ARMIN: Towards a More Efficient and Light-weight Recurrent Memory Network

Zhangheng Li[1,2] , **Jia-Xing Zhong**[1,2] , **Jingjia Huang**[1,2] , **Tao Zhang**[1,2] , **Thomas Li**[1] and **Ge Li**[1,2*]
[1]School of Electronic and Computer Engineering, Peking University
[2]Peng Cheng Laboratory
geli@ece.pku.edu.cn

Abstract

In recent years, memory-augmented neural networks(MANNs) have shown promising power to enhance the memory ability of neural networks for sequential processing tasks. However, previous MANNs suffer from complex memory addressing mechanism, making them relatively hard to train and causing computational overheads. Moreover, many of them reuse the classical RNN structure such as LSTM for memory processing, causing inefficient exploitations of memory information. In this paper, we introduce a novel MANN, the Auto-addressing and Recurrent Memory Integrating Network (ARMIN) to address these issues. The ARMIN only utilizes hidden state h_t for automatic memory addressing, and uses a novel RNN cell for refined integration of memory information. Empirical results on a variety of experiments demonstrate that the ARMIN is more light-weight and efficient compared to existing memory networks. Moreover, we demonstrate that the ARMIN can achieve much lower computational overhead than vanilla LSTM while keeping similar performances. Codes are available on github.com/zoharli/armin.

1 Introduction

Recurrent neural networks, such as the Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997] and Gated Recurrent Unit (GRU) [Cho *et al.*, 2014] have shown good performance for processing sequential data. However, it's known that RNNs suffer from gradient vanishing problem. Moreover, as pointed out by Rae $et\ al.$ [2016], the number of parameters grows proportionally to the square of the size of the hidden units, which carry the historical information. Recently, memory-augmented neural networks exhibit promising power to address these issues, by decoupling memory capacity from model parameters, $i.e.$ maintaining an external memory, and backpropagating the gradients through the memory.

Neural Turing Machine (NTM) [Graves *et al.*, 2014] first emerged as a recurrent model that incorporates external memory abilities. NTM maintains a memory matrix, and at every time-step, the network reads and writes (with erasing) to the memory matrix using certain soft-attentional mechanism, controlled by an LSTM that produces read and write vectors. NTM and its successor, the Differentiable Neural Computer[Graves *et al.*, 2016], have shown success on some algorithmic tasks such as copying, priority sorting and some real-world tasks such as question answering. But one limitation of the NTM is that due to its smooth read and write mechanism, NTM has to do propagations on the entire memory, which is not neccessary and may cause high computational overhead when the memory is in large-scale. However, these two external memory models have relatively complicated and hand-crafted memory addressing mechanisms, making the backpropagations through memory not straightforward and also causing high computational overheads. Moreover, they basically reuse the classical RNN structure such as LSTM for memory processing, which causes inefficient exploitations of memory information. The RNN in these models plays a simple role of being a controller, but the gradient vanishing problem of RNN itself is not given enough attention, which usually causes a degradation in the training speed and final performance.

Inspired by prior memory models, progresses have been made to build a bridge between simple RNNs and complicated memory models. Kurach $et\ al.$ [2015] propose the Neural Random-access Machines (NRAM) that can manipulate and dereference pointers to an external variable-size random-access memory. Danihelka $et\ al.$ [2016] improve LSTM with ideas from holographic reduced representations that enables key-value storage of data. Grave $et\ al.$ [2016] propose a method of augmenting LSTM by storing previous (hidden state, input word) pairs in memory and using the current hidden state as a query vector to recover historical input words. This method requires no backpropagation through memory and performs well on word-level language tasks.Grefenstette [2015], Dyer [2015], Joulin [2015] augment RNNs with a stack structure that works as a natural parsing tool, and use them to process algorithmic and nature language processing (NLP) tasks; nonetheless, the running speed of stack-augmented RNNs is rather slow due to multiple push-pop operations at every time-step. Rae $et\ al.$ [2016] proposes the Sparse Access Memory(SAM) network, by thresholding memory modifications to a sparse subset and

*Contact Author

using the approximate nearest neighbor (ANN) index to query the memory. Ke *et al.* [2018] propose the Sparse Attentive Backtracking (SAB) architecture, which recalls a few past hidden sates at every time-step and do "mental" backpropagations to the nearby hidden states with respect to the recalled hidden states. Gulcehre *et al.* [2017] propose the TARDIS network, which recalls a single memory entry at each time-step and use an LSTM-resembled RNN to process memory information. However, the TARDIS still involves some hand-crafted memory addressing methods which cause a considerable amount of computational overhead.

Based on the motivation of proposing a more efficient, light-weight and universal method of combining RNNs and MANNs, we introduce the ARMIN architecture, a recurrent MANN with a simple and straightforward memory addressing mechanism and a novel RNN cell for refined memory information processing. Concretely, our contributions are as follows:

- We propose a simple yet effective memory addressing mechanism for the external memory, namely Auto-addressing, by encoding the information for memory addressing directly via the inputs $\boldsymbol{x}_t$ and the hidden states $\boldsymbol{h}_{t-1}$.
- We propose a novel RNN cell that allows refined control of information flow and integration between memory and RNN structure. With only a single such cell, it achieves better performance than many hierarchical RNN structures in our character-level language modelling task.
- We show that the ARMIN is robust to small iteration lengths when training long sequential data, which enables training with large batch sizes and boost the training speed.
- We demonstrate competitive results on various tasks while keeping efficient time and memory consumption during training and inference time.

2 Background

2.1 Gumbel-softmax Estimator

Categorical distribution is a natural choice for representing discrete structure in the world. However, it's rarely used in neural networks due to its inability to backpropagate through samples [Jang *et al.*, 2016]. To this end, Maddison [2016] and Jang [2016] propose a continuous relaxation of categorical distribution and the corresponding gumbel-softmax gradient estimator that replaces the non-differentiable sample from a categorical distribution with a differentiable sample. Specifically, given a probability distribution $\boldsymbol{p} = (\pi_1, \pi_2, ..., \pi_k)$ over k categories, the gumbel-softmax estimator produces an one-hot sample vector $\boldsymbol{y}$ with its i-th element calculated as follows:

$$y_i = \frac{\exp((\log(\pi_i) + g_i)/\tau)}{\sum_{j=1}^{k} \exp((\log(\pi_j) + g_j)/\tau)} \quad \text{for } i = 1, 2, ..., k, \tag{1}$$

where $g_1, ..., g_k$ are i.i.d samples drawn from Gumbel distribution [Gumbel, 1954]:

$$g_i = -\log(-\log(u_i)) \, , u_i \sim \text{Uniform}(0, 1), \tag{2}$$

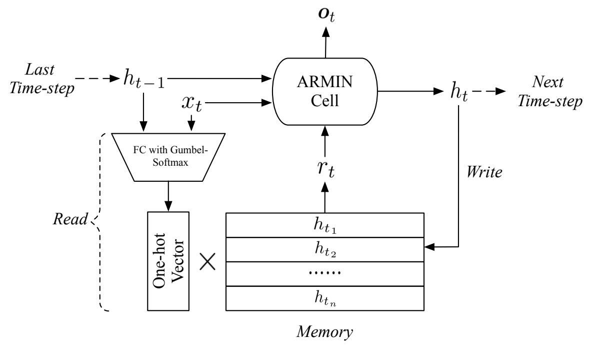


Figure 1: The ARMIN structure. At each time-step, the ARMIN performs read operation, cell processing and write operation in chronological order: **(a)** It reads out a historical hidden state $\boldsymbol{r}_t$ from memory with an one-hot read vector produced via passing $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ to a fully connected layer followed by a gumbel-softmax function. **(b)** The ARMIN cell receives $\boldsymbol{x}_t$, $\boldsymbol{h}_{t-1}$ and $\boldsymbol{r}_t$ as inputs and outputs $\boldsymbol{o}_t$ and $\boldsymbol{h}_t$. $\boldsymbol{o}_t$ is passed to output layers, and $\boldsymbol{h}_t$ is passed to next time-step. **(c)** $\boldsymbol{h}_t$ is written to the previous location of $\boldsymbol{r}_t$.

and τ is the $temperature$ parameter. In practice, we usually start at a high temperature and anneal to a small but non-zero temperature [Jang *et al.*, 2016]. The gumbel-softmax function have been used in many recurrent networks such as TARDIS [Gulcehre *et al.*, 2017], and Gumbel-Gate LSTM [Li *et al.*, 2018b].

3 Auto-addressing and Recurrent Memory Integrating Network

In this section, we describe the structure of the ARMIN network as shown in Figure 1. It consists of a recurrent cell and an external memory that stores historical hidden states. While processing sequential data, the ARMIN performs reading from memory, cell processing and writing to memory operations in chronological order during each time-step. In the following subsections, we first explain the structure of the recurrent cell, and then discuss the read and write operations.

3.1 The Recurrent Cell of ARMIN

By combining the gating mechanism of classical LSTM structure, we propose a novel recurrent cell structure for memory information processing, namely the ARMIN cell. At every time-step, it takes in an input $\boldsymbol{x}_t$, the last hidden state $\boldsymbol{h}_{t-1}$ and a recovered historical hidden state $\boldsymbol{r}_t$ chosen by a read operation, and produces an output vector $\boldsymbol{o}_t$ and the new hidden state $\boldsymbol{h}_t$. The computation process is as follows:

$$\begin{Bmatrix} \boldsymbol{g}_t^h \\ \boldsymbol{g}_t^r \end{Bmatrix} = \begin{Bmatrix} \sigma \\ \sigma \end{Bmatrix} \boldsymbol{W}_{ig}[\, \boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{r}_t \,] + \boldsymbol{b}_{ig} \, , \tag{3}$$

$$\boldsymbol{h}_{t-1}^g = \boldsymbol{g}_t^h \circ \boldsymbol{h}_{t-1} \, , \tag{4}$$

$$\boldsymbol{r}_t^g = \boldsymbol{g}_t^r \circ \boldsymbol{r}_t \, , \tag{5}$$

$$\begin{Bmatrix} \boldsymbol{i}_t \\ \boldsymbol{f}_t \\ \boldsymbol{g}_t \\ \boldsymbol{o}_t^h \\ \boldsymbol{o}_t^r \end{Bmatrix} = \begin{Bmatrix} \sigma \\ \sigma \\ \tanh \\ \sigma \\ \sigma \end{Bmatrix} \boldsymbol{W}_{go}[\, \boldsymbol{x}_t, \boldsymbol{h}_{t-1}^g, \boldsymbol{r}_t^g \,] + \boldsymbol{b}_{go} \;, \tag{6}$$

$$\boldsymbol{h}_t = \boldsymbol{f}_t \circ \boldsymbol{h}_{t-1} \;+ \boldsymbol{i}_t \;\circ \boldsymbol{g}_t \;, \tag{7}$$

$$\boldsymbol{o}_t = [\; \boldsymbol{o}_t^h \circ \tanh(\boldsymbol{h}_t) \;, \boldsymbol{o}_t^r \circ \tanh(\boldsymbol{r}_t) \;] \;. \tag{8}$$

where $\boldsymbol{h}_{t-1}, \boldsymbol{h}_t \in \mathbb{R}^{d_h}, \boldsymbol{r}_t \in \mathbb{R}^{d_r}, \boldsymbol{x}_t \in \mathbb{R}^{d_i}$, and $\boldsymbol{W}_{ig} \in \mathbb{R}^{2d_h \times (d_i+d_h+d_r)}, \boldsymbol{W}_{go} \in \mathbb{R}^{(4d_h+d_r)\times(d_i+d_h+d_r)}, \boldsymbol{b}_{ig} \in \mathbb{R}^{d_h+d_r}, \boldsymbol{b}_{go} \in \mathbb{R}^{4d_h+d_r}, \boldsymbol{o}_t \in \mathbb{R}^{d_h+d_r}$. We refer to d_h as the *hidden size* of the recurrent cell of ARMIN. Usually we have $d_h = d_r$ and allocate equal number of weight parameters for $\boldsymbol{h}_t$ and $\boldsymbol{r}_t$.

In equation 3 $\sim$ 5, two gates are calculated to control the information flow for $\boldsymbol{h}_{t-1}$ and $\boldsymbol{r}_t$ respectively, generating gated hidden state $\boldsymbol{h}_{t-1}^g$ and historical state $\boldsymbol{r}_t^g$. Then as shown in equation 6, we compute the input gate $\boldsymbol{i}_t$, forget gate $\boldsymbol{f}_t$, cell state $\boldsymbol{g}_t$ and output gate $\boldsymbol{o}_t^h$ for the new hidden state just like in classical LSTM structure. Additionally, an output gate $\boldsymbol{o}_t^r$ for historical state $\boldsymbol{r}_t$ is computed. Next in equation 7, we compute new hidden state $\boldsymbol{h}_t$ that is the sum of $\boldsymbol{h}_{t-1}$ and cell state $\boldsymbol{g}_t$, leveraged by forget gate $\boldsymbol{f}_t$ and input gate $\boldsymbol{i}_t$. Finally in equation 8, we calculate the output of this time-step, which is the concatenation of the gated contents from $\boldsymbol{h}_t$ and $\boldsymbol{r}_t$.

Intuitively, the $\boldsymbol{h}_{t-1}$ acts as the old working memory, and $\boldsymbol{r}_t$ is treated as the long-term memory. The cell processes them with the input $\boldsymbol{x}_t$ to generate the new working memory $\boldsymbol{h}_t$ and the output $\boldsymbol{o}_t$. More specifically, each $\boldsymbol{r}_t$ is a summary of historical hidden states selected by auto-addressing mechanism. The ARMIN cell learns to recurrently integrate the summary of long-term information from $\boldsymbol{r}_t$ into the working memory $\boldsymbol{h}_t$.

The main innovation of the ARMIN cell is using 3 element-wise and soft gates (*i.e.* $\boldsymbol{g}_t^h$, $\boldsymbol{g}_t^r$, and $\boldsymbol{o}_t^r$) to control the information flow for memory processing. *Firstly*, by using $\boldsymbol{g}_t^h$ and $\boldsymbol{g}_t^r$ gates, the network can dependently filter out the irrelevant information for current time-step from $\boldsymbol{h}_{t-1}$ and $\boldsymbol{r}_t$ in an element-wise fashion, and keep the useful information for later information integration. This refined control can avoid noise and make the weight parameters easier to be trained. In extreme cases, if the read operation chooses a completely useless or wrong $\boldsymbol{r}_t$, the network can shut down the $\boldsymbol{g}_t^r$ gate in the first place; if the network needs to reset hidden state to a historical state, it can shut down $\boldsymbol{g}_t^h$ gate and let $\boldsymbol{g}_t^r$ open. As a result, these two gate can bring more fault-tolerance for training read operation and increase flexibility of the RNN hidden state transmission. *Secondly*, by using the $\boldsymbol{o}_t^r$ gate, we can *selectively* output useful information from $\boldsymbol{r}_t$ for later computations. *Thirdly*, all gates in the ARMIN cell are element-wise and soft gates, which means we can smoothly apply existing RNN regularization techniques such as recurrent batch-norm and layer normalization to stabilize training process.

3.2 Read Operation with Auto-addressing

In this subsection, we introduce the auto-addressing mechanism. Specifically, the ARMIN maintains a memory matrix $\boldsymbol{M} \in \mathbb{R}^{n_{mem} \times d_h}$, where the constant n_{mem} denotes the number of memory slots. At each recurrence, the ARMIN chooses a historical state r_t from memory according to the information in $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$,which is formulated as follows:

$$\boldsymbol{s}_t = \text{gumbel-softmax}(\boldsymbol{W}_s[\, \boldsymbol{x}_t \;, \boldsymbol{h}_{t-1} \,] \;+\; \boldsymbol{b}_s) \;, \tag{9}$$

$$\boldsymbol{r}_t = \sum_{i=0}^{n-1} s_t(i) \boldsymbol{M}(i,:) \;. \tag{10}$$

where $\boldsymbol{W}_s \in \mathbb{R}^{n_{mem} \times (d_i+d_h)}$, $\boldsymbol{b}_s \in \mathbb{R}^{n_{mem}}$, $\boldsymbol{s}_t$ is a one-hot vector sampled by gumbel-softmax function, $s_t(i)$ denotes the i-th element of $\boldsymbol{s}_t$, $\boldsymbol{M}(i,:)$ denotes the i-th row of $\boldsymbol{M}$.

As opposed to previous fully-differentiable addressing mechanisms of recurrent MANNs such as NTM and TARDIS, the auto-addressing mechanism doesn't use the information from memory or any extra hand-crafted features to assist memory addressing, instead, it directly encodes the historical memory accessing information via the hidden state $\boldsymbol{h}_{t-1}$, which way shows sufficient power for memory addressing in our empirical evaluations. The time and space complexities of auto-addressing are only proportional to the size of $d_i + d_h$, whereas the complexities of previous addressing mechanism are usually proportional to the size of entire memory. Furthermore, the simple form of the auto-addressing mechanism makes the gradient flow more straightforward, and leads to a faster training convergence speed and better performance, as is shown in our experiments.

3.3 Write Operation

After the reading and cell processing stages, the ARMIN writes the new hidden state $\boldsymbol{h}_t$ to the memory $\boldsymbol{M}$. Following previous memory networks, we simply overwrite $\boldsymbol{h}_t$ to the memory slot where we just read out the $\boldsymbol{r}_t$ (for conditions where d_h is not equal to d_r, we first use a linear layer to transform $\boldsymbol{h}_t$ from d_h dimension to d_r dimension); but at the initial time-steps, we write the hidden states to the empty memory slots, until all empty slots are filled with historical states. Our network shares some similar ideas with the TARDIS network, including the discrete and one-hot memory addressing and memory overwriting mechanism, but our network has a simpler addressing mechanism and more flexible cell computations. For a detailed theoretical comparison between TARDIS and ARMIN, please refer to our supplemental material[1]. We also compare them across our experiments to validate the efficiency of our network.

4 Experiments

We evaluate our model mainly on algorithmic tasks, pMNIST task and character-level language modelling task, and additionally the temporal action proposal task (please refer to the supplemental material). We compare our network

[1]Please view it on https://sites.google.com/view/armin-network.

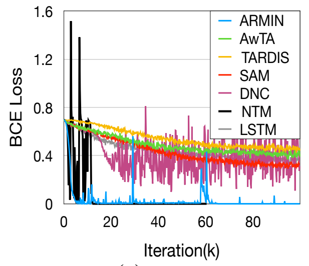
with many previous MANNs and vanilla LSTM networks[2]. As our network is designed based on LSTM and the GRU has similar performance/cost with LSTM, we omit the comparison with the GRU. We also directly compare the auto-addressing against the TARDIS addressing mechanism, by replacing auto-addressing with the TARDIS addressing method in ARMIN. We refer to this network as AwTA (ARMIN with TARDIS-Addr) in our experiments. Please refer to the supplemental material for the implementation details of our experiments.

4.1 Algorithmic Tasks

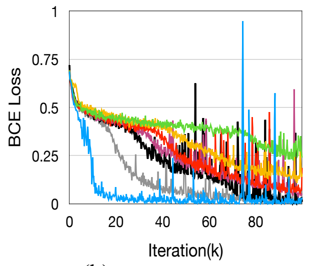
Along with the NTM, Graves *et al.* [2014] introduced a set of synthetic algorithm tasks to examine the efficiency of MANNs. Here we use 4 out of 5 of these tasks (due to the page limit, we exclude the N-gram task that is not adopted in previous works such as SAM and TARDIS) to examine if our network can choose correct time-steps from the past and effectively make use of them: *(a) copy*: copy a 6-bit binary sequence of length 1$\sim$50, *(b) repeat copy*: copy a sequence of length 1$\sim$10 for 1$\sim$10 times, *(c) associative recall* given a sequence of 2$\sim$6 (key,value) pairs and then a query with a key, return the associated value, *(d) priority sort*: given a sequence of 40 (key, priority value) pairs, return the top 30 keys in descending order of priority. We keep the parameter counts of MANNs roughly the same, and we use a strong LSTM baseline with about 4 times larger parameter count than MANNs. Following Gulcehre [2017], Campos [2017], in all tasks, we consider a task solved if the averaged binary cross-entropy loss of validation is at least two orders of magnitude below the initial loss which is around 0.70, *i.e.* the validation loss (the validation set is generated randomly) converges to less than 0.01 (with less than 30% in 10 consecutive validation of sharp losses that are higher than 0.01). In our evaluation, we are interested in if a model can successfully solve the task in 100k iterations and the iterations and elapsed training time till the model succeeds on the task. We run all experiments with the batch size of 1 under 2.8 GHz Intel Core i7 CPU, and report the wall clock time and the number of iterations for solved tasks. For models that fail to converge to less than 0.01 loss in 100k iterations, we report the average loss of the final 10 validations (denoted in underlines) and the elapsed training time for 100k iterations. The results are shown in Table 1. Please also see the supplemental material for the standard deviations of the losses of final 10 epochs of all models. The training curves for copy and priority sort tasks are shown in Figure 2.

The results show that the ARMIN can solve 3 out of 4 tasks with a fast converge speed in terms of both wall clock time and iteration numbers. By comparing the ARMIN with NTM, we observe that the NTM is able to almost solve all of the tasks, but with a much slower training speed compared to the ARMIN, for example, the training time of NTM is 5 times larger than ARMIN to solve the copy task. In fact, the training speed of ARMIN for each iteration is about 3 $\sim$ 4 times

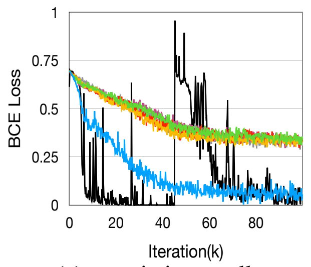
[2]We implement NTM based on github.com/vlgiitr/ntm-pytorch, and DNC, SAM based on github.com/ixaxaar/pytorch-dnc. For SAB network, we only compare available results in their paper because the official code is not released.



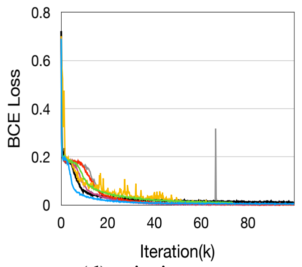
(a) copy



(b) repeat copy



(c) associative recall



(d) priority sort

Figure 2: The curves of validation losses on algorithmic tasks.

faster than the NTM's speed. We observe that the TARDIS and AwTA fail to solve 3 out of 4 tasks in the given 100k iterations. By comparing them with the ARMIN, *we confirm the efficiency of the auto-addressing of ARMIN*. The superiority of ARMIN in these tasks can be ascribed to the auto-addressing that favors a straightforward gradient propagation, and makes itself easier to learn, as in sec 3.2. We note that in the same loss range, ARMIN curves are smoother than other MANNs. The peaks at low loss are because part of the correct output array shifts left or right by one time-step, as explained in Graves [2014].

4.2 Permuted Pixel-by-pixel MNIST Classification

The Permuted pixel-by-pixel MNIST classification (pMNIST) task consists of predicting the handwritten digits of images on the MNIST dataset after being given its 784 pixels as a sequence permuted in a fixed random order. The model outputs its prediction of the handwritten digit at the last time step, so that it must integrate and remember observed information from previous pixels. We use this task to validate if our network can better store and integrate the information of previously observed pixels when the local structure of the image is destroyed by the pixel permutation. We run the experiments using double-precision floating point calculation under a TESLA K80 GPU and report the test set accuracy along with the time and memory consumption of the training stage.

The results are shown in Table 2. *ARMIN and DNC outperform other networks in classification accuracy, but the ARMIN saves 65.6% GPU memory and 62.5% training time compared to DNC.* We observe that ARMIN outperforms LSTM in terms of accuracy but not in terms of memory and time consumption. There indeed exists a performance/cost trade-off when comparing LSTM and ARMIN in sequence classification tasks such as pMNIST, because the memory ad-

	Hidden Size	**Mem. Size**	**Param. Count**	**Copy**		**Repeat Copy**		**Associative Recall**		**Priority Sort**	
				iter.	time	iter.	time	iter.	time	iter.	time
LSTM	300	–	376k	0.359	66	0.016	45	0.325	34	37.0	32
NTM	120	128×20	88k	12.4	32	0.014	171	19.6	22	0.012	360
DNC	120	128×20	104k	0.429	348	0.053	208	0.338	145	34.4	160
SAM	120	128×20	109k	0.321	145	0.064	87	0.326	64	78	40
TARDIS	120	50×32	90k	0.451	157	0.166	97	0.330	66	62.6	133
AwTA	100	50×32	91k	0.410	152	0.297	95	0.336	64	71.4	151
ARMIN	100	50×32	90k	7.6	6	67.8	36	0.052	33	33.4	37

Table 1: The average iterations(k) and elapsed time(min) of different networks till task solved in the given 100k iterations. The wall clock training time for 100k iterations and the average losses of final 10 validations (denoted in underlines) are shown for the unsolved tasks.

Model	**Test Acc.(%)**	**Hidden Size**	**Memory Size**	**Param. Count**	**GPU memory(GB)**	**Time (min/epoch)**
LSTM	92.7	128	–	69k	1.687	7
NTM	92.5	100	28×28	68k	2.912	40
DNC	94.3	100	28×28	70k	7.908	48
SAM	NaN	100	28×28	72k	–	–
SAB	94.2	–	–	–	–	–
TARDIS	94.1	100	28×28	74k	6.919	24
AwTA	94.1	100	28×28	81k	7.884	23
ARMIN	94.3	100	28×28	81k	2.713	15

Table 2: Test accuracy and memory/time consumption during training on pMNIST task.

dressing learning (which we assume is relatively hard compared with learning the parameters of LSTM) can only receive gradients from the classification label at the last timestep. However, this won't happen in sequence tagging tasks such as language modelling, as we will show in section 5.

4.3 Character-level Language Modelling

The character-level language modelling task consists of predicting the probability distribution of the next character given all the previous ones. We benchmark our network over the Penn Treebank and Hutter Prize Wikipedia (also known as $enwik8$) datasets. In this experiment, we also compare our network with some state-of-the-art RNN variants on these datasets, such as HM-LSTM [Chung *et al.*, 2016], HyperLSTM [Ha *et al.*, 2016], NASCell [Zoph and Le, 2016], IndRNN [Li *et al.*, 2018a], HyperRHN [Suarez, 2017] and FS-LSTM [Mujika *et al.*, 2017].

We ensure similar parameter counts and the same hyperparameters for all MANNs. Layer normalization (LN) [Ba *et al.*, 2016] and zoneout [Krueger *et al.*, 2016] are applied for the MANNs to avoid overfitting. Following prior works, we apply truncated backpropagation through time(TBPTT) [Rumelhart *et al.*, 1986; Elman, 1990] to approximate the gradients: at each iteration, the network predict the next 150 characters, and the hidden state $\boldsymbol{h}_t$ and memory state $\boldsymbol{M}$ are passed to the next iteration. The gradients are truncated between different iterations.

The results are shown in table 3. On Penn Treebank dataset, our best performing network achieves competitive 1.198 BPC on Penn Treebank dataset, which is the best single cell performance that we are aware of. *By comparing TARDIS and AwTA, we observe about 4 points of improvement, which shows the efficiency of the ARMIN cell. By comparing AwTA and ARMIN, we observe a further improvement of around 2 points, which shows the efficiency of the auto-addressing mechanism.* We also find that the same architecture of NTM and DNC that perform well in algorithmic tasks or pMNIST task fail to converge to a lower BPC than vanilla LSTM, which in turn shows the *generality* of ARMIN network.

The results on Penn Treebank show our single ARMIN cell learns better representations than many hierarchical RNN structures, such as the HM-LSTM, 2-Layer HyperLSTM and 21 layer IndRNN. Our network is outperformed by HyperRHN and FS-LSTM which are both multi-scale and deep transition RNNs and are state-of-the-art RNNs on this dataset. By "better representations", we refer to the concatenation of the gated contents from $\boldsymbol{h}_t$ and $\boldsymbol{r}_t$ as in equation 8. If we remove the gated contents of $\boldsymbol{r}_t$ from $\boldsymbol{o}_t$, the ARMIN undergoes a BPC performance drop from 1.198 to 1.220, which is still better than the best performing BPC of 1.24 of the LSTM. We believe the rest of the performance gain comes from the recurrent memory integration of the ARMIN cell, which also implicitly favors the function of deep transition, as is shown by the success of the deep transition RNNs on this task.

For more ablation study on the Penn Treebank dataset regarding the auto-addressing mechanism and the ARMIN cell, please refer to the supplemental material.

Model	PTB BPC	PTB Params	enwik8 BPC	enwik8 Params
LSTM	1.36	–	1.45	–
LSTM+Zoneout	1.27	–	–	–
LSTM+LN	1.267	4.26M	1.39	–
LSTM+LN+Zoneout	1.24	4.79M	1.37	–
HM-LSTM+LN	1.24	–	1.32	35M
HyperLSTM+LN	1.219	14.41M	1.340	26.5M
NASCell	1.214	16.28M	–	–
IndRNN (21 layers)	1.21	–	–	–
HyperRHN	1.19	15.5M	–	–
FS-LSTM+L-N+Zoneout	1.190	7.2M	1.277	27M
NTM(800 units)	1.535	8.28M	–	–
DNC(800 units)	1.390	8.20M	–	–
SAM(800 units)	NaN	9.48M	–	–
SAB(paper)	1.37	9.48M	–	–
TARDIS(paper)	1.25	–	–	–
TARDIS(1000 units)	1.268	9.2M	–	–
AwTA(800 units)	1.223	10.2M	–	–
ARMIN (500 units)	1.236	4.03M	–	–
ARMIN (800 units)	1.198	9.80M	–	–
ARMIN (800 units, 2 layer)	–	–	1.331	21.6M

Table 3: Bits-per-character on Penn Treebank and enwik8 test set. The lower is the better.

The result on enwik8 demonstrates a simple 2-layer ARMIN can achieve competitive BPC performance of 1.33, with less parameter count compared to the HyperLSTM and HM-LSTM. By constructing deeper ARMIN network or even combining with other multi-scale and hierarchical RNN architectures, we believe the performance can be further improved.

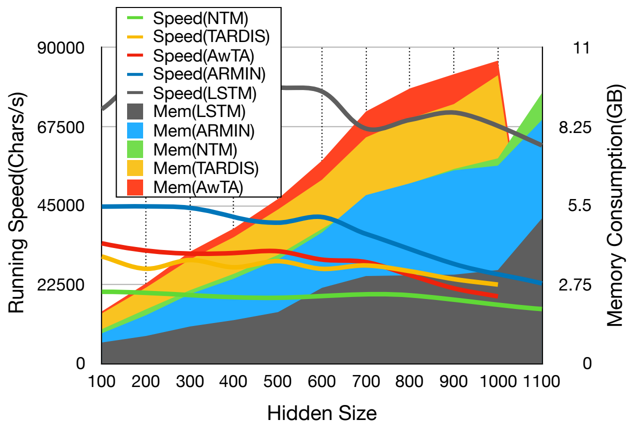
5 Towards a More Light-weight Recurrent Memory Network

Thorough Comparison of MANNs. In previous sections, we have shown in some of the algorithmic tasks that the ARMIN can be trained $3 \sim 4$ times faster than the NTM. In the next, we conduct a more thorough comparison among the vanilla LSTM and memory networks under different hidden units. Without the loss of generality, we do the benchmarks on Penn Treebank dataset and reuse the experimental setup as in section 4.3. We keep all memory matrices the same size, $i.e.$ $20 \times d_h$. We run the experiment using single-precision under a Titan XP GPU that has 12GB memory space. The results are depicted in figure 3. From the results we conclude that the ARMIN consistently outperforms other memory networks shown in the graph in terms of running speed both at training and inference stages, and the main contribution to this comes from the simple auto-addressing mechanism of the ARMIN. Moreover, at inference stage, *we can replace the memory matrix with a list of discrete memory slots, and update memory by simply replacing the old hidden states with the new ones, furthermore, we can replace the gumbel-softmax function with argmax*. Using these methods, ARMIN's inference speed obtains significant improvement than in training stage. At inference stage, we observe TARDIS has smaller memory consumption than ARMIN, this is due to the fact that TARDIS has smaller parameter counts under the same hidden size. Actually, TARDIS and ARMIN have roughly the same inference memory consumption under the same parameter counts.

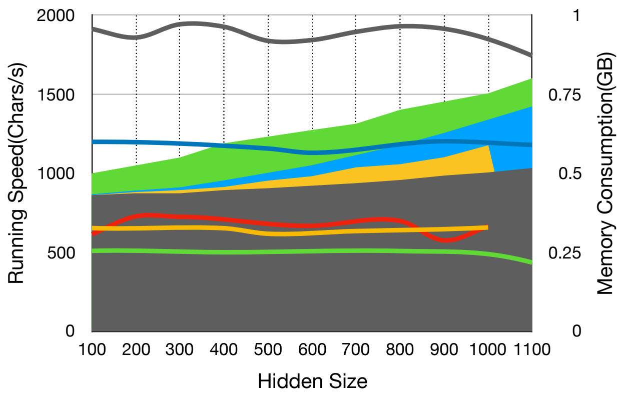
Model	LSTM		ARMIN	
Setup	1	2	1	2
Hidden size	1k	1k	500	550
n_{param}(M)	4.79	4.79	4.02	4.81
n_{mem}	–	–	5	10
T_{trunc}	100	150	50	50
batch size	128	128	384	300
Memory(GB)	2.36	3.27	3.49	3.56
Speed (chars/s)	71k	70k	98k	75k
BPC	1.27	1.24	1.238	1.223

Table 4: Comparison for performance and cost of different setups on Penn Treebank dataset.

Accelerating ARMIN by Short-length TBPTT. TBPTT is widely used in long sequential tasks to decrease training memory consumption and speed up the training process. However, it's obvious that a trade-off exists when applying TBPTT: the shorter the truncate length is, the worse the RNN performs. The subsequent sequences can only receive a single hidden state without gradient backpropagation enabled from their previous sequences, and thus, the RNNs can't access historical hidden states via backpropagations and effectively learn long-term dependencies. To alleviate this issue, we explore combining external memory with short-length TBPTT to allow the truncated sequences to receive more information from history while keeping the truncate length short, and thus, can accelerate the training process using large batch size while keeping good performance and similar memory consumption. Specifically, when we shorten the TBPTT length from 150 to 50 in the language-modelling task in section 4.3, we find the LSTM undergoes a big BPC performance drop from 1.24 to 1.39, but the ARMIN only has minor performance drop from 1.198 to 1.223, which shows the efficiency of combining the external memory with TBPTT. We further conduct an experiment to compare the ARMIN and LSTM under different setups. The hyperparameter setups and results are shown in Table 4. By comparing ARMIN setup 1 with LSTM setup 2, we observe that with only 6.7% more memory consumption, we *obtain* 40% *training speed gain* while keeping a slightly better BPC performance and 16% less parameter count; by comparing ARMIN setup 2 with LSTM setup 2, we observe that with only 8.8% more memory consumption, we obtain about 1.7 points of BPC performance gain and 7.1% training speed gain under similar parameter count. The results imply that on the long sequential tasks, we can use ARMIN to boost training speed or achieve better performance depending on the specific cases.



(a) Training (batch size=128)



(b) Inference (batch size=1)

Figure 3: The running speed and memory consumption at the training and inference stages ($n_{mem} = 20$). The solid blocks denote the memory consumption (GB) and the curves denote the running speed (characters/s). **(a)** shows the training stage, and **(b)** shows the inference stage. DNC and SAM is not shown in the graph, becuase 1.The time and memory consumption of DNC is worse than NTM. 2.SAM doesn't converge in our pMNIST and language modelling tasks. Note that AwTA has almost the same inference memory consumption with ARMIN.

6 Conclusion

In this paper, we introduce the ARMIN, a light-weight MANN with a novel ARMIN cell. The ARMIN incorporates an efficient external memory with the light-weight auto-addressing mechanism. We demonstrate competitive performance of ARMIN in various tasks, and the generality of our model. It's observed that our network is robust to short-length TBPTT which enables using large batch size to speed up the training while keeping good performance.

Acknowledgements

This work was supported in part by the Project of Shenzhen Municipal Science and Technology Program (No. JCYJ20170818141146428), in part by the National Engineering Laboratory for Video Technology-Shenzhen Division, in part by Shenzhen Key Laboratory for Intelligent Multimedia and Virtual Reality (No. ZDSYS201703031405467), and in part by National Natural Science Foundation of China and Guangdong Province Scientific Research on Big Data (No. U1611461). We appreciate our anonymous reviewers for their valuable comments. In addition, we would like to thank Jerry for English language editing.

References

[Ba *et al.*, 2016] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[Campos *et al.*, 2017] Víctor Campos, Brendan Jou, Xavier Giró-i Nieto, Jordi Torres, and Shih-Fu Chang. Skip rnn: Learning to skip state updates in recurrent neural networks. *arXiv preprint arXiv:1708.06834*, 2017.

[Cho *et al.*, 2014] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[Chung *et al.*, 2016] Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*, 2016.

[Danihelka *et al.*, 2016] Ivo Danihelka, Greg Wayne, Benigno Uria, Nal Kalchbrenner, and Alex Graves. Associative long short-term memory. In *Proceedings of the 33rd International Conference on ICML-Volume 48*, pages 1986–1994. JMLR. org, 2016.

[Dyer *et al.*, 2015] Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. Transition-based dependency parsing with stack long short-term memory. *arXiv preprint arXiv:1505.08075*, 2015.

[Elman, 1990] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[Grave *et al.*, 2016] Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. *arXiv preprint arXiv:1612.04426*, 2016.

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

[Graves *et al.*, 2016] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

[Grefenstette *et al.*, 2015] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in NIPS*, pages 1828–1836, 2015.

[Gulcehre *et al.*, 2017] Caglar Gulcehre, Sarath Chandar, and Yoshua Bengio. Memory augmented neural networks with wormhole connections. *arXiv preprint arXiv:1701.08718*, 2017.

[Gumbel, 1954] Emil Julius Gumbel. *Statistical theory of extreme values and some practical applications: a series of lectures*. Number 33. US Govt. Print. Office, 1954.

[Ha *et al.*, 2016] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

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

[Jang *et al.*, 2016] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[Joulin and Mikolov, 2015] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, pages 190–198, 2015.

[Ke *et al.*, 2018] Nan Rosemary Ke, Anirudh Goyal, Olexa Bilaniuk, Jonathan Binas, Michael C Mozer, Chris Pal, and Yoshua Bengio. Sparse attentive backtracking: Temporal creditassignment through reminding. *arXiv preprint arXiv:1809.03702*, 2018.

[Krueger *et al.*, 2016] David Krueger, Tegan Maharaj, János Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Aaron Courville, and Chris Pal. Zoneout: Regularizing rnns by randomly preserving hidden activations. *arXiv preprint arXiv:1606.01305*, 2016.

[Kurach *et al.*, 2015] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

[Li *et al.*, 2018a] Shuai Li, Wanqing Li, Chris Cook, Ce Zhu, and Yanbo Gao. Independently recurrent neural network (indrnn): Building a longer and deeper rnn. In *Proceedings of the IEEE Conference on CVPR*, pages 5457–5466, 2018.

[Li *et al.*, 2018b] Zhuohan Li, Di He, Fei Tian, Wei Chen, Tao Qin, Liwei Wang, and Tie-Yan Liu. Towards binary-valued gates for robust lstm training. *arXiv preprint arXiv:1806.02988*, 2018.

[Maddison *et al.*, 2016] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

[Mujika *et al.*, 2017] Asier Mujika, Florian Meier, and Angelika Steger. Fast-slow recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 5915–5924, 2017.

[Rae *et al.*, 2016] Jack Rae, Jonathan J Hunt, Ivo Danihelka, Timothy Harley, Andrew W Senior, Gregory Wayne, Alex Graves, and Tim Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes. In *Advances in NIPS*, pages 3621–3629, 2016.

[Rumelhart *et al.*, 1986] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[Suarez, 2017] Joseph Suarez. Language modeling with recurrent highway hypernetworks. In *Advances in Neural Information Processing Systems*, pages 3267–3276, 2017.

[Zoph and Le, 2016] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.