strategies on the rapid action association problem. OPGOR, as well as both baselines were run within the external memory of the architecture outlined in Appendix C. All agents were trained with the AC(λ) variant outlined in Appendix C (except that only the OPGOR agent included the importance weight term to the trace). All agents used a relatively small memory size $n = 5$. This small memory size was chosen to ensure the memory selection problem was nontrivial on the problem at hand. For the OPGOR reservoir agent we also fixed the denominator reservoir size $\tilde{n} = 5$. The step-size α for each agent was tuned from $\{2^{-i} : i \in \{5, 6, ..., 10\}\}$ to optimize average performance over the final 100 training episodes. We restrict ourselves to the single-agent online RL case, using no experience replay or multiple parallel actors. We now describe the two baseline agents against which we compared OPGOR, and discuss their significance.

Uniform Reservoir: This agent is similar to our OPGOR reservoir agent, but with importance weights fixed to one instead of learned using OPGOR. This reservoir sampling procedure reduces to the simple procedure for sampling items uniformly, introduced at the start of Section A. The performance of this agent gives us an idea of how effective OPGOR was at tuning importance weights and how important this was.

Least Recently Used (LRU): For each state variable in memory, this agent keeps track of a usage timer equal to the number of time-steps passed since each state variable in memory was either written or returned as a query result. It always replaces the state variable with the largest usage timer with the current state variable. This is based on the reasoning that state variables 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.

The results of this experiment are displayed in Figure 1, each curve is the average of 30 runs, with error bars showing standard error in the mean. This figure also displays the return for a memoryless agent as 0. We did not test a memoryless agent but include this to clarify that memory is essential for this task. Except for intra-episode learning, which is unlikely to play a significant role in a conventional agent, a memoryless agent would have no means to discern the correct action for a given food item any better than random, for which the expected reward is 0.