Spoken Language Understanding Using Long Short-Term Memory Neural Networks

E4040.2016Fall.LCFR.report
Sanjar Ahmadov sa3312, Tianhe Shen ts2957, Matthew Petersen mfp2124
Columbia University

December 19, 2016

Abstract

We follow Yao et al. (2015) in their construction of various LSTM extensions. These extensions include a moving average regression over past predictions, a deep two-layer LSTM, and dropping different LSTM gates. We reproduce their results, and then construct a deep attention LSTM, which outperforms all other models, achieving an F1 error of 95.09% on the ATIS-3 dataset without training on named-entity feature. Secondly, we construct a deep RNN for comparison of the depth benefits of the deep LSTM. Lastly, we construct a bidirectional attention LSTM, which in half the iterations had achieved comparable results with other models - F1 of 94.35% after 15 iterations. We plan to fully train bidirectional LSTM by the end of December 2016, and also apply dropout. Future experiments could try combinations of the extensions discussed in this paper, such as a moving average regression on top of a bidirectional attention LSTM, or depth-gated LSTM (Yao et al. 2015).

1 Introduction

Recurrent neural networks (RNNs) are a family of neural networks designed to process sequential data. There are two defining features of RNNs: feedback loops and shared parameters. Feedback loops allow the present value of a neuron to depend on its past value at some point, acting like memory. Parameter sharing is similar to its use in convolutional networks, and imposes a prior that the interesting characteristics of a sequence should be invariant to certain transformations of that sequence. Using recurrence, we can concisely write the hidden state of an RNN at time step t as

$$\begin{aligned}\boldsymbol{h}^{(t)} &= g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, x^{(2)}, x^{(1)}) \\ &= f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}).\end{aligned}$$

By adding output layers that read from $\boldsymbol{h}$, we can make predictions at various time steps, forcing $\boldsymbol{h}^{(t)}$ to learn a simplified representation of many aspects of the past.

A long-short term memory (LSTM) neural network is a type of RNN that has been shown to be capable of learning long-term dependencies. It was invented by Hochreiter et al. (1997), who showed that it can learn dependencies across 1000 time steps or more. An LSTM network allows some information to flow across time largely unimpeded, by adding *cell state* nodes. Information is added or deducted from the cell state at each step according to the activity of gating neurons. In contrast to LSTM's, RNN's highly prefer nearby context. For an analysis of why RNN's fail to exploit long-term information, see Bengio, Simard, and Frasconi (1994). The following description of LSTM architecture is credited to (Olah 2015).

The task our LSTM solves is word labeling. Given an air-travel related sentence, it outputs a label for each word. To solve this task, we closely follow the paper by Yao et al. (2015). We then extend their results by implementing an attention mechanism within a deep two-layer LSTM, and achieve a higher test accuracy than all other models. We also implement a bidirectional-attention-LSTM, which trains, but requires one hour per iteration on an Nvidia K520 instance of Amazon cloud computing services. After just 15 iterations it had reached an F1 of 94.35%, and we plan to fully train it by the end of December 2016.

2 LSTM Overview

2.1 Memory cells

To allow long-term information storage, LSTM networks add a *memory cell* with linear activation function. The linear activation allows the gradient to flow freely across memory cells. This is due to the fact that gradient-based optimization always multiplies the error at a layer by the derivative of that layer's activation function. Alternative activation functions, such as sigmoid and hyperbolic tangent, have derivatives that decrease with absolute input size. Such a feature scales down the gradient at every single time step, diluting long-term information about how to update long-term parameters.

Memory cells with linear activation are a great tool for communicating long-term information, but we need a way to carefully decide which information is added and deducted from the memory cell across time steps. For that purpose, we introduce *gates*. A gate applies a sigmoid activation function to its input before point-wise multiplying in a way that updates the memory cell's carrying state. There are three types of gates: forget gates, input gates, and output gates.

2.2 Forget gates

A forget gate layer learns which information to *discard* from the memory cell's prior state. A forget gate layer accepts two inputs: the hidden layer of the past time step, h_{t-1}, and the input layer of the current time step, x_t. It then applies a sigmoid activation function to restrict its activation to the range $(0, 1)$. The closer the forget gate's activation is to zero for a given element, the more information we discard from the corresponding element of the memory cell. The forget gate layer's activation can be written as

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f).$$

2.3 Input gates

An input gate layer learns what information to *add* to the memory cell's prior state. An input gate layer accepts the same inputs as the forget gate layer, namely, the hidden layer of the past time step, h_{t-1}, and the input layer of the current time step, x_t. It then applies a sigmoid activation function to restrict its activation to the range $(0, 1)$. The closer the input gate's activation tensor is to zero for a given element, the less we add to the corresponding element of the memory cell. The input gate *chooses* which elements of the memory cell to update. A separate computation determines the scale of the update. Together, we write this as

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \end{aligned}$$

2.4 Updating the memory cell

We use two operations to update the memory cell's activation per the information contained in the last time step's hidden layer activation, h_{t-1} and the current time step's input layer activation, x_t. Firstly, we matrix multiply the memory cell's prior activation, denoted C_{t-1}, by the forget gate's current activation, f_t. This destroys information in the memory cell. Secondly, we matrix multiply the input gate's current activation, i_t, by the memory cell's update scale, denoted $\tilde{C}$. The first determines which elements of C_{t-1} are updated, and the second determines the degree of the update. We can write the update to our memory cell's activation as

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t,$$

where $\odot$ denotes the element-wise product, as opposed to the matrix product.

2.5 Output gates

An output gate layer learns what information to *output* from the memory cell to the current time step's hidden layer, h_t. An output gate layer accepts the same inputs as the forget gate layer and input gate layer, namely the hidden layer of the past time step, h_{t-1}, and the input layer of the current time step, x_t. It then applies a sigmoid activation function to learn *which* elements of the memory cell to output. Lastly it applies a hyperbolic tangent to the current memory cell state, C_t, which determines the scale of the output. The product of these two determines the current hidden layer state, h_t. Here we write the computation of the output gate layer activation, and its use to compute the current hidden layer activation:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \odot \tanh(C_t)$$

where $\odot$ again represents the element-wise product.

3 Summary of Original Paper

3.1 Base LSTM

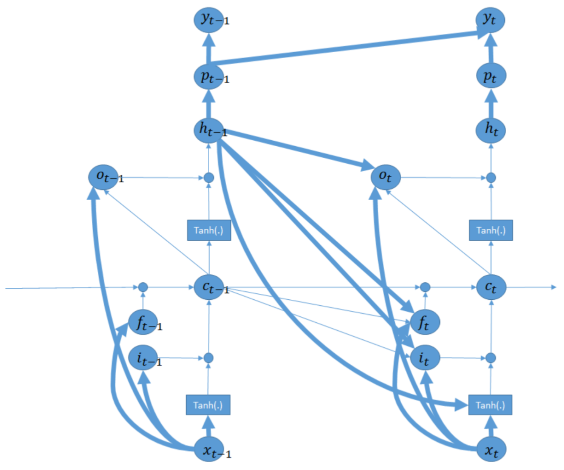
The authors Yao et al. (2015) first build a standard LSTM with linear activation applied to the cell state, c_t. The layer states at time step t, in order or computation, are as follows: i_t is the input gate, f_t is the forget gate, c_t is the cell state, o_t is the output gate, and h_t is the hidden state. Additionally, the LSTM uses peephole connections (Gers and Schmidhuber, 2000). Peephole connections just let the gate layers observe the cell state c_t and a past cell state c_{t-1}. They do so by adding a term $W \cdot c$ to the activity of the gate before the nonlinearity is applied. The formula for each layer of the LSTM is therefore as follows:

$$i_t = \sigma\big(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\big)$$
$$f_t = \sigma\big(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\big)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh\big(W_{xc}x_t + W_{hc}h_{t-1} + b_c\big)$$
$$o_t = \sigma\big(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\big)$$
$$h_t = o_t \odot \tanh(c_t)$$

The weight matrices from cell state to gates, i.e. W_{ci}, are diagonal, so that each element m of each gate only receives input from element m of the cell state. All other weight matrices are full. The biases for the gates, b_i, b_f, b_c, and b_o, are initialized to large positive values to encourage memorization of past states at the outset of training. The input to the LSTM at time t, denoted x_t, consists of the current input word and the next two words, for a context window of size 3. Additionally, the authors used 300 neurons in the hidden layer h_t, and a minibatch size of 30.

The authors extend this base LSTM in three non-overlapping ways. Firstly, they predict the current word label by computing a moving average regression over past M unnormalized label distributions. Secondly, they try adding a second identical LSTM on top of the first, forming a deep LSTM. Lastly, they examine the effect of dropping LSTM gates one at a time from the base model, by setting their value to one.

The block diagram that follows is directly from Yao et al. (2015). We take no creative credit.



3.2 Moving average regression

For natural language comprehension, it could further help to model some dependency between the output labels. Moving average regression models a simple linear dependency. At a basic level, it increases the probability of a particular label, based on the distribution of M prior labels. The prior labels are scaled by learned weights $W_{p,i}$ for $i = 1, 2, \ldots, M$, before being added together and having an additive bias, b_q, applied. Recalling that h_t is the activity of the hidden layer at time t, we can compute the predicted label, y_t, as follows:

$$p_t = W_{hp} h_t$$
$$q_t = \sum_{i=0}^{M} W_{pi} p_{t-i} + b_q$$
$$y_t = \text{softmax}(q_t)$$

Here, p_t is a matrix containing the unnormalized probabilities of each label for each observation. In our case, observations are words. So, each column of p_t represents a word, and as we go across rows we read its probability distribution over labels. $W_{h,p}$ is a learned weight matrix that transforms our hidden layer activation h_t to the dimension of our label set. We then compute the moving average of $p_t, p_{t-1}, p_{t-2}, \ldots, p_{t-M}$, the past M unnormalized label prediction matrices. Finally, we use softmax to convert our unnormalized moving average prediction matrix into a proper distribution over labels. For our experiment we choose $M = 3$, following Yao et al (2015).

3.3 Deep LSTM

The authors' deep LSTM consists of two LSTM's stacked sequentially. The lower LSTM has 200 hidden units, and the upper LSTM has 300 hidden units. The learning rate is 0.1 per sample, and they use minibatches of size 10. For each time step t, the hidden layer of the lower LSTM, denoted $h_{\text{lower},t}$, acts as the input layer of the upper LSTM, denoted $x_{\text{upper},t}$. For dimensionality matching, we must first multiply $h_{\text{lower},t}$ by a matrix whose number of columns is the desired dimension of $x_{\text{upper},t}$. This multiplication allows us to use a different number of hidden neurons for the lower and upper LSTM's.

3.4 LSTM simplification

To understand the importance of each component of the base LSTM, the authors drop one gate at a time by setting its value to one. The base LSTM contains three gate layers, i_t, f_t, and o_t, representing the input gate, forget gate, and output gate. A graph of the simplified LSTM's accuracy is given in the next section. For the simplified LSTM, 100 hidden neurons h_t are used, the minibatch size is 8, and the learning rate is 0.01.

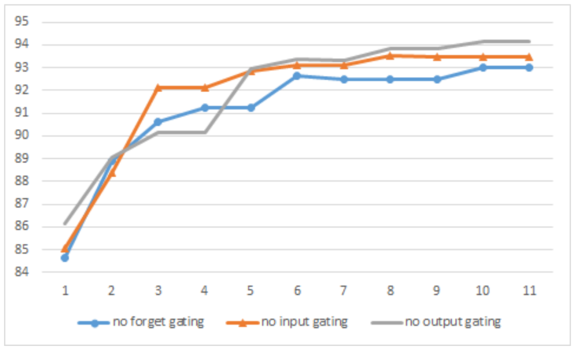
3.5 Key results of original paper

Yao et al. (2015) found that deep LSTM achieved the highest F1 score, 95.08%, and that moving average regression offers small but robust improvement in F1 score. For example, when only 100 hidden neurons are used (as opposed to 300), moving average LSTM widens its lead over standard LSTM from 0.07 percentage points to 0.24 percentage points.

Yao et al. results

CRF	RNN	CNN	LSTM	ma(3)	Deep
92.94	94.11	94.35	94.85	94.92	95.08

The next graph shows the test accuracy of the simplified LSTM over training. Each line represents the LSTM with one of its gates ignored (i.e. set to a value of one). Every simplified LSTM converges in 10 iterations, to a best F1 score of 94.14% for the case when the output gate is ignored.



4 Our Methodology

Our base LSTM model exactly mirrors that of Yao et al. (2015), but our implementation is subtly different. Firstly, we use full batch training for a batch size of 4,978, as opposed to minibatches of size 30. Secondly, since Yao et al. (2015) did not mention their choice of embedded dimension for word vectors, we used grid search for a standard LSTM and found 90 to be the optimal embedding dimension. This embedding dimension resulted in an increase from 93 to 94 percent test accuracy for the standard model, and we proceeded to use this embedding dimension for every model. The last difference between our implementation and that of Yao et al. (2015) is that we build our model in Theano as opposed to CNTK. Despite these variations, we closely replicate the authors' results for RNN, LSTM, LSTM with moving average regression, and deep LSTM. We then extend their analysis in three ways, implementing a deep RNN to compare against deep LSTM, including an attention mechanism applied to a deep LSTM, and including an attention mechanism applied to a bidirectional shallow LSTM.

4.1 Objectives and challenges

Our objective was to recreate all results of Yao et al. (2015), and to further compare against three new networks. The first is a deep RNN, which adds perspective to the performance improvement of deep LSTM. The second is an attention mechanism, applied to both shallow and deep LSTM's. Due to time constraint, we failed to recreate the author's convolutional neural network (CNN) and conditional random field (CRF). Furthermore, the attention mechanism applied to bidirectional shallow LSTM was taking prohibitively long per epoch to train (about one hour per iteration on an Nvidia K520 instance of Amazon cloud computing). Even after just 15 iterations, the bidirectional attention LSTM achieves an error rate 94.35%, and we fully expect it to be the winning model after full training, which will be completed by the end of December 2016.

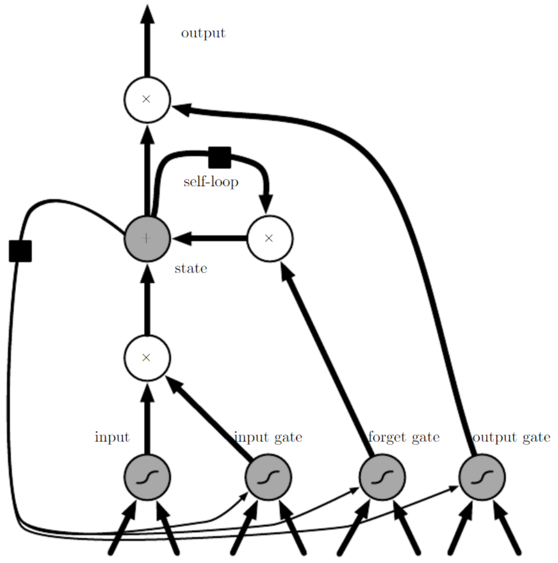
4.2 Problem formulation and design

We follow the base LSTM formulation of Yao et al. (2015). The gates of our LSTM are as follows: i_t for the input gate, f_t for the forget gate, c_t for the cell state, o_t for the output gate, and h_t for the hidden layer activation. Here, t is the time, but for our dataset it represents the word index in the sentence being evaluated. The model that follows is identical to that of section 3.1, and we note again that it includes peephole connections:

$$\begin{aligned}
i_t &= \sigma\big(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\big)\\
f_t &= \sigma\big(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\big)\\
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh\big(W_{xc}x_t + W_{hc}h_{t-1} + b_c\big)\\
o_t &= \sigma\big(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\big)\\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

Following the authors, we use diagonal cell-to-gate weight matrices, W_{ci}. Likewise, all other weight matrices are full, and the biases for the gates are initialized to large positive values. We also use a context window of size 3, a hidden dimension of 300. Unlike Yao et al. (2015), we use full-batch training, for a batch size of 4,978.

Goodfellow, Bengio, and Courville (2016) provide an excellent circuit diagram of an LSTM network for one time step. Their graph follows:



4.3 Deep RNN

Our deep RNN follows the same methodology as the deep LSTM of Yao et al. (2015). Namely, we let the hidden activity at time t act as the input activity of the higher RNN.

4.4 Attention LSTM

As an extension of the base LSTM described by Yao et al. (2015), we implement an attention mechanism on top of our deep LSTM. An attention mechanism modifies the output of the hidden layer activity in a way that focuses on certain words more than others. Letting W_a be the weight of our attention matrix, and h_t be the hidden state of our base LSTM at time t, we write our attention mechanism as follows:

$$\alpha = \text{softmax}(\tanh(h_t)W_\text{a})$$
$$r_j = \tanh(\sum_i \alpha_{ij} \odot h_{i,j,t})$$
$$c_t = [r_1, r_2, \ldots r_j]$$

The first operation here is that we restrict our hidden layer activity matrix h_t to have every element lie within $(-1, 1)$. Next we weight these elements by a learned attention matrix W_a. Lastly, we convert the output matrix to have normalized rows by applying softmax. Therefore the matrix α is a probability distribution matrix over word labels. We multiply it element-wise by the unnormalized class probability matrix h_t, and sum across rows to get a vector r. We then apply tanh to restrict each element of r to $(-1, 1)$, and let this be our cell state at time t, given by c_t. After the attention operation we have another LSTM on top, so we have an attention-based writing operation of unnormalized predictions which serve as the input layer of the top LSTM.

This set of operations allows our LSTM to focus more on certain context windows than others, and achieves the highest test accuracy of any model, though we expect bidirectional attention LSTM to outperform. Importantly, our attention LSTM

4.5 Bidirectional attention LSTM

Our bidirectional attention LSTM passes our input sequence once from start to end, and once from end to start. The results of each pass, denoted $h_{t,1}$ and $h_{t,2}$ are summed at the end before being sent to the decoding phase (the upper LSTM). Bidirectional LSTM are supposed to give better results because of their ability to pass in reversed order and get more information about dependencies. However, since more passes and more weights are introduced for this experiment, each iteration (full batch of 4,978) was taking roughly one hour. Therefore we could not run it for more than 15 epochs. In general, it could overfit, because allowing bidirectional learning increases the model complexity. Thus we might want to use dropout as a regularizer in the future. Still, after only 15 epochs, our bidirectional attention LSTM was close to overtaking the results of our deep attention LSTM, which had run for over 30 epochs.

5 Implementation

Our code is implemented in Python using Theano with no wrapper. Theano does automatic differentiation which allows us to train our model parameters W and b using backpropogation.

5.1 Deep learning network

We train our network in Theano using full-batch for the training of every model variation. For the learning rate we use adaptive moment estimation (Adam), which computes a unique learning rate for each parameter (Kingma and Ba 2014). Theoretically this should be a good optimization method for LSTM's, for two reasons. Firstly, since we have many weights and biases to update at each time step, they're unlikely to all need the same scale of update. Secondly, the long-term information flowing through our cell state receives infrequent changes, and therefore needs a uniquely high learning rate applied to these infrequent changes. Adam was created by Diederick Kingma and Jimmy Ba of the University of Toronto and Amsterdam respectively. Their paper gives many nice graphs of the robustly strong performance of Adam compared to other learning rate annealing methods.

For dataset preparation, we exactly follow Yao et al. (2015). We use the ATIS database, which includes audio recordings of people making air travel reservations, and semantic interpretation of their sentences. Words are labeled based on their semantic context. For training data, we combined the ATIS-2 and ATIS-3 corpa. Training data includes 4,978 sentences for a total of 56,590 words. For test data, we combined the Nov93 and Nov94 datasets, extracted from the ATIS-3 corpa. Test data includes 893 sentences for a total of 9198 words. All datasets have 127 distinct slot labels, including the null label. There are a total of 25,509 non-null slot label occurrences across both the training and test data. Therefore, the chance of a given word having a non-null label is about 38.78%. We do not use the named-entity feature of the ATIS dataset. We use only lexicon features. If we assume independent errors then an F1 accuracy increase of 0.6% or greater is significant at the 95% level.

5.2 Software design

Our software is written in several Python files. Each file is called from an identically-named Jupyter notebook file. The Jupyter notebook file defines a dictionary of network's parameters, and then imports and executes the corresponding Python file.

In the "lstm standard" Jupyter notebook, we perform grid search over word embedding dimensions to determine that an embedding dimension of 90 is optimal for this task. We later apply that embedding dimension to every other model as well. All software is written in Theano using Python 2.7.

6 Results

Our RNN converged in 24 iterations to an F1 of 94.1%, which almost exactly matches the authors' results. Interestingly, our deep RNN takes longer to converge, 36 iterations, and achieves only 93.74%. The standard LSTM converges in 34 iterations to 94.64%. Our deep LSTM converged in 20 iterations to 95.04%. Our moving average LSTM converged in 16 iterations to 95.03%, which is slightly higher than the authors' results. This could be attributable to initialization and the stochastic nature of Adam. Our deep LSTM with attention converged in 37 iterations to 95.09%, which is the highest of all experiments mentioned in our paper. It achieves this error rate by focusing on difficult parts to label and relatively ignoring the easy parts. Our shallow bidirectional-attention LSTM was training slowly, but had achieved an error rate of 94.35% in just 15 epochs. Our simplified LSTM with no output gate achieved 93.32%, compared with 94.0%. The simplified LSTM with no forget gate achieved 92.47%, compared with about 93.0%. Our simplified LSTM with no input gate achieved 93.61% which is around or slightly higher than the authors'. Similar to Yao et al. (2015), all of our simplified LSTM's (each dropping one gate) converged in under 10 iterations.

6.1 Comparison of Results

Yao et al. results

CRF	RNN	CNN	LSTM	ma(3)	Deep
92.94	94.11	94.35	94.85	94.92	95.08

Our results

CRF	RNN	CNN	LSTM	ma(3)	Deep
N/A	94.10	N/A	94.64	95.03	95.04

Our extensions

RNN-Deep	LSTM-attn	BLSTM-attn
93.74	95.09	94.35

7 Conclusion

We followed Yao et al. (2015) in their construction of various LSTM extensions. These extensions include a moving average regression over past predictions as an input to the current prediction, and a deep two-layer LSTM. We then recreate their experiments of dropping different gates (i.e. fixing their value to one), in order to visualize what role each LSTM gate plays in test accuracy. After closely reproducing their results in all experiments attempted, we build a deep attention LSTM, which outperforms all other models discussed in this paper, achieving an F1 error of 95.09%. We also construct a deep RNN for comparison of the depth benefits of the deep LSTM. Lastly, we construct a bidirectional attention LSTM, which takes very long to train per iteration, but in half the iterations had achieved comparable results with the other models, sitting at 94.35% F1 after 15 iterations. We plan to fully train bidirectional LSTM by the end of December 2016, and also apply dropout. Future experiments could try combinations of the extensions discussed in this paper, such as a moving average regression on top of a bidirectional attention LSTM, or depth-gated LSTM (Yao et al. 2015).

References

[1] Y. Bengio, P. Simard, and P. Frasconi, "Learning Long-Term Dependencies with Gradient Descent is Difficult," *IEEE Transactions on Neural Networks*, vol. 5, pp. 157–166, Mar. 1994.

[2] D. Britz, "Attention and Memory in Deep Learning and NLP," Jan. 2016.

[3] D. Dahl, M. Bates, M. Brown, and W. Fisher, "ATIS3 Training Data - Linguistic Data Consortium."

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. 2016. Book in preparation for MIT Press.

[5] H. et al., "Long Short-term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[6] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Dec. 2014. arXiv: 1412.6980.

[7] C. Olah and S. Carter, "Attention and Augmented Recurrent Neural Networks," *Distill*, Sept. 2016.

[8] C. Olah, "Understanding LSTM Networks," Aug. 2015.

[9] S. Ruder, "An overview of gradient descent optimization algorithms," Jan. 2016.

[10] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," *arXiv:1502.03044 [cs]*, Feb. 2015. arXiv: 1502.03044.

[11] Z. e. a. Yang, "Hierarchical Attention Networks for Document Classification," *CMU*, 2016.

[12] K. Yao, T. Cohn, K. Vylomova, K. Duh, and C. Dyer, "Depth-Gated LSTM," *arXiv:1508.03790 [cs]*, Aug. 2015. arXiv: 1508.03790.

[13] K. Yao, B. Peng, Y. Zhang, D. Yu, G. Zweig, and Y. Shi, "Spoken Language Understanding Using Long Short-Term Memory Neural Networks," *Microsoft Research*, Dec. 2014.

[14] Z. e. a. Yu, "Using bidirectional lstm recurrent neural networks to learn high-level abstractions of sequential features for automated scoring of non-native spontaneous speech," *CMU*, 2013.