Gated Feedback Recurrent Neural Networks

Junyoung Chung JUNYOUNG.CHUNG@UMONTREAL.CA
Caglar Gulcehre CAGLAR.GULCEHRE@UMONTREAL.CA
Kyunghyun Cho KYUNGHYUN.CHO@UMONTREAL.CA
Yoshua Bengio[*] FIND-ME@THE.WEB
Dept. IRO, Université de Montréal, [*]CIFAR Senior Fellow

Abstract

In this work, we propose a novel recurrent neural network (RNN) architecture. The proposed RNN, gated-feedback RNN (GF-RNN), extends the existing approach of stacking multiple recurrent layers by allowing and controlling signals flowing from upper recurrent layers to lower layers using a global gating unit for each pair of layers. The recurrent signals exchanged between layers are gated adaptively based on the previous hidden states and the current input. We evaluated the proposed GF-RNN with different types of recurrent units, such as $\tanh$, long short-term memory and gated recurrent units, on the tasks of character-level language modeling and Python program evaluation. Our empirical evaluation of different RNN units, revealed that in both tasks, the GF-RNN outperforms the conventional approaches to build deep stacked RNNs. We suggest that the improvement arises because the GF-RNN can adaptively assign different layers to different timescales and layer-to-layer interactions (including the top-down ones which are not usually present in a stacked RNN) by learning to gate these interactions.

1. Introduction

Recurrent neural networks (RNNs) have been widely studied and used for various machine learning tasks which involve sequence modeling, especially when the input and output have variable lengths. Recent studies have revealed that RNNs using gating units can achieve promising results in both classification and generation tasks (see, e.g., Graves, 2013; Bahdanau et al., 2014; Sutskever et al., 2014).

Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37. Copyright 2015 by the author(s).

Although RNNs can theoretically capture any long-term dependency in an input sequence, it is well-known to be difficult to train an RNN to actually do so (Hochreiter, 1991; Bengio et al., 1994; Hochreiter, 1998). One of the most successful and promising approaches to solve this issue is by modifying the RNN architecture e.g., by using a gated activation function, instead of the usual state-to-state transition function composing an affine transformation and a point-wise nonlinearity. A gated activation function, such as the long short-term memory (LSTM, Hochreiter & Schmidhuber, 1997) and the gated recurrent unit (GRU, Cho et al., 2014), is designed to have more persistent memory so that it can capture long-term dependencies more easily.

Sequences modeled by an RNN can contain both fast changing and slow changing components, and these underlying components are often structured in a hierarchical manner, which, as first pointed out by El Hihi & Bengio (1995) can help to extend the ability of the RNN to learn to model longer-term dependencies. A conventional way to encode this hierarchy in an RNN has been to stack multiple levels of recurrent layers (Schmidhuber, 1992; El Hihi & Bengio, 1995; Graves, 2013; Hermans & Schrauwen, 2013). More recently, Koutník et al. (2014) proposed a more explicit approach to partition the hidden units in an RNN into groups such that each group receives the signal from the input and the other groups at a separate, predefined rate, which allows feedback information between these partitions to be propagated at multiple timescales. Stollenga et al. (2014) recently showed the importance of feedback information across multiple levels of feature hierarchy, however, with feedforward neural networks.

In this paper, we propose a novel design for RNNs, called a gated-feedback RNN (GF-RNN), to deal with the issue of learning multiple adaptive timescales. The proposed RNN has multiple levels of recurrent layers like stacked RNNs do. However, it uses gated-feedback connections from upper recurrent layers to the lower ones. This makes the hidden states across a pair of consecutive timesteps fully connected. To encourage each recurrent layer to work at different timescales, the proposed GF-RNN controls the strength of the temporal (recurrent) connection adaptively. This effectively lets the model to adapt its structure based on the input sequence.

We empirically evaluated the proposed model against the conventional stacked RNN and the usual, single-layer RNN on the task of language modeling and Python program evaluation (Zaremba & Sutskever, 2014). Our experiments reveal that the proposed model significantly outperforms the conventional approaches on two different datasets.

2. Recurrent Neural Network

An RNN is able to process a sequence of arbitrary length by recursively applying a transition function to its internal hidden states for each symbol of the input sequence. The activation of the hidden states at timestep t is computed as a function f of the current input symbol $\mathbf{x}_t$ and the previous hidden states $\mathbf{h}_{t-1}$:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}).$$

It is common to use the state-to-state transition function f as the composition of an element-wise nonlinearity with an affine transformation of both $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$:

$$\mathbf{h}_t = \phi(W\mathbf{x}_t + U\mathbf{h}_{t-1}), \tag{1}$$

where W is the input-to-hidden weight matrix, U is the state-to-state recurrent weight matrix, and ϕ is usually a logistic sigmoid function or a hyperbolic tangent function.

We can factorize the probability of a sequence of arbitrary length into

$$p(x_1, \cdots, x_T) = p(x_1)p(x_2 \mid x_1) \cdots p(x_T \mid x_1, \cdots, x_{T-1}).$$

Then, we can train an RNN to model this distribution by letting it predict the probability of the next symbol x_{t+1} given hidden states $\mathbf{h}_t$ which is a function of all the previous symbols $x_1, \cdots, x_{t-1}$ and current symbol x_t:

$$p(x_{t+1} \mid x_1, \cdots, x_t) = g(\mathbf{h}_t).$$

This approach of using a neural network to model a probability distribution over sequences is widely used, for instance, in language modeling (see, e.g., Bengio et al., 2001; Mikolov, 2012).

2.1. Gated Recurrent Neural Network

The difficulty of training an RNN to capture long-term dependencies has been known for long (Hochreiter, 1991; Bengio et al., 1994; Hochreiter, 1998). A previously successful approaches to this fundamental challenge has been to modify the state-to-state transition function to encourage some hidden units to adaptively maintain long-term memory, creating paths in the time-unfolded RNN, such that gradients can flow over many timesteps.

Long short-term memory (LSTM) was proposed by Hochreiter & Schmidhuber (1997) to specifically address this issue of learning long-term dependencies. The LSTM maintains a separate memory cell inside it that updates and exposes its content only when deemed necessary. More recently, Cho et al. (2014) proposed a gated recurrent unit (GRU) which adaptively remembers and forgets its state based on the input signal to the unit. Both of these units are central to our proposed model, and we will describe them in more details in the remainder of this section.

2.1.1. Long Short-Term Memory

Since the initial 1997 proposal, several variants of the LSTM have been introduced (Gers et al., 2000; Zaremba et al., 2014). Here we follow the implementation provided by Zaremba et al. (2014).

Such an LSTM unit consists of a memory cell c_t, an *input* gate i_t, a *forget* gate f_t, and an *output* gate o_t. The memory cell carries the memory content of an LSTM unit, while the gates control the amount of changes to and exposure of the memory content. The content of the memory cell c_t^j of the j-th LSTM unit at timestep t is updated similar to the form of a gated leaky neuron, i.e., as the weighted sum of the new content $\tilde{c}_t^j$ and the previous memory content c_{t-1}^j modulated by the input and forget gates, i_t^j and f_t^j, respectively:

$$c_t^j = f_t^j c_{t-1}^j + i_t^j \tilde{c}_t^j, \tag{2}$$

where

$$\tilde{\mathbf{c}}_t = \tanh(W_c\mathbf{x}_t + U_c\mathbf{h}_{t-1}). \tag{3}$$

The input and forget gates control how much new content should be *memorized* and how much old content should be *forgotten*, respectively. These gates are computed from the previous hidden states and the current input:

$$\mathbf{i}_t = \sigma(W_i\mathbf{x}_t + U_i\mathbf{h}_{t-1}), \tag{4}$$

$$\mathbf{f}_t = \sigma(W_f\mathbf{x}_t + U_f\mathbf{h}_{t-1}), \tag{5}$$

where $\mathbf{i}_t = \left[i_t^k\right]_{k=1}^p$ and $\mathbf{f}_t = \left[f_t^k\right]_{k=1}^p$ are respectively the vectors of the input and forget gates in a recurrent layer composed of p LSTM units. $\sigma(\cdot)$ is an element-wise logistic sigmoid function. $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ are the input vector and previous hidden states of the LSTM units, respectively.

Once the memory content of the LSTM unit is updated, the hidden state h_t^j of the j-th LSTM unit is computed as:

$$h_t^j = o_t^j \tanh\left(c_t^j\right).$$

The output gate o_t^j controls to which degree the memory content is exposed. Similarly to the other gates, the output gate also depends on the current input and the previous hidden states such that

$$\mathbf{o}_t = \sigma \left(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} \right). \tag{6}$$

In other words, these gates and the memory cell allow an LSTM unit to adaptively *forget*, *memorize* and *expose* the memory content. If the detected feature, i.e., the memory content, is deemed important, the forget gate will be closed and carry the memory content across many timesteps, which is equivalent to capturing a long-term dependency. On the other hand, the unit may decide to reset the memory content by opening the forget gate. Since these two modes of operations can happen simultaneously across different LSTM units, an RNN with multiple LSTM units may capture both fast-moving and slow-moving components.

2.1.2. Gated Recurrent Unit

The GRU was recently proposed by Cho et al. (2014). Like the LSTM, it was designed to adaptively *reset* or *update* its memory content. Each GRU thus has a *reset* gate r_t^j and an *update* gate z_t^j which are reminiscent of the forget and input gates of the LSTM. However, unlike the LSTM, the GRU fully exposes its memory content each timestep and balances between the previous memory content and the new memory content strictly using leaky integration, albeit with its adaptive time constant controlled by update gate z_t^j.

At timestep t, the state h_t^j of the j-th GRU is computed by

$$h_t^j = (1 - z_t^j) h_{t-1}^j + z_t^j \tilde{h}_t^j, \tag{7}$$

where h_{t-1}^j and $\tilde{h}_t^j$ respectively correspond to the previous memory content and the new candidate memory content. The update gate z_t^j controls how much of the previous memory content is to be forgotten and how much of the new memory content is to be added. The update gate is computed based on the previous hidden states $\mathbf{h}_{t-1}$ and the current input $\mathbf{x}_t$:

$$\mathbf{z}_t = \sigma \left(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1} \right), \tag{8}$$

The new memory content $\tilde{h}_t^j$ is computed similarly to the conventional transition function in Eq. (1):

$$\tilde{\mathbf{h}}_t = \tanh \left(W \mathbf{x}_t + \mathbf{r}_t \odot U \mathbf{h}_{t-1} \right), \tag{9}$$

where $\odot$ is an element-wise multiplication.

One major difference from the traditional transition function (Eq. (1)) is that the states of the previous step $\mathbf{h}_{t-1}$ is modulated by the reset gates $\mathbf{r}_t$. This behavior allows a GRU to ignore the previous hidden states whenever it is deemed necessary considering the previous hidden states and the current input:

$$\mathbf{r}_t = \sigma \left(W_r \mathbf{x}_t + U_r \mathbf{h}_{t-1} \right). \tag{10}$$

The update mechanism helps the GRU to capture long-term dependencies. Whenever a previously detected feature, or the memory content is considered to be important for later use, the update gate will be closed to carry the current memory content across multiple timesteps. The reset mechanism helps the GRU to use the model capacity efficiently by allowing it to reset whenever the detected feature is not necessary anymore.

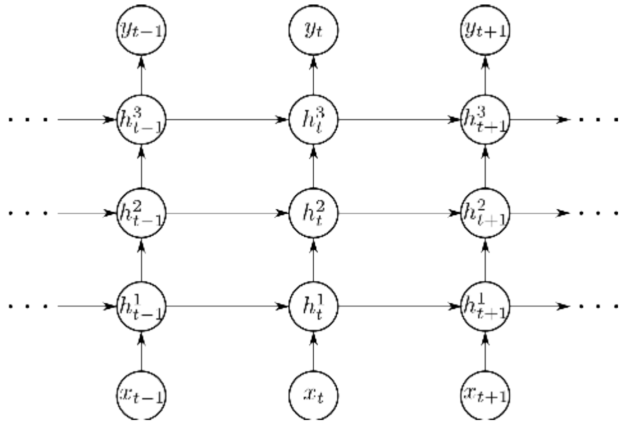
3. Gated Feedback Recurrent Neural Network

Although capturing long-term dependencies in a sequence is an important and difficult goal of RNNs, it is worthwhile to notice that a sequence often consists of both slow-moving and fast-moving components, of which only the former corresponds to long-term dependencies. Ideally, an RNN needs to capture both long-term and short-term dependencies.

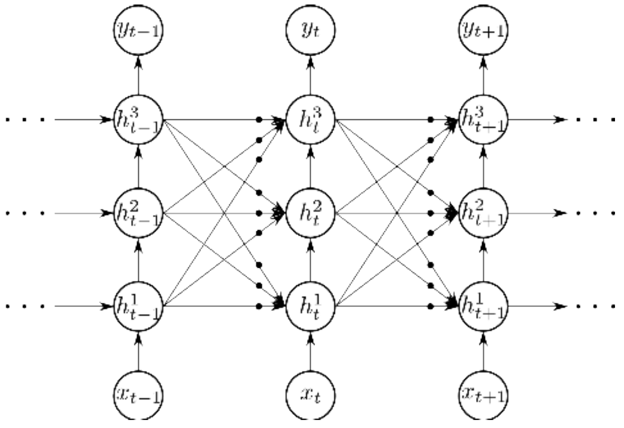
El Hihi & Bengio (1995) first showed that an RNN can capture these dependencies of different timescales more easily and efficiently when the hidden units of the RNN is explicitly partitioned into groups that correspond to different timescales. The clockwork RNN (CW-RNN) (Koutník et al., 2014) implemented this by allowing the i-th module to operate at the rate of 2^{i-1}, where i is a positive integer, meaning that the module is updated only when $t \bmod 2^{i-1} = 0$. This makes each module to operate at different rates. In addition, they precisely defined the connectivity pattern between modules by allowing the i-th module to be affected by j-th module when $j > i$.

Here, we propose to generalize the CW-RNN by allowing the model to adaptively adjust the connectivity pattern between the hidden layers in the consecutive timesteps. Similar to the CW-RNN, we partition the hidden units into multiple modules in which each module corresponds to a different layer in a stack of recurrent layers.

Unlike the CW-RNN, however, we do not set an explicit rate for each module. Instead, we let each module operate at different timescales by hierarchically stacking them. Each module is fully connected to all the other modules across the stack and itself. In other words, we do not define the connectivity pattern across a pair of consecutive timesteps. This is contrary to the design of CW-RNN and the conventional stacked RNN. The recurrent connection between two modules, instead, is gated by a logistic unit ($[0, 1]$) which is computed based on the current input and the previous states of the hidden layers. We call this gating



(a) Conventional stacked RNN



(b) Gated Feedback RNN

Figure 1. Illustrations of (a) conventional stacking approach and (b) gated-feedback approach to form a deep RNN architecture. Bullets in (b) correspond to global reset gates. Skip connections are omitted to simplify the visualization of networks.

unit a *global reset* gate, as opposed to a unit-wise reset gate which applies only to a single unit (See Eqs. (2) and (9)).

The global reset gate is computed as:

$$g^{i\to j} = \sigma\left(\mathbf{w}_g^{i\to j}\, \mathbf{h}_t^{j-1} + \mathbf{u}_g^{i\to j}\, \mathbf{h}_{t-1}^{*}\right),$$

where $\mathbf{h}_{t-1}^{*}$ is the concatenation of all the hidden states from the previous timestep $t-1$. The superscript ${}^{i\to j}$ is an index of associated set of parameters for the transition from layer i in timestep $t-1$ to layer j in timestep t. $\mathbf{w}_g^{i\to j}$ and $\mathbf{u}_g^{i\to j}$ are respectively the weight vectors for the current input and the previous hidden states. When $j=1$, $\mathbf{h}_t^{j-1}$ is $\mathbf{x}_t$.

In other words, the signal from $\mathbf{h}_{t-1}^{i}$ to $\mathbf{h}_t^{j}$ is controlled by a single scalar $g^{i\to j}$ which depends on the input $\mathbf{x}_t$ and all the previous hidden states $\mathbf{h}_{t-1}^{*}$.

We call this RNN with a fully-connected recurrent transitions and global reset gates, a *gated-feedback RNN* (GF-RNN). Fig. 1 illustrates the difference between the conventional stacked RNN and our proposed GF-RNN. In both models, information flows from lower recurrent layers to upper recurrent layers. The GF-RNN, however, further allows information from the upper recurrent layer, corresponding to coarser timescale, flows back into the lower recurrent layers, corresponding to finer timescales.

In the remainder of this section, we describe how to use the previously described LSTM unit, GRU, and more traditional $\tanh$ unit in the GF-RNN.

3.1. Practical Implementation of GF-RNN

$\tanh$ **Unit.** For a stacked $\tanh$-RNN, the signal from the previous timestep is gated. The hidden state of the j-th layer is computed by

$$\mathbf{h}_t^{j} = \tanh\left(W^{j-1\to j}\mathbf{h}_t^{j-1} + \sum_{i=1}^{L} g^{i\to j} U^{i\to j}\mathbf{h}_{t-1}^{i}\right),$$

where L is the number of hidden layers, $W^{j-1\to j}$ and $U^{i\to j}$ are the weight matrices of the current input and the previous hidden states of the i-th module, respectively. Compared to Eq. (1), the only difference is that the previous hidden states are from multiple layers and controlled by the global reset gates.

Long Short-Term Memory and Gated Recurrent Unit. In the cases of LSTM and GRU, we do not use the global reset gates when computing the unit-wise gates. In other words, Eqs. (4)–(6) for LSTM, and Eqs. (8) and (10) for GRU are not modified. We only use the global reset gates when computing the new state (see Eq. (3) for LSTM, and Eq. (9) for GRU).

The new memory content of an LSTM at the j-th layer is computed by

$$\tilde{\mathbf{c}}_t^{j} = \tanh\left(W_c^{j-1\to j}\mathbf{h}_t^{j-1} + \sum_{i=1}^{L} g^{i\to j} U_c^{i\to j}\mathbf{h}_{t-1}^{i}\right).$$

In the case of a GRU, similarly,

$$\tilde{\mathbf{h}}_t^{j} = \tanh\left(W^{j-1\to j}\mathbf{h}_t^{j-1} + \mathbf{r}_t^{j}\odot\sum_{i=1}^{L} g^{i\to j} U^{i\to j}\mathbf{h}_{t-1}^{i}\right).$$

4. Experiment Settings

4.1. Tasks

We evaluated the proposed GF-RNN on character-level language modeling and Python program evaluation. Both tasks are representative examples of discrete sequence modeling, where a model is trained to minimize the negative log-likelihood of training sequences:

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} - \log p \left(x_t^n \mid x_1^n, \ldots, x_{t-1}^n; \boldsymbol{\theta} \right),$$

where $\boldsymbol{\theta}$ is a set of model parameters.

4.1.1. Language Modeling

We used the dataset made available as a part of the human knowledge compression contest (Hutter, 2012). We refer to this dataset as the *Hutter dataset*. The dataset, which was built from English Wikipedia, contains 100 MBytes of characters which include Latin alphabets, non-Latin alphabets, XML markups and special characters. Closely following the protocols in (Mikolov et al., 2012; Graves, 2013), we used the first 90 MBytes of characters to train a model, the next 5 MBytes as a validation set, and the remaining as a test set, with the vocabulary of 205 characters including a token for an unknown character. We used the average number of bits-per-character (BPC, $E[-\log_2 P(x_{t+1}|\mathbf{h}_t)]$) to measure the performance of each model on the Hutter dataset.

4.1.2. Python Program Evaluation

Zaremba & Sutskever (2014) recently showed that an RNN, more specifically a stacked LSTM, is able to execute a short Python script. Here, we compared the proposed architecture against the conventional stacking approach model on this task, to which refer as *Python program evaluation*.

Scripts used in this task include addition, multiplication, subtraction, for-loop, variable assignment, logical comparison and if-else statement. The goal is to generate, or predict, a correct return value of a given Python script. The input is a program while the output is the result of a print statement: every input script ends with a print statement. Both the input script and the output are sequences of characters, where the input and output vocabularies respectively consist of 41 and 13 symbols.

The advantage of evaluating the models with this task is that we can artificially control the difficulty of each sample (input-output pair). The difficulty is determined by the number of nesting levels in the input sequence and the length of the target sequence. We can do a finer-grained analysis of each model by observing its behavior on examples of different difficulty levels.

In Python program evaluation, we closely follow (Zaremba & Sutskever, 2014) and compute the test accuracy as the next step symbol prediction given a sequence of correct preceding symbols.

Table 1. The sizes of the models used in character-level language modeling. Gated Feedback L is a GF-RNN with a same number of hidden units as a Stacked RNN (but more parameters). The number of units is shown as `(number of hidden layers)` $\times$ `(number of hidden units per layer)`.

Unit	Architecture	# of Units
tanh	Single	1×1000
	Stacked	3×390
	Gated Feedback	3×303
GRU	Single	1×540
	Stacked	3×228
	Gated Feedback	3×165
	Gated Feedback L	3×228
LSTM	Single	1×456
	Stacked	3×191
	Gated Feedback	3×140
	Gated Feedback L	3×191

4.2. Models

We compared three different RNN architectures: a single-layer RNN, a stacked RNN and the proposed GF-RNN. For each architecture, we evaluated three different transition functions: tanh + affine, long short-term memory (LSTM) and gated recurrent unit (GRU). For fair comparison, we constrained the number of parameters of each model to be roughly similar to each other.

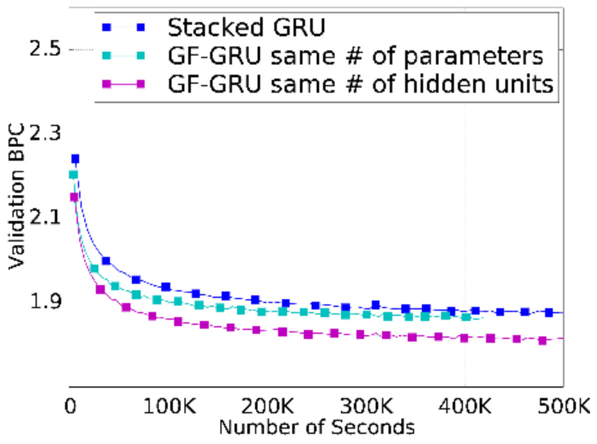
For each task, in addition to these capacity-controlled experiments, we conducted a few extra experiments to further test and better understand the properties of the GF-RNN.

4.2.1. Language Modeling

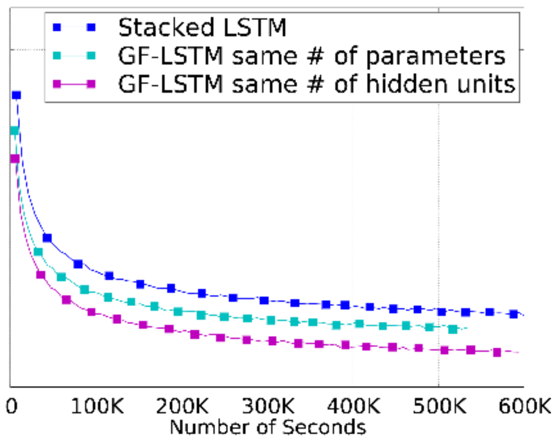
For the task of character-level language modeling, we constrained the number of parameters of each model to correspond to that of a single-layer RNN with 1000 tanh units (see Table 1 for more details). Each model is trained for at most 100 epochs.

We used RMSProp (Hinton, 2012) and momentum to tune the model parameters (Graves, 2013). According to the preliminary experiments and their results on the validation set, we used a learning rate of 0.001 and momentum coefficient of 0.9 when training the models having either GRU or LSTM units. It was necessary to choose a much smaller learning rate of 5×10^{-5} in the case of tanh units to ensure the stability of learning. Whenever the norm of the gradient explodes, we halve the learning rate.

Each update is done using a minibatch of 100 subsequences of length 100 each, to avoid memory overflow problems when unfolding in time for backprop. We approximate full back-propagation by carrying the hidden states computed at the previous update to initialize the hidden units in the next update. After every 100-th update, the hidden states



(a) GRU



(b) LSTM

Figure 2. Validation learning curves of three different RNN architectures; Stacked RNN, GF-RNN with the same number of model parameters, and GF-RNN with the same number of hidden units. The curves represent training up to 100 epochs. Best viewed in colors.

were reset to all zeros.

Table 2. Test set BPC (lower is better) of models trained on the Hutter dataset for a 100 epochs. ($*$) The gated-feedback RNN with the global reset gates fixed to 1 (see Sec. 5.1 for details). Bold indicates statistically significant winner over the column (same type of units, different overall architecture).

	tanh	GRU	LSTM
Single-layer	1.937	1.883	1.887
Stacked	**1.892**	1.871	1.868
Gated Feedback	1.949	**1.855**	**1.842**
Gated Feedback L	–	**1.813**	**1.789**
Feedback*	–	–	1.854

4.2.2. Python Program Evaluation

For the task of Python program evaluation, we used an RNN encoder-decoder based approach to learn the mapping from Python scripts to the corresponding outputs as done by Cho et al. (2014); Sutskever et al. (2014) for machine translation. When training the models, Python scripts are fed into the encoder RNN, and the hidden state of the encoder RNN is unfolded for 50 timesteps. Prediction is performed by the decoder RNN whose initial hidden state is initialized with the last hidden state of the encoder RNN. The first hidden state of encoder RNN h_0 is always initialized to a zero vector.

For this task, we used GRU and LSTM units either with or without the gated-feedback connections. Each encoder or decoder RNN has three hidden layers. For GRU, each hidden layer contains 230 units, and for LSTM each hidden layer contains 200 units.

Following Zaremba & Sutskever (2014), we used the *mixed* curriculum strategy for training each model, where each training example has a random difficulty sampled uniformly. We generated $320,000$ examples using the script provided by Zaremba & Sutskever (2014), with the nesting randomly sampled from $[1, 5]$ and the target length from $[1, 10^{10}]$.

We used Adam (Kingma & Ba, 2014) to train our models, and each update was using a minibatch with 128 sequences. We used a learning rate of 0.001 and β_1 and β_2 were both set to 0.99. We trained each model for 30 epochs, with early stopping based on the validation set performance to prevent over-fitting.

At test time, we evaluated each model on multiple sets of test examples where each set is generated using a fixed target length and number of nesting levels. Each test set contains $2,000$ examples which are ensured not to overlap with the training set.

5. Results and Analysis

5.1. Language Modeling

It is clear from Table 2 that the proposed gated-feedback architecture outperforms the other baseline architectures that we have tried when used together with widely used gated units such as LSTM and GRU. However, the proposed architecture failed to improve the performance of a vanilla-RNN with $\tanh$ units. In addition to the final modeling performance, in Fig. 2, we plotted the learning curves of some models against wall-clock time (measured in seconds). RNNs that are trained with the proposed gated-feedback architecture tends to make much faster progress over time. This behavior is observed both when the number of parameters is constrained and when the number of hid-

Table 3. Generated texts with our trained models. Given the seed at the left-most column (bold-faced font), the models predict next $200 \sim 300$ characters. Tabs, spaces and new-line characters are also generated by the models.

Seed	Stacked LSTM	GF-LSTM
`[[pl:Icon]]` `[[pt:Icon]]` `[[ru:Icon]]` `[[sv:Programspraket Icon]]</text>` `</revision>` `</page>` `<page>` `<title>Iconology</title>` `<id>14802</id>` `<revi`	**`<revi`**`sion>` `<id>15908383</id>` `<timestamp>` `2002-07-20T18:33:34Z` `</timestamp>` `<contributor>` `<username>The Courseichi</userrand` `vehicles in [[enguit]].` `==The inhibitors and alphabetsy and moral/` `hande in===In four [[communications]] and`	**`<revi`**`sion>` `<id>41968413</id>` `<timestamp>` `2006-09-03T11:38:06Z` `</timestamp>` `<contributor>` `<username>Navisb</username>` `<id>46264</id>` `</contributor>` `<comment>The increase from the time`
`<title>Inherence relation</title>` `<id>14807</id>` `<revision>` `<id>34980694</id>` `<timestamp>` `2006-01-13T04:19:25Z` `</timestamp>` `<contributor>` `<username>Ro`	**`<username>Ro`**`bert]]` `[[su:20 aves]]` `[[vi:10 Februari]]` `[[bi:16 agostoferosín]]` `[[pt:Darenetische]]` `[[eo:Hebrew selsowen]]` `[[hr:2 febber]]` `[[io:21 februari]]` `[[it:18 de februari]]`	**`<username>Ro`**`ma</username>` `<id>48</id>` `</contributor>` `<comment>Vly''' and when one hand` `is angels and [[ghost]] borted and` `''mask r:centrions]], [[Afghanistan]],` `[[Glencoddic tetrahedron]], [[Adjudan]],` `[[Dghacn]], for example, in which materials` `dangerous (carriers) can only use with one`

den units is constrained. This suggests that the proposed GF-RNN significantly facilitates optimization/learning.

Effect of Global Reset Gates

After observing the superiority of the proposed gated-feedback architecture over the single-layer or conventional stacked ones, we further trained another GF-RNN with LSTM units, but this time, after fixing the global reset gates to 1 to validate the need for the global reset gates. Without the global reset gates, feedback signals from the upper recurrent layers influence the lower recurrent layer fully without any control. The test set BPC of GF-LSTM without global reset gates was 1.854 which is in between the results of conventional stacked LSTM and GF-LSTM with global reset gates (see the last row of Table 2) which confirms the importance of adaptively gating the feedback connections.

Qualitative Analysis: Text Generation

Here we qualitatively evaluate the stacked LSTM and GF-LSTM trained earlier by generating text. We choose a subsequence of characters from the test set and use it as an initial seed. Once the model finishes reading the seed text, we let the model generate the following characters by sampling a symbol from *softmax* probabilities of a timestep and then provide the symbol as next input.

Given two seed snippets selected randomly from the test set, we generated the sequence of characters ten times for each model (stacked LSTM and GF-LSTM). We show one of those ten generated samples per model and per seed snippet in Table 3. We observe that the stacked LSTM failed to close the tags with `</username>` and `</contributor>` in both trials. However, the GF-LSTM succeeded to close both of them, which shows that it learned about the structure of XML tags. This type of behavior could be seen throughout all ten random generations.

Table 4. Test set BPC of neural language models trained on the Hutter dataset, MRNN = multiplicative RNN results from Sutskever et al. (2011) and Stacked LSTM results from Graves (2013).

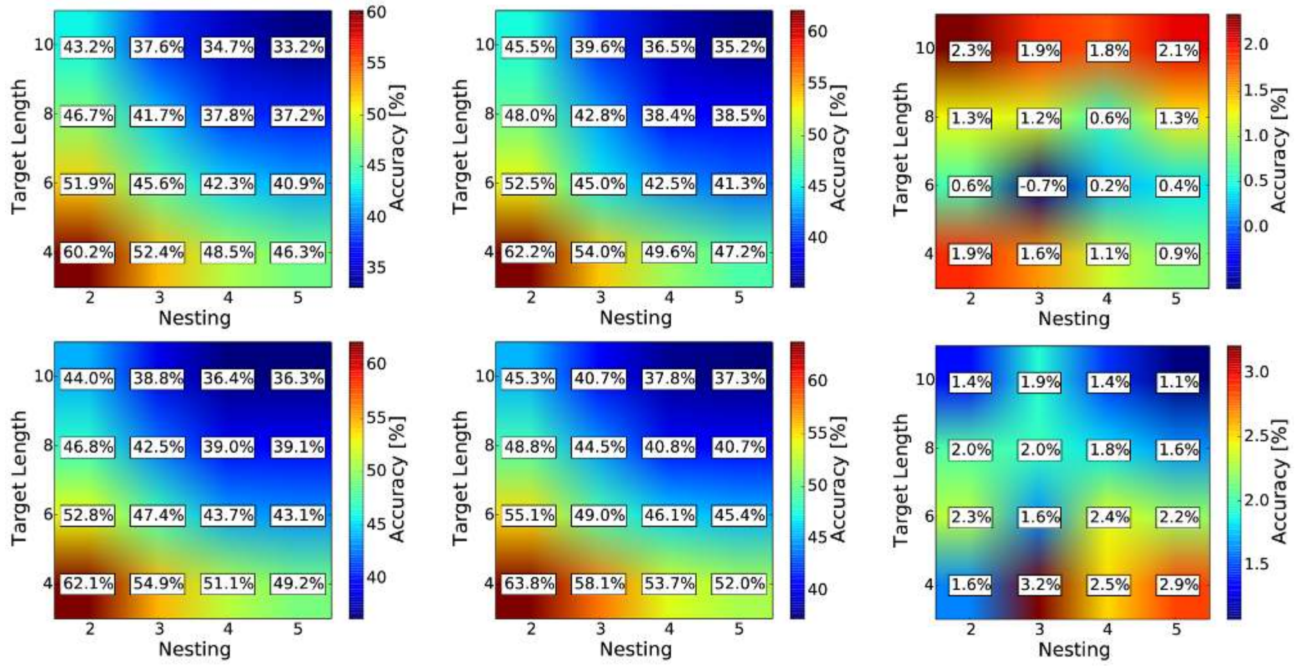
MRNN	Stacked LSTM	GF-LSTM
1.60	1.67	**1.58**

Large GF-RNN

We trained a larger GF-RNN that has five recurrent layers, each of which has 700 LSTM units. This makes it possible for us to compare the performance of the proposed architecture against the previously reported results using other types of RNNs. In Table 4, we present the test set BPC by a multiplicative RNN (Sutskever et al., 2011), a stacked LSTM (Graves, 2013) and the GF-RNN with LSTM units. The performance of the proposed GF-RNN is comparable to, or better than, the previously reported best results. Note that Sutskever et al. (2011) used the vocabulary of 86 characters (removed XML tags and the Wikipedia markups), and their result is not directly comparable with ours. In this experiment, we used Adam instead of RMSProp to optimize the RNN. We used learning rate of 0.001 and β_1 and β_2 were set to 0.9 and 0.99, respectively.

5.2. Python Program Evaluation

Fig. 3 presents the test results of each model represented in heatmaps. The accuracy tends to decrease by the growth



(a) Stacked RNN (b) Gated Feedback RNN (c) Gaps between (a) and (b)

Figure 3. Heatmaps of (a) Stacked RNN, (b) GF-RNN, and (c) difference obtained by substracting (a) from (b). The top row is the heatmaps of models using GRUs, and the bottom row represents the heatmaps of the models using LSTM units. Best viewed in colors.

of the length of target sequences or the number of nesting levels, where the difficulty or complexity of the Python program increases. We observed that in most of the test sets, GF-RNNs are outperforming stacked RNNs, regardless of the type of units. Fig. 3 (c) represents the gaps between the test accuracies of stacked RNNs and GF-RNNs which are computed by subtracting (a) from (b). In Fig. 3 (c), the red and yellow colors, indicating large gains, are concentrated on top or right regions (either the number of nesting levels or the length of target sequences increases). From this we can more easily see that the GF-RNN outperforms the stacked RNN, especially as the number of nesting levels grows or the length of target sequences increases.

6. Conclusion

We proposed a novel architecture for deep stacked RNNs which uses gated-feedback connections between different layers. Our experiments focused on challenging sequence modeling tasks of character-level language modeling and Python program evaluation. The results were consistent over different datasets, and clearly demonstrated that gated-feedback architecture is helpful when the models are trained on complicated sequences that involve long-term dependencies. We also showed that gated-feedback architecture was faster in wall-clock time over the training and achieved better performance compared to standard stacked RNN with a same amount of capacity. Large GF-LSTM was able to outperform the previously reported best results on character-level language modeling. This suggests that GF-RNNs are also scalable. GF-RNNs were able to outperform standard stacked RNNs and the best previous records on Python program evaluation task with varying difficulties.

We noticed a deterioration in performance when the proposed gated-feedback architecture was used together with a tanh activation function, unlike when it was used with more sophisticated gated activation functions. More thorough investigation into the interaction between the gated-feedback connections and the role of recurrent activation function is required in the future.

Acknowledgments

The authors would like to thank the developers of Theano (Bastien et al., 2012) and Pylearn2 (Goodfellow et al., 2013). Also, the authors thank Yann N. Dauphin and Laurent Dinh for insightful comments and discussion. We acknowledge the support of the following agencies for research funding and computing support: NSERC, Samsung, Calcul Québec, Compute Canada, the Canada Research Chairs and CIFAR.

References

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. Technical report, arXiv preprint arXiv:1409.0473, 2014.

Bastien, Frédéric, Lamblin, Pascal, Pascanu, Razvan, Bergstra, James, Goodfellow, Ian J., Bergeron, Arnaud, Bouchard, Nicolas, and Bengio, Yoshua. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

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

Bengio, Yoshua, Ducharme, Réjean, and Vincent, Pascal. A neural probabilistic language model. In *Adv. Neural Inf. Proc. Sys. 13*, pp. 932–938, 2001.

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

El Hihi, Salah and Bengio, Yoshua. Hierarchical recurrent neural networks for long-term dependencies. In *Advances in Neural Information Processing Systems*, pp. 493–499. Citeseer, 1995.

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

Goodfellow, Ian J., Warde-Farley, David, Lamblin, Pascal, Dumoulin, Vincent, Mirza, Mehdi, Pascanu, Razvan, Bergstra, James, Bastien, Frédéric, and Bengio, Yoshua. Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*, 2013.

Graves, Alex. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Hermans, Michiel and Schrauwen, Benjamin. Training and analysing deep recurrent neural networks. In *Advances in Neural Information Processing Systems*, pp. 190–198, 2013.

Hinton, Geoffrey. Neural networks for machine learning. Coursera, video lectures, 2012.

Hochreiter, Sepp. Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991. URL http://www7.informatik.tu-muenchen.de/~Ehochreit.

Hochreiter, Sepp. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

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

Hutter, Marcus. The human knowledge compression contest. 2012. URL http://prize.hutter1.net/.

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

Koutník, Jan, Greff, Klaus, Gomez, Faustino, and Schmidhuber, Jürgen. A clockwork rnn. In *Proceedings of the 31st International Conference on Machine Learning (ICML'14)*, 2014.

Mikolov, Tomas. *Statistical Language Models based on Neural Networks*. PhD thesis, Brno University of Technology, 2012.

Mikolov, Tomas, Sutskever, Ilya, Deoras, Anoop, Le, Hai-Son, Kombrink, Stefan, and Cernocky, J. Subword language modeling with neural networks. *Preprint*, 2012.

Schmidhuber, Jürgen. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992.

Stollenga, Marijn F, Masci, Jonathan, Gomez, Faustino, and Schmidhuber, Jürgen. Deep networks with internal selective attention through feedback connections. In *Advances in Neural Information Processing Systems*, pp. 3545–3553, 2014.

Sutskever, Ilya, Martens, James, and Hinton, Geoffrey E. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML'11)*, pp. 1017–1024, 2011.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc VV. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014.

Zaremba, Wojciech and Sutskever, Ilya. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

Zaremba, Wojciech, Sutskever, Ilya, and Vinyals, Oriol. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.