Using Fast Weights to Attend to the Recent Past

Jimmy Ba
University of Toronto
jimmy@psi.toronto.edu

Geoffrey Hinton
University of Toronto and Google Brain
geoffhinton@google.com

Volodymyr Mnih
Google DeepMind
vmnih@google.com

Joel Z. Leibo
Google DeepMind
jzl@google.com

Catalin Ionescu
Google DeepMind
cdi@google.com

Abstract

Until recently, research on artificial neural networks was largely restricted to systems with only two types of variable: Neural activities that represent the current or recent input and weights that learn to capture regularities among inputs, outputs and payoffs. There is no good reason for this restriction. Synapses have dynamics at many different time-scales and this suggests that artificial neural networks might benefit from variables that change slower than activities but much faster than the standard weights. These "fast weights" can be used to store temporary memories of the recent past and they provide a neurally plausible way of implementing the type of attention to the past that has recently proved very helpful in sequence-to-sequence models. By using fast weights we can avoid the need to store copies of neural activity patterns.

1 Introduction

Ordinary recurrent neural networks typically have two types of memory that have very different time scales, very different capacities and very different computational roles. The history of the sequence currently being processed is stored in the hidden activity vector, which acts as a short-term memory that is updated at every time step. The capacity of this memory is $\mathrm{O}(H)$ where H is the number of hidden units. Long-term memory about how to convert the current input and hidden vectors into the next hidden vector and a predicted output vector is stored in the weight matrices connecting the hidden units to themselves and to the inputs and outputs. These matrices are typically updated at the end of a sequence and their capacity is $\mathrm{O}(H^2) + \mathrm{O}(IH) + \mathrm{O}(HO)$ where I and O are the numbers of input and output units.

Long short-term memory networks [Hochreiter and Schmidhuber, 1997] are a more complicated type of RNN that work better for discovering long-range structure in sequences for two main reasons: First, they compute *increments* to the hidden activity vector at each time step rather than recomputing the full vector[1]. This encourages information in the hidden states to persist for much longer. Second, they allow the hidden activities to determine the states of gates that scale the effects of the weights. These multiplicative interactions allow the effective weights to be dynamically adjusted by the input or hidden activities via the gates. However, LSTMs are still limited to a short-term memory capacity of $\mathrm{O}(H)$ for the history of the current sequence.

Until recently, there was surprisingly little practical investigation of other forms of memory in recurrent nets despite strong psychological evidence that it exists and obvious computational reasons why it was needed. There were occasional suggestions that neural networks could benefit from a third form of memory that has much higher storage capacity than the neural activities but much faster dynamics than the standard slow weights. This memory could store information specific to the history of the current sequence so that this information is available to influence the ongoing processing

[1]This assumes the "remember gates " of the LSTM memory cells are set to one.

without using up the memory capacity of the hidden activities. Hinton and Plaut [1987] suggested that fast weights could be used to allow true recursion in a neural network and Schmidhuber [1993] pointed out that a system of this kind could be trained end-to-end using backpropagation, but neither of these papers actually implemented this method of achieving recursion.

2 Evidence from physiology that temporary memory may not be stored as neural activities

Processes like working memory, attention, and priming operate on timescale of 100ms to minutes. This is simultaneously too slow to be mediated by neural activations without dynamical attractor states (10ms timescale) and too fast for long-term synaptic plasticity mechanisms to kick in (minutes to hours). While artificial neural network research has typically focused on methods to maintain temporary state in activation dynamics, that focus may be inconsistent with evidence that the brain also—or perhaps primarily—maintains temporary state information by short-term synaptic plasticity mechanisms [Tsodyks et al., 1998, Abbott and Regehr, 2004, Barak and Tsodyks, 2007].

The brain implements a variety of short-term plasticity mechanisms that operate on intermediate timescale. For example, short term facilitation is implemented by leftover $[\text{Ca}^{2+}]$ in the axon terminal after depolarization while short term depression is implemented by presynaptic neurotransmitter depletion Zucker and Regehr [2002]. Spike-time dependent plasticity can also be invoked on this timescale [Markram et al., 1997, Bi and Poo, 1998]. These plasticity mechanisms are all synapse-specific. Thus they are more accurately modeled by a memory with $O(H^2)$ capacity than the $O(H)$ of standard recurrent artificial recurrent neural nets and LSTMs.

3 Fast Associative Memory

One of the main preoccupations of neural network research in the 1970s and early 1980s [Willshaw et al., 1969, Kohonen, 1972, Anderson and Hinton, 1981, Hopfield, 1982] was the idea that memories were not stored by somehow keeping copies of patterns of neural activity. Instead, these patterns were reconstructed when needed from information stored in the weights of an associative network and the very same weights could store many different memories An auto-associative memory that has N^2 weights cannot be expected to store more that N real-valued vectors with N components each. How close we can come to this upper bound depends on which storage rule we use. Hopfield nets use a simple, one-shot, outer-product storage rule and achieve a capacity of approximately $0.15N$ binary vectors using weights that require $\log(N)$ bits each. Much more efficient use can be made of the weights by using an iterative, error correction storage rule to learn weights that can retrieve each bit of a pattern from all the other bits [Gardner, 1988], but for our purposes maximizing the capacity is less important than having a simple, non-iterative storage rule, so we will use an outer product rule to store hidden activity vectors in fast weights that decay rapidly. The usual weights in an RNN will be called slow weights and they will learn by stochastic gradient descent in an objective function taking into account the fact that changes in the slow weights will lead to changes in what gets stored automatically in the fast associative memory.

A fast associative memory has several advantages when compared with the type of memory assumed by a Neural Turing Machine (NTM) [Graves et al., 2014], Neural Stack [Grefenstette et al., 2015], or Memory Network [Weston et al., 2014]. First, it is not at all clear how a real brain would implement the more exotic structures in these models e.g., the tape of the NTM, whereas it is clear that the brain could implement a fast associative memory in synapses with the appropriate dynamics. Second, in a fast associative memory there is no need to decide where or when to write to memory and where or when to read from memory. The fast memory is updated all the time and the writes are all superimposed on the same fast changing component of the strength of each synapse. Every time the input changes there is a transition to a new hidden state which is determined by a combination of three sources of information: The new input via the slow input-to-hidden weights, C, the previous hidden state via the slow transition weights, W, and the recent history of hidden state vectors via the fast weights, A. The effect of the first two sources of information on the new hidden state can be computed once and then maintained as a sustained boundary condition for a brief iterative settling process which allows the fast weights to influence the new hidden state. Assuming that the fast weights decay exponentially, we now show that the effect of the fast weights on the hidden vector

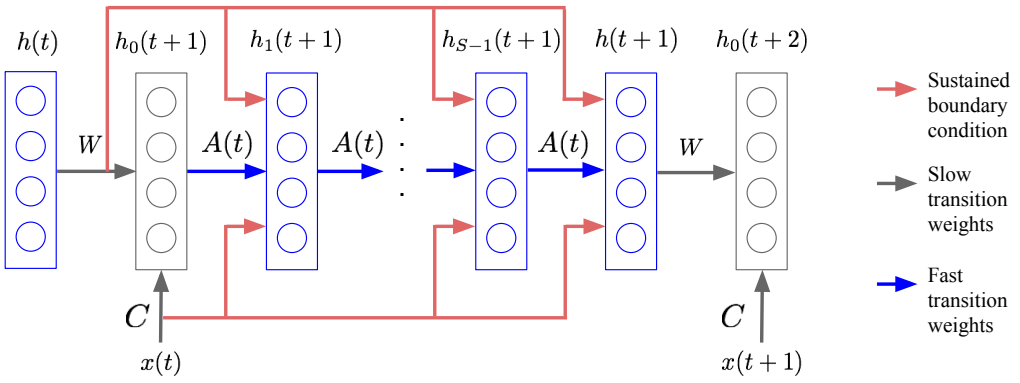


Figure 1: The fast associative memory model.

during an iterative settling phase is to provide an additional input that is proportional to the sum over all recent hidden activity vectors of the scalar product of that recent hidden vector with the current hidden activity vector, with each term in this sum being weighted by the decay rate raised to the power of how long ago that hidden vector occurred. So fast weights act like a kind of attention to the recent past but with the strength of the attention being determined by the scalar product between the current hidden vector and the earlier hidden vector rather than being determined by a separate parameterized computation of the type used in neural machine translation models [Bahdanau et al., 2015].

The update rule for the fast memory weight matrix, A, is simply to multiply the current fast weights by a decay rate, λ, and add the outer product of the hidden state vector, $h(t)$, multiplied by a learning rate, η:

$$A(t) = \lambda A(t-1) + \eta h(t)h(t)^T \tag{1}$$

The next vector of hidden activities, $h(t+1)$, is computed in two steps. The "preliminary" vector $h_0(t+1)$ is determined by the combined effects of the input vector $x(t)$ and the previous hidden vector: $h_0(t+1) = f(Wh(t) + Cx(t))$, where W and C are slow weight matrices and $f(.)$ is the nonlinearity used by the hidden units. The preliminary vector is then used to initiate an "inner loop" iterative process which runs for S steps and progressively changes the hidden state into $h(t+1) = h_S(t+1)$

$$h_{s+1}(t+1) = f([Wh(t) + Cx(t)] + A(t)h_s(t+1)), \tag{2}$$

where the terms in square brackets are the sustained boundary conditions. In a real neural net, A could be implemented by rapidly changing synapses but in a computer simulation that uses sequences which have fewer time steps than the dimensionality of h, A will be of less than full rank and it is more efficient to compute the term $A(t)h_s(t+1)$ without ever computing the full fast weight matrix, A. Assuming A is 0 at the beginning of the sequence,

$$A(t) = \eta \sum_{\tau=1}^{\tau=t} \lambda^{t-\tau} h(\tau)h(\tau)^T \tag{3}$$

$$A(t)h_s(t+1) = \eta \sum_{\tau=1}^{\tau=t} \lambda^{t-\tau} h(\tau)[h(\tau)^T h_s(t+1)] \tag{4}$$

The term in square brackets is just the scalar product of an earlier hidden state vector, $h(\tau)$, with the current hidden state vector, $h_s(t+1)$, during the iterative inner loop. So at each iteration of the inner loop, the fast weight matrix is exactly equivalent to attending to past hidden vectors in proportion to their scalar product with the current hidden vector, weighted by a decay factor. During the inner loop iterations, attention will become more focussed on past hidden states that manage to attract the current hidden state.

The equivalence between using a fast weight matrix and comparing with a set of stored hidden state vectors is very helpful for computer simulations. It allows us to explore what can be done with fast

weights without incurring the huge penalty of having to abandon the use of mini-batches during training. At first sight, mini-batches cannot be used because the fast weight matrix is different for every sequence, but comparing with a set of stored hidden vectors does allow mini-batches.

3.1 Layer normalized fast weights

A potential problem with fast associative memory is that the scalar product of two hidden vectors could vanish or explode depending on the norm of the hidden vectors. Recently, layer normalization [Ba et al., 2016] has been shown to be very effective at stablizing the hidden state dynamics in RNNs and reducing training time. Layer normalization is applied to the vector of summed inputs to all the recurrent units at a particular time step. It uses the mean and variance of the components of this vector to re-center and re-scale those summed inputs. Then, before applying the nonlinearity, it includes a learned, neuron-specific bias and gain. We apply layer normalization to the fast associative memory as follows:

$$h_{s+1}(t+1) = f(\mathcal{LN}[Wh(t) + Cx(t) + A(t)h_s(t+1)]) \tag{5}$$

where $\mathcal{LN}[.]$ denotes layer normalization. We found that applying layer normalization on each iteration of the inner loop makes the fast associative memory more robust to the choice of learning rate and decay hyper-parameters. For the rest of the paper, fast weight models are trained using layer normalization and the outer product learning rule with fast learning rate of 0.5 and decay rate of 0.95, unless otherwise noted.

4 Experimental results

To demonstrate the effectiveness of the fast associative memory, we first investigated the problems of associative retrieval (section 4.1) and MNIST classification (section 4.2). We compared fast weight models to regular RNNs and LSTM variants. We then applied the proposed fast weights to a facial expression recognition task using a fast associative memory model to store the results of processing at one level while examining a sequence of details at a finer level (section 4.3). The hyper-parameters of the experiments were selected through grid search on the validation set. All the models were trained using mini-batches of size 128 and the Adam optimizer [Kingma and Ba, 2014]. A description of the training protocols and the hyper-parameter settings we used can be found in the Appendix. Lastly, we show that fast weights can also be used effectively to implement reinforcement learning agents with memory (section 4.4).

4.1 Associative retrieval

We start by demonstrating that the method we propose for storing and retrieving temporary memories works effectively for a toy task to which it is very well suited. Consider a task where multiple key-value pairs are presented in a sequence. At the end of the sequence, one of the keys is presented and the model must predict the value that was temporarily associated with the key. We used strings that contained characters from English alphabet, together with the digits 0 to 9. To construct a training sequence, we first randomly sample a character from the alphabet without replacement. This is the first key. Then a single digit is sampled as the associated value for that key. After generating a sequence of K character-digit pairs, one of the K different characters is selected at random as the query and the network must predict the associated digit. Some examples of such string sequences and their targets are shown below:

Input string	Target
c9k8j3f1??c	9
j0a5s5z2??a	5

where '?' is the token to separate the query from the key-value pairs. We generated 100,000 training examples, 10,000 validation examples and 20,000 test examples. To solve this task, a standard RNN has to end up with hidden activities that somehow store all of the key-value pairs after the keys and values are presented sequentially. This makes it a significant challenge for models only using slow weights.

We used a neural network with a single recurrent layer for this experiment. The recurrent network processes the input sequence one character at a time. The input character is first converted into a

Model	R=20	R=50	R=100
IRNN	62.11%	60.23%	0.34%
LSTM	60.81%	1.85%	0%
A-LSTM	60.13%	1.62%	0%
Fast weights	1.81%	0%	0%

Table 1: Classification error rate comparison on the associative retrieval task.

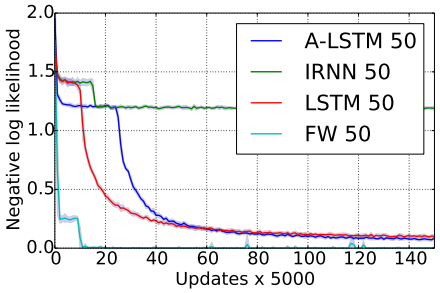


Figure 2: Comparison of the test log likelihood on the associative retrieval task with 50 recurrent hidden units.

learned 100-dimensional embedding vector which then provides input to the recurrent layer[2]. The output of the recurrent layer at the end of the sequence is then processed by another hidden layer of 100 ReLUs before the final softmax layer. We augment the ReLU RNN with a fast associative memory and compare it to an LSTM model with the same architecture. Although the original LSTMs do not have explicit long-term storage capacity, recent work from Danihelka et al. [2016] extended LSTMs by adding complex associative memory. In our experiments, we compared fast associative memory to both LSTM variants.

Figure 2 and Table 1 show that when the number of recurrent units is small, the fast associative memory significantly outperforms the LSTMs with the same number of recurrent units. The result fits with our hypothesis that the fast associative memory allows the RNN to use its recurrent units more effectively. In addition to having higher retrieval accuracy, the model with fast weights also converges faster than the LSTM models.

4.2 Integrating glimpses in visual attention models

Despite their many successes, convolutional neural networks are computationally expensive and the representations they learn can be hard to interpret. Recently, visual attention models [Mnih et al., 2014, Ba et al., 2015, Xu et al., 2015] have been shown to overcome some of the limitations in ConvNets. One can understand what signals the algorithm is using by seeing where the model is looking. Also, the visual attention model is able to selectively focus on important parts of visual space and thus avoid any detailed processing of much of the background clutter. In this section, we show that visual attention models can use fast weights to store information about object parts, though we use a very restricted set of glimpses that do not correspond to natural parts of the objects.

Given an input image, a visual attention model computes a sequence of glimpses over regions of the image. The model not only has to determine where to look next, but also has to remember what it has seen so far in its working memory so that it can make the correct classification later. Visual attention models can learn to find multiple objects in a large static input image and classify them correctly, but the learnt glimpse policies are typically over-simplistic: They only use a single scale of glimpses and they tend to scan over the image in a rigid way. Human eye movements and fixations are far more complex. The ability to focus on different parts of a whole object at different scales allows humans to apply the very same knowledge in the weights of the network at many different scales, but it requires some form of temporary memory to allow the network to integrate what it discovered in a set of glimpses. Improving the model's ability to remember recent glimpses should help the visual attention model to discover non-trivial glimpse policies. Because the fast weights can store all the glimpse information in the sequence, the hidden activity vector is freed up to learn how to intelligently integrate visual information and retrieve the appropriate memory content for the final classifier.

To explicitly verify that larger memory capacity is beneficial to visual attention-based models, we simplify the learning process in the following way: First, we provide a pre-defined glimpse control signal so the model knows where to attend rather than having to learn the control policy through reinforcement learning. Second, we introduce an additional control signal to the memory cells so the attention model knows when to store the glimpse information. A typical visual attention model is

[2]To make the architecture for this task more similar to the architecture for the next task we first compute a 50 dimensional embedding vector and then expand this to a 100-dimensional embedding.

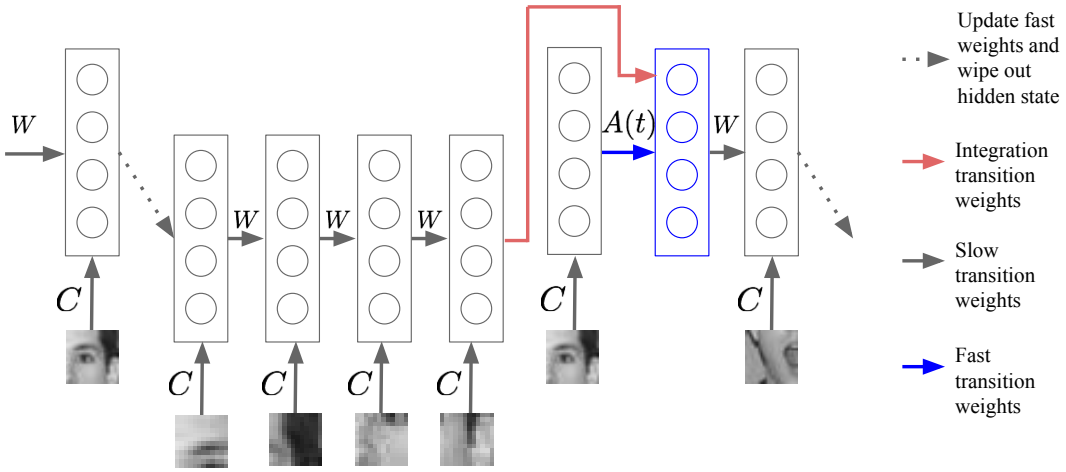


Figure 3: The multi-level fast associative memory model.

Model	50 features	100 features	200 features
IRNN	12.95%	1.95%	1.42%
LSTM	12%	1.55%	1.10%
ConvNet	**1.81%**	**1.00%**	**0.9%**
Fast weights	7.21%	1.30%	**0.85%**

Table 2: Classification error rates on MNIST.

complex and has high variance in its performance due to the need to learn the policy network and the classifier at the same time. Our simplified learning procedure enables us to discern the performance improvement contributed by using fast weights to remember the recent past.

We consider a simple recurrent visual attention model that has a similar architecture to the RNN from the previous experiment. It does not predict where to attend but rather is given a fixed sequence of locations: the static input image is broken down into four non-overlapping quadrants recursively with two scale levels. The four coarse regions, down-sampled to 7×7, along with their the four 7×7 quadrants are presented in a single sequence as shown in Figure 1. Notice that the two glimpse scales form a two-level hierarchy in the visual space. In order to solve this task successfully, the attention model needs to integrate the glimpse information from different levels of the hierarchy. One solution is to use the model's hidden states to both store and integrate the glimpses of different scales. A much more efficient solution is to use a temporary "cache" to store any of the unfinished glimpse computation when processing the glimpses from a finer scale in the hierarchy. Once the computation is finished at that scale, the results can be integrated with the partial results at the higher level by "popping" the previous result from the "cache". Fast weights, therefore, can act as a neurally plausible "cache" for storing partial results. The slow weights of the same model can then specialize in integrating glimpses at the same scale. Because the slow weights are shared for all glimpse scales, the model should be able to store the partial results at several levels in the same set of fast weights, though we have only demonstrated the use of fast weights for storage at a single level.

We evaluated the multi-level visual attention model on the MNIST handwritten digit dataset. MNIST is a well-studied problem on which many other techniques have been benchmarked. It contains the ten classes of handwritten digits, ranging from 0 to 9. The task is to predict the class label of an isolated and roughly normalized 28x28 image of a digit. The glimpse sequence, in this case, consists of 24 patches of 7×7 pixels.

Table 2 compares classification results for a ReLU RNN with a multi-level fast associative memory against an LSTM that gets the same sequence of glimpses. Again the result shows that when the number of hidden units is limited, fast weights give a significant improvement over the other

Figure 4: Examples of the near frontal faces from the MultiPIE dataset.

	IRNN	LSTM	ConvNet	Fast Weights
Test accuracy	81.11	81.32	88.23	86.34

Table 3: Classification accuracy comparison on the facial expression recognition task.

models. As we increase the memory capacities, the multi-level fast associative memory consistently outperforms the LSTM in classification accuracy.

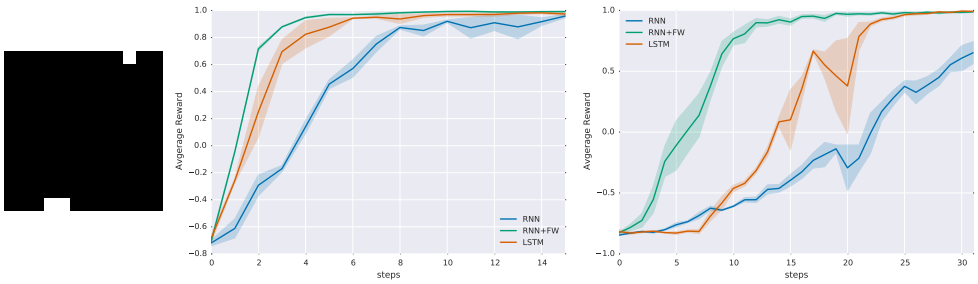
Unlike models that must integrate a sequence of glimpses, convolutional neural networks process all the glimpses in parallel and use layers of hidden units to hold all their intermediate computational results. We further demonstrate the effectiveness of the fast weights by comparing to a three-layer convolutional neural network that uses the same patches as the glimpses presented to the visual attention model. From Table 2, we see that the multi-level model with fast weights reaches a very similar performance to the ConvNet model without requiring any biologically implausible weight sharing.

4.3 Facial expression recognition

To further investigate the benefits of using fast weights in the multi-level visual attention model, we performed facial expression recognition tasks on the CMU Multi-PIE face database [Gross et al., 2010]. The dataset was preprocessed to align each face by eyes and nose fiducial points. It was downsampled to 48×48 greyscale. The full dataset contains 15 photos taken from cameras with different viewpoints for each illumination $\times$ expression $\times$ identity $\times$ session condition. We used only the images taken from the three central cameras corresponding to $-15^\circ, 0^\circ, 15^\circ$ views since facial expressions were not discernible from the more extreme viewpoints. The resulting dataset contained $> 100,000$ images. 317 identities appeared in the training set with the remaining 20 identities in the test set.

Given the input face image, the goal is to classify the subject's facial expression into one of the six different categories: neutral, smile, surprise, squint, disgust and scream. The task is more realistic and challenging than the previous MNIST experiments. Not only does the dataset have unbalanced numbers of labels, some of the expressions, for example squint and disgust, are are very hard to distinguish. In order to perform well on this task, the models need to generalize over different lighting conditions and viewpoints. We used the same multi-level attention model as in the MNIST experiments with 200 recurrent hidden units. The model sequentially attends to non-overlapping 12x12 pixel patches at two different scales and there are, in total, 24 glimpses. Similarly, we designed a two layer ConvNet that has a 12x12 receptive fields.

From Table 3, we see that the multi-level fast weights model that knows when to store information outperforms the LSTM and the IRNN. The results are consistent with previous MNIST experiments. However, ConvNet is able to perform better than the multi-level attention model on this near frontal face dataset. We think the efficient weight-sharing and architectural engineering in the ConvNet combined with the simultaneous availability of all the information at each level of processing allows the ConvNet to generalize better in this task. Our use of a rigid and predetermined policy for where to glimpse eliminates one of the main potential advantages of the multi-level attention model: It can process informative details at high resolution whilst ignoring most of the irrelevant details. To realize this advantage we will need to combine the use of fast weights with the learning of complicated policies.



(a) (b) (c)

Figure 5: a) Sample screen from the game "Catch" b) Performance curves for Catch with $N = 16, M = 3$. c) Performance curves for Catch with $N = 24, M = 5$.

4.4 Agents with memory

While different kinds of memory and attention have been studied extensively in the supervised learning setting [Graves, 2014, Mnih et al., 2014, Bahdanau et al., 2015], the use of such models for learning long range dependencies in reinforcement learning has received less attention.

We compare different memory architectures on a partially observable variant of the game "Catch" described in [Mnih et al., 2014]. The game is played on an $N \times N$ screen of binary pixels and each episode consists of N frames. Each trial begins with a single pixel, representing a ball, appearing somewhere in the first row of the column and a two pixel "paddle" controlled by the agent in the bottom row. After observing a frame, the agent gets to either keep the paddle stationary or move it right or left by one pixel. The ball descends by a single pixel after each frame. The episode ends when the ball pixel reaches the bottom row and the agent receives a reward of $+1$ if the paddle touches the ball and a reward of -1 if it doesn't. Solving the fully observable task is straightforward and requires the agent to move the paddle to the column with the ball. We make the task partially-observable by providing the agent blank observations after the Mth frame. Solving the partially-observable version of the game requires remembering the position of the paddle and ball after M frames and moving the paddle to the correct position using the stored information.

We used the recently proposed asynchronous advantage actor-critic method [Mnih et al., 2016] to train agents with three types of memory on different sizes of the partially observable Catch task. The three agents included a ReLU RNN, an LSTM, and a fast weights RNN. Figure 5 shows learning progress of the different agents on two variants of the game $N = 16, M = 3$ and $N = 24, M = 5$. The agent using the fast weights architecture as its policy representation (shown in green) is able to learn faster than the agents using ReLU RNN or LSTM to represent the policy. The improvement obtained by fast weights is also more significant on the larger version of the game which requires more memory.

5 Conclusion

This paper contributes to machine learning by showing that the performance of RNNs on a variety of different tasks can be improved by introducing a mechanism that allows each new state of the hidden units to be attracted towards recent hidden states in proportion to their scalar products with the current state. Layer normalization makes this kind of attention work much better. This is a form of attention to the recent past that is somewhat similar to the attention mechanism that has recently been used to dramatically improve the sequence-to-sequence RNNs used in machine translation. The paper has interesting implications for computational neuroscience and cognitive science. The ability of people to recursively apply the very same knowledge and processing apparatus to a whole sentence and to an embedded clause within that sentence or to a complex object and to a major part of that object has long been used to argue that neural networks are not a good model of higher-level cognitive abilities. By using fast weights to implement an associative memory for the recent past, we have shown how the states of neurons could be freed up so that the knowledge in the connections of a neural network can be applied recursively. This overcomes the objection that these models can only do recursion by storing copies of neural activity vectors, which is biologically implausible.

References

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Geoffrey E Hinton and David C Plaut. Using fast weights to deblur old memories. In *Proceedings of the ninth annual conference of the Cognitive Science Society*, pages 177–186. Erlbaum, 1987.

J Schmidhuber. Reducing the ratio between learning complexity and number of time varying variables in fully recurrent nets. In *ICANN93*, pages 460–463. Springer, 1993.

Misha Tsodyks, Klaus Pawelzik, and Henry Markram. Neural networks with dynamic synapses. *Neural computation*, 10(4):821–835, 1998.

LF Abbott and Wade G Regehr. Synaptic computation. *Nature*, 431(7010):796–803, 2004.

Omri Barak and Misha Tsodyks. Persistent activity in neural networks with dynamic synapses. *PLoS Comput Biol*, 3(2):e35, 2007.

Robert S Zucker and Wade G Regehr. Short-term synaptic plasticity. *Annual review of physiology*, 64(1): 355–405, 2002.

Henry Markram, Joachim Lübke, Michael Frotscher, and Bert Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps. *Science*, 275(5297):213–215, 1997.

Guo-qiang Bi and Mu-ming Poo. Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *The Journal of neuroscience*, 18(24):10464–10472, 1998.

David J Willshaw, O Peter Buneman, and Hugh Christopher Longuet-Higgins. Non-holographic associative memory. *Nature*, 1969.

Teuvo Kohonen. Correlation matrix memories. *Computers, IEEE Transactions on*, 100(4):353–359, 1972.

James A Anderson and Geoffrey E Hinton. Models of information processing in the brain. *Parallel models of associative memory*, pages 9–48, 1981.

John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

Elizabeth Gardner. The space of interactions in neural network models. *Journal of physics A: Mathematical and general*, 21(1):257, 1988.

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

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1819–1827, 2015.

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

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, 2015.

J. Ba, R. Kiros, and G. Hinton. Layer normalization. arXiv:1607.06450, 2016.

D. Kingma and J. L. Ba. Adam: a method for stochastic optimization. arXiv:1412.6980, 2014.

Ivo Danihelka, Greg Wayne, Benigno Uria, Nal Kalchbrenner, and Alex Graves. Associative long short-term memory. *arXiv preprint arXiv:1602.03032*, 2016.

V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu. Recurrent models of visual attention. In *Neural Information Processing Systems*, 2014.

J. Ba, V. Mnih, and K. Kavukcuoglu. Multiple object recognition with visual attention. In *International Conference on Learning Representations*, 2015.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, 2015.

Ralph Gross, Iain Matthews, Jeffrey Cohn, Takeo Kanade, and Simon Baker. Multi-pie. *Image and Vision Computing*, 28(5):807–813, 2010.

A. Graves. Generating sequences with recurrent neural networks. arXiv:1308.0850, 2014.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, 2016.

Supplementary Material

A Experimental details

A.1 Associative retrieval

We used a single hidden layer recurrent neural network which takes a 100 dimensional embedding vector as its input. We compared the fast weights memory against three other different RNN architecture: IRNN, standard LSTM and associative LSTM. The non-recurrent slow recurrent weights are initialized from uniform distribution between $(-1/\sqrt{H}, 1/\sqrt{H})$, where H is the number outgoing weights. The slow weights learning rate is tuned using the 10,000 validation examples.

Below, we provide the specific hyper-parameter settings for the models used in the experiments:

Fast weights: The fast weights learning rate, η, is set to 0.5 and the fast weights decay rate, λ, is set to 0.9. The fast weights are updated once at every time step. We experimented with more iterations for the "inner loop" and the performance are similar. The recurrent slow weights are initialized to an identity matrix scaled by 0.05. We use the ReLU activation for $f(\cdot)$ in the recurrent layer.

IRNN: The recurrent slow weights are initialized to an identity matrix scaled by 0.5. ReLU is used as the non-linearity in the recurrent layer.

Associative LSTM: We used 4 copies of memory cells for the associative LSTM. There are 3 read-write heads used for storage and retrieval memory access.

A.2 Integrating glimpses in visual attention models: MNIST and Facial expression recognition

Both tasks used the similar parameter initialization and the hyper-parameter setup that are comparable to the associative retrieval task mentioned above.

A.3 Agents with memory

All agents used recurrent networks to represent the policy. At each time step the input was passed through a hidden layer with 128 ReLU units and then passed to the recurrent core. All agents used 128 recurrent cells. The output at every step was a softmax over the valid actions and a single linear output for the estimate of the value function. We used random search to find hyperparameters values for the learning rate, the number of Hebbian steps, and fast weight learning rate and decay where applicable. We averaged results over the top 5 models.

B Implementing the fast weights "inner loop" in biological neural networks

We considered two different ways of performing this inner loop settling. In method 1 (which is what we use) the inputs to the hidden units after an outer loop transition using W are stored and provide sustained boundary conditions during the inner loop settling. In method 2 (which is more biologically plausible) we simply add the identity matrix to the fast weight matrix so that the inner loop settling tends to sustain the hidden activity vector. For ReLUs, these two methods are equivalent when the fast weight matrix is zero . They are similar but not exactly equivalent when the fast weights are non-zero. Using layer normalization, we found that method 1 worked slightly better than Method 2, but Method 2 would be much easier to implement in a biological network.