An FPGA Implementation of a Long Short-Term Memory Neural Network

Jose Pedro Castro Fonseca
Faculty of Engineering University of Porto
Porto, Portugal

(Working Version)

***Abstract*—This work proposes a hardware architecture for a Long Short-Term Memory (LSTM) Neural Network, aiming to outperform software implementations, by exploiting the inherent parallelism in it. The main design decisions are presented, along with the proposed network architecture. A description of the main building blocks of the network is also presented. The network is synthesized for various sizes, and performance results are presented and analysed. The synthesized network achieves a 195 times speed-up over a custom-built software network, proving the benefits of parallel computation for this kind of network.**

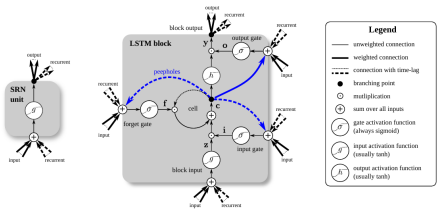

Fig. 1. A complete LSTM neuron, with all the features as described in [5]. Source: [6]

I. Introduction

NEURAL Networks are one of the most commonly used techniques in Deep Learning. This particular type of network, LSTM, is a recursive network, given that the neuron outputs in a certain time step are also fed as the input in the next time step, and that way their memory elements can make sense of patterns within data sequences, unlike classical recursive neural networks. These algorithms have been profusely implemented in software, and their practical applications are plentiful. However, the benefits of the inherent parallelism offered by a dedicated hardware platform are not availed, and there are relatively few implementations of Machine Learning algorithms in these kind of platforms.

Hardware platforms would achieve a considerable speed-up over software implementations, which could prove useful for high data throughput systems, where the calculation overhead is critical and limits the performance of the systems. Furthermore, a hardware implementation can even benefit *offline* Deep Learning tasks by providing the results of a given experiment faster than in a regular CPU, yielding an increase in scientific productivity.

Hitherto, there is only one implementation [1], but its performance is undermined by the external memory access. This implementation aims to make use of internal FPGA memory resources, and therefore achieve a higher speed performance.

In overview, this article begins by explaining what are LSTM networks – and their mathematical details – in Section II, followed by providing a quick overview of their state of the art in Section III. Section IV outlines the proposed hardware architecture and its main constituent modules, and the performance and synthesis results are reported in Section V. A final concluding remark is given in Section VI.

II. LSTM Neural Networks

LSTM Networks were originally formulated in [2], but its formulation has been incrementally updated in [3] and [4], and the most current version is the one in [5]. One of the inital proposers of LSTM, Prof. Jrgen Schmidhber, did a survey on the most common variations of the model last year [6], which is the reference for this short discussion.

A single LSTM neuron is presented in Figure 1. As it can be seen in the picture, we still have the recurrent connections from the regular RNNs, but now there are multiple entry points that control the flow of information through the network. Although omitted from the picture, all the gates are biased, as is suggested in Equations 1. The main components, their role and relevance, are explained as follows.

- **Input Gate** – this is the input gate, where the relative importance of each feature of the input vector at time t, $\mathbf{x}^{(t)}$, and the output vector at the previous time step, $\mathbf{y}^{(t-1)}$, are weighed in, producing an output $\mathbf{i}^{(t)}$.
- **Block Input Gate** – as the name implies, this gate controls the flow of information from the input gate to the memory cell. It also receives the input vector and the previous output set, producing an output $\mathbf{z}^{(t)}$. The **activation function of this gate can be either**, but the most common choice is the **Hyperbolic Tangent**.
- **Forget Gate** – its role is to control the contents of the Memory Cell, either to set or reset them, using the *Hadamard Elementwise* vector multiplication of its output at time t, $\mathbf{c}^{(t)}$, with the contents of the memory unit at the previous time step, $\mathbf{c}^{(t-1)}$. The activation function of this gate is **always sigmoid**, and the resulting signal is $\mathbf{f}^{(t)}$.
- **Output Block Gate** – this gate has a role very similar to that of the Block Input Gate, but now it controls

the information flow *out* of the LSTM neuron, namely the activated Memory Cell output. The control signal it produces is $\mathbf{o}^{(t)}$.

- **Memory Cell** – the cornerstone of the LSTM neuron. This is the memory element of the neuron, where the previous state is kept, and updated accordingly to the dynamics of the gates that connect to it. Also, this is where the peephole connections come from.
- **Output Activation** – the output of the Memory Cell goes through this activation function that, as the gate activation function, can be any, but the *hyperbolic tangent* is the most common choice.
- **Peepholes** – direct connections *from* the memory cell that allow for gates to 'peep' at the states of the memory cell. They were added after the initial 1997 formulation, and their absence was proven to have a minimal performance impact [6]. For this reason, they were ommitted in this architecture.

The operation of each set of gates of the layer is given by the following set of equations

$$\begin{aligned}
\mathbf{z}^{(t)} &= g(\mathbf{W}_z\mathbf{x}^{(t)} + \mathbf{R}_z\mathbf{y}^{(t-1)} + \mathbf{b}_z) \\
\mathbf{i}^{(t)} &= \sigma(\mathbf{W}_i\mathbf{x}^{(t)} + \mathbf{R}_i\mathbf{y}^{(t-1)} + \mathbf{p}_i \odot \mathbf{c}^{(t-1)} + \mathbf{b}_i) \\
\mathbf{f}^{(t)} &= \sigma(\mathbf{W}_f\mathbf{x}^{(t)} + \mathbf{R}_f\mathbf{y}^{(t-1)} + \mathbf{p}_f \odot \mathbf{c}^{(t-1)} + \mathbf{b}_f) \\
\mathbf{o}^{(t)} &= \sigma(\mathbf{W}_o\mathbf{x}^{(t)} + \mathbf{R}_o\mathbf{y}^{(t-1)} + \mathbf{p}_o \odot \mathbf{c}^{(t)} + \mathbf{b}_o) \\
\mathbf{c}^{(t)} &= \mathbf{i}^{(t)} \odot \mathbf{z}^{(t)} + \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} \\
\mathbf{y}^{(t)} &= \mathbf{o}^{(t)} \odot h(\mathbf{z}^{(t)})
\end{aligned} \tag{1}$$

where $\odot$ is the Hadamard multiplication. The i-th element of the previous vectors in bold corresponds to the value of the i-th neuron in the layer, which is a very convenient and compact representation of the whole layer. Furthermore, if the layer has N LSTM neurons and M inputs (i.e. the size of the layer that precedes this), we see that the input weight matrices $\mathbf{W}_*$ have size $N \times M$, and the recurrent weight matrices $\mathbf{R}_*$ are squre matrices of size $N \times N$, and that the bias weight matrices $\mathbf{b}_*$ and the matrices $\mathbf{y}^{(t)}$ through $\mathbf{c}^{(t)}$ have size $N \times 1$, and are thus, in fact, vectors.

III. Related Work

LSTM Networks are nowadays one of the state of the art algorithms in deep-learning, and their performance is superior to that of other kinds of RNNs and Hidden Markov Models, both of which are generally used to address the same set of problems where LSTM are employed, namely predicting and producing classification decisions from time-series data. A very comprehensive description of applications can be found in one of the initial authors webpage dedicated to the subject [1].

The use of LSTM Networks are plentiful: in **Handwriting Recognition** [7], where they surpass HMM-based models in optical character recognition [8]; in **Speech Recognition**, where, for instance, Graves et al. [9], in 2013, set a new record on the TIMT Phoneme Recognition Benchmark, and also in **Large Scale Acoustic Modelling of Speech** [10]. Other uses include **Handwriting Synthesis** [11], **Translation** [12], **Biomedical Applications** such as protein homology detection [13], **Music Analysis and Composition**, such as the transcription of piano music to MIDI [14] and automated composition [15] and improvisation [16], and lastly **Video and Image Analysis** as in [17], [18] and [19].

Hitherto, there is but one actual implementation of an LSTM network in hardware, published recently (March 2016) by Chang et al. [1]. A 2 layer LSTM network (two layers in series) was implemented, with 128 neurons each, which, for processing 1000 samples, yielded an execution time of $0.932\,\mathrm{s}$. Assuming that both layers are equal, and thus have the same execution time, this yields an approximate execution time of $29.13\,\mu\mathrm{s}$ per incoming sample (dividing the total execution time by 2×1000). Therefore, this network is able to perform around 34.3 thousands of forward propagations per second. Although the network is large, its performance is well below this work, because it does not have a full level of parallelism when compared to the proposed design of Section IV, and it makes use of external memory to store the weights. This last point is understandable, since the authors use the FPGA core as a co-processor for the main CPU.

IV. Proposed Architecture

Before presenting the

A. Activation Function Calculation

In order to evaluate the transcendental activation functions $\sigma(x)$ and $\tanh(x)$, **Polynomial Approximations** were used, as detailed in [20], since evaluating a polynomial does not have high memory usage needs (as opposed to Table Methods, for instance) and, if the polynomial degree is sufficiently low, the number of multiplications needed is low enough to not pose a restriction both on resources (now DSP slices, and not memory) and in speed (number of number of clock cycles needed to output a result).

The error minimization strategy used to find the optimal polynomial was the **Least Maximum Approximation**, where the *maximum* error is *minimized*, making use of Remez's Algorithm, which sets a system of $n+2$ linear equations such as 2

$$p(x_i) - f(x_i) = (-1)^{n+1}\epsilon \Leftrightarrow p_0 + p_1 x_i + p_2 x_i^2 + \cdots + p_n x_i^n - f(x_i) = (-1)^r \tag{2}$$

where $i \in [0, n+1]$. Of course, both functions have horizontal assymptotes, which can be used as the value in the edges, but instead of performing the optimization in the single interval in between the chosen points from where the module evaluates the function as the value of the assymptote, I have further split the interval in 4, in order to have lower degree polynomials – simpler to evaluate – at a reduced error cost. The algorithm was run using Python, and targeted second degree polynomials for each interval. After that, the Verilog model was described, where the correct coefficients are loaded according to the interval where the incoming operand is located. The evaluation of the polynomial was accomplished using the **Horner's Rule**, noting that

$$p(x) = p_0 + p_1 x + p_2 x^2 = p_0 + x(p_1 + x p_2) \tag{3}$$

[1] http://people.idsia.ch/ juergen/rnn.html

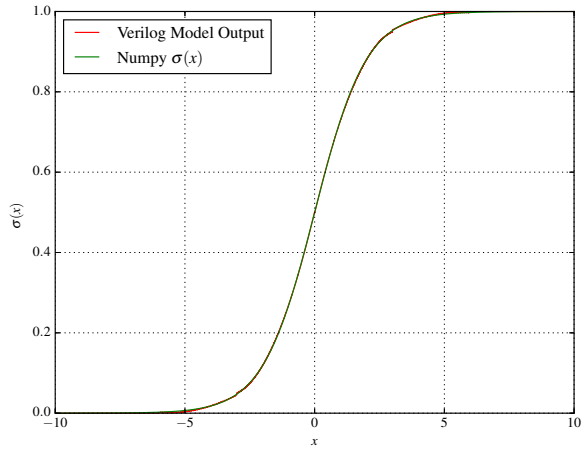

Fig. 2. Plot of the output of the Sigmoid Calculator HDL module

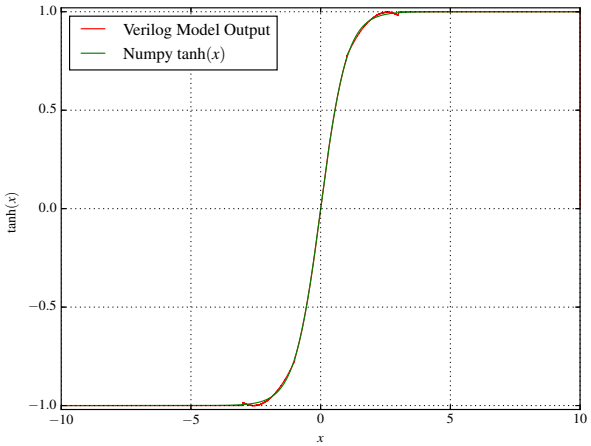

Fig. 3. Plot of the output of the Hyperbolic Tangent HDL module

and, therefore, the calculation is simply the procedure of multiplying the operand by a value and adding a constant to the result, repeated two times. Again the internal machine state controls which values are multiplied and added according to the pipeline state. The output of the Sigmoid HDL block and the tanh HDL block are presented in Figures 2 and 3, respectively. Due to the internal datapath pipeline, the module takes 5 clock cycles to output a result.

B. Matrix-vector Dot Product Calculation

From Equations 1, we see that the weight matrices $\mathbf{W}_*$ and $\mathbf{R}_*$ are multiplied by the input vector $\mathbf{x}$ and the layer output vector $\mathbf{y}$, respectively. This way, this block implements matrix-vector multiplication to perform those calculations, and is parameterized in order to accommodate networks of various sizes, since $\mathbf{W}_*$ has size $N \times M$, and $\mathbf{R}_*$ has size $N \times N$ – x has length M (the number of inputs to the layer), while y has length N (the number of neurons in the layer). This way, if a layer with different parameters is needed, either in terms of number of inputs or number of neurons, we only need to change the respective parameter before the synthesis stage, instead of having to redesign the whole block for that particular size.

The matrix-vector dot product of a matrix A of size $N \times M$ by a vector x of size M, if performed in a linear non-parallel way, can be described in terms of Algorithm 1

Algorithm 1 Matrix-vector multiplication of a matrix

```
for i = 1 : N do
    for j = 1 : M do
        y_i := y_i + A_ij · x_j
    end for
end for
```

This operation has a computational complexity of $O(n^2)$. Each of the i-th components of the output vector y can be calculated **in parallel**, each only requiring the corresponding i-th line from the matrix. Follow this approach, matrix-vector multiplication can now be performed in **linear time**, which is one of the great advantages of custom-tailored hardware solutions.

Although this solution only requires one multiplication per row of the input matrix (i.e. N multiplications), if the row size is large, we may run out of resources in the FPGA; therefore, some sort of *resource multiplexing* strategy must be used to ensure the flexibility of the solution to accomodate networks of larger dimensions. The solution found for this design was to *share* the multiplication slice among the rows of the matrix: in a direct implementation of Algorithm 1, each multiplication slice was responsible for producing the i-th element of the output vector y (of size N), therefore the final result for the vector would be ready in M clock cycles (i.e. the number of columns); now, defining a parameter $K_G = \frac{\text{Number of rows}}{\text{Number of multipliers}}$ – the number of rows that share the same multiplier – the same multiplier is responsible for producing several i-th elements of the output vector, in consecutive time slots of M clock cycles. For instance, in an 8×2 matrix scenario, with $K_G = 2$, we would have 4 multipliers, and the output vector elements y_0, y_2, y_4 and y_6 would be ready after $M = 2$ clock cycles, and the remaining – y_1, y_3, y_5 and y_7 – are ready after another two clock cycles, that is $2M = 4$ clock cycles after the calculations began.

Figures-4 and 5 depict a diagram of the memory access for the Matrix, and the row multiplication units within the module, respectively, where I have set $K_G = 4$, for the same matrix and vector sizes as before. Note that in this situation, we would only have 2 multipliers, and the module would be composed of two multiplication units, such as those in Figure 5, that work in parallel. They address a particular column using the signal `colSel`, which is used by the RAM module to output the corresponding column of the matrix (in regard to the input vector, obviously this signal selects only a single position), depicted in Figure 4. The dark shaded part of the memory is used by the first multiplier, and the light shaded is used by the other, in parallel for a fixed `rowMux` – this signal is produced by the control unit of the module, and essentially operates the left multiplexer and right demultiplexer of Figure 5, that allows to choose the proper position of the weight column and to write

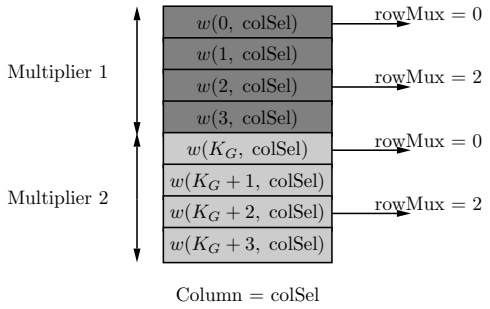

Fig. 4. A column of the matrix that serves as input to the module. The dark shaded part is for the first multiplier, and the light shaded is for the other, in parallel. The rowMux signal addresses the position within each shaded area

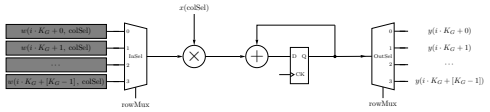

Fig. 5. The i-th row multiplication unit of the Module, where rowMux and colSel are internal signals produced by the control unit of the module. The flip-flop accumulates the sum, and the output demux selects the appropriate memory position on where to store this value, within the slot attributed to this multiplication, from $i \cdot K_G$ to $i \cdot K_G + [K_G - 1]$

to the correct output register, respectively. In this example, for `rowMux`=0, the control unit increments `colSel` from 0 to M, and thus evaluating y_0 and y_4. After this, `rowMux` is incremented to 1, and the process repeats until `rowMux` reaches $K_G - 1$. Therefore, we have the correct result vector in a time proportional to $K_G \cdot M$: a 3 clock cycle overhead is added to the previous estimate, since the memory only outputs the appropriate column in the *next clock cycle*, and that this module is pipelined both at the input and at the output in order to increase the maximum clock frequency.

C. Weight storage

The weights are stored in LUTRAM, and for that purpose, they are declared as a matrix of registers, where the access, both in terms of write and read operations, is made to each column. The write/read access can be performed at the same time, since the memory has separate communication ports for input/output, provided that the **addresses do not coincide**.

D. Gate Module

The Gate modules are responsible for producing the internal signal vectors for $\mathbf{z}^{(t)}$, $\mathbf{i}^{(t)}$, $\mathbf{f}^{(t)}$ and $\mathbf{o}^{(t)}$. If we note that, according to [6], the removal of peepholes (the signals $\mathbf{p}_*$) does not compromise significantly the performance of the network **??**, we can ommit them in order to simplify the Gate Module and reduce the usage of DSP slices. This way, a Gate module needs to perform three tasks

1) Multiply matrix $\mathbf{W}_*$ by the input vector $\mathbf{x}^{(t)}$
2) Multiply matrix $\mathbf{R}_*$ by the previous layer output vector $\mathbf{y}^{(t-1)}$
3) Sum the bias vector $\mathbf{b}_*$ to the remaining matrix-vector dot product results.

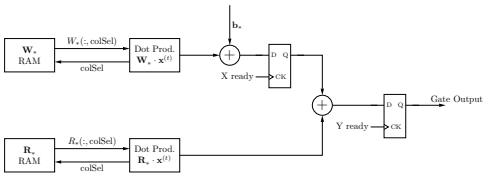

Fig. 6. Diagram of the hardware block that implements the Gate

Assuming that the network size N is always larger than the input size M, if we use the matrix-vector dot product units of Section IV-B, the multiplication in task 1 takes approximately $K_G \cdot M$ cycles and the one in task 2 takes $K_G \cdot N$ cycles. This way, task 2 and task 1 can be performed in parallel, and we can use the extra time that task 2 takes, relative to task 1, to perform task 3, and sum the bias vector to the output of task 1, whose result is ready by that time. The module is triggered by a `beginCalc` input signal that activates the internal state machine, and outputs a `dataReady` signal that informs the network that the calculations have been concluded, and the value at the output is the actual final result. After validating and simulating this block, and taking into account the internal state-machine and that the internal datapath is pipelined, the exact number of clock cycles this module takes to produce an output is $6 + K_G \cdot N$.

E. Network Architecture

Equations 1 suggest that the signals $\mathbf{z}^{(t)}$, $\mathbf{i}^{(t)}$, $\mathbf{f}^{(t)}$ and $\mathbf{o}^{(t)}$ do not depend on each other – they operate only on the current input vector $\mathbf{x}^{(t)}$ and the previous layer output $\mathbf{y}^{(t-1)}$ – and therefore can be calculated in parallel. This way we need four Gate Modules working in parallel, each one with its respective weigth RAMs for $\mathbf{W}_*$ and $\mathbf{R}_*$, and followed by the respective activation function calculator (detailed in Section **??**). There are three elementwise multiplications, two for producing signal $\mathbf{c}^{(t)}$ (which can be done in parallel and then summed elementwise) and one for $\mathbf{y}^{(t)}$ (which can be done only after applying the activation function $\mathbf{c}^{(t)}$).

Furthermore, we can avoid a nave translation of the Equations 1, which would replicate unnecessary resources (such as elementwise multipliers and activation function calculatores) and require a higher area usage to save a negligible amount of clock cycles, by noting that one of the operands is the output of a $\tanh(\mathbf{x})$ block and the other of a $\sigma(\mathbf{x})$, and they are then multiplied (elementwise, of course) together. This way, instead of replicating these $\tanh$-σ-$(\cdot)$wise structures, we use a *single one* and choose the operand accordingly to the state that the network is currently in. The issue about the elementwise multiplier for $\mathbf{c}^{(t)}$, which does not use the $\tanh$ activation function, can be solved by adding another multiplexer that chooses between the output of the $\tanh(\mathbf{x})$ module or the signal $\mathbf{c}^{(t-1)}$. These ideas resulted in the LSTM network design of Figure 7, which is mathematically equivalent to Equations 1.

The two left multiplexers control the operands that are fed to the activation function modules, and the selecting signal is generated by the network's state machine, and its value is incremented after each complete usage of the $\tanh$-σ-$(\cdot)$wise

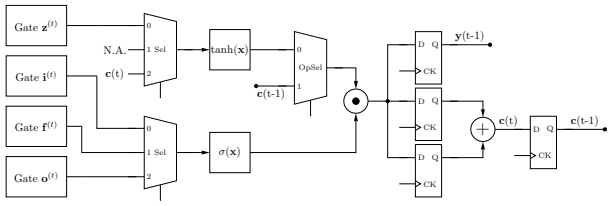

Fig. 7. Block diagram of the hardware LSTM Neural Network

structure: this is where the time multiplexing of the structure takes place. Since in state $\texttt{Sel} = 1$ the left operand of the elementwise multiplier (the one that preceded the flip-flops in the previous design) is the signal $\mathbf{c}^{(t-1)}$, another multiplexer was added before the elementwise multiplication, to select the $\mathbf{c}^{(t-1)}$ signal in that particular case, and the output from the $\tanh(\mathbf{x})$ block, otherwise.

The flip-flops on the right hand side of Figure 7 are activated by signals generated within the network's state machine that enable the appropriate flip-flop, placing the output from the elementwise multiplicator in the correct place. The first activated flip-flop is the middle one, which keeps the result from the elementwise mutiplication of the $\mathbf{z}^{(t)}$ and $\mathbf{i}^{(t)}$ vectors, then, after a full operation of the $\tanh$-σ-$(\cdot)$wise structure, the bottom flip-flop saves the other portion of the sum that evaluates to the $\mathbf{c}^{(t)}$ signal. Lastly, the top flip-flop saves the network output $\mathbf{y}^{(t)}$, which in the next incoming sample becomes $\mathbf{y}^{(t-1)}$ and is used by the Gate modules in this next batch of calculations.

Now, since there is only a single elementwise multiplier and only two activation function calculators, the total requirement for DSP slices is simply

$$4\frac{2N}{K_G} + 2N + N = N\left(\frac{8}{K_G} + 3\right). \tag{4}$$

where we see that we saved $5N$ multipliers, which for a large value of N can have a decisive impact. In terms of speed performance, although the Gate calculation time remains the same, now the $\tanh$-σ-$(\cdot)$wise structure runs for 3 consecutive times, in a non parallel fashion. After adjustments to the state machine, and accounting for pipelining and synchronization within the datapath, the clock cycles needed after the gate module calculations are 27, so the total clock cycles needed are

$$(N \cdot K_G + 6) + 27 = 33 + N \cdot K_G \tag{5}$$

which is only 13 clock cycles more than a fully-parallel, nave architecture. For instance, an $N = 32$ neuron network would require 320 DSP slices, while this new architecture would only require 160, only at the expense of 13 more clock cycles.

V. Results

A. Validation

B. Synthesis

C. Performance

VI. Conclusion

References

[1] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on FPGA," *CoRR*, vol. abs/1511.05552, 2015. [Online]. Available: http://arxiv.org/abs/1511.05552

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735 – 80, 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[3] F. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with lstm," *Neural Computation*, vol. 12, no. 10, pp. 2451 – 71, 2000. [Online]. Available: http://dx.doi.org/10.1162/089976600300015015

[4] F. Gers and J. Schmidhuber, "Recurrent nets that time and count," *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, vol. vol.3, pp. 189 – 94, 2000. [Online]. Available: http://dx.doi.org/10.1109/IJCNN.2000.861302

[5] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural Networks*, pp. 5–6, 2005.

[6] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *CoRR*, vol. abs/1503.04069, 2015. [Online]. Available: http://arxiv.org/abs/1503.04069

[7] E. Grosicki and H. El Abed, "Icdar 2009 handwriting recognition competition," July 2009, pp. 1398–1402.

[8] T. M. Breuel, A. Ul-Hasan, M. A. Al-Azawi, and F. Shafait, "High-performance ocr for printed english and fraktur using lstm networks," in *Proceedings of the 2013 12th International Conference on Document Analysis and Recognition*, ser. ICDAR '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 683–687. [Online]. Available: http://dx.doi.org/10.1109/ICDAR.2013.140

[9] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2013, pp. 6645–6649.

[10] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," pp. 338–342, 2014. [Online]. Available: http://www.isca-speech.org/archive/interspeech$_2$014/i14$_0$338.$html$

[11] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850

[12] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," vol. 4, no. January, Montreal, QC, Canada, 2014, pp. 3104 – 3112.

[13] S. Hochreiter, M. Heusel, and K. Obermayer, "Fast model-based protein homology detection without alignment," *BIOINFORMATICS*, vol. 23, no. 14, pp. 1728–1736, JUL 15 2007.

[14] S. Bock and M. Schedl, "Polyphonic piano note transcription with recurrent neural networks," *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pp. 121 – 124, 2012. [Online]. Available: http://dx.doi.org/10.1109/ICASSP.2012.6287832

[15] A. E. Coca, D. C. Correa, and L. Zhao, "Computer-aided music composition with lstm neural network and chaotic inspiration," Dallas, TX, United states, 2013. [Online]. Available: http://dx.doi.org/10.1109/IJCNN.2013.6706747

[16] D. Eck and J. Schmidhuber, "Finding temporal structure in music: Blues improvisation with LSTM recurrent networks," *Neural Networks for Signal Processing - Proceedings of the IEEE Workshop*, vol. 2002-January, pp. 747 – 756, 2002. [Online]. Available: http://dx.doi.org/10.1109/NNSP.2002.1030094

[17] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," *CoRR*, vol. abs/1411.4555, 2014. [Online]. Available: http://arxiv.org/abs/1411.4555

[18] S. Venugopalan, H. Xu, J. Donahue, M. Rohrbach, R. J. Mooney, and K. Saenko, "Translating videos to natural language using deep recurrent neural networks," *CoRR*, vol. abs/1412.4729, 2014. [Online]. Available: http://arxiv.org/abs/1412.4729

[19] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," *CoRR*, vol. abs/1411.4389, 2014. [Online]. Available: http://arxiv.org/abs/1411.4389

[20] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*, 2nd ed. Birkhauser, 2005.