SEQUENCE MODELLING WITH MEMORY-AUGMENTED RECURRENT NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Processing sequential data with long term dependencies is a major challenge in many deep learning applications. In this paper, we introduce a novel architecture, the Memory-Augmented RNN (MARNN) to address this issue. The MARNN explicitly stores previous hidden states and makes use of them by an efficient memory addressing mechanism at every time-step. Compared to existing memory networks, the MARNN is more light-weight and allows direct backpropagation from output to memory. Our network can be trained on small slices of long sequential data, and thus, can theoretically boost training speed. We test the MARNN on two typical sequential modelling tasks. We achieve a competitive 1.202 Bits-per-character on the Penn Treebank character-level language modelling task, and achieve state-of-the-art performance of recall at high tIoUs on the THUMOS' 14 temporal action detection and proposal task.

1 INTRODUCTION

Recurrent neural networks, such as the Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) and Gated Recurrent Unit (GRU) (Cho et al., 2014) have shown promising 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. Recent memory-based approaches exhibit potential to address these issues, by decoupling memory capacity from model parameters, 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, usually causing huge amount of memory consumption. To this end, Rae et al. (2016) proposes the Sparse Access Memory(SAM) network, by thresholding memory modifications to a sparse subset, $i.e.$ all read and write operations are limited to several memory words. This allows memory and time efficient propagations while maintaining NTM's performance. However, these external memory models have relatively complicated memory addressing mechanisms, making them suitable for only large-scale memory. Moreover, the RNN in these models plays a simple role of being a controller, but the gradient vanishing problem of RNN itself is not given attention. In contrast to these networks, our network uses a light-weight read mechanism to address a small external memory, and relieves the gradient vanishing problem by directly concatenating memory word to output.

Inspired by prior memory models, efforts 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 (Plate, 2003) 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 is well-suited to word-level language tasks. Grefenstette et al. (2015); Dyer et al. (2015); Joulin & Mikolov (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 speed of stack-augmented RNNs is rather slow due to multiple push-pop operations at every time-step. Gulcehre et al. (2017) propose the TARDIS network, which is an LSTM-resembled RNN that directly stores a fixed number of previous hidden states. At every time-step, the network reads out one historical state $\boldsymbol{r}_t$ from the memory, and uses it, along with the input $\boldsymbol{x}_t$ and last hidden state $\boldsymbol{h}_{t-1}$, to produce new hidden state $\boldsymbol{h}_t$, then overwrites $\boldsymbol{h}_t$ to the location of $\boldsymbol{r}_t$. TARDIS optionally uses the gumbel-softmax estimator(Maddison et al., 2016; Jang et al., 2016) to sample the location of $\boldsymbol{r}_t$ (we will explain this in section 2.2), so the whole network is differentiable. However, the TARDIS still involves some hand-crafted memory addressing method and the speed and memory consumption of the whole network remains unclear.

Inspired by the TARDIS, we introduce the MARNN, a more light-weight and memory-based architecture with a very simple memory addressing mechanism and in-depth modification of the existing LSTM structure. Concretely, our contributions are as follows:

- We distill previous memory addressing methods and propose a simple yet effective memory addressing mechanism for the external memory, 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 recurrent cell that combines the gating advantages of LSTM and allows direct gradient backpropagation from output to memory. With only a single cell, it learns better representations than many hierarchical RNN structures.
- We show in section 4.1.2 that the MARNN is robust to small iteration lengths when training long sequential data, which theoretically enables training with very large batch sizes. Thus it can potentially boost training speed and also achieve better performance.
- We achieve competitive results both on language modelling task on the Penn Treebank dataset and video detection task on the THUMOS' 14 dataset while the time and memory consumption of the MARNN is only less than 2 times higher compared to the LSTM structure at inference time.

2 BACKGROUND

2.1 RNN AND LONG SHORT-TERM MEMORY

A recurrent neural network (RNN) is a class of neural network that recurrently processes a sequence of inputs $\{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_T\}$, and returns a sequence of outputs $\{\boldsymbol{y}_1, \boldsymbol{y}_2, ..., \boldsymbol{y}_T\}$ where $\boldsymbol{x}_t \in \mathbb{R}^{d_i}$, $\boldsymbol{y}_t \in \mathbb{R}^{d_o}$, and d_i, d_o is the input size and output size, respectively. In a vanilla RNN, $\boldsymbol{y}_i$ is given by following equations:

$$\boldsymbol{h}_t = \sigma(\boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{W}\boldsymbol{h}_{t-1} + \boldsymbol{b}) \ , \tag{1}$$

$$\boldsymbol{y}_t = \boldsymbol{V}\boldsymbol{h}_t + \boldsymbol{c} \ , \tag{2}$$

where the hidden state $\boldsymbol{h}_t \in \mathbb{R}^{d_h}$ is passed to the next recurrence. d_h is the hidden size, and $\boldsymbol{U}, \boldsymbol{W}, \boldsymbol{V}, \boldsymbol{b}, \boldsymbol{c}$ are learnable parameters.

The LSTM (Hochreiter & Schmidhuber, 1997) is designed to overcome the gradient vanishing and exploding problem in vanilla RNN and learn long-term dependencies, by passing a tuple of $(\boldsymbol{h}_t, \boldsymbol{c}_t)$ to the next recurrence. Its computation process is defined as follows:

$$\begin{Bmatrix} \boldsymbol{i}_t \\ \boldsymbol{f}_t \\ \boldsymbol{g}_t \\ \boldsymbol{o}_t \end{Bmatrix} = \begin{Bmatrix} \sigma \\ \sigma \\ \tanh \\ \sigma \end{Bmatrix} \boldsymbol{W}_0 [\ \boldsymbol{x}_t, \boldsymbol{h}_{t-1}\ ] + \boldsymbol{b}_0 \ , \tag{3}$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} \ + \boldsymbol{i}_t \circ \boldsymbol{g}_t \ , \tag{4}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh(\boldsymbol{c}_t), \tag{5}$$

where $\circ$ is the element-wise product, $\boldsymbol{W}_0 \in \mathbb{R}^{4d_h \times (d_i + d_h)}$, $\boldsymbol{b}_0 \in \mathbb{R}^{4d_h}$, $\boldsymbol{i}_t, \boldsymbol{f}_t, \boldsymbol{g}_t\ \boldsymbol{o}_t$ are the input gate, forget gate, cell state, output gate at time t, respectively, and they control the information flow in the LSTM.

2.2 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 et al. (2016) and Jang et al. (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:

$$y_i = \frac{\exp((\log(\pi_i) + g_i)/\tau)}{\sum_{j=1}^{k} \exp((\log(\pi_j) + g_j)/\tau)} \quad \text{for } i = 1, 2, ..., k, \tag{6}$$

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

$$g_i = -\log(-\log(u_i)) \, , u_i \sim \text{Uniform}(0, 1), \tag{7}$$

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).

3 MEMORY-AUGMENTED RNN

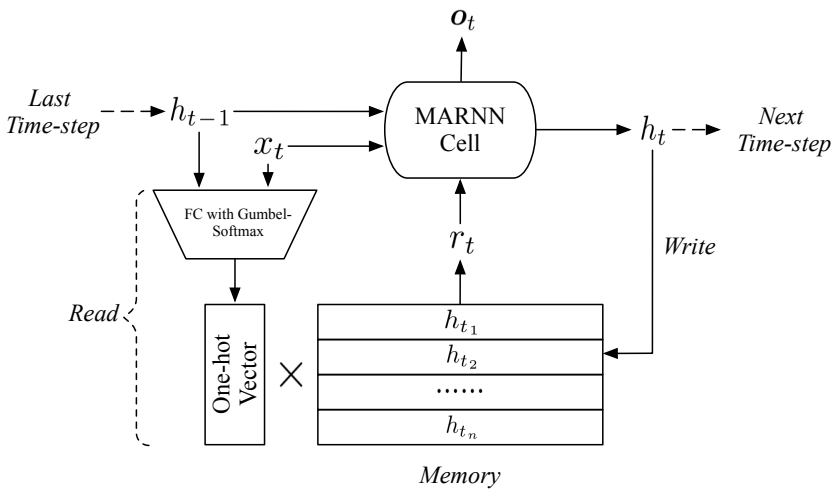


Figure 1: The MARNN structure. At each time-step, the MARNN performs read operation, cell processing and write operation in chronological order: **(a)** It reads out a historical hidden state $\boldsymbol{r}_t$ from memory with an one-hot read vector produced via passing $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ to a fully connected layer followed by a gumbel-softmax function. **(b)** The MARNN 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 $\boldsymbol{r}_t$.

In this section, we describe the structure of the MARNN network as shown in Figure 1. It consists of a recurrent cell and a small external memory that stores historical hidden states. While processing sequential data, the MARNN 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 MARNN

Inspired by classical LSTM structure, we propose a novel memory-augmented recurrent cell structure, which we call the MARNN 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:

$$\begin{Bmatrix} \boldsymbol{g}_t^h \\ \boldsymbol{g}_t^r \end{Bmatrix} = \begin{Bmatrix} \sigma \\ \sigma \end{Bmatrix} \boldsymbol{W}_{ig}[\, \boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{r}_t \,] + \boldsymbol{b}_{ig} \ , \tag{8}$$

$$\boldsymbol{h}_{t-1}^g = \boldsymbol{g}_t^h \circ \boldsymbol{h}_{t-1} \ , \tag{9}$$

$$\boldsymbol{r}_t^g = \boldsymbol{g}_t^r \circ \boldsymbol{r}_t \ , \tag{10}$$

$$\begin{Bmatrix} \boldsymbol{i}_t \\ \boldsymbol{f}_t \\ \boldsymbol{g}_t \\ \boldsymbol{o}_t^h \\ \boldsymbol{o}_t^r \end{Bmatrix} = \begin{Bmatrix} \sigma \\ \sigma \\ \tanh \\ \sigma \\ \sigma \end{Bmatrix} \boldsymbol{W}_{go}[\, \boldsymbol{x}_t, \boldsymbol{h}_{t-1}^g, \boldsymbol{r}_t^g \,] + \boldsymbol{b}_{go} \ , \tag{11}$$

$$\boldsymbol{h}_t = \boldsymbol{f}_t \circ \boldsymbol{h}_{t-1} \ + \boldsymbol{i}_t \ \circ \boldsymbol{g}_t \ , \tag{12}$$

$$\boldsymbol{o}_t = [\, \boldsymbol{o}_t^h \circ \tanh(\boldsymbol{h}_t) \ , \boldsymbol{o}_t^r \circ \tanh(\boldsymbol{r}_t) \,] \ . \tag{13}$$

where $\boldsymbol{h}_{t-1}, \boldsymbol{h}_t, \boldsymbol{r}_t \in \mathbb{R}^{d_h}, \boldsymbol{x}_t \in \mathbb{R}^{d_i}$, and $\boldsymbol{W}_{ig} \in \mathbb{R}^{2d_h \times (d_i + 2d_h)}, \boldsymbol{W}_{go} \in \mathbb{R}^{5d_h \times (d_i + 2d_h)}, \boldsymbol{b}_{ig} \in \mathbb{R}^{2d_h}, \boldsymbol{b}_{go} \in \mathbb{R}^{5d_h}, \boldsymbol{o}_t \in \mathbb{R}^{2d_h}$. We refer to d_h as the $hidden\ size$ of the recurrent cell of MARNN.

In equation 8 $\sim$ 10, two gates are calculated to control the information flow for h_{t-1} and r_t respectively, generating gated hidden state $\boldsymbol{h}_{t-1}^g$ and historical state $\boldsymbol{r}_t^g$; using this method, we can filter out the irrelevant information for the current time-step. Then as shown in equation 11, 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 12, 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 13, we calculate the output of this time-step, which is the concatenation of the gated contents from $\boldsymbol{h}_t$ and $\boldsymbol{r}_t$. Using this method, we can $directly\ backpropagate\ gradients$ from the output to the historical state $\boldsymbol{r}_t$. We involve $\boldsymbol{r}_t$ (or $\boldsymbol{r}_t^g$) in all gates' computations. 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 $\boldsymbol{o}_t$.

Unlike the LSTM that passes a tuple of $(\boldsymbol{h}_t, \boldsymbol{c}_t)$ to the next recurrence and outputs $\boldsymbol{h}_t$ at the same time, the MARNN only passes one hidden state h_t and separately uses $\boldsymbol{o}_t$ as output, which has 2 benefits: one is that it takes lower memory consumption to store one hidden state rather than two, and the other is we can decouple the information needed for the output $\boldsymbol{o}_t$, from the information ($i.e.$ $\boldsymbol{h}_t$) that is needed for memorizing and being used at later time-steps.

3.2 READ OPERATION

The MARNN 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 MARNN 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:

$$\boldsymbol{s}_t = \text{gumbel-softmax}(\boldsymbol{W}_s[\, \boldsymbol{x}_t \ , \boldsymbol{h}_{t-1} \,] \ + \ \boldsymbol{b}_s) \ , \tag{14}$$

$$\boldsymbol{r}_t = \sum_{i=0}^{n-1} s_t(i) \boldsymbol{M}(i, :) \ . \tag{15}$$

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 memory networks such as Santoro et al. (2016) and Gulcehre et al. (2017), we don't use any extra $usage$ vectors that manually encodes historical memory accessing information to assist memory addressing. As our ablation study (Section 4.1.2) shows, the hidden state $\boldsymbol{h}_{t-1}$ is sufficient to encode the historical memory accessing information.

3.3 WRITE OPERATION

After the reading and cell processing stages, the MARNN writes the new hidden state $\boldsymbol{h}_t$ to the memory $\boldsymbol{M}$. Following the TARDIS network (Gulcehre et al., 2017), we simply overwrite $\boldsymbol{h}_t$ to the memory slot where we just read out the $\boldsymbol{r}_t$. 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. In this way, we can maximally preserve useful historical information, because the MARNN cell can learn to copy the useful information from $\boldsymbol{r}_t$ to $\boldsymbol{h}_t$, and then write $\boldsymbol{h}_t$ to the previous location of $\boldsymbol{r}_t$. The read/write mechanism can be viewed as a form of a skip-connection of hidden states. By training the whole network, the MARNN can easily learn long-term dependencies via direct access to historical hidden states. From this point of view, our network is similar to the Skip RNN proposed by Campos Camunez et al. (2018), which differs from our network in that they use a binary state update gate to select whether the state of the RNN will be updated or copied from the previous time-step.

4 EXPERIMENTS

Our goal is to improve the performance of deep learning architectures that use RNNs as components, by replacing the RNNs with our MARNN without increasing the parameter count of the networks. We evaluate our model on the tasks of character-level language modelling and temporal action detection and proposals. Our model can be trained end-to-end, due to the utilization of gumbel-softmax function. As pointed out in Maddison et al. (2016), when the temperature $\tau \leq (n-1)^{-1}$, the density function of gumbel-softmax estimator is log-convex, which gives us a good guarantee for optimization. However, smaller τ will also bring higher variance of the gradients (Jang et al., 2016). So in all experiments, we initialize the reciprocal of τ as 1, and increase it by 1 after each epoch, until the reciprocal equals to $n-1$. This scheme has shown better performance than other schemes.

In all experiments, we apply Layer Normalization(Ba et al., 2016) to equation 8, 11 and 12, which we find important to regulate the hidden state. To avoid overfitting, we apply dropout (Srivastava et al., 2014) to the input and output layers, and Zoneout(Krueger et al., 2016) is applied for recurrent connections. The networks are trained with Adam optimizer (Kingma & Ba, 2014). All weight matrices are initialized to orthogonal matrices, and the bias of the forget gate $\boldsymbol{f}_t$ is initialized to 1.

4.1 CHARACTER-LEVEL LANGUAGE MODELLING ON PENN TREEBANK

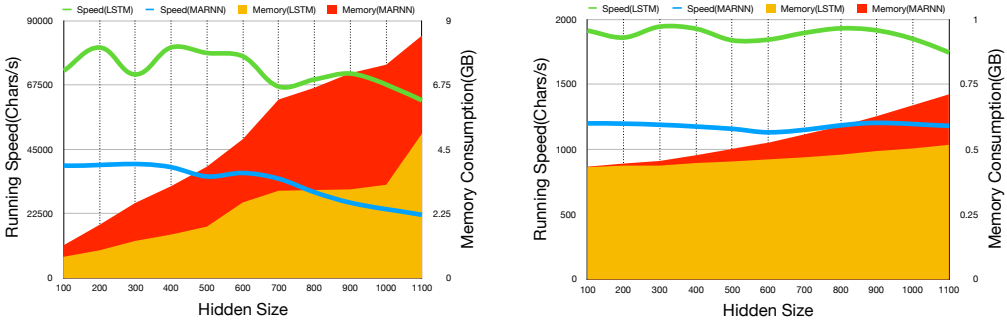
4.1.1 PERFORMANCE COMPARED WITH OTHER RNNS

The character-level language modelling task consists of predicting the probability distribution of the next character given all the previous ones, and Penn Treebank corpus is a widely used benchmark dataset for evaluating the performance of RNN variants. We use the train/validation/test split outlined in Mikolov et al. (2012). The train/validation/test batch size are 128/64/1 respectively. Following prior works on this task (Ha et al., 2016; Suarez, 2017; Mujika et al., 2017), the model is optimized over the cross-entropy loss between the predictions and the training labels, and evaluated using a bits-per-character measure. We use a single MARNN cell of different hidden sizes, with a fully connected layer to output the predictive probabilities. The input embedding size d_i is set to 128. The number of memory slots n_{mem} is 20. The zoneout probability is 0.3. We train the network for 200 epochs with a learning rate of 0.002, and decay with a factor of 10 at the last 20 epochs. We apply gradients norm clipping to regulate the training. 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.

We compare our network with vanilla LSTMs and the TARDIS, as well as some popular RNN variants on this task. See table 1. Our network achieves competitive 1.202 BPC on Penn Treebank dataset, which is the best single cell performance that we are aware of, with relatively small parameter count. The results show our single MARNN cell learns better representations than many hierarchical RNN structures. Additionally, it's known that the speed and computer memory consumption have important influences on the practicability of an RNN, especially when external memory is involved. Based on this, we conduct a comparison experiment between the vanilla LSTM and the MARNN of different hidden units. For a fair comparison, we implement both networks

Table 1: Bits-per-character on Penn Treebank test set.

Model	BPC	Param Count
LSTM	1.36	–
LSTM+Batch Norm (Cooijmans et al., 2016)	1.32	–
LSTM+Recurrent Dropout (Semeniuta et al., 2016)	1.30	–
LSTM+Zoneout (Krueger et al., 2016)	1.27	–
LSTM+Layer Norm(1000 units)[1]	1.267	4.26M
LSTM+Layer Norm+Zoneout(1000 units)[2]	1.24	4.26M
HM-LSTM+Layer Norm (Chung et al., 2016)	1.24	–
Layer Norm HyperLSTM (Ha et al., 2016)	1.25	4.92M
2-Layer Norm HyperLSTM (Ha et al., 2016)	1.219	14.41M
NASCell (Zoph & Le, 2016)	1.214	16.28M
IndRNN (21 layers) (Li et al., 2018)	1.21	–
TARDIS (Gulcehre et al., 2017)	1.25	–
MARNN (500 units, $p_{dropout}$=0.4) (Ours)	1.236	4.03M
MARNN (800 units, $p_{dropout}$=0.6) (Ours)	1.202	9.8M



(a) Training (batch size=128)

(b) Inference (batch size=1)

Figure 2: The running speed and memory consumption at the training and inference stages ($n_{mem} = 20$). The **red** area shows the extra GPU memory consumption by the MARNN. During training, the MARNN has to backpropagate gradients on the entire memory matrix, and producing new memory matrices at every time-step, therefore it consumes extra GPU memory. Nonetheless, 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 method, the memory consumption is much lower than in training stage. Empirically speaking, at training stage, the increasing memory consumption mainly comes from the increasing n_{mem}, while at inference stage it mainly comes from more computations in the MARNN cell.

with near-optimal implementations using Pytorch, an efficient deep-learning framework. We use the aforementioned experiment setup. We run the experiment using single-precision floating point calculations under a Titan XP GPU that has 12GB memory space. The results are depicted in figure 2. Under the same hidden size, the time and memory consumption of the MARNN are only $2 \sim 3$ times higher than LSTM, and at the inference stage it's less than 2 times higher, showing promising efficiency.

4.1.2 Ablation Study on Memory and Addressing Mechanism

In this subsection, we prove the efficacy of our external memory and addressing mechanism by ablation experiments. We use the aforementioned experiment setup. To validate the efficacy of our external memory, we conduct the experiments as follows and the results are depicted in table 2:

[1] As implemented in Ha et al. (2016).

[2] Our implementation. We don't see performance growth when we further increase the hidden size.

• **Short TBPTT Length** We shorten the truncated length in TBPTT from 150 to 50. The gradients are truncated so we can't do backpropagation on hidden states or memory from last iteration, but the MARNN still has direct access to historical hidden states. By comparing the performance of the LSTM and MARNN under such circumstances, we can know the efficacy of the external memory. The results show that the LSTM has a dramatic performance drop of 1.390 BPC while the MARNN only has a minor performance drop of 1.220 BPC. The results confirm that the direct access to memory is crucial when the networks can't access historical information by backpropagating gradients. And notably, our experiment shows that the MARNN is very robust to short-length TBPTT, $i.e.$ small iteration length, which means lower memory consumption. It enables using very large batch size to speed up the training and further increasing performance on sequence modelling tasks.

• **No Control Gates** We remove the control gate $\boldsymbol{g}_t^h$ and $\boldsymbol{g}_t^r$ in MARNN cell as depicted in equation 8, and directly use $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ to compute the gates in equation 11. The result shows a BPC of 1.354 that has a dramatic performance drop compared the 1.202 BPC of normal MARNN. From this, we deduce that the control gates successfully filter out irrelevant information. Without the control gates, the historical state $\boldsymbol{r}_t$ would have bring more noise than useful information.

To validate the efficacy of the addressing mechanism of the MARNN, we do the following modifications to the read/write operations of the MARNN, and the results are also depicted in table 2:

• **Random Read** The MARNN randomly chooses a historical state while reading. The result shows a BPC of 1.224, compared to the 1.202 BPC of normal MARNN. It proves that the read regime we proposed in section 3.2 is effective and can choose more useful historical state than random read.

• **Queue-style Write** While writing, the MARNN follows a fashion of first-in-first-out, $i.e.$ always chooses the oldest memory slot to write. It makes the memory $\boldsymbol{M}$ stubbornly store the hidden states of the last 20 time-steps. The result shows a BPC of 1.352, which proves our normal read/write mechanism has much more flexibility and can learn long-term patterns.

• **Independent Read/write** While writing, the MARNN doesn't write to the last read memory slot, but independently choose a memory slot to overwrite according to $\boldsymbol{x}_t$ and $\boldsymbol{h}_t$, in a similar fashion with the read operation we presented in section 3.2. The result shows a BPC of 1.245, proving that the normal writing regime in the MARNN can better learn long-term patterns and maximally preserve historical information, as we explain in section 3.3.

• **Smooth Read/write** We replace the gumbel-softmax function with the vanilla $\mathrm{softmax}$ function, so the MARNN reads and writes in a smooth way. The results show a minor performance drop of 1.212 BPC. The result shows that our discrete read/write regime is still better. Moreover, the smooth read/write regime would consume much more memory at both training and inference stages.

4.2 Temporal action detection and proposals on THUMOS' 14

In this subsection, we evaluate the MARNN on a more complicated real-world task concerning video analysis, $i.e.$ the temporal action detection and proposals, which consists of taking an input video and producing a set of temporal intervals that are likely to contain human actions (Buch et al., 2017). We use the Single-Stream Temporal Action Proposal(SST) (Buch et al., 2017) as our framework, and evaluate the model on the THUMOS' 14 dataset (Jiang et al., 2014), which contains 20+ hours of video with 200 train/validation and 213 test untrimmed video sequences. Specifically, the SST consists of a GRU and a fully connected output layer. It takes in sequential inputs of a video and outputs the existing probability predictions of action proposals with different time lengths that end at each time-step, and uses a multi-label loss to optimize its parameters. The sequential inputs are 4096-dimensional features extracted from every 16 consecutive video frames using C3D network (Tran et al., 2015), followed by PCA to downsize the input feature dimensionality.

In our experiment, we replace the GRU encoder in SST with a single MARNN cell, and we call the modified network MA-SST. For fairness, we also implement a modified SST with LSTM encoder and compare them with the original SST in Buch et al. (2017). Our experiment setup is basically the same as Buch et al. (2017), except that we use a simple fully connected layer to downsize the input features instead of using PCA. The input size of the two networks are both set to 256. The MA-SST have 256 hidden units, while the SST with LSTM have 512 hidden units. This ensures the 2 networks have roughly the same parameter count. The dropout and zoneout probabilities are both set to 0.2 and 0.3. n_{mem} of the MA-SST is set to 35. We train the networks for 300 epochs with

Table 2: Ablation on Penn Treebank (d_h=800).

Abaltion Model	BPC
No Control Gates	1.354
LSTM (TBPTT length=50)	1.390
MARNN (TBPTT length=50)	1.220
Random Read	1.224
Queue-style Write	1.352
Independent Read/write	1.245
Smooth Read/write	1.212

Table 3: Comparison of proposal generation performance in terms of recall at 1000 proposals. We choose tIoU=0.6 and 0.8 to compare for consistency with Buch et al. (2017). Our MA-SST achieves highest recall at tIoU=0.8.

Model	tIoU=0.6	tIoU=0.8
SST (Paper)	**0.920**	0.672
SST (We implement)	0.897	0.732
MA-SST(Ours)	0.911	**0.759**

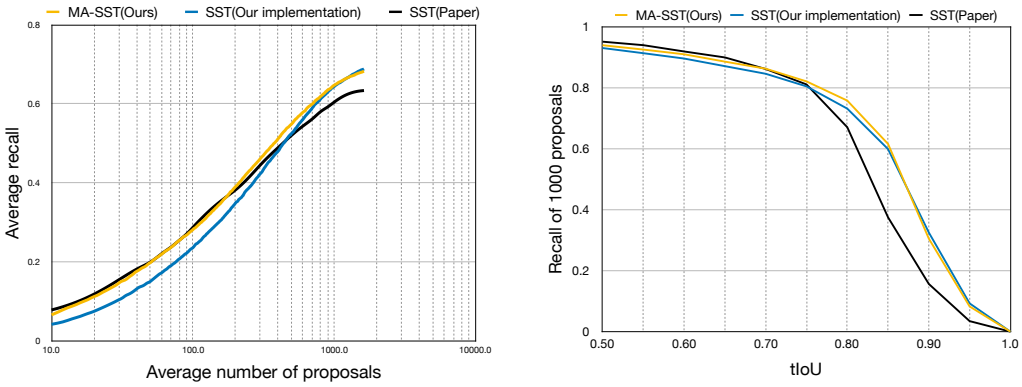


Figure 3: Comparison of proposal generation performance. For data generation, we use the code offered in Buch et al. (2017) for consistency. (Left) The average number of proposals v.s. average recall for tIoU ≥ 0.5. The MA-SST outperforms our SST when average number of proposals is less than 500, and outperforms the original SST when average number of proposals is more than 500. (Right) Different tIoU v.s. recall of 1000 average proposals. The MA-SST strongly outperforms the original SST and outperform our SST by a maximal margin of 2.7%.

initial learning rate of 0.002, and decay with a factor of 10 at the last 60 epochs. The average recall under different proposal numbers and tIoUs[3] are depicted in figure 3. MA-SST outperforms the original SST and our SST simultaneously. To the best of our knowledge, MA-SST achieves highest performance of recall at around tIoU=0.8 (see table 3), even if other competitive non-RNN networks (Escorcia et al., 2016; Caba Heilbron et al., 2016; Gao et al., 2017; Guo et al., 2018) are taken into account. The MA-SST tends to precisely generate proposals that has a high overlapping area with the ground truth action intervals, leading to high recall performance at high tIoUs. We believe the MA-SST has better performance than the SST mainly because of the direct access to historical states from which it can better locate the starting frame of an action. Moreover, the MA-SST can better learn long-term dependencies among important frames using skip connections.

5 Conclusion

In this paper, we have introduced the MARNN, a light-weight and memory-augmented RNN architecture with a novel MARNN cell. The MARNN incorporates an efficient external memory and allows direct backpropagation from output to memory. Our experiments shows that a single MARNN cell learns better representations than many hierarchical RNNs. Our ablation study suggests the efficacy of our external memory and addressing mechanism, and notably, our network is robust to short-length TBPTT which enables using very large batch size to theoretically speed up the training and further increase performance on sequence modelling tasks. Further research may lead to the efficient hierarchical and multi-scale structures of the MARNN and its successful applications in Seq2Seq models and complicated reasoning tasks.

[3]tIoU denotes temporal Intersection over Union between a proposal and its maximally overlapped ground truth action.

REFERENCES

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

Shyamal Buch, Victor Escorcia, Chuanqi Shen, Bernard Ghanem, and Juan Carlos Niebles. Sst: Single-stream temporal action proposals. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6373–6382. IEEE, 2017.

Fabian Caba Heilbron, Juan Carlos Niebles, and Bernard Ghanem. Fast temporal activity proposals for efficient detection of human actions in untrimmed videos. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1914–1923, 2016.

Victor Campos Camunez, Brendan Jou, Xavier Giró Nieto, Jordi Torres Viñals, and Shih-Fu Chang. Skip rnn: learning to skip state updates in recurrent neural networks. In *Sixth International Conference on Learning Representations: Monday April 30-Thursday May 03, 2018, Vancouver Convention Center, Vancouver:[proceedings]*, pp. 1–17, 2018.

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.

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

Tim Cooijmans, Nicolas Ballas, César Laurent, Çağlar Gülçehre, and Aaron Courville. Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*, 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 International Conference on Machine Learning-Volume 48*, pp. 1986–1994. JMLR. org, 2016.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pp. 334–343, 2015.

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

Victor Escorcia, Fabian Caba Heilbron, Juan Carlos Niebles, and Bernard Ghanem. Daps: Deep action proposals for action understanding. In *European Conference on Computer Vision*, pp. 768–784. Springer, 2016.

Jiyang Gao, Zhenheng Yang, Chen Sun, Kan Chen, and Ram Nevatia. Turn tap: Temporal unit regression network for temporal action proposals. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, pp. 3648–3656. IEEE, 2017.

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

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

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.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pp. 1828–1836, 2015.

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

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

Dashan Guo, Wei Li, and Xiangzhong Fang. Fully convolutional network for multiscale temporal action proposals. *IEEE Transactions on Multimedia*, 2018.

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

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

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

YG Jiang, Jingen Liu, A Roshan Zamir, G Toderici, I Laptev, Mubarak Shah, and Rahul Sukthankar. Thumos challenge: Action recognition with a large number of classes, 2014.

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

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

David Krueger, Tegan Maharaj, Janos Kramar, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Aaron Courville, and Christopher Pal. Zoneout: Regularizing rnns by randomly preserving hidden activations. 2016.

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

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 Computer Vision and Pattern Recognition*, pp. 5457–5466, 2018.

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.

Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*, 8, 2012.

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

Tony A Plate. Holographic reduced representation: Distributed representation for cognitive structures. 2003.

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 Neural Information Processing Systems*, pp. 3621–3629, 2016.

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

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. One-shot learning with memory-augmented neural networks. *arXiv preprint arXiv:1605.06065*, 2016.

Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. Recurrent dropout without memory loss. *arXiv preprint arXiv:1603.05118*, 2016.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

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

Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pp. 4489–4497, 2015.

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