Journal of Machine Learning Research 16 (2015) 547-551 Submitted 7/13; Published 3/15

# Introducing CURRENNT: The Munich Open-Source CUDA RecurREnt Neural Network Toolkit

**Felix Weninger** WENINGER@TUM.DE
**Johannes Bergmann** PRIVAT@JOHANNES-BERGMANN.DE
**Björn Schuller**[*] SCHULLER@TUM.DE
*Machine Learning & Signal Processing, Technische Universität München, 80290 Munich, Germany*

**Editor:** Mikio Braun

## Abstract

In this article, we introduce CURRENNT, an open-source parallel implementation of deep recurrent neural networks (RNNs) supporting graphics processing units (GPUs) through NVIDIA's Computed Unified Device Architecture (CUDA). CURRENNT supports uni- and bidirectional RNNs with Long Short-Term Memory (LSTM) memory cells which overcome the vanishing gradient problem. To our knowledge, CURRENNT is the first publicly available parallel implementation of deep LSTM-RNNs. Benchmarks are given on a noisy speech recognition task from the 2013 2nd CHiME Speech Separation and Recognition Challenge, where LSTM-RNNs have been shown to deliver best performance. In the result, double digit speedups in bidirectional LSTM training are achieved with respect to a reference single-threaded CPU implementation. CURRENNT is available under the GNU General Public License from http://sourceforge.net/p/currennt.

**Keywords:** parallel computing, deep neural networks, recurrent neural networks, Long Short-Term Memory

## 1. Introduction

Recurrent neural networks (RNNs) are known as powerful sequence learners. In particular, the Long Short-Term Memory (LSTM) architecture has been proven to provide excellent modeling of language (Sundermeyer et al., 2012), music (Eck and Schmidhuber, 2002), speech (Graves et al., 2013), and facial expressions (Wöllmer et al., 2012). LSTM units overcome the vanishing gradient problem of traditional RNNs by the introduction of a memory cell which can be controlled by input, output and reset operations (Gers et al., 2000). In particular, recent research demonstrates that deep LSTM-RNNs exhibit superior performance in speech recognition in comparison to state-of-the-art deep feed forward networks (Graves et al., 2013). However, in contrast to the widespread usage of the latter (Hinton et al., 2012), RNNs are still not adopted by the research community at large. One of the major barriers is the lack of high-performance implementations for training RNNs; at the same time, such implementations are non-trivial due to the limited parallelism caused by time dependencies. To the best of our knowledge, there is no publicly available software dedicated to parallel LSTM-RNN training. Thus, we introduce our CUDA RecurREnt Neural Network Toolkit (CURRENNT) which exploits a mini-batch learning scheme performing parallel weight

[*]. B. Schuller is also with the Department of Computing, Imperial College London, UK.

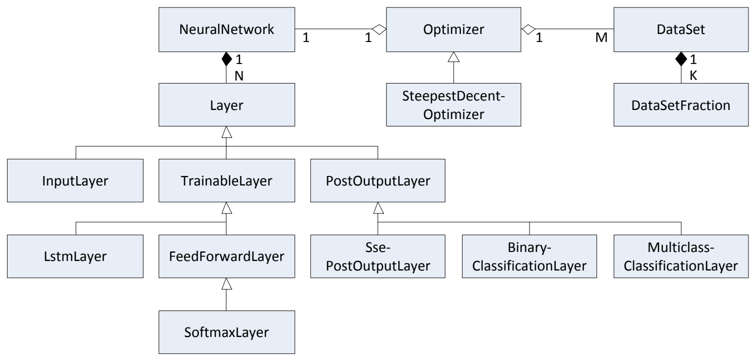

Figure 1: CURRENNT's C++ classes for deep feedforward and LSTM-RNN modeling.

updates from multiple sequences. CURRENNT implements learning from big data sets that do not fit into memory by means of a random access data format. GPUs are supported through NVIDIA's Computed Unified Device Architecture (CUDA). The RNN structure implemented in CURRENNT is based on LSTM units and in addition, feedforward network training is supported. Besides simple regression, CURRENNT also includes logistic and softmax output layers for training of binary and multi-way classification.

To briefly refer some related studies and freely available implementations: A 'reference' CPU implementation of LSTM-RNNs as used by Graves (2008) is available as open-source C++ code (Graves, 2013). A Python library for many machine learning algorithms including LSTM-RNN has been introduced by Schaul et al. (2010); however, it does not directly support parallel processing. Multi-core training of (standard) RNNs has been investigated by Cernanský (2009), but the source code is not available. Pascanu et al. have recently released a Python implementation ('GroundHog') of various RNN types described in their study (Pascanu et al., 2014), exploiting GPU-accelerated training through Theano; yet, it does not provide LSTM-RNNs, and at the moment there is no user-friendly interface.

## 2. Design

CURRENNT provides a C++ class library for deep LSTM-RNN modeling (cf. Figure 1) and a command line application for network training and evaluation. The network architecture can be specified by the user in JavaScript Object Notation (JSON), and trained parameters are saved in the same format, allowing, e.g., for deep learning with pre-training. For efficiency reasons, features for training and evaluation are given in binary format, adhering to the NetCDF standard, but network outputs can also be saved in CSV format to facilitate post-processing. All C++ code is designed to be platform independent and has been tested on Windows and various Linux distributions. The required CUDA compute capability is 1.3 (2008), allowing usage on virtually all of the consumer grade NVIDIA GPUs deployed in today's desktop PCs. The behavior of the gradient descent training algorithm is controlled by various switches of the command line application, allowing, e.g., for on-line or batch learning and fine-tuning of the training algorithm such as adding Gaussian noise to the input activations and randomly shuffling training data in on-line learning to improve generalization (Graves et al., 2013). The interested reader is referred to the documentation for more details.

## 3. Implementation

Deep LSTM-RNNs with $N$ layers are implemented as follows. An input sequence $\mathbf{x}_t$ is mapped to the output sequence $\mathbf{y}_t$, $t = 1, \ldots, T$ through the iteration (*forward pass*):

$$\begin{aligned}\mathbf{h}_t^{(0)} &:= \mathbf{x}_t,\\ \mathbf{h}_t^{(n)} &:= \mathcal{L}_t^{(n)}\left(\mathbf{h}_t^{(n-1)}, \mathbf{h}_{t-1}^{(n)}\right),\\ \mathbf{y}_t &:= \mathcal{S}\left(\mathbf{W}^{(N),(N+1)}\mathbf{h}_t^{(N)} + \mathbf{b}^{(N+1)}\right).\end{aligned}$$

In the above and the ongoing, $\mathbf{W}$ denotes weight matrices and $\mathbf{b}$ stands for bias vectors (with superscripts denoting layer indices). $\mathbf{h}_t^{(n)}$ denotes the hidden feature representation of time frame $t$ in the level $n$ units, $n = 1, \ldots, N$. The 0-th layer is the input layer and the $N+1$-th layer the output layer. $\mathcal{S}$ is the (vector valued) output layer function, e.g., a softmax function for multi-way classification (cf. Figure 1). $\mathcal{L}_t^{(n)}$ denotes the composite LSTM activation function which is used instead of the common simple sigmoid-shaped functions. The crucial point is to augment each unit with a state variable $c_t$, resulting in an automaton-like structure. The hidden layer activations correspond to the state variables ('memory cells') scaled by the activations of the 'output gates' $\mathbf{o}_t^{(n)}$,

$$\begin{aligned}\mathbf{h}_t^{(n)} &= \mathbf{o}_t^{(n)} \otimes \tanh(\mathbf{c}_t^{(n)}),\\ \mathbf{c}_t^{(n)} &= \mathbf{f}_t^{(n)} \otimes \mathbf{c}_{t-1}^{(n)} + \mathbf{i}_t^{(n)} \otimes \tanh\left(\mathbf{W}^{(n-1),(n)}\mathbf{h}_t^{(n-1)} + \mathbf{W}^{(n),(n)}\mathbf{h}_{t-1}^{(n)} + \mathbf{b}_c^{(n)}\right), \qquad (1)\end{aligned}$$

where $\otimes$ denotes element-wise multiplication and tanh is also applied element-wise. Thus, the state is scaled by a 'forget' gate (Gers et al., 2000) with dynamic activation $\mathbf{f}_t^{(n)}$ instead of a recurrent connection with static weight. $\mathbf{i}_t^{(n)}$ is the activation of the input gate that regulates the 'influx' from the feedforward and recurrent connections. The activations of the input, output and forget gates are calculated in a similar fashion as $\mathbf{c}_t^{(n)}$ (Graves et al., 2013). From the dependencies between layers ($n-1 \rightsquigarrow n$) and time steps ($t-1 \rightsquigarrow t$) in the above, it is obvious that parallel computation of feedforward activations cannot be performed across layers; further, parallel computation of recurrent activations is not possible across time steps. Thus, to increase the degree of parallelization, we consider *data fractions* (cf. Figure 1) of size $P$ out of $S$ sequences in parallel, each having exactly $T$ time steps (creating 'dummy' time steps for shorter sequences which are neglected in the error calculation). For instance, we consider a state matrix $\mathbf{C}^{(n)}$ for the $n$-th layer,

$$\mathbf{C}^{(n)} = [\mathbf{c}_{1,p}^{(n)} \cdots \mathbf{c}_{1,p+P-1}^{(n)} \cdots \mathbf{c}_{T,p}^{(n)} \cdots \mathbf{c}_{T,p+P-1}^{(n)}], \qquad (2)$$

where $\mathbf{c}_{t,p}^{(n)}$ is the state for sequence $p$ in layer $n$ at time $t$. To realize the update equation (1) we can now compute the feedforward part for all time steps and $P$ sequences in parallel simply by pre-multiplication with $\mathbf{W}^{(n-1),(n)}$. For the recurrent part, we can update $\mathbf{C}^{(n)}$ from 'left to right' using $\mathbf{W}^{(n),(n)}$. Input, output and forget gate activations are calculated in an analogous fashion. In this process, the matrix structure (2) ensures memory locality of the data corresponding to one time step (matrices are stored in column-major order). For bidirectional layers the above matrix structure is replicated at each layer; in the 'backward'

| | RNNLIB (Graves, 2013) | CURRENNT | | | |
|---|---|---|---|---|---|
| Parallel sequences ($P$) | 1 | 1 | 10 | 50 | 200 |
| Validation set error (10 ep.) | 0.138 | 0.138 | 0.135 | 0.137 | 0.144 |
| Validation set error (50 ep.) | 0.120 | 0.119 | 0.116 | 0.118 | 0.119 |
| Training time / epoch [s] | 7 420 | 3 805 | 580 | 392 | 334 |
| Speedup | (1.0) | 2.0 | 12.8 | 18.9 | 22.2 |

Table 1: Performance (error / speedup) on CHiME 2013 noisy speech recognition task.

part, the recurrent parts are updated from 'right to left', and activations are collected in a single vector before passing them to the subsequent layer (Graves et al., 2013).

During network training, the *backward pass* for the hidden layers is realized similarly, by splitting the matrix of weight changes into a part propagated to the preceding layer and a recurrent part propagated to the previous time step, resulting in a parallel implementation of the backpropagation through time (BPTT) algorithm. The weight changes are applied for all sequences (batch learning) or for each data fraction. Thus, if $1 < P < S$ we perform mini-batch learning. In this case, only $P$ sequences have to be kept in memory at once, allowing for learning from large data sets.

## 4. Benchmark

We conclude with a benchmark on a word recognition task with convolutive non-stationary noise from the 2013 2nd CHiME Challenge's track 1 (Vincent et al., 2013), where bidirectional LSTM decoding has been shown to deliver best performance (Geiger et al., 2013). We consider frame-wise word error rate as well as computation speedup in training with respect to the open source C++ reference implementation by Graves (2013) running in a single CPU thread on an Intel Core2Quad PC with 4 GB of RAM. The GPU is an NVIDIA GTX 560 with 2 GB of RAM. We compare results for different values of $P$ while fixing the other training parameters. The corresponding NetCDF, network configuration, and training parameter files are distributed with CURRENNT. Results (Figure 1) show that the error rate after 50 epochs is not heavily influenced by the batch size for parallel processing, while speedups of up to 22.2 can be achieved.

## 5. Conclusions

CURRENNT, our GPU implementation of deep LSTM-RNN for labeling sequential data, has been shown to deliver double digit training speedups at equal accuracy in a noisy speech recognition task. Future work will be concentrated on discriminative training objectives and cost functions for transcription tasks (Graves, 2008).

## Acknowledgments

This research has been supported by DFG grant SCHU 2502/4-1. The authors would like to thank Alex Graves for helpful discussions.

## References

M. Cernanský. Training recurrent neural network using multistream extended Kalman filter on multicore processor and CUDA enabled graphic processor unit. In *Proc. of ICANN*, volume 1, pages 381–390, 2009.

D. Eck and J. Schmidhuber. Learning the long-term structure of the blues. In *Proc. of ICANN*, pages 284–289, 2002.

J. T. Geiger, F. Weninger, A. Hurmalainen, J. F. Gemmeke, M. Wöllmer, B. Schuller, G. Rigoll, and T. Virtanen. The TUM+TUT+KUL approach to the CHiME Challenge 2013: Multi-stream ASR exploiting BLSTM networks and sparse NMF. In *Proc. 2nd CHiME Workshop*, pages 25–30, Vancouver, Canada, 2013.

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

A. Graves. *Supervised Sequence Labelling with Recurrent Neural Networks.* PhD thesis, Technische Universität München, 2008.

A. Graves. RNNLIB: A recurrent neural network library for sequence learning problems. `http://sourceforge.net/projects/rnnl/`, 2013.

A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Proc. of ICASSP*, pages 6645–6649, Vancouver, Canada, 2013.

G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio. How to construct deep recurrent neural networks. In *Proc. of ICLR*, 2014.

T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 11:743–746, 2010.

M. Sundermeyer, R. Schlüter, and H. Ney. LSTM neural networks for language modeling. In *Proc. of INTERSPEECH*, Portland, OR, USA, 2012.

E. Vincent, J. Barker, S. Watanabe, J. Le Roux, F. Nesta, and M. Matassoni. The second 'CHiME' speech separation and recognition challenge: Datasets, tasks and baselines. In *Proc. of ICASSP*, pages 126–130, Vancouver, Canada, 2013.

M. Wöllmer, M. Kaiser, F. Eyben, F. Weninger, B. Schuller, and G. Rigoll. Fully automatic audiovisual emotion recognition – voice, words, and the face. In T. Fingscheidt and W. Kellermann, editors, *Proceedings of Speech Communication; 10. ITG Symposium*, pages 1–4, Braunschweig, Germany, 2012.