– Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks

Ralf C. Staudemeyer
Faculty of Computer Science
Schmalkalden University of Applied Sciences, Germany
E-Mail: r.staudemeyer@hs-sm.de

Eric Rothstein Morris
(Singapore University of Technology and Design, Singapore
E-Mail: eric_rothstein@sutd.edu.sg)

September 23, 2019

Abstract

Long Short-Term Memory Recurrent Neural Networks (LSTM-RNN) are one of the most powerful dynamic classifiers publicly known. The network itself and the related learning algorithms are reasonably well documented to get an idea how it works. This paper will shed more light into understanding how LSTM-RNNs evolved and why they work impressively well, focusing on the early, ground-breaking publications. We significantly improved documentation and fixed a number of errors and inconsistencies that accumulated in previous publications. To support understanding we as well revised and unified the notation used.

1 Introduction

This article is an tutorial-like introduction initially developed as supplementary material for lectures focused on Artificial Intelligence. The interested reader can deepen his/her knowledge by understanding Long Short-Term Memory Recurrent Neural Networks (LSTM-RNN) considering its evolution since the early nineties. Todays publications on LSTM-RNN use a slightly different notation and a much more summarized representation of the derivations. Nevertheless the authors found the presented approach very helpful and we are confident this publication will find its audience.

Machine learning is concerned with the development of algorithms that automatically improve by practice. Ideally, the more the learning algorithm is run, the better the algorithm becomes. It is the task of the learning algorithm to create a classifier function from the training data presented. The performance of this built classifier is then measured by applying it to previously unseen data.

Artificial Neural Networks (ANN) are inspired by biological learning systems and loosely model their basic functions. Biological learning systems are

complex webs of interconnected neurons. Neurons are simple units accepting a vector of real-valued inputs and producing a single real-valued output. The most common standard neural network type are feed-forward neural networks. Here sets of neurons are organised in layers: one input layer, one output layer, and at least one intermediate hidden layer. Feed-forward neural networks are limited to static classification tasks. Therefore, they are limited to provide a static mapping between input and output. To model time prediction tasks we need a so-called dynamic classifier.

We can extend feed-forward neural networks towards dynamic classification. To gain this property we need to feed signals from previous timesteps back into the network. These networks with recurrent connections are called Recurrent Neural Networks (RNN) [74], [75]. RNNs are limited to look back in time for approximately ten timesteps [38], [56]. This is due to the fed back signal is either vanishing or exploding. This issue was addressed with Long Short-Term Memory Recurrent Neural Networks (LSTM-RNN) [22], [41], [23], [60]. LSTM networks are to a certain extend biologically plausible [58] and capable to learn more than 1,000 timesteps, depending on the complexity of the built network [41].

In the early, ground-breaking papers by Hochreiter [41] and Graves [34], the authors used different notations which made further development prone to errors and inconvenient to follow. To address this we developed a unified notation and did draw descriptive figures to support the interested reader in understanding the related equations of the early publications.

In the following, we slowly dive into the world of neural networks and specifically LSTM-RNNs with a selection of its most promising extensions documented so far. We successively explain how neural networks evolved from a single perceptron to something as powerful as LSTM. This includes vanilla LSTM, although not used in practice anymore, as the fundamental evolutionary step. With this article, we support beginners in the machine learning community to understand how LSTM works with the intention motivate its further development.

This is the first document that covers LSTM and its extensions in such great detail.

2 Notation

In this article we use the following notation:

- The learning rate of the network is η.
- A time unit is τ. Initial times of an epoch are denoted by t' and final times by t.
- The set of units of the network is N, with generic (unless stated otherwise) units $u, v, l, k \in N$.
- The set of input units is I, with input unit $i \in I$.
- The set of output units is O, with output unit $o \in O$.
- The set of non-input units is U.

- The output of a unit u (also called the activation of u) is y_u, and unlike the input, it is a single value.
- The set of units with connections to a unit u; i.e., its predecessors, is $\texttt{Pre}(u)$
- The set of units with connections from a unit u; i.e., its successors, is $\texttt{Suc}(u)$
- The weight that connects the unit v to the unit u is $W_{[v,u]}$.
- The input of a unit u coming from a unit v is denoted by $X_{[v,u]}$
- The weighted input of the unit u is z_u.
- The bias of the unit u is b_u.
- The state of the unit u is s_u.
- The squashing function of the unit u is $\texttt{f}_u$.
- The error of the unit u is e_u.
- The error signal of the unit u is ϑ_u.
- The output sensitivity of the unit k with respect to the weight $W_{[u,v]}$ is p^k_{uv}.

3 Perceptron and Delta Learning Rule

Artificial Neural Networks consist of a densely interconnected group of simple neuron-like threshold switching units. Each unit takes a number of real-valued inputs and produces a single real-valued output. Based on the connectivity between the threshold units and element parameters, these networks can model complex global behaviour.

3.1 The Perceptron

The most basic type of artificial neuron is called a perceptron. Perceptrons consist of a number of external input links, a threshold, and a single external output link. Additionally, perceptrons have an internal input, b, called bias. The perceptron takes a vector of real-valued input values, all of which are weighted by a multiplier. In a previous perceptron training phase, the perceptron learns these weights on the basis of training data. It sums all weighted input values and 'fires' if the resultant value is above a pre-defined threshold. The output of the perceptron is always Boolean, and it is considered to have fired if the output is '1'. The deactivated value of the perceptron is '-1', and the threshold value is, in most cases, '0'.

As we only have one unit for the perceptron, we omit the subindexes that refer to the unit. Given the input vector $x = \langle x_1, ..., x_n \rangle$ and trained weights $W_1, ..., W_n$, the perceptron outputs y; which is computed by the formula

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} W_i x_i + b > 0; \\ -1 & \text{otherwise.} \end{cases}$$

We refer to $z = \sum_{i=1}^{n} W_i x_i$ as the weighted input, and to $s = z + b$ as the state of the perceptron. For the perceptron to fire, its state s must exceed the value of the threshold.

Single perceptron units can already represent a number of useful functions. Examples are the Boolean functions AND, OR, NAND and NOR. Other functions are only representable using networks of neurons. Single perceptrons are limited to learning only functions that are linearly separable. In general, a problem is linear and the classes are linearly separable in an n-dimensional space if the decision surface is an $(n-1)$-dimensional hyperplane.

The general structure of a perceptron is shown in Figure 1.

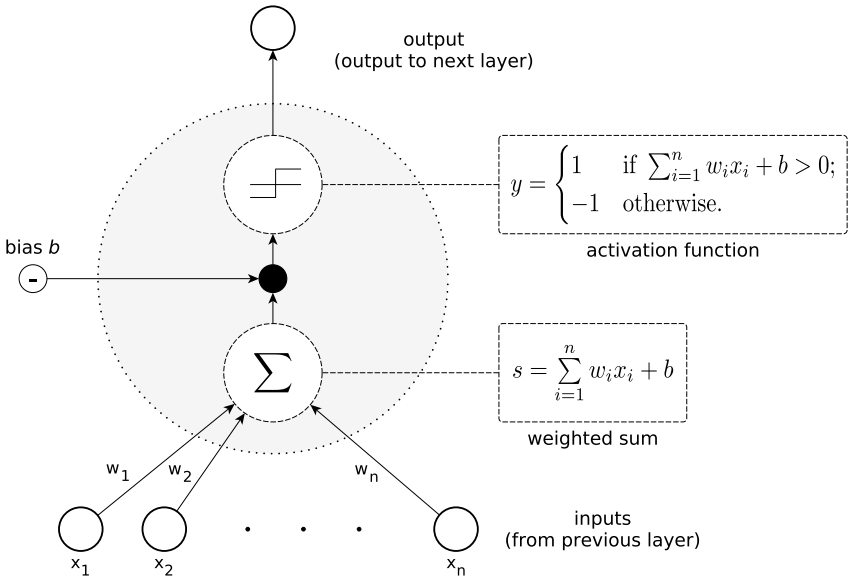
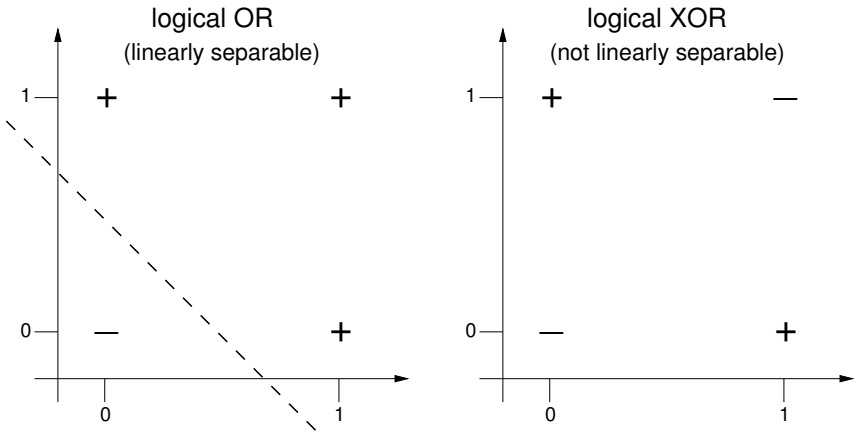


Figure 1: The general structure of the most basic type of artificial neuron, called a perceptron. Single perceptrons are limited to learning linearly separable functions.

3.2 Linear Separability

To understand linear separability, it is helpful to visualise the possible inputs of a perceptron on the axes of a two-dimensional graph. Figure 2 shows representations of the Boolean functions OR and XOR. The OR function is linearly separable, whereas the XOR function is not. In the figure, pluses are used for an input where the perceptron fires and minuses, where it does not. If the pluses and minuses can be completely separated by a single line, the problem is linearly separable in two dimensions. The weights of the trained perceptron should represent that line.



Input $_1$	Input $_2$	Output
1	1	1
1	0	1
0	1	1
0	0	0

Input $_1$	Input $_2$	Output
1	1	0
1	0	1
0	1	1
0	0	0

Figure 2: Representations of the Boolean functions OR and XOR. The figures show that the OR function is linearly separable, whereas the XOR function is not.

3.3 The Delta Learning Rule

Perceptron training is learning by imitation, which is called 'supervised learning'. During the training phase, the perceptron produces an output and compares it with a derived output value provided by the training data. In cases of misclassification, it then modifies the weights accordingly. [55] show that in a finite time, the perceptron will converge to reproduce the correct behaviour, provided that the training examples are linearly separable. Convergence is not assured if the training data is not linearly separable.

A variety of training algorithms for perceptrons exist, of which the most common are the perceptron learning rule and the delta learning rule. Both start with random weights and both guarantee convergence to an acceptable hypothesis. Using the perceptron learning rule algorithm, the perceptron can learn from a set of samples A sample is a pair $\langle x, d\rangle$ where x is the input and d is its label. For the sample $\langle x, d\rangle$, given the input $x = \langle x_1, \ldots, x_n\rangle$, the old weight vector $W = \langle W_1, \ldots, W_n\rangle$ is updated to the new vector W' using the rule

$$W_i' = W_i + \Delta W_i,$$

with

$$\Delta W_i = \eta(d - y)x_i,$$

where y is the output calculated using the input x and the weights W and η is the learning rate. The learning rate is a constant that controls the degree to which the weights are changed. As stated before, the initial weight vector W^0 has random values. The algorithm will only converge towards an optimum if the training data is linearly separable, and the learning rate is sufficiently small. The perceptron rule fails if the training examples are not linearly separable.

The delta learning rule was specifically designed to handle linearly separable and linearly non-separable training examples. It also calculates the errors between calculated output and output data from training samples, and modifies the weights accordingly. The modification of weights is achieved by using the gradient optimisation descent algorithm, which alters them in the direction that produces the steepest descent along the error surface towards the global minimum error. The delta learning rule is the basis of the error backpropagation algorithm, which we will discuss later in this section.

3.4 The Sigmoid Threshold Unit

The sigmoid threshold unit is a different kind of artificial neuron, very similar to the perceptron, but uses a sigmoid function to calculate the output. The output y is computed by the formula

$$y = \frac{1}{(1 - e^{-l \times s})},$$

with

$$s = \sum_{i=1}^{n} W_i x_i + b,$$

where b is the bias and l is a positive constant that determines the steepness of the sigmoid function. The major effect on the perceptron is that the output

of the sigmoid threshold unit now has more than two possible values; now, the output is "squashed" by a continuous function that ranges between 0 and 1. Accordingly, the function $\frac{1}{(1-e^{-l \times s})}$ is called the 'squashing' function, because it maps a very large input domain onto a small range of outputs. For a low total input value, the output of the sigmoid function is close to zero, whereas it is close to one for a high total input value. The slope of the sigmoid function is adjusted by the threshold value. The advantage of neural networks using sigmoid units is that they are capable of representing non-linear functions. Cascaded linear units, like the perceptron, are limited to representing linear functions. A sigmoid threshold unit is sketched in Figure 3.

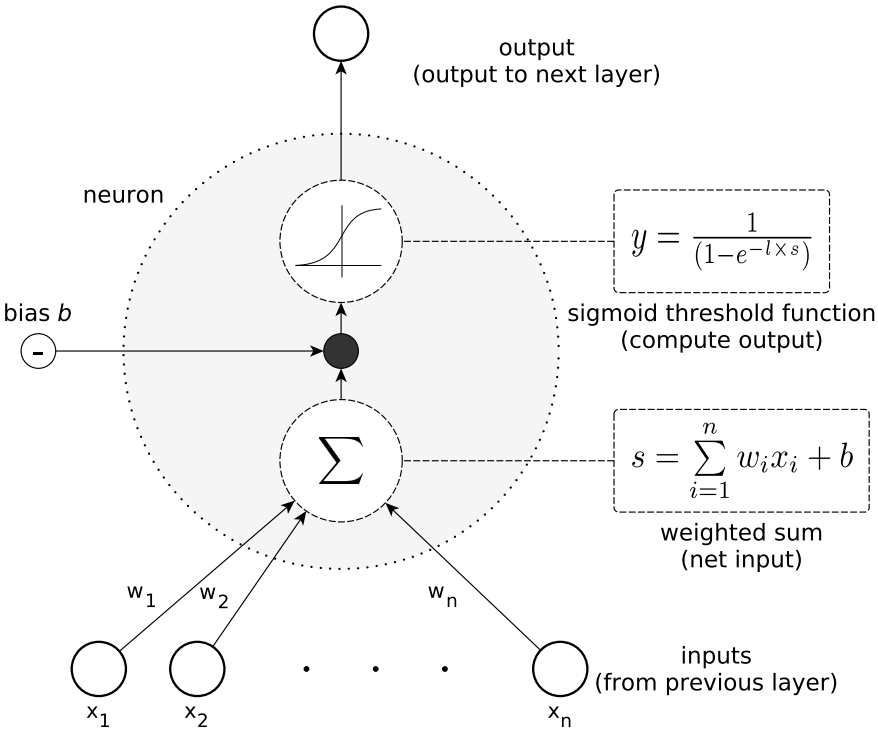


Figure 3: The sigmoid threshold unit is capable of representing non-linear functions. Its output is a continuous function of its input, which ranges between 0 and 1.

4 Feed-Forward Neural Networks and Backpropagation

In feed-forward neural networks (FFNNs), sets of neurons are organised in layers, where each neuron computes a weighted sum of its inputs. Input neurons take signals from the environment, and output neurons present signals to the environment. Neurons that are not directly connected to the environment, but

which are connected to other neurons, are called hidden neurons.

Feed-forward neural networks are loop-free and fully connected. This means that each neuron provides an input to each neuron in the following layer, and that none of the weights give an input to a neuron in a previous layer.

The simplest type of neural feed-forward networks are single-layer perceptron networks. Single-layer neural networks consist of a set of input neurons, defined as the input layer, and a set of output neurons, defined as the output layer. The outputs of the input-layer neurons are directly connected to the neurons of the output layer. The weights are applied to the connections between the input and output layer.

In the single-layer perceptron network, every single perceptron calculates the sum of the products of the weights and the inputs. The perceptron fires '1' if the value is above the threshold value; otherwise, the perceptron takes the deactivated value, which is usually '-1'. The threshold value is typically zero.

Sets of neurons organised in several layers can form multilayer, forward-connected networks. The input and output layers are connected via at least one hidden layer, built from set(s) of hidden neurons. The multilayer feed-forward neural network sketched in Figure 4, with one input layer and three output layers (two hidden and one output), is classified as a 3-layer feed-forward neural network. For most problems, feed-forward neural networks with more than two layers offer no advantage.

Multilayer feed-forward networks using sigmoid threshold functions are able to express non-linear decision surfaces. Any function can be closely approximated by these networks, given enough hidden units.

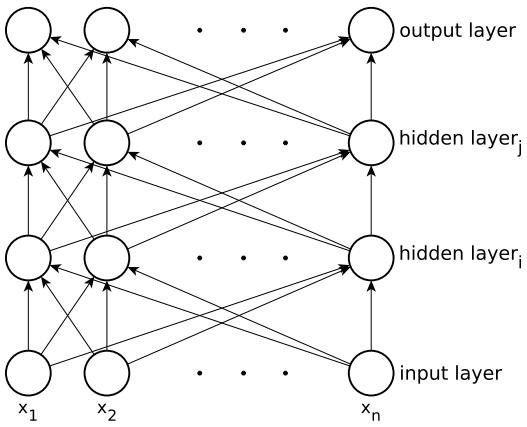


Figure 4: A multilayer feed-forward neural network with one input layer, two hidden layers, and an output layer. Using neurons with sigmoid threshold functions, these neural networks are able to express non-linear decision surfaces.

The most common neural network learning technique is the error backpropagation algorithm. It uses gradient descent to learn the weights in multilayer networks. It works in small iterative steps, starting backwards from the output layer towards the input layer. A requirement is that the activation function of the neuron is differentiable.

Usually, the weights of a feed-forward neural network are initialised to small, normalised random numbers using bias values. Then, error backpropagation applies all training samples to the neural network and computes the input and output of each unit for all (hidden and) output layers.

The set of units of the network is $N \triangleq I \sqcup H \sqcup O$, where $\sqcup$ is disjoint union, and I, H, O are the sets of input, hidden and output units, respectively. We denote input units by i, hidden units by h and output units by o. For convenience, we define the set of non-input units $U \triangleq H \sqcup O$. For a non-input unit $u \in U$, the input to u is denoted by x_u, its state by s_u, its bias by b_u and its output by y_u. Given units $u, v \in U$, the weight that connects u with v is denoted by W_{uv}.

To model the external input that the neural network receives, we use the external input vector $x = \langle x_1, \ldots, x_n \rangle$. For each component of the external input vector we find a corresponding input unit that models it, so the output of the i^{th} input unit should be equal i^{th} component of the input to the network (i.e., x_i), and consequently $|I| = n$.

For the non-input unit $u \in U$, the output of u, written y_u, is defined using the sigmoid activation function by

$$y_u = \frac{1}{1 + e^{-s_u}} \tag{1}$$

where s_u is the state of u, and it is defined by

$$s_u = z_u + b_u; \tag{2}$$

where b_u is the bias of u, and z_u is the weighted input of u, defined in turn by

$$\begin{aligned} z_u &= \sum_v W_{[v,u]} X_{[v,u]}, \quad \text{with } v \in \texttt{Pre}\,(u) \\ &= \sum_v W_{[v,u]} y_v; \end{aligned} \tag{3}$$

where $X_{[v,u]}$ is the information that v passes as input to u, and $\texttt{Pre}\,(u)$ is the set of units v that preceed u; that is, input units, and hidden units that feed their outputs y_v (see Equation (1)) multiplied by the corresponding weight $W_{[v,u]}$ to the unit u.

Starting from the input layer, the inputs are propagated forwards through the network until the output units are reached at the output layer. Then, the output units produce an observable output (the network output) y. More precisely, for $o \in O$, its output y_o corresponds to the o^{th} component of y.

Next, the backpropagation learning algorithm propagates the error backwards, and the weights and biases are updated such that we reduce the error with respect to the present training sample. Starting from the output layer, the algorithm compares the network output y_o with the corresponding desired target output d_o. It calculates the error e_o for each output neuron using some error function to be minimised. The error e_o is computed as

$$e_o = (d_o - y_o)$$

and we have the following notion of overall error of the network

$$E = \frac{1}{2} \sum_{o \in O} e_o^2$$

To update the weight $W_{[u,v]}$, we will use the formula

$$\Delta W_{[u,v]} = -\eta \frac{\partial E}{\partial W_{[u,v]}}$$

where η is the learning rate. We now make use of the factors $\frac{\partial y_u}{\partial y_u}$ and $\frac{\partial s_u}{\partial s_u}$ to calculate the weight update by deriving the error with respect to the activation, and the activation in terms of the state, and in turn the derivative of the state with respect to the weight:

$$\Delta W_{[u,v]} = -\eta \frac{\partial E}{\partial y_u} \frac{\partial y_u}{\partial s_u} \frac{\partial s_u}{\partial W_{[u,v]}}.$$

The derivative of the error with respect to the activation for output units is

$$\frac{\partial E}{\partial y_o} = -(d_o - y_o),$$

now, the derivative of the activation with respect to the state for output units is

$$\frac{\partial y_o}{\partial s_o} = y_o(1 - y_o),$$

and the derivative of the state with respect to a weight that connects the hidden unit h to the output unit o is

$$\frac{\partial s_u}{\partial W_{[u,v]}} = y_h$$

Let us define, for the output unit o, the error signal by

$$\vartheta_o = -\frac{\partial E}{\partial y_o} \frac{\partial y_o}{\partial s_o} \tag{4}$$

for output units we have that

$$\vartheta_o = (d_o - y_o) y_o (1 - y_o), \tag{5}$$

and we see that we can update the weight between the hidden unit h and the output unit o by

$$\Delta W_{[h,o]} = \eta \vartheta_o y_h.$$

Now, for a hidden unit h, if we consider that its notion of error is related to how much it contributed to the production of a faulty output, then we can backpropagate the error from the output units that h sends signals to; more precisely, for an input unit i, we need to expand the equation $\Delta W_{[i,h]} = -\eta \frac{\partial E}{\partial W_{[i,h]}}$ to

$$\Delta W_{[i,h]} = -\eta \sum_o \frac{\partial E}{\partial y_o} \frac{\partial y_o}{\partial s_o} \frac{\partial s_o}{\partial y_h} \frac{\partial y_h}{\partial s_h} \frac{\partial s_h}{\partial W_{[i,h]}} \quad \text{with } o \in \mathtt{Suc}\,(h)\,.$$

where $\mathtt{Suc}\,(h)$ is the set of units that succeed h; that is, the units that are fed with the output of h as part of their input. By solving the partial derivatives, we obtain

$$\begin{aligned} \Delta W_{[i,h]} &= -\eta \sum_o \left(\vartheta_o W_{[h,o]}\right) \frac{\partial y_h}{\partial s_h} \frac{\partial s_h}{\partial W_{[i,h]}} \\ &= \eta \sum_o \left(\vartheta_o W_{[h,o]}\right) y_h (1 - y_h) y_i. \end{aligned}$$

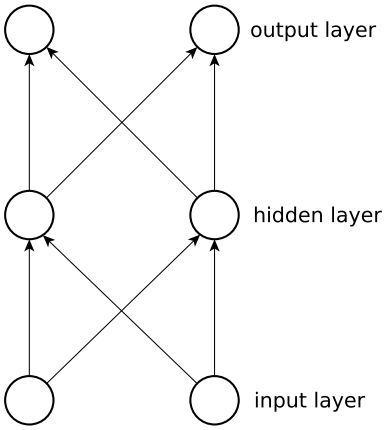


Figure 5: This figure shows a feed-forward neural network.

If we define the error signal of the hidden unit h by

$$\vartheta_h = \sum_o \left(\vartheta_o W_{[h,o]}\right) y_h(1 - y_h); \quad \text{with } o \in \mathtt{Suc}\,(h)\,,$$

then we have a uniform expression for weight change; that is,

$$\Delta W_{[v,u]} = \eta \vartheta_u y_v.$$

We calculate $\Delta W_{[v,u]}$ again and again until all network outputs are within an acceptable range, or some other terminating condition is reached.

5 Recurrent Neural Networks

Recurrent neural networks (RNNs) [74, 75] are dynamic systems; they have an internal state at each time step of the classification. This is due to circular connections between higher- and lower-layer neurons and optional self-feedback connections. These feedback connections enable RNNs to propagate data from earlier events to current processing steps. Thus, RNNs build a memory of time series events.

5.1 Basic Architecture

RNNs range from partly to fully connected, and two simple RNNs are suggested by [46] and [16]. The Elman network is similar to a three-layer neural network, but additionally, the outputs of the hidden layer are saved in so-called 'context cells'. The output of a context cell is circularly fed back to the hidden neuron along with the originating signal. Every hidden neuron has its own context cell and receives input both from the input layer and the context cells. Elman networks can be trained with standard error backpropagation, the output from the context cells being simply regarded as an additional input. Figures 5 and 6 show a standard feed-forward network in comparison with such an Elman network.

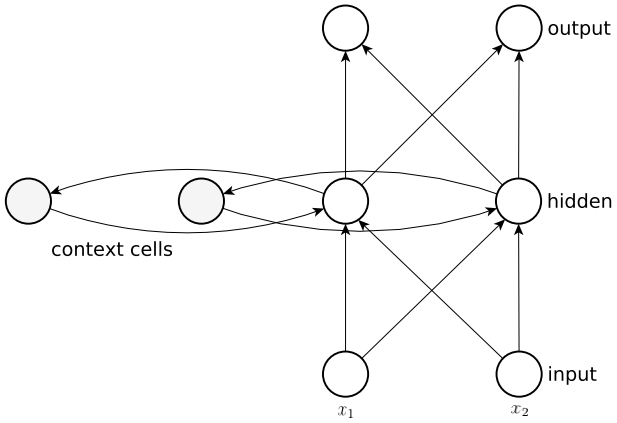


Figure 6: This figure shows an Elman neural network.

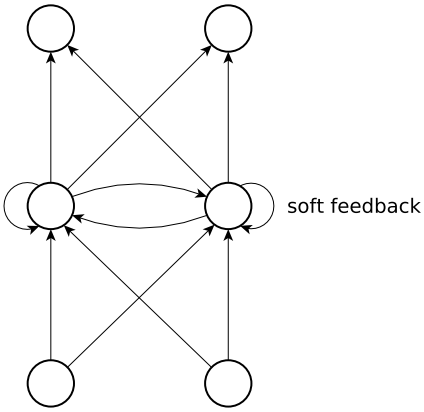


Figure 7: This figure shows a partially recurrent neural network with self-feedback in the hidden layer.

Jordan networks have a similar structure to Elman networks, but the context cells are instead fed by the output layer. A partial recurrent neural network with a fully connected recurrent hidden layer is shown in Figure 7. Figure 8 shows a fully connected RNN.

RNNs need to be trained differently to the feed-forward neural networks (FFNNs) described in Section 4. This is because, for RNNs, we need to propagate information through the recurrent connections in-between steps. The most common and well-documented learning algorithms for training RNNs in temporal, supervised learning tasks are backpropagation through time (BPTT) and real-time recurrent learning (RTRL). In BPTT, the network is unfolded in time to construct an FFNN. Then, the generalised delta rule is applied to update the weights. This is an offline learning algorithm in the sense that we first collect the data and then build the model from the system. In RTRL, the gradient information is forward propagated. Here, the data is collected online from the

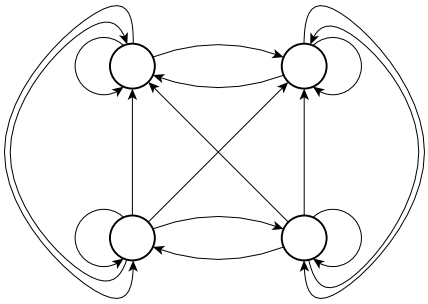


Figure 8: This figure shows a fully recurrent neural network (RNN) with self-feedback connections.

system and the model is learned during collection. Therefore, RTRL is an online learning algorithm.

6 Training Recurrent Neural Networks

The most common methods to train recurrent neural networks are Backpropagation Through Time (BPTT) [62, 74, 75] and Real-Time Recurrent Learning (RTRL) [75, 76], whereas BPTT is the most common method. The main difference between BPTT and RTRL is the way the weight changes are calculated. The original formulation of LSTM-RNNs used a combination of BPTT and RTRL. Therefore we cover both learning algorithms in short.

6.1 Backpropagation Through Time

The BPTT algorithm makes use of the fact that, for a finite period of time, there is an FFNN with identical behaviour for every RNN. To obtain this FFNN, we need to unfold the RNN in time. Figure 9a shows a simple, fully recurrent neural network with a single two-neuron layer. The corresponding feed-forward neural network, shown in Figure 9b, requires a separate layer for each time step with the same weights for all layers. If weights are identical to the RNN, both networks show the same behaviour.

The unfolded network can be trained using the backpropagation algorithm described in Section 4. At the end of a training sequence, the network is unfolded in time. The error is calculated for the output units with existing target values using some chosen error measure. Then, the error is injected backwards into the network and the weight updates for all time steps calculated. The weights in the recurrent version of the network are updated with the sum of its deltas over all time steps.

We calculate the error signal for a unit for all time steps in a single pass, using the following iterative backpropagation algorithm. We consider discrete time steps $1, 2, 3...$, indexed by the variable τ. The network starts at a point in time t' and runs until a final time t. This time frame between t' and t is called an epoch. Let U be the set of non input units, and let $\mathtt{f}_u$ be the differentiable, non-linear squashing function of the unit $u \in U$; the output $y_u(\tau)$ of u at time

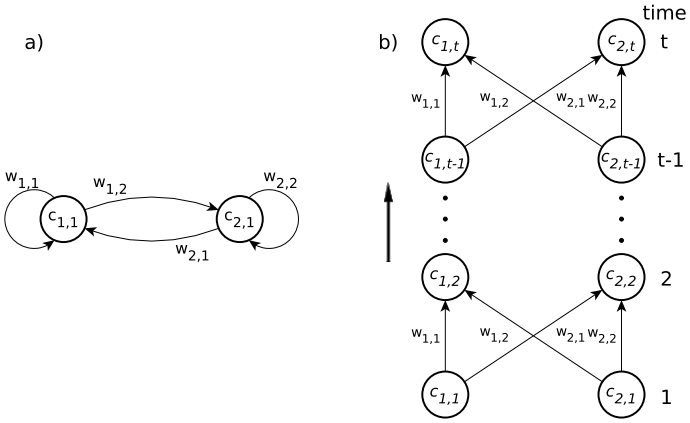


Figure 9: Figure a shows a simple fully recurrent neural network with a two-neuron layer. The same network unfolded over time with a separate layer for each time step is shown in Figure b. The latter representation is a feed-forward neural network.

τ is given by

$$y_u(\tau) = \mathtt{f}_u(z_u(\tau)) \tag{6}$$

with the weighted input

$$\begin{aligned} z_u(\tau+1) &= \sum_l W_{[u,l]} X_{[l,u]}(\tau+1), \quad \text{with } l \in \mathtt{Pre}\,(u) \\ &= \sum_v W_{[u,v]} y_v(\tau) + \sum_i W_{[u,i]} y_i(\tau+1) \end{aligned} \tag{7}$$

where $v \in U \cap \mathtt{Pre}\,(u)$ and $i \in I$, the set of input units. Note that the inputs to u at time $\tau+1$ are of two types: the environmental input that arrives at time $\tau+1$ via the input units, and the recurrent output from all non-input units in the network produced at time τ. If the network is fully connected, then $U \cap \mathtt{Pre}\,(u)$ is equal to the set U of non-input units. Let $T(\tau)$ be the set of non-input units for which, at time τ, the output value $y_u(\tau)$ of the unit $u \in T(\tau)$ should match some target value $d_u(\tau)$. The cost function is the summed error $E_{total}(t', t)$ for the epoch $t', t'+1, \ldots, t$, which we want to minimise using a learning algorithm. Such total error is defined by

$$E_{total}(t', t) = \sum_{\tau=t'}^{t} E(\tau), \tag{8}$$

with the error $E(\tau)$ at time τ defined using the squared error as an objective function by

$$E(\tau) = \frac{1}{2} \sum_{u \in U} (e_u(\tau))^2, \tag{9}$$

and with the error $e_u(\tau)$ of the non-input unit u at time τ defined by

$$e_u(\tau) = \begin{cases} d_u(\tau) - y_u(\tau) & \text{if } u \in T(\tau), \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

To adjust the weights, we use the error signal $\vartheta_u(\tau)$ of a non-input unit u at a time τ, which is defined by

$$\vartheta_u(\tau) = \frac{\partial E(\tau)}{\partial z_u(\tau)}. \tag{11}$$

When we unroll ϑ_u over time, we obtain the equality

$$\vartheta_u(\tau) = \begin{cases} \mathtt{f}'_u(z_u(\tau))e_u(\tau) & \text{if } \tau = t, \\ \mathtt{f}'_u(z_u(\tau))\left(\sum_{k\in U} W_{[k,u]}\vartheta_k(\tau+1)\right) & \text{if } t' \leq \tau < t \end{cases}. \tag{12}$$

After the backpropagation computation is performed down to time t', we calculate the weight update $\Delta W_{[u,v]}$ in the recurrent version of the network. This is done by summing the corresponding weight updates for all time steps:

$$\Delta W_{[u,v]} = -\eta \frac{\partial E_{total}(t',t)}{\partial W_{[u,v]}}$$

with

$$\begin{aligned} \frac{\partial E_{total}(t',t)}{\partial W_{[u,v]}} &= \sum_{\tau=t'}^{t} \vartheta_u(\tau)\frac{\partial z_u(\tau)}{\partial W_{[u,v]}} \\ &= \sum_{\tau=t'}^{t} \vartheta_u(\tau) X_{[u,v]}(\tau). \end{aligned}$$

BPTT is described in more detail in [74], [62] and [76].

6.2 Real-Time Recurrent Learning

The RTRL algorithm does not require error propagation. All the information necessary to compute the gradient is collected as the input stream is presented to the network. This makes a dedicated training interval obsolete. The algorithm comes at significant computational cost per update cycle, and the stored information is non-local; i.e., we need an additional notion called sensitivity of the output, which we'll explain later. Nevertheless, the memory required depends only on the size of the network and not on the size of the input.

Following the notation from the previous section, we will now define for the network units $v \in I \cup U$ and $u, k \in U$, and the time steps $t' \leq \tau \leq t$. Unlike BPTT, in RTRL we assume the existence of a label $d_k(\tau)$ at every time τ (given that it is an online algorithm) for every non-input unit k, so the training objective is to minimise the overall network error, which is given at time step τ by

$$E(\tau) = \frac{1}{2}\sum_{k\in U}(d_k(\tau) - y_k(\tau))^2.$$

We conclude from Equation 8 that the gradient of the total error is also the sum of the gradient for all previous time steps and the current time step:

$$\nabla_W E_{total}(t', t+1) = \nabla_W E_{total}(t', t) + \nabla_W E(t+1).$$

During presentation of the time series to the network, we need to accumulate the values of the gradient at each time step. Thus, we can also keep track of

the weight changes $\Delta W_{[u,v]}(\tau)$. After presentation, the overall weight change for $W_{[u,v]}$ is then given by

$$\Delta W_{[u,v]} = \sum_{\tau=t'+1}^{t} \Delta W_{[u,v]}(\tau). \tag{13}$$

To get the weight changes we need to calculate

$$\Delta W_{[u,v]}(\tau) = -\eta \frac{\partial E(\tau)}{\partial W_{[u,v]}}$$

for each time step t. After expanding this equation via gradient descent and by applying Equation 9, we find that

$$\begin{aligned} \Delta W_{[u,v]}(\tau) &= -\eta \sum_{k \in U} \frac{\partial E(\tau)}{\partial y_k(\tau)} \frac{\partial y_k(\tau)}{\partial W_{[u,v]}} \\ &= -\eta \sum_{k \in U} (d_k(\tau) - y_k(\tau)) \left(\frac{\partial y_k(\tau)}{\partial W_{[u,v]}} \right). \end{aligned} \tag{14}$$

Since the error $e_k(\tau) = d_k(\tau) - y_k(\tau)$ is always known, we need to find a way to calculate the second factor only. We define the quantity

$$p^k_{uv}(\tau) = \frac{\partial y_k(\tau)}{\partial W_{[u,v]}}, \tag{15}$$

which measures the sensitivity of the output of unit k at time τ to a small change in the weight $W_{[u,v]}$, in due consideration of the effect of such a change in the weight over the entire network trajectory from time t' to t. The weight $W_{[u,v]}$ does not have to be connected to unit k, which makes the algorithm non-local. Local changes in the network can have an effect anywhere in the network.

In RTRL, the gradient information is forward-propagated. Using Equations 6 and 7, the output $y_k(t+1)$ at time step $t+1$ is given by

$$y_k(t+1) = \mathtt{f}_k(z_k(t+1)) \tag{16}$$

with the weighted input

$$\begin{aligned} z_k(t+1) &= \sum_l W_{[k,l]} X_{[k,l]}(t+1), \quad \text{with } l \in \mathtt{Pre}\,(k) \\ &= \sum_{v \in U} W_{[k,v]} y_v(t) + \sum_{i \in I} W_{[k,i]} y_i(t+1). \end{aligned} \tag{17}$$

By differentiating Equations 15, 16 and 17, we can calculate results for all

time steps $\geq t+1$ with

$$
\begin{aligned}
p_{uv}^{k}(t+1) = \frac{\partial y_k(t+1)}{\partial W_{[u,v]}} &= \frac{\partial}{\partial W_{[u,v]}} \left[\mathtt{f}_k \left(\sum_{l \in \mathtt{Pre}(k)} W_{[k,l]} X_{[k,l]}(t+1) \right) \right] \\
&= \mathtt{f}'_k(z_k(t+1)) \left[\frac{\partial}{\partial W_{[u,v]}} \left(\sum_{l \in \mathtt{Pre}(k)} W_{[k,l]} X_{[k,l]}(t+1) \right) \right] \\
&= \mathtt{f}'_k(z_k(t+1)) \left[\left(\sum_{l \in \mathtt{Pre}(k)} \frac{\partial W_{[k,l]}}{\partial W_{[u,v]}} X_{[k,l]}(t+1) \right) + \left(\sum_{l \in \mathtt{Pre}(k)} W_{[k,l]} \frac{\partial X_{[k,l]}(t+1)}{\partial W_{[u,v]}} \right) \right] \\
&= \mathtt{f}'_k(z_k(t+1)) \left[\delta_{uk} X_{[u,v]}(t+1) + \left(\sum_{l \in U} W_{[k,l]} \frac{\partial y_l(t)}{\partial W_{[u,v]}} + \underbrace{\sum_{i \in I} W_{[k,i]} \frac{\partial y_i(t+1)}{\partial W_{[u,v]}}}_{\substack{=0 \text{ because } y_i(t+1) \\ \text{is independent of } W_{[u,v]}}} \right) \right] \\
&= \mathtt{f}'_k(z_k(t+1)) \left[\delta_{uk} X_{[u,v]}(t+1) + \sum_{l \in U} W_{[k,l]} p_{uv}^{l}(t) \right] .
\end{aligned}
\tag{18}
$$

where δ_{uk} is the Kronecker delta; that is,

$$
\delta_{uk} = \begin{cases} 1 & \text{if } u = k \\ 0 & \text{if } otherwise, \end{cases}
$$

Assuming that the initial state of the network has no functional dependency on the weights, the derivative for the first time step is

$$
p_{uv}^{k}(t') = \frac{\partial y_k(t')}{\partial W_{[u,v]}} = 0 \ . \tag{19}
$$

Equation 18 shows how $p_{uv}^{k}(t+1)$ can be calculated in terms of $p_{uv}^{k}(t)$. In this sense, the learning algorithm becomes incremental, so that we can learn as we receive new inputs (in real time), and we no longer need to perform back-propagation through time.

Knowing the initial value for p_{uv}^{k} at time t' from Equation 19, we can recursively calculate the quantities p_{uv}^{k} for the first and all subsequent time steps using Equation 18. Note that $p_{uv}^{k}(\tau)$ uses the values of $W_{[u,v]}$ at t', and not values in-between t' and τ. Combining these values with the error vector $e(\tau)$ for that time step, using Equation 14, we can finally calculate the negative error gradient $\bigtriangledown WE(\tau)$. The final weight change for $W_{[u,v]}$ can be calculated using Equations 14 and 13.

A more detailed description of the RTRL algorithm is given in [75] and [76].

7 Solving the Vanishing Error Problem

Standard RNN cannot bridge more than 5–10 time steps ([22]). This is due to that back-propagated error signals tend to either grow or shrink with every time

step. Over many time steps the error therefore typically blows-up or vanishes ([5, 42]). Blown-up error signals lead straight to oscillating weights, whereas with a vanishing error, learning takes an unacceptable amount of time, or does not work at all.

The explanation of how gradients are computed by the standard backpropagation algorithm and the basic vanishing error analysis is as follows: we update weights after the network has trained from time t' to time t using the formula

$$\Delta W_{[u,v]} = -\eta \frac{\partial E_{total}(t',t)}{\partial W_{[u,v]}},$$

with

$$\frac{\partial E_{total}(t',t)}{\partial W_{[u,v]}} = \sum_{\tau=t'}^{t} \vartheta_u(\tau) X_{[u,v]}(\tau),$$

where the backpropagated error signal at time τ (with $t' \leq \tau < t$) of the unit u is

$$\vartheta_u(\tau) = \mathtt{f}'_u(z_u(\tau)) \left(\sum_{v \in U} W_{vu} \vartheta_v(\tau + 1) \right). \tag{20}$$

Consequently, given a fully recurrent neural network with a set of non-input units U, the error signal that occurs at any chosen output-layer neuron $o \in O$, at time-step τ, is propagated back through time for $t - t'$ time-steps, with $t' < t$ to an arbitrary neuron v. This causes the error to be scaled by the following factor:

$$\frac{\partial \vartheta_v(t')}{\partial \vartheta_o(t)} = \begin{cases} \mathtt{f}'_v(z_v(t')) W_{[o,v]} & \text{if } t - t' = 1, \\ \mathtt{f}'_v(z_v(t')) \left(\sum_{u \in U} \frac{\partial \vartheta_u(t'+1)}{\partial \vartheta_o(t)} W_{[u,v]} \right) & \text{if } t - t' > 1 \end{cases}$$

To solve the above equation, we unroll it over time. For $t' \leq \tau \leq t$, let u_τ be a non-input-layer neuron in one of the replicas in the unrolled network at time τ. Now, by setting $u_t = v$ and $u_{t'} = o$, we obtain the equation

$$\frac{\partial \vartheta_v(t')}{\partial \vartheta_o(t)} = \sum_{u_{t'} \in U} \cdots \sum_{u_{t-1} \in U} \left(\prod_{\tau=t'+1}^{t} \mathtt{f}'_{u_\tau}(z_{u_\tau}(t - \tau + t')) W_{[u_\tau, u_{\tau-1}]} \right). \tag{21}$$

Observing Equation 21, it follows that if

$$|\mathtt{f}'_{u_\tau}(z_{u_\tau}(t - \tau + t')) W_{[u_\tau, u_{\tau-1}]}| > 1 \tag{22}$$

for all τ, then the product will grow exponentially, causing the error to blow-up; moreover, conflicting error signals arriving at neuron v can lead to oscillating weights and unstable learning. If now

$$|\mathtt{f}'_{u_\tau}(z_{u_\tau}(t - \tau + t')) W_{[u_\tau, u_{\tau-1}]}| < 1 \tag{23}$$

for all τ, then the product decreases exponentially, causing the error to vanish, preventing the network from learning within an acceptable time period. Finally, the equation

$$\sum_{o \in O} \frac{\partial \vartheta_v(t')}{\partial \vartheta_o(t)}$$

shows that if the local error vanishes, then the global error also vanishes.

A more detailed theoretical analysis of the problem with long-term dependencies is presented in [39]. The paper also briefly outlines several proposals on how to address this problem.

8 Long Short-Term Neural Networks

One solution that addresses the vanishing error problem is a gradient-based method called long short-term memory (LSTM) published by [41], [42], [22] and [23]. LSTM can learn how to bridge minimal time lags of more than 1,000 discrete time steps. The solution uses constant error carousels (CECs), which enforce a constant error flow within special cells. Access to the cells is handled by multiplicative gate units, which learn when to grant access.

8.1 Constant Error Carousel

Suppose that we have only one unit u with a single connection to itself. The local error back flow of u at a single time-step τ follows from Equation 20 and is given by

$$\vartheta_u(\tau) = \mathtt{f}'_u(z_u(\tau))W_{[u,u]}\vartheta_u(\tau+1).$$

From Equations 22 and 23 we see that, in order to ensure a constant error flow through u, we need to have

$$\mathtt{f}'_u(z_u(\tau))W_{[u,u]} = 1.0$$

and by integration we have

$$\mathtt{f}_u(z_u(\tau)) = \frac{z_u(\tau)}{W_{[u,u]}}.$$

From this, we learn that $\mathtt{f}_u$ must be linear, and that u's activation must remain constant over time; i.e.,

$$y_u(\tau+1) = \mathtt{f}_u(z_u(\tau+1)) = \mathtt{f}_u(y_u(\tau)W_{[u,u]}) = y_u(\tau).$$

This is ensured by using the identity function $\mathtt{f}_u = id$, and by setting $W_{[u,u]} = 1.0$. This preservation of error is called the constant error carousel (CEC), and it is the central feature of LSTM, where short-term memory storage is achieved for extended periods of time. Clearly, we still need to handle the connections from other units to the unit u, and this is where the different components of LSTM networks come into the picture.

8.2 Memory Blocks

In the absence of new inputs to the cell, we now know that the CEC's backflow remains constant. However, as part of a neural network, the CEC is not only connected to itself, but also to other units in the neural network. We need to take these additional weighted inputs and outputs into account. Incoming connections to neuron u can have conflicting weight update signals, because the same weight is used for storing and ignoring inputs. For weighted output

connections from neuron u, the same weights can be used to both retrieve u's contents and prevent u's output flow to other neurons in the network.

To address the problem of conflicting weight updates, LSTM extends the CEC with input and output gates connected to the network input layer and to other memory cells. This results in a more complex LSTM unit, called a memory block; its standard architecture is shown in Figure 11.

The input gates, which are simple sigmoid threshold units with an activation function range of $[0, 1]$, control the signals from the network to the memory cell by scaling them appropriately; when the gate is closed, activation is close to zero. Additionally, these can learn to protect the contents stored in u from disturbance by irrelevant signals. The activation of a CEC by the input gate is defined as the cell state. The output gates can learn how to control access to the memory cell contents, which protects other memory cells from disturbances originating from u. So we can see that the basic function of multiplicative gate units is to either allow or deny access to constant error flow through the CEC.

9 Training LSTM-RNNs - the Hybrid Learning Approach

In order to preserve the CEC in LSTM memory block cells, the original formulation of LSTM used a combination of two learning algorithms: BPTT to train network components located after cells, and RTRL to train network components located before and including cells. The latter units work with RTRL because there are some partial derivatives (related to the state of the cell) that need to be computed during every step, no matter if a target value is given or not at that step. For now, we only allow the gradient of the cell to be propagated through time, truncating the rest of the gradients for the other recurrent connections.

We define discrete time steps in the form $\tau = 1, 2, 3, ...$ Each step has a forward pass and a backward pass; in the forward pass the output/activation of all units are calculated, whereas in the backward pass, the calculation of the error signals for all weights is performed.

9.1 The Forward Pass

Let M be the set of memory blocks. Let m_c be the c-th memory cell in the memory block m, and $W_{[u,v]}$ be a weight connecting unit u to unit v.

In the original formulation of LSTM, each memory block m is associated with one input gate $\mathtt{in}_m$ and one output gate $\mathtt{out}_m$. The internal state of a memory cell m_c at time $\tau + 1$ is updated according to its state $s_{m_c}(\tau)$ and according to the weighted input $z_{m_c}(\tau + 1)$ multiplied by the activation of the input gate $y_{\mathtt{in}_m}(\tau + 1)$. Then, we use the activation of the output gate $z_{\mathtt{out}_m}(\tau + 1)$ to calculate the activation of the cell $y_{m_c}(\tau + 1)$.

The activation $y_{\mathtt{in}_m}$ of the input gate $\mathtt{in}_m$ is computed as

$$y_{\mathtt{in}_m}(\tau + 1) = \mathtt{f}_{\mathtt{in}_m}(z_{\mathtt{in}_m}(\tau + 1)) \tag{24}$$

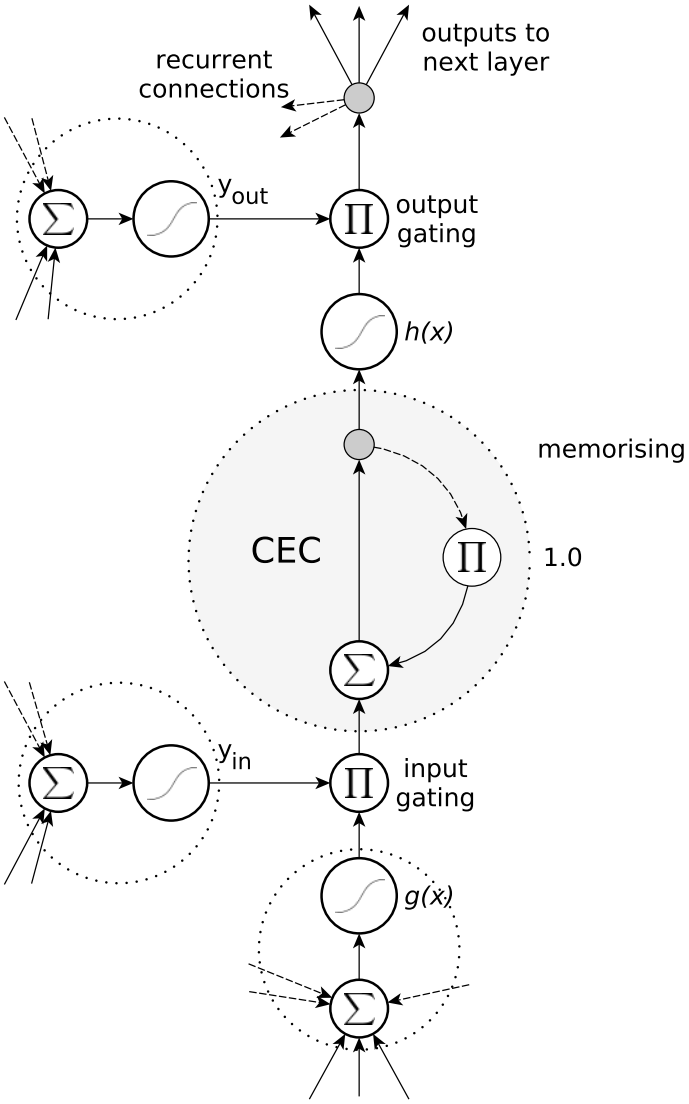


Figure 10: A standard LSTM memory block. The block contains (at least) one cell with a recurrent self-connection (CEC) and weight of '1'. The state of the cell is denoted as s_c. Read and write access is regulated by the input gate, y_{in}, and the output gate, y_{out}. The internal cell state is calculated by multiplying the result of the squashed input, g, by the result of the input gate, y_{in}, and then adding the state of the last time step, $s_c(t-1)$. Finally, the cell output is calculated by multiplying the cell state, s_c, by the activation of the output gate, y_{out}.

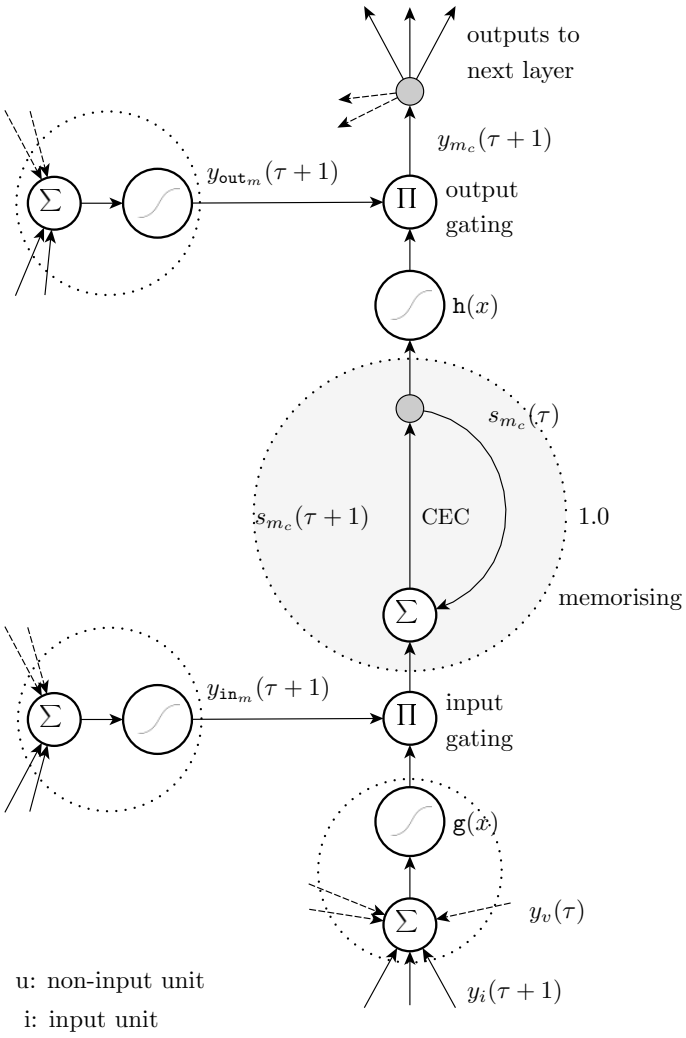


Figure 11: A standard LSTM memory block. The block contains (at least) one cell with a recurrent self-connection (CEC) and weight of '1'. The state of the cell is denoted as s_c. Read and write access is regulated by the input gate, $y_{\mathtt{in}}$, and the output gate, $y_{\mathtt{out}}$. The internal cell state is calculated by multiplying the result of the squashed input, $\mathtt{g}(x)$, by the result of the input gate and then adding the state of the current time step, $s_{m_c}(\tau)$, to the next, $s_{m_c}(\tau+1)$. Finally, the cell output is calculated by multiplying the cell state by the activation of the output gate.

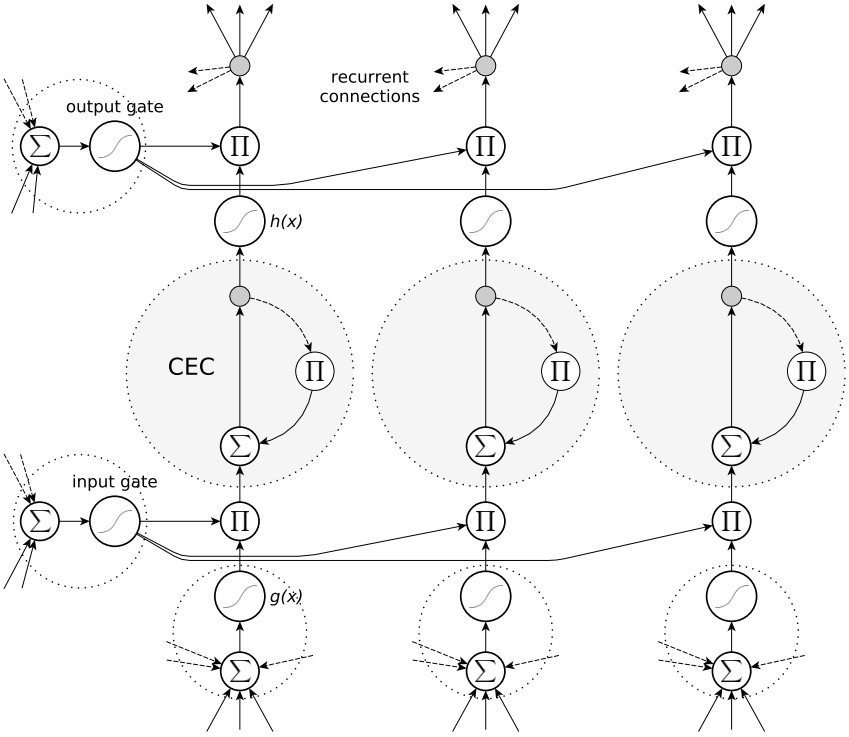


Figure 12: A three cell LSTM memory block with recurrent self-connections

with the input gate input

$$\begin{aligned} z_{\mathtt{in}_m}(\tau+1) &= \sum_u W_{[\mathtt{in}_m,u]} X_{[u,\mathtt{in}_m]}(\tau+1), \quad \text{with } u \in \mathtt{Pre}\left(\mathtt{in}_m\right), \\ &= \sum_{v\in U} W_{[\mathtt{in}_m,v]} y_v(\tau) + \sum_{i\in I} W_{[\mathtt{in}_m,i]} y_i(\tau+1). \end{aligned} \tag{25}$$

The activation of the output gate $\mathtt{out}_m$ is

$$y_{\mathtt{out}_m}(\tau+1) = \mathtt{f}_{\mathtt{out}_m}(z_{\mathtt{out}_m}(\tau+1)) \tag{26}$$

with the output gate input

$$\begin{aligned} z_{\mathtt{out}_m}(\tau+1) &= \sum_u W_{[\mathtt{out}_m,u]} X_{[u,\mathtt{out}_m]}(\tau+1), \quad \text{with } u \in \mathtt{Pre}\left(\mathtt{out}_m\right). \\ &= \sum_{v\in U} W_{[\mathtt{out}_m,v]} y_v(\tau) + \sum_{i\in I} W_{[\mathtt{out}_m,i]} y_i(\tau+1). \end{aligned} \tag{27}$$

The results of the gates are scaled using the non-linear squashing function $\mathtt{f}_{\mathtt{in}_m} = \mathtt{f}_{\mathtt{out}_m} = \mathtt{f}$, defined by

$$\mathtt{f}(s) = \frac{1}{1+e^{-s}} \tag{28}$$

so that they are within the range $[0,1]$. Thus, the input for the memory cell will only be able to pass if the signal at the input gate is sufficiently close to '1'.

For a memory cell m_c in the memory block m, the weighted input $z_{m_c}(\tau+1)$ is defined by

$$\begin{aligned} z_{m_c}(\tau+1) &= \sum_u W_{[m_c,u]} X_{[u,m_c]}(\tau+1), \quad \text{with } u \in \mathtt{Pre}\left(m_c\right). \\ &= \sum_{v\in U} W_{[m_c,v]} y_v(\tau) + \sum_{i\in I} W_{[m_c,i]} y_i(\tau+1). \end{aligned} \tag{29}$$

As we mentioned before, the internal state $s_{m_c}(\tau+1)$ of the unit in the memory cell at time $\tau+1$ is computed differently; the weighted input is squashed and then multiplied by the activation of the input gate, and then the state of the last time step $s_{m_c}(\tau)$ is added. The corresponding equation is

$$s_{m_c}(\tau+1) = s_{m_c}(\tau) + y_{\mathtt{in}_m}(\tau+1)\mathtt{g}(z_{m_c}(\tau+1)) \tag{30}$$

with $s_{m_c}(0) = 0$ and the non-linear squashing function for the cell input

$$\mathtt{g}(z) = \frac{4}{1+e^{-z}} - 2 \tag{31}$$

which, in this case, scales the result to the range $[-2,2]$.

The output y_{m_c} is now calculated by squashing and multiplying the cell state s_{m_c} by the activation of the output gate $y_{\mathtt{out}_m}$:

$$y_{m_c}(\tau+1) = y_{\mathtt{out}_m}(\tau+1)\mathtt{h}(s_{m_c}(\tau+1)). \tag{32}$$

with the non-linear squashing function

$$\mathtt{h}(z) = \frac{2}{1+e^{-z}} - 1 \tag{33}$$

with range $[-1, 1]$.

Assuming a layered, recurrent neural network with standard input, standard output and hidden layer consisting of memory blocks, the activation of the output unit o is computed as

$$y_o(\tau + 1) = \mathtt{f}_o(z_o(\tau + 1)) \tag{34}$$

with

$$z_o(\tau + 1) = \sum_{u \in U - G} W_{[o,u]} y_u(\tau + 1). \tag{35}$$

where G is the set of gate units, and we can again use the logistic sigmoid in Equation 28 as a squashing function $\mathtt{f}_o$.

9.2 Forget Gates

The self-connection in a standard LSTM network has a fixed weight set to '1' in order to preserve the cell state over time. Unfortunately, the cell states s_m tend to grow linearly during the progression of a time series presented in a continuous input stream. The main negative effect is that the entire memory cell loses its memorising capability, and begins to function like an ordinary RNN network neuron.

By manually resetting the state of the cell at the beginning of each sequence, the cell state growth can be limited, but this is not practical for continuous input where there is no distinguishable end, or subdivision is very complex and error prone.

To address this problem, [22] suggested that an adaptive forget gate could be attached to the self-connection. Forget gates can learn to reset the internal state of the memory cell when the stored information is no longer needed. To this end, we replace the weight '1.0' of the self-connection from the CEC with a multiplicative, forget gate activation y_φ, which is computed using a similar method as for the other gates:

$$y_{\varphi_m}(\tau + 1) = \mathtt{f}_{\varphi_m}(z_{\varphi_m}(\tau + 1) + b_{\varphi_m}) \ , \tag{36}$$

where $\mathtt{f}$ is the squashing function from Equation 28 with a range $[0, 1]$, b_{φ_m} is the bias of the forget gate, and

$$\begin{aligned} z_{\varphi_m}(\tau + 1) &= \sum_u W_{[\varphi_m,u]} X_{[u,\varphi_m]}(\tau + 1), \quad \text{with } u \in \mathtt{Pre}\,(\varphi_m)\,. \\ &= \sum_{v \in U} W_{[\varphi_m,v]} y_v(\tau) + \sum_{i \in I} W_{[\varphi_m,i]} y_i(\tau + 1). \end{aligned} \tag{37}$$

Originally, b_{φ_m} is set to 0, however, following the recommendation by [47], we fix b_{φ_m} to 1, in order to improve the performance of LSTM (see Section 10.3).

The updated equation for calculating the internal cell state s_{m_c} is

$$s_{m_c}(\tau + 1) = s_{m_c}(\tau) \underbrace{y_{\varphi_m}(\tau + 1)}_{\substack{=1 \text{ without} \\ \text{forget gate}}} + y_{\mathtt{in}_m}(\tau + 1)\mathtt{g}(z_{m_c}(\tau + 1)) \tag{38}$$

with $s_{m_c}(0) = 0$ and using the squashing function in Equation 31, with a range $[-2, 2]$. The extended forward pass is given simply by exchanging Equation 30 for Equation 38.

The bias weights of input and output gates are initialised with negative values, and the weights of the forget gate are initialised with positive values. From this, it follows that at the beginning of training, the forget gate activation will be close to '1.0'. The memory cell will behave like a standard LSTM memory cell without a forget gate. This prevents the LSTM memory cell from forgetting, before it has actually learned anything.

9.3 Backward Pass

LSTM incorporates elements from both BPTT and RTRL. Thus, we separate units into two types: those units whose weight changes are computed using a variation of BPTT (i.e, output units, hidden units, and the output gates), and those whose weight changes are computed using a variation of RTRL (i.e., the input gates, the forget gates and the cells).

Following the notation used in previous sections, and using Equations 8 and 10, the overall network error at time step τ is

$$E(\tau) = \frac{1}{2} \sum_{o \in O} (\underbrace{d_o(\tau) - y_o(\tau)}_{e_o(\tau)})^2. \tag{39}$$

Let us first consider units that work with BPTT. We define the notion of individual error of a unit u at time τ by

$$\vartheta_u(\tau) = -\frac{\partial E(\tau)}{\partial z_u(\tau)}, \tag{40}$$

where z_u is the weighted input of the unit. We can expand the notion of weight contribution as follows

$$\begin{aligned} \Delta W_{[u,v]}(\tau) &= -\eta \frac{\partial E(\tau)}{\partial W_{[u,v]}} \\ &= -\eta \frac{\partial E(\tau)}{\partial z_u(\tau)} \frac{\partial z_u(\tau)}{\partial W_{[u,v]}}. \end{aligned}$$

The factor $\frac{\partial z_u(\tau)}{\partial W_{[u,v]}}$ corresponds to the input signal that comes from the unit v to the unit u. However, depending on the nature of u, the individual error varies. If u is equal to an output unit o, then

$$\vartheta_o(\tau) = \mathtt{f}'_o(z_o(\tau))(d_o(\tau) - y_o(\tau));$$

thus, the weight contribution of output units is

$$\Delta W_{[o,v]}(\tau) = \eta \vartheta_o(\tau) X_{[v,o]}(\tau).$$

Now, if u is equal to a hidden unit h located between cells and output units, then

$$\vartheta_h(\tau) = \mathtt{f}'_h(z_h(\tau)) \left(\sum_{o \in O} W_{[o,h]} \vartheta_o(\tau) \right);$$

where O is the set of output units, and the weight contribution of hidden units is

$$\Delta W_{[h,v]}(\tau) = \eta \vartheta_h(\tau) X_{[v,h]}(\tau).$$

Finally, if u is equal to the output gate $\texttt{out}_m$ of the memory block m, then

$$\vartheta_{\texttt{out}_m}(\tau) \overset{\text{tr}}{=} \texttt{f}'_{\texttt{out}_m}(z_{\texttt{out}_m}(\tau)) \left(\sum_{m_c \in m} \texttt{h}(s_{m_c}(\tau)) \sum_{o \in O} W_{[o,m_c]} \vartheta_o(\tau) \right);$$

where $\overset{\text{tr}}{=}$ means the equality only holds if the error is truncated so that it does not propagate "too much"; that is, it prevents the error from propagating back to the unit via its own feedback connection. Finally, the weight contribution for output gates is

$$\Delta W_{[\texttt{out}_m,v]}(\tau) = \eta \vartheta_{\texttt{out}_m}(\tau) X_{[v,\texttt{out}_m]}(\tau).$$

Let us now consider units that work with RTRL. In this case, the individual errors of the input gate and the forget gate revolve around the individual error of the cells in the memory block. We define the individual error of the cell m_c of the memory block m by

$$\begin{aligned}
\vartheta_{m_c}(\tau) &\overset{\text{tr}}{=} -\frac{\partial E(\tau)}{\partial s_{m_c}(\tau)} + \underbrace{\vartheta_{m_c}(\tau+1) y_{\varphi_m}(\tau+1)}_{\text{recurrent connection}} \\
&\overset{\text{tr}}{=} \frac{\partial y_{m_c}(\tau)}{\partial s_{m_c}(\tau)} \left(\sum_{o \in O} \frac{\partial z_o(\tau)}{\partial y_{m_c}(\tau)} \left(-\frac{\partial E(\tau)}{\partial z_o(\tau)} \right) \right) + \vartheta_{m_c}(\tau+1) y_{\varphi_m}(\tau+1) \\
&\overset{\text{tr}}{=} y_{\texttt{out}_m}(\tau) \texttt{h}'(s_{m_c}(\tau)) \left(\sum_{o \in O} W_{[o,m_c]} \vartheta_o(\tau) \right) + \vartheta_{m_c}(\tau+1) y_{\varphi_m}(\tau+1).
\end{aligned} \tag{41}$$

Note that this equation does not consider the recurrent connection between the cell and other units, propagating back in time only the error through its recurrent connection (accounting for the influence of the forget gate). We use the following partial derivatives to expand the weight contribution for the cell as follows

$$\begin{aligned}
\Delta W_{[m_c,v]}(\tau) &= -\eta \frac{\partial E(\tau)}{\partial W_{[m_c,v]}} \\
&= -\eta \frac{\partial E(\tau)}{\partial s_{m_c}(\tau)} \frac{\partial s_{m_c}(\tau)}{\partial W_{[m_c,v]}} \\
&= \eta \vartheta_{m_c}(\tau) \frac{\partial s_{m_c}(\tau)}{\partial W_{[m_c,v]}}
\end{aligned} \tag{42}$$

and the weight contribution for forget and input gates as follows

$$\begin{aligned}
\Delta W_{[u,v]}(\tau) &= -\eta \frac{\partial E(\tau)}{\partial W_{[u,v]}} \\
&= -\eta \sum_{m_c \in m} \frac{\partial E(\tau)}{\partial s_{m_c}(\tau)} \frac{\partial s_{m_c}(\tau)}{\partial W_{[u,v]}} \\
&= \eta \sum_{m_c \in m} \vartheta_{m_c}(\tau) \frac{\partial s_{m_c}(\tau)}{\partial W_{[u,v]}}.
\end{aligned} \tag{43}$$

Now, we need to define what is the value of $\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[u,v]}}$. As expected, these also depend on the nature of the unit u. If u is equal to the cell m_c, then

$$\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[m_c,v]}} \overset{\text{tr}}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[m_c,v]}} y_{\varphi_m}(\tau+1) + \mathtt{g}'(z_{m_c}(\tau+1))\mathtt{f}_{\mathtt{in}_m}(z_{\mathtt{in}_m}(\tau+1))y_v(\tau). \quad (44)$$

Now, if u is equal to the input gate $\mathtt{in}_m$, then

$$\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[\mathtt{in}_m,v]}} \overset{\text{tr}}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[\mathtt{in}_m,v]}} y_{\varphi_m}(\tau+1) + \mathtt{g}(z_{m_c}(\tau+1))\mathtt{f}'_{\mathtt{in}_m}(z_{\mathtt{in}_m}(\tau+1))y_v(\tau). \quad (45)$$

Finally, if u is equal to a forget gate φ_m, then

$$\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[\varphi_m,v]}} \overset{\text{tr}}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[\varphi_m,v]}} y_{\varphi_m}(\tau+1) + s_{m_c}(\tau)\mathtt{f}'_{\varphi_m}(z_{\varphi_m}(\tau+1))y_v(\tau). \quad (46)$$

with $s_{m_c}(0) = 0$. A more detailed version of the LSTM backward pass with forget gates is described in [22].

9.4 Complexity

In this section, we present a complexity measure following the same principles that Gers used in [22]; namely, we assume that every memory block contains the same number of cells (usually one), and that output units only receive signals from cell units and not from other units in the network. Let B, C, In, Out be the number of of memory blocks, memory cells in each block, input units and output units, respectively. Now, for each memory block we need to resolve the (recurrent) connections for each cell, input gate, forget gate and output gate. Solving these connections yields a complexity measure of

$$B\left(C\left(\underbrace{(B.C)}_{\text{cells}} + \underbrace{(B.C)}_{\text{input gates}} + \underbrace{(B.C)}_{\text{forget gates}}\right) + \underbrace{B.C}_{\text{output gates}}\right) \sim \mathcal{O}\left(B^2.C^2\right). \quad (47)$$

We also need to solve the connections from input units and to output units; these are, respectively

$$In.B.S \sim \mathcal{O}\left(In.B.S\right), \quad (48)$$

and

$$Out.B.S \sim \mathcal{O}\left(Out.B.S\right). \quad (49)$$

The numbers B, C, In and Out do not change as the network executes, and, at each step, the number of weight updates is bounded by the number of connections; thus, we can say that LSTM's computational complexity per step and weight is $\mathcal{O}(1)$.

9.5 Strengths and limitations of LSTM-RNNs

According to [23], LSTM excels on tasks in which a limited amount of data must be remembered for a long time. This property is attributed to the use of memory blocks. Memory blocks are interesting constructions: they have access control in the form of input and output gates; which prevent irrelevant

information from entering or leaving the memory block. Memory blocks also have a forget gate which weights the information inside the cells, so whenever previous information becomes irrelevant for some cells, the forget gate can reset the state of the different cell inside the block. Forget gates also enable continuous prediction [54], because they can make cells completely forget their previous state; preventing biases in prediction.

Like other algorithms, LSTM requires the topology of the network to be fixed a priori. The number of memory blocks in networks does not change dynamically, so the memory of the network is ultimately limited. Moreover, [23] point out that it is unlikely to overcome this limitation by increasing the network size homogeneously, and suggest that modularisation promotes effective learning. The process of modularisation is, however, "not generally clear".

10 Problem specific topologies

LSTM-RNN permits many different variants and topologies. These partially problem specific and can be derived [3] from the basic method [41], [21] covered in Section 8 and 9. More recently the basic method is referenced to as 'vanilla' LSTM, which used in practise these days only with various extensions and modifications. In the following sections we cover the most common in use, namely bidirectional LSTM (BLSTM-CTC) ([34], [27], [31]), Grid LSTM (or N-LSTM) [49] and Gated Recurrent Unit (GRU) ([10], [13]). There are various variants of Grid LSTM. The most important to note are Multidimensional LSTM ([29], [35]), Stacked LSTM ([18], [33], [68]). Specifically we would like to also point out the more recent variant Sequence-to-Sequence ([68], [36], [8], [80], [69]) and attention-based learning [12], which are both important to mention in the context of cognitive learning tasks.

10.1 Bidirectional LSTM

Conventional RNNs analyse, for any given point in a sequence, just one direction during processing: the past. The work published in [34] explores the possibility of analysing both the future as well as the past of a given point in the context of LSTM. At a very high level, bidirectional means that the input is presented forwards and backwards to two separate LSTM networks, both of which are connected to the same output layer. According to [34], bidirectional training possesses an architectural advantage over unidirectional training if used to classify phonemes.

Bidirectional LSTM removes the one-step truncation originally present in LSTM, and implements a full error gradient calculation. This full error gradient approach eased the implementation of bidirectional LSTM, and allowed it to be trained using standard BPTT.

In 2006 [28] introduced an RNN objective function named Connectionist Temporal Classification (CTC). The advantage of CTC is that it enables the LSTM-RNN to handle input data not segmented into sequences. This is important if the correct segmentation of data is difficult to achieve (e.g. separation of letters in handwriting). Later this lead to the now common variant BLSTM-CTC as documented by [52], [19], [27].

10.2 Grid LSTM

Grid LSTM presented by [49] is an attempt to generalise the advantages of LSTM – including its ability to select or ignore inputs– into deep networks of a unified architecture. An N-dimensional grid LSTM or N-LSTM is a network arranged in a grid of N dimensions, with LSTM cells along and in-between some (or all) of the dimensions, enabling communication among consecutive layers.

Grid LSTM is analogous to the stacked LSTM [33], but it adds cells along the depth dimension too, i.e., in-between layers. Additionally, N-LSTM networks with $N > 2$ are analogous to multidimensional LSTM [29], but they differ again by the cells along the depth dimension, and by the ability of grid LSTM networks to modulate the interaction among layers such that it is not prone to the instability present in Multidimensional LSTM.

Consider a trained LSTM network with weights W, whose hidden cells emit a collection of signals represented by the vector $\vec{y_h}$ and whose memory units emit a collection of signals represented by the vector $\vec{y_h}$. Whenever this LSTM network is provided an input vector $\vec{x}$, there is a change in the signals emitted by both hidden units and memory cells; let $\vec{y_h}'$ and $\vec{s_m}'$ represent the new values of signals. Let P be a projection matrix, the concatenation of the new input signals and the recurrent signals is given by

$$X = \begin{bmatrix} P\vec{x} \\ \vec{y_h} \end{bmatrix} \tag{50}$$

An LSTM transform, which changes the values of hidden and memory signals as previously mentioned, can be formulated as follows:

$$(X, \vec{s_m}) \xrightarrow{W} (\vec{y_h}', \vec{s_m}') \tag{51}$$

Before we explain in detail the architecture of Grid LSTM blocks, we quickly review Stacked LSTM and Multidimensional LSTM architectures.

10.2.1 Stacked LSTM

A stacked LSTM [33], as its name suggests, stacks LSTM layers on top of each other in order to increase capacity. At a high level, to stack N LSTM networks, we make the first network have X_1 as defined in Equation (52), but we make the i-th network have X_i defined by

$$X_i = \begin{bmatrix} \vec{y_h}_{i-1} \\ \vec{y_h}_i \end{bmatrix} \tag{52}$$

instead, replacing the input signals $\vec{x}$ with the hidden signals from the previous LSTM transform, effectively "stacking" them.

10.2.2 Multidimensional LSTM

In Multidimensional LSTM networks [29], inputs are structured in an N-dimensional grid instead of being sequences of values; for example, a solid expressed as a three-dimensional array of voxels. To use this structure of inputs, Multidimensional LSTM networks increase the number of recurrent connections from 1 to N; thus, an N-dimensional LSTM receives N hidden vectors $\vec{y_h}_1, \ldots, \vec{y_h}_N$ and

N memory vectors $\vec{s_{m1}}, \ldots, \vec{s_{mN}}$ as input, then the network outputs a single hidden vector $\vec{y_h}$ and a single memory vector $\vec{s_m}$. For multidimensional LSTM networks, we define X by

$$X = \begin{bmatrix} P\vec{x} \\ \vec{y_{h1}} \\ \vdots \\ \vec{y_{hN}} \end{bmatrix} \tag{53}$$

and the memory signal vector $\vec{s_m}$ is calculated using

$$\vec{s_m} = \sum_{i=1}^{N} \vec{\varphi_i} \odot \vec{s_{mi}} + \vec{\mathtt{in}_m} \odot \vec{z_m} \tag{54}$$

where $\odot$ is the Hadamard product, $\vec{\varphi}$ is a vector consisting of N forget signals (one for each $\vec{y_{hi}}$), and $\vec{\mathtt{in}_m}$ and $\vec{z_m}$ respectively correspond to the signals of the input gate and the weighted input of the memory cell (see Equation (38) to compare Equation (54) with the standard calculation of $\vec{s_m}$).

10.2.3 Grid LSTM Blocks

Due to the high number of connections, large multidimensional LSTM networks are usually unstable [49]. Grid LSTM offers an alternate way of computing the new memory vector. However, unlike multidimensional LSTM, a Grid LSTM block outputs N hidden vectors $\vec{y_{h1}}', \ldots, \vec{y_{hN}}'$ and N memory vectors $\vec{s_{m1}}', \ldots, \vec{s_{mN}}'$ that are all distinct. To do so, the model concatenates the hidden vectors from the N dimensions as follows

$$X = \begin{bmatrix} \vec{y_{h1}} \\ \vdots \\ \vec{y_{hN}} \end{bmatrix} \tag{55}$$

The grid LSTM block computes N LSTM transforms, one for each dimension, as follows

$$\begin{aligned} (X, \vec{s_{m1}}) &\xrightarrow{W_1} (\vec{y_{h1}}', \vec{s_{m1}}') \\ &\vdots \\ (X, \vec{s_{mN}}) &\xrightarrow{W_N} (\vec{y_{hN}}', \vec{s_{mN}}') \end{aligned} \tag{56}$$

Each transform applies standard LSTM across its respective dimension. Having X as input to all transforms represents the sharing of hidden signals across the different dimension of the grid; note that each transform independently manages its memory signals.

10.3 Gated Recurrent Unit (GRU)

[10] propose the Gated Recurrent Unit (GRU) architecture for RNN as an alternative to LSTM. GRU has empirically been found to outperform LSTM on nearly all tasks, except language modelling with naive initialization [47]. GRU units, unlike LSTM memory blocks, do not have a memory cell; although they

do have gating units: a reset gate and an update gate. More precisely, let H be the set of GRU units; if $u \in H$, then we define the activation $y_{\mathtt{res}_u}(\tau+1)$ of the reset gate $\mathtt{res}_u$ at time $\tau+1$ by

$$y_{\mathtt{res}_u}(\tau+1) = \mathtt{f}_{\mathtt{res}_u}\left(s_{\mathtt{res}_u}(\tau+1)\right), \tag{57}$$

where $\mathtt{f}_{\mathtt{res}_u}$ is the squashing function of the reset gate (usually a sigmoid function), and $s_{\mathtt{res}_u}(\tau+1)$ is the state of the reset gate $\mathtt{res}_u$ at time $\tau+1$, which is defined by

$$s_{\mathtt{res}_u}(\tau+1) = z_{\mathtt{res}_u}(\tau+1) + b_{\mathtt{res}_u}, \tag{58}$$

where $b_{\mathtt{res}_u}$ is the bias of the reset gate, and $z_{\mathtt{res}_u}(\tau+1)$ is the weighted input of the reset gate at time $\tau+1$, which is in turn defined by

$$z_{\mathtt{res}_u}(\tau+1) = \sum_u W_{[\mathtt{res}_u,u]} X_{[u,\mathtt{res}_u]}(\tau+1), \quad \text{with } u \in \mathtt{Pre}\left(\mathtt{res}_u\right); \tag{59}$$

$$= \sum_{h\in H} W_{[\mathtt{res}_u,h]} y_h(\tau) + \sum_{i\in I} W_{[\mathtt{res}_u,i]} y_i(\tau+1), \tag{60}$$

where I is the set of input units.

Similarly, we define define the activation $y_{\mathtt{upd}_u}(\tau+1)$ of the update gate $\mathtt{upd}_u$ at time $\tau+1$ by

$$y_{\mathtt{upd}_u}(\tau+1) = \mathtt{f}_{\mathtt{upd}_u}\left(s_{\mathtt{upd}_u}(\tau+1)\right) \tag{61}$$

where $\mathtt{f}_{\mathtt{upd}_u}$ is the squashing function of the update gate (again, usually a sigmoid function), and $s_{\mathtt{upd}_u}(\tau+1)$ is the state of the update gate $\mathtt{upd}_u$ at time $\tau+1$, defined by

$$s_{\mathtt{upd}_u}(\tau+1) = z_{\mathtt{upd}_u}(\tau+1) + b_{\mathtt{upd}_u}, \tag{62}$$

where $b_{\mathtt{upd}_u}$ is the bias of the update gate, and $z_{\mathtt{upd}_u}(\tau+1)$ is the weighted input of the update gate at time $\tau+1$, which in turn is defined by

$$z_{\mathtt{upd}_u}(\tau+1) = \sum_u W_{[\mathtt{upd}_u,u]} X_{[u,\mathtt{upd}_u]}(\tau+1), \quad \text{with } u \in \mathtt{Pre}\left(\mathtt{upd}_u\right); \tag{63}$$

$$= \sum_{h\in H} W_{[\mathtt{upd}_u,h]} y_h(\tau) + \sum_{i\in I} W_{[\mathtt{upd}_u,i]} y_i(\tau+1), \tag{64}$$

GRU reset and input gates behave like normal units in a recurrent network. The main characteristic of GRU is the way the activation of the GRU units is defined. A GRU unit $u \in H$ has an associated candidate activation $\widetilde{y}_u(\tau+1)$ at time $\tau+1$, formally defined by

$$\widetilde{y}_u(\tau+1) = \mathtt{f}_u\left(\underbrace{\sum_{i\in I} W_{[u,i]} y_i(\tau+1)}_{\text{External input at time } \tau+1} + \underbrace{y_{\mathtt{res}_u}(\tau+1) \sum_{h\in H} \left(W_{[u,h]} y_h(\tau)\right)}_{\text{Gated recurrent connection}} + \underbrace{b_u}_{\text{Bias}}\right) \tag{65}$$

where $\mathtt{f}_u$ is usually $\mathtt{tanh}$, and the activation $y_u(\tau+1)$ of the GRU unit u at time $\tau+1$ is defined by

$$y_u(\tau+1) = y_{\mathtt{upd}_u}(\tau+1)y_u(\tau) + (1 - y_{\mathtt{upd}_u}(\tau+1))\widetilde{y}_u(\tau+1) \tag{66}$$

Note the similarities between Equations (38) and (66). The factor $y_{\mathtt{upd}_u}(\tau+1)$ appears to emulate the function of the forget gate of LSTM, while the factor $(1-y_{\mathtt{upd}_u}(\tau+1))$ appears to emulate the function of the the input gate of LSTM.

11 Applications of LSTM-RNN

In this final section we cover a selection of well-known publications which proved relevant over time.

11.1 Early learning tasks

In early experiments LSTM proved applicable to various learning tasks, previously considered impossible to learn. This included recalling high precision real numbers over extended noisy sequences [41], learning context free languages [21], and various tasks that require precise timing and counting [23]. In [43] LSTM was successfully introduced to meta-learning with a program search tasks to approximate a learning algorithm for quadratic functions. The successful application of reinforcement learning to solve non-Markovian learning tasks with long-term dependencies was shown by [2].

11.2 Cognitive learning tasks

LSTM-RNNs proved great strengths in solving a large variety of cognitive learning tasks. Speech and handwriting recognition, and more recently machine translation are the most predominant in literature. Other cognitive learning tasks include emotion recognition from speech [78], text generation [67], handwriting generation [24], constituency parsing [71], and conversational modelling [72].

11.2.1 Speech recognition

A first indication of the capabilities of neural networks in tasks related to natural language was given by [4] with a neural language modelling task. In 2003 good results applying standard LSTM-RNN networks with a mix of LSTM and sigmoidal units to speech recognition tasks were obtained by [25, 26]. Better results comparable to Hidden-Markov-Model (HMM)-based systems [7] were achieved using bidirectional training with BLSTM [6, 34]. A variant named BLSTM-CTC [28, 19, 17] finally outperformed HMMs, with recent improvements documented in [44, 77]. A deep variant of stacked BLSTM-CTC was used in 2013 by [33] and later extended with a modified CTC objective function by [30], both achieving outstanding results. The performance of different LSTM-RNN architectures on large vocabulary speech recognition tasks was investigated by [63], with best results using an LSTM/HMM hybrid architecture. Comparable results were achieved by [20].

More recently LSTM was improving results using the sequence-to-sequence framework ([68]) and attention-based learning ([11], [12]). In 2015 [8] introduced an specialised architecture for speech recognition with two functions, the first called 'listener' and the latter called 'attend and spell'. The 'listener' function uses BLSTM with a pyramid structure (pBLSTM), similar to clockwork RNNs introduced by [50]. The other function, 'attend and spell', uses an attention-based LSTM transducer developed by [1] and [12]. Both functions are trained with methods introduced in the sequence-to-sequence framework [68] and in attention-based learning [1].

11.2.2 Handwriting recognition

In 2007 [52] introduced BLSTM-CTC and applied it to online handwriting recognition, with results later outperforming Hidden-Markov-based recognition systems presented by [32]. [27] combined BLSTM-CTC with a probabilistic language model and by this developed a system capable of directly transcribing raw online handwriting data. In a real-world use case this system showed a very high automation rate with an error rate comparable to a human on this kind of task ([57]). In another approach [35] combined BLSTM-CTC with multidimensional LSTM and applied it to an offline handwriting recognition task, as well outperforming classifiers based on Hidden-Markov models. In 2013 [81, 61] applied the very successful regularisation method dropout as proposed by [37, 64]).

11.2.3 Machine translation

In 2014 [10] the authors applied the RNN encoder-decoder neural network architecture to machine translation and improved the performance of a statistical machine translation system. The RNN Encoder-Decoder architecture is based on an approach communicated by [48]. A very similar deep LSTM architecture, referred to as sequence-to-sequence learning, was investigated by [68] confirming these results. [53] addressed the rare word problem using sequence-to-sequence, which improves the ability to translate words not in the vocabulary. The architecture was further improved by [1] addressing issues related to the translation of long sentences by implementing an attention mechanism into the decoder.

11.2.4 Image processing

In 2012 BSLTM was applied to keyword spotting and mode detection distinguishing different types of content in handwritten documents, such as text, formulas, diagrams and figures, outperforming HMMs and SVMs [44, 45, 59]. At approximately the same period of time [51] investigated the classification of high-resolution images from the ImageNet database with considerable better results then previous approaches. In 2015 the more recent LSTM variant using the Sequence-to-Sequence framework was successfully trained by [73, 79] to generate natural sentences in plain English describing images. Also in 2015 [14] the authors combined LSTMs with a deep hierarchical visual feature extractor and applied the model to image interpretation and classification tasks, like activity recognition and image/video description.

11.3 Other learning tasks

Early papers applied LSTM-RNN to a number of real world problems pushing its evolution further. Covered problems include protein secondary structure prediction [40, 9] and music generation [15]. Network security was covered in [65, 66] were the authors apply LSTM-RNN to the DARPA intrusion detection dataset.

In [80, 70] the authors apply computational tasks to LSTM-RNN. In 2014 the authors of [80] evaluate short computer programs using the Sequence-to-Sequence framework. One year later the authors of [70] use a modified version of the framework to learn solutions of combinatorial optimisation problems.

12 Conclusions

In this article, we covered the derivation of LSTM in detail, summarising the most relevant literature. Specifically, we highlighted the vanishing error problem, which is a serious shortcoming of RNNs. LSTM provides a possible solution to this problem by introducing a constant error flow through the internal states of special memory cells. In this way, LSTM is able to tackle long time-lag problems, bridging time intervals in excess of 1,000 time steps. Finally, we introduced two LSTM extensions that enable LSTM to learn self-resets and precise timing. With self-resets, LSTM is able to free memory of irrelevant information.

Acknowledgements

This work was mainly pushed as a private project from Ralf C. Staudemeyer spanning a period of ten years from 2007–17. During the time 2013–15 it was partially supported by post-doctoral fellowship research funds provided by the South African National Research Foundation, Rhodes University, the University of South Africa, and the University of Passau. The co-author Eric Rothstein Morris picked-up the loose ends, developed the unified notation for this article in 2015–16.

We acknowledge support for this work from Ralf's Ph.D. supervisor Christian W. Omlin for raising the authors interest to investigate the capabilities of Long Short-Term Memory Recurrent Neural Networks. Very special thanks go to Arne Janza for doing the internal review. Without his dedicated support to eliminate a number of hard to find logical inconsistencies this publication would not have found its way to the reader.

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Proc. of the Int. Conf. on Learning Representations (ICLR 2015), volume 26, page 15, sep 2015.

[2] Bram Bakker. Reinforcement learning with long short-term memory. In Advances in Neural Information Processing Systems (NIPS'02), 2002.