CMPUT 656 Course Project: Learning What to Remember

Author: Kenny Young
Course Instructor: Dale Schuurmans

Introduction

Most reinforcement learning techniques assume the environment has the Markov property, meaning that the future state is independent of the past states given the present state. Put another way this assumes the agent has all the information it needs to make an optimal decision at each time and therefore has no need to remember the past. This is however not realistic in general, realistic problems often require significant information from the past in order to make an informed decision in the present and there is often no obvious way to incorporate the relevant information into an expanded present state. It is therefore desirable to establish general techniques for learning a compact representation of the relevant details of the past (i.e. a memory, or learned state) in order to facilitate decision making in the present. Most modern attempts to tackle this problem make use of variations of recurrent neural networks trained with backpropagation through time. This can work well for many tasks, but generally requires backpropagating many steps into the past which is not practical in an online reinforcement learning setting. In this work we attempt to address the problem of memory without explicit backpropagation through time. To do this we propose a method for learning over time which information should be written to and read from a lossy external memory and which information we can afford to ignore. If an RNN can be considered a working memory, the technique presented here is more similar to episodic memory, choosing which complete states to save and recall in order to perform the present task.

Problem

As a proof of concept for our proposed method we propose a toy problem we call "the secret informant problem" which requires an agent to figure out what information from past states is informative about the correct decision to make in the future. The environment consists of a chain of states, each represented by a feature vector. In each state k possible actions are available (we use k=3 throughout), however the action choice only changes the outcome in the final state where 1 of the 3 actions will yield a reward of +1 and the other 2 will yield 0 reward. Which action will yield the reward is drawn uniformly for each episode of the problem and can be determined for a given episode only by looking at the feature vector of some past state (which we refer to as the informant state) prior to the state in which the choice must be made. The feature vector associated with each state is shown in figure 1.

The agent starts with no knowledge of the semantic interpretation of the state representation and must learn through successive trials what information is relevant to allow it to select the correct action in the final state. It also does not initially know that reward is only available in the final state and all other actions are irrelevant so this too must be learned. We consider rewards to be undiscounted so the true optimal action value is always 1. An instance of the full problem is shown in 2.

Related Work

Deep learning systems which make use of an external memory have received a lot of interest lately. One somewhat prototypical example is found in [1] and the more recent followup [4] . 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. They use an addressing system based on a combination of content addressable, local iteration and random access in order to maneuver both the read and write heads through an arbitrary sized memory using output from the LSTM controller. Training is done entirely by backpropogation through time.

More directly related to this work is the application of reinforcement learning to a non-markov task implemented in [2]. They experiment with architectures using a combination of key-value memory and recurrent networks. In this case the key-value memory saved keys and values corresponding to the last M observations for some integer M thus it was inherently limited in temporal extent but did not require any mechanism for information triage. They test on

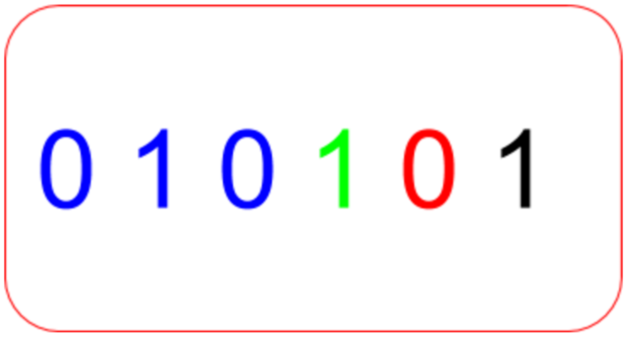


Figure 1: State representation for our test problem, blue digits are a one hot representation of which action the state is suggesting is correct (i.e. will yield +1 reward), the green digit will be 1 if and only if this state actually determines the correct action to take (i.e. it is the informant state). The red digit will be 1 if and only if the blue indicator is random and uncorrelated with the correct action choice (i.e. it is a noisy state). The final digit is simply an always on bias which may aid in the memory mechanism we will describe later. In addition to the general interpretation of these digits there is a special start node with code 000001 and the final node (at which the action choice will give a reward) with code 000011.

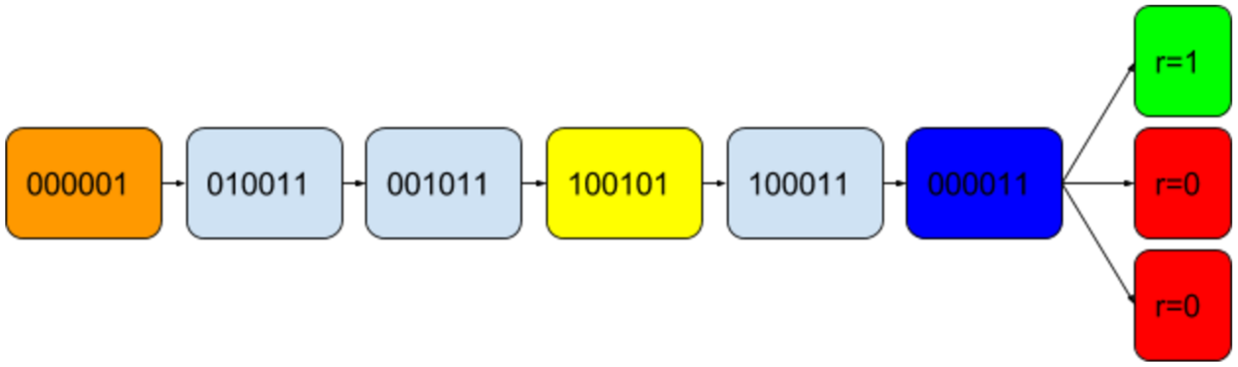


Figure 2: An instance of the secret informant problem, the start state is orange, the final state (at which a decision must be made) is dark blue, the informant state (with the 4th bit set to 1) is yellow while the noisy states (with the 4th bit set to 0) are pale blue. Notice that because the 1st bit in the informant state is 1 and the rest are 0 the reward must be given for the top action and no other. In each instance of the problem the informant state will be placed randomly between the start and final state. Here we have 3 noisy states in addition to the informant state, in our tests so far we use 4 noisy states.

several reinforcement learning problems including one which is essentially a more complex version of the problem investigated in the present work. This problem consisted of a hallway with a coloured square at the start and two possible exits at the end. Depending on the colour of the square the agent will be rewarded for picking one exit or the other at the end, thus it must remember the colour of the square and recall it as needed to complete the task. State information is given as raw screen pixels. The problems investigated in [2] may provide compelling testbeds for a followup to the present work.

Our memory mechanism is inspired by the Holographic Reduced Representation used in [3]. They make use of the interference properties of complex numbers to write an arbitrary number of key-value pairs into a fixed size memory such that each added item contributes noise to the recalled value for a given key. This memory mechanism is then operated on by an LSTM and the combined system trained to achieve a variety of tasks.

The memory mechanism used in [5] is somewhat similar to ours in that they use an external memory and train the writing process without backpropagation through time using a training process separate from but coupled with the training of the main network. In this case their memory is based on key value pairs where the keys are queries from the network and the values are class labels. The main network is trained by backpropagation and uses the memory output for classification. The memory itself uses a nearest neighbour mechanism and on the write side is trained by adding new queries to the memory with appropriate ground truth values if the current nearest neighbour does not match the ground truth label. If the current nearest neighbour already does match the ground truth label the key of this nearest neighbour is instead averaged with the most recent query. They apply their model to classification and sequence learning tasks but do not attempt to adapt this approach to a reinforcement learning environment.

Proposed Method

The main contribution of this work is to suggest a new kind of memory module designed to be trainable to both read and write using only information local in time (i.e. without requiring something like backpropagation through time). The instantiation of this idea presented here is meant only as a proof of concept with much room for improvement in terms of both theoretical foundation and actual implementation.

Memory Module

As mentioned in the above section, our memory module is inspired by (though currently quite different from) the Holographic Reduced Representation used in [3], and is intended to store and recall vectors of a fixed length n. It consists of m complex copies, each of length n. Each time a write is performed it is accompanied by a positive weight r (fixed here to be in $[0, 1]$) which simply multiplies the item to be stored and corresponds to how strongly we wish to remember the current state. Additionally each copy is multiplied by a uniformly random complex phase $e^{i\theta_k}$ chosen separately for each copy stored. If we take the memory to be an (m, n) complex matrix M writing a vector $\boldsymbol{s}$ with weight r is done as follows:

$$M_{k,j} \mathrel{+}= re^{i\theta_k}\boldsymbol{s}_j$$

Where here j indexes the elements of the stored vector $\boldsymbol{s}$ and k indexes the m copies stored in the memory. Now consider how we can go about recalling stored items from this memory. If we knew the set of random phases $e^{i\theta_k}$ with which the desired item was stored, we could multiply each copy by its complex conjugate $e^{-i\theta_k}$ and take the real part of the result to retrieve an estimate $\boldsymbol{u_{0k}}$ of $r\boldsymbol{s}$. Since the phases used for all other stored items are uniformly random they will still be uniformly random when multiplied by $e^{-i\theta_k}$. i.e. for a particular k and $\boldsymbol{s_t}$ where t is the time-step we wish to recover our written value from, and $\boldsymbol{s_t}$ is the input state written to memory at that time we get an estimate that looks like this:

$$\begin{aligned}
\boldsymbol{u_{0k}} &= \Re(e^{-i\theta_{kt}}M_{k,:}) \\
&= \Re(r_t\boldsymbol{s_t} + \sum_{\tau\neq t} e^{i(\theta_{k\tau}-\theta_{kt})}r_\tau\boldsymbol{s_\tau}) \\
&= r_t\boldsymbol{s_t} + \Re(\sum_{\tau\neq t} e^{i(\theta_{k\tau}-\theta_{kt})}r_\tau\boldsymbol{s_\tau}) \\
&= r_t\boldsymbol{s_t} + (\textit{Expectation 0 Noise})
\end{aligned}$$

By Averaging the resulting estimates $\boldsymbol{u_{0k}}$ over all m copies in memory we get a better unbias estimate $\boldsymbol{u_0}$ of $r_t\boldsymbol{s_t}$. In order to get rid of the scaling factor r_t we enforce that all the $\boldsymbol{s_t}$ are normalized before being written to memory, thus we simply normalize the resulting estimate of $r_t\boldsymbol{s_t}$ to obtain an estimate $\boldsymbol{u}$ of $\boldsymbol{s_t}$ itself.

The bad news is we do not actually know the complex phases $e^{-i\theta_k}$ for the particular item we are trying to recall. The only information we are storing is the m complex vectors that make up the memory. In the present work we use a particular method for querying the memory and estimating these phases, however the theoretical details have not been fully hashed out. Refining our memory mechanism to have better understood theoretical properties (which hopefully translates into better results and greater applicability) is a task for future work.

The way we handle queries to the memory is a form of content addressable read mechanism. A query to the memory consists of an arbitrary real vector of length n (the same as the length of the stored items). For each copy in the memory we then estimate the phase of the desired read value to be the phase which results in the maximum dot product between the query $\boldsymbol{q}$ and the returned estimate $\tilde{\boldsymbol{s}}_{\boldsymbol{t}k}$. As we will now show this maximization process is a differentiable operation and can thus be trained by backpropagation directly.

$$\begin{aligned}
\theta_k &= \underset{\theta}{argmax}\ \Re(e^{i\theta}\boldsymbol{q}\cdot M_{k,:}) \\
&= \underset{\theta}{argmax}\ \Re(\cos(\theta)\boldsymbol{q}\cdot M_{k,:} + i\sin(\theta)\boldsymbol{q}\cdot M_{k,:} \\
&= \underset{\theta}{argmax}\ \cos(\theta)\Re(\boldsymbol{q}\cdot M_{k,:}) - \sin(\theta)\Im(\boldsymbol{q}\cdot M_{k,:}) \\
&\implies \frac{d}{d\theta}(\cos(\theta)\Re(\boldsymbol{q}\cdot M_{k,:}) - \sin(\theta)\Im(\boldsymbol{q}\cdot M_{k,:})) = 0, \text{ at } \theta_k \\
&\implies -\sin(\theta_k)\Re(\boldsymbol{q}\cdot M_{k,:}) - \cos(\theta_k)\Im(\boldsymbol{q}\cdot M_{k,:}) = 0 \\
&\implies \tan(\theta_k) = -\frac{\Im(\boldsymbol{q}\cdot M_{k,:})}{\Re(\boldsymbol{q}\cdot M_{k,:})} \\
&\implies \theta_k = n\pi - \arctan\left(\frac{\Im(\boldsymbol{q}\cdot M_{k,:})}{\Re(\boldsymbol{q}\cdot M_{k,:})}\right), \text{ for some integer } n
\end{aligned}$$

Note that angles differing by 2π are equivalent hence there are really 2 extrema here, one is a minimum and one is a maxima. To figure out which is which we can look at the second derivative. After doing this and performing some simple algebra we find that the maxima we desire is given by:

$$\theta_k = \begin{cases} -\arctan\left(\frac{\Im(\boldsymbol{q}\cdot M_{k,:})}{\Re(\boldsymbol{q}\cdot M_{k,:})}\right) & \text{if } \Re(\boldsymbol{q}\cdot M_{k,:}) \geq 0 \\ \pi - \arctan\left(\frac{\Im(\boldsymbol{q}\cdot M_{k,:})}{\Re(\boldsymbol{q}\cdot M_{k,:})}\right) & \text{if } \Re(\boldsymbol{q}\cdot M_{k,:}) < 0 \end{cases}$$

We can use this angle to find the estimated $\boldsymbol{s}$ in memory “closest” to the query $\boldsymbol{q}$ as described above. Note that since this optimal angle and the subsequent computation of the estimate $\boldsymbol{s}$ described above are both differentiable almost everywhere, we can backpropogate through this to improve our generated queries $\boldsymbol{q}$ in order to make use of our stored memories to achieve our specified task. We refer to the estimated $\boldsymbol{s}$ recovered by this process (i.e. the value retrieved from memory in response to query $\boldsymbol{q}$) as $\boldsymbol{u}$.

Training Write Network

As highlighted above the query portion of our network is fully differentiable and therefore trainable directly by back-propagation. In addition we have a second network whose purpose is to choose the write weight r for the state at each time-step. While in principle the write portion of the network is also differentiable and could also be trained by a variant of backpropagation through time, the purpose of the present work is to design a method which avoids this, thus we instead use an update method which is strictly local in time. Our method attempts to approximate $\frac{d\sigma^2(Q(\boldsymbol{s},a))}{dr(\boldsymbol{u})}$ (i.e. the change in the variance of each action value in the current state over the change in the write weight associated with the current read value, where variance here means over the random complex phases applied in each write). Using this estimate we then compute $\frac{d\sigma^2(Q(\boldsymbol{s},a))}{\theta_{write}} = \frac{d\sigma^2(Q(\boldsymbol{s},a))}{dr(\boldsymbol{u})}\frac{dr(\boldsymbol{u})}{\theta_{write}}$ for each parameter θ_{write} of the write network. This value is then used to perform gradient descent to tune the write network to generate weights which produce low average variance in the computed action values over all actions.

To approximate $\frac{d\sigma^2(Q(\boldsymbol{s},a))}{dr(\boldsymbol{u})}$ we first assume $\sigma^2(Q(\boldsymbol{s},a)) = \left(\frac{dQ(\boldsymbol{s},a)}{d\boldsymbol{u}}\right)^2\sigma^2(\boldsymbol{u})$, which is an assumption commonly employed to perform propagation of experimental errors in the physical sciences. Differentiating with respect to r then gives us:

$$\frac{d\sigma^2(Q(\boldsymbol{s},a))}{dr(\boldsymbol{u})} = \left(\frac{dQ(\boldsymbol{s},a)}{d\boldsymbol{u}}\right)^2\frac{d\sigma^2(\boldsymbol{u})}{dr(\boldsymbol{u})}$$

$\frac{dQ(\boldsymbol{s},a)}{d\boldsymbol{u}}$ is easily computed by ordinary backpropagation, but it remains to estimate $\frac{d\sigma^2(\boldsymbol{u})}{dr(\boldsymbol{u})}$. Since $\boldsymbol{u}$ is generated from m independent estimates, we can approximate $\sigma^2(\boldsymbol{u})$ as the standard error in the mean computed from these m samples.

Approximating its derivative with respect to $r(\boldsymbol{u})$ requires a bit more creativity. To do this we take $\boldsymbol{u}$ to be the "true" value and take the difference of the estimates directed from individual samples from this value to be added noise terms $\boldsymbol{N_k}$ defined as follows:

$$\begin{aligned}
\boldsymbol{u_0} &= \frac{1}{m}\sum_{k=1}^{m}\boldsymbol{u_{0k}} \\
\bar{r} &= |\boldsymbol{u_0}|_2 \\
\boldsymbol{u} &= \frac{\boldsymbol{u_0}}{\bar{r}} \\
\boldsymbol{u_k} &= \frac{\boldsymbol{u_{0k}}}{\bar{r}} \\
\boldsymbol{N_k} &= \boldsymbol{u_{0k}} - \boldsymbol{u_0} \\
\implies \boldsymbol{u_{0k}} &= \boldsymbol{u_0} + \boldsymbol{N_k}
\end{aligned}$$

We then make the assumption that $\boldsymbol{u}$ does in fact correspond to an actual value $\boldsymbol{s}$ which was written to the memory, plus some additive noise with some $r = \bar{r}$ which can be estimated as the magnitude of $\boldsymbol{u_0}$ ($\boldsymbol{u}$ prior to normalizing). Given this we can reasonably assume that modifying $r(\boldsymbol{u})$ to a value other than $\bar{r}$ will change the magnitude of $\boldsymbol{u_o}$ while holding the added noise terms $\boldsymbol{N_k}$ constant. Hence letting $\boldsymbol{u_o}(r)$ be the scaled value of $\boldsymbol{u_o}$ at a new r value, we arrive at the following expression for $\boldsymbol{u_k}(r)$ (the k_{th} sampled estimate of $\boldsymbol{u}$).

$$\begin{aligned}
\boldsymbol{u_0}(r) &= \frac{\boldsymbol{u_0}r}{\bar{r}} \\
\boldsymbol{u_{ok}}(r) &= \boldsymbol{u_0}(r) + \boldsymbol{N_k} \\
&= \frac{\boldsymbol{u_0}r}{\bar{r}} + \boldsymbol{N_k} \\
\boldsymbol{u_k}(r) &= \frac{\boldsymbol{u_{ok}}(r)}{r} \\
&= \frac{\boldsymbol{u_0}}{\bar{r}} + \frac{\boldsymbol{N_k}}{r} \\
&= \boldsymbol{u} + \frac{\boldsymbol{N_k}}{r}
\end{aligned}$$

So the final result is just that the difference in each sampled estimate from the mean is divided by r instead of $\bar{r}$. From this we can compute an estimate of $\sigma^2(\boldsymbol{u}(r))$ as a function of r as follows:

$$\begin{aligned}
\sigma^2(\boldsymbol{u}(r)) &= \frac{1}{m}\sum_{k=1}^{m}(\boldsymbol{u} - \boldsymbol{u}_k(r))^2 \\
&= \frac{1}{m}\sum_{k=1}^{m}\left(\frac{\boldsymbol{N_k}}{r}\right)^2
\end{aligned}$$

And finally we can differentiate this approximation with respect to r to yield:

$$\left.\frac{d\sigma^2(\boldsymbol{u}(r))}{dr}\right|_{\bar{r}} = -\frac{2}{m}\sum_{k=1}^{m}\frac{\boldsymbol{N_k}^2}{\bar{r}^3}$$

At this point we now have all the ingredients to estimate $\frac{d\sigma^2(Q(\boldsymbol{s},a))}{\theta_{write}}$ for a particular read value $\boldsymbol{u}$ at a particular time-step. Note however that there is a hidden tradeoff, because the nature of the memory is that each written item interferes with all others. Hence if we increase the write weight assigned to one particular input state $\boldsymbol{s}$ it will also increase the noise terms associated with all other written values. Applying the above approximate gradients naively will lead to all write weights gradually increasing (though at differing rates), which is not acceptable. To counter this, for now, we simply maintain an exponential moving average of $\frac{d\sigma^2(Q(\boldsymbol{s},a))}{dr(\boldsymbol{u})}$ and subtract this from the value at each time-step before performing the gradient descent update. This means that the write weight associated with the current read value $\boldsymbol{u}$ will be increased only if it's associated $\frac{d\sigma^2(Q(\boldsymbol{s},a))}{dr(\boldsymbol{u})}$ surpasses the current moving average and will be decreased otherwise. In retrospect this is probably far from being a good way to correct for the effect of other written memories. The query network is always aiming to create useful queries so it has no motivation to generate training examples for the write network that aren't useful in estimating action values and hence we will not get a strong signal indicating which state value should not be written to memory. What is really needed is a way to account for states that are visited but never prove useful in predicting any future action value, prehaps simply by reducing the write weight slightly for each state upon visitation, how exactly to do this is another question for future work. To summarize we use:

$$\text{Effective Gradient} = \left(\left(\frac{dQ(\boldsymbol{s_t},a)}{d\boldsymbol{u_t}}\right)^2 \frac{d\sigma^2(\boldsymbol{u_t})}{dr(\boldsymbol{u_t})} - S_t\right)\frac{dr(\boldsymbol{u_t})}{\theta_{write}}$$

$$S_{t+1} = \alpha\left(\left(\frac{dQ(\boldsymbol{s_t},a)}{d\boldsymbol{u_t}}\right)^2 \frac{d\sigma^2(\boldsymbol{u_t})}{dr(\boldsymbol{u_t})}\right) + (1-\alpha)S_t$$

Where each term is approximated as discussed above, and we insert this "effective gradient" as the gradient for some gradient descent optimizer acting on all θ_{write} parameters of the write network at each time-step. In our experiments we use $\alpha = 0.5$ in the moving average.

Architecture

Our architecture for interacting with the memory and generating action values is shown in Figure 3. It consists of 4 major parts. Except where otherwise specified all activations are rectified linear.

First is the query network which takes the state itself as input and consists of 2 relu layers followed by an identity layer which generates the query which is sent to the memory. Additionally the 2_{nd} layer of the query network is passed forward so that information about the current state is available for computing action values.

Second we have the memory itself which works as described above, receiving queries $\boldsymbol{q}$ from the query network and writes $\boldsymbol{v_k} = re^{i\theta_k}\frac{\boldsymbol{s}}{|\boldsymbol{s}|_2}$ with weight r decided by the write network, and sending responses $\boldsymbol{u}$ to the action value network.

Third is the action value network itself which takes the memory responses along with information about the current state from the 2_{nd} layer of the query network and generates estimated values for each of the 3 actions. The action values in this case are bounded between 0 and 1 (in fact the true action values are either 0 or 1), hence the output of the action value network uses a sigmoid activation. The action value network is trained using deep Q-learning and propagates gradients directly through the memory to improve the $\boldsymbol{q}$ values produced, as well as along the path directly to the input without passing through the memory.

Last we have the write network which is trained separately from the others and outputs a single sigmoid activation r. This r is multiplied by the normalized input along with a separately drawn uniformly random complex phase for each copy stored in the memory. The training process for the write network is described in detail in the memory section above.

Results

We run experiments on the secret informant problem (see Figure 2) with 4 noisy states and run for 50,000 episodes. Each layer of our model uses 15 units. We use Nesterov Momentum as our optimizer for both the write and action evaluation portions of the network as it was found to be reasonably stable compared to some other options tried. Learning rates were tuned by hand to achieve reasonable performance, it was found that in general training the write network with around $1/10_{th}$ the learning rate of the action evaluation network worked well. We use a memory with just 3 copies in our experiments. Results for 50 repetitions of this experiment, including the output of the write network in the informant state v.s. the average over all other states, are shown in Figure 4.

In addition to this main model we test three baselines. First to test the efficacy of the write network we run the model with r fixed to 1 and the write network disabled. This means each visited state is written to the memory with

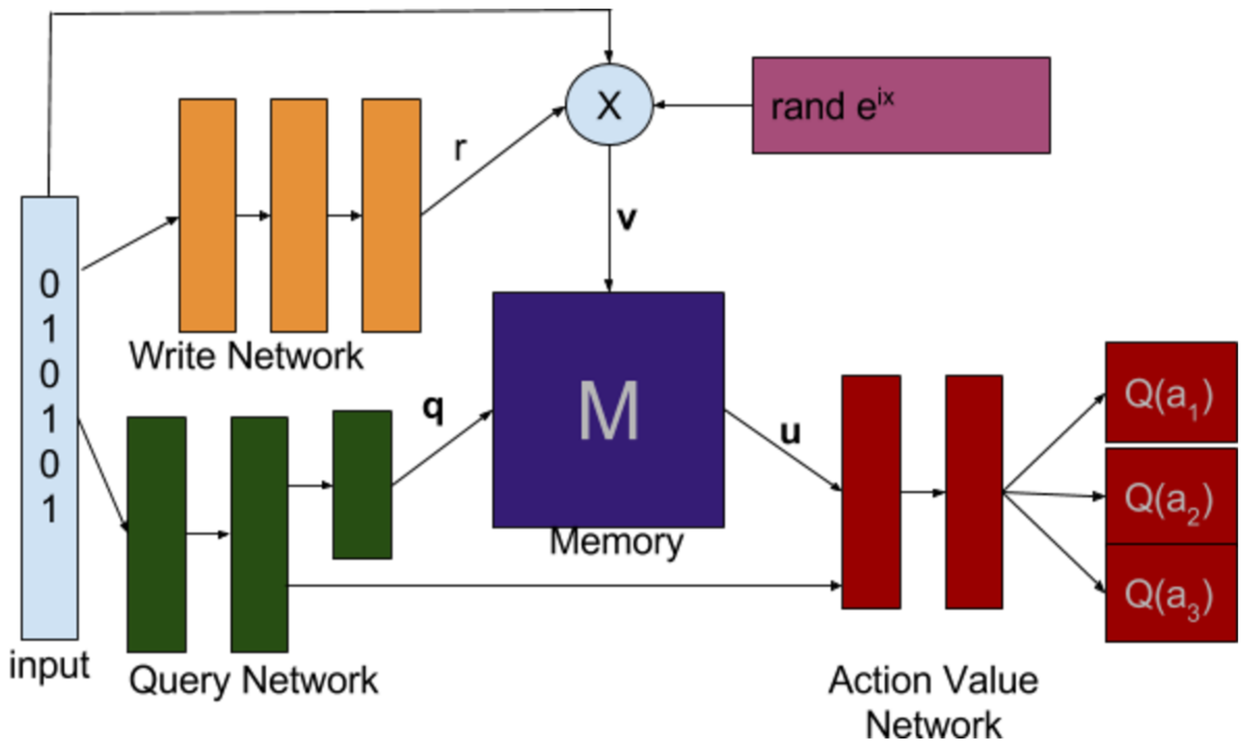
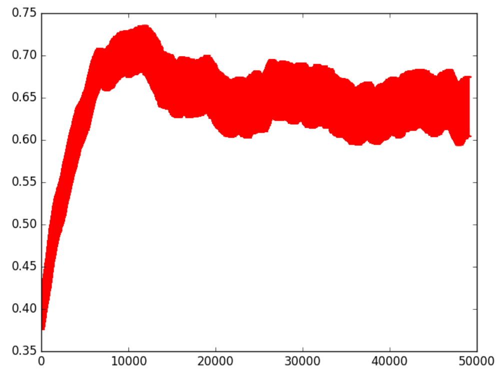


Figure 3: Our architecture, input is given separately to both query and write network. Write network output is sigmoid to give a weight between 0 and 1, this weight multiplies the input vector input along with one random complex phase per copy to give the values are actually written to memory. The activation of the query network output **q** to memory is identity (i.e. potentially spans the whole space) the memory returns a vector **u**. The input to the action value network consists of the output of the second last layer of the query network, along with the value **u** read from memory. The action value network outputs action values for all available actions.

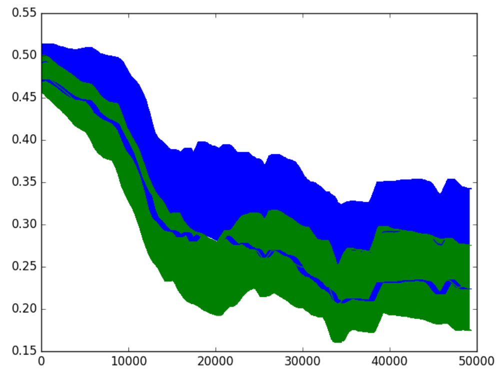
equal intensity. Second we run a model with essentially the same architecture as our action evaluation network but with the memory replaced by a GRU with 5 recurrent units trained using standard backpropagation through time. This model is meant to give an idea of how difficult the task is for a traditional recurrent architecture but as it hasn't been all that thoroughly tuned it is not a very rigorous baseline. Note that our goal in this work was to avoid the need to perform backpropagation through time explicitly so any comparison there is not really fair. For this reason our third baseline is the same recurrent model but trained with truncated backpropagation through time backpropagating only one step. Figure 5 shows the results for our main model compared against the three baselines.

We also experimented with variants of our architecture which allowed for multiple queries on each time step which appeared in initial tests to speed up early learning somewhat, however it has so far been prone to instability so results are not included here. This would be more useful in cases where it is actually necessary to query the values of two distinct past states to make a decision in the current state.

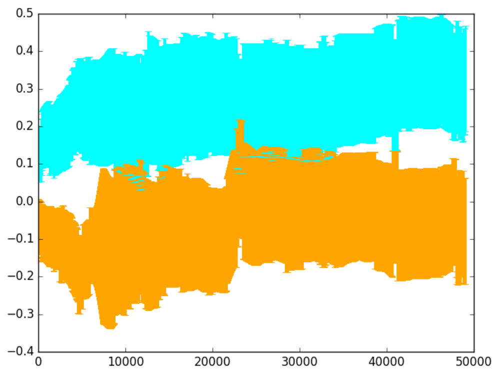
In all cases the action values are learned with Q-learning with an epsilon greedy behaviour policy, with $\epsilon = 0.1$. Returns are computed during learning under the same epsilon greedy policy, hence the maximum possible expected return is $0.9\bar{3}$, while random action selection will yield $0.\bar{3}$. We use a very simplified form of deep Q-learning [7] with no experience replay or other frills for our experiments.



(a) Returns over time for memory based architecture. y-axis is return, x-axis is number of training episodes.



(b) Write weight for informant state (blue) and average write weight for all noisy states (green). y-axis is write weight, x-axis is number of training episodes.



(c) Average query network output for the final state assigned to 4_{th} bit which indicates the informant state (cyan) and 5_{th} bit (orange) which indicates a noisy state. Each query weight is multiplied by the corresponding sign of the 6_{th} bit (the always on bias). Multiplication by the sign of the bias is done because there is an inherent symmetry between positive and negative queries which this adjustment breaks.

Figure 4: Results for memory based architecture. In each case displayed results are averaged over 45 runs (this would have been 50 but 5 runs terminated early due to occurrence of nan values, the cause of which still needs to be determined) with the width of the curve indicating standard error in the mean.

Discussion

The results are mixed in terms of demonstrating the feasibility of our suggested memory system and write weight training. While Figure 4b suggests (though weakly) that our write weight training mechanism is able to learn to somewhat separate informative from uninformative states, we can see in Figure 5 that our write weight tuning nevertheless hurts performance compared to simply fixing all write weights at 1. The reason for this is likely that although on average we learn to write useful states more strongly the variance over runs is rather high and occasionally the useful states do end up being written significantly less strongly. What is likely happening is that the performance cost of these occasional

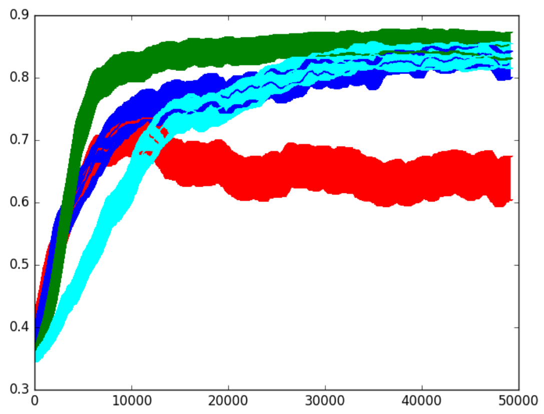


Figure 5: Returns over time for memory based architecture and three baselines. Main memory based architecture with write weight tuning is shown in red, memory based architecture with write weight tuning disabled is shown in blue, recurrent architecture with full backpropagation through time is shown in green, recurrent architecture with 1-step truncated backpropagation through time is shown in cyan. y-axis is return, x-axis is number of training episodes.

failures is high enough compared to the gain from the more frequent successes that average performance is hindered.

Despite the underwhelming result for training write weights we do clearly see that our system is able to learn to query the memory to achieve the given task and achieve reasonably good return values, though perhaps this is not surprising as this part of the task amounts mostly to just learning to query the correct bit, and then learning to interpret the associated memory output. We also see in Figure 4c that this does appear to be essentially what the system is doing as it is able to learn a positive weight for the bit indicating relevance and negative weight for the one indicating irrelevance.

Our system also seems to lose out somewhat to a more standard recurrent architecture, this in itself is not too meaningful as our purpose here is to explore memory mechanisms which avoid the need to perform explicit backpropagation through time which is exactly what is done by the recurrent model tested. Without write tuning our model seems to perform comparably to the recurrent model trained using 1-step backpropagation through time, and even trains somewhat faster off the start.

It should be noted that the mechanism presented here for training the write network is not particular to the memory system used here but is actually quite broadly applicable. Essentially all that is required to use our write weight training algorithm is a system with roughly the following components:

- A memory is capable of storing and recalling vectors with a tuneable fidelity parameter, where improved recall of one memory can be traded off for worse recall of others
- A differentiable function mapping the recalled memory to some value whose variance we wish to minimize (in our case learned action values)
- A way of estimating the derivative of standard error in the mean of a value read from memory with respect to it's associated fidelity parameter at write time(in our case we use an approximated derivative of the sample standard deviation over several independent copies)

Thus despite the failure of the tuned write weights to aid in performance we are encouraged by the evidence of our ability to successfully tune write weights based on utility of the information contained in a state, even with the shaky theoretical ground on which our system currently stands.

Conclusion

We investigate the integration of a new memory mechanism designed to be incorporate learned reads and writes without backpropagation through time. To test this we suggest a toy problem we call "the secret informant" problem, designed to require a system to learn to recall past state information to inform present decisions. This problem itself could provide an interesting test case for a variety of other systems designed to incorporate memory in a reinforcement learning context. We show clearly that our system is capable of learning to read from the memory to achieve the task. The results for learning to write to the memory were less clear but nonetheless encouraging. The task selected here was found to not be sufficiently challenging to test the limits of our memory system as we are able to perform quite well without write weight learning (and in fact learning the write weights hindered expected performance). More theoretical analysis is needed to allow the present system to match our original ambition, as well as testing on a wider variety of problems to gain a better understanding of the range of applicability.

Limitations and Future Work

This work is intended primarily as a proof of concept for systems that operate on some form of variable fidelity memory mechanism and learn write weights based on how the read values impact a systems ability to consistently achieve some primary task. As such there are likely numerous theoretical and practical improvements that could be made, as well as far more empirical testing that should be done.

First of all the memory mechanism itself is far from perfect. Significant improvement could likely be derived from more thorough theoretical investigation of such mechanisms and more precise formulation of exactly what properties we would like the memory to display. Under the current mechanism it is not clear at all how we should interpret the read value when the query made has a nonzero dot-product with multiple previously written values which is something we hope to improve on through more investigation.

The write weight tuning is also far from perfect, as discussed above we do not believe that our current method of accounting for the effect of other written states on the fidelity of the current memory being read is at all sufficient. One way we believe this could be improved is to somehow account for each state visitation and reduce write weights specifically for those states that are visited but never prove to be useful later on. Aside from this our derivation of the gradients involved in the write weight tuning employed a number of assumptions, the validity of which could be rightly questioned, which should be further investigated.

Another issue with the present system is that the variance of recalled memories will grow without bound as more and more items are written because there is presently no way to remove information from the memory. It should be possible to add some mechanism to intelligently delete memories that are no longer needed in order to free up space.

Further down the line we would like to be able to learn a state representation to write to memory, instead of writing the input state to memory directly. This is complicated by the fact that our aim is to avoid explicit backpropagation through time which would provide the most direct way to do this. A variation of the synthetic gradient method used in [6] may be one route toward this.

We would also like to test our method on more realistic and varied problems, perhaps including something like the Minecraft environment used in [2]. The task examined here is obviously not sufficient to test the limits of the memory as we see that even without write weight tuning our architecture is able to perform quite well.

References

[1] Graves, A., Wayne, G., and Danihelka, I. (2014) Neural Turing Machines. arXiv preprint arXiv:1410.5401.

[2] Oh, J., Chockalingam, V., Singh, S., and Lee, H. (2016) Control of Memory, Active Perception, and Action in Minecraft. ICML, New York City, 2016. JMLR Workshop and Conference Proceedings Volume 48.

[3] Danihelka, I., Wayne, G., Uria, B., Kalchbrenner, N., and Graves, A. (2016) Associative Long Short-Term Memory. ICML, New York City, 2016. JMLR Workshop and Conference Proceedings Volume 48.

[4] Graves, A. et al. (2016) Hybrid Computing Using a Neural Network with Dynamic External Memory. Nature 538, 471-476.

[5] Kaiser, Lukasz, et al. (2017) Learning to remember rare events. arXiv preprint arXiv:1703.03129.

[6] Jaderberg, Max, et al. (2016) Decoupled neural interfaces using synthetic gradients. arXiv preprint arXiv:1608.05343.

[7] Mnih, Volodymyr, et al. (2015) Human-level control through deep reinforcement learning. Nature 518.7540: 529-533.