– 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.

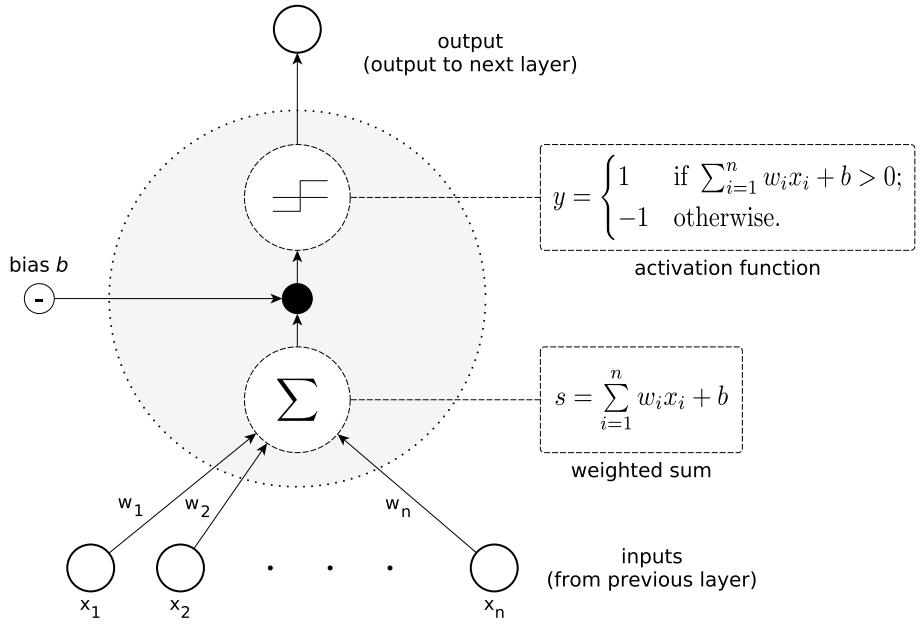
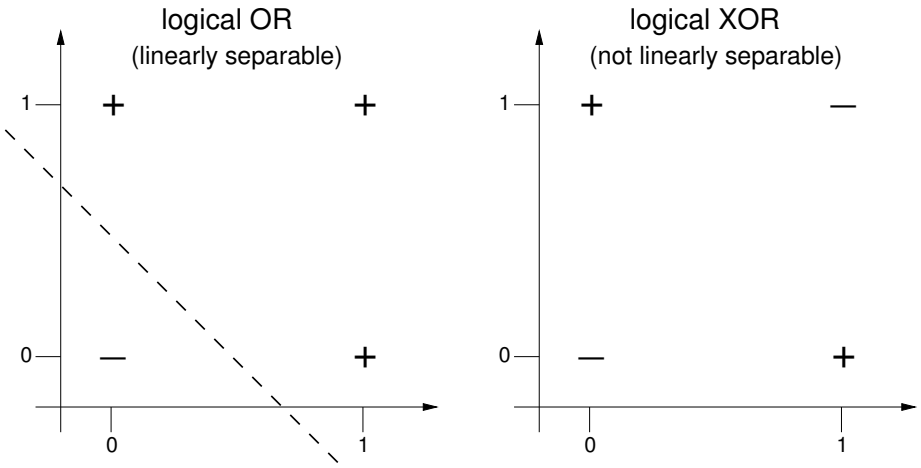


Figure 1: The general structure of the most basic type of artificial neuron, called a perceptron. Single perceptrons are limited to learning linearly separable functions.

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.

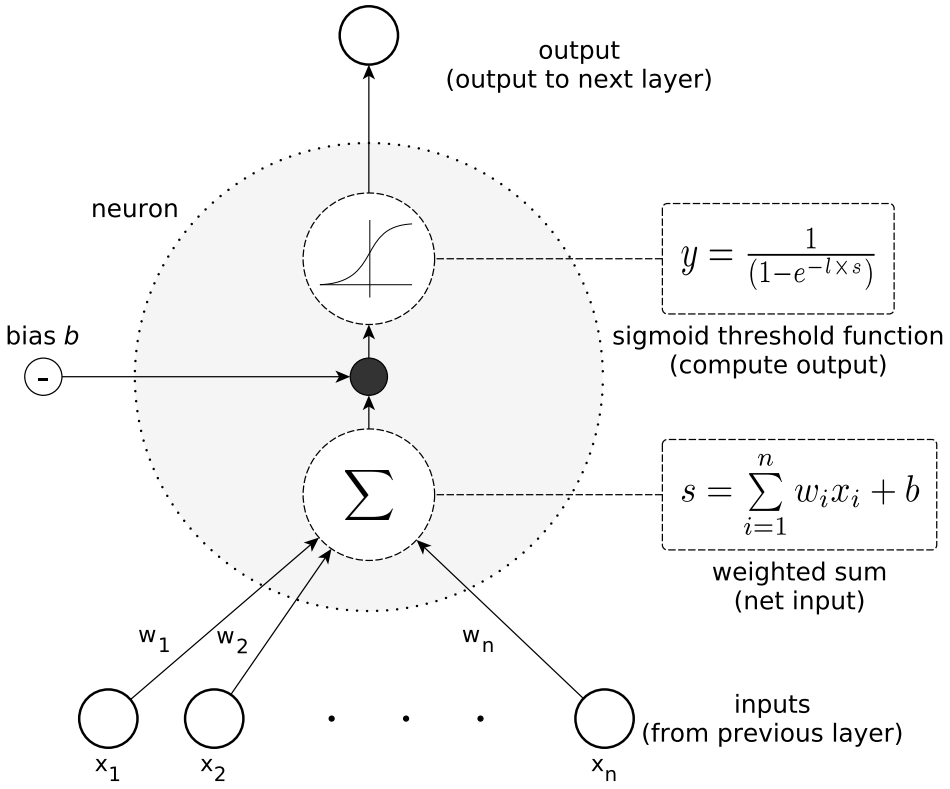


Figure 3: The sigmoid threshold unit is capable of representing non-linear functions. Its output is a continuous function of its input, which ranges between 0 and 1.

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.

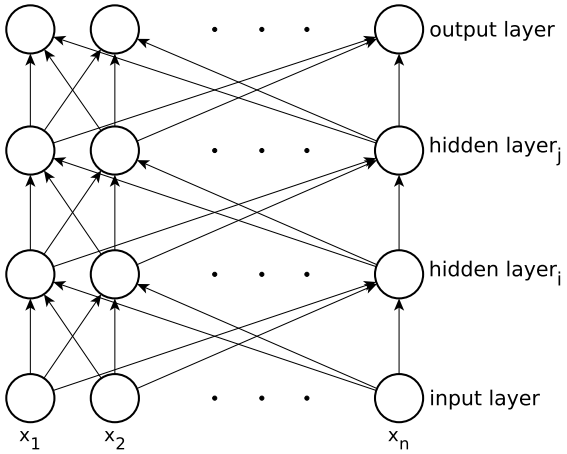


Figure 4: A multilayer feed-forward neural network with one input layer, two hidden layers, and an output layer. Using neurons with sigmoid threshold functions, these neural networks are able to express non-linear decision surfaces.

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

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

where $\texttt{Suc}\,(h)$ 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

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

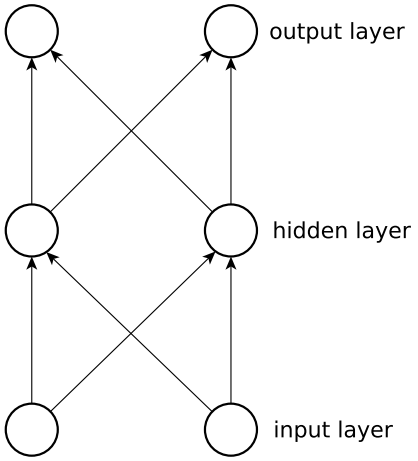


Figure 5: This figure shows a feed-forward neural network.

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.

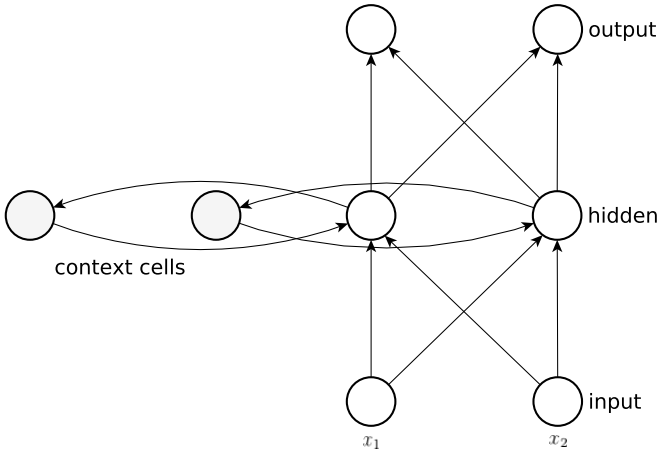


Figure 6: This figure shows an Elman neural network.

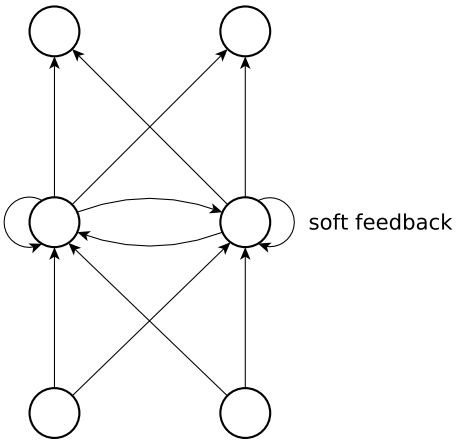


Figure 7: This figure shows a partially recurrent neural network with self-feedback in the hidden layer.

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

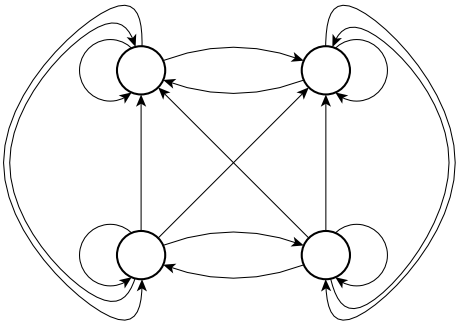


Figure 8: This figure shows a fully recurrent neural network (RNN) with self-feedback connections.

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

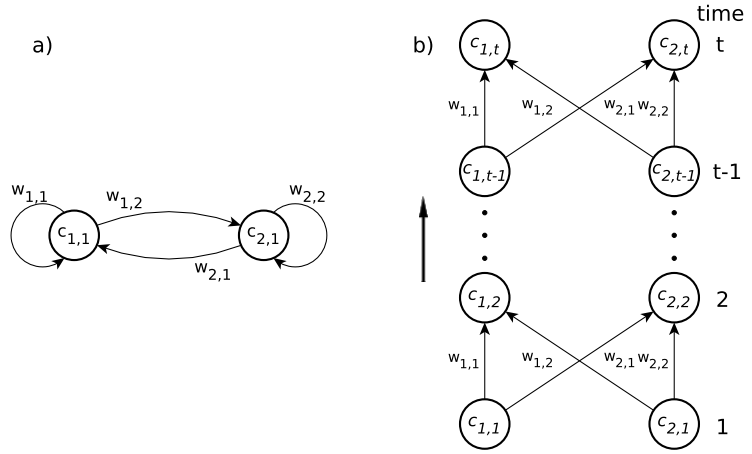


Figure 9: Figure a shows a simple fully recurrent neural network with a two-neuron layer. The same network unfolded over time with a separate layer for each time step is shown in Figure b. The latter representation is a feed-forward neural network.

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

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

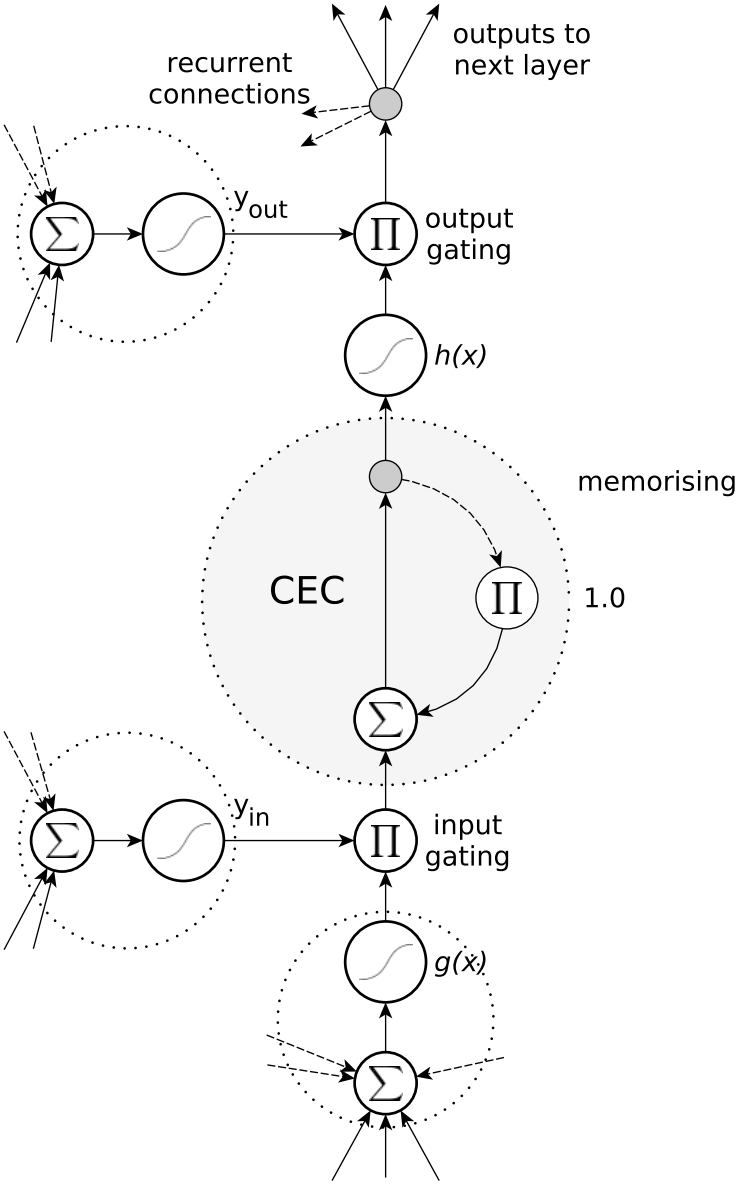


Figure 10: A standard LSTM memory block. The block contains (at least) one cell with a recurrent self-connection (CEC) and weight of '1'. The state of the cell is denoted as s_c. Read and write access is regulated by the input gate, y_{in}, and the output gate, y_{out}. The internal cell state is calculated 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}.

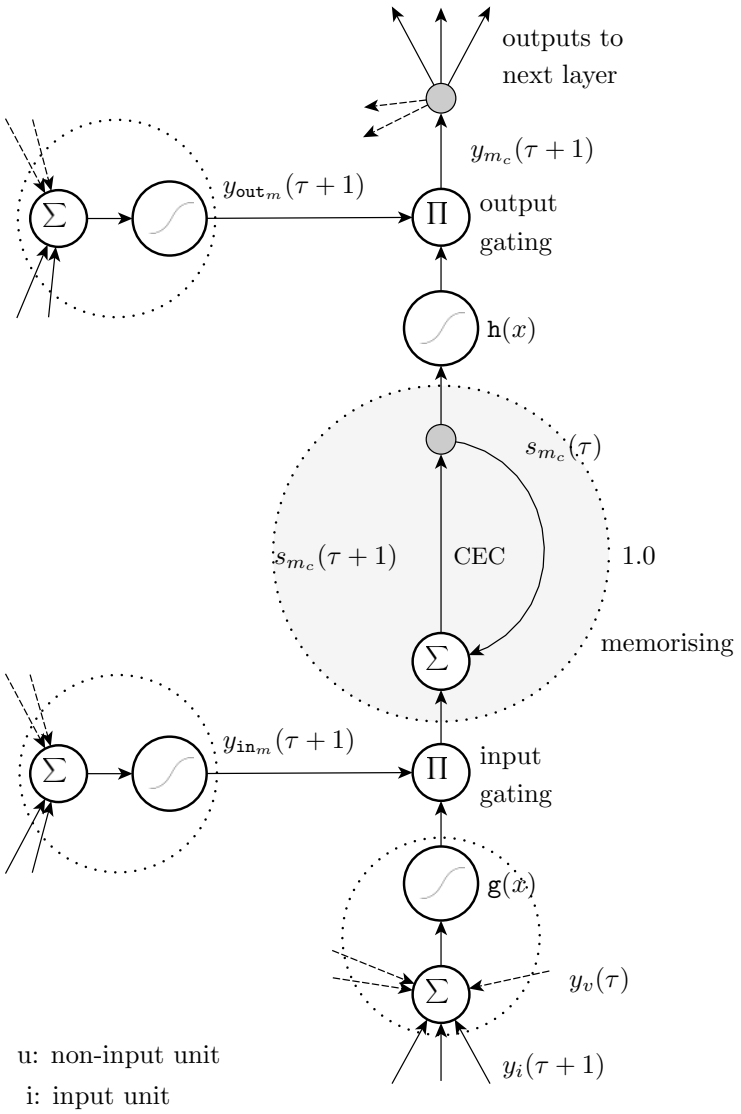


Figure 11: A standard LSTM memory block. The block contains (at least) one cell with a recurrent self-connection (CEC) and weight of '1'. The state of the cell is denoted as s_c. Read and write access is regulated by the input gate, $y_{\mathtt{in}}$, and the output gate, $y_{\mathtt{out}}$. The internal 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.

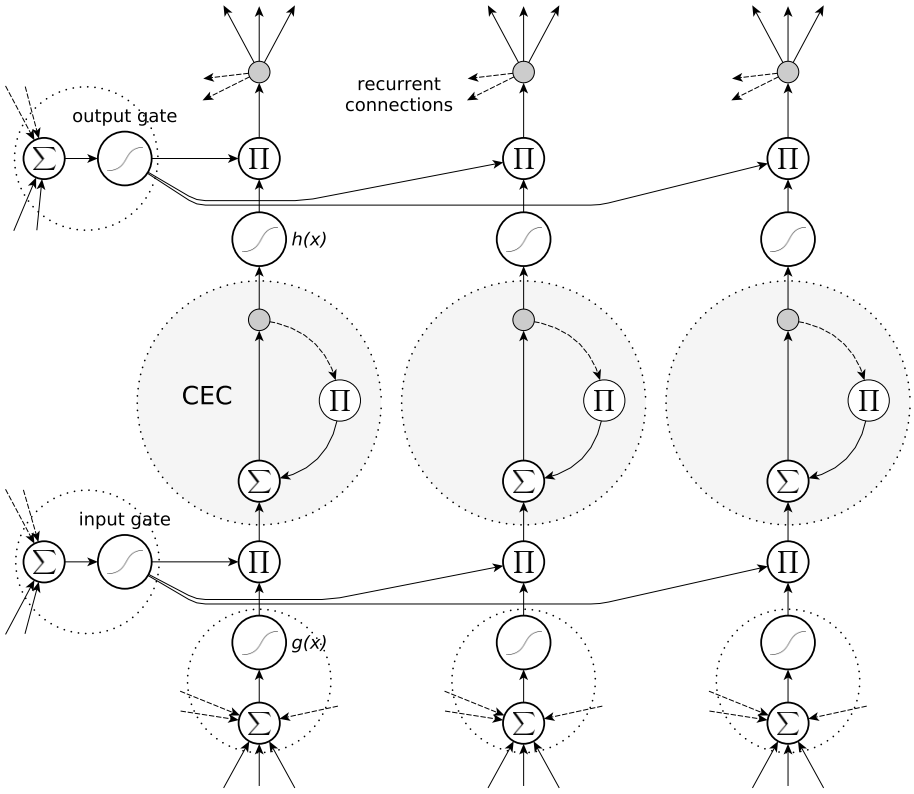


Figure 12: A three cell LSTM memory block with recurrent self-connections

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 $\mathtt{out}_m$ of the memory block m, then

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

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

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

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

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

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

$$X = \begin{bmatrix} P\vec{x} \\ \vec{y_h}_1 \\ \vdots \\ \vec{y_h}_N \end{bmatrix} \tag{53}$$

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

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

where $\odot$ is the Hadamard product, $\vec{\varphi}$ is a vector consisting of N forget signals (one for each $\vec{y_h}_i$), and $\vec{\texttt{in}}_m$ 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 $\vec{y_h}'_1, \ldots, \vec{y_h}'_N$ and N memory vectors $\vec{s_m}'_1, \ldots, \vec{s_m}'_N$ that are all distinct. To do so, the model concatenates the hidden vectors from the N dimensions as follows

$$X = \begin{bmatrix} \vec{y_h}_1 \\ \vdots \\ \vec{y_h}_N \end{bmatrix} \tag{55}$$

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

$$\begin{aligned} (X, \vec{s_{m1}}) &\xrightarrow{W_1} (\vec{y_h}'_1, \vec{s_m}'_1) \\ &\vdots \\ (X, \vec{s_{mN}}) &\xrightarrow{W_N} (\vec{y_h}'_N, \vec{s_m}'_N) \end{aligned} \tag{56}$$

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 `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.

[3] Justin Bayer, Daan Wierstra, Julian Togelius, and Jürgen Schmidhuber. Evolving memory cell structures for sequence learning. In Int. Conf. on Artificial Neural Networks, pages 755–764, 2009.

[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A Neural Probabilistic Language Model. The Journal of Machine Learning Research, 3:1137–1155, 2003.

[5] Yoshua Bengio, Patrice Simard, Paolo Frasconi, and Paolo Frasconi Yoshua Bengio, Patrice Simard. Learning long-term dependencies with gradient descent is difficult. IEEE trans. on Neural Networks / A publication of the IEEE Neural Networks Council, 5(2):157–66, jan 1994.

[6] N Beringer, A Graves, F Schiel, and J Schmidhuber. Classifying unprompted speech by retraining LSTM Nets. In W Duch, J Kacprzyk, E Oja, and S Zadrozny, editors, Artificial Neural Networks: Biological Inspirations (ICANN), volume 3696 LNCS, pages 575–581. Springer-Verlag Berlin Heidelberg, 2005.

[7] Hervé A. Bourlard and Nelson Morgan. Connectionist Speech Recognition - a hybrid approach. Kluwer Academic Publishers, Boston, MA, 1994.

[8] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. Listen, Attend and Spell. arXiv preprint, pages 1–16, aug 2015.

[9] Jinmiao Chen and Narendra S. Chaudhari. Capturing Long-Term Dependencies for Protein Secondary Structure Prediction. In Fu-Liang Yin, Jun Wang, and Chengan Guo, editors, Advances in Neural Networks - Proc. of the Int. Symp. on Neural Networks (ISNN 2004), pages 494–500, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[10] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Proc. of the Conf. on Empirical Methods in Natural Language Processing (EMNLP'14), pages 1724–1734, Stroudsburg, PA, USA, 2014. Association for Computational Linguistics.

[11] Jan Chorowski, Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. End-to-end continuous speech recognition using attention-based recurrent NN: first results. Deep Learning and Representation Learning Workshop (NIPS 2014), pages 1–10, dec 2014.

[12] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In C Cortes, N D Lawrence, D D Lee, M Sugiyama, and R Garnett, editors, Advances in Neural Information Processing Systems 28, pages 577–585. Curran Associates, Inc., jun 2015.

[13] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In arXiv, pages 1–9, dec 2014.

[14] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, Trevor Darrell, U T Austin, Umass Lowell, U C Berkeley, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-Term Recurrent Convolutional Networks for Visual Recognition and Description. In Proc. of the Conf. on Computer Vision and Pattern Recognition (CVPR'15), pages 2625–2634, jun 2015.

[15] Douglas Eck and Jürgen Schmidhuber. Finding temporal structure in music: Blues improvisation with LSTM recurrent networks. In Proc. of the 12th Workshop on Neural Networks for Signal Processing, pages 747–756. IEEE, IEEE, 2002.

[16] Jeffrey L. Elman. Finding structure in time. Cognitive Science, 14(2):179–211, mar 1990.

[17] Florian Eyben, Martin Wollmer, Bjorn Schuller, and Alex Graves. From speech to letters - using a novel neural network architecture for grapheme based ASR. In Workshop on Automatic Speech Recognition & Understanding, pages 376–380. IEEE, dec 2009.

[18] Santiago Fernandez, Alex Graves, and Jürgen Schmidhuber. Sequence labelling in structured domains with hierarchical recurrent neural networks. In Proc. of the 20th Int. Joint Conf. on Artificial Intelligence (IJCAI'07), pages 774–779, 2007.

[19] Santiago Fernández, Alex Graves, and Jürgen Schmidhuber. Phoneme recognition in TIMIT with BLSTM-CTC. Arxiv preprint arXiv08043269, abs/0804.3:8, 2008.

[20] T Geiger, Zixing Zhang, Felix Weninger, Gerhard Rigoll, Jürgen T Geiger, Zixing Zhang, Felix Weninger, Björn Schuller, and Gerhard Rigoll. Robust speech recognition using long short-term memory recurrent neural networks for hybrid acoustic modelling. Proc. of the Ann. Conf. of International Speech Communication Association (INTERSPEECH 2014), (September):631–635, 2014.

[21] Felix A. Gers and Jürgen Schmidhuber. LSTM recurrent networks learn simple context-free and context-sensitive languages. IEEE Trans. on Neural Networks, 12(6):1333–1340, jan 2001.

[22] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. Neural Computation, 12(10):2451–2471, oct 2000.

[23] Felix A. Gers, Nicol N. Schraudolph, and Jürgen Schmidhuber. Learning precise timing with LSTM recurrent networks. Journal of Machine Learning Research (JMLR), 3(1):115–143, 2002.

[24] Alex Graves. Generating sequences with recurrent neural networks. Proc. of the 23rd Int. Conf. on Information and Knowledge Management (CIKM '14), pages 101–110, aug 2014.

[25] Alex Graves, Nicole Beringer, and Jürgen Schmidhuber. A comparison between spiking and differentiable recurrent neural networks on spoken digit recognition. In Proc. of the 23rd IASTED Int. Conf. on Modelling, Identification, and Control, Grindelwald, 2003.

[26] Alex Graves, Nicole Beringer, and Jürgen Schmidhuber. Rapid retraining on speech data with LSTM recurrent networks. Technical Report IDSIA-09-05, IDSIA, 2005.

[27] Alex Graves, S Fernández, and Marcus Liwicki. Unconstrained online handwriting recognition with recurrent neural networks. Neural Information Processing Systems (NIPS'07), 20:577–584, 2007.

[28] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In Proc. of the 23rd Int. Conf. on Machine Learning (ICML'06), number January, pages 369–376, New York, New York, USA, 2006. ACM Press.

[29] Alex Graves, Santiago Fernandez, and Jürgen Schmidhuber. Multi-Dimensional Recurrent Neural Networks. Proc. of the Int. Conf. on Artificial Neural Networks (ICANN'07), 4668(1):549–558, may 2007.

[30] Alex Graves and Navdeep Jaitly. Towards End-To-End Speech Recognition with Recurrent Neural Networks. JMLR Workshop and Conference Proceedings, 32(1):1764–1772, 2014.

[31] Alex Graves, Navdeep Jaitly, and Abdel Rahman Mohamed. Hybrid speech recognition with Deep Bidirectional LSTM. In Proc. of the workshop on Automatic Speech Recognition and Understanding (ASRU'13), pages 273–278, 2013.

[32] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. IEEE trans. on Pattern Analysis and Machine Intelligence, 31(5):855–68, may 2009.

[33] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP'13), number 3, pages 6645–6649. IEEE, may 2013.

[34] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM networks. In Proc. of the Int. Joint Conf. on Neural Networks, volume 18, pages 2047–2052, Oxford, UK, UK, jun 2005. Elsevier Science Ltd.

[35] Alex Graves and Jürgen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In Advances in Neural Information Processing Systems 21 (NIPS'09), pages 545–552. MIT Press, 2009.

[36] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. Arxiv, pages 1–26, 2014.

[37] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. ArXiv e-prints, pages 1–18, 2012.

[38] Josef Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Master's thesis, Institut fur Informatik, Technische Universitat, Munchen, (April 1991):1–71, 1991.

[39] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. A Field Guide to Dynamical Recurrent Neural Networks, page 15, 2001.

[40] Sepp Hochreiter, Martin Heusel, and Klaus Obermayer. Fast model-based protein homology detection without alignment. Bioinformatics, 23(14):1728–1736, jul 2007.

[41] Sepp; Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. Neural computation, 9(8):1735–1780, 1997.

[42] Sepp Hochreiter and Jürgen Schmidhuber. LSTM can solve hard long time lag problems. Neural Information Processing Systems, pages 473–479, 1997.

[43] Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In Georg Dorffner, Horst Bischof, and Kurt Hornik, editors, Proc. of the Int. Conf. of Artificial Neural Networks (ICANN 2001), pages 87–94, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[44] Emanuel Indermuehle, Volkmar Frinken, Andreas Fischer, and Horst Bunke. Keyword spotting in online handwritten documents containing text and non-text using BLSTM neural networks. In Int. Conf. on Document Analysis and Recognition (ICDAR), pages 73–77. IEEE, 2011.

[45] Emanuel Indermuhle, Volkmar Frinken, and Horst Bunke. Mode detection in online handwritten documents using BLSTM neural networks. In Int. Conf. on Frontiers in Handwriting Recognition (ICFHR), pages 302–307. IEEE, 2012.

[46] Michael I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In Proc. of the Eigth Annual Conf. of the Cognitive Science Society, pages 531–546. IEEE Press, jan 1986.

[47] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of Recurrent Network Architectures. In Proc. of the 32nd Int. Conf on Machine Learning, pp. 23422350, 2015, pages 2342—-2350, 2015.

[48] Nal Kalchbrenner and Phil Blunsom. Recurrent Continuous Translation Models. In Proc. of the Conf. on Empirical Methods in Natural Language Processing (EMNLP'13), volume 3, page 413. Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, 2013.

[49] Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid Long Short-Term Memory. In arXiv preprint, page 14, jul 2016.

[50] Jan Koutnik, Klaus Greff, Faustino Gomez, and Jürgen Schmidhuber. A Clockwork RNN. In Proc. of the 31st Int. Conf. on Machine Learning (ICML 2014), volume 32, pages 1863–1871, 2014.

[51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F Pereira, C J C Burges, L Bottou, and K Q Weinberger, editors, Advances in Neural Information Processing Systems 25, pages 1–9. Curran Associates, Inc., 2012.

[52] M Liwicki, A Graves, H Bunke, and J Schmidhuber. A novel approach to on-line handwriting recognition based on bidirectional long short-term memory networks. In Proc. of the 9th Int. Conf. on Document Analysis and Recognition, pages 367{—-}371, 2007.

[53] Minh-Thang Luong, Ilya Sutskever, Quoc V. Le, Oriol Vinyals, and Wojciech Zaremba. Addressing the rare word problem in neural machine translation. Arxiv, pages 1–11, 2014.

[54] Qi Lyu and Jun Zhu. Revisit long short-term memory: an optimization perspective. In Deep Learning and Representation Learning Workshop (NIPS 2014), pages 1–9, 2014.

[55] Marvin L. Minsky and Seymour A. Papert. Perceptrons: An introduction to computational geometry. Expanded. MIT Press, Cambridge, 1988.

[56] Michael C Mozer. Induction of Multiscale Temporal Structure. In Advances in Neural Information Processing Systems 4, pages 275–282. Morgan Kaufmann, 1992.

[57] Thibauld Nion, Fares Menasri, Jerome Louradour, Cedric Sibade, Thomas Retornaz, Pierre Yves Metaireau, and Christopher Kermorvant. Handwritten information extraction from historical census documents. Proc. of the Int. Conf. on Document Analysis and Recognition (ICDAR 2013), pages 822–826, 2013.

[58] Randall C O'Reilly and Michael J Frank. Making working memory work: a computational model of learning in the prefrontal cortex and basal ganglia. Neural Computation, 18(2):283–328, feb 2006.

[59] Sebastian Otte, Dirk Krechel, Marcus Liwicki, and Andreas Dengel. Local Feature Based Online Mode Detection with Recurrent Neural Networks. Int. Conf. on Frontiers in Handwriting Recognition, pages 533–537, 2012.

[60] JA A Pérez-Ortiz, FA A Felix A. Gers, Douglas Eck, J??rgen U. Schmidhuber, Juan Antonio P??rez-Ortiz, FA A Felix A. Gers, Douglas Eck, and J??rgen U. Schmidhuber. Kalman filters improve LSTM network performance in problems unsolvable by traditional recurrent nets. Neural Networks, 16(2):241–250, 2003.

[61] Vu Pham, Théodore Theodore Théodore Bluche, Christopher Kermorvant, and Jérôme Jerome Jérôme Louradour. Dropout improves recurrent neural networks for handwriting recognition. In Proc. of the Int. Conf. on Frontiers in Handwriting Recognition (ICFHR'13), volume 2014-Decem, pages 285–290. IEEE, nov 2013.

[62] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In J L McClelland and D E Rumelhart, editors, Parallel distributed processing: Explorations in the microstructure of cognition, volume 1, pages 318–362. MIT Press, jan 1985.

[63] Haim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory based recurrent neural network architectures for large scale acoustic modeling. In Interspeech 2014, number September, pages 338–342, feb 2014.

[64] Nitisch Srivastava. Improving Neural Networks with Dropout. PhD thesis, University of Toronto, 2013.

[65] Ralf C. Staudemeyer. The importance of time: Modelling network intrusions with long short-term memory recurrent neural networks. PhD thesis, 2012.

[66] Ralf C. Staudemeyer. Applying long short-term memory recurrent neural networks to intrusion detection. South African Computer Journal, 56(1):136–154, jul 2015.

[67] Ilya Sutskever, James Martens, and Geoffrey E. Hinton. Generating text with recurrent neural networks. In Proc. of the 28th Int. Conf. on Machine Learning (ICML-11)., pages 1017–1024, 2011.

[68] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to Sequence learning with neural networks. Advances in Neural Information Processing Systems (NIPS'14), pages 3104–3112, sep 2014.

[69] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order Matters: Sequence to Sequence for sets. In Proc. of the 4th Int. Conf. on Learning Representations (ICLR'17), pages 1–11, 2016.

[70] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer Networks. Neural Information Processing Systems 2015, pages 1–9, 2015.

[71] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In C Cortes, N D Lawrence, D D Lee, M Sugiyama, and R Garnett, editors, Advances in Neural Information Processing Systems 28, pages 2773–2781. Curran Associates, Inc., dec 2014.

[72] Oriol Vinyals and Quoc V. Le. A neural conversational model. arXiv, 37, jun 2015.

[73] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and Tell: A Neural Image Caption Generator. In Conf. on Computer Vision and Pattern Recognition (CVPR'15), pages 3156–3164, jun 2015.

[74] Paul J. Werbos. Backpropagation through time: What it does and how to do it. Proc. of the IEEE, 78(10):1550–1560, 1990.

[75] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. Neural Computation, 1(2):270–280, jun 1989.

[76] Ronald J. Williams and David Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. In Back-propagation: Theory, Architectures and Applications, pages 1–45. L. Erlbaum Associates Inc., jan 1995.

[77] Martin Woellmer, Björn Schuller, and Gerhard Rigoll. Keyword spotting exploiting Long Short-Term Memory. Speech Communication, 55(2):252–265, feb 2013.

[78] Martin Wöllmer, Florian Eyben, Stephan Reiter, Björn Schuller, Cate Cox, Ellen Douglas-Cowie, and Roddy Cowie. Abandoning emotion classes - Towards continuous emotion recognition with modelling of long-range dependencies. In Proc. of the Ann. Conf. of the Int. Speech Communication Association (INTERSPEECH'08), number January, pages 597–600, 2008.

[79] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, Attend and Tell: Neural image caption generation with visual attention. IEEE Transactions on Neural Networks, 5(2):157–166, feb 2015.

[80] Wojciech Zaremba and Ilya Sutskever. Learning to Execute. arXiv preprint, pages 1–25, oct 2014.

[81] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent Neural Network Regularization. Icrl, (2013):1–8, sep 2014.

Gate-Variants of Gated Recurrent Unit (GRU) Neural Networks

Rahul Dey and Fathi M. Salem
Circuits, Systems, and Neural Networks (CSANN) LAB
Department of Electrical and Computer Engineering
Michigan State University
East Lansing, MI 48824-1226, USA
deyrahul@msu.edu || salemf@msu.edu

***Abstract* – The paper evaluates three variants of the Gated Recurrent Unit (GRU) in recurrent neural networks (RNN) by reducing parameters in the update and reset gates. We evaluate the three variant GRU models on MNIST and IMDB datasets and show that these GRU-RNN variant models perform as well as the original GRU RNN model while reducing the computational expense.**

I. INTRODUCTION

Gated Recurrent Neural Network (RNN) have shown success in several applications involving sequential or temporal data [1-13]. For example, they have been applied extensively in speech recognition, natural language processing, machine translation, etc. [2, 5]. Long Short-Term Memory (LSTM) RNN and the recently introduced Gated Recurrent Unit (GRU) RNN have been successfully shown to perform well with long sequence applications [2-5, 8-12].

Their success is primarily due to the gating network signals that control how the present input and previous memory are used to update the current activation and produce the current state. These gates have their own sets of weights that are adaptively updated in the learning phase (i.e., the training and evaluation process). While these models empower successful learning in RNN, they introduce an increase in parameterization through their gate networks. Consequently, there is an added computational expense vis-à-vis the simple RNN model [2, 5, 6]. It is noted that the LSTM RNN employs 3 distinct gate networks while the GRU RNN reduce the gate networks to two. In [14], it is proposed to reduce the external gates to the minimum of one with preliminary evaluation of sustained performance.

In this paper, we focus on the GRU RNN and explore three new gate-variants with reduced parameterization. We comparatively evaluate the performance of the original and the variant GRU RNN on two public datasets. Using the MNIST dataset, one generates two sequences [2, 5, 6, 14]. One sequence is obtained from each 28x28 image sample as pixel-wise long sequence of length 28x28=784 (basically, scanning from the upper left to the bottom right of the image). Also, one generates a row-wise short sequence of length 28, with each element being a vector of dimension 28 [14, 15]. The third sequence type employs the IMDB movie review dataset where one defines the length of the sequence in order to achieve high performance sentiment classification from a given review paragraph.

II. BACKGROUND: RNN, LSTM AND GRU

In principal, RNN are more suitable for capturing relationships among sequential data types. The so-called simple RNN has a recurrent hidden state as in

$$h_t = g(Wx_t + Uh_{t-1} + b) \qquad (1)$$

where x_t is the (external) *m-dimensional* input vector at time t, h_t the *n-dimensional* hidden state, g is the (point-wise) activation function, such as the logistic function, the hyperbolic tangent function, or the rectified Linear Unit (ReLU) [2, 6], and W, U *and* b are the appropriately sized parameters (two weights and bias). Specifically, in this case, W is an $n \times m$ matrix, U is an $n \times n$ matrix, and b is an $n \times 1$ matrix (or vector).

Bengio at al. [1] showed that it is difficult to capture long-term dependencies using such simple RNN because the (stochastic) gradients tend to either vanish or explode with long sequences. Two particular models, the Long Short-Term Memory (LSTM) unit RNN [3, 4] and Gated Recurrent Unit (GRU) RNN [2] have been proposed to solve the "vanishing" or "exploding" gradient problems. We will present these two models in sufficient details for our purposes below.

A. Long Short-Term Memory (LSTM) RNN

The LSTM RNN architecture uses the computation of the simple RNN of Eqn (1) as an intermediate candidate for the internal memory cell (state), say $\tilde{c}_t$, and add it in a (element-wise) weighted-sum to the previous value of the internal memory state, say c_{t-1}, to produce the current value of the memory cell (state) c_t. This is expressed succinctly in the following discrete dynamic equations:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \qquad (2)$$
$$\tilde{c}_t = g(W_c x_t + U_c h_{t-1} + b_c) \qquad (3)$$
$$h_t = o_t \odot g(c_t) \qquad (4)$$

In Eqns (3) and (4), the activation nonlinearity g is typically the hyperbolic tangent function but more recently may be implemented as a rectified Linear Unit (reLU). The weighted

sum is implemented in Eqn (2) via element-wise (Hadamard) multiplication denoted by $\odot$ to gating signals. The gating (control) signals $i_t,\ f_t$ *and* o_t denote, respectively, the *input*, *forget*, and *output* gating signals at time t. These control gating signals are in fact replica of the basic equation (3), with their own parameters and replacing g by the logistic function. The logistic function limits the gating signals to within 0 and 1. The specific mathematical form of the gating signals are thus expressed as the vector equations:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
$$f_t = \sigma\big(W_f x_t + U_f h_{t-1} + b_f\big)$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

where σ is the logistic nonlinearity and the parameters for each gate consist of two matrices and a bias vector. Thus, the total number of parameters (represented as matrices and bias vectors) for the 3 gates and the memory cell structure are, respectively, $W_i, U_i, b_i, W_f, U_f, b_f, W_o, U_o, b_o, W_c, U_c$ *and* b_c. These parameters are all updated at each training step and stored. It is immediately noted that the number of parameters in the LSTM model is increased 4-folds from the simple RNN model in Eqn (1). Assume that the cell state is *n-dimensional*. (Note that the activation and all the gates have the same dimensions). Assume also that the input signal is *m-dimensional*. Then, the total parameters in the LSTM RNN are equal to $4\times(n^2 + nm + n)$.

B. Gated Recurrent Unit (GRU) RNN

The GRU RNN reduce the gating signals to two from the LSTM RNN model. The two gates are called an update gate z_t and a reset gate r_t. The GRU RNN model is presented in the form:

$$h_t = (1 - z_t)\odot h_{t-1} + z_t \odot \tilde{h}_t \qquad (5)$$
$$\tilde{h}_t = g(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \qquad (6)$$

with the two gates presented as:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \qquad (7)$$
$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \qquad (8)$$

One observes that the GRU RNN [Eqns (5)-(6)] is similar to the LSTM RNN [Eqns (2)-(3)], however with less external gating signal in the interpolation Eqn (5). This saves one gating signal and the associated parameters. We defer further information to reference [2], and the references therein. In essence, the GRU RNN has 3-folds increase in parameters in comparison to the simple RNN of Eqn (1). Specifically, the total number of parameters in the GRU RNN equals $3\times(n^2 + nm + n)$.

In various studies, e.g., in [2] and the references therein, it has been noted that GRU RNN is comparable to, or even outperforms, the LSTM in most cases. Moreover, there are other reduced gated RNNs, e.g. the Minimal Gated Unit (MGU) RNN, where only one gate equation is used and it is reported that this (MGU) RNN performance is comparable to the LSTM RNN and the GRU RNN, see [14] for details.

In this paper, we focus on the GRU RNN model and evaluate new variants. Specifically, we retain the architecture of Eqns (5)-(6) unchanged, and focus on variation in the structure of the gating signals in Eqns (7) and (8). We apply the variations identically to the two gates for uniformity and simplicity.

III. THE VARIANT GRU ARCHITECTURES

The gating mechanism in the GRU (and LSTM) RNN is a replica of the simple RNN in terms of parametrization. The weights corresponding to these gates are also updated using the backpropagation through time (BTT) stochastic gradient descent as it seeks to minimize a loss/cost function [3, 4]. Thus, each parameter update will involve information pertaining to the state of the overall network. Thus, all information regarding the current input and the previous hidden states are reflected in the latest state variable. There is a redundancy in the signals driving the gating signals. The key driving signal should be the internal state of the network. Moreover, the adaptive parameter updates all involve components of the internal state of the system [16, 17]. In this study, we consider three distinct variants of the gating equations applied uniformly to both gates:

Variant 1: called **GRU1**, where each gate is computed using only the previous hidden state and the bias.

$$z_t = \sigma(U_z h_{t-1} + b_z) \qquad (9-a)$$
$$r_t = \sigma(U_r h_{t-1} + b_r) \qquad (9-b)$$

Thus, the total number of parameters is now reduced in comparison to the GRU RNN by $2\times nm$.

Variant 2: called **GRU2**, where each gate is computed using only the previous hidden state.

$$z_t = \sigma(U_z h_{t-1}) \qquad (10-a)$$
$$r_t = \sigma(U_r h_{t-1}) \qquad (10-b)$$

Thus, the total number of parameters is reduced in comparison to the GRU RNN by $2\times (nm+n)$.

Variant 3: called **GRU3**, where each gate is computed using only the bias.

$$z_t = \sigma(b_z) \qquad (11-a)$$
$$r_t = \sigma(b_r) \qquad (11-b)$$

Thus the total number of parameters is reduced in comparison to the GRU RNN by $2\times (nm+n^2)$.

We have performed an empirical study of the performance of each of these variants as compared to the GRU RNN on, first, sequences generated from the MNIST dataset and then on the IMDB movie review dataset. In the subsequent figures and tables, we refer to the base GRU RNN model as GRU0 and the three variants as GRU1, GRU2, and GRU3 respectively.

Our architecture consists of a single layer of one of the variants of GRU units driven by the input sequence and the activation function g set as ReLU. (Initial experiments using

$g = tanh$ have produced similar results). For the MNIST dataset, we generate the pixel-wise and the row-wise sequences as in [15]. The networks have been generated in Python using the Keras library [15] with Theano as a backend library. As Keras has a GRU layer class, we modified this class to classes for GRU1, GRU2, and GRU3. All of these classes used the ReLU activation function. The RNN layer of units is followed by a softmax layer in the case of the MNIST dataset or a traditional logistic activation layer in the case of the IMDB dataset to predict the output category. The Root Mean Square Propagation (RMSprop) is used as the choice of optimizer that is known to adapt the learning rate for each of the parameters. To speed up training, we also decay the learning rate exponentially with the cost in each epoch

$$\eta = \eta_0 e^{cost} \qquad (12)$$

where η_0 represents a base constant learning rate and $cost$ is the cost computed in the previous epoch. The details of our models are delineated in Table I.

Table I: Network model characteristics

Model	**MNIST Pixel-wise**	**MNIST Row-wise**	**IMDB**
Hidden Units	100	100	128
Gate Activation	Sigmoid	Sigmoid	sigmoid
Activation	ReLU	ReLU	ReLU
Cost	Categorical Cross-entropy	Categorical Cross-entropy	Binary Cross-entropy
Epochs	100	50	100
Optimizer	RMSProp	RMSProp	RMSProp
Dropout	20%	20%	20%
Batch Size	32	32	32

IV. Results and Discussion

A. *Application to MNIST Dataset – pixel-wise sequences*

The MNIST dataset [15] consists of 60000 training images and 10000 test images, each of size 28x28 of handwritten digits. We evaluated our three variants against the original GRU model on the MNIST dataset by generating the sequential input in one case (pixel-wise, one pixel at a time) and in the second case (row-wise, one row at a time). The pixel-wise sequence generated from each image are 1-element signal of length 784, while the 28-element row-wise produces a sequence of length 28. For each case, we performed different iterations by varying the constant base learning rate η_0. The results of our experiments are depicted in Fig. 1, 2, and 3, with summary in Table II below.

Table II: MNIST pixel-wise sequences: performance summary of different architectures using 4 constant base learning rates η_0, in 100 epochs.

	Lr = 1e-3		Lr = 5e-4		1e-4	5e-5	# Params
Architecture	Train	Test	Train	Test	Train	Test	
GRU0	99.19	98.59	98.59	98.04	-	-	30600
GRU1	98.88	98.37	98.52	-	-	-	30400
GRU2	98.90	98.10	98.61	-	-	-	30200
GRU3	-	-	10.44	-	60.97	59.60	10400

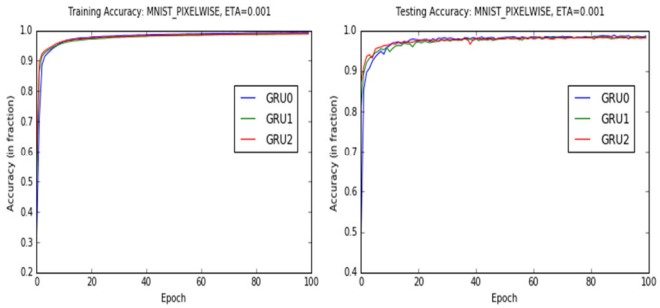


Fig. 1 Training (left) and Testing (right) Accuracy of GRU0, GRU1 and GRU2 on MNIST pixel-wise generated sequences at eta=0.001

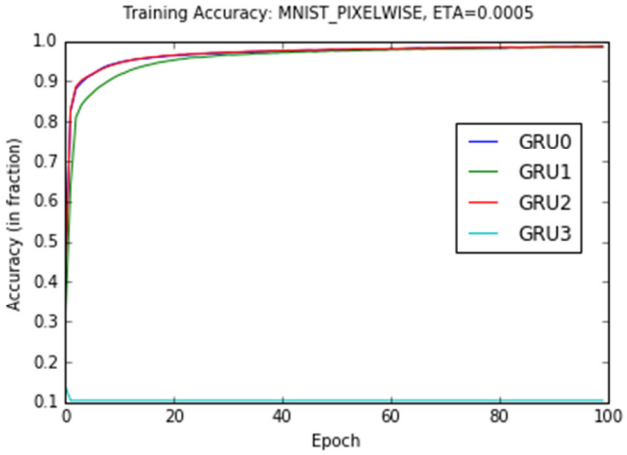


Fig. 2 Training Accuracy of GRU0, GRU1, GRU2 and GRU3 on MNIST generated sequences at eta=5e-4

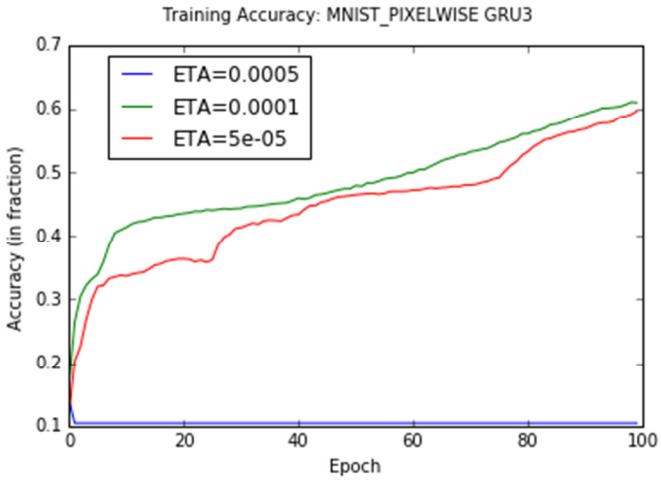


Fig. 3 Performance of GRU3 on MNIST generated sequences for 3 constant base learning rates η_0.

From Table II and Fig. 1 and 2, GRU1 and GRU2 perform almost as well as GRU0 on MNIST pixel-wise generated sequence inputs. While GRU3 does not perform as well for this (constant base) learning rate. Figure 3 shows that reducing the (constant base) learning rate to (0.0001) and below has enabled GRU3 to increase its (test) accuracy performance to 59.6% after 100 epochs, and with a positive slope indicating that it would increase further after more epochs. Note that in

this experiment, GRU3 has about 33% of the number of (adaptively computed) parameters compared to GRU0. Thus, there exists a potential trade-off between the higher accuracy performance and the decrease in the number of parameters. In our experiments, using 100 epochs, the GRU3 architecture never attains saturation. Further experiments using more epochs and/or more units would shed more light on the comparative evaluation of this trade-off between performance and parameter-reduction.

B. Application to MNIST Dataset – row-wise sequences

While pixel-wise sequences represent relatively long sequences, row-wise generated sequences can test short sequences (of length 28) with vector elements. The accuracy profile performance vs. epochs of the MNIST dataset with row-wise input of all four GRU RNN variants are depicted in Fig. 4, Fig. 5, and Fig. 6, using several constant base learning rates. Accuracy performance results are then summarized in Table III below.

Table III: MNIST row-wise generated sequences: Accuracy (%) performance of different variants using several constant base learning rates over 50 epochs

	Lr = 1e-2		Lr = 1e-3		Lr = 1e-4		# Params
Architecture	Train	Test	Train	Test	Train	Test	
GRU0	96.99	98.49	98.14	98.85	93.02	96.66	38700
GRU1	97.24	98.55	97.46	98.93	91.54	96.58	33100
GRU2	96.95	98.71	97.33	98.93	91.20	96.23	32900
GRU3	97.19	98.85	97.04	97.39	80.33	87.96	13100

Fig. 4 Training and testing accuracy on MNIST row-wise generated sequences at a constant base learning rate of 1e-2

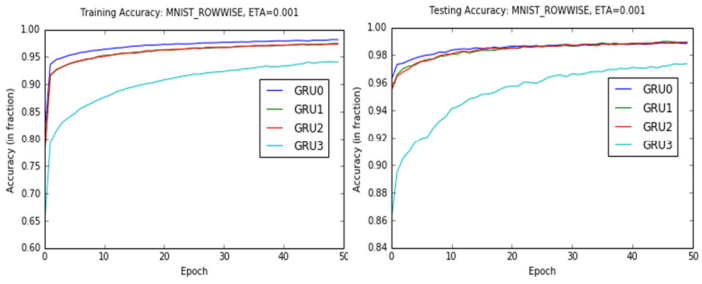


Fig. 5 Training and testing accuracy on MNIST row-wise generated sequences at a constant base learning rate of 1e-3

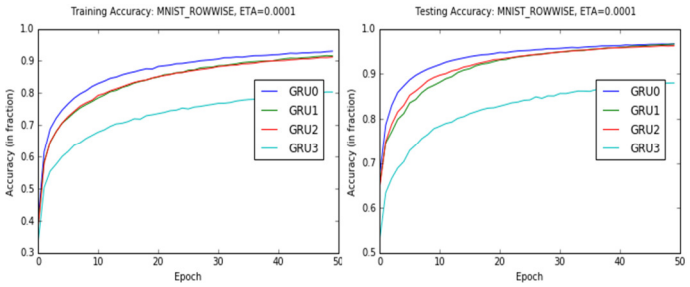


Fig. 6 Training and testing accuracy on MNIST row-wise generated sequences at a constant base learning rate of 1e-4

From Table III and Fig. 4, Fig.5, and Fig. 6, all the four variants GRU0, GRU1, GRU2, and GRU3 appear to exhibit comparable accuracy performance over three constant base learning rates. GRU3 exhibits lower performance at the base learning rate of 1e-4 where, after 50 epochs, is still lagging. From Fig. 6, however, it appears that the profile has not yet levelled off and has a positive slope. More epochs are likely to increase performance to comparable levels with the other variants. It is noted that in this experiment, GRU3 can achieve comparable performance with roughly one third of the number of (adaptively computed) parameters. Computational expense savings may play a role in favoring one variant over the others in targeted applications and/or available resources.

C. Application to the IMDB Dataset– natural sequence

The IMDB dataset is composed of 25000 test data and 25000 training data consisting of movie reviews and their binary sentiment classification. Each review is represented by a maximum of 80 (most frequently occurring) words in a vocabulary of 20000 words [7]. We have trained the dataset on all 4 GRU variants using the two constant base learning rates of 1e-3 and 1e-4 over 100 epochs. In the training, we employ *128-dimensional* GRU RNN variants and have adopted a batch size of 32. We have observed that, using the constant base learning rate of 1e-3, performance fluctuates visibly (see Fig. 7), whereas performance is uniformly progressing over profile-curves as shown in Fig. 8. Table IV summarizes the results of accuracy performance which show comparable performance among GRU0, GRU1, GRU2, and GRU3. Table IV also lists the number of parameters in each.

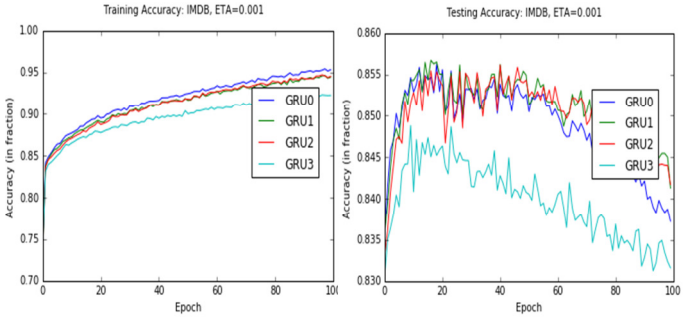


Fig. 7 Test and validation accuracy on IMDB dataset using a base learning rate of 1e-3

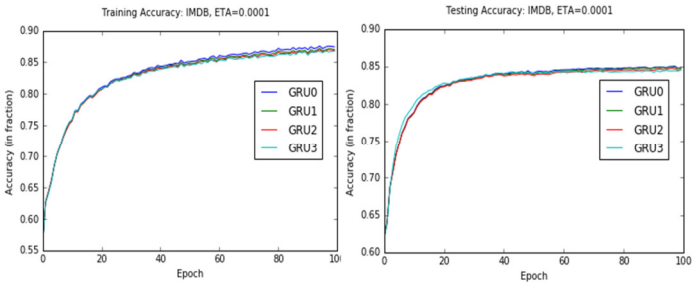


Fig. 8 Training and testing accuracy on IMDB dataset using a base learning rate of 1e-4

Table IV: IMDB dataset: Accuracy (%) performance of different architectures using two base learning rates over 100 epochs

	Lr = 1e-3		Lr = 1e-4		# Params
Architecture	Train	Test	Train	Test	
GRU0	95.3	83.7	87.4	84.8	98688
GRU1	94.5	84.1	87.0	84.8	65920
GRU2	94.5	84.2	86.9	84.6	65664
GRU3	92.3	83.2	86.8	84.5	33152

The IMDB data experiments provide the most striking results. It can be clearly seen that all the 3 GRU variants perform comparably to the GRU RNN while using less number of parameters. The learning pace of GRU3 was also similar to those of the other variants at the constant base learning rate of 1e-4. From Table IV, it is noted that more saving in computational load is achieved by all variant GRU RNN as the input is represented as a large 128-*dimensional* vector.

V. Conclusion

The experiments on the variants GRU1, GRU2, and GRU3 verse the GRU RNN have demonstrated that their accuracy performance is comparable on three example sequence lengths. Two sequences generated from the MNIST dataset and one from the IMDB dataset. The main driving signal of the gates appear to be the (recurrent) state as it contains essential information about other signals. Moreover, the use of the stochastic gradient descent implicitly carries information about the network state. This may explain the relative success in using the bias alone in the gate signals as its adaptive update carries information about the state of the network. The GRU variants reduce this redundancy and thus their performance has been comparable to the original GRU RNN. While GRU1 and GRU2 have indistinguishable performance from the GRU RNN, GRU3 frequently lags in performance, especially for relatively long sequences and may require more execution time to achieve comparable performance,

We remark that the goal of this work to comparatively evaluate the performance of GRU1, GRU2 and GRU3, which possess less gate parameters, and thus less computational expense, than the original GRU RNN. By performing more experimental evaluations using constant or varying learning rates, and training for longer number of epochs, one can validate the performance on broader domain. We remark that the three GRU RNN variants need to be further comparatively evaluated on diverse datasets for a broader empirical performance evidence.

References

[1] Bengio, Y., Simard, P., and Frasconi, P. Learning Longterm Dependencies with Gradient Descent is Difficult. IEEE Trans.Neural Networks, 5(2):157–166, 1994. H. Simpson, *Dumb Robots*, 3rd ed., Springfield: UOS Press, 2004, pp.6-9.

[2] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. arXiv preprint arXiv:1412.3555, 2014. Gers, F. A., Schraudolph, N. N., and Schmidhuber, J. Learning Precise Timing with LSTM Recurrent Networks. Journal of Machine Learning Research, 3:115–143, 2002. J.-G. Lu, "Title of paper with only the first word capitalized," *J. Name Stand. Abbrev.*, in press.

[3] Hochreiter, S. and Schmidhuber, J. Long Short-Term Memory. Neural Computation, 9(8):1735–1780, 1997

[4] Jozefowicz, R., Zaremba, W., and Sutskever, I. An Empirical Exploration of Recurrent Network Architectures. In Proc., Int'l Conf. on Machine Learning, pp. 2342–2350, 2015.

[5] Le, Q. V., Jaitly, N., and Hinton, G. E. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. arXiv preprint arXiv:1504.00941, 2015.

[6] Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. Learning Word Vectors for Sentiment Analysis. In Proc. 49th Annual Meeting of the ACL, pp. 142–150, 2011.

[7] Mikolov, T., Joulin, A., Chopra, S., Mathieu, M., and Ranzato, M. Learning Longer Memory in Recurrent Neural Networks. In Int'l Conf Learning Represenations, 2015.

[8] Zaremba, W., Sutskever, I., and Vinyals, O. Recurrent Neural Network Regularization. arXiv preprint arXiv:1409.2329, 2014.

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

[10] Gers, Felix A, Schmidhuber, J¨urgen, and Cummins, Fred. Learning to forget: Continual prediction with lstm. Neural computation, 12(10):2451–2471, 2000.

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

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

[13] Zhou G. B., Wu J., Zhang C. L., and Zhou Z. H. Minimal Gated Unit for Recurrent Neural Networks. arXiv preprint arXiv:1603.09420v1 [cs.NE] 31 Mar 2016

[14] https://github.com/fchollet/keras/blob/master/examples/imdb_lstm.py.

[15] F. M. Salem, `"Reduced Parameterization in Gated Recurrent Neural Networks," Memorandum 7.11.2016, MSU, Nov 2016.

[16]F. M. Salem, "A Basic Recurrent Neural Network Model," arXiv Preprint arXiv: 1612.09022, Dec. 2016.

Recurrent Neural Network

TINGWU WANG,

MACHINE LEARNING GROUP,

UNIVERSITY OF TORONTO

FOR CSC 2541, SPORT ANALYTICS

Contents

Part One

Why do we need Recurrent Neural Network?

1. What Problems are Normal CNNs good at?
2. What is Sequence Learning?
3. Ways to Deal with Sequence Labeling.

1. What Problems are CNNs normally good at?

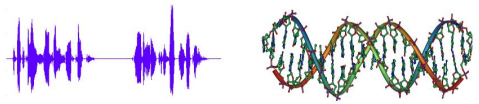
1. Image classification as a naive example
   1. Input: one image.
   2. Output: the probability distribution of classes.
   3. You need to provide one guess (output), and to do that you only need to look at one image (input).

P(Cat|image) = 0.1

P(Panda|image) = 0.9

2. What is Sequence Learning?

1. Sequence learning is the study of machine learning algorithms designed for sequential data [1].



2. Language model is one of the most interesting topics that use sequence labeling.
    1. Language Translation
        1. Understand the meaning of each word, and the relationship between words
        2. Input: one sentence in German

           input = "Ich will **stark** Steuern senken"
        3. Output: one sentence in English

           output = "I want to cut taxes **bigly**" (**big league**?)

2. What is Sequence Learning?

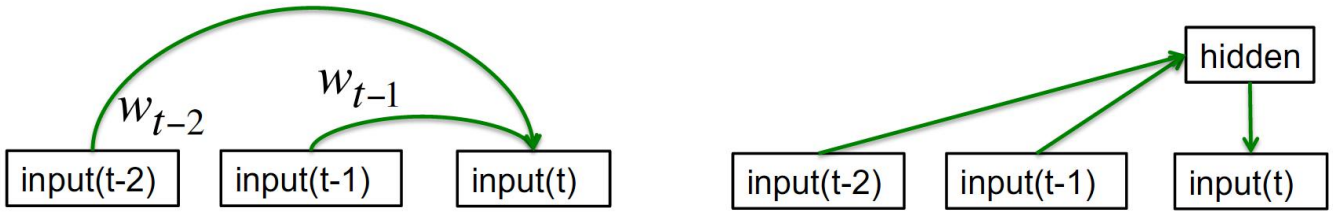
1. To make it easier to understand why we need RNN, let's think about a simple speaking case (let's violate neuroscience a little bit)
    1. We are given a hidden state (free mind?) that encodes all the information in the sentence we want to speak.
    2. We want to generate a list of words (sentence) in an one-by-one fashion.
        1. At each time step, we can only choose a single word.
        2. The hidden state is affected by the words chosen (so we could remember what we just say and complete the sentence).

2. What is Sequence Learning?

1. Plain CNNs are not born good at length-varying input and output.
    1. Difficult to define input and output
        1. Remember that
            1. Input image is a 3D tensor (width, length, color channels)
            2. Output is a distribution on fixed number of classes.
        2. Sequence could be:
            1. "I know that you know that I know that you know that I know that you know that I know that you know that I know that you know that I know that you know that I don't know"
            2. "I don't know"
    2. Input and output are strongly correlated within the sequence.
    3. Still, people figured out ways to use CNN on sequence learning (e.g. [8]).

3. Ways to Deal with Sequence Labeling

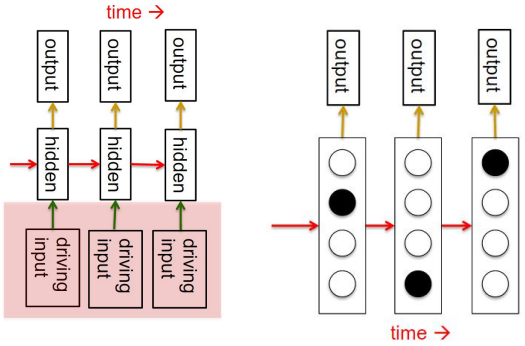
1. Autoregressive models
   1. Predict the next term in a sequence from a fixed number of previous terms using delay taps.
2. Feed-forward neural nets
   1. These generalize autoregressive models by using one or more layers of non-linear hidden units



Memoryless models: limited word-memory window; hidden state cannot be used efficiently.

3. Ways to Deal with Sequence Labeling

1. Linear Dynamical Systems
   1. These are generative models. They have a real-valued hidden state that cannot be observed directly.
2. Hidden Markov Models
   1. Have a discrete one-of-N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The outputs produced by a state are stochastic.

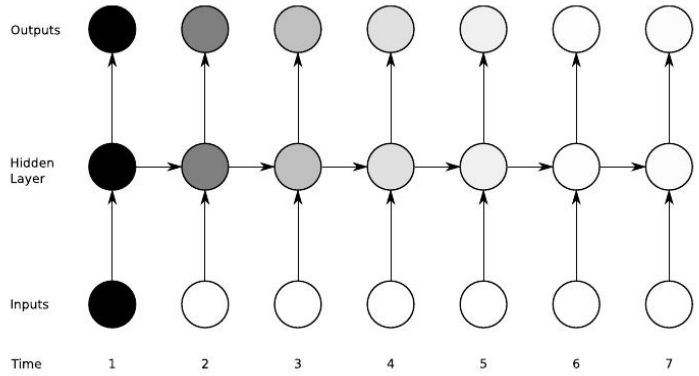


Memoryful models,

time-cost to infer the hidden state distribution.

3. Ways to Deal with Sequence Labeling

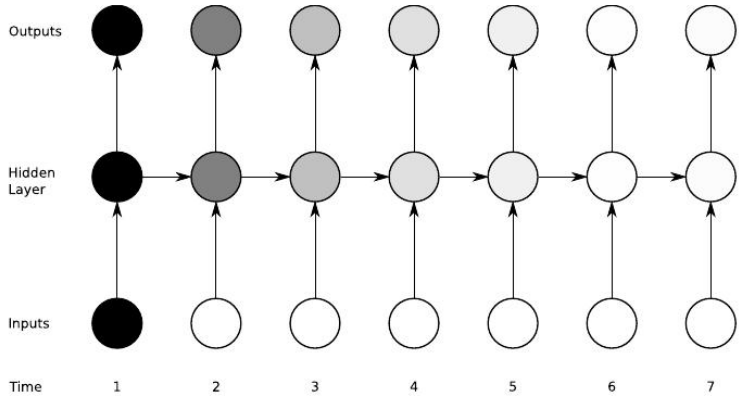
1. Finally, the RNN model!
   1. Update the hidden state in a deterministic nonlinear way.
   2. In the simple speaking case, we send the chosen word back to the network as input.



materials from [4]

3. Ways to Deal with Sequence Labeling

1. RNNs are very powerful, because they:
   1. Distributed hidden state that allows them to store a lot of information about the past efficiently.
   2. Non-linear dynamics that allows them to update their hidden state in complicated ways.
   3. No need to infer hidden state, pure deterministic.
   4. Weight sharing



Part Two

Math in a Vanilla Recurrent Neural Network

1. Vanilla Forward Pass
2. Vanilla Backward Pass
3. Vanilla Bidirectional Pass
4. Training of Vanilla RNN
5. Vanishing and exploding gradient problems

1.Vanilla Forward Pass

1. The forward pass of a vanilla RNN
   1. The same as that of an MLP with a single hidden layer
   2. Except that activations arrive at the hidden layer from both the current external input and the hidden layer activations one step back in time.

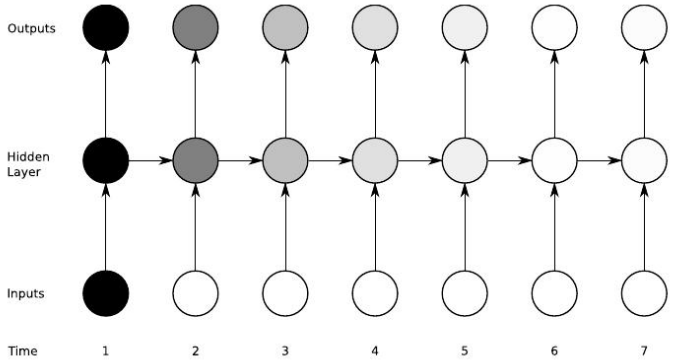
2. For the input to hidden units we have

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1}$$

$$b_h^t = \theta_h(a_h^t)$$

3. For the output unit we have

$$a_k^t = \sum_{h=1}^{H} w_{hk} b_h^t$$



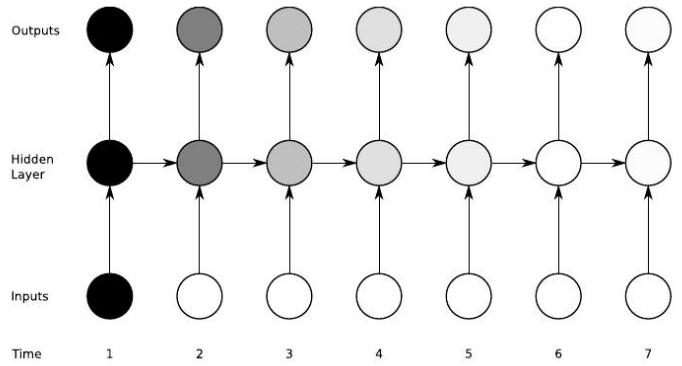
1.Vanilla Forward Pass

1. The complete sequence of hidden activations can be calculated by starting at t = 1 and recursively applying the three equations, incrementing t at each step.

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1}$$

$$b_h^t = \theta_h(a_h^t)$$

$$a_k^t = \sum_{h=1}^{H} w_{hk} b_h^t$$



2.Vanilla Backward Pass

1. Given the partial derivatives of the objective function with respect to the network outputs, we now need the derivatives with respect to the weights.
2. We focus on BPTT since it is both conceptually simpler and more efficient in computation time (though not in memory). Like standard back-propagation, BPTT consists of a repeated application of the chain rule.

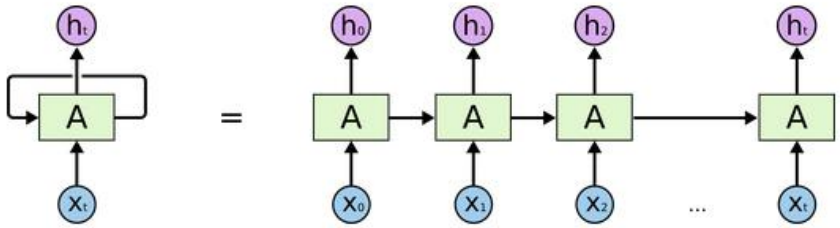
$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1}$$

$$b_h^t = \theta_h(a_h^t)$$

$$a_k^t = \sum_{h=1}^{H} w_{hk} b_h^t$$

2.Vanilla Backward Pass

1. Back-propagation through time
   1. Don't be fooled by the fancy name. It's just the standard back-propagation.



An unrolled recurrent neural network.

$$\delta_h^t = \theta'(a_h^t)\left(\sum_{k=1}^{K}\delta_k^t w_{hk} + \sum_{h'=1}^{H}\delta_{h'}^{t+1} w_{hh'}\right),$$

$$\delta_j^t \stackrel{\text{def}}{=} \frac{\partial O}{\partial a_j^t}$$

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1}$$

$$b_h^t = \theta_h(a_h^t)$$

$$a_k^t = \sum_{h=1}^{H} w_{hk} b_h^t$$

materials from [6]

2.Vanilla Backward Pass

1. Back-propagation through time
   1. The complete sequence of delta terms can be calculated by starting at t = T and recursively applying the below functions, decrementing t at each step.
   2. Note that $\delta_j^{T+1} = 0 \, \forall j$, since no error is received from beyond the end of the sequence.

$$\delta_h^t = \theta'(a_h^t)\left(\sum_{k=1}^{K} \delta_k^t w_{hk} + \sum_{h'=1}^{H} \delta_{h'}^{t+1} w_{hh'}\right), \qquad a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1}$$

$$b_h^t = \theta_h(a_h^t)$$

$$\delta_j^t \stackrel{\text{def}}{=} \frac{\partial O}{\partial a_j^t} \qquad a_k^t = \sum_{h=1}^{H} w_{hk} b_h^t$$

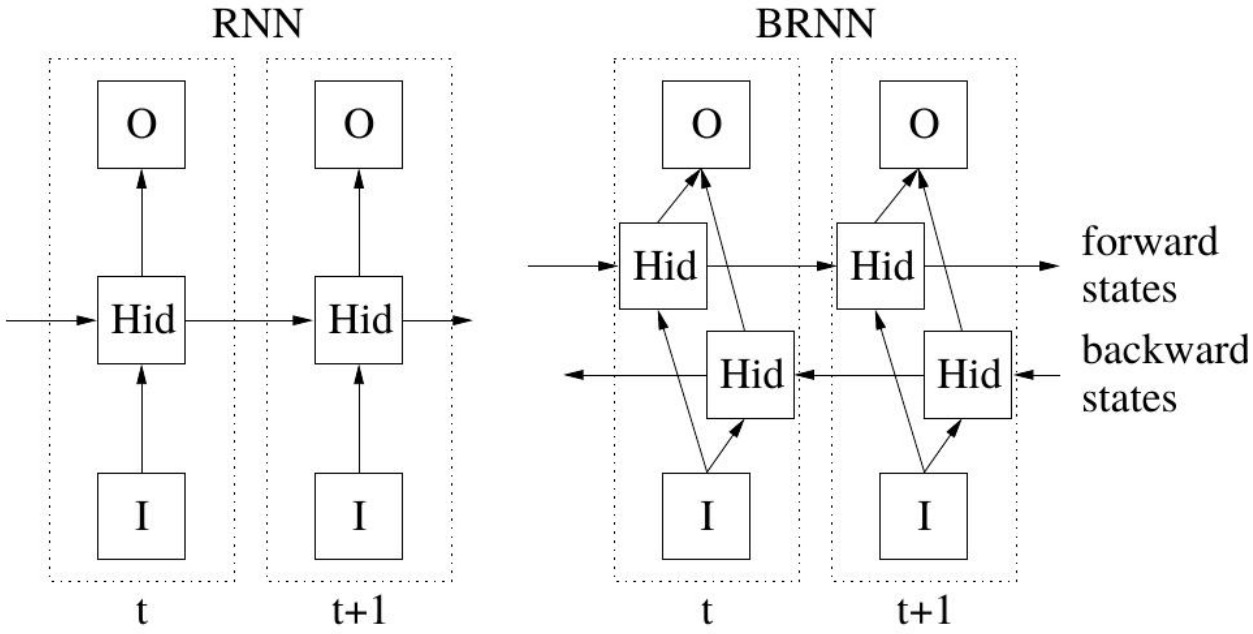
   3. Finally, bearing in mind that the weights to and from each unit in the hidden layer are the same at every time-step, we sum over the whole sequence to get the derivatives with respect to each of the network weights

$$\frac{\partial O}{\partial w_{ij}} = \sum_{t=1}^{T} \frac{\partial O}{\partial a_j^t} \frac{\partial a_j^t}{\partial w_{ij}} = \sum_{t=1}^{T} \delta_j^t b_i^t$$

materials from [4]

3.Vanilla Bidirectional Pass

1. For many sequence labeling tasks, we would like to have access to future.



3.Vanilla Bidirectional Pass

1. Algorithm looks like this

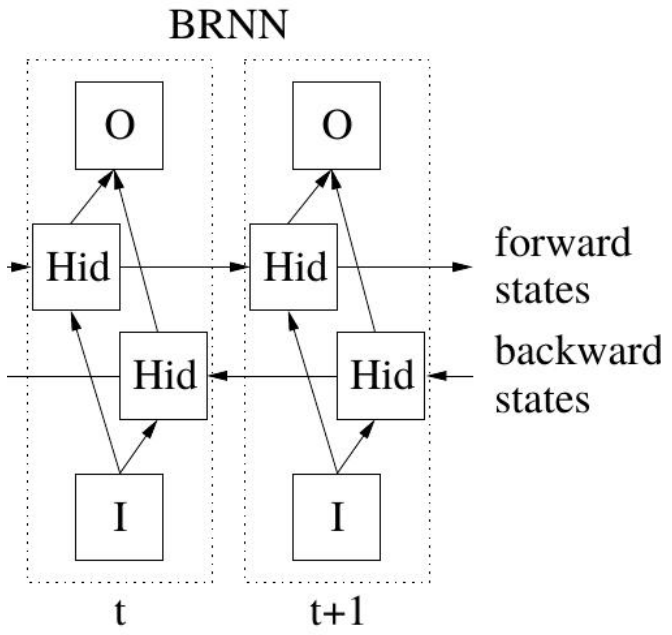
for $t = 1$ to T **do**
Do forward pass for the forward hidden layer, storing activations at each timestep
for $t = T$ to 1 **do**
Do forward pass for the backward hidden layer, storing activations at each timestep
for $t = 1$ to T **do**
Do forward pass for the output layer, using the stored activations from both hidden layers

Algorithm 3.1: BRNN Forward Pass

Similarly, the backward pass proceeds as for a standard RNN trained with BPTT, except that all the output layer δ terms are calculated first, then fed back to the two hidden layers in opposite directions:

for $t = T$ to 1 **do**
Do BPTT backward pass for the output layer only, storing δ terms at each timestep
for $t = T$ to 1 **do**
Do BPTT backward pass for the forward hidden layer, using the stored δ terms from the output layer
for $t = 1$ to T **do**
Do BPTT backward pass for the backward hidden layer, using the stored δ terms from the output layer

Algorithm 3.2: BRNN Backward Pass



4.Training of Vanilla RNN

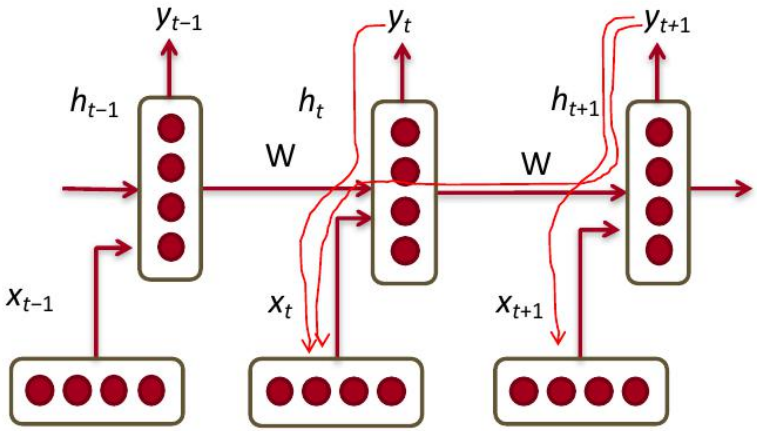
1. So far we have discussed how RNN can be differentiated with respect to suitable objective functions, and thereby they could be trained with any gradient-descent based algorithm
   1. just treat them as a normal CNN

$$\Delta \mathbf{w}(n) = -\alpha \frac{\partial O}{\partial \mathbf{w}(n)}$$

2. One of the great things about RNN: lots of engineering choices
   1. Preprocessing and postprocessing

5.Vanishing and exploding gradient problems

1. Multiply the same matrix at each time step during back-prop



5.Vanishing and exploding gradient problems

1. Toy example how gradient vanishes
   1. Similar but simpler RNN formulation:

$$h_t = W f(h_{t-1}) + W^{(hx)} x_{[t]}$$

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W^T \text{diag}[f'(h_{j-1})]$$

$$\left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq \|W^T\| \|\text{diag}[f'(h_{j-1})]\| \leq \beta_W \beta_h$$

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq (\beta_W \beta_h)^{t-k}$$

2. Solutions?
   1. For vanishing gradients: Initialization + ReLus
   2. Trick for exploding gradient: clipping trick

Part Three

From Vanilla to LSTM

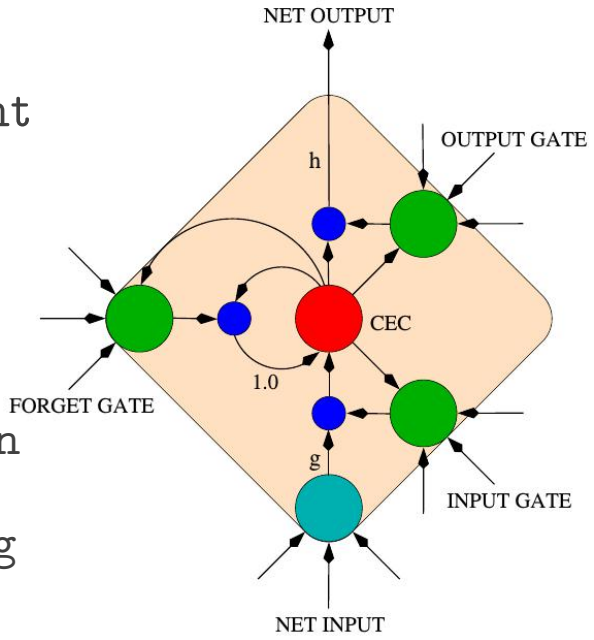
1. Definition
2. Forward Pass
3. Backward Pass

1. Definition

1. As discussed earlier, for standard RNN architectures, the range of context that can be accessed is limited.
    1. The problem is that the influence of a given input on the hidden layer, and therefore on the network output, either decays or blows up exponentially as it cycles around the network's recurrent connections.
2. The most effective solution so far is the Long Short Term Memory (LSTM) architecture (Hochreiter and Schmidhuber, 1997).
3. The LSTM architecture consists of a set of recurrently connected subnets, known as memory blocks. These blocks can be thought of as a differentiable version of the memory chips in a digital computer. Each block contains one or more self-connected memory cells and three multiplicative units that provide continuous analogues of write, read and reset operations for the cells
    1. The input, output and forget gates.

1. Definition

1. The multiplicative gates allow LSTM memory cells to store and access information over long periods of time, thereby avoiding the vanishing gradient problem
    1. For example, as long as the input gate remains closed (i.e. has an activation close to 0), the activation of the cell will not be overwritten by the new inputs arriving in the network, and can therefore be made available to the net much later in the sequence, by opening the output gate.



1. Definition

1. Comparison

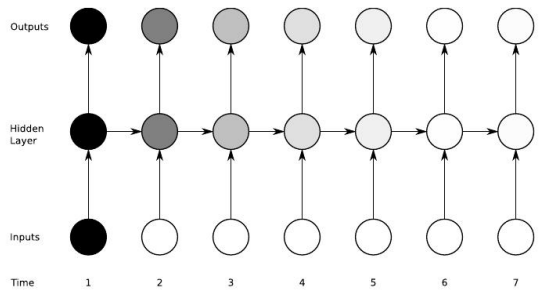


Figure 4.1: **Vanishing gradient problem for RNNs.** The shading of the nodes indicates the sensitivity over time of the network nodes to the input at time one (the darker the shade, the greater the sensitivity). The sensitivity decays exponentially over time as new inputs overwrite the activation of hidden unit and the network 'forgets' the first input.

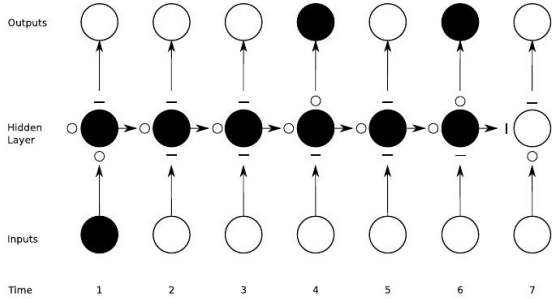


Figure 4.3: **Preservation of gradient information by LSTM.** As in Figure 4.1 the shading of the nodes indicates their sensitivity to the input unit at time one. The state of the input, forget, and output gate states are displayed below, to the left and above the hidden layer node, which corresponds to a single memory cell. For simplicity, the gates are either entirely open ('O') or closed ('—'). The memory cell 'remembers' the first input as long as the forget gate is open and the input gate is closed, and the sensitivity of the output layer can be switched on and off by the output gate without affecting the cell.

2. Forward Pass

1. Basically very similar to the vanilla RNN forward pass
   1. But it's a lot more complicated
   2. Can you do the backward pass by yourself?
      1. **Quick quiz**, get a white sheet of paper. Write your student number and name.
      2. We are going to give you 10 minutes
      3. No discussion
      4. 10% of your final grades

Input Gates

$$a_\iota^t = \sum_{i=1}^{I} w_{i\iota} x_i^t + \sum_{h=1}^{H} w_{h\iota} b_h^{t-1} + \sum_{c=1}^{C} w_{c\iota} s_c^{t-1}$$
$$b_\iota^t = f(a_\iota^t)$$

Forget Gates

$$a_\phi^t = \sum_{i=1}^{I} w_{i\phi} x_i^t + \sum_{h=1}^{H} w_{h\phi} b_h^{t-1} + \sum_{c=1}^{C} w_{c\phi} s_c^{t-1}$$
$$b_\phi^t = f(a_\phi^t)$$

Cells

$$a_c^t = \sum_{i=1}^{I} w_{ic} x_i^t + \sum_{h=1}^{H} w_{hc} b_h^{t-1}$$
$$s_c^t = b_\phi^t s_c^{t-1} + b_\iota^t g(a_c^t)$$

Output Gates

$$a_\omega^t = \sum_{i=1}^{I} w_{i\omega} x_i^t + \sum_{h=1}^{H} w_{h\omega} b_h^{t-1} + \sum_{c=1}^{C} w_{c\omega} s_c^t$$
$$b_\omega^t = f(a_\omega^t)$$

Cell Outputs

$$b_c^t = b_\omega^t h(s_c^t)$$

3. Backward Pass

1. Just kidding! The math to get the backward pass should be very similar to the one used in vanilla RNN backward pass
    1. But it's a lot more complicated, too...
    2. We are not going to derive that in the class

$$\epsilon_c^t \stackrel{\text{def}}{=} \frac{\partial O}{\partial b_c^t} \qquad \epsilon_s^t \stackrel{\text{def}}{=} \frac{\partial O}{\partial s_c^t}$$

Cell Outputs

$$\epsilon_c^t = \sum_{k=1}^{K} w_{ck}\delta_k^t + \sum_{h=1}^{H} w_{ch}\delta_h^{t+1}$$

Output Gates

$$\delta_\omega^t = f'(a_\omega^t)\sum_{c=1}^{C} h(s_c^t)\epsilon_c^t$$

States

$$\epsilon_s^t = b_\omega^t h'(s_c^t)\epsilon_c^t + b_\phi^{t+1}\epsilon_s^{t+1} + w_{c\iota}\delta_\iota^{t+1} + w_{c\phi}\delta_\phi^{t+1} + w_{c\omega}\delta_\omega^t$$

Cells

$$\delta_c^t = b_\iota^t g'(a_c^t)\epsilon_s^t$$

Forget Gates

$$\delta_\phi^t = f'(a_\phi^t)\sum_{c=1}^{C} s_c^{t-1}\epsilon_s^t$$

Input Gates

$$\delta_\iota^t = f'(a_\iota^t)\sum_{c=1}^{C} g(a_c^t)\epsilon_s^t$$

Part Four

Miscellaneous

1. More than Language Model
2. GRU

1. More than Language Model

1. Like I said, RNN could do a lot more than modeling language
    1. Drawing pictures:

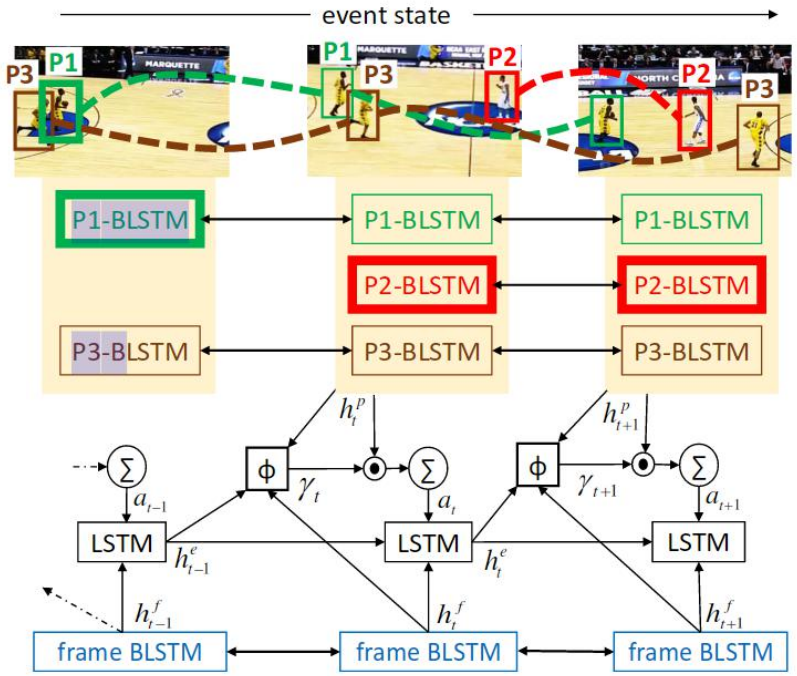
       [9] DRAW: A Recurrent Neural Network For Image Generation
    2. Computer-composed music

       [10] Song From PI: A Musically Plausible Network for Pop Music Generation
    3. Semantic segmentation

       [11] Conditional random fields as recurrent neural networks

1. More than Language Model

1. RNN in sports
   1. Sport is a sequence of event (sequence of images voices)
   2. Detecting events and key actors in multi-person videos [12]
      1. "In particular, we track people in videos and use a recurrent neural network (RNN) to represent the track features. We learn time-varying attention weights to combine these features at each time-instar The attended features are then processed using another RNN for event detection/classification"



1. More than Language Model

1. RNN in sports
    1. Applying Deep Learning to Basketball Trajectories
        1. This paper applies recurrent neural networks in the form of sequence modeling to predict whether a three-point shot is successful [13]

    2. Action Classification in Soccer Videos with Long Short-Term Memory Recurrent Neural Networks [14]

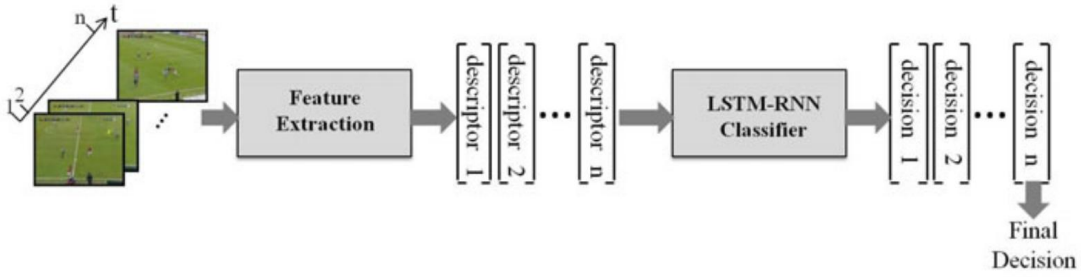
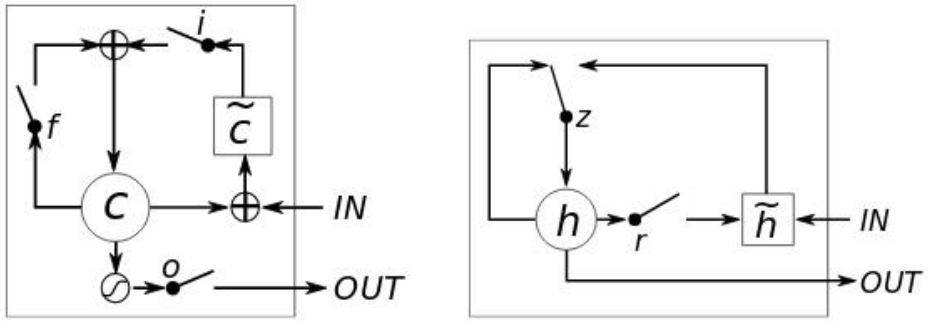


Fig. 1. Proposed classification scheme

2. GRU

1. A new type of RNN cell (Gated Feedback Recurrent Neural Networks)
    1. Very similar to LSTM
    2. It merges the cell state and hidden state.
    3. It combines the forget and input gates into a single "update gate".
    4. Computationally more efficient.
        1. less parameters, less complex structure.
2. Gaining popularity nowadays [15,16]



(a) Long Short-Term Memory (b) Gated Recurrent Unit

Part Five

Implementing RNN in Tensorflow

1. Implementing RNN in Tensorflow

1. The best way should be reading the docs on Tensorflow website [17].
2. Let's assume you already manage how to use CNN in Tensorflow (toy sequence decoder model)

```
lstm = rnn_cell.BasicLSTMCell(lstm_size)
# Initial state of the LSTM memory.
state = tf.zeros([batch_size, lstm.state_size])
probabilities = []
loss = 0.0
for current_batch_of_words in words_in_dataset:
    # The value of state is updated after processing each batch of words.
    output, state = lstm(current_batch_of_words, state)

    # The LSTM output can be used to make next word predictions
    logits = tf.matmul(output, softmax_w) + softmax_b
    probabilities.append(tf.nn.softmax(logits))
    loss += loss_function(probabilities, target_words)
```

1. Implementing RNN in Tensorflow

1. Standard feed dictionary just like other CNN models in Tensorflow

```
# A numpy array holding the state of LSTM after each batch of words.
numpy_state = initial_state.eval()
total_loss = 0.0
for current_batch_of_words in words_in_dataset:
    numpy_state, current_loss = session.run([final_state, loss],
        # Initialize the LSTM state from the previous iteration.
        feed_dict={initial_state: numpy_state, words: current_batch_of_words})
    total_loss += current_loss
```

2. Word-embedding (make the words in the sentence understandable by the program)

```
# embedding_matrix is a tensor of shape [vocabulary_size, embedding size]
word_embeddings = tf.nn.embedding_lookup(embedding_matrix, word_ids)
```

1. Implementing RNN in Tensorflow

1. Simple example using Tensorflow.
    1. The task: let the robot learn the atom behavior it should do, by following human instructions
    2. The result we could get by using RNN.

2. Task:
    1. Input: "Sit down on the couch and watch T.V. When you are done watching television turn it off. Put the pen on the table. Toast some bread in the toaster and get a knife to put butter on the bread while you sit down at the table."

1. Implementing RNN in Tensorflow

1. Simple demo result.

http://www.cs.toronto.edu/~tingwuwang/outputscript_synthetic_data_clean_is_rnn_encoder_True_decoder_dim_150_model.ckpt.html

GT

1. Walk , TELEVISION , None
2. SwitchOn , TELEVISION , None
3. Walk , SOFA , None
4. Sit , SOFA , None
5. Watch , TELEVISION , None
6. StandUp , None , None
7. Walk , TELEVISION , None
8. SwitchOff , TELEVISION , None
9. Walk , PEN , None
10. Grab , PEN , None
11. Walk , TABLE , None
12. PutBack , PEN , TABLE
13. Walk , KNIFE , None
14. Grab , KNIFE , None
15. Walk , TOASTER , None
16. SwitchOn , TOASTER , None
17. SwitchOff , TOASTER , None
18. Walk , TABLE , None
19. PutBack , KNIFE , TABLE

PRED

1. Walk , TELEVISION , None
2. SwitchOn , TELEVISION , None
3. Walk , SOFA , None
4. Sit , SOFA , None
5. Watch , TELEVISION , None
6. StandUp , None , None
7. Walk , TELEVISION , None
8. SwitchOff , TELEVISION , None
9. Walk , PEN , None
10. Grab , PEN , None
11. Walk , TABLE , None
12. PutBack , PEN , TABLE
13. Walk , TOASTER , None
14. SwitchOn , TOASTER , None
15. SwitchOff , TOASTER , None
16. Walk , KNIFE , None
17. Grab , KNIFE , None
18. Walk , TABLE , None
19. PutBack , KNIFE , TABLE

References

Most of the materials in the slides come from the following tutorials / lecture slides:

[1] Machine Learning I Week 14: Sequence Learning Introduction, Alex Graves, Technische Universitaet Muenchen.

[2] CSC2535 2013: Advanced Machine Learning, Lecture 10: Recurrent neural networks, Geoffrey Hinton, University of Toronto.

[3] CS224d Deep NLP, Lecture 8: Recurrent Neural Networks, Richard Socher, Stanford University.

[4] Supervised Sequence Labelling with Recurrent Neural Networks, Alex Graves, Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigten Dissertation.

[5] The Unreasonable Effectiveness of Recurrent Neural Networks, Andrej Karpathy, blog About Hacker's guide to Neural Networks.

[6] Understanding LSTM Networks, Christopher Olah, github blog.

Other references

[7] Kiros, Ryan, et al. "Skip-thought vectors." Advances in neural information processing systems. 2015.

[8] Dauphin, Yann N., et al. "Language Modeling with Gated Convolutional Networks." arXiv preprint arXiv:1612.08083 (2016).

[9] Gregor, Karol, et al. "DRAW: A recurrent neural network for image generation." arXiv preprint arXiv:1502.04623 (2015).

References

[10] Chu, Hang, Raquel Urtasun, and Sanja Fidler. "Song From PI: A Musically Plausible Network for Pop Music Generation." arXiv preprint arXiv:1611.03477 (2016).

[11] Zheng, Shuai, et al. "Conditional random fields as recurrent neural networks." Proceedings of the IEEE International Conference on Computer Vision. 2015.

[12] Ramanathan, Vignesh, et al. "Detecting events and key actors in multi-person videos." arXiv preprint arXiv:1511.02917 (2015).

[13] Shah, Rajiv, and Rob Romijnders. "Applying Deep Learning to Basketball Trajectories." arXiv preprint arXiv:1608.03793 (2016).

[14] Baccouche, Moez, et al. "Action classification in soccer videos with long short-term memory recurrent neural networks." International Conference on Artificial Neural Networks. Springer Berlin Heidelberg, 2010.

[15] Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." arXiv preprint arXiv:1412.3555 (2014).

[16] Chung, Junyoung, et al. "Gated feedback recurrent neural networks." CoRR, abs/1502.02367 (2015).

[17] Tutorials on Tensorflow. https://www.tensorflow.org/tutorials/

Q&A

Thank you for listening ;P