An Empirical Exploration of Recurrent Network Architectures

Rafal Jozefowicz RAFALJ@GOOGLE.COM
Google Inc.

Wojciech Zaremba WOJ.ZAREMBA@GMAIL.COM
New York University, Facebook[1]

Ilya Sutskever ILYASU@GOOGLE.COM
Google Inc.

Abstract

The Recurrent Neural Network (RNN) is an extremely powerful sequence model that is often difficult to train. The Long Short-Term Memory (LSTM) is a specific RNN architecture whose design makes it much easier to train. While wildly successful in practice, the LSTM's architecture appears to be ad-hoc so it is not clear if it is optimal, and the significance of its individual components is unclear.

In this work, we aim to determine whether the LSTM architecture is optimal or whether much better architectures exist. We conducted a thorough architecture search where we evaluated over ten thousand different RNN architectures, and identified an architecture that outperforms both the LSTM and the recently-introduced Gated Recurrent Unit (GRU) on some but not all tasks. We found that adding a bias of 1 to the LSTM's forget gate closes the gap between the LSTM and the GRU.

1. Introduction

The Deep Neural Network (DNN) is an extremely expressive model that can learn highly complex vector-to-vector mappings. The Recurrent Neural Network (RNN) is a DNN that is adapted to sequence data, and as a result the RNN is also extremely expressive. RNNs maintain a vector of activations for each timestep, which makes the RNN extremely deep. Their depth, in turn, makes them difficult to train due to the exploding and the vanishing gradient problems (Hochreiter, 1991; Bengio et al., 1994).

There have been a number of attempts to address the difficulty of training RNNs. Vanishing gradients were successfully addressed by Hochreiter & Schmidhuber (1997), who developed the Long Short-Term Memory (LSTM) architecture, which is resistant to the vanishing gradient problem. The LSTM turned out to be easy to use, causing it to become the standard way of dealing with the vanishing gradient problem. Other attempts to overcome the vanishing gradient problem include the use of powerful second-order optimization algorithms (Martens, 2010; Martens & Sutskever, 2011), regularization of the RNN's weights that ensures that the gradient does not vanish (Pascanu et al., 2012), giving up on learning the recurrent weights altogether (Jaeger & Haas, 2004; Jaeger, 2001), and a very careful initialization of RNN's parameters (Sutskever et al., 2013). Unlike the vanishing gradient problem, the exploding gradient problem turned out to be relatively easy to address by simply enforcing a hard constraint over the norm of the gradient (Mikolov, 2012; Pascanu et al., 2012).

A criticism of the LSTM architecture is that it is ad-hoc and that it has a substantial number of components whose purpose is not immediately apparent. As a result, it is also not clear that the LSTM is an optimal architecture, and it is possible that better architectures exist.

Motivated by this criticism, we attempted to determine whether the LSTM architecture is optimal by means of an extensive evolutionary architecture search. We found specific architectures similar to the Gated Recurrent Unit (GRU) (Cho et al., 2014) that outperformed the LSTM and the GRU by on most tasks, although an LSTM variant achieved the best results whenever dropout was used. In addition, by adding a bias of 1 to the LSTM's forget gate (a practice that has been advocated by Gers et al. (2000) but that has not been mentioned in recent LSTM papers),

[1]Work done while the author was at Google.

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

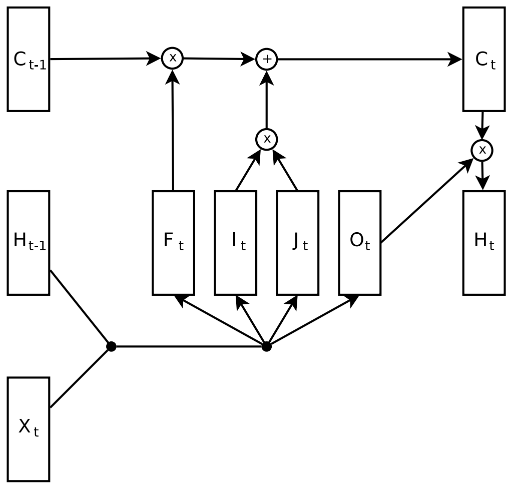

Figure 1. The LSTM architecture. The value of the cell is increased by $i_t \odot j_t$, where $\odot$ is element-wise product. The LSTM's output is typically taken to be h_t, and c_t is not exposed. The forget gate f_t allows the LSTM to easily reset the value of the cell.

we can close the gap between the LSTM and the better architectures. Thus, we recommend to increase the bias to the forget gate before attempting to use more sophisticated approaches.

We also performed ablative experiments to measure the importance of each of the LSTM's many components. We discovered that the input gate is important, that the output gate is unimportant, and that the forget gate is extremely significant on all problems except language modelling. This is consistent with Mikolov et al. (2014), who showed that a standard RNN with a hard-coded integrator unit (similar to an LSTM without a forget gate) can match the LSTM on language modelling.

2. Long Short-Term Memory

In this section, we briefly explain why RNNs can be difficult to train and how the LSTM addresses the vanishing gradient problem.

Standard RNNs suffer from both exploding and vanishing gradients (Hochreiter, 1991; Bengio et al., 1994). Both problems are caused by the RNN's iterative nature, whose gradient is essentially equal to the recurrent weight matrix raised to a high power. These iterated matrix powers cause the gradient to grow or to shrink at a rate that is exponential in the number of timesteps.

The exploding gradients problem is relatively easy to handle by simply shrinking gradients whose norms exceed a threshold, a technique known as gradient clipping (Mikolov, 2012; Pascanu et al., 2012). While learning would suffer if the gradient is reduced by a massive factor too frequently, gradient clipping is extremely effective whenever the gradient has a small norm the majority of the time.

The vanishing gradient is more challenging because it does not cause the gradient itself to be small; while the gradient's component in directions that correspond to long-term dependencies is small, while the gradient's component in directions that correspond to short-term dependencies is large. As a result, RNNs can easily learn the short-term but not the long-term dependencies.

The LSTM addresses the vanishing gradient problem by reparameterizing the RNN. Thus, while the LSTM does not have a representational advantage, its gradient cannot vanish. In the discussion that follows, let S_t denote a hidden state of an unspecified RNN architecture. The LSTM's main idea is that, instead of computing S_t from S_{t-1} directly with a matrix-vector product followed by a nonlinearity, the LSTM directly computes ΔS_t, which is then added to S_{t-1} to obtain S_t. At first glance, this difference may appear insignificant since we obtain the same S_t in both cases. And it is true that computing ΔS_t and adding it to S_t does not result in a more powerful model. However, just like a tanh-based network has better-behaved gradients than a sigmoid-based network, the gradients of an RNN that computes ΔS_t are nicer as well, since they cannot vanish.

More concretely, suppose that we run our architecture for 1000 timesteps to compute S_{1000}, and suppose that we wish to classify the entire sequence into two classes using S_{1000}. Given that $S_{1000} = \sum_{t=1}^{1000} \Delta S_t$, every single ΔS_t (including ΔS_1) will receive a sizeable contribution from the gradient at timestep 1000. This immediately implies that the gradient of the long-term dependencies cannot vanish. It may become "smeared", but it will never be negligibly small.

The full LSTM's definition includes circuitry for computing ΔS_t and circuitry for decoding information from S_t. Unfortunately, different practitioners use slightly different LSTM variants. In this work, we use the LSTM architecture that is precisely specified below. It is similar to the architecture of Graves (2013) but without peep-hole connections:

$$\begin{aligned}
i_t &= \tanh(W_{\mathrm{xi}} x_t + W_{\mathrm{hi}} h_{t-1} + b_{\mathrm{i}}) \\
j_t &= \mathrm{sigm}(W_{\mathrm{xj}} x_t + W_{\mathrm{hj}} h_{t-1} + b_{\mathrm{j}}) \\
f_t &= \mathrm{sigm}(W_{\mathrm{xf}} x_t + W_{\mathrm{hf}} h_{t-1} + b_{\mathrm{f}}) \\
o_t &= \tanh(W_{\mathrm{xo}} x_t + W_{\mathrm{ho}} h_{t-1} + b_{\mathrm{o}}) \\
c_t &= c_{t-1} \odot f_t + i_t \odot j_t \\
h_t &= \tanh(c_t) \odot o_t
\end{aligned}$$

In these equations, the W_* variables are the weight matrices and the b_* variables are the biases. The operation $\odot$

denotes the element-wise vector product. The variable c_t is the equivalent of S_t in the previous discussion, due to the equation $c_t = c_{t-1} \odot f_t + i_t \odot j_t$. The LSTM's hidden state is the concatenation (h_t, c_t). Thus the LSTM has two kinds of hidden states: a "slow" state c_t that fights the vanishing gradient problem, and a "fast" state h_t that allows the LSTM to make complex decisions over short periods of time. It is notable that an LSTM with n memory cells has a hidden state of dimension $2n$.

2.1. Arguments Against Attractors

There is a view that the information processing that takes place in biological recurrent neural networks is based on attractors (Amit, 1992; Plaut, 1995; Hinton & Shallice, 1991). It states that the RNN's state stores information with stable attractors that are robust to external perturbations. The attractor view is appealing because it allows the RNN to store a bit of information for an indefinite amount of time, which is a desirable property of information processing systems.

Bengio et al. (1994) showed that any RNN that stores one bit of information with a stable attractor must necessarily exhibit a vanishing gradient. As Theorem 4 of Bengio et al. (1994) does not make assumptions on the model architecture, it follows that an LSTM would also suffer from vanishing gradients had it stored information using attractors. But since LSTMs do not suffer from vanishing gradients, they should not be compatible with attractor-based memory systems.

2.2. Forget Gates Bias

There is an important technical detail that is rarely mentioned in discussions of the LSTM, and that is the initialization of the forget gate bias b_f. Most applications of LSTMs simply initialize the LSTMs with small random weights which works well on many problems. But this initialization effectively sets the forget gate to 0.5. This introduces a vanishing gradient with a factor of 0.5 per timestep, which can cause problems whenever the long term dependencies are particularly severe (such as the problems in Hochreiter & Schmidhuber (1997) and Martens & Sutskever (2011)).

This problem is addressed by simply initializing the forget gates b_f to a large value such as 1 or 2. By doing so, the forget gate will be initialized to a value that is close to 1, enabling gradient flow. This idea was present in Gers et al. (2000), but we reemphasize it since we found many practitioners to not be familiar with it.

If the bias of the forget gate is not properly initialized, we may erroneously conclude that the LSTM is incapable of learning to solve problems with long-range dependencies, which is not the case.

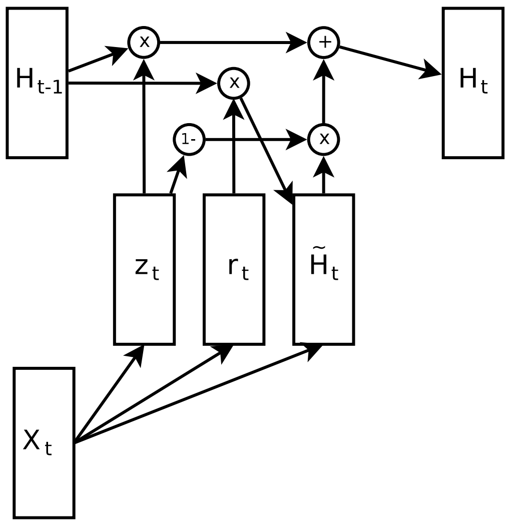

Figure 2. The Gated Recurrent Unit. Like the LSTM, it is hard to tell, at a glance, which part of the GRU is essential for its functioning.

2.3. LSTM Alternatives

It is easy to see that the LSTM's cell is essential since it is mechanism that prevents the vanishing gradient problem in the LSTM. However, the LSTM has many other components whose presence may be harder to justify, and they may not be needed for good results.

Recently, Cho et al. (2014) introduced the Gated Recurrent Unit (GRU), which is an architecture that is similar to the LSTM, and Chung et al. (2014) found the GRU to outperform the LSTM on a suite of tasks. The GRU is defined by the following equations:

$$\begin{aligned}
r_t &= \text{sigm}\left(W_\text{xr}x_t + W_\text{hr}h_{t-1} + b_\text{r}\right)\\
z_t &= \text{sigm}(W_\text{xz}x_t + W_\text{hz}h_{t-1} + b_\text{z})\\
\tilde{h}_t &= \tanh(W_\text{xh}x_t + W_\text{hh}(r_t \odot h_{t-1}) + b_\text{h})\\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t
\end{aligned}$$

The GRU is an alternative to the LSTM which is similarly difficult to justify. We compared the GRU to the LSTM and its variants, and found that the GRU outperformed the LSTM on nearly all tasks except language modelling with the naive initialization, but also that the LSTM nearly matched the GRU's performance once its forget gate bias was initialized to 1.

3. Methods

Our main experiment is an extensive architecture search. Our goal was to find a single architecture that outperforms the LSTM on three tasks (sec. 3.5) simultaneously. We succeeded in finding an architecture that outperformed the LSTM on all tasks; the same architecture also outperformed the GRU on all tasks though by a small margin. To outperform the GRU by a significant margin we needed to use different architectures on for different tasks.

3.1. The Search Procedure

Our search procedure is straightforward. We maintain a list of the 100 best architectures that we encountered so far, and estimate the quality of each architecture which is the performance of its best hyperparameter setting up to the current point. This top-100 list is initialized with only the LSTM and the GRU, which been evaluated for every allowable hyperparameter setting (sec. 3.7).

At each step of its execution the algorithm performs one of the following three steps:

1. It selects a random architecture from the list of its 100 best architectures, evaluates it on an 20 randomly chosen hyperparameter settings (see sec. 3.3 for the hyperparameter distribution) for each of the three tasks, and updates its performance estimate. We maintained a list of the evaluated hyperparameters to avoid evaluating the same hyperparameter configuration more than once. Each task is evaluated on an independent set of 20 hyperparameters, and the architecture's performance estimate is updated to

   $$\min_{\text{task}} \frac{\text{architecture's best accuracy on task}}{\text{GRU's best accuracy on task}} \quad (1)$$

   For a given task, this expression represents the performance of the best known hyperparameter setting over the performance of the best GRU (which has been evaluated on all allowable hyperparameter settings). By selecting the minimum over all tasks, we make sure that our search procedure will look for an architecture that works well on every task (as opposed to merely on most tasks).

2. Alternatively, it can propose a new architecture by choosing an architecture from the list of 100 best architectures mutating it (see sec. 3.2).

   First, we use the architecture to solve a simple memorization problem, where 5 symbols in sequence (with 26 possibilities) are to be read in sequence and then reproduced in the same order. This task is exceedingly easy for the LSTM, which can solve it with 25K parameters under four minutes with all but the most extreme settings of its hyperparameters. We evaluated the candidate architecture on *one* random setting of its hyperparameters. We would discard the architecture if its performance was below 95% with teacher forcing (Zaremba & Sutskever, 2014). The algorithm is allowed to re-examine a discarded architecture in future rounds, but it is not allowed to run it with a previously-run hyperparameter configuration.

   If an architecture passes the first stage, we evaluate the architecture on the first task on a set of 20 random hyperparameter settings from the distribution described in sec. 3.3. If the architecture's best performance exceeds 90% of the GRU's best performance, then the same procedure is applied to the second, and then to the third task, after which the architecture may be added to the list of the 100 best architectures (so it will displace one of the architectures in the top-100 list). If the architecture does not make it to the top-100 list after this stage, then the hyperparameters that it has been evauated on are stored, and if the architecture is randomly chosen again then it will not need to pass the memorization task and it will only be evaluated on fresh hyperparameter settings.

3.2. Mutation Generation

A candidate architecture is represented with a computational graph, similar to ones that are easily expressed in Theano (Bergstra et al., 2010) or Torch (Collobert et al., 2011). Each layer is represented with a node, where all nodes have the same dimensionality. Each architecture receives $M + 1$ nodes as inputs, and outputs M nodes as input, so the first M inputs are the RNN's hidden state at the previous timestep, while the $M + 1$th input is the external input to the RNN. So for example, the LSTM architecture has three inputs (x, h and c) and two outputs (h and c).

We mutate a given architecture by choosing a probability p uniformly at random from $[0, 1]$ and independently apply a randomly chosen transformations from the below with probability p to each node in the graph. The value of p determines the magnitude of the mutation — this way, most of the mutations are small while some are large. The transformations below are anchored to a "current node".

1. If the current node is an activation function, replace it with a randomly chosen activation function. The set of permissible nonlinearities is $\{\tanh(x)$, $\mathrm{sigmoid(x)}$, $\mathrm{ReLU(x)}, \mathrm{Linear}(0, x)$, $\mathrm{Linear}(1, x)$, $\mathrm{Linear}(0.9, x)$, $\mathrm{Linear}(1.1, x)\}$. Here the operator $\mathrm{Linear}(a, x)$ introduced a new square parameter matrix W and a bias b which replaces x by $W \cdot x + b$. The new matrix is initialized to small random values whose magnitude is determined by the scale parameter, and the scalar value a is added to its diagonal entries.

2. If the current node is an elementwise operation, replace it with a different randomly-chosen elementwise operation (multiplication, addition, subtraction) .

3. Insert a random activation function between the current node and one of its parents.
4. Remove the current node if it has one input and one output.
5. Replace the current node with a randomly chosen node from the current node's ancestors (i.e., all nodes that A depends on). This allows us to reduce the size of the graph.
6. Randomly select a node (node A). Replace the current node with either the sum, product, or difference of a random ancestor of the current node and a random ancestor of A. This guarantees that the resulting computation graph will remain well-defined.

We apply 1-3 of these transformations to mutate an architecture. We reject architectures whose number of output vectors is incompatible with the number of its input vectors. If an architecture is rejected we try again, until we find an architecture that is accepted.

3.3. Hyperparameter Selection for New Architectures

The random hyperparameters of a candidate architecture are chosen as follows. If the parent architecture has been evaluated on more than 420 different hyperparameters, then 80% of the hyperparameters for the child (i.e., candidate) would come from best 100 hyperparameter settings of the parent, and the remaining 20% would be generated completely randomly. If its parent had fewer than 420 hyperparameter evaluations, then 33% of the hyperparameters are generated completely randomly, 33% of the hyperparameters would are selected from the best 100 hyperparameters of the LSTM and the GRU (which have been evaluated with a full hyperparameter search and thus have known settings of good hyperparameters), and 33% of the hyperparameters are selected from the hyperparameter that have been tried on the parent architecture.

3.4. Some Statistics

We evaluated 10,000 different architectures and 1,000 of them made it past the initial filtering stage on the memorization problem. Each such architecture has been evaluated on 220 hyperparameter settings on average. These figures suggest that every architecture that we evaluated had a reasonable chance at achieving its highest performance. Thus we evaluated 230,000 hyperparameter configurations in total.

3.5. The Problems

We applied our architecture search to find an architecture that achieves good performance on the following three problems simultaneously.

- **Arithmetic** In this problem, the RNN is required to compute the digits of the sum or difference of two numbers, where it first reads the sequence which has the two arguments, and then emits their sum, one character at a time (as in Sutskever et al. (2014); Zaremba & Sutskever (2014)). The numbers can have up to 8 digits, and the numbers can be both positive and negative. To make the task more difficult, we introduced a random number of distractor symbols between successive input characters. Thus a typical instance would be **3**e**36**d**9-**h**1**h**39**f**94**eeh**43**keg**3**c**= -13991064**, which represents 3369-13994433= -13991064. The RNN reads the input sequence until it observes the "=" sign, after which it is to output the answer, one digit at a time. We used a next step prediction formulation, so the LSTM observes the correct value of the past digits when predicting a present digit. The total number of different symbols was 40: 26 lowercase characters, the 10 digits, and "+", "-","=", and "." (the latter represents the end of the output).

- **XML modelling**. The goal of this problem is to predict the next character in synthetic XML dataset. We would create an XML tag consisting of a random number (between 2 and 10) of lowercase characters. The generation procedure is as follows:
  - With 50% probability or after 50 iterations, we either close the most recently opened tag, or stop if no open tags remain
  - Otherwise, open a new random tag

  A short random sample is the following: "⟨etdomp⟩ ⟨pegshmnaj⟩ ⟨zbhbmg⟩ ⟨/zbhbmg⟩ ⟨/pegshmnaj⟩ ⟨autmh⟩ ⟨/autmh⟩ ⟨/etdomp⟩"

- **Penn Tree-Bank** (PTB). We also included a word-level language modelling task on the Penn TreeBank (Marcus et al., 1993) following the precise setup of Mikolov et al. (2010), which has 1M words with a vocabulary of size 10,000. We did not use dropout in the search procedure since the dropout models were too large and too expensive to properly evaluate, although we did evaluate our candidate architectures with dropout later.

Although we optimized our architectures on only the previous three tasks, we added the following Music datasets to measure the generalization ability of our architecture search procedure.

- **Music**. We used the polyphonic music datasets from Boulanger-Lewandowski et al. (2012). Each timestep is a binary vector representing the notes played at a given timestep. We evaluated the Nottingham and the Piano-midi datasets. Unlike the previous task, where we needed to predict discrete symbols and therefore used the softmax nonlinearity, we were required to predict binary vectors so we used the sigmoid. We did not use the music datasets in the search procedure.

For all problems, we used a minibatch of size 20, and unrolled the RNNs for 35 timesteps. We trained on sequences of length greater than 35 by moving the minibatch sequentially through the sequence, and ensuring that the hidden state of the RNN is initialized by the last hidden state of the RNN in the previous minibatch.

3.6. The RNN Training Procedure

We used a simple training procedure for each task. We trained a model with a fixed learning rate, and once its validation error stopped improving, we would lower the learning rate by a constant factor. The precise learning rate schedules are as follows:

- **Schedule for Architecture Search**. If we would observe no improvement in three consecutive epochs on the validation set, we would start lowering the learning rate by a factor of 2 at each epoch, for four additional epochs. Learning would then stop. We used this learning-rate schedule in the architecture search procedure, and all the results presented in table 1 are obtained with this schedule. We used this schedule in the architecture search because it was more aggressive and would therefore converge faster.
- **MUSIC Schedule for Evaluation**. If we observed no improvement over 10 consecutive epochs, we reduced the learning rate by a factor of 0.8. We would then "reset the counter", and reduce the learning rate by another factor 0.8 only after observing another 10 consecutive epochs of no improvement. We stopped learning once the learning rate became smaller than 10^{-3}, or when learning would progress beyond 800 epochs. The results in table 2 were produced with this learning rate schedule.
- **PTB Schedule for Evaluation**. We used the learning rate schedule of Zaremba et al. (2014). Specifically, we added an additional hyperparameter of "max-epoch" that determined the number of epochs with a constant learning rate. After training for this number of epochs, we begin to lower the learning rate by a factor of $decay$ each epoch for a total of 30 training epochs in total. We introduced three additional hyperparameters:
  1. max-epoch in $\{4, 7, 11\}$
  2. decay in $\{0.5, 0.65, 0.8\}$
  3. dropout in $\{0.0, 0.1, 0.3, 0.5\}$

  Although this increased the size of the hyperparameter space, we did not increase the number of random hyperparameter evaluations in this setting. This schedule was used for all results in table 3.

Arch.	Arith.	XML	PTB
Tanh	0.29493	0.32050	0.08782
LSTM	0.89228	0.42470	0.08912
LSTM-f	0.29292	0.23356	0.08808
LSTM-i	0.75109	0.41371	0.08662
LSTM-o	0.86747	0.42117	0.08933
LSTM-b	0.90163	0.44434	0.08952
GRU	0.89565	0.45963	0.09069
MUT1	**0.92135**	**0.47483**	0.08968
MUT2	0.89735	**0.47324**	0.09036
MUT3	0.90728	0.46478	**0.09161**

Table 1. Best next-step-prediction *accuracies* achieved by the various architectures (so larger numbers are better). LSTM-{f,i,o} refers to the LSTM without the forget, input, and output gates, respectively. LSTM-b refers to the LSTM with the additional bias to the forget gate.

3.7. Hyperparameter Ranges

We used the following ranges for the hyperparameter search.

- The initialization scale is in $\{0.3, 0.7, 1, 1.4, 2, 2.8\}$, where the neural network was initialized with the uniform distribution in $U[-x, x]$, where $x = \text{scale}/\sqrt{\text{number-of-units-in-layer}}$
- The learning rate was chosen from $\{0.1, 0.2, 0.3, 0.5, 1, 2, 5\}$, where the gradient was divided by the size of the minibatch
- The maximal permissible norm of the gradient was set to $\{1, 2.5, 5, 10, 20\}$
- The number of layers was chosen from $\{1, 2, 3, 4\}$
- The number of parameters is also a hyperparameter, although the range was task-dependent, as the different tasks required models of widely different sizes; We used binary search to select the appropriate number of units for a given number of parameters. For ARITH and XML the number of parameters was 250K or 1M; for Music it was 100K or 1M; and for PTB it was 5M. Surprisingly, we observed only slight performance gains for the larger models.

4. Results

We present the three best architectures that our search has discovered. While these architectures are similar to the

GRU, they outperform it on the tasks we considered.

MUT1:

$$\begin{aligned} z &= \mathrm{sigm}(W_{\mathrm{xz}}x_t + b_{\mathrm{z}}) \\ r &= \mathrm{sigm}(W_{\mathrm{xr}}x_t + W_{\mathrm{hr}}h_t + b_{\mathrm{r}}) \\ h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + \tanh(x_t) + b_{\mathrm{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT2:

$$\begin{aligned} z &= \mathrm{sigm}(W_{\mathrm{xz}}x_t + W_{\mathrm{hz}}h_t + b_{\mathrm{z}}) \\ r &= \mathrm{sigm}(x_t + W_{\mathrm{hr}}h_t + b_{\mathrm{r}}) \\ h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh}x_t + b_{\mathrm{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

MUT3:

$$\begin{aligned} z &= \mathrm{sigm}(W_{\mathrm{xz}}x_t + W_{\mathrm{hz}}\tanh(h_t) + b_{\mathrm{z}}) \\ r &= \mathrm{sigm}(W_{\mathrm{xr}}x_t + W_{\mathrm{hr}}h_t + b_{\mathrm{r}}) \\ h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh}x_t + b_{\mathrm{h}}) \odot z \\ &+ h_t \odot (1 - z) \end{aligned}$$

We report the performance of a baseline Tanh RNN, the GRU, the LSTM, the LSTM where the forget gate bias is set to 1 (LSTM-b), and the three best architectures discovered by the search procedure (named MUT1, MUT2, and MUT3). We also evaluated an LSTM without input gates (LSTM-i), an LSTM without output gates (LSTM-o), and an LSTM without forget gates (LSTM-f).

Our findings are summarized below:

- The GRU outperformed the LSTM on all tasks with the exception of language modelling
- MUT1 matched the GRU's performance on language modelling and outperformed it on all other tasks
- The LSTM significantly outperformed all other architectures on PTB language modelling when dropout was allowed
- The LSTM with the large forget bias outperformed both the LSTM and the GRU on almost all atsks
- While MUT1 is the best performer on two of the music datasets, it is notable that the LSTM-i and LSTM-o achieve the best results on the music dataset when dropout is used.

Results from the Music datasets are shown in table 2. We report the negative log probabilities of the various architectures on the PTB dataset in table 3.

Arch.	N	N-dropout	P
Tanh	3.612	3.267	6.809
LSTM	3.492	3.403	6.866
LSTM-f	3.732	3.420	6.813
LSTM-i	3.426	**3.252**	6.856
LSTM-o	3.406	**3.253**	6.870
LSTM-b	3.419	3.345	6.820
GRU	3.410	3.427	6.876
MUT1	**3.254**	3.376	**6.792**
MUT2	3.372	3.429	6.852
MUT3	3.337	3.505	6.840

Table 2. Negative Log Likelihood on the music datasets. N stands for Nottingham, N-dropout stands for Nottingham with nonzero dropout, and P stands for Piano-Midi.

Arch.	5M-tst	10M-v	20M-v	20M-tst
Tanh	4.811	4.729	4.635	4.582 (97.7)
LSTM	4.699	4.511	4.437	4.399 (81.4)
LSTM-f	4.785	4.752	4.658	4.606 (100.8)
LSTM-i	4.755	4.558	4.480	4.444 (85.1)
LSTM-o	4.708	4.496	4.447	4.411 (82.3)
LSTM-b	4.698	4.437	4.423	**4.380 (79.83)**
GRU	4.684	4.554	4.559	4.519 (91.7)
MUT1	4.699	4.605	4.594	4.550 (94.6)
MUT2	4.707	4.539	4.538	4.503 (90.2)
MUT3	4.692	4.523	4.530	4.494 (89.47)

Table 3. Perplexities on the PTB. The prefix (e.g., 5M) denotes the number of parameters in the model. The suffix "v" denotes validation negative log likelihood, the suffix"tst" refers to the test set. The perplexity for select architectures is reported in parentheses. We used dropout only on models that have 10M or 20M parameters, since the 5M models did not benefit from dropout at all, and most dropout-free models achieved a test perplexity of 108, and never greater than 120. In particular, the perplexity of the best models without dropout is below 110, which outperforms the results of Mikolov et al. (2014).

Our results also shed light on the significance of the various LSTM gates. The forget gate turns out to be of the greatest importance. When the forget gate is removed, the LSTM exhibits drastically inferior performance on the ARITH and the XML problems, although it is relatively unimportant in language modelling, consistent with Mikolov et al. (2014). The second most significant gate turns out to be the input gate. The output gate was the least important for the performance of the LSTM. When removed, h_t simply becomes $\tanh(c_t)$ which was sufficient for retaining most of the LSTM's performance. However, the results on the music dataset with dropout do not conform to this pattern, since there LSTM-i and LSTM-o achieved the best performance.

But most importantly, we determined that adding a positive bias to the forget gate greatly improves the performance of the LSTM. Given that this technique the simplest to implement, we recommend it for every LSTM implementation. Interestingly, this idea has already been stated in the paper that introduced the forget gate to the LSTM (Gers et al., 2000).

5. Discussion and Related Work

Architecture search for RNNs has been previously conducted by Bayer et al. (2009). They attempted to address the same problem, but they have done many fewer experiments, with small models (5 units). They only considered synthetic problems with long term dependencies, and they were able to find architectures that outperformed the LSTM on these tasks. The simultaneous work of Greff et al. (2015) have reached similar conclusions regarding the importance of the different gates of the LSTM.

We have evaluated a variety of recurrent neural network architectures in order to find an architecture that reliably outperforms the LSTM. Though there were architectures that outperformed the LSTM on some problems, we were unable to find an architecture that consistently beat the LSTM and the GRU in all experimental conditions.

The main criticism of this work is that our search procedure failed to find architectures that were significantly different from their parents. Indeed, reviewing the three highest performing architectures, we see that they are all similar to the GRU. While a much longer search procedure would have found more diverse architectures, the high cost of evaluating new candidates greatly reduces the feasibility of doing so. Nonetheless, the fact a reasonable search procedure failed to dramatically improve over the LSTM suggests that, at the very least, if there are architectures that are much better than the LSTM, then they are not trivial to find.

Importantly, adding a bias of size 1 significantly improved the performance of the LSTM on tasks where it fell behind the GRU and MUT1. Thus we recommend adding a bias of 1 to the forget gate of every LSTM in every application; it is easy to do often results in better performance on our tasks. This adjustment is the simple improvement over the LSTM that we set out to discover.

References

Amit, Daniel J. *Modeling brain function: The world of attractor neural networks*. Cambridge University Press, 1992.

Bayer, Justin, Wierstra, Daan, Togelius, Julian, and Schmidhuber, Jürgen. Evolving memory cell structures for sequence learning. In *Artificial Neural Networks–ICANN 2009*, pp. 755–764. Springer, 2009.

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

Bergstra, James, Breuleux, Olivier, Bastien, Frédéric, Lamblin, Pascal, Pascanu, Razvan, Desjardins, Guillaume, Turian, Joseph, Warde-Farley, David, and Bengio, Yoshua. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010.

Boulanger-Lewandowski, Nicolas, Bengio, Yoshua, and Vincent, Pascal. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv preprint arXiv:1206.6392*, 2012.

Cho, Kyunghyun, van Merrienboer, 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.

Chung, Junyoung, Gulcehre, Caglar, Cho, KyungHyun, and Bengio, Yoshua. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Collobert, Ronan, Kavukcuoglu, Koray, and Farabet, Clément. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

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

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

Greff, Klaus, Srivastava, Rupesh Kumar, Koutník, Jan, Steunebrink, Bas R, and Schmidhuber, Jürgen. Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

Hinton, Geoffrey E and Shallice, Tim. Lesioning an attractor network: investigations of acquired dyslexia. *Psychological review*, 98(1):74, 1991.

Hochreiter, Sepp. Untersuchungen zu dynamischen neuronalen netzen. *Master's thesis, Institut fur Informatik, Technische Universitat, Munchen*, 1991.

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

Jaeger, Herbert. The echo state approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148:34, 2001.

Jaeger, Herbert and Haas, Harald. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004.

Marcus, Mitchell P, Marcinkiewicz, Mary Ann, and Santorini, Beatrice. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

Martens, James. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 735–742, 2010.

Martens, James and Sutskever, Ilya. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1033–1040, 2011.

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

Mikolov, Tomas, Karafiát, Martin, Burget, Lukas, Cernockỳ, Jan, and Khudanpur, Sanjeev. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pp. 1045–1048, 2010.

Mikolov, Tomas, Joulin, Armand, Chopra, Sumit, Mathieu, Michael, and Ranzato, Marc'Aurelio. Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753*, 2014.

Pascanu, Razvan, Mikolov, Tomas, and Bengio, Yoshua. On the difficulty of training recurrent neural networks. *arXiv preprint arXiv:1211.5063*, 2012.

Plaut, David C. Semantic and associative priming in a distributed attractor network. In *Proceedings of the 17th annual conference of the cognitive science society*, volume 17, pp. 37–42. Pittsburgh, PA, 1995.

Sutskever, Ilya, Martens, James, Dahl, George, and Hinton, Geoffrey. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1139–1147, 2013.

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.