Towards Non-Saturating Recurrent Units for Modelling Long-Term Dependencies

Sarath Chandar,[1,2*†] **Chinnadhurai Sankar,**[1*] **Eugene Vorontsov,**[1‡]
Samira Ebrahimi Kahou,[3] **Yoshua Bengio**[§1]
[1]Mila, Université de Montréal
[2]Google Brain
[3]Microsoft Research

Abstract

Modelling long-term dependencies is a challenge for recurrent neural networks. This is primarily due to the fact that gradients vanish during training, as the sequence length increases. Gradients can be attenuated by transition operators and are attenuated or dropped by activation functions. Canonical architectures like LSTM alleviate this issue by skipping information through a memory mechanism. We propose a new recurrent architecture (Non-saturating Recurrent Unit; NRU) that relies on a memory mechanism but forgoes both saturating activation functions and saturating gates, in order to further alleviate vanishing gradients. In a series of synthetic and real world tasks, we demonstrate that the proposed model is the only model that performs among the top 2 models across all tasks with and without long-term dependencies, when compared against a range of other architectures.

Introduction

Vanishing and exploding gradients remain a core challenge in the training of recurrent neural networks. While exploding gradients can be alleviated by gradient clipping or norm penalties (Pascanu, Mikolov, and Bengio, 2013), vanishing gradients are difficult to tackle. Vanishing gradients prevent the network from learning long-term dependencies Hochreiter (1991); Bengio, Simard, and Frasconi (1994) and arise when the network dynamics have attractors necessary to store bits of information over the long term (Bengio, Simard, and Frasconi, 1994). At each step of backpropagation, gradients in a recurrent neural network can grow or shrink across linear transition operators and shrink across nonlinear activation functions.

Successful architectures, like the LSTM Hochreiter and Schmidhuber (1997) and GRU Cho et al. (2014), alleviate vanishing gradients by allowing information to skip transition operators (and their activation functions) via an additive path that serves as memory. Both gates and transition operators use bounded activation functions (sigmoid, tanh). These help keep representations stable but attenuate gradients in two ways: they are contractive and they saturate at the bounds. Activation functions whose gradients contract as they saturate are typically referred to as saturating nonlinearities; in that sense, rectified linear units (ReLU) Nair and Hinton (2010) are non-saturating and thus help reduce vanishing gradients in deep feed-forward networks Krizhevsky, Sutskever, and Hinton (2012); Xu et al. (2015).

Saturating activation functions still dominate in the literature for recurrent neural networks, in part because of the success of LSTM on sequential problems (Sutskever, Vinyals, and Le, 2014; Vinyals and Le, 2015; Merity, Keskar, and Socher, 2017; Mnih et al., 2016). As these functions saturate, gradients contract. In addition, while saturated gates may push more information through memory, improving gradient flow across a long sequence, the gradient to the gating units themselves vanishes. We expect that non-saturating activation functions and gates could improve the modelling of long-term dependencies by avoiding these issues. We propose a non-saturating recurrent unit (NRU) that forgoes both saturating activation functions and saturating gates. We present a new architecture with the rectified linear unit (ReLU) and demonstrate that it is well equipped to process data with long distance dependencies, without sacrificing performance on other tasks.

Background

Vanishing Gradient Problem in RNNs

Recurrent Neural Networks are the most successful class of architectures for solving sequential problems. A vanilla RNN, at any time step t, takes $\mathbf{x}_t$ as input and updates its hidden state as follows:

$$\mathbf{z}_t = \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t \quad (1)$$
$$\mathbf{h}_t = f(\mathbf{z}_t), \quad (2)$$
$$(3)$$

where $\mathbf{W}$ and $\mathbf{U}$ are the recurrent and the input weights of the RNN respectively and $f(.)$ is any non-linear activation function like a sigmoid or tanh. Consider a sequence of length T with loss $\mathcal{L}$ computed at the end of the sequence. To compute $\frac{\partial \mathcal{L}}{\partial W_{ij}}$ at time step t, we should first compute $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}$ using the

*Equal Contribution
†Corresponding author: apsarathchandar@gmail.com.
‡Work done while at Microsoft Research.
§CIFAR Fellow.

Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

following chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_T} \frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_t} \tag{4}$$

$$= \frac{\partial \mathcal{L}}{\partial \mathbf{h}_T} \prod_{k=t}^{T-1} \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k} \tag{5}$$

$$= \frac{\partial \mathcal{L}}{\partial \mathbf{h}_T} \prod_{k=t}^{T-1} \left(\mathrm{diag}[f'(\mathbf{z}_k)]\mathbf{W}\right). \tag{6}$$

For a very long sequence length T, repeated multiplication of $\mathbf{W}$ in Equation 6 can cause exponential growth or decay of backpropagated gradients. Exploding gradients are caused by eigenvalues greater than one in $\mathbf{W}$. Conversely, eigenvalues less than one lead to vanishing gradients. The second cause of vanishing gradients is the activation function f. If the activation function is a saturating function like a sigmoid or tanh, then its Jacobian $\mathrm{diag}[f'(\mathbf{z}_k)]$ has eigenvalues less than or equal to one, causing the gradient signal to decay during backpropagation. Long term information may decay when the spectral norm of $\mathrm{diag}[f'(\mathbf{z}_k)]$ is less than 1 but this condition is actually necessary in order to store this information reliably (Bengio, Simard, and Frasconi, 1994; Pascanu, Mikolov, and Bengio, 2013). While gradient clipping can limit exploding gradients, vanishing gradients are harder to prevent and so limit the network's ability to learn long term dependencies (Hochreiter, 1991; Bengio, Simard, and Frasconi, 1994).

Existing Solutions

A key development in addressing the vanishing gradient issue in RNNs was the introduction of gated memory in the Long Short Term Memory (LSTM) network Hochreiter and Schmidhuber (1997). The LSTM maintains a cell state that is updated by addition rather than multiplication and serves as an "error carousel" that allows gradients to skip over many transition operations.

There have been several improvements proposed in the literature to make LSTM learn longer term dependencies. Gers, Schmidhuber, and Cummins (2000) proposed to initialize the bias of the forget gate to 1 which helped in modelling medium term dependencies when compared with zero initialization. This was also recently highlighted in the extensive exploration study by Jozefowicz, Zaremba, and Sutskever (2015). Tallec and Ollivier (2018) proposed chrono-initialization to initialize the bias of the forget gate ($\mathbf{b}_f$) and input gate ($\mathbf{b}_i$). Chrono-initialization requires the knowledge of maximum dependency length (T_{max}) and it initializes the gates as follows:

$$\mathbf{b}_f \sim \log(\mathcal{U}[1, T_{\text{max}} - 1]) \tag{7}$$

$$\mathbf{b}_i = -\mathbf{b}_f. \tag{8}$$

This initialization method encourages the network to remember information for approximately T_{max} time steps. While these forget gate bias initialization techniques encourage the model to retain information longer, the model is free to unlearn this behaviour.

The Gated Recurrent Unit (GRU) (Cho et al., 2014) is a simplified version of the LSTM (with fewer gates) which works equally well Chung et al. (2014). Recently, van der Westhuizen and Lasenby (2018) proposed a forget-gate only version of LSTM called JANET. JANET outperforms the LSTM on some long-term dependency tasks. We suspect that this may be because JANET omits a few saturating gates in the LSTM which are not crucial for the performance of the model. In that sense, JANET is similar to the proposed NRU. However, the NRU uses non-saturating gates and hence should have better gradient flow than JANET.

The non-saturating ReLU activation function should reduce the incidence of vanishing gradients in a vanilla RNN but can make training unstable through unbounded growth of the hidden state. Better initialization of the recurrent weight matrix has been explored in the literature for such RNNs. Le, Jaitly, and Hinton (2015) proposed initializing the recurrent weight matrix using the identity matrix. Orthogonal initialization of recurrent weight matrices has been studied by Henaff, Szlam, and LeCun (2016). These initialization techniques, like bias initialization in the LSTM, encourage better gradient flow at the beginning of the training. Vorontsov et al. (2017) proposed various ways to parameterize the recurrent weight matrix so that it could stay orthogonal or nearly orthogonal. They observed that allowing slight deviation from orthogonality helps improve model performance and convergence speed.

The idea of using unitary matrices for recurrent weight matrices has been explored in Arjovsky, Shah, and Bengio (2016); Wisdom et al. (2016); Jing et al. (2017b). While Arjovsky, Shah, and Bengio (2016); Jing et al. (2017b) parameterize the matrix to be unitary, Wisdom et al. (2016) reprojects the matrix after every gradient update. Orthogonal and unitary matrices do not filter out information, preserving gradient norm but making forgetting depend on a separate mechanism such as forget gates. This has been explored with saturating gates in Jing et al. (2017a); Dangovski, Jing, and Soljacic (2017).

A non-gated Statistical Recurrent Unit (SRU) was proposed in Oliva, Póczos, and Schneider (2017). SRU uses only a ReLU activation function and recurrent multi-scale summary statistics. The summary statistics are calculated using exponential moving averages which might shrink the gradients. We compare NRU with SRU in our experiments.

Other related lines of work which aim to model long-term dependencies in RNNs are Memory Augmented Neural Networks (MANNs) Graves, Wayne, and Danihelka (2014); Graves et al. (2016); Gülçehre et al. (2018); Gülçehre, Chandar, and Bengio (2017). Graves, Wayne, and Danihelka (2014); Graves et al. (2016) have continuous addressing schemes for the external memory, which also suffer from the vanishing gradient problem. Gülçehre et al. (2018); Gülçehre, Chandar, and Bengio (2017) attempts to address this issue by moving to a discrete addressing mechanism so that gradients could flow better. However, they use approximate gradient methods like REINFORCE Williams (1992) or the Gumbel-softmax trick Maddison, Mnih, and Teh (2016); Jang, Gu, and Poole (2016) which are either difficult to train or to scale. All these architectures use saturating activation functions and saturating gates for memory control.

Data flow normalization techniques like layer normaliza-

tion Ba, Kiros, and Hinton (2016) and recurrent batch normalization Cooijmans et al. (2016) were also proposed in the literature to improve the gradient flow in recurrent architectures. Gülçehre et al. (2016) proposed to inject noise in the saturating gates so that the gradients may flow easily.

Non-saturating Recurrent Unit (NRU) Network

Saturating gating functions introduce a trade-off where distant gradient propagation may occur at the cost of vanishing updates to the gating mechanism. We seek to avoid this trade-off between distant gradient propagation but not updateable gates (saturated gates) and short gradient propagation but updateable gates (non-saturated gates). To that end, we propose the Non-saturating Recurrent Unit — a gated recurrent unit with no saturating activation functions.

Model

At any time step t, NRU takes some input $\mathbf{x}_t$ and updates its hidden state as follows:

$$\mathbf{h}_t = f(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_i \mathbf{x}_t + \mathbf{W}_c \mathbf{m}_{t-1}) \tag{9}$$

where $\mathbf{h}_{t-1} \in \mathbb{R}^h$ is the previous hidden state, $\mathbf{m}_{t-1} \in \mathbb{R}^m$ is the previous memory cell, and f is a ReLU non-linearity. The memory cell in NRU is a flat vector similar to the cell vector in LSTM. However, in NRU, the hidden state and the memory cell need not be of the same size. We allow the memory cell to be larger than the hidden state to hold more information.

At every time step t, memory cell is updated as as follows:

$$\mathbf{m}_t = \mathbf{m}_{t-1} + \sum_{k_1} \alpha_i \mathbf{v}_i^w - \sum_{k_2} \beta_i \mathbf{v}_i^e \tag{10}$$

where

- $\mathbf{m}_t \in \mathbb{R}^m$ is the memory vector at time t.
- k_1 is the number of writing heads and k_2 is the number of erasing heads. We often set $k_1 = k_2 = k$.
- $\mathbf{v}_i^w \in \mathbb{R}^m$ is a normalized vector which defines where to write in the memory. Similarly $\mathbf{v}_i^e \in \mathbb{R}^m$ is a normalized vector which defines where to erase in the memory.
- α_i is a scalar value which represents the content which is written in the memory along the direction of $\mathbf{v}_i^w$. Similarly, β_i is a scalar value which represents the content which is removed from the memory along the direction of $\mathbf{v}_i^e$.

Intuitively, the network first computes a low-dimensional projection of the current hidden state by generating a k-dimensional α vector. Then it writes each unit of this k-dimensional vector along k different basis directions ($\mathbf{v}_i^w$s). These k basis directions specify *where to write* as continuous-valued vectors and hence there is no need for saturating non-linearities or discrete addressing schemes. The network also similarly erases some content from the memory (by using β and $\mathbf{v}_i^e$s).

Scalars α_i and β_i are computed as follows:

$$\alpha_i = f_\alpha(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \tag{11}$$

$$\beta_i = f_\beta(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \tag{12}$$

where f_α and f_β are linear functions followed by an optional ReLU non-linearity. Vectors $\mathbf{v}_i^w$ and $\mathbf{v}_i^e$ are computed as follows:

$$\mathbf{v}_i^w = f_w(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \tag{13}$$

$$\mathbf{v}_i^e = f_e(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \tag{14}$$

where f_w and f_e are linear functions followed by an optional ReLU non-linearity followed by an explicit normalization. We used L_5 normalization in all of our experiments.

Each $\mathbf{v}$ vector is m-dimensional and there are $2k$ such vectors that needs to be generated. This requires a lot of parameters which could create problems both in terms of computational cost and in terms of overfitting. We reduce this large number of parameters by using the following outer product trick to factorize the $\mathbf{v}$ vectors:

$$\mathbf{p}_i^w = f_{w_1}(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \in \mathbb{R}^{\sqrt{m}} \tag{15}$$

$$\mathbf{q}_i^w = f_{w_2}(\mathbf{x}_t, \mathbf{h}_t, \mathbf{m}_{t-1}) \in \mathbb{R}^{\sqrt{m}} \tag{16}$$

$$\mathbf{v}_i^w = g(\text{vec}(\mathbf{p}_i^w \mathbf{q}_i^{w^T})) \tag{17}$$

where f_{w_1} and f_{w_2} are linear functions, vec() vectorizes a given matrix, g is an optional ReLU non-linearity followed by an explicit normalization. Thus, instead of producing an m-dimensional vector for the direction, we only need to produce a $2\sqrt{m}$-dimensional vector which requires substantially fewer parameters and scales more reasonably. We can further reduce the number of parameters by generating $2\sqrt{km}$-dimensional vector instead of $2k\sqrt{m}$-dimensional vector and then use the outer product trick to generate the required km-dimensional vector. We do this trick in our implementation of NRU.

If there is no final ReLU activation function while computing α, β, and $\mathbf{v}$, then there is no distinction between write and erase heads. For example, either β_i or $\mathbf{v}_i^e$ could become negative, changing the erasing operation into one that adds to the memory. Having an explicit ReLU activation in all these terms forces the architecture to use writing heads to add information to the memory and erasing heads to erase information from the memory. However, we treat this enforcement as optional and in some experiments, we let the network decide how to use the heads by removing these ReLU activations.

Discussion

In vanilla RNNs, the hidden state acts as the memory of the network. So there is always a contradiction between stability (which requires small enough $\mathbf{W}_h$ to avoid explosions) and long-term storage (which requires high enough $\mathbf{W}_h$ to avoid vanishing gradients). This contradiction is exposed in a ReLU RNN which avoids the saturation from gates at the cost of network stability. LSTM and GRU reduce this problem by introducing memory in the form of information skipped forward across transition operators, with gates determining which information is skipped. However, the gradient on the gating units themselves vanishes when the unit activations saturate. Since distant gradient propagation across memory is aided by gates that are locked to ON (or OFF), this introduces an unfortunate trade-off where either the gate

mechanism receives updates or gradients are skipped across many transition operators.

NRU also maintains a separate memory vector like LSTM and GRU. Unlike LSTM and GRU, NRU does not have saturating gates. Even if the paths purely through hidden states h vanish (with small enough $\mathbf{W}_h$), the actual long term memories are stored in memory vector $\mathbf{m}$ which has additive updates and hence no vanishing gradient.

Although the explicit normalization in vectors $\mathbf{v}_i^w$ and $\mathbf{v}_i^e$ may cause gradient saturation for large values, we focus on the nonlinearities because of their outsized effect on vanishing gradients. In comparison, the effect of normalization is less pronounced than that of explicitly saturating nonlinearities like tanh or sigmoid which are also contractive based on our experiments. Also, normalization has been employed to avoid the saturation regimes of these nonlinearities (Ioffe and Szegedy, 2015; Cooijmans et al., 2016). Furthermore, we observe that forgoing these nonlinearities allows the NRU to converge much faster than other models.

Experiments

In this section, we compare the performance of NRU networks with several other RNN architectures in three synthetic tasks (5 including variants) and two real tasks. Specifically, we consider the following extensive list of recurrent architectures: RNN with orthogonal initialization (RNN-orth), RNN with identity initialization (RNN-id), LSTM, LSTM with chrono initialization (LSTM-chrono), GRU, JANET, SRU, EURNN Jing et al. (2017b), GORU Jing et al. (2017a). We used the FFT-like algorithm for the orthognal transition operators in EURNN and GORU as it tends to be much faster than the tunable algorithm, both proposed by Jing et al. (2017b). While these two algorithms have a low theoretical computational complexity, they are difficult to parallelize.

We mention common design choices for all the experiments here: RNN-orth and RNN-id were highly unstable in all our experiments and we found that adding layer normalization helps. Hence all our RNN-orth and RNN-id experiments use layer normalization. We did not see any significant benefit in using layer normalization for other architectures and hence it is turned off by default for other architectures. NRU with linear writing and erasing heads performed better than with ReLU based heads in all our experiments except the language model experiment. Hence we used linear writing and erasing heads unless otherwise mentioned. We used the Adam optimizer Kingma and Ba (2014) with a default learning rate of 0.001 in all our experiments. We clipped the gradients by norm value of 1 for all models except GORU and EURNN since their transition operators do not expand norm. We used a batch size of 10 for most tasks, unless otherwise stated.

Table-1 summarizes the results of this section. Out of 10 different architectures that we considered, NRU is the only model that performs among the top 2 models across all 7 tasks. The code for NRU Cell is available at https://github.com/apsarath/NRU.

Copying Memory Task

The copying memory task was introduced in Hochreiter and Schmidhuber (1997) as a synthetic task to test the network's ability to remember information over many time steps. The task is defined as follows. Consider n different symbols. In the first k time steps, the network receives an initial sequence of k random symbols from the set of n symbols sampled with replacement. Then the network receives a "blank" symbol for $T-1$ steps followed by a "start recall" marker. The network should learn to predict a "blank" symbol, followed by the initial sequence after the marker. Following Arjovsky, Shah, and Bengio (2016), we set $n = 8$ and $k = 10$. The copying task is a pathologically difficult long-term dependency task where the output at the end of the sequence depends on the beginning of the sequence. We can vary the dependency length by varying the length of the sequence T. A memoryless model is expected to achieve a sub-optimal solution which predicts a constant sequence after the marker, for any input (referred to as the "baseline" performance). The cross entropy for such a model would be $\frac{k \log n}{T+2n}$ Arjovsky, Shah, and Bengio (2016).

Table 1: Number of tasks where the models are in top-1 and top-2. Maximum of 7 tasks. Note that there are ties between models for some tasks so the column for top-1 performance would not sum up to 7.

Model	in Top-1	in Top-2
RNN-orth	1	1
RNN-id	1	1
LSTM	1	1
LSTM-chrono	1	1
GRU	1	2
JANET	1	3
SRU	1	1
EURNN	2	3
GORU	0	1
NRU	**4**	**7**

We trained all models with approximately the same number of parameters ($\sim$23.5k), in an online fashion where every mini-batch is dynamically generated. We consider $T = 100$ and $T = 200$. In Figure 1 we plot the cross entropy loss for all the models. Unsurprisingly, EURNN solves the task in a few hundred updates as it is close to an optimal architecture tuned for this task Henaff, Szlam, and LeCun (2016). Conversely, RNN-orth and RNN-id get stuck at baseline performance. NRU converges faster than all other models, followed by JANET which requires two to three times more updates to solve the task.

We observed the change in the memory vector across the time steps (in the appendix[1]). We can see that the network has learnt to add information into the memory in the beginning of the sequence and then it does not access the memory until it sees the marker. Then it makes changes in the memory in the last 10 time steps to copy the sequence.

We performed additional experiments following the observation by Henaff, Szlam, and LeCun (2016) that gated models like the LSTM outperform a simple non-gated orthogonal network (similar to the EURNN) when the time lag T is varied in the copying task. This variable length task highlights

[1]For appendix, refer https://arxiv.org/abs/1902.06704

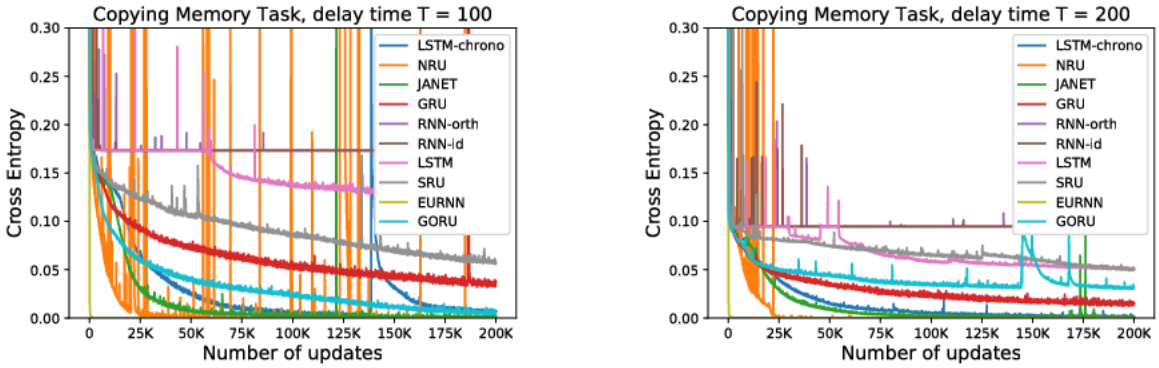

Figure 1: Copying memory task for T = 100 (in left) and T = 200 (in right). Cross-entropy for random baseline : 0.17 and 0.09 for T=100 and T=200 respectively.

the non-generality of the solution learned by models like EURNN. Only models with a dynamic gating mechanism can solve this problem. In Figure-2, we plot the cross-entropy loss for all the models. The proposed NRU model is the fastest to solve this task, followed by JANET, while the EURNN performs poorly, as expected according to Henaff, Szlam, and LeCun (2016). These two experiments highlight the fact that NRU can store information longer than other recurrent architectures. Unlike EURNN which behaves like a fixed clock mechanism, NRU learns a gating function to lock the information in the memory as long as needed. This is similar to the behaviour of other gated architectures. However, NRU beats other gated architectures mainly due to better gradient flow which results in faster learning. Figure-3 shows that NRU converges significantly faster than its strongest competitors (JANET and LSTM-chrono) in all the 4 tasks.

Denoising Task

The denoising task was introduced in Jing et al. (2017a) to test both the memorization and forgetting ability of RNN architectures. This is similar to the copying memory task. However, the data points are located randomly in a long noisy sequence. Consider again an alphabet of n different symbols from which k random symbols are sampled with replacement. These symbols are then separated by random lengths of strings composed of a "noise" symbol, in a sequence of total length T. The network is tasked with predicting the k symbols upon encountering a "start recall" marker, after the length T input sequence. Again we set $n = 8$ and $k = 10$. This task requires both an ability to learn long term dependencies and also an ability to filter out unwanted information (considered noise).

The experimental procedure is exactly the same as described for the copying memory task. In Figure 4 we plot the cross-entropy loss for all the models for $T = 100$. In this task, all the models converge to the solution except EURNN. While NRU learns faster in the beginning, all the algorithms converge after approximately the same number of updates.

Table 2: Bits Per Character (BPC) and Accuracy in test set for character level language modelling in PTB.

Model	BPC	Accuracy
LSTM	1.48	67.98
LSTM-chrono	1.51	67.41
GRU	**1.45**	**69.07**
JANET	1.48	68.50
GORU	1.53	67.60
EURNN	1.77	63.00
NRU	1.47	68.48

Character Level Language Modelling

Going beyond synthetic data, we consider character level language modelling with the Penn Treebank Corpus (PTB) Marcus, Santorini, and Marcinkiewicz (1993). In this task, the network is fed one character per time step and the goal is to predict the next character. Again we made sure that all the networks have approximately the same capacity ($\sim$2.15M parameters). We use a batch size of 128 and perform truncated backpropagation through time, every 150 time steps. We evaluate according to bits per character (BPC) and accuracy.

All models were trained for 20 epochs and evaluated on the test set after selecting for each the model state which yields the lowest BPC on the validation set. The test set BPC and accuracy are reported in Table-2. We did not add dropout or batch normalization to any model. From the table, we can see that GRU is the best performing model, followed closely by NRU. We note that language modelling does not require very long term dependencies. This is supported by the fact that changing the additive memory updates in NRU to multiplicative updates does not hurt performance (it actually improves it). This is further supported by our observation that setting T_{max} to 50 was better than setting T_{max} to 150 in chrono-initialization for LSTM and JANET. All the best performing NRU models used ReLU activations in the writing and erasing heads.

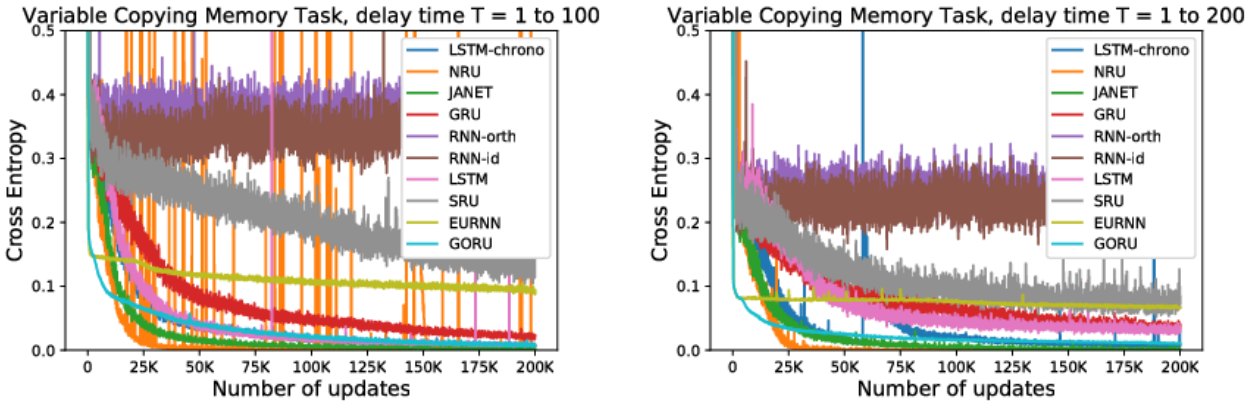

Figure 2: Variable Copying memory task for T = 100 (in left) and T = 200 (in right).

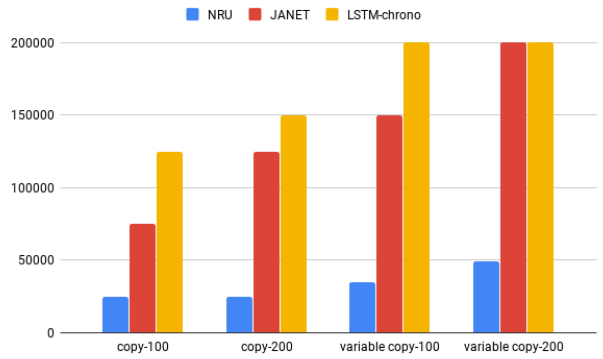

Figure 3: Comparison of top-3 models w.r.t the number of the steps to converge for different tasks. NRU converges significantly faster than JANET and LSTM-chrono.

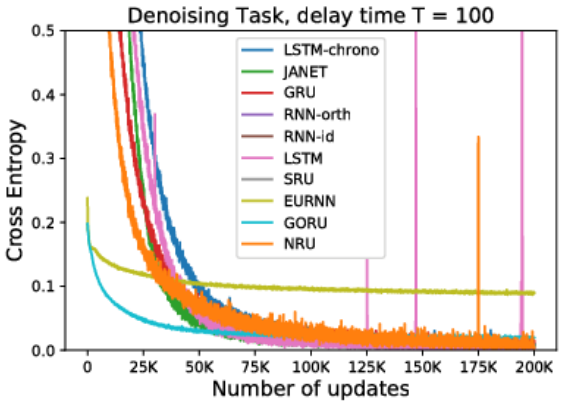

Figure 4: Denoising task for T = 100.

Permuted Sequential MNIST

The Permuted Sequential MNIST task was introduced in Le, Jaitly, and Hinton (2015) as a benchmark task to measure the performance of RNNs in modelling complex long term dependencies. In a sequential MNIST task, all the 784 pixels of an MNIST digit are fed to the network one pixel at a time and the network must classify the digit in the 785th time step. Permuted sequential MNIST (psMNIST) makes the problem harder by applying a fixed permutation to the pixel sequence, thus introducing longer term dependencies between pixels in a more complex order.

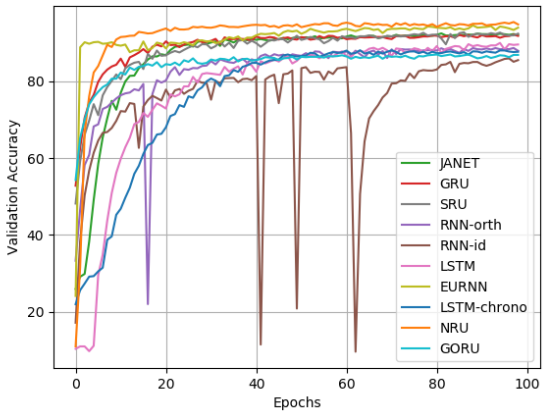

Figure 5: Validation curve for psMNIST task.

For this task, we used a batch size of 100. All the networks have approximately the same number of parameters ($\sim$165k). This corresponds to 200 to 400 hidden units for most of the architectures. Since the FFT-style algorithm used with EURNN requires few parameters, we used a large 1024 unit hidden state still achieving fewer parameters ($\sim$17k parameters) at maximum memory usage. We report validation and test accuracy in Table-3 and plot the validation curves in Figure-5. On this task, NRU performs better than all the other architectures, followed by the EURNN. The good performance of the

EURNN could be attributed to its large hidden state, since it is trivial to store all 784 input values in 1024 hidden units. This model excels at preserving information, so performance is bounded by the classification performance of the output layer. While NRU remains stable, RNN-orth and RNN-id are not stable as seen from the learning curve. Surprisingly, LSTM-chrono is not performing better than regular LSTM.

Table 3: Validation and test set accuracy for psMNIST task.

Model	valid	test
RNN-orth	88.70	89.26
RNN-id	85.98	86.13
LSTM	90.01	89.86
LSTM-chrono	88.10	88.43
GRU	92.16	92.39
JANET	92.50	91.94
SRU	92.79	92.49
EURNN	94.60	94.50
GORU	86.90	87.00
NRU	**95.46**	**95.38**

Model Analysis

Stability: While we use gradient clipping to limit exploding gradients in NRU, we observed gradient spikes in some of our experiments. We suspect that the network recovers due to consistent gradient flow. To verify that the additive nature of the memory does not cause memory to explode for longer time steps, we performed a sanity check experiment where we trained the model on the copy task (with T=100, 200, and 500) with random labels and observed that training did not diverge. We observed some instabilities with all the models when training with longer time steps (T = 2000). With NRU, we observed that the model converged faster and was more stable with a higher memory size. For instance, our model converged almost twice as early when we increased the memory size from 64 to 144.

Forgetting Ability: We performed another sanity check experiment to gauge the forgetting ability of our model. We trained it on the copy task but reset the memory only once every k= 2, 5, and 10 examples and observed that NRU learned to reset the memory in the beginning of every example.

Gradient Flow: To give an overview of the gradient flow across different time steps during training, we compared the total gradient norms in the copying memory task for the top 3 models: NRU, JANET and LSTM-Chrono (in the appendix). We see that the NRU's gradient norm is considerably higher during the initial stages of the training while the other model's gradient norms rise after about 25k steps. After 25k steps, the drop in the NRU gradient norm coincides with the model's convergence, as expected. We expect that this ease of gradient flow in NRU serves as an additional evidence that NRU can model long-term dependencies better than other architectures.

Conclusion

In this paper, we introduce Non-saturating Recurrent Units (NRUs) for modelling long term dependencies in sequential problems. The gating mechanism in NRU is additive (like in LSTM and GRU) and non-saturating (unlike in LSTM and GRU). This results in better gradient flow for longer durations. We present empirical evidence in support of non-saturating gates in the NRU with (1) improved performance on long term dependency tasks, (2) higher gradient norms, and (3) faster convergence when compared to baseline models. NRU was the best performing general purpose model in all of the long-term dependency tasks that we considered and is competitive with other gated architectures in short-term dependency tasks. This work opens the door to other potential non-saturating gated recurrent network architectures.

We would also like to apply NRU in several real world sequential problems in natural language processing and reinforcement learning where the LSTM is the default choice for recurrent computation.

References

Arjovsky, M.; Shah, A.; and Bengio, Y. 2016. Unitary evolution recurrent neural networks. In *Proceedings of ICML 2016*, 1120–1128.

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

Bengio, Y.; Simard, P.; and Frasconi, P. 1994. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on* 5(2):157–166.

Cho, K.; van Merrienboer, B.; Gülçehre, Ç.; Bahdanau, D.; Bougares, F.; Schwenk, H.; and Bengio, Y. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*.

Chung, J.; Gülçehre, Ç.; Cho, K.; and Bengio, Y. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR* abs/1412.3555.

Cooijmans, T.; Ballas, N.; Laurent, C.; and Courville, A. C. 2016. Recurrent batch normalization. *CoRR* abs/1603.09025.

Dangovski, R.; Jing, L.; and Soljacic, M. 2017. Rotational unit of memory. *CoRR* abs/1710.09537.

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

Graves, A.; Wayne, G.; Reynolds, M.; Harley, T.; Danihelka, I.; Grabska-Barwinska, A.; Colmenarejo, S. G.; Grefenstette, E.; Ramalho, T.; Agapiou, J.; Badia, A. P.; Hermann, K. M.; Zwols, Y.; Ostrovski, G.; Cain, A.; King, H.; Summerfield, C.; Blunsom, P.; Kavukcuoglu, K.; and Hassabis, D. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538(7626):471–476.

Graves, A.; Wayne, G.; and Danihelka, I. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Gülçehre, Ç.; Moczulski, M.; Denil, M.; and Bengio, Y. 2016. Noisy activation functions. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016*, 3059–3068.

Gülçehre, Ç.; Chandar, S.; Cho, K.; and Bengio, Y. 2018. Dynamic neural turing machine with continuous and discrete addressing schemes. *Neural Computation* 30(4).

Gülçehre, Ç.; Chandar, S.; and Bengio, Y. 2017. Memory augmented neural networks with wormhole connections. *CoRR* abs/1701.08718.

Henaff, M.; Szlam, A.; and LeCun, Y. 2016. Recurrent orthogonal networks and long-memory tasks. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016*.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural Computation* 9(8):1735–1780.

Hochreiter, S. 1991. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München* 91.

Ioffe, S., and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR* abs/1502.03167.

Jang, E.; Gu, S.; and Poole, B. 2016. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*.

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

Jing, L.; Shen, Y.; Dubcek, T.; Peurifoy, J.; Skirlo, S. A.; LeCun, Y.; Tegmark, M.; and Soljacic, M. 2017b. Tunable efficient unitary neural networks (EUNN) and their application to rnns. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*.

Jozefowicz, R.; Zaremba, W.; and Sutskever, I. 2015. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning*.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *CoRR* abs/1412.6980.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012*.

Le, Q. V.; Jaitly, N.; and Hinton, G. E. 2015. A simple way to initialize recurrent networks of rectified linear units. *CoRR* abs/1504.00941.

Maddison, C. J.; Mnih, A.; and Teh, Y. W. 2016. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*.

Marcus, M. P.; Santorini, B.; and Marcinkiewicz, M. A. 1993. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics* 19(2):313–330.

Merity, S.; Keskar, N. S.; and Socher, R. 2017. Regularizing and optimizing LSTM language models. *CoRR* abs/1708.02182.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T. P.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016*, 1928–1937.

Nair, V., and Hinton, G. E. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of ICML 2010)*, 807–814.

Oliva, J. B.; Póczos, B.; and Schneider, J. G. 2017. The statistical recurrent unit. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, 2671–2680.

Pascanu, R.; Mikolov, T.; and Bengio, Y. 2013. On the difficulty of training recurrent neural networks. *ICML (3)* 28:1310–1318.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014.*, 3104–3112.

Tallec, C., and Ollivier, Y. 2018. Can recurrent neural networks warp time? *CoRR* abs/1804.11188.

van der Westhuizen, J., and Lasenby, J. 2018. The unreasonable effectiveness of the forget gate. *ArXiv e-prints*.

Vinyals, O., and Le, Q. V. 2015. A neural conversational model. *CoRR* abs/1506.05869.

Vorontsov, E.; Trabelsi, C.; Kadoury, S.; and Pal, C. 2017. On orthogonality and learning recurrent networks with long term dependencies. In *Proceedings of the 34th International Conference on Machine Learning, 2017*.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8:229–256.

Wisdom, S.; Powers, T.; Hershey, J. R.; Roux, J. L.; and Atlas, L. E. 2016. Full-capacity unitary recurrent neural networks. In *Annual Conference on Neural Information Processing Systems 2016*, 4880–4888.

Xu, B.; Wang, N.; Chen, T.; and Li, M. 2015. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*.