Gated Fast Weights for On-The-Fly Neural Program Generation

Imanol Schlag
The Swiss AI Lab IDSIA / USI / SUPSI
imanol@idsia.ch

Jürgen Schmidhuber
The Swiss AI Lab IDSIA / USI / SUPSI
juergen@idsia.ch

Abstract

We improve previous end-to-end differentiable neural networks (NNs) with fast weight memories. A gate mechanism updates fast weights at every time step of a sequence through two separate outer-product-based matrices generated by slow parts of the architecture. The system is trained on a complex sequence to sequence variation of the Associative Retrieval Problem with roughly 50 times more temporal memory (i.e. time-varying variables) than similar-sized standard recurrent NNs (RNNs). In terms of accuracy and number of parameters, our architecture outperforms a variety of RNNs, including Long Short-Term Memory, Hypernetworks, and related fast weight architectures. We relate this to meta-learning through an experiment which shows how the slow weights can learn an on-line learning program which can generate a smaller program able to answer a set of queries.

1 Introduction and Related Work

Recurrent Neural Networks (RNNs) are general parallel-sequential computers that can implement algorithms which map input sequences to output sequences. In practical applications, most RNNs are *Long Short-Term Memory* (LSTM) networks [12, 7, 9, 23], now used billions of times per day for automatic translation [27, 16], speech recognition [17], and many other tasks [23]. However, both plain RNNs and LSTMs have difficulties with certain memorization tasks, such as copying long input sequences [28], or various high-level cognitive tasks [14, 4].

Here we explore a generalization of the Associative Retrieval problem. We follow Danihelka et. al. [4] but turn the task into a general sequence to sequence problem and substantially increased its complexity. The underlying mechanism behaves like a dictionary that communicates a bounded number of key-value pairs using a syntax of storage and query tokens. To overcome current RNN limitations on this task, we propose a fast weight architecture able to learn and generalize using many fewer parameters.

Networks with non-differentiable fast weights or "dynamic links" have been published since 1981 and emerged out of biological evidence and the efforts of storing activation patterns in the weights of an associative network [25, 6, 11]. Subsequent work showed that a slow network can use gradient descent learning to control fast weights of a separate network (or of itself) in end-to-end differentiable fashion [19, 21, 20]. Recently there has been a resurgence of fast weight-inspired architectures [1, 10, 3, 15].

Large standard RNNs have a very small ratio between the numbers of time-varying variables (the activations of their units) and gradient-descent-learnable parameters (their weights) [21]. In fast weight systems, however, the number of time-varying variables (including the fast weights) can exceed the number of learnable parameters. This can help to greatly reduce model complexity.

Workshop on Meta-Learning, @NIPS 2017, Long Beach, CA, USA.

We improved the update mechanism through which the slow network learns to write its fast weight memory. The fast weights can be viewed as a program, efficiently computed on the fly by the slow net.

2 Method

Our architecture consists of the two networks s and f which both operate on the input sequence in parallel. The fast network f outputs the targets while the slow network s generates on-the-fly weight-updates for f using two separate outer-product-based matrices and a gating mechanism. s is called the slow network because its weights change only after every mini-batch according to the gradient-based learning algorithm. f, on the other hand, is called the fast network because its weights can change after every time step.

For simplicity, we chose both to be standard RNNs. The following formulas are for a single sample or a batch size of 1. Uppercase letters refer to real-valued matrices while lower case letters refer to vectors. Biases, a shared input embedding, and a shared output projection are omitted for simplicity. The weights of the slow network are abbreviated by S and the generated weights for the time-varying fast network by F_t. In both cases we use a 2-layer transition RNN (but other recurrent architectures could be used).

Recall the basic RNN formulation,

$$h_{t+1} = \phi(W[h_t; x_t]) \tag{1}$$

where h is the hidden state, ϕ is a non-linear activation function such as $\tanh$, x_t is the current input from the input sequence $X = (x_1, x_2, ..., x_T)$, and the weights $W \in \mathbb{R}^{m \times n}$ are fixed parameters.

Analogously, we define the the slow network $s(x_t, h_t^S)$:

$$[z_t^S; \Delta_t^{(1)}; \Delta_t^{(2)}] = S^{(2)} \tanh(S^{(1)}[h_t^S; x_t]) \tag{2}$$

$$h_{t+1}^S = \tanh(z_t^S) \tag{3}$$

Where h^S is the hidden state of the slow network, $S^{(1)} \in \mathbb{R}^{p \times o}$ and $S^{(2)} \in \mathbb{R}^{p \times p}$ are ordinary weights, and $\Delta_t^{(1)} \in \mathbb{R}^{2(n+m)}$ and $\Delta_t^{(2)} \in \mathbb{R}^{(4m)}$ are the update representations for $F_{t+1}^{(1)}$ and $F_{t+1}^{(2)}$, respectively.

Similarly, we define our fast network $f_t(x_t, h_t^F)$:

$$h_{t+1}^F = \mathcal{LN}(\tanh(F_t^{(2)} \mathcal{LN}(\tanh(F_t^{(1)}[h_t^F; x_t])))) \tag{4}$$

Where h^F is the hidden state of the fast network, $\mathcal{LN}(.)$ refers to layer normalization (LN) as introduced in previous work [2], and $F^{(1)} \in \mathbb{R}^{m \times n}$ and $F^{(2)} \in \mathbb{R}^{m \times m}$ are generated weights which can change after every step.

For each update representation Δ_t we compute the corresponding F_{t+1} as follows:

$$[\alpha_t; \beta_t; \gamma_t; \delta_t] = \Delta_t \tag{5}$$

$$H_t = \tanh(\alpha_t) \tanh(\beta_t)^T \tag{6}$$

$$T_t = \sigma(\gamma_t)\sigma(\delta_t)^T \tag{7}$$

$$F_{t+1} = T_t \odot H_t + (1 - T_t) \odot F_t \tag{8}$$

Where H and T are outer products to generate weight matrices in a Hebb-like manner [19] and have the same dimensionality as the respective F, $\odot$ is the element-wise product, and 1 denotes a matrix of ones. We observed instabilities in the beginning of training, attributing this to extreme weight updates of the not yet trained slow network. However, using LN after the activation function mitigates this problem while also improving generalization. LN before the activation led to inferior results. In equation 8, a gating matrix blends the current fast matrix with a matrix update, like in the gating mechanism for activations in a highway network [24]. We also evaluated other multiplicative and additive interactions, using more than one matrix, but achieved best results with the specific update mechanism above.

Note that at a given time, the slow network s is generating weights for the next time step. This prevents s from learning a prediction which bypasses f. We did not experiment with update delays greater than one step.

3 Experiments

Dataset We created a challenging sequence to sequence task based on the Associative Retrieval Problem, using a simple syntax of storage and query tokens. The network must emit correct responses to query tokens, and a default value in response to storage tokens. Since non-trivial problem-relevant outputs are rather sparse, our performance measure called *partial accuracy* ignores the trivial outputs. However, cost and respective gradients are computed over all outputs.

Model Our final architecture uses an embedding size of 15. The fast network is defined such that $h^F \in \mathbb{R}^{40}$, $F^{(1)} \in \mathbb{R}^{55\times 40}$, $F^{(2)} \in \mathbb{R}^{40\times 40}$ and the slow network such that $h^S \in \mathbb{R}^{40}$, $S^{(1)} \in \mathbb{R}^{55\times 100}$, $S^{(2)} \in \mathbb{R}^{100\times 394}$, both using a shared embedding of $\mathbb{R}^{15\times 15}$ and a shared output projection $W \in \mathbb{R}^{40\times 15}$ summing up to 46'234 trainable parameters (biases not listed for simplicity). We aim for a small, context-specific fast network and a slow network big enough to support a wide variety of fast network configurations while using as few weights as possible. Although the slow network has more weights than the fast one, it contains only 40 time-varying variables, namely the state vector h^S. The fast network, however, has 3840 time-varying variables (h^F, $F^{(1)}$, and $F^{(2)}$). This greatly increases the ratio between the numbers of time-varying variables and gradient-descent-learnable parameters [21].

Different configurations are possible. A larger s seems to converge faster but does not improve performance.

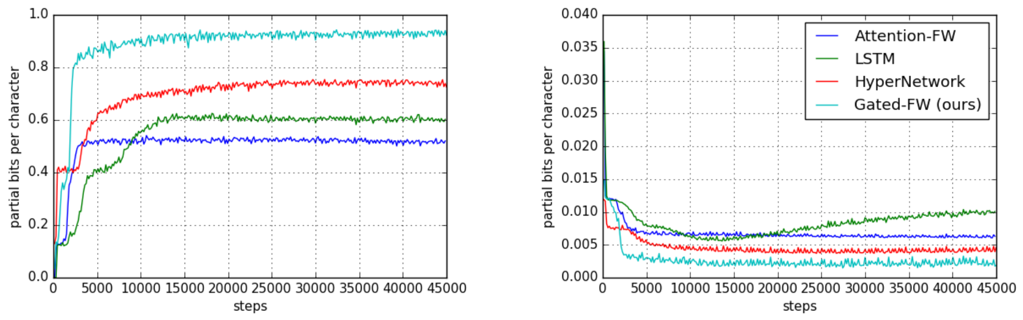

Figure 1: The left figure represents the accuracy over non-trivial targets and the right figure the respective bits per character. These are the validation set results of the best models of the four examined architectures due to our hyper parameter search.

Results We provide our best experimental results for other architectures for which we performed a hyper parameter search using a mix of grid and random search over each architecture's individual parameters. We compare our architecture (Gated-FW) to the LSTM, fast weights as a means of attention to the recent past (Attention-FW) [1], and HyperNetworks [10]. We also performed experiments with previous feed-forward and recurrent fast weight systems [19, 22] but were unable to get them to work well on this task. We trained all models using a sequence length of 32 and a batch size of 256. We didn't perform learning rate decay or similar strategies but included the learning rate in our hyper parameter search. In the end, a learning rate of 0.002 achieved over all architectures yielded best results, and was fixed in a second phase for all models, to allow for a comparison of convergence qualities. For all models and experiments, we used the Nesterov accelerated Adam (Nadam) [5]. Adam achieved similar performance but tended to converge slower than Nadam.

Our model achieves the highest partial accuracy and the lowest partial bits per character (BPC) while using 83 times fewer parameters than the next best model. Again, "partial" refers only to the non-space targets.

Program Generation Experiment Meta-Learning is the process of learning to learn [18]. Inspired by previous work on RNN-based meta-learning [22, 13, 8], we show how our slow network can learn a simple on-line learning algorithm for our fast net. We consider the 20 bAbI tasks [26], a set of synthetic question answering datasets, designed to test various reasoning capabilities. bAbI is a

Table 1: The test set results of the best models on our Associative Retrieval Problem.

Model	Total Accuracy	Partial Accuracy	Total BPC	Partial BPC	Parameters
Attention-FW	0.9922	0.5323	0.0274	0.0063	100'140
LSTM	0.9936	0.6252	0.0267	0.0061	1'487'640
Hypernetwork	0.9963	0.7804	0.0137	0.0031	3'848'215
Gated-FW (ours)	0.9979	**0.9522**	0.0149	**0.0016**	**46'234**

widely used benchmark for neural networks with external memory mechanisms. Each bAbI task involves 10k training samples, each consisting of a story sequence, a query sequence and a single prediction.

We slightly modify our architecture as follows. During the story sequence we run the slow and fast networks in parallel. Then we reset the hidden state of the fast network h_t^F to zeros and stop running the slow network. Then only the fast network processes the query sequence to produce the desired output in the end. Now every sample in the training set consists of an individual training sequence, the story, and an individual test sequence, the query with the final output.

That is, after the story sequence, the weights of the fast network are frozen. The only way of transferring information from the story is through fast weight generation prior to the query sequence. To solve this problem, the system has to transfer the important information from the story into a program able to correctly answer a variety of queries, given the story.

4 Conclusion

We provide a new way of updating the weight matrix of a fast NN through a slow RNN controlling a gate and two outer product-based matrices. We also introduce a more complex variation of the Associative Retrieval Problem which requires the system to store a number of associations from the input sequence, to retrieve them if necessary, and to forget them in favor of new associations. Our so-called Gated Fast Weights Architecture significantly increases the number of time-varying variables per learnable parameter, and outperforms other architectures in terms of convergence, accuracy, and number of parameters.

Acknowledgments

We are grateful to NVIDIA Corporation for providing a DGX-1 computer to IDSIA as part of the Pioneers of AI Research award.

References

[1] J. Ba, G. E. Hinton, V. Mnih, J. Z. Leibo, and C. Ionescu. Using fast weights to attend to the recent past. In *Advances In Neural Information Processing Systems*, pages 4331–4339, 2016.

[2] L. J. Ba, R. Kiros, and G. E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[3] B. D. Brabandere, X. Jia, T. Tuytelaars, and L. V. Gool. Dynamic filter networks. *CoRR*, abs/1605.09673, 2016.

[4] I. Danihelka, G. Wayne, B. Uria, N. Kalchbrenner, and A. Graves. Associative long short-term memory. *CoRR*, abs/1602.03032, 2016.

[5] T. Dozat. Incorporating nestrov momentum into adam. In *International Conference on Learning Representations (ICLR2016)*. CBLS, April 2016. OpenReview.net ID: OM0jvwB8jIp57ZJjtNEZ.

[6] J. A. Feldman. Dynamic connections in neural networks. *Biological cybernetics*, 46(1):27–39, 1982.

[7] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

[8] F. J. Gomez and J. Schmidhuber. Evolving modular fast-weight networks for control. In W. Duch, J. Kacprzyk, E. Oja, and S. Zadrozny, editors, *Artificial Neural Networks: Biological Inspirations - ICANN 2005, LNCS 3697*, pages 383–389. Springer-Verlag Berlin Heidelberg, 2005.

[9] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for improved unconstrained handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5), 2009.

[10] D. Ha, A. Dai, and Q. Le. Hypernetworks. 2016.

[11] G. E. Hinton and D. C. Plaut. Using fast weights to deblur old memories. In *IN PROCEEDINGS OF THE 9TH ANNUAL CONFERENCE OF THE COGNITIVE SCIENCE SOCIETY*, pages 177–186. Erlbaum, 1987.

[12] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997. Based on TR FKI-207-95, TUM (1995).

[13] S. Hochreiter, A. S. Younger, and P. R. Conwell. Learning to learn using gradient descent. In *Lecture Notes on Comp. Sci. 2130, Proc. Intl. Conf. on Artificial Neural Networks (ICANN-2001)*, pages 87–94. Springer: Berlin, Heidelberg, 2001.

[14] L. Jing, Ç. Gülçehre, J. Peurifoy, Y. Shen, M. Tegmark, M. Soljacic, and Y. Bengio. Gated orthogonal recurrent units: On learning to forget. *CoRR*, abs/1706.02761, 2017.

[15] T. Munkhdalai and H. Yu. Meta networks. *CoRR*, abs/1703.00837, 2017.

[16] J. Pino, A. Sidorov, and N. Ayan. Transitioning entirely to neural machine translation. *Facebook Research Blog, 2017, https://code.facebook.com/posts/289921871474277/transitioning-entirely-to-neural-machine-translation/.*

[17] H. Sak, A. Senior, K. Rao, F. Beaufays, and J. Schalkwyk. Google voice search: faster and more accurate. *Google Research Blog, 2015, http://googleresearch.blogspot.ch/2015/09/google-voice-search-faster-and-more.html.*

[18] J. Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook. Diploma thesis, Inst. f. Inf., Tech. Univ. Munich, 1987. http://www.idsia.ch/˜juergen/diploma.html.

[19] J. Schmidhuber. Learning to control fast-weight memories: An alternative to recurrent nets. *Neural Computation*, 4(1):131–139, 1992.

[20] J. Schmidhuber. A neural network that embeds its own meta-levels. In *Proc. of the International Conference on Neural Networks '93, San Francisco*. IEEE, 1993.

[21] J. Schmidhuber. On decreasing the ratio between learning complexity and number of time-varying variables in fully recurrent nets. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 460–463. Springer, 1993.

[22] J. Schmidhuber. A self-referential weight matrix. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 446–451. Springer, 1993.

[23] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. Published online 2014; 888 references; based on TR arXiv:1404.7828 [cs.NE].

[24] R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[25] von der Malsburg. *The Correlation Theory of Brain Function*. Internal report. Department of Neurobiology, Max-Planck-Institute for Biophysical Chemistry. 1981.

[26] J. Weston, A. Bordes, S. Chopra, and T. Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *CoRR*, abs/1502.05698, 2015.

[27] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *Preprint arXiv:1609.08144*, 2016.

[28] W. Zaremba and I. Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014.

Supplementary Material

A Model Visualization

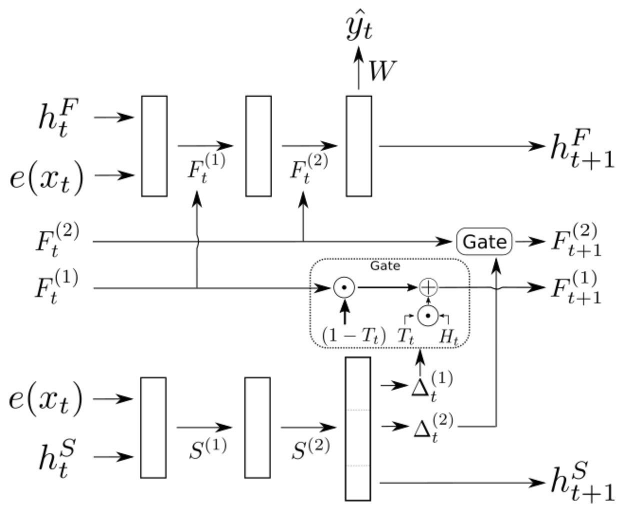

Figure 2: An informal pictogram which visualises the fast and slow network of our architecture at time step t. $e(x_t)$ refers to the embedding of x_t. Biases, activation functions, layernorm, and outer-products are not displayed.

B The Associative Retrieval Problem

Our version of the Associative Retrieval Problem consists of a simple syntax of storage and query tokens. All tokens are part of the inputs sequence while the generated answers to the query tokens are part of the output sequence. Storage tokens are key-value pairs while query tokens come with only a key to which the network must output the respective value according to a previously seen storage token. Whenever there is no query to respond to the network is supposed to output some default value which in our case is the empty space character. This means that non-trivial problem-relevant outputs are rather sparse.

Instead of measuring the accuracy of all outputs we use the *partial accuracy* which is the percentage of the correct outputs over all non-space characters. This is because all models learn very quickly to output spaces at every step which very quickly yields a high accuracy without actually having learned much.

The keys are 2 to 4 characters long while the respective values are a single character. All characters are uniformly sampled with replacement from a set of 8 possible ASCII characters (a to h). Each query token has to be a valid key which the network must have seen after the previous query token. Preceding every query token there are between 1 and 10 storage tokens. All query tokens and their respective storage tokens are then concatenated into one large sequence to not only value learning but also forgetting. The following is an example with only 2 queries with quotes to show the beginning and end of both sequences.

```
x: "S(hgb,c),S(ceaf,e),S(df,g),S(hac,b),Q(ceaf)e.S(hf,h),S(cc,d),Q(cc)d."
y: "                                           e                      d "
```

We generate and concatenate 100’000 queries for the training set and 5’000 queries for the test and validation set and use truncated Backpropagation Through Time (truncated BPTT) to train the models. This results in a single training sequence of roughly 5.7 million characters and a test and validation sequence of roughly 288’000 characters each.