Shuffling Recurrent Neural Networks

Michael Rotman and Lior Wolf
The School of Computer Science, Tel Aviv University
rotmanmi@post.tau.ac.il, wolf@cs.tau.ac.il

Abstract

We propose a novel recurrent neural network model, where the hidden state h_t is obtained by permuting the vector elements of the previous hidden state h_{t-1} and adding the output of a learned function $\beta(x_t)$ of the input x_t at time t. In our model, the prediction is given by a second learned function, which is applied to the hidden state $s(h_t)$. The method is easy to implement, extremely efficient, and does not suffer from vanishing nor exploding gradients. In an extensive set of experiments, the method shows competitive results, in comparison to the leading literature baselines. We share our implementation at https://github.com/rotmanmi/SRNN.

Introduction

Recurrent Neural Networks (RNN) architectures have been successful in solving sequential or time dependent problems. Such methods maintain a latent representation, commonly referred to as the "hidden state", and apply the same learned functions repeatedly to the input at each time step, as well as to the current hidden state.

A well-known challenge with RNNs is that of exploding or vanishing gradients. The various methods devised in order to solve this problem can be roughly divided into two groups. The first group utilizes a gating mechanism to stabilize the gradient flow between subsequent hidden states (Hochreiter and Schmidhuber 1997; Cho et al. 2014), whereas the second group focuses on preserving the norm of the hidden states by employing constraints on the family of matrices used as the network's parameters (Arjovsky, Shah, and Bengio 2016; Henaff, Szlam, and LeCun 2016).

An alternative view of the problem of exploding and vanishing gradients considers it as the symptom of a deeper issue and not as the root cause. Current RNN architectures perform a matrix multiplication operation, with learned weights, over previously seen hidden states during each time step. Therefore, inputs appearing in different times are processed using different powers of the weight matrices (with interleaving non-linearities): the first input of a sequence of length T is processed by the same learned sub-network T times, whereas the last input at time T is processed only once. This creates an inherent gap in the way that each time step influences the network weights during training.

In this work, we propose an alternative RNN framework. Instead of using a learned set of parameters to determine the transition between subsequent hidden states, our method uses a parameter-free shift mechanism, in order to distinguish between inputs fed at different times. This shift mechanism forms an orthogonal matrix operation, and is, therefore, not prone to the gradient explosion or to its vanishing. Furthermore, the various time steps are treated in a uniform manner, leading to an efficient and perhaps more balanced solution. This allows us, for example, to solve problems with much larger sequence lengths than reported in the literature.

Our experiments show that our Shuffling Recurrent Neural Network (SRNN) is indeed able to tackle long-term memorization tasks successfully, and shows competitive results, in comparison to the current state of the art of multiple tasks. SRNN is elegant, easy to implement, efficient, and insensitive to its hyperparameters.

Background and Related Work

RNNs have been the architecture of choice, when solving sequential tasks. The most basic structure, which we refer to as the vanilla RNN, updates at each time a hidden state vector h_t using an input x_t,

$$h_t = \sigma\left(W_1 h_{t-1} + W_2 x_t\right) \equiv \sigma(z_t)\,, \tag{1}$$

where σ is a non-linear activation function, such as $\tanh$ or ReLU. Given a sequence of length T, $\{x_t\}_{t=1}^T$, computing the gradients of a loss function, $\mathcal{L}$, w.r.t to W_1, $\frac{\partial \mathcal{L}}{\partial W_1}$, requires backpropagation throughout all the hidden states $\{h_t\}_{t=1}^T$,

$$\frac{\partial \mathcal{L}}{\partial W_1} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial h_t}\frac{\partial h_t}{\partial W_1} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial h_t}\sigma'(z_t)h_{t-1}\,, \tag{2}$$

where $\frac{\partial \mathcal{L}}{\partial h_t} = \sum_{t'=t+1}^{T} \frac{\partial \mathcal{L}}{\partial h_{t'}}\frac{\partial h_{t'}}{\partial h_t}$ and $\frac{\partial h_{t'}}{\partial h_t} = \prod_{i=t}^{t'} \frac{\partial h_{i+1}}{\partial h_i} = \prod_{i=t}^{t'} \sigma'(z_i)W_1$. Depending on the maximal eigenvalue of W_1, the repeated multiplication by W_1 in $\frac{\partial \mathcal{L}}{\partial h_t}$ may lead to exponential growth or decay in $\frac{\partial \mathcal{L}}{\partial W_1}$ when $T \gg 1$.

Many of the successful RNN methods utilize a gating mechanism, where the hidden state h_t can be either suppressed or scaled, depending on a function of the previous hidden state and the input. Among these solutions, there is the seminal Long-Short Term Memory network (LSTM)

(Hochreiter and Schmidhuber 1997) that utilizes a gating mechanism together with a memory cell, the powerful Gated Recurrent Unit (GRU) (Cho et al. 2014), and the recent Non-Saturating Recurrent Unit (NRU) (Chandar et al. 2019) that makes use of a non-saturating function, such as a ReLU, for the activation function. These units often make use of a gradient clipping scheme while training, since they contain no inherent mechanism to deal with exploding gradients over very long time sequences.

A second family of RNNs focuses on constraining the weight matrix W_1 of the RNN to be orthogonal or unitary. Unitary Recurrent Neural Networks (uRNN) (Arjovsky, Shah, and Bengio 2016) force a strict structure regime on the parameter matrices, thus modifying these matrices in a sub-manifold of unitary matrices. Noting that the method neglects some types of unitary matrices, the Full-Capacity Unitary Recurrent Neural Network (Wisdom et al. 2016) uses a weight parameterization that spans the complete set of unitary matrices, by constraining the gradient to reside on a Stiefel manifold. EUNN (Jing et al. 2017) employs a more efficient method in order to span this set. Another efficient approach to optimize in this space, which is based on Householder reflections, was proposed by Mhammedi et al. (2017), and a Cayley transform parameterization was used in (Helfrich, Willmott, and Ye 2018; Maduranga, Helfrich, and Ye 2019). The recent nnRNN (Kerg et al. 2019) method parameterizes the transformations between successive hidden states using both a normal matrix and a non-normal one, where the first is responsible for learning long-scale dynamics and the second adds to the expressibility of the model.

Method

The SRNN layer contains two hidden-state processing components, the learned network β that is comprised of fully connected layers, and a fixed permutation matrix W_p. At each time step, the layer, like other RNNs, receives two input signals: the hidden state of the previous time step, $h_{t-1} \in \mathbb{R}^{d_h}$, and the input at the current time step, $x_t \in \mathbb{R}^{d_i}$, where d_h and d_i are the dimensions of the hidden state and the input, respectively. The layer computes the following, (we redefine the notation, disregarding the definitions of Sec.):

$$h_t = \sigma\left(W_p h_{t-1} + \beta\left(x_t\right)\right) \equiv \sigma\left(z_t\right) , \tag{3}$$

where σ is the activation function (such as ReLU or $\tanh$), see Fig. 1(a). The permutation operator W_p reflects the time dependency, whereas β is agnostic to the ordering of the input. Without loss of generality, the operator can be chosen as the shift operation that can represented by the off-diagonal matrix

$$W_p = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 1 \\ 1 & 0 & \cdots & 0 & 0 \end{pmatrix} . \tag{4}$$

Network β employs a two-headed structure for a self-gating mechanism. The primary branch is composed of an MLP $f_r{:}\mathbb{R}^{d_i} \to \mathbb{R}^{d_h}$. The gating branch scales β's output by applying a single affine layer with a sigmoid activation,

$$\beta\left(x_t\right) = f_r\left(x_t\right) \odot \text{sigmoid}\left(W_g x_t + b_g\right) , \tag{5}$$

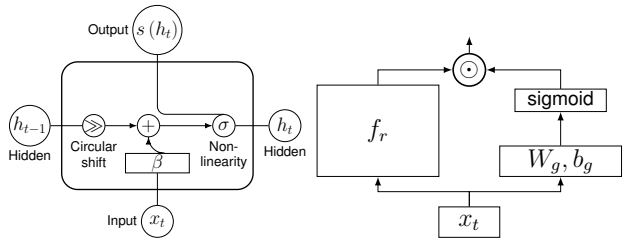

Figure 1: (a) The SRNN layer. The output of β is added to the shifted version of the previous hidden state, followed by a non-linearity, σ. The output at time t is obtained by a function s that is applied to the new hidden state. (b) The structure of network β. The primary sub-network f_r is an MLP. The gating sub-network (right branch) has a single affine layer and a sigmoid non-linearity. The outputs of the two branches are multiplied elementwise to produce β's output.

where $W_g \in \mathbb{R}^{d_h \times d_i}$ and $b_g \in \mathbb{R}^{d_h}$ are the weights and biases of the gating branch. Network β is depicted in Fig. 1(b).

The output of the network at time step t, o_t, is obtained by using a single affine layer s, $o_t = s\left(h_t\right)$.

Analysis of gradient dynamics Since W_p is a permutation matrix, it is also orthogonal. Our method benefits from this property, since the successive applications of W_p do not increase nor decrease the norm of the hidden state vector, such that the gradients w.r.t the hidden state do not suffer from an exponential growth or decay. Therefore, the method can be applied without the use of any gradient clipping schemes.

In addition, since the operator W_p is not learned, none of weights inside the layer appear with powers greater than one in the gradient. The gradient of the loss w.r.t a parameter β_k in network β is

$$\frac{\partial \mathcal{L}}{\partial \beta_k} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial h_t} \frac{\partial h_t}{\partial \beta_k} . \tag{6}$$

The derivative of Eq. 3 w.r.t β_k yields a recursive equation,

$$\frac{\partial h_t}{\partial \beta_k} = W_p \sigma'\left(z_t\right) \frac{\partial h_{t-1}}{\partial \beta_k} + \sigma'\left(z_t\right) \frac{\partial \beta\left(x_t\right)}{\partial \beta_k} . \tag{7}$$

Expanding Eq. 7 yields the closed form,

$$\frac{\partial h_t}{\partial \beta_k} = \sum_{i=1}^{t} \sigma'\left(z_i\right) \left(\prod_{j=1}^{i-1} \left[W_p \sigma'\left(z_j\right) \right] \right) \frac{\partial \beta\left(x_i\right)}{\partial \beta_k} . \tag{8}$$

Since W_p is orthogonal, the only terms that may influence the gradients' growth or decay are $\sigma'\left(z_t\right)$. For most commonly used activation functions, such as the sigmoid, $\tanh$ and ReLU, $\left|\sigma'\left(z_i\right)\right|$ is bounded by 1. Therefore, one obtains

$$\left| \frac{\partial h_t}{\partial \beta_k} \right| \leq \left| \sum_{i=1}^{t} \frac{\partial \beta\left(x_i\right)}{\partial \beta_k} \right| . \tag{9}$$

Combining this result with Eq. 6 reveals that

$$\left| \frac{\partial \mathcal{L}}{\partial \beta_k} \right| \leq \left| \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial h_t} \sum_{i=1}^{t} \frac{\partial \beta\left(x_i\right)}{\partial \beta_k} \right| . \tag{10}$$

Method	Time complexity	Method	Time complexity
Vanilla RNN	$O\left(Td_h^2 + Td_h d_i\right)$	uRNN	$O(Td_h \log d_h + Td_h d_i)$
GRU	$O(Td_h^2 + Td_h d_i)$	NRU	$O(Td_h^2 + Td_h d_i)$
LSTM	$O(Td_h^2 + Td_h d_i)$	SRNN	$O(Td_h d_i + Tkd_i^2)$

Table 1: Time complexity for one gradient step for sequence length T. d_h and d_i are the hidden state and input dimensions. k is the number of layers in f_r.

Method	LSTM	GRU	uRNN	NRU	SRNN
pMNIST	89.50%	91.87%	91.74%	91.64%	96.43%
Big pMNIST	33.60%	9.45%	OOM	11.01%	90.31%

Table 2: Accuracy for the pMNIST and its $4\times$ larger version. OOM=out of memory.

As $\frac{\partial \mathcal{L}}{\partial \beta_k}$ gains only linear contributions from the derivatives of β, it cannot explode.

For the ReLU activation function, the gradients in Eq. 8 vanish if and only if there is a time $t' > t$, where $z_{t'} < 0$. Since previous hidden states can only increase h_t, negative contributions only arise due to the outputs of network β. These outputs, $\beta(x_t)$, approximately follow a normal distribution of $\mathcal{N}(0, 1)$ (He et al. 2015). An estimation for the number of time steps it takes for a neuron in h_t to vanish is achieved by assuming that the probability of either obtaining a negative or a positive contribution to its activation at step t is $\frac{1}{2}$. This scenario is known as the Gambler's Ruin problem (Steele 2012). Although the probability of having the hidden state vanish, $p(z_t \leq 0) = \frac{T}{1+T}$, approaches one for $T \gg 1$, the expected number of steps until this happens is T.

Time complexity The computation of h_t does not involve any matrix multiplications between previous hidden-state h_{t-1}, and the permutation operator can be applied in $O(d_h)$. The most time-consuming operator is the application of the function β. However, since β does not rely on previous states, it can be applied in parallel to all time steps, thus greatly reducing the total runtime. The time complexity, in comparison to the literature methods, is presented in Table 1 for a minibatch of size one. It assumes that the number of hidden units in each layer of f_r is $O(d_i)$ and that $d_i \ll d_h$. Our method is the only one that is linear in d_h.

Experiments

We compare our SRNN architecture to the leading RNN architectures from the current literature. The baseline methods include: (1) a vanilla RNN, (2) LSTM (Hochreiter and Schmidhuber 1997), (3) GRU (Cho et al. 2014), (4) uRNN (Arjovsky, Shah, and Bengio 2016), (5) NRU (Chandar et al. 2019), and (6) nnRNN (Kerg et al. 2019). All methods, except NRU, employed the RMSProp (Bengio 2015) optimizer with a learning rate of 0.001 and a decay rate of 0.9. For NRU, we have used the suggested ADAM (Kingma and Ba 2014) optimizer with a learning rate of 0.001, and employed gradient clipping with a norm of one. The optimization parameters for nnRNN were taken from the official repository. For all problems involving

S.	Mtd	d_h	MSE	S.	Mtd	d_h	MSE	S.	Mtd	d_h	MSE
Lezcano et al.	LSTM	84	14.3	Lezcano	RGD	128	14.6	Our runs	uRNN	128	12.1
	LSTM	120	13.0		RGD	128	14.6		uRNN	256	10.8
	LSTM	158	12.6		RGD	192	14.5		uRNN	512	11.9
	EXPRNN	224	5.3		RGD	256	14.7		NRU	128	12.3
	EXPRNN	322	4.4	Our runs	LSTM	128	19.8		NRU	256	5.9
	EXPRNN	425	5.5		LSTM	256	15.9		NRU	512	3.2
	SCORNN	224	8.5		LSTM	512	12.3		NRU	1024	0.4
	SCORNN	322	7.8		LSTM	1024	8.7		SRNN	128	12.7
	SCORNN	425	7.4		GRU	128	39.7		SRNN	256	7.9
	EURNN	158	18.5		GRU	256	37.2		SRNN	512	4.7
	EURNN	256	15.3		GRU	512	33.0		SRNN	1024	1.3
	EURNN	378	15.2		GRU	1024	26.5		SRNN	2048	0.6

Table 3: MSE on TIMIT. d_h is the size of the hidden state. S=source; Mtd=Method.

Method	Source	Negative Log-likelihood
Vanilla RNN	Bai et al.	4.05
GRU	Bai et al.	3.46
LSTM	Bai et al.	3.29
SRNN	Our run	**3.22**
RNN R-transformer	Our run	2.34
GRU R-transformer	Our run	2.42
LSTM R-transformer	Our run	2.46
SRNN R-transformer	Our run	**2.27**

Table 4: Test set loss for the Nottingham Polyphonic Music Modeling task.

one-hot inputs, we have added an embedding layer before the RNN, since it benefited all methods. We believe that reports, which have deemed GRU as ineffective in some of the proposed benchmarks, did not include such a layer. The activation function σ in Eq. 3 and within the network β was a ReLU. The activation function of the vanilla RNN (denoted 'RNN' in the figures) was $\tanh$.

Copying Memory Problem RNNs are often challenged by the need to take into account information that has occurred in the distant past. The Copying Memory (MemCopy) task of (Arjovsky, Shah, and Bengio 2016) was designed to test the network's ability to recall information seen in the past. The objective is to memorize the first 10 characters of the sequence. Let $A = \{a_i\}_{i=1}^{8}$ be a set of 8 symbols, a_9 the blank symbol and a_{10}, the delimiter symbol. We create a sequence with the length $T + 20$, where the first 10 symbols are taken from A, then the next $T - 1$ symbols are the blank symbol a_9 followed by a single appearance of the delimiter symbol, a_{10}. The last 10 symbols are again set to the blank symbol a_9. The required output is a sequence of $T + 10$ blank symbols followed by the first 10 symbols taken from A of the original sequence. The baseline model for this task predicts some constant sequence after the appearance of the delimiter symbol, a_{10}. The cross-entropy for this solution is $\frac{10 \ln 8}{T+20}$.

We trained all models with a minibatch of size 20. We used a hidden size of $d_h = 128$ for all models. Network β contains one hidden layer with $f_r = 8$, i.e., f_r projects the

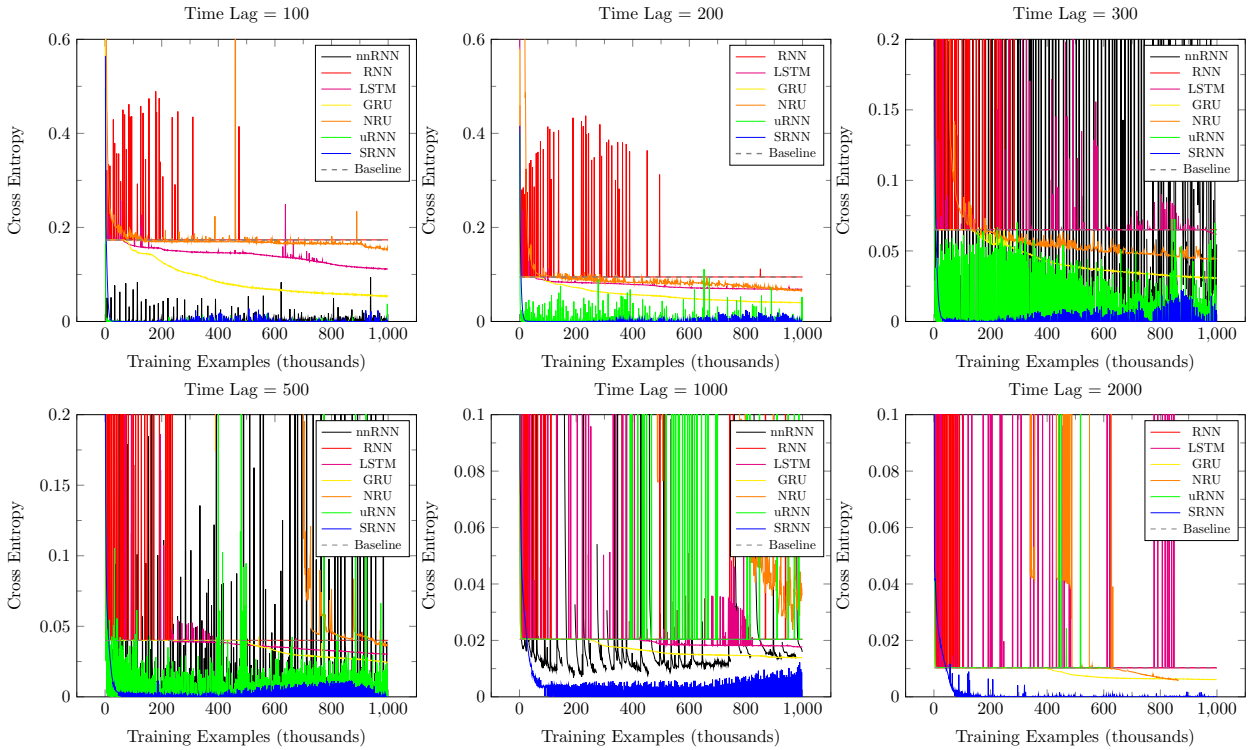

Figure 2: Results for the MemCopy task. Shown is the cross entropy as a function of the number of training examples. Each plot depicts the results for a different sequence length.

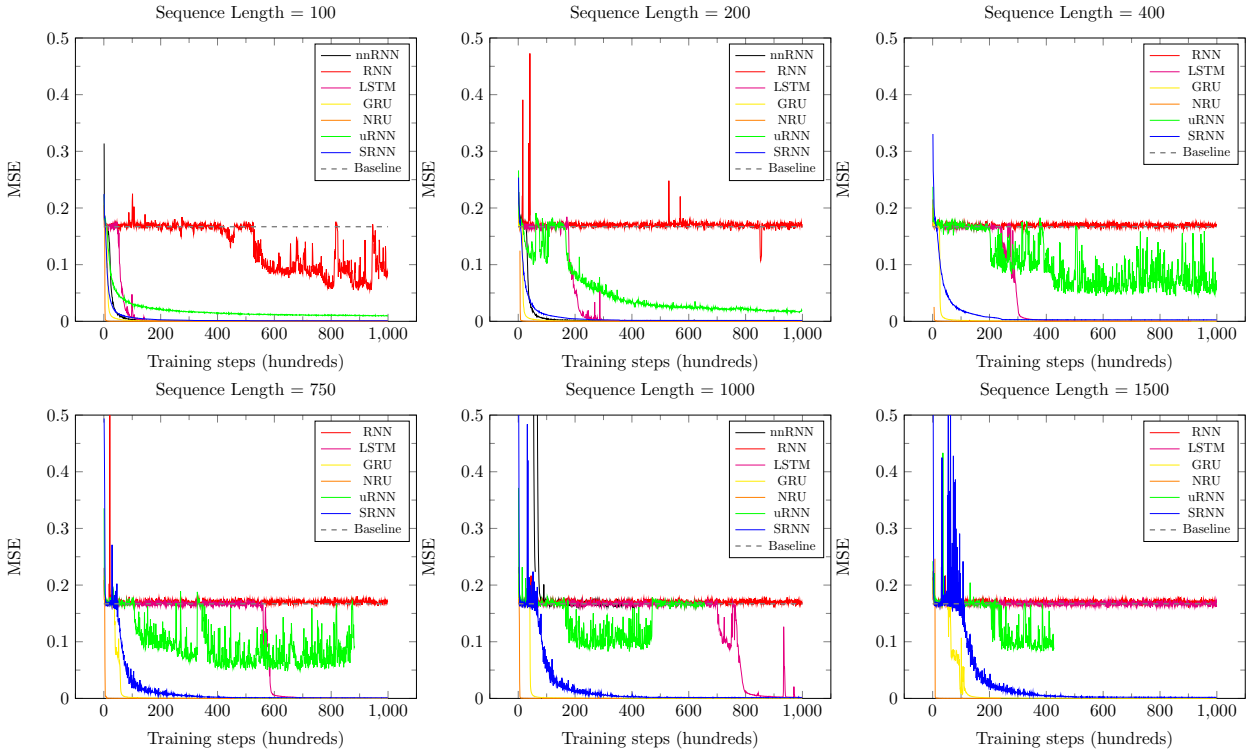

Figure 3: Results for the Adding problem for varying sequence lengths, noted on the top of each plot. Shown are MSE as a function of the number of training steps for our method and the baseline methods. For a longer sequence length, we sometimes stopped the run of uRNN before the allocated number of training iterations, due to the slow runtime of this method.

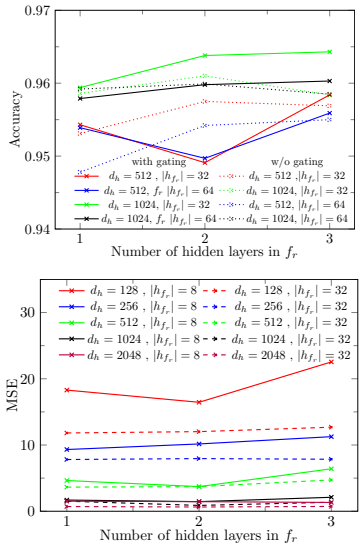

Figure 4: SRNN accuracy on pMNIST for multiple configurations. For each configuration, we report the accuracy for up to three hidden layers. Dashed lines employ no gating in network β (top). Same for TIMIT, reporting MSE. (bottom)

	Runtime per dataset (sec)				param
	MemCpy	Add	pMNIST	TIMIT	Add
RNN	0.18	1.53	19.64	1.76	17k
LSTM	0.26	1.76	22.88	1.80	67k
GRU	0.19	1.63	17.43	1.78	50k
NRU	7.13	61.77	835.47	42.98	102k
uRNN	13.48	112.78	1510.54	76.10	**2k**
SRNN	**0.07**	**0.77**	**8.59**	**1.27**	5k

Table 5: The runtime in seconds for one training epoch for various methods on a minibatch of 100. For MemCopy and Adding $T = 300$, SRNN's β network contains one hidden layer in f_r with 32 units. The number of samples in MemCopy epoch is 1000, and the number of samples in Adding epoch is 10000. In all cases $d_h = 128$. The right column reports the number of parameters in the Adding task.

input to activations in $\mathbb{R}^8$ and then to $\mathbb{R}^{128}$.

Fig. 2 shows the cross-entropy loss for all models for the time lag of $T = 100, 200, 300, 500, 1000, 2000$. While most methods are successful with sequences of length $T = 100$, only SRNN is able to deal efficiently with longer time lags. In contrast to results reported in the literature, GRU and LSTM are able to partially solve this problem, and they even do so better than uRNN for sequences larger than 1000, where it becomes unstable. However, the convergence rate of LSTM and GRU is very slow. Note that the cross entropy obtained for SRNN, while better than other methods, can increase as training progresses. This is similar to what is observed in Arjovsky, Shah, and Bengio (2016), for their uRNN method, in the cases where uRNN is successful. nnRNN is successful for $T \leq 300$ and is unstable for longer sequences.

Adding Problem The Adding problem was first introduced in Hochreiter and Schmidhuber (1997); We follow a close variant formulated in Arjovsky, Shah, and Bengio (2016). The input for this task consists of two sequences of length T (the two are concatenated to form x_t). The first sequence contains real numbers that have been uniformly sampled from $\mathcal{U}(0, 1)$. The second sequence is an indicator sequence, that is set to 0, except for two random entries that are set to 1. One of these entries is located in the first half of the sequence and the other in the last half of the sequence. The objective of this task is to output the sum of the two numbers in the first sequence that correspond to the location of the 1s in the second. The baseline to this task is the model which predicts 1, no matter what the input sequences are. The expected mean squared error (MSE) for this case is 0.167. A hidden size of 128 was used for all methods. All models were fed with a minibatch of 50. As in the MemCopy problem, all the training samples were generated on the fly. Network β of SRNN contains a hidden layer with size 8.

Fig. 3 shows the MSE for all models for sequence lengths of $T = 100, 200, 400, 750, 1000, 1500$. NRU, GRU and SRNN solve this problem quickly, with NRU showing very fast convergence. LSTM is also successful in solving this task, but its convergence is much slower. nnRNN is able to solve this task whenever it was able to initialize properly for sequence sizes shorter than 200.

Permuted Sequential MNIST The permuted MNIST (pMNIST) benchmark by Le, Jaitly, and Hinton (2015) measures the performance of RNNs, when modeling complex long-term dependencies. In this task, each MNIST image is reshaped into a 784-dimensional vector. This vector is fed to the RNN one entry at a time. The objective is to classify the image at the last time step. In order to increase the difficulty of the problem, a fixed permutation matrix is applied to each of the vectors before feeding them to the neural network. The same permutation and samples were used to train and evaluate all methods.

In this task, all models had on the order of 165k parameters, except for SRNN with 50k parameters. We tried to initialize nnRNN with the same initialization presented in the literature of 800k parameters. However, it failed, so it was trained with less. Unfortunately, we were not able to get the vanilla RNN to converge on this task and nnRNN initialization (which depends on a numerical solver) failed, despite multiple efforts done using the official code of this method.

For SRNN, a hidden state size of $d_h = 1024$ was used, and the function f_r of network β contained three hidden layers of size 32. A minibatch size of 100 was used for training, similar to the experiments performed for NRU. Models were trained for 60 epochs. Table 2 presents the accuracy over the test set for the epoch with the lowest cross entropy over the validation set. As can be seen, SRNN is more effective than all other methods. We cannot compare with IndRNN (Li et al. 2018), which uses a different split.

In order to verify that our method is not overly sensitive to its parameters, namely the dimensionality of the hidden state and the depth and number of hidden units per layer of network f_r (the primary sub-network of β), we tested multiple configurations. The results are reported in Fig. 4(a) and show

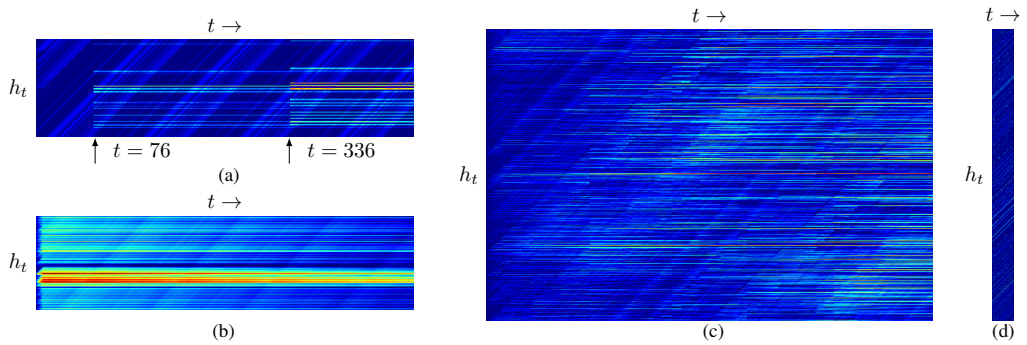

Figure 5: (a) The shifted hidden state of the SRNN for the Adding task. The arrows indicate the positions where the indicator sequence in x_t is set to 1. These positions split the hidden state into three regions. (b) The shifted hidden state of the SRNN for the MemCopy task. After $T = 10$ there is almost no change to the hidden state, as expected. (c) The shifted hidden state of the SRNN for the pMNIST task. Unlike the Adding or the MemCopy Task, the hidden state evolves through time, and the cell is able to suppress activations or accumulate additional information. (d) Shifted hidden state of the SRNN with 2048 units for the TIMIT task with a sequence length of $T = 151$. For better visualization, the shift that occurs in each time step has been removed.

that the method's accuracy is stable with respect to its parameters. As an ablation, we also report results in which gating is not used (the effect on the number of parameters is negligible). Without gating, performance somewhat decreases, especially for larger networks. However, overall, gating does not seem to be the most crucial part of the architecture.

Another version of pMNIST was tested, in which the MNIST image is padded with zeros, so that it is four times as large. The results are also reported in Table 2. Evidently, our method has an advantage in this dataset, which requires a much larger amount of memorization. Note that uRNN could not run due to GPU memory limitations, despite an effort to optimize the code using PyTorch JIT (Paszke et al. 2019).

TIMIT The TIMIT (Garofolo et al. 1993) speech frames prediction task was introduced by Wisdom et al. (2016) and later (following a fix to the way the error is computed) used in Lezcano-Casado and Martínez-Rubio (2019). The train/validation/test splits and exactly the same data used by previous work are employed here: using 3640 utterances for training , 192 for validating, and 400 for testing.

Table 3 reports the results of various methods, while varying the dimensionality of the hidden state, both for our runs, and for baselines obtained from Lezcano-Casado and Martínez-Rubio (2019). Despite some effort, we were not able to have the published code of nnRNN run on this dataset (failed initialization). Running uRNN for $d_h > 512$ was not feasible. As can be seen, our SRNN outperforms almost all of the baseline methods, including EXPRNN (Lezcano-Casado and Martínez-Rubio 2019), SCORNN (Helfrich, Willmott, and Ye 2018), EURNN (Jing et al. 2017), and RGD (Wisdom et al. 2016). The only method that outperforms SRNN is NRU. However, the NRU architecture has an order of magnitude more parameters than our method for a given hidden state size; NRU with $d_h = 1024$ has more than ten times the number of parameters than SRNN with $d_h = 2048$.

The SRNN model used on TIMIT has three hidden layers in f_r, each with 32 hidden neurons in each. However, the method is largely insensitive to these parameters, as can be seen in Fig. 4(b). Networks with 8 hidden neurons in each layer perform slightly worse than those with 32, and the number of layers has a small effect when the dimensionality of the hidden state is high enough.

Music Modeling The Nottingham Polyphonic Music dataset (Boulanger-Lewandowski, Bengio, and Vincent 2012) is a collection of British and American Folk tunes. It is a commonly used benchmark for the task of music modeling. We follow the same settings and data splitting presented by Bai, Kolter, and Koltun (2018), who applied RNNs followed by a transformer model (Vaswani et al. 2017). The training set consists of 694 tunes. 173 and 170 tunes are used as the validation and test set, respectively. Evaluation is done with the Negative Log-likelihood. For modeling with SRNN, we use a stack of 3 SRNN layers with one hidden layer inside network β with $f_r = 128$, and a hidden state size of $d_h = 2048$. The optimizer used was Adam with a learning rate of 0.001, where between each intermediate SRNN layer we also apply dropout with $p = 0.3$ to avoid overfitting.

We further show that SRNN can be integrated as a building block in methods that depend on a recurrent component. This is done for the R-Transformer (Wang et al. 2019), where we replaced the recurrent cell with SRNN. For all cell types, the hyperparameters used are the same as in R-transformer's official repository. The hidden size d_h for all methods is 160, where for SRNN, β contains one hidden layer with $f_r = 160$.

Table 4 reports the results of the various methods for music modeling. As can be seen, SRNN outperforms all other tested methods with and without the transformer layers.

Visualization of the hidden units Fig. 5 provides a visualization of the hidden state of the SRNN, h_t, for the Adding, MemCopy, pMNIST and TIMIT tasks. For better visualization, the plot has been modified so that the shift is removed. As can be seen, the method can retain information in cases in which memorization is needed, such as in the Adding problem. It also has the capacity to perform non-trivial computations in cases in which processing is needed, such as pMNIST. SRNN does not suffer from the vanishing

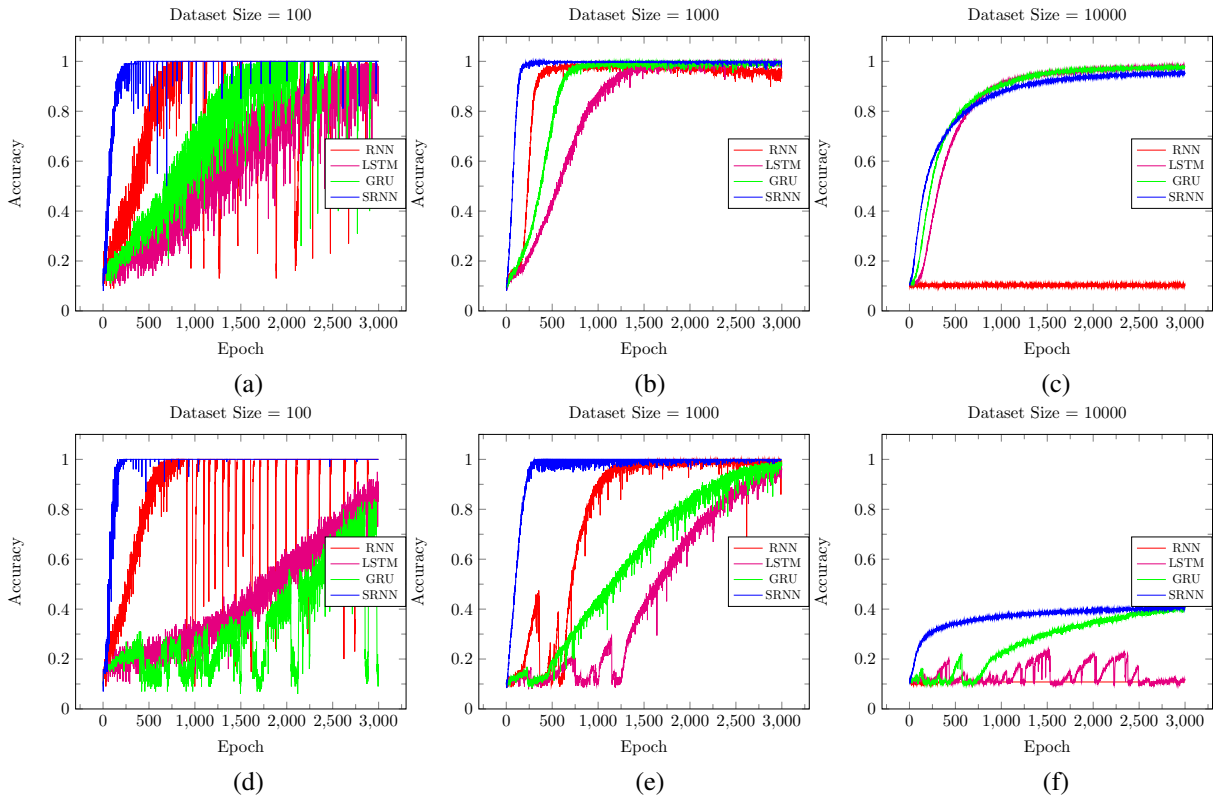

Figure 6: Fitting random labels for 100/1000/10000 samples of 8×8 MNIST patches (a-c, respectively). (d-f): same for 16×16.

gradient problem, since the activations are maintained over time. SRNN is able to remove irrelevant information for the future from the hidden state, an ability that is not required for the addition or the Memory Copy Tasks.

Capacity Since the orthogonal matrix SRNN employs is fixed, one may wonder if it is less expressive than other RNNs. The experiments below indicate that the capacity of SRNN is comparable to that of other methods with the same number of parameters. One way to measure capacity is to measure the ability to learn random labels (Collins, Sohl-Dickstein, and Sussillo 2016). In our experiments, the labels of MNIST are shuffled and a subset of N samples is randomly selected. To limit the sequence size, the MNIST images are cropped to a 8×8 or 16×16 center patches, so the generated sequences are now 64 or 256 steps long. For this experiment the models were selected to have an order of $15K$ parameters. Testing is done on the training samples, since the aim is to measure the ability to overfit random labels.

Fig. 6(a-c) presents the classification accuracy with respect to the random label for SRNN, LSTM, GRU and the vanilla RNN architectures for different subset sizes, $N = 100, 1000, 10000$ and cropped patches of size 8×8. As can be seen, for $N = 100, 1000$, all models are able to differentiate between the different samples, i.e., overfit to the random labels. For the case of $N = 10000$, all models succeed, except for the vanilla RNN. Fig. 6(d-f) depicts a similar experiment for the case of 16×16 patches. For $N = 100$ and $N = 1000$ all methods are able to overfit, however, SRNN shows a marked advantage in the number of required training epochs. For $N = 10,000$, only SRNN and GRU are able to (partially) overfit the data. Note that SRNN also fits the data faster than the other methods across the experiments, indicating that it may have a higher capacity.

Runtime Table 5 compares the runtime of our method, for a single epoch on each listed dataset, in comparison to the official implementations of other methods. As can be seen, our method is much more efficient than the other methods. In the same set of experiments, we also recorded the number of parameters reported by PyTorch. This number is the same across benchmarks, except for the embedding size, so the results reported are from the Adding problem, where the embedding size is negligible. The only method with fewer parameters than our method is uRNN, which parameterizes the unitary transformation very efficiently.

Conclusions

In this work, we introduce a new RNN architecture that does not suffer from vanishing and exploding gradients. The method employs gating only on the sub-network that processes the input, makes use of a fixed shifting operator as an orthogonal transformation of the hidden states, is efficient, and has a lower complexity than other RNN architectures. The new method obtains competitive results in comparison with previous methods, which rely on much more sophisticated machinery.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation program (grant ERC CoG 725974). The authors would like to thank Amit Dekel for valuable insights, Ameen Ali for his help with the music modeling task, and Daniel Dubinsky for the CUDA-optimized implementation. The contribution of the first author is part of a Ph.D. thesis research conducted at TAU.

References

Arjovsky, M.; Shah, A.; and Bengio, Y. 2016. Unitary Evolution Recurrent Neural Networks. In *ICML*, 1120–1128.

Bai, S.; Kolter, J. Z.; and Koltun, V. 2018. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271* .

Bengio, Y. 2015. Rmsprop and equilibrated adaptive learning rates for nonconvex optimization. *corr abs/1502.04390* .

Boulanger-Lewandowski, N.; Bengio, Y.; and Vincent, P. 2012. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv preprint arXiv:1206.6392* .

Chandar, S.; Sankar, C.; Vorontsov, E.; Kahou, S. E.; and Bengio, Y. 2019. Towards non-saturating recurrent units for modelling long-term dependencies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 3280–3287.

Cho, K.; van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; and Bengio, Y. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *EMNLP*.

Collins, J.; Sohl-Dickstein, J.; and Sussillo, D. 2016. Capacity and trainability in recurrent neural networks. *arXiv preprint arXiv:1611.09913* .

Garofolo, J. S.; Lamel, L. F.; Fisher, W. M.; Fiscus, J. G.; and Pallett, D. S. 1993. DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1. *NASA STI/Recon technical report n* 93.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, 1026–1034.

Helfrich, K.; Willmott, D.; and Ye, Q. 2018. Orthogonal Recurrent Neural Networks with Scaled Cayley Transform. In *International Conference on Machine Learning*, 1969–1978.

Henaff, M.; Szlam, A.; and LeCun, Y. 2016. Orthogonal RNNs and Long-Memory Tasks. *CoRR* abs/1602.06662. URL http://arxiv.org/abs/1602.06662.

Hochreiter, S.; and Schmidhuber, J. 1997. Long Short-Term Memory. *Neural Comput.* 9(8): 1735–1780. ISSN 0899-7667. doi:10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Jing, L.; Shen, Y.; Dubcek, T.; Peurifoy, J.; Skirlo, S.; LeCun, Y.; Tegmark, M.; and Soljačić, M. 2017. Tunable efficient unitary neural networks (eunn) and their application to rnns. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 1733–1741. JMLR. org.

Kerg, G.; Goyette, K.; Touzel, M. P.; Gidel, G.; Vorontsov, E.; Bengio, Y.; and Lajoie, G. 2019. Non-normal Recurrent Neural Network (nnRNN): learning long time dependencies while improving expressivity with transient dynamics. In *Advances in Neural Information Processing Systems*, 13591–13601.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* .

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

Lezcano-Casado, M.; and Martínez-Rubio, D. 2019. Cheap Orthogonal Constraints in Neural Networks: A Simple Parametrization of the Orthogonal and Unitary Group. In *International Conference on Machine Learning*, 3794–3803.

Li, S.; Li, W.; Cook, C.; Zhu, C.; and Gao, Y. 2018. Independently recurrent neural network (indrnn): Building a longer and deeper rnn. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 5457–5466.

Maduranga, K. D.; Helfrich, K. E.; and Ye, Q. 2019. Complex unitary recurrent neural networks using scaled cayley transform. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 4528–4535.

Mhammedi, Z.; Hellicar, A.; Rahman, A.; and Bailey, J. 2017. Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 2401–2409. JMLR. org.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, 8026–8037.

Steele, J. M. 2012. *Stochastic calculus and financial applications*, volume 45. Springer Science & Business Media.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *Advances in neural information processing systems*, 5998–6008.

Wang, Z.; Ma, Y.; Liu, Z.; and Tang, J. 2019. R-Transformer: Recurrent Neural Network Enhanced Transformer. *arXiv preprint arXiv:1907.05572* .

Wisdom, S.; Powers, T.; Hershey, J. R.; Roux, J. L.; and Atlas, L. 2016. Full-Capacity Unitary Recurrent Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, 4887–4895. Red Hook, NY, USA: Curran Associates Inc. ISBN 9781510838819.