



Departamento de Matemáticas, Facultad de Ciencias
Universidad Autónoma de Madrid

# Traditional and Long Short-Term Memory Recurrent Neural Networks: A theoretical and applied comparison

Degree in Mathematics

*Author:* Javier Delgado del Cerro
*Tutor:* Davide Barbieri

Course 2020-2021

## Resumen

La *Inteligencia Artificial*, y en concreto las *Redes Neuronales Artificiales*, conforman un nicho de la tecnología que se ha expandido tremendamente en los últimos años, llegando a todas nuestras vidas mediante las predicciones meteorológicas, la publicidad online o múltiples aplicaciones en nuestros smartphones.

Dentro de este trabajo vamos a analizar un tipo específico de red neuronal conocido como Redes Neuronales Recurrentes, encargadas de desarrollar tareas relacionadas con series temporales, como puede ser el reconocimiento de voz. Mostraremos su desarrollo a partir de sistemas dinámicos, las ecuaciones que las caracterizan, y analizaremos el principal problema que impide su aplicación práctica. Una vez mostrado este problema, explicaremos una posible solución, que pasa por el desarrollo de las redes neuronales recurrentes *Long Short-Term Memory*, y analizaremos porqué éste nuevo tipo de redes no sufren los problemas de las tradicionales. A lo largo del trabajo señalaremos también algunos errores más frecuentes dentro de la literatura científica al tratar con este tipo de redes. Finalmente, haremos una comparativa aplicada entre las redes neuronales recurrentes tradicionales y las LSTM mediante el framework *Tensorflow*, demostrando como los problemas teóricos vistos se trasladan a la práctica, en una tarea como es la generación de texto.



## Abstract

*Artificial Intelligence* and specifically *Artificial Neural Networks*, are a technology niche that has expanded tremendously in recent years, reaching all our lives through weather forecasts, online advertising or multiple applications on our smartphones.

Within this work we are going to analyze a specific type of neural network known as *Recurrent Neural Networks*, usually used on tasks related to time series, such as speech recognition. We will show their development from dynamical systems, the equations that characterize them, and we will analyze the main problem that prevents their practical application. Once this problem has been shown, we will explain a possible solution, which involves the development of *Long Short-Term Memory* recurrent neural networks, and we will analyze why this new type of network does not suffer the problems of the traditional ones. Throughout the paper we will also point out some of the most frequent errors in the scientific literature when dealing with this type of networks. Finally, we will make an applied comparison between traditional RNN and LSTMs using the *Tensorflow* framework, demonstrating how theoretical problems are transferred to practice, in a task such as text generation.

# Contents

# CHAPTER 1
# Introduction

## 1.1. What is a neural network?

Nowadays the term *Artificial Intelligence* is completely rooted in our society, and it has more importance every day due to its use on mobile phones, weather forecasting or advertising recommendations. However, although the idea of trying to describe how the mind works can be traced back to Aristoteles (384BC - 322BC), and the idea of achieving some kind of artificial reasoning was raised by Ramon Llull in his book *Ars magna* in 1315, it wasn't until 1956 at the so called *Dartmouth workshop* (a research project that took place at the Dartmouth College for about eight months in which multiple experts aimed to clarify and develop new ideas about thinking machines) where the term Artificial intelligence was created.

One of the main fields that make up the Artificial Intelligence are the Artificial Neural Networks (from now on, simply referred as Neural Networks) which emerge as a way to replicate the natural working of neurons in the human brain. To understand this we need to mention two specially important investigations about the human brain [10]:

- In 1943, the neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper on how neurons might work, and in order to describe their hypothesis, they modeled a simple neural network using electrical circuits.

- In 1949, Donald Hebb wrote *The Organization of Behavior*, a work which pointed out the fact that neural pathways are strengthened each time they are used, a concept fundamentally essential to the ways in which humans learn. If two nerves fire at the same time, he argued, the connection between them is enhanced. This began the journey of quantifying the complex processes of the brain.

After the emergence of these ideas, researches began trying to translate these networks onto computational systems. After some failed attempts, the first trainable neural network, the **perceptron**, was demonstrated by the Cornell University psychologist Frank Rosenblatt in 1957. The perceptron's design was much like that of the modern neural net, except that it had only one layer with adjustable weights and thresholds, sandwiched between input and output layers [7]. That perceptron represented in Figure 1.1 is currently

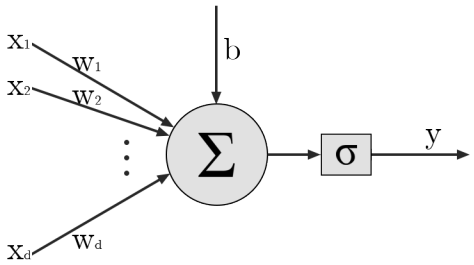



Figure 1.1: The basic perceptron.

described as a single cell with the following equation:

$$y(t) = \sigma(\sum_{k=1}^{d} w_k(t)x_k(t) + b)$$

where:

- $x(t) \in \mathbb{R}^d$ is the cell input data at time t.
- $y(t) \in \{-1, 1\}$ is the cell output data at time t.
- $w(t) \in \mathbb{R}^d$ is the cell vector of synaptic weights at time t.
- $b \in \mathbb{R}$ is the cell bias, and can be considered as an input $x_0$ whose weight is always $w_0 = 1$.
- $\sigma$ is a hard delimiter function which outputs +1 if its input is positive and -1 if it's negative.

Defining the bias as $x_0 = b$, $w_0 = 1$, the equation can be written as $y(t) = \sigma\big(w^T(t)\, x(t)\big)$. The well-known Hebb's rule provides an intuitive mechanism to update the weights in order to learn a functional relation between inputs and outputs [8]. It reads:

$$w(t+1) = w(t) + \eta[\xi(t) - y(t)]x(t)$$

where $\eta$ is known as the learning rate and $\xi(t)$ is the label or expected output at time t, hence relying on the so-called supervised learning paradigm.

Therefore, the goal of this perceptron was to classify the set of external inputs into one of two classes by adapting its weights iteration by iteration during its training. The main limitation of this simple perceptron is that it can only separate the classes linearly, making it impossible to work as an xor gate, for example. Moreover, Hebb's rule in its basic form is unstable and typically causes divergent behaviours

In order to obtain non-linear classifiers, the idea of using more than one layer came along, but due to the computational power required by the training algorithms that solution was discarded, and it wasn't until the 1980s that new, more efficient algorithms, were developed and allowed for networks with two or three layers, limited again by the machines of that time. The main scheme, or architecture, is known as the **multilayer perceptron** as it's basically a set of sequential layers made up of a number of perceptrons, see Fig. 1.3.

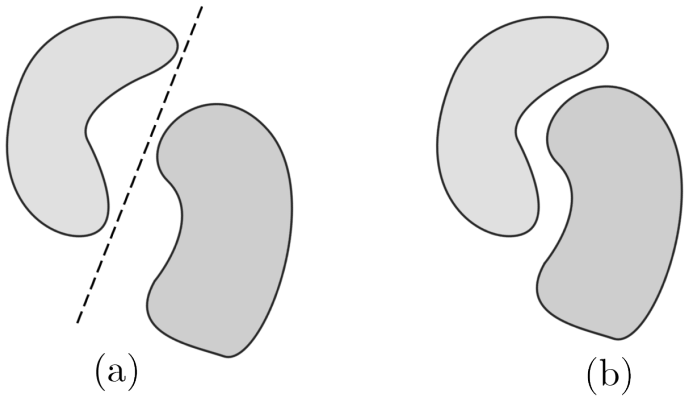



Figure 1.2: The figure (a) shows a pair of linearly separable classes that could be distinguished by a perceptron while figure (b) represents a pair of non linearly separable classes.

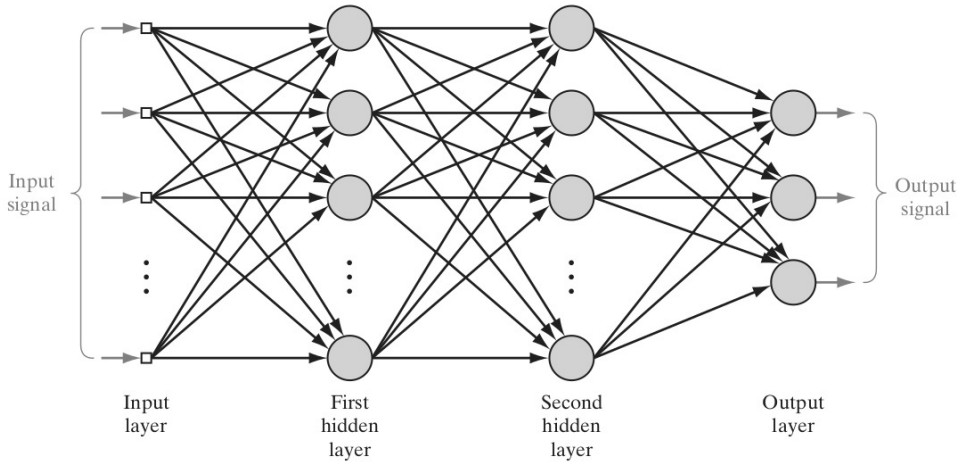



Figure 1.3: Architectural graph of a multilayer perceptron with two hidden layers where every grey ball represents a single perceptron. Extracted from S.Haykin. 2009.

In this case, the training is more complex and it's divided into two different phases whose repetition over a set of inputs, called a training set, allows the network to learn the weights that best suit a given task:

- The **forward propagation**, where the weights of the network are fixed and the input is propagated layer by layer until it reaches the output. It uses the perceptron as a function, storing the output related to each input.

- The **back propagation**, where an error signal is calculated by comparing the generated output to the expected one. This error is propagated from the output layer to the input layer, using differential calculus, and the weights on each layer are updated. Basically, this is a gradient iteration aimed to minimize the difference between the output of the network and the ground truth given by the training labels.

Finally, modern GPUs, which are capable of doing a huge number of mathematical operations in a tiny fraction of time, enabled the use of 10, 15, 50 or even more layers on a single network, making the technology much more accessible.

## 1.2. The origin of RNN

The idea behind the multilayer perceptron opened a new world of possibilities, but it also came with its own limitations. One of those limitations is that the number of inputs is fixed and constant, and the neural network considers the inputs as completely independent events, meaning that, while processing an input, it doesn't take into account the previous ones, which makes it impossible to understand temporal series. For example, imagine we want a network to predict the next number given a series: if we were using a traditional neural network, it would need the same numbers of entries every time to predict the next one. To fix this, the simplest approach is to use the output $y(t)$ of the network at time $t$ as an input at the time $t+1$,as represented on Figure 1.4.

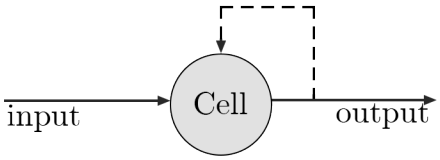



Figure 1.4: Traditional RNN scheme.

This is how the traditional recurrent neural network works, and it allows the network to make decisions considering the previous ones. It also allows the processing of sequences of variable lengths, as we can input one word at a time, and the sentence could (theoretically) have an arbitrary number of words.
For example, for a vector-valued input $x(t)$ and a vector-valued output $y(t)$, the Figure 1.4 could be represented by equation (1.1), which, as we will see later, is the equation that describes the traditional RNN.

$$y(t+1) = \sigma(Ry(t) + Wx(t)). \tag{1.1}$$

Some applications of RNN are machine translation, where the original text is fed to a RNN and it produces the translated text as output, or sentiment analysis, where the network is trained to predict whether a comment or a review is positive or negative, speech recognition, chatbots, etc. They can also be used in two dimensions, for example, by processing a series of images and determining the motion inside those images.

## 1.3. Plan of the work

The rest of this document is organized as follows. In Chapter 2 we study the traditional RNN: we introduce them as a dynamical systems, derive the learning algorithm and discuss its weaknesses. In Chapter 3 we explain the general method that should be used to derive the learning equations of any RNN, construct a solid proof, and apply it to traditional

RNN as an example, showing that result matches with the previously derived equations. In Chapter 4 we introduce the Long Short-Term Memory RNN, derive its equations, present some problems on the current literature on the topic, and provide a quantitative proof of why it fixes the weaknesses of traditional RNN. Finally, in Chapter 5 we use the Tensorflow framework to make an applied comparison between traditional RNN and LSTM, giving an example on how the theoretical weaknesses of traditional RNN are translated to the practice.

Although we have used numerous references throughout the development of this work, it is worth mentioning the statistical tools provided at [3], along with the contributions of [8] and [4], which allowed us to enter the field of Neural Networks and Recurrent Neural Networks. Later in the work, [5], [12] and [6] have been very helpful in the development of the different equations shown throughout the text. Of course, special mention must be made to [11], whose reasoning has been extremely useful, and whose author, A. Sherstinsky, was kind enough to lend us a hand at one point in the work.

# CHAPTER 2
# Traditional RNN and its problems

This chapter focuses on the discussion of the simplest model of Recurrent Neural Network and follows the ideas developed in [11] with some help from [3].

## 2.1. Introduction to RNN from dynamical systems

In this section we are going to give a brief explanation on how the traditional Recurrent Neural Network Equations can be derived from differential equations.

Let $c(t)$ be a $k-$dimensional vector whose evolution can be described by the following ordinary differential equation:

$$\frac{dc(t)}{dt} = f(t) \tag{2.1}$$

with $f(t)$ being a $k-$dimensional function of time $t \in \mathbb{R}^+$. Now, we can use as $f$ what is usually known in the Brain Dynamics research literature as the *additive model*, in which $f(t) = \widetilde{\alpha}(t) + \widetilde{\beta}(t) + \widetilde{\gamma}(t)$. This is called the additive model because it adds the possibly non-linear terms that determine the rate of change, and which can incorporate delays, saturation, etc. Now, lets consider:

$$\begin{aligned}
\widetilde{\alpha}(t) &= \sum_{k=0}^{K_c-1} \alpha\Big(c\big(t-\tau_c(k)\big)\Big) \\
\widetilde{\beta}(t) &= \sum_{k=0}^{K_y-1} \beta\Big(y\big(t-\tau_y(k)\big)\Big) \\
y\big(t-\tau_y(k)\big) &= \sigma\Big(c\big(t-\tau_y(k)\big)\Big) \\
\widetilde{\gamma}(t) &= \sum_{k=0}^{K_x-1} \gamma\Big(x\big(t-\tau_x(k)\big)\Big)
\end{aligned}$$

where $\alpha, \beta, \gamma$ are model functions, and

- $x(t) \in \mathbb{R}^d$ is the input data at time t.
- $y(t) \in \mathbb{R}^k$ is the output data at time t, and it's just a warped version of the state signal $c(t)$.

- $\sigma$ is a sigmoid function. Note that, in general, by a sigmoid function one considers a S-shaped function from $\mathbb{R}$ to $\mathbb{R}$ that is applied component-wise to vectors, hence producing a vector of the same dimension. Typically, a sigmoid can be seen as a smooth approximation of a step function. An example usually used on neural networks is given by $\sigma(x) = tanh(x)$.

By introducing these equations into the previously defined $f(t)$, we get to an ordinary delay differential equation with discrete delays in which the first additive component includes $K_c$ possibly time-shifted functions of the state signal itself (which have a strong effect on the system's stability), the second component is a combination of $K_y$ possibly time-shifted functions of the readout signal (which capture most of the long-term interactions), and the last term is a combination of $K_x$ possibly time-shifted functions of the external input. These time delays constitute the "memory" of the system, and the idea behind the use of a sigmoid function is to keep the state bounded.
The evolution (2.1) in this model is then described by:

$$\frac{dc(t)}{dt} = \sum_{k=0}^{K_c-1} \alpha\Big(c\big(t-\tau_c(k)\big)\Big) + \sum_{k=0}^{K_y-1} \beta\Big(y\big(t-\tau_y(k)\big)\Big) + \sum_{k=0}^{K_x-1} \gamma\Big(x\big(t-\tau_x(k)\big)\Big). \tag{2.2}$$

Once the general system is defined, we can apply some simplifications, starting with setting $K_c = K_y = K_x = 1$ and supposing that $\alpha, \beta, \gamma$ are linear functions represented by matrices $A, B, C$ respectively. Finally, let $\tau_c(0) = \tau_x(0) = 0,\ \tau_y(0) = \tau_0$ so that only the readout signal is time-shifted. With this simplifications, equation (2.2) is turned into:

$$\frac{dc(t)}{dt} = Ac(t) + By(t-\tau_0) + Cx(t) \tag{2.3}$$

and the backward Euler discretization rule can be applied using $\Delta T$ as the duration of a time step:

$$t = \Delta T$$
$$\frac{dc(t)}{dt} \approx \frac{c(n\Delta T + \Delta T) - c(n\Delta T)}{\Delta T}$$

$$\begin{aligned} &Ac(t+\Delta T) + By(t+\Delta T-\tau_0) + Cx(t+\Delta T) \\ &\quad = Ac(n\Delta T + \Delta T) + By(n\Delta T + \Delta T - \tau_0) + Cx(n\Delta T + \Delta T) \end{aligned}$$

$$\frac{c(n\Delta T + \Delta T) - c(n\Delta T)}{\Delta T} \approx Ac(n\Delta T + \Delta T) + By(n\Delta T + \Delta T - \tau_0) + Cx(n\Delta T + \Delta T).$$

By setting the remaining delay as one time step $\tau_0 = \Delta T$, replacing the approximation sign with an equal sign for convenience, and solving the equation, we get to:

$$c((n+1)\Delta T) - c(n\Delta T) = \Delta T\Big(Ac((n+1)\Delta T) + By(n\Delta T) + Cx((n+1)\Delta T)\Big).$$

Now, the discretization is complete, and the measurements are all multiples of the time step $\Delta T$, so it can be dropped from the arguments leaving the time axis dimensionless:

$$c(n+1) = c(n) + \Delta T\big(Ac(n+1) + By(n) + Cx(n+1)\big)$$

or, equivalently

$$\big(I - (\Delta T)A\big)c(n+1) = c(n) + \big((\Delta T)B\big)y(n) + \big((\Delta T)C\big)x(n+1). \tag{2.4}$$

Let

$$M = \big(I - (\Delta T)A\big)^{-1}$$

and solve on equation (2.4)

$$c(n) = Mc(n-1) + \big((\Delta T)MB\big)y(n-1) + \big((\Delta T)MC\big)x(n).$$

If we now define

$$R = \big((\Delta T)MB\big)$$
$$W = \big((\Delta T)MC\big),$$

the evolution (2.3) is transformed into a recurrent network-type equation:

$$c(n) = Mc(n-1) + Ry(n-1) + Wx(n)$$
$$y(n) = \sigma(c(n)).$$

By network equation here we mean an interpretation of the above iteration where each row of the matrices describes the action of a neuron, and the components of $y$ describe the outputs of the system. In order to simplify the network even further, let us consider the case of a diagonal matrix $A$ with large negative entries and $\Delta T = 1$, so that $M \approx -A^{-1}$ will be a diagonal matrix with small positive entries. This means we can ignore its contributions to the system, reducing the network equations to:

$$c(n) = Ry(n-1) + Wx(n)$$
$$y(n) = \sigma(c(n)).$$

This differs only by a minor change of the time indexing from the standard definition of a RNN:

$$\begin{cases} c(t) &= Ry(t) + Wx(t) \\ y(t) &= \sigma(c(t-1)) \end{cases} \tag{2.5}$$

where, summarizing:

- $c(t) \in \mathbb{R}^k$ is the cell state variable, called the cell value at time t.
- $x(t) \in \mathbb{R}^d$ is the cell input data at time t.
- $y(t) \in \mathbb{R}^k$ is the cell output data at time t.
- $R \in \mathbb{R}^{k,k}$
- $W \in \mathbb{R}^{k,d}$
- $\sigma$ is a sigmoid function.
- $\xi(t) \in \mathbb{R}^k$ is the expected output at time t as noted in chapter 1. It's usually called *label* or *ground truth*.

It's important to note that, throughout this work, matrices will be designated with uppercase letters, while vectors will be designated with lowercase letters.

## 2.2. Learning for RNN

The learning process for such an architecture is developed according to the following principles. The dataset is given by a sequence of inputs $x = \{x(t)\}_{t\in\mathbb{N}} \subset \mathbb{R}^d$ that are assumed to influence the dynamics of the sequence of expected outputs $\xi = \{\xi(t)\}_{t\in\mathbb{N}} \subset \mathbb{R}^k$. The pair $(x, \xi)$ is generally called the training dataset, and provides the information for a supervised learning.

The objective is that of learning a dynamics $y = \{y(t)\}_{t\in\mathbb{N}} \subset \mathbb{R}^k$ generated by (2.5), that resembles as much as possible that of $\xi$ over a finite number of steps $T$.

This is formalized by setting a measure of error $e : \mathbb{R}^k \times \mathbb{R}^k \longrightarrow \mathbb{R}_+$ typically given by $e(y, \xi) = \frac{1}{2} \|y - \xi\|^2$ and by defining an error function

$$\mathcal{L} = \sum_{t=1}^{T} e(y(t), \xi(t)). \tag{2.6}$$

The learning process is then aimed at the minimization of the error $\mathcal{L}$ with respect to the parameters that govern the dynamics of $y$, that are the elements of the matrices $R, W$. This is achieved by means of a gradient descent over $\mathcal{L}$

$$\begin{cases} W^{(k+1)} &= W^{(k)} - \eta \frac{\partial \mathcal{L}}{\partial W} \\ R^{(k+1)} &= R^{(k)} - \eta \frac{\partial \mathcal{L}}{\partial R} \end{cases} \tag{2.7}$$

where $\eta$ is called *learning rate* (note that, although in most gradient descent implementations, $\eta$ is not constant, we will not consider this issue on the current work). In general, as for the multilayer perceptron, the training is divided in a forward pass and a backward pass, and, due to the dynamical structure of the problem, in this case, the backward pass is called back propagation through time (BPTT).

Several issues concerning BPTT will be treated in all the following sections. At this point, it is just worth observing that the recursive structure of (2.5) and the presence of several time steps in (2.6) make the computation of the gradient rather cumbersome. Indeed, whilst the main tool for this computation is the familiar chain rule, it may be highly non-trivial to establish the correct dependencies. This issue becomes even more true for more sophisticated architectures as LSTM, described in Chapter 4, for which all the examined literature contain flaws in the computations, due to the wrong applications of the chain rule.

## 2.3. Gradient descent for RNN

In this section we show the traditional derivation of the BPTT algorithm used to implement the gradient descent for the minimization of $\mathcal{L}$ with respect to $W, R$. Recall that this optimization scheme consists on updating the parameters according to the iteration (2.7). The BPTT algorithm allows for an efficient computation of the gradient of $\mathcal{L}$ in equation (2.6) by exploiting the chain rule and the recurrent structure of the network.

Since the loss function $\mathcal{L}$ depends on the output $y(t)$ at all values of $t \in [1, ..., T]$, let us first compute the gradient of $\mathcal{L}$ with respect to $y(t)$. As $y(t)$ depends on $c(t-1)$, it's

obvious that $c(t-1)$ also influences $\mathcal{L}$. So we define:

$$\chi(t) \equiv \nabla_{y(t)}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial y(t)}$$

$$\psi(t) \equiv \nabla_{c(t)}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial c(t)}.$$

Note that, since these are gradients, they will be treated as row vectors. Due to the definition of $\mathcal{L}$, we cannot use the chain rule directly and state that $\psi(t) = \frac{\partial e(y(t+1))}{\partial y(t+1)} \odot \left(\frac{dy(t+1)}{dc(t)}\right)^T$[1]. This is due to the fact that, for any $t$, $y(t)$ influences $y(t+1)$. Once we are aware of this, we can calculate $\chi(t)$ and $\psi(t)$ as:

$$\begin{aligned}\chi(t) &= \frac{\partial e(y(t),\xi(t))}{\partial y(t)} + \frac{\partial \mathcal{L}}{\partial c(t)}\frac{\partial c(t)}{\partial y(t)} \\ &= \frac{\partial e(y(t),\xi(t))}{\partial y(t)} + \psi(t)R \end{aligned} \tag{2.8}$$

because $\frac{\partial c(t)}{\partial y(t)} = R$, and, using (2.8),

$$\begin{aligned}\psi(t) &= \frac{\partial \mathcal{L}}{\partial y(t+1)} \odot \left(\frac{d\, y(t+1)}{d\, c(t)}\right)^T = \chi(t+1) \odot \left(\frac{d\, y(t+1)}{d\, c(t)}\right)^T \\ &= \left[\frac{\partial e\big(y(t+1),\xi(t+1)\big)}{\partial y(t+1)} + \psi(t+1)R\right] \odot \big(\sigma'(c(t))\big)^T. \end{aligned} \tag{2.9}$$

We have thus obtained two equations that progress in the backward direction of $t$, where $\chi(T+1)$ is initialized as a 0 vector. Now, using these equations along with equation (2.5) and the chain rule, we have[2]:

$$\frac{\partial \mathcal{L}}{\partial R_{ij}} = \sum_{t=1}^{T-1} \frac{\partial \mathcal{L}}{\partial c_i(t)}\frac{\partial c_i(t)}{\partial R_{ij}} = \sum_{t=1}^{T-1} \psi_i(t) y_j(t)$$

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial R} = \sum_{t=1}^{T-1} \psi^T(t) y^T(t). \tag{2.10}$$

Analogously:

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{t=1}^{T-1} \psi^T(t) x^T(t). \tag{2.11}$$

These, together with (2.8) and (2.9) are the equations used to train the network through BPTT.

[1] $\odot$ denotes the point-wise multiplication of two vectors, and $\frac{dy(t+1)}{c(t)} = \sigma'(c(t))$ is a column vector where $\sigma'$ is computed component-wise.

[2] In section 3.3 we will explain the origin of these formulas in depth. Although it should be pretty straightforward, note that on $\psi^T(t), y^T(t)$ and $x^T(t)$, i.e., when used with a vector, $T$ denotes the transpose vector, not the maximum time index used.

## 2.4. Stability and the problem of vanishing/exploding gradients

A first observation on (2.5) is that it is possible to characterize its stability, intended as the asymptotic convergence in the absence of inputs, in terms of the spectral radius of the recurrence matrix $R$.

**Proposition.** *Let $\sigma$ be a smooth sigmoid with $|\sigma'| < 1$. For the iteration (2.5) to be stable, it's necessary and sufficient that the spectral radius $\mu$ of the matrix $R$ satisfies $\mu < 1$.*

*Proof.* Let us consider the autonomous system $c(t) = R\sigma(c(t-1)) = F(c(t-1))$ with $F(x) = R\sigma(x), \quad F : \mathbb{R}^k \longrightarrow \mathbb{R}^k$.

Let $p^*$ be a point such that $p^* = F(p^*)$. Now, we use the Taylor series of $F$ around $p^*$ so that:

$$F(x) = F(p^*) + J_F(p^*)(x - p^*) + o(\|x - p^*\|)$$

where $J_F(p^*)$ denotes the Jacobian matrix of $F$ at $p^*$. As we've previously chosen $p^*$ so that $p^* = R\sigma(p^*)$, we now have:

$$c(t+1) = p^* + J_F(p^*)(c(t) - p^*) + o(\|c(t) - p^*\|).$$

If we let $\alpha_t = c(t) - p^*$, the equation becomes $\alpha_{t+1} \approx J_F(p^*)\alpha_t$ and, by iterating this, we get to the main point:

$$\alpha_t \to 0 \iff \text{the spectral radius of } J_F(p^*) \text{ is } < 1.$$

Now, we have $(F(x))_i = (R\sigma(x))_i = \sum_{j=0}^{k-1} R_{ij}\sigma(x_j)$, so that

$$\frac{\partial F_i}{\partial x_j}(x) = R_{ij}\sigma'(x_j) \Rightarrow J_F(x) = \begin{pmatrix} \sigma'(x_1) & 0 & \cdot & \cdot & 0 \\ 0 & \sigma'(x_2) & 0 & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot & \sigma'(x_k) \end{pmatrix} \cdot R$$

Using $diag[v]$ to denote a diagonal matrix in which the elements of $v$ occupy the main diagonal, we can rewrite the equation as

$$J_F(x) = diag[\sigma'(x)] \cdot R.$$

If $\sigma'$ is bounded between 0 and 1, then the necessary and sufficient condition for local stability of the autonomous system is that the spectral radius of $R$ is $< 1$, which is exactly what we mentioned earlier. ■

An example of sigmoid function that satisfies the assumptions is the commonly used $\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \in [-1, 1]$. Here, $\sigma'(x) = 1 - \sigma(x)^2 \in [-1, 1]$.

Even considering a RNN on a finite time $T$ on the previous proposition, the process of learning the optimal matrices $W, R$ can still be problematic in practice due to some phenomena know as *vanishing gradients* and *exploding gradients* that can occur during Back Propagation Through Time (BPTT) training.

Note that, due to equations (2.10), (2.11), all of the quantities used for updating the network's parameters are proportional to $\psi(t)$. This is exactly what causes problems when training these kind of networks by making it impossible to relate two distinct outputs $y(t), y(t_2)$ with $t_2 >> t$.

Let's see what fraction of $\psi(t)$ is retained from back propagating $\psi(t_2)$. If these contributions to $\psi(t)$ were well-behaved numerically, the parameters resulting from gradient descent learning would be able to incorporate long-term interactions among the samples. Expanding the equation (2.9) we have:

$$\frac{\partial\,\psi(t)}{\partial\,\psi(t_2)} = \frac{\partial\,\psi(t)}{\partial\,\psi(t+1)}\frac{\partial\,\psi(t+1)}{\partial\,\psi(t_2)} = \left[R \cdot diag\left[\frac{dy(t+1)}{dc(t)}\right]\right]\frac{\partial\,\psi(t+1)}{\partial\,\psi(t_2)}$$

where $\frac{dy(t+1)}{dc(t)} = \sigma'(c(t))$, so that

$$\frac{\partial\,\psi(t)}{\partial\,\psi(t_2)} = \prod_{\tau=t}^{t_2-1} R \cdot diag\left[\sigma'(c(t))\right]. \tag{2.12}$$

This means that the magnitude of the Jacobian matrix $\frac{\partial\,\psi(t)}{\partial\,\psi(t_2)}$ depends on the product of $t_2 - t$ terms. So, although the system is theoretically stable, as the number of samples $T$ can be very large, and the distance between $t$ and $t_2$ is supposed to be large too, we can find some problems:

- Suppose that $\sigma$ is a sigmoid function with $|\sigma'| < 1$ and that the spectral radius $\mu$ of the matrix $R$ satisfies $\mu < 1$, which means that there exists a matrix norm such that $\|R\| < 1$, and $\|diag\left[\sigma'(c(t))\right]\| < 1$, so that:

$$\left\|\frac{\partial\,\psi(t)}{\partial\,\psi(t_2)}\right\| \leq \|R\|^{t_2-t}\prod_{\tau=t}^{t_2-1}\left\|diag\left[\sigma'(c(t))\right]\right\| \leq \|R\|^{t_2-t} \approx 0.$$

  Thus, when the distance between two time steps is big enough, the network is not able to capture the dependencies between those two time steps, so it wont learn properly.

- Conversely, if the spectral radius of $R$ does not satisfy the stability requirement $\mu < 1$, depending on the behaviour of $c(t)$, there are two possible outcomes:
  - If the cell state leaves the quasi-linear region of the sigmoid (the region in which it can be approximated by a linear function), the output can be saturated by that sigmoid, so we would have $\sigma'(c(t)) \approx 0$, and the result could be again $\left\|\frac{\partial\,\psi(t)}{\partial\,\psi(t_2)}\right\| \approx 0$, so the network wont be able to learn long-term dependencies as explained previously.
  - If the cell state starts in the quasi-linear region of the sigmoid, and stays there for a large number of steps, then $\|diag\left[\sigma'(c(t))\right]\| \approx 1$, and $\left\|\frac{\partial\,\psi(t)}{\partial\,\psi(t_2)}\right\|$ may grow, resulting in high magnifications of numerical errors or even a possible overflow. This means that, even if the the problem has a finite $T$, $\left\|\frac{\partial\,\psi(t)}{\partial\,\psi(t_2)}\right\|$ may become useless due to the limitations of the floating-point representation of real numbers, meaning that the network wont be able to correctly interpret the long-term dependencies.

So, summarizing, if the distance between $t$ and $t_2$ is large enough, the fraction of $\psi(t)$ retained from back propagating $\psi(t_2)$ will either vanish or explode, so the parameters resulting from gradient descent learning won't be able to incorporate long-term interactions among the samples.

As we've just seen, for a large number of training samples the gradients may either explode, or vanish outside critical points, making gradient descent useless for training. The most effective solution to this problem is the use of Long Short-Term Memory (LSTM) cells, that will be described in Chapter 4.

# CHAPTER 3

# Back Propagation Through Time Theorem

In this section we present a BPTT algorithm for learning a general Recurrent Neural Network given a loss function defined over a finite amount of time-dependent errors [2]. This is based on an iterative differentiation scheme for the computation of the gradient, to be proved in the main theorem of the section. This algorithm, in particular, will be shown to produce the same BPTT formulas derived in the previous chapter. Since the main argument of the proof does not require the network parametrization to remain constant in time, we state and prove the result for a time-dependent set of parameters and output dimensions. The main objects we are going to be using on the theorem are the following[1]:

- Let $T > 0$ be the number of time steps used on the network. Let $\{d_t\}_{t=0}^{T} \subset \mathbb{N}$ be the output dimension at time $t$, and let $\{n_t\}_{t=0}^{T} \subset \mathbb{N}$ be the dimension of network parameters at time $t$. Now, we let $f_t \in C^1(\mathbb{R}^{d_{t-1} \times n_t}, \mathbb{R}^{d_t})$ be the main function describing the network, and denote by $\mathcal{L}_t \in C^1(\mathbb{R}^{d_t})$ the error at time $t$ and by $\Theta_t \in \mathbb{R}^{n_t}$ the network parameters at time step $t$.

- For $t = 1, ..., T$, denote by $P_t = (\Theta_t, \Theta_{t-1}, ..., \Theta_1) \in \mathbb{R}^{n_t} \times ... \times \mathbb{R}^{n_1}$, and define recursively the functions $z_t \in C^1(\mathbb{R}^{n_t} \times ... \times \mathbb{R}^{n_1}, \mathbb{R}^{d_t})$ by

$$\begin{cases} z_t(P_t) = f_t(z_{t-1}(P_{t-1}), \Theta_t), & t = 2, ..., T \\ z_1(\Theta_1) = f_1(z_0, \Theta_1), & z_0 \in \mathbb{R}^{d_0} \end{cases} \tag{3.1}$$

- Let $\mathcal{L} \in C^1(\mathbb{R}^{n_t} \times ... \times \mathbb{R}^{n_1}, \mathbb{R})$ be a loss function given by:

$$\mathcal{L}(P_T) = \sum_{t=1}^{T} \mathcal{L}_t(z_t(P_t))$$

Our goal to differentiate $\mathcal{L}$ with respect to one of the scalar parameters $\theta \in \Theta$ is achieved by the following theorem.

[1] Note that in this case we are using the time step $t$ as a sub-index instead of a function parameter in order to simplify the notation.

**Theorem.** *under the above notations and assumptions, for $\theta$ a component of $P_t$ and $t \in \{1, ..., T\}$, denote by*

$$\begin{cases} \partial_\theta^+ z_t = \partial_2 f_t(z_{t-1}, \Theta_t) \frac{\partial \Theta_t}{\partial \theta} \\ \frac{\partial z_t}{\partial z_{t-1}} = \partial_1 f_t(z_{t-1}, \Theta_t) \end{cases} \tag{3.2}$$

*Where $\partial_1 f_t$ and $\partial_2 f_t$ stand for the Jacobian matrices of $f_t$ with respect to its first $\mathbb{R}^{d_{t-1}}$ variables and to its second $\mathbb{R}^{n_t}$ variables. Thus, $\partial_\theta^+ z_t$ is a column vector of $\mathbb{R}^{d_t}$ and $\frac{\partial z_t}{\partial z_{t-1}}$ is a matrix of $d_t$ rows and $d_{t-1}$ columns.*

*Now, for each $k = 1, ..., T$, let $\Gamma_k \in \mathbb{R}^{d_k}$ be defined recursively as:*

$$\begin{cases} \Gamma_T = \nabla \mathcal{L}_T(z_T) \\ \Gamma_k = \nabla \mathcal{L}_t(z_k) + \Gamma_{k+1} \frac{\partial z_{k+1}}{\partial z_k}, \quad k = T-1, ..., 1. \end{cases} \tag{3.3}$$

*Then*

$$\frac{\partial \mathcal{L}(P_T)}{\partial \theta} = \sum_{k=T}^{1} < \Gamma_k, \partial_\theta^+ z_k > . \tag{3.4}$$

Note that we have denoted the inverse time index with $k$ and not with $t$. As it will be clear from the proof of the theorem, this formulation is indeed the result of a rearrangement of the operations performed in the forward pass, and $k$ indexes the actual propagation of recursive differentiation backward in time.

## 3.1. Usage of the theorem: a BPTT algorithm

Imagine we have a system (3.1) that we want to train using BPTT. Provided we can get reliable (explicit) expressions for the Jacobians (3.2), , an efficient algorithm can be obtained from the above expressions as follows:

1. Initialize the parameters $P_1$ and $z_0$.

2. Compute the forward pass and store every $z_t$, $t = 1, ..., T$.

3. The theorem allows us to save lots of memory by computing elements in the order at which they appear at (3.4) and therefore, not having to store them all, and performing each differentiation only once. The main iteration is:

   - compute $\Gamma_T, \partial_\theta^+ z_T$;
   - $\frac{\partial \mathcal{L}(P_T)}{\partial \theta} = < \Gamma_T, \partial_\theta^+ z_T >$;
   - for $k = T-1, ..., 1$:
     - compute $\partial_\theta^+ z_k$, $\frac{\partial z_{k+1}}{\partial z_k}$ with (3.2);
     - update $\Gamma_k$ with (3.3);
     - $\frac{\partial \mathcal{L}(P_T)}{\partial \theta} = \frac{\partial \mathcal{L}(P_T)}{\partial \theta} + < \Gamma_k, \partial_\theta^+ z_k >$;

## 3.2. Proof of the theorem

First of all, by the definition of $\mathcal{L}$ and by the chain rule, we have:

$$\frac{\partial \mathcal{L}(P_T)}{\partial \theta} = \sum_{t=1}^{T} < \nabla \mathcal{L}_t(z_t(P_t)), \frac{\partial z_t(P_t)}{\partial \theta} > . \tag{3.5}$$

Taking into account the definition of $z_t$ and applying the chain rule again:

$$\frac{\partial z_t(P_t)}{\partial \theta} = \partial_1 f_t(z_{t-1}(P_{t-1}), \Theta_t) \frac{\partial z_{t-1}(P_{t-1})}{\partial \theta} + \partial_2 f_t(z_{t-1}(P_{t-1}), \Theta_t) \frac{\partial \Theta_t}{\partial \theta} \tag{3.6}$$

which are two matrix-vector multiplications in which the first term is the matrix that we have denoted by $\frac{\partial z_t}{\partial z_{t-1}}$, and the second term is $\partial_\theta^+ z_t$ as in (3.2). Now, for this intermediate step we're going to denote $v_t = \frac{\partial z_t(P_t)}{\partial \theta}$, $M_t = \frac{\partial z_t}{\partial z_{t-1}}$, so that we can rewrite (3.6) as:

$$\begin{aligned} v_t &= M_t v_{t-1} + \partial_\theta^+ z_t \\ &= M_t M_{t-1} v_{t-2} + M_t \partial_\theta^+ z_{t-1} + \partial_\theta^+ z_t \\ &= \ldots = \sum_{l=0}^{t-1} \left(M_t \ldots M_{t-l+1}\right) \partial_\theta^+ z_{t-l} \end{aligned}$$

where we assume that, in the term $l = 0$, the matrix product is just the identity matrix. As $M_k$'s are non commuting matrices, we use the following notation to denote an ordered matrix product:

$$v_t = \sum_{l=0}^{t-1} \left( \uparrow \prod_{j=0}^{l-1} M_j \right) \partial_\theta^+ z_{t-l}$$

and finally, using a variable change with $k = t - l$,

$$v_t = \sum_{k=1}^{t} \left( \uparrow \prod_{j=0}^{t-k-1} M_{t-j} \right) \partial_\theta^+ z_k$$

where, again, the product when $k = t$ is assumed to be the identity.

To simplify notation, we omit to write the $P_k$ parameter dependency of every $z_k(P_k)$ and define

$$\frac{\partial z_t}{\partial z_k} := \begin{cases} \uparrow \prod_{j=0}^{t-k-1} M_{t-j} = \frac{\partial z_t}{\partial z_{t-1}} \frac{\partial z_{t-1}}{\partial z_{t-2}} \ldots \frac{\partial z_{k+1}}{\partial z_k} & k < t \\ I & k = t \end{cases} \tag{3.7}$$

so that we obtain

$$\frac{\partial z_t}{\partial \theta} = \sum_{k=1}^{t} \frac{\partial z_t}{\partial z_k} \partial_\theta^+ z_k.$$

Therefore, (3.5) needs:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^{T} < \nabla \mathcal{L}_t(z_t), \sum_{k=1}^{t} \frac{\partial z_t}{\partial z_k} \partial_\theta^+ z_k > . \tag{3.8}$$

Now, as the Equation (3.8) is made up of two finite sums, we can apply Fubini's theorem to rearrange the sums so that me move from $\sum_{t=1}^{T}\sum_{k=1}^{t}$ to $\sum_{k=1}^{T}\sum_{t=k}^{T}$, and get

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{k=1}^{T} < \Gamma_k, \partial_\theta^+ z_k > \tag{3.9}$$

with

$$\Gamma_k := \sum_{t=k}^{T} \nabla \mathcal{L}_t(z_t) \frac{\partial z_t}{\partial z_k}.$$

This means the only thing left is to check that $\Gamma_k$ satisfies the recursive relation (3.3). In order to see this, we insert (3.7) in the above expression:

$$\begin{aligned}
\Gamma_k :=& \nabla \mathcal{L}_k(z_k) + \sum_{t=k+1}^{T} \nabla \mathcal{L}_t(z_t) \left( \uparrow \prod_{j=k+1}^{t} \frac{\partial z_j}{\partial z_{j-1}} \right) \\
=& \nabla \mathcal{L}_k(z_k) + \left[ \sum_{t=k+1}^{T} \nabla \mathcal{L}_t(z_t) \left( \uparrow \prod_{j=k+2}^{t} \frac{\partial z_j}{\partial z_{j-1}} \right) \right] \frac{\partial z_{k+1}}{\partial z_k} \\
=& \nabla \mathcal{L}_k(z_k) + \Gamma_{k+1} \frac{\partial z_{k+1}}{\partial z_k}.
\end{aligned}$$

This concludes the proof. ■

## 3.3. Application to RNN

Finally, let's apply the theorem to traditional RNN and show how it allows to obtain the BPTT formulas (2.8) - (2.11).
We first restate formula (2.5) as:

$$z_t = R\sigma(z_{t-1}) + Wx(t)$$

and, for simplicity, assume

$$\mathcal{L} = \frac{1}{2} \sum_{t=1}^{T} \|\sigma(z_t) - \xi(t+1)\|^2 .$$

It's evident that in this particular case, the system's parameters remain constant through the forward propagation, so $P_t = \Theta_1 = \{R \in \mathbb{R}^{k,k}, W \in \mathbb{R}^{k,d}\}$ for every $t = 1, ..., T$. Now we need to calculate the algebraic expression of equations (3.2) for each $R_{ij}, W_{ij}$[2]:

$$\frac{(\partial z_t)_l}{(\partial z_{t-1})_m} = \frac{\partial}{(\partial z_{t-1})_m} \left( \sum_{q=1}^{k} R_{lq} \sigma(z_{t-1})_q \right) = R_{lm} \sigma'(z_{t-1})_m$$

$$\Rightarrow \frac{\partial z_t}{\partial z_{t-1}} = R \cdot diag[\sigma'(z_{t-1})] \tag{3.10}$$

[2]These are the only formulas of the present work in which $\delta$ denotes the Kronecker delta.

$$\left(\partial^{+}_{R_{ij}} z_t\right)_l = \frac{\partial}{\partial R_{ij}} \left( \sum_{q=1}^{k} R_{lq} \sigma(z_{t-1})_q \right) = \delta_{il}\, \sigma(z_{t-1})_j \tag{3.11}$$

$$\left(\partial^{+}_{W_{ij}} z_t\right)_l = \frac{\partial}{\partial W_{ij}} \left( \sum_{q=1}^{d} W_{lq} x(t)_q \right) = \delta_{il}\, x(t)_j. \tag{3.12}$$

Finally, we calculate the expressions for equations (3.3) so that we have everything needed:

$$\begin{cases} \Gamma_{T-1} = \big(\sigma(z_{T-1}) - \xi(T)\big) \odot \sigma'(z_{T-1}) \\ \Gamma_t = \big(\sigma(z_t) - \xi(t+1)\big) \odot \sigma'(z_t) + \Gamma_{t+1} \cdot R \cdot diag\big[\sigma'(z_t)\big], \quad t = T-2, ..., 1 \end{cases} \tag{3.13}$$

This means that, for any $i = 1, ..., k$ we have:

$$\begin{cases} (\Gamma_{T-1})_i = \big[\sigma(z_{T-1})_i - \xi(T)_i\big] \sigma'(z_{T-1})_i \\ (\Gamma_t)_i = \Big[\sigma(z_t)_i - \xi(t+1)_i + \sum_{l=1}^{k} (\Gamma_{t+1})_l\, R_{li}\Big]\, \sigma'(z_t)_i, \quad t = T-2, ..., 1 \end{cases} \tag{3.14}$$

So, using the algorithm, we obtain:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial R_{ij}} &= \sum_{t=1}^{T-1} (\Gamma_t)_i\, \sigma(z_{t-1})_j \\ &= \sum_{t=1}^{T-1} \left[ \sigma(z_t)_i - \xi(t+1)_i + \sum_{l=1}^{k} (\Gamma_{t+1})_l\, R_{li} \right] \sigma'(z_t)_i\, \sigma(z_{t-1})_j \end{aligned}$$

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial W_{ij}} &= \sum_{t=1}^{T-1} (\Gamma_t)_i\, x(t)_j \\ &= \sum_{t=1}^{T-1} \left[ \sigma(z_t)_i - \xi(t+1)_i + \sum_{l=1}^{k} (\Gamma_{t+1})_l\, R_{li} \right] \sigma'(z_t)_i\, x(t)_j \end{aligned}$$

Those equations coincide with the expressions for the gradient given by (2.8) - (2.11), which read

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial R_{ij}} &= \sum_{t=1}^{T-1} \frac{\partial \mathcal{L}}{\partial c_i(t)} \frac{\partial c_i(t)}{\partial R_{ij}} = \sum_{t=1}^{T-1} \psi(t)_i y(t)_j \\ &= \sum_{t=1}^{T-1} \left[ y(t+1)_i - \xi(t+1)_i + \sum_{l=1}^{k} \psi(t+1)_l R_{li} \right] \sigma'\big(c(t)\big)_i\, \sigma\big(c(t-1)\big)_j \end{aligned}$$

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial W_{ij}} &= \sum_{t=1}^{T-1} \frac{\partial \mathcal{L}}{\partial c_i(t)} \frac{\partial c_i(t)}{\partial W_{ij}} = \sum_{t=1}^{T-1} \psi(t)_i x(t)_j \\ &= \sum_{t=1}^{T-1} \left[ y(t+1)_i - \xi(t+1)_i + \sum_{l=1}^{k} \psi(t+1)_l R_{li} \right] \sigma'\big(c(t)\big)_i\, x(t)_j. \end{aligned}$$

Taking into account some differences in notation such as $z_t \equiv c(t),\ y(t+1) = \sigma\big(c(t)\big) = \sigma(z_t)$, it's completely clear that both formulas are the same whether calculated with the theorem or with the first method.

CHAPTER 4

# Long Short-Term Memory Cells

In this section we are going to study LSTM, one of the most relevant Recurrent Neural Network architectures designed to overcome the main problems of traditional RNN given by the vanishing/exploding gradients, described in sections 2.3, 2.4. The main novelty of Long Short-Term Memory (from now on simply LSTM) cells is that of incorporating three trainable gates: input gate, forget gate and output gate (along with three peephole connections on some variations of this LSTM) to control the cell's internal state. The schematic of the LSTM block we are going to use on this work is presented in Figure 4.1. The main references used on this chapter are [4], [6], [12], [5] and [11].

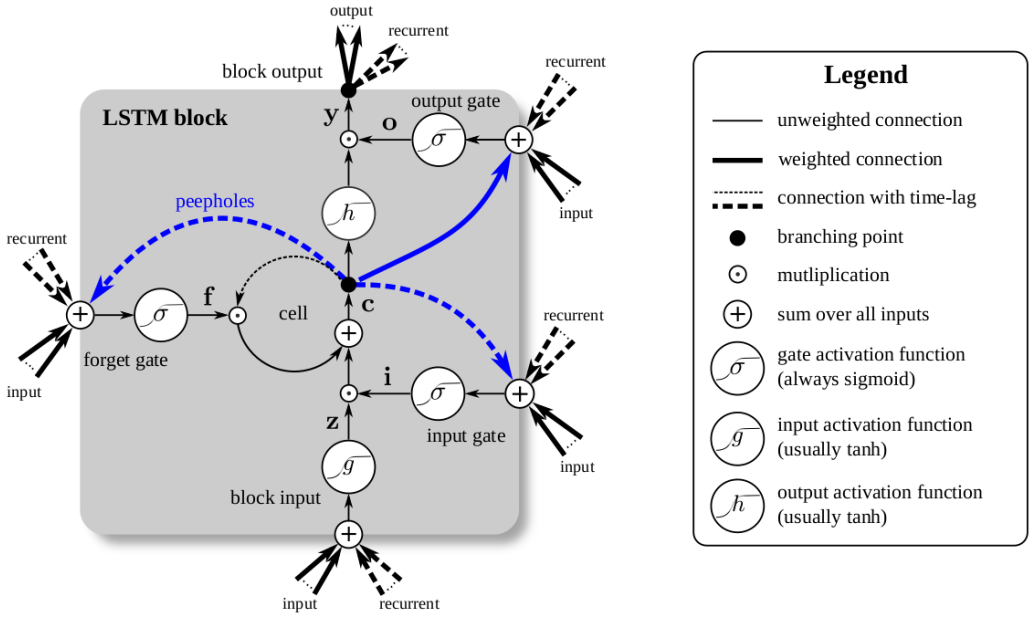



Figure 4.1: Figure of LSTM block. Extracted from Greff et al. 2017.

Again, as in the RNN chapter, the main data is going to be:

- $\xi(t) \in \mathbb{R}^k$: label at time t (ground truth).
- $x(t) \in \mathbb{R}^d$: input data at time t.

- $c(t) \in \mathbb{R}^k$: internal cell state variable at time t.
- $y(t) \in \mathbb{R}^k$: output data at time t.

Once we have defined the data, we need to define the matrices and vectors we're going to use:

- $W^z, W^i, W^f, W^o \in \mathbb{R}^{k,d}$: Weight matrices of the connections from $x(t)$ to block input, input gate, forget gate and output gate respectively.
- $R^z, R^i, R^f, R^o \in \mathbb{R}^{k,k}$: Weight matrices of the connections from $y(t)$ to block input, input gate, forget gate and output gate respectively.
- $p^i, p^f, p^o \in \mathbb{R}^k$: Weight vectors of the peephole connections to block input, input gate, forget gate and output gate respectively.

Now, we can define the block input, each of the gates, and the cell itself.

Block input:

$$\begin{cases} \overline{z}(t+1) = W^z x(t) + R^z y(t) \\ z(t+1) = g\big(\overline{z}(t+1)\big) \end{cases} \tag{4.1}$$

Input gate:

$$\begin{cases} \overline{i}(t+1) = W^i x(t) + R^i y(t) + p^i \odot c(t) \\ i(t+1) = \sigma\big(\overline{i}(t+1)\big) \end{cases} \tag{4.2}$$

Forget gate:

$$\begin{cases} \overline{f}(t+1) = W^f x(t) + R^f y(t) + p^f \odot c(t) \\ f(t+1) = \sigma\big(\overline{f}(t+1)\big) \end{cases} \tag{4.3}$$

Output gate:

$$\begin{cases} \overline{o}(t+1) = W^o x(t) + R^o y(t) + p^o \odot c(t+1) \\ o(t+1) = \sigma\big(\overline{o}(t+1)\big) \end{cases} \tag{4.4}$$

Main LSTM iteration:

$$\begin{cases} c(t+1) &= i(t+1) \odot z(t+1) + f(t+1) \odot c(t) \\ y(t) &= o(t) \odot h\big(c(t)\big) \end{cases} \tag{4.5}$$

Here, $\sigma$, $g$ and $h$ are sigmoid functions and $\odot$ denotes the point-wise multiplication of two vectors.

## 4.1. BPTT Theorem on LSTM

Once the forward propagation formulas of LSTM cells are clear, we can use the BPTT Theorem as explained in Chapter 3 to calculate the exact formulas needed to train the model. However, in this case the formulas are much more complex than in basic RNN. We first rewrite the system of equations described above in terms of a new combined variable $\zeta$ as:

$$\zeta_t = \begin{pmatrix} c_t \\ y_t \end{pmatrix}, \quad F(\zeta, \Theta, x) = \begin{pmatrix} \phi(\zeta, \Theta, x) \\ \psi(\zeta, \Theta, x) \end{pmatrix}$$

where $\phi, \psi$ are just simple rewrites of the previous equations to use them as functions of $c, y, x$ and the set of parameters $\Theta \coloneqq \{W^z, W^i, W^f, W^o, R^z, R^i, R^f, R^o, p^i, p^f, p^o\}$, which remain constant through the forward propagation, as happened on the RNN.

$$\begin{aligned}
c(t+1) &= i(t+1) \odot z(t+1) + f(t+1) \odot c(t) = \phi\big(y(t), c(t), \Theta, x(t)\big) \\
&=: \phi(\zeta_t, \Theta, x_t) \\
y(t+1) &= o(t+1) \odot h\big(c(t+1)\big) = \sigma\big(W^o x(t) + R^o y(t) + p^o \odot c(t+1)\big) \odot h\big(c(t+1)\big) \\
&= \sigma\Big(W^o x(t) + R^o y(t) + p^o \odot \phi\big(y(t), c(t), \Theta, x(t)\big)\Big) \odot h\Big(\phi\big(y(t), c(t), \Theta, x(t)\big)\Big) \\
&= \omega\Big(y(t), c(t), \Theta, x(t)\Big) \odot h\Big(\phi\big(y(t), c(t), \Theta, x(t)\big)\Big) \\
&= \psi\big(y(t), c(t), \Theta, x(t)\big) \\
&=: \psi(\zeta_t, \Theta, x_t)
\end{aligned}$$

Once we have reduced the LSTM equations to the general recurrent form of Chapter 3, in order to use the theorem and obtain the explicit BPTT formulas we need to calculate the algebraic expression of equations (3.2).
On one side we have:

$$\frac{\partial\, \zeta_t}{\partial\, \zeta_{t-1}} = \partial_1 F(\zeta, \Theta, x) = \left(\begin{array}{c|c} \partial_c \phi(\zeta, \Theta, x) & \partial_y \phi(\zeta, \Theta, x) \\ \hline \partial_c \psi(\zeta, \Theta, x) & \partial_y \psi(\zeta, \Theta, x) \end{array}\right)$$

which is a Jacobian matrix easy to calculate.

Derivatives needed to calculate the Jacobian matrix:

$$z_l = g\Big(\underbrace{\sum_{r=1}^{k} R^z_{lr} y_r + (W^z x)_l}_{\overline{z}_l}\Big) \qquad \Rightarrow \begin{cases} \dfrac{\partial\, z_l}{\partial\, c_j} = 0 \\ \dfrac{\partial\, z_l}{\partial\, y_j} = g'(\overline{z}_l) R^z_{lj} \end{cases}$$

$$i_l = \sigma\Big(\underbrace{\sum_{r=1}^{k} R^i_{lr} y_r + p^i_l c_l + (W^z x)_l}_{\overline{i}_l}\Big) \qquad \Rightarrow \begin{cases} \dfrac{\partial\, i_l}{\partial\, c_j} = \sigma'(\overline{i}_l) p^i_l \delta_{lj} \\ \dfrac{\partial\, i_l}{\partial\, y_j} = \sigma'(\overline{i}_l) R^i_{lj} \end{cases}$$

$$f_l = \sigma\Big(\underbrace{\sum_{r=1}^{k} R^f_{lr} y_r + p^f_l c_l + (W^f x)_l}_{\overline{f}_l}\Big) \qquad \Rightarrow \begin{cases} \dfrac{\partial\, f_l}{\partial\, c_j} = \sigma'(\overline{f}_l) p^f_l \delta_{lj} \\ \dfrac{\partial\, f_l}{\partial\, y_j} = \sigma'(\overline{f}_l) R^f_{lj} \end{cases}$$

$$\omega_l = \sigma\Big(\underbrace{\sum_{r=1}^{k} R^o_{lr} y_r + p^o_l \phi_l + (W^o x)_l}_{\overline{\omega}_l}\Big) \qquad \Rightarrow \begin{cases} \dfrac{\partial\, \omega_l}{\partial\, c_j} = \sigma'(\overline{\omega}_l) p^o_l \dfrac{\partial\, \phi_l}{\partial\, c_j} \\ \dfrac{\partial\, \omega_l}{\partial\, y_j} = \sigma'(\overline{\omega}_l)\big(R^o_{lj} + p^o_l \dfrac{\partial\, \phi_l}{\partial\, y_j}\big) \end{cases}$$

$$\phi_l = i_l z_l + f_l c_l \qquad \Rightarrow \begin{cases} \dfrac{\partial\, \phi_l}{\partial\, c_j} = \dfrac{\partial\, i_l}{\partial\, c_j} z_l + i_l \underbrace{\dfrac{\partial\, z_l}{\partial\, c_j}}_{=0} + \dfrac{\partial\, f_l}{\partial\, c_j} c_l + f_l \delta_{jl} \\ \dfrac{\partial\, \phi_l}{\partial\, y_j} = \dfrac{\partial\, i_l}{\partial\, y_j} z_l + i_l \dfrac{\partial\, z_l}{\partial\, y_j} + \dfrac{\partial\, f_l}{\partial\, y_j} c_l + f_l \delta_{jl} \end{cases}$$

$$\psi_l = \omega_l h(c_l) \qquad \Rightarrow \begin{cases} \dfrac{\partial\, \psi_l}{\partial\, c_j} = \dfrac{\partial\, \omega_l}{\partial\, c_j} h(\phi_l) + \omega_l h'(\phi_l) \dfrac{\partial\, \phi_l}{\partial\, c_j} \\ \qquad\quad = \Big(h(\phi_l)\sigma'(\omega_l) p^o_l + \omega_l h'(\phi_l)\Big) \dfrac{\partial\, \phi_l}{\partial\, c_j} \\ \dfrac{\partial\, \psi_l}{\partial\, y_j} = \dfrac{\partial\, \omega_l}{\partial\, y_j} h(\phi_l) + \omega_l h'(\phi_l) \dfrac{\partial\, \phi_l}{\partial\, y_j} \\ \qquad\quad = h(\phi_l)\sigma'(\overline{\omega}_l)\big(R^o_{lj} + p^o_l \dfrac{\partial\, \phi_l}{\partial\, y_j}\big) + \omega_l h'(\phi_l) \dfrac{\partial\, \phi_l}{\partial\, y_j} \end{cases}$$

Thus, the Jacobian of $F$ with respect to its $\zeta$ variables reads:

$$\begin{aligned}
\frac{\partial \phi_l}{\partial c_j} &= \big(\sigma'(\bar{i}_l)p_l^i z_l \ + \ \sigma'(\overline{f}_l)p_l^f c_l + f_l\big)\delta_{lj} \\
\frac{\partial \phi_l}{\partial y_j} &= z_l\sigma'(\bar{i}_l)R_{lj}^i \ + \ i_l g'(\overline{z}_l)R_{lj}^z \ + \ c_l\sigma'(\overline{f}_l)R_{lj}^f \\
\frac{\partial \psi_l}{\partial c_j} &= \Big(h(c_l)\sigma'(\overline{\omega}_l)p_l^o \ + \ \omega_l h'(\phi_l)\Big)\Big(\sigma'(\bar{i}_l)p_l^i z_l \ + \ \sigma'(\overline{f}_l)p_l^f c_l + f_l\Big)\delta_{lj} \\
\frac{\partial \psi_l}{\partial y_j} &= h(\phi_l)\sigma'(\overline{\omega}_l)R_{lj}^o \\
&\quad + \Big(h(\phi_l)\sigma'(\overline{\omega}_l)p_l^o \ + \ \omega_l h'(\phi_l)\Big)\Big(z_l\sigma'(\bar{i}_l)R_{lj}^i \ + \ i_l g'(\overline{z}_l)R_{lj}^z \ + \ c_l\sigma'(\overline{f}_l)R_{lj}^f\Big)
\end{aligned}$$

On the other side, we would need to calculate $\partial_\theta^+ \zeta_t$ for each $\theta \in \Theta$. In this case we are going to exemplify the procedure just with the parameters $R^z$.

- First, for $\phi(\zeta, \Theta, x)$ we have:

$$\begin{aligned}
\Big(\partial^+_{R^z_{lj}} \phi(\zeta,\Theta,x)\Big)_r &= i(\zeta,\Theta,x)_r\sigma'(\overline{z}(\zeta,\Theta,x)_r)\frac{\partial}{\partial R^z_{lj}}\sum_{s=1}^{k} R^z_{rs}y_s \\
&= i(\zeta,\Theta,x)_r\sigma'(\overline{z}(\zeta,\Theta,x)_r)\delta_{rl}y_j
\end{aligned}$$

  where $\delta$ is the kronecker delta.

$$\Rightarrow \partial^+_{R^z_{lj}} \phi(\zeta,\Theta,x) = \Big[i(\zeta,\Theta,x)_l\sigma'(\overline{z}(\zeta,\Theta,x)_l)y_j\Big]e_l$$

  with $e_l$ the canonical vector of zeros with 1 on the $l-th$ position.

- Then, for $\psi(\zeta, \Theta, x)$ we have:

$$\begin{aligned}
&\Big(\partial^+_{R^z_{lj}} \psi(\zeta,\Theta,x)\Big)_r = \partial^+_{R^z_{lj}}\Big(\omega(\zeta,\Theta,x)_r \odot h\big(\phi(\zeta,\Theta,x)_r\big)\Big) \\
&= \sigma'(\overline{\omega}(\zeta,\Theta,x)_r)p_r^o\frac{\partial \phi}{\partial R^z_{lj}}(\zeta,\Theta,x)\ h\big(\phi(\zeta,\Theta,x)_r\big) \\
&\quad + \omega(\zeta,\Theta,x)_r h'\big(\phi(\zeta,\Theta,x)_r\big)\frac{\partial \phi}{\partial R^z_{lj}}(\zeta,\Theta,x) \\
&= \Big(\sigma'(\overline{\omega}(\zeta,\Theta,x)_r)p_r^o\ h\big(\phi(\zeta,\Theta,x)_r\big) + \omega(\zeta,\Theta,x)_r h'\big(\phi(\zeta,\Theta,x)_r\big)\Big)\frac{\partial \phi}{\partial R^z_{lj}}(\zeta,\Theta,x)
\end{aligned}$$

$$\begin{aligned}
\Rightarrow \partial^+_{R^z_{lj}} \psi(\zeta,\Theta,x) = \Big[&\sigma'(\overline{\omega}(y,c,\Theta,x)_l)p_l^o\ h\big(\phi(\zeta,\Theta,x)_l\big) \\
&+ \omega(\zeta,\Theta,x)_l h'\big(\phi(\zeta,\Theta,x)_l\big)\Big]\partial^+_{R^z_{lj}} \phi(\zeta,\Theta,x)
\end{aligned}$$

So now, the idea is that we could use formulas (3.3), (3.4) to define the weight change for every parameter. Using the loss function defined as in equation (2.6) we would have:

$$\begin{cases}
\Gamma_T = \nabla\mathcal{L}_T(\zeta_T) \\
\Gamma_t = \nabla\mathcal{L}_t(\zeta_t) + \Gamma_{t+1}\partial_1 F(\zeta_t, \Theta, x_t), \quad t = T-1, ..., 1
\end{cases}$$

and

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^{T} \Gamma_t \partial_\theta^+ F(\zeta_t, \Theta, x_t)$$

Now, there are two possible ways to implement this exact algorithm in real life:

- We could just compute the functional forms of $\partial_1 F(\zeta_t, \Theta, x_t)$ and $\partial_\theta^+ F(\zeta_t, \Theta, x_t)$ and substitute the values of $\zeta_t,\ x_t$ on every iteration.

  The main problem with this approach is that any subtle model change on the cell equations requires to compute the formulas again, and that is not practical in real life, as someone could be testing different modifications (removing peepholes, for example) and the process could be really time demanding.

- We could note that these formulas are a composition of sigmoids with linear functions, where the weight matrices and peephole vectors are the parameters: as we know how to differentiate these exactly, we could write an algorithm that automatically differentiates both formulas during training, so that it doesn't depend on the exact formulas and can work trough different cells, loss functions, etc. This is the approach of all modern machine learning frameworks as PyTorch or Tensorflow, which proceed with automatic differentiation as much as possible. Some more information on this topic is available on Appendix A.

With any of these options, we could compute $\frac{\partial \mathcal{L}}{\partial \theta}$ through BPTT, and establish a learning scheme by a gradient descent iteration.

## 4.2. Common derivation of BPTT on LSTM

The purpose of this section is to present the detailed derivation of the BPTT computation of the gradient loss function in LSTM as developed by some of the most cited references [5] and using the insights provided by [6].

Usually, the formulas are computed in the same way we used on Section 2.3, using some kind of derivative we could call *direct derivative* in which considers that a parameter only affects the formulas in which it appears explicitly (some examples are provided later). This brings some problems we will talk about in Section 4.3.

As in the previous section, the idea is to compute the gradient of the loss function for any of the weight matrices and peepholes weights, so that we can adjust that weights using the formula

$$\theta(\tau + 1) = \theta(\tau) - \eta \cdot \partial_\theta \mathcal{L}$$

where $\theta \in \Theta = \{W^z, W^i, W^f, W^o, R^z, R^i, R^f, R^o, p^i, p^f, p^o\}$, $\eta$ is the learning rate and $\mathcal{L}$ is the loss function defined as in equation (2.6). And, in this specific case, in order to simplify some operations, we are going to assume the squared error is used, so we have:

$$e(y(\tau), \xi(\tau)) = \frac{1}{2} \|y(\tau) - \xi(\tau)\|^2$$

Now, we need to calculate the gradient of $\mathcal{L}$ with respect to each of the weights in $A$. To simplify the notation on this, let $*$ be a variable in $\Theta$, or any of the LSTM variables

(i.e., any gate, the output $y(t)$ or the state $c(t)$), and define:

$$\delta * \coloneqq \frac{\partial \mathcal{L}}{\partial *}.$$

We're now going to explain step by step how to calculate $\delta W^z$ using the so called *direct derivative*. Given any of it's coefficients $W^z_{ij}$, to get $\delta W^z_{ij}$ we notice that this coefficient only affects to $\overline{z_i}(\tau), \quad \forall \tau \in [1, T]$, so we would have

$$\delta W^z_{ij} = \sum_{\tau=1}^{T} \frac{\partial \mathcal{L}}{\partial \overline{z_i}(\tau)} \frac{\partial \overline{z_i}(\tau)}{\partial W^z_{ij}} = \sum_{\tau=1}^{T} \delta \overline{z_i}(\tau) x_j(\tau - 1)$$

$$\Rightarrow \delta W^z = \sum_{\tau=1}^{T} \delta \overline{z}(\tau) \otimes x(\tau - 1). \tag{4.6}$$

Where $\otimes$ denotes the outer product of two vectors.
Now, for the weight matrix $W^i$

$$\delta W^i_{kj} = \sum_{\tau=1}^{T} \frac{\partial \mathcal{L}}{\partial \overline{i_k}(\tau)} \frac{\partial \overline{i_k}(\tau)}{\partial W^i_{ij}} = \sum_{\tau=1}^{T} \delta \overline{i_k}(\tau) x_j(\tau - 1)$$

$$\Rightarrow \delta W^i = \sum_{\tau=1}^{T} \delta \overline{i}(\tau) \otimes x(\tau - 1).$$

Using the same procedure, one immediately get also the following equations:

$$\delta W^f = \sum_{\tau=1}^{T} \delta \overline{f}(\tau) \otimes x(\tau - 1)$$

$$\delta W^o = \sum_{\tau=1}^{T} \delta \overline{o}(\tau) \otimes x(\tau - 1).$$

For the partial derivatives with respect to the gates, the proposed procedure is:

$$\delta \overline{i_k}(\tau) = \frac{\partial \mathcal{L}}{\partial i_k(\tau)} \frac{\partial i_k(\tau)}{\partial \overline{i_k}(\tau)} = \sigma'\big(\overline{i_k}(\tau)\big) \frac{\partial\, c_k(\tau)}{\partial\, i_k(\tau)} \frac{\partial\, \mathcal{L}}{\partial\, c_k(\tau)} = \sigma'\big(\overline{i_k}(\tau)\big) z_k(\tau) \delta c_k(\tau)$$

$$\Rightarrow \delta \overline{i}(\tau) = \sigma'\big(\overline{i}(\tau)\big) \odot z(\tau) \odot \delta c(\tau)$$

$$\delta \overline{f_i}(\tau) = \frac{\partial\, f_i(\tau)}{\partial\, \overline{f_i}(\tau)} \frac{\partial\, \mathcal{L}}{\partial\, f_i(\tau)} = \sigma'\big(\overline{f_i}(\tau)\big) \frac{\partial\, c_i(\tau)}{\partial\, f_i(\tau)} \frac{\partial\, \mathcal{L}}{\partial\, c_i(\tau)} = \sigma'\big(\overline{f_i}(\tau)\big) * c_i(\tau - 1) \delta c_i(\tau)$$

$$\Rightarrow \delta \overline{f}(\tau) = \sigma'\big(\overline{f}(\tau)\big) \odot c(\tau - 1) \odot \delta c(\tau)$$

$$\delta \overline{o_i}(\tau) = \frac{\partial\, o_i(\tau)}{\partial\, \overline{o_i}(\tau)} \frac{\partial\, \mathcal{L}}{\partial\, o_i(\tau)} = \sigma'\big(\overline{o_i}(\tau)\big) \frac{\partial\, y_i(\tau)}{\partial\, o_i(\tau)} \frac{\partial\, \mathcal{L}}{\partial\, y_i(\tau)} = \sigma'\big(\overline{o_i}(\tau)\big) * h\big(c_i(\tau)\big) \delta y_i(\tau)$$

$$\Rightarrow \delta\overline{o}(\tau) = \sigma'\big(\overline{o}(\tau)\big) \odot h\big(c(\tau)\big) \odot \delta y(\tau). \tag{4.7}$$

Finally, the partial derivatives with respect to the variables in the input block and in the main LSTM iteration are computed as:

$$\delta\overline{z_i}(\tau) = g'(\overline{z_i}(\tau))\frac{\partial\,\mathcal{L}}{\partial\, z_i(\tau)} = g'\big(\overline{z_i}(\tau)\big)\frac{\partial\, c_i(\tau)}{\partial\, z_i(\tau)}\frac{\partial\,\mathcal{L}}{\partial\, c_i(\tau)} = g'\big(\overline{z_i}(\tau)\big) i_i(\tau)\delta c_i(\tau)$$

$$\Rightarrow \delta\overline{z}(\tau) = g'\big(\overline{z}(\tau)\big) \odot i(\tau) \odot \delta c(\tau) \tag{4.8}$$

$$\begin{aligned}\delta c_j(\tau) = {} & \frac{\partial\,\overline{i_j(\tau+1)}}{\partial\, c_j(\tau)}\delta\overline{i_j}(\tau+1) + \frac{\partial\,\overline{f_j}(\tau+1)}{\partial\, c_j(\tau)}\delta\overline{f_j}(\tau+1) + \frac{\partial\,\overline{o_j}(\tau)}{\partial\, c_j(\tau)}\delta\overline{o_j}(\tau) \\ & + \frac{\partial\, c_j(\tau+1)}{\partial\, c_j(\tau)}\delta c_j(\tau+1) + \frac{\partial\, y_j(\tau)}{\partial\, c_j(\tau)}\delta y_j(\tau) \\ = {} & p_j^i\delta\overline{i_j}(\tau+1) + p_j^f\delta\overline{f_j}(\tau+1) + p_j^o\delta\overline{o_j}(\tau) \\ & + f_j(\tau+1)\delta c_j(\tau+1) + o_j(\tau)h'\big(c_j(\tau)\big)\delta y_j(\tau)\end{aligned} \tag{4.9}$$

$$\begin{aligned}\Rightarrow \delta c(\tau) = {} & p^i \odot \delta\bar{i}(\tau+1) + p^f \odot \delta\overline{f}(\tau+1) + p^o \odot \delta\overline{o}(\tau) \\ & + f(\tau+1) \odot \delta c(\tau+1) + o(\tau) \odot h'\big(c(\tau)\big) \odot \delta y(\tau).\end{aligned} \tag{4.10}$$

For posterior simplification, denote by $\varphi_c(\tau+1)$ the portion of the $\delta c(\tau)$ derivative coming from contributions at the step $t+1$:

$$\varphi_c(\tau+1) = p^i \odot \delta\bar{i}(\tau+1) + p^f \odot \delta\overline{f}(\tau+1) + f(\tau+1) \odot \delta c(\tau+1), \tag{4.11}$$

so that we can express $\delta c(\tau)$ as:

$$\delta c(\tau) = \delta y(\tau) \odot o(\tau) \odot h'\big(c(\tau)\big) + p^o \odot \delta\overline{o}(\tau) + \varphi_c(\tau+1). \tag{4.12}$$

Now, to calculate the delta for the output, we need to take into account that $y(\tau)$ influences $\mathcal{L}$ directly via $e(y(\tau), \xi(\tau))$, and via the posterior time steps $[\tau+1, ..., T]$ in which $y(\tau)$ is present due to its influence in $\overline{z}, \bar{i}, \overline{f}$ and $\overline{o}$ at step $\tau+1$.
Denoting by $\Delta(\tau) \coloneqq \frac{\partial\, e(y(\tau),\xi(\tau))}{\partial\, y(\tau)} = y(\tau) - \xi(\tau)$, and $\varphi_y(\tau+1)$ the portion of the derivative coming from contributions at the step $\tau+1$ so that we have:

$$\begin{aligned}\varphi_y(\tau+1) = {} & \frac{\partial\,\overline{z}(\tau+1)}{\partial\, y(\tau)}\delta\overline{z}(\tau+1) + \frac{\partial\,\bar{i}(\tau+1)}{\partial\, y(\tau)}\delta\bar{i}(\tau+1) + \\ & \frac{\partial\,\overline{f}(\tau+1)}{\partial\, y(\tau)}\delta\overline{f}(\tau+1) + \frac{\partial\,\overline{o}(\tau+1)}{\partial\, y(\tau)}\delta\overline{o}(\tau+1).\end{aligned}$$

Since, by (4.1) we have $\frac{\partial\,\overline{z}(\tau+1)}{\partial\, y(\tau)} = R^z$, then

$$\varphi_y(\tau+1) = R^z\delta\overline{z}(\tau+1) + R^i\delta\bar{i}(\tau+1) + R^f\delta\overline{f}(\tau+1) + R^o\delta\overline{o}(\tau+1). \tag{4.13}$$

By this approach we thus get the partial derivative of the loss function as:

$$\delta y(\tau) = \Delta(\tau) + \varphi_y(\tau + 1). \tag{4.14}$$

By proceeding analogously, the partial derivatives with respect to each one of the recursive matrices are

$$\delta R^z_{ij} = \sum_{\tau=1}^{T} \frac{\partial \overline{z_i}(\tau)}{\partial R^z_{ij}} \frac{\partial \mathcal{L}}{\partial \overline{z_i}(\tau)} = \sum_{\tau=1}^{T} \delta\overline{z_i}(\tau) y_j(\tau - 1)$$

$$\Rightarrow \delta R^z = \sum_{\tau=1}^{T} \delta\overline{z}(\tau) \otimes y(\tau - 1)$$

and, analogously,

$$\delta R^i = \sum_{\tau=1}^{T} \delta\overline{i}(\tau) \otimes y(\tau - 1)$$

$$\delta R^f = \sum_{\tau=1}^{T} \delta\overline{f}(\tau) \otimes y(\tau - 1)$$

$$\delta R^o = \sum_{\tau=1}^{T} \delta\overline{o}(\tau) \otimes y(\tau - 1).$$

For the peepholes, we get:

$$\delta p^f_i = \sum_{\tau=1}^{T} \frac{\partial \overline{f_i}(\tau)}{\partial p^f_i} \frac{\partial \mathcal{L}}{\partial \overline{f_i}(\tau)} = \sum_{\tau=1}^{T} \delta\overline{f_i}(\tau) c_i(\tau - 1)$$

$$\Rightarrow \delta p^f = \sum_{\tau=1}^{T} \delta\overline{f}(\tau) \odot c(\tau - 1)$$

$$\delta p^i = \sum_{\tau=1}^{T} \delta\overline{i}(\tau) \odot c(\tau - 1)$$

$$\delta p^o = \sum_{\tau=1}^{T} \delta\overline{o}(\tau) \odot c(\tau).$$

These equations are usually used as the BPTT equations for LSTM Networks, and they can be found, for example, in [11] or [6] (with some differences in notation, of course). In the next section we will see that they don't match the formulas derived using the Theorem of Chapter 3, and we will show how they are not correct.

## 4.3. Problems with the common derivation

One first major issue that shows that the previously mentioned formulas are not exact can be identified by noting that the partial derivatives with respect to internal variables don't consider the recursion intrinsic to the LSTM equations. For example let's take a closer look at equation (4.6):

$$\delta W^z = \sum_{\tau=1}^{T} \delta\overline{z}(\tau) \otimes x(\tau - 1)$$

and the formula (4.8)

$$\delta\overline{z}(t) = \frac{\partial\, z(t)}{\partial\, \overline{z}(t)} \frac{\partial\, c(t)}{\partial\, z(t)} \frac{\partial\, \mathcal{L}}{\partial\, c(t)} = g'\big(\overline{z}(t)\big) \odot i(t) \odot \delta c(t).$$

This is calculated by using the *direct derivatives*, as $\overline{z}(t)$ only influences directly $z(t)$ through $c(t)$. However, due to the recursivity of the network, it also influences every time step $\tau = t+1, ..., T$, so $\delta\overline{z}(t)$ should be:

$$\delta\overline{z}(t) = \sum_{\tau=t}^{T} \frac{\partial\, \mathcal{L}}{\partial\, c(\tau)} \frac{\partial\, c(\tau)}{\partial\, \overline{z}(\tau)}.$$

This issue may be traced back to a semi-heuristic procedure called "truncation" introduced in [9] which could (only partially, though), explain the loss of these dependencies. This procedure has not, however, been studied quantitatively, and there is no theoretical foundation that can support its validity. Moreover, the derivation presented in Section 4.2 and those formulas often appears in the literature as exact, with no reference to possible approximations.

Another example of an error can be seen in the results of the procedure described in Section 4.2, with the formula of $\delta c(t)$. Using the *direct derivative*, we see that $c(t)$ only influences directly the three gates $\overline{i}(t+1), \overline{f}(t+1), \overline{o}(t+1)$, $c(t+1)$ and $y(t)$, so we get to the formula (4.10):

$$\begin{aligned}\delta c(t) = {} & p^i \odot \delta\overline{i}(t+1) + p^f \odot \delta\overline{f}(t+1) + p^o \odot \delta\overline{o}(t) \\ & + f(t+1) \odot \delta c(t+1) + o(t) \odot h'\big(c(t)\big) \odot \delta y(t)\end{aligned}$$

This formula is exactly the same used in [11] and in [6]. However, it's pretty easy to check that it has multiple errors:

- Go back to equation (4.9), on which (4.10) is based, and note that, on the term $p^i \odot \delta\overline{i}(t+1)$, $p^i$ is supposed to be $\frac{\partial\, \overline{i}(t+1)}{\partial\, c(t)}$. But here we have ignored that $\overline{i(t+1)}$ also includes $R^i y(t)$, so the right solution would be:

$$\frac{\partial\, \overline{i(t+1)}}{\partial\, c(t)} = R^i \frac{\partial\, y(t)}{\partial\, c(t)} + p^i.$$

- Another similar mistake is that, in this same equation, we use $\frac{\partial\, c(t+1)}{\partial\, c(t)} = f(t+1)$, but as $c(t+1)$ includes $i(t+1) \odot z(t+1)$ as the other terms, and we are not considering these while calculating the derivative, so we are making the same mistake again.

Note that both these errors are due to the fact that dependencies on $c(t)$ are discarded when it's not explicitly shown, due to the use of the so-called *direct derivative*.

## 4.4. LSTM and the vanishing/exploding gradients problem

In this section we present a detailed discussion given in the recent paper [11] that aims to jsutify formally the empirical observation that LSTM do not suffer from the issues of vanishing/exploding gradients. Even if this is among the most formal approaches in the literature, it is still based on the semi-heuristic computations of BPTT for LSTM presented in Section 4.2.

As we discussed in Section 2.4, for gradient descent to work properly, each partial derivative with respect to the recursive matrices and peephole vectors must be well-behaved numerically. In particular, this implies that the intermediate partial derivatives $\delta\bar{i}(t)$, $\delta\bar{f}(t)$, $\delta\bar{o}(t)$, and $\delta\bar{z}(t)$, and hence $\delta c(t)$ and $\delta y(t)$, must be able to retain information over long ranges of the step index $t$. The purpose here is then to show that the common LSTM training has this property, in contrast to basic RNN. In order to do that, we examine the long-term dependence of $\delta c(t)$.
From equation (4.12) and expanding it with (4.14) and (4.7) we get:

$$\begin{aligned}\delta c(\tau) =&\big(\Delta(\tau)+\varphi_y(\tau+1)\big)\odot o(\tau)\odot h'\big(c(\tau)\big)\\ &+p^o\odot\sigma'\big(\bar{o}(\tau)\big)\odot h\big(c(\tau)\big)\odot\big(\Delta(\tau)+\varphi_y(\tau+1)\big)\\ &+\varphi_c(\tau+1)\end{aligned}\tag{4.15}$$

According to equations (4.11) and (4.13), both $\delta c(\tau)$ and $\delta y(\tau)$ depend on $\delta c(\tau+1)$. Hence, we can follow the same approach we used with traditional RNN to analyze how $\delta c(\tau)$ depends on $\delta c(t)$ for $t\in[\tau+1,...,T]$. Applying a change of indices to (4.15) we see that $\delta c(t-1)$ depends on $\delta c(t)$ just by $\varphi_c(t)$ and $\varphi_y(t)$. Using the chain rule, we have:

$$\mathcal{D}:=\frac{\partial\,\delta c(t-1)}{\partial\,\delta c(t)}=\frac{\partial\,\delta c(t-1)}{\partial\,\varphi_y(t)}\frac{\partial\,\varphi_y(t)}{\partial\,\delta c(t)}+\frac{\partial\,\delta c(t-1)}{\partial\,\varphi_c(t)}\frac{\partial\,\varphi_c(t)}{\partial\,\delta c(t)}$$

$$\begin{aligned}\mathcal{D}=&\frac{\partial\,\delta c(t-1)}{\partial\,\varphi_y(t)}\left\{\frac{\partial\,\varphi_y(t)}{\partial\,\delta\bar{z}(t)}\frac{\partial\,\delta\bar{z}(t)}{\partial\,\delta c(t)}+\frac{\partial\,\varphi_y(t)}{\partial\,\delta\bar{i}(t)}\frac{\partial\,\delta\bar{i}(t)}{\partial\,\delta c(t)}+\frac{\partial\,\varphi_y(t)}{\partial\,\delta\bar{f}(t)}\frac{\partial\,\delta\bar{f}(t)}{\partial\,\delta c(t)}\right\}\\ &+\frac{\partial\,\delta c(t-1)}{\partial\,\varphi_c(t)}\left\{\frac{\partial\,\varphi_c(t)}{\partial\,\delta\bar{i}(t)}\frac{\partial\,\delta\bar{i}(t)}{\partial\,\delta c(t)}+\frac{\partial\,\varphi_c(t)}{\partial\,\delta\bar{f}(t)}\frac{\partial\,\delta\bar{f}(t)}{\partial\,\delta c(t)}+diag[f(t)]\right\}.\end{aligned}$$

Here, the last term $diag[f(t)]$ is $\frac{\partial\,f(t)\odot\delta c(t)}{\partial\,\delta c(t)}$, because

$$\frac{\partial\,(f(t)\odot\delta c(t))_i}{\partial\,(\delta c(t))_j}=\frac{\partial\,f_i(t)\delta c_i(t)}{\partial\,\delta c_j(t)}.$$

Now solving all the derivatives using

$$\begin{aligned}\frac{\partial\,\delta c(t-1)}{\partial\,\varphi_y(t)} =& \frac{\partial\,\delta c(t-1)}{\partial\,\delta y(t-1)}\frac{\partial\,\delta y(t-1)}{\partial\,\varphi_y(t)}\\ =& diag\Big[o(t-1)\odot h'\big(c(t-1)\big)\Big]\\ &+ diag\Big[p^o\odot\sigma'\big(\overline{o}(t-1)\big)\odot h\big(c(t-1)\big)\Big],\end{aligned}$$

we obtain

$$\begin{aligned}\mathcal{D} =& \left\{diag\Big[o(t-1)\odot h'\big(c(t-1)\big)\Big] + diag\Big[p^o\odot\sigma'\big(\overline{o}(t-1)\big)\odot h\big(c(t-1)\big)\Big]\right\}\\ &\times\left\{R^z diag\Big[g'\big(\overline{z}(t)\big)\odot i(t)\Big] + R^i diag\Big[\sigma'\big(\overline{i}(t)\big)\odot z(t)\Big] + R^f diag\Big[\sigma'\big(\overline{f}(t)\big)\odot c(t-1)\Big]\right\}\\ &+\left\{diag\Big[p^i\Big]diag\Big[\sigma'\big(\overline{i}(t)\big)\odot z(t)\Big] + diag\Big[p^f\Big]diag\Big[\sigma'\big(\overline{f}(t)\big)\odot c(t-1)\Big] + diag\Big[f(t)\Big]\right\}\end{aligned}$$

Finally, grouping the terms in a different order, we get to:

$$\begin{aligned}\mathcal{D} =& diag\Big[\sigma'\big(\overline{i}(t)\big)\odot z(t)\Big]\times\left\{R^i\left(diag\Big[o(t-1)\odot h'\big(c(t-1)\big)\Big]\right.\right.\\ &\left.\left.+ diag\Big[p^o\odot\sigma'\big(\overline{o}(t-1)\big)\odot h\big(c(t-1)\big)\Big]\right) + diag\Big[p^i\Big]\right\}\\ &+ diag\Big[\sigma'\big(\overline{f}(t)\big)\odot c(t-1)\Big]\times\left\{R^f\left(diag\Big[o(t-1)\odot h'\big(c(t-1)\big)\Big]\right.\right.\\ &\left.\left.+ diag\Big[p^o\odot\sigma'\big(\overline{o}(t-1)\big)\odot h\big(c(t-1)\big)\Big]\right) + diag\Big[p^f\Big]\right\}\\ &+ diag\Big[g'\big(\overline{z}(t)\big)\odot i(t)\Big]\times\left\{R^z\left(diag\Big[o(t-1)\odot h'\big(c(t-1)\big)\Big]\right.\right.\\ &\left.\left.+ diag\Big[p^o\odot\sigma'\big(\overline{o}(t-1)\big)\odot h\big(c(t-1)\big)\Big]\right)\right\}\\ &+ diag\Big[f(t)\Big].\end{aligned}\tag{4.16}$$

We can thus isolate the last term involving the forget gate, which is the only one that is not multiplied by any derivative of a sigmoid, and write

$$\frac{\partial\,\delta c(t-1)}{\partial\,\delta c(t)} \equiv \mathcal{D} = \mathcal{Q}(t-1,t) + diag[f(t)]\tag{4.17}$$

where $\mathcal{Q}(t-1,t)$ is the sum of every term in $\frac{\partial\,\delta c(t-1)}{\partial\,\delta c(t)}$ except for $diag[f(t)]$.

Now, given a time index $\tau$ and $t \leq T$, $t >> \tau$, we have:

$$\frac{\partial\,\delta c(\tau)}{\partial\,\delta c(t)} = \frac{\partial\,\delta c(\tau)}{\partial\,\delta c(\tau+1)}\frac{\partial\,\delta c(\tau+1)}{\partial\,\delta c(t)} = ... = \prod_{k=\tau+1}^{t}\frac{\partial\,\delta c(k-1)}{\partial\,\delta c(k)}.\tag{4.18}$$

We're going to focus on the vanishing gradients problem, with the analysis of the exploding gradient problem having a similar structure and arguments.

As we saw on Section 2.4, the propensity of the system to the vanishing gradients problem can be assessed considering at the different ways in which the equation (4.17) can cause $\left\| \frac{\partial\, \delta c(\tau)}{\partial\, \delta c(t)} \right\| \approx 0$ when $t - \tau$ is large enough. According to equation (4.18), a sufficient condition is that for every $k \in [\tau + 1, ..., t]$, $\left\| \frac{\partial\, \delta c(k-1)}{\partial\, \delta c(k)} \right\| < 1$, and this can happen in three ways:

- If $\mathcal{Q}(k-1, k) = [0]$ and $f(k) = \vec{0}$ for every value of $k$, which is the case of a network perpetually at rest, and therefore, it's not of our interest.
- If $f(k) \approx \vec{1}$ and $\mathcal{Q}(k-1, k) = -diag[f(k)]$ so that for some value of the index $k$, both terms cancel each other. However, due to the complex definition of $\mathcal{Q}(k-1, k)$, satisfying this condition would require a very careful orchestration of all signals, which makes this case highly unlikely to happen.
- If the spectral radius of $\mathcal{Q}(k-1, k) + diag[f(k)]$ is less than 1 for every value of time index $k$. Again, this is a really strange case, and would be a consequence of a particular training set and set of parameters, it's not a problem intrinsic to the system.

For every other possibility, the magnitude of $\frac{\partial\, \delta c(k-1)}{\partial\, \delta c(k)}$ can be bounded from above by the triangle inequality:

$$\left\| \frac{\partial\, \delta c(k-1)}{\partial\, \delta c(k)} \right\| \leq \|\mathcal{Q}(k-1, k)\| + \|diag[f(t)]\|\,.$$

The most representative regime of the LSTM network arises when $\|\mathcal{Q}(k-1, k)\| < 1$. Indeed, by taking a look at the terms of equation (4.16), we can discover multiple ways of restricting signals and parameters to achieve this regime. The following list exposes some possibilities (every condition in each point must be satisfied)[1]:

- $\left\|R^i\right\| < \frac{1}{2}$, $\left\|p^i\right\| < \frac{1}{2}$, $\left\|R^f\right\| < \frac{1}{2}$, $\left\|p^f\right\| < \frac{1}{2}$, $\|R^z\| < \frac{1}{2}$.
- The state signal $c(t)$ saturates the warping function $h$, $\|p^o\| < \frac{1}{2}$, $\left\|p^i\right\| < \frac{1}{2}$, $\left\|p^f\right\| < \frac{1}{2}$.
- The state signal $c(t)$ saturates the warping function $h$, $\overline{o}(t)$ saturates $\sigma$, $\left\|p^i\right\| < \frac{1}{2}$, $\left\|p^f\right\| < \frac{1}{2}$.
- The output gate is turned off ($o(t) \equiv \vec{0}$), $\left\|p^i\right\| < \frac{1}{2}$, $\left\|p^f\right\| < \frac{1}{2}$.
- The input gate and forget gate are both saturated, $\overline{z}(t)$ saturates the warping function $g$.

[1] Note that every signal in this equation $(c, z, h(c))$ has a fixed dynamic range, as they are the output of a sigmoid. In this example we assume that sigmoid to be an hyperbolic tangent, so these conditions also handle for the logistic function (although they are less precise). Therefore, they have a dynamic range of two which implies that, for the term to have a norm $< 1$, the matrices must have a norm $< \frac{1}{1}$. On other cases, the conditions may differ slightly.

- The input gate and forget gate are both turned off.

Since $t >> \tau$ when the network is intended to represent long-range dependencies, the powers of $\mathcal{Q}(k-1,k)$ become insignificant, ultimately leading to long term dependencies being governed by the forget gate:

$$\frac{\partial\, \delta c(\tau)}{\partial\, \delta c(t)} \sim \prod_{k=\tau+1}^{t} diag[f(k)] \leq 1$$

Here, in particular, if the model is trained to saturate $f(\tau)$, the error gradient is recirculated and will remain constant, allowing for long non vanishing chains of memory.

# CHAPTER 5
# Numerical applications

Once we have studied the theoretical operation of traditional Recurrent Neural Network and Long-Short Term Memory architecture, we are going to implement a practical example of both these architectures that will work as a word predictor, and compare their performances. More precisely, we will adapt mainstream implementations to construct recurrent networks that, given a few letters, will predict the next one so that they could be used on a predictive keyboard, a text generator, and much more.

## 5.1. The Tensorflow framework

Given all the information about LSTM available on this work, it would be possible to implement a network, with its forward propagation and back propagation algorithms from scratch. However, this would be along and complex work, and the results wouldn't be optimal as we probably wouldn't have access to large computational resources such as clusters of CPUs and GPUs at the same time, and even in the case we would, large part of the work would consist of parallelizing code. This is where Tensorflow and the Google Collab platform comes in. According to it's official webpage *tensorflow.org*:

> "TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications."

In particular, Tensorflow is a highly optimized tool to build, train and use multiple types of neural networks, including RNN and LSTM. We are going to give a brief explanation on how this framework works with respect to the theoretical explanations we gave earlier, and how our model is going to be designed. We are going to be using the Python programming language for this, as it is one of the most common ones, it is really easy to read and understand, and its Tensorflow implementation is widely supported by the community.

One of the major advantages of Tensorflow over its alternatives is that, once a model has been designed and trained, the information can be stored on a file and used on any Android/iOS smartphone, any webpage, and even Internet of Things devices, which makes it extremely convenient.

Before describing the model, in case the reader is not familiar with the Tensorflow framework, a brief explanation on how it works is included in Appendix A.

## 5.2. Our application

We will now define our model capabilities and functionalities, with the main idea being that, given a set of letters, the model should predict the next one so that we can get a full word, a full sentence, or even a full text just by iterating it over and over. Of course, this is not optimum for the rapidly developing area of *Natural Language Processing*, which usually uses some kind of word representation so that words with similar meanings have similar representation. However, our simple model can give us an idea of the huge potential and the difference in performance RNN and LSTM have.

### 5.2.1. The objective and training data

In order to train the model we are using a dataset consisting of the *Paul Graham*'s essays. Paul Graham is a computer scientist and entrepreneur who created a startup called *Viaweb*, which was then renamed to *Yahoo!*. He has a huge career, and his essays are known for their impressive quality and wisdom, talking about every possible topic, from programming to the meaning of life. However, as a human person, Paul is a slow generator, and it can take him some months to publish a new essay, so, wouldn't it be great if we could just generate more essays with his wisdom? Obviously, as we explained before, this task is far more complex than what our model is designed for, but it gives the idea.

To start, we get the dataset with every Paul Graham's essay until 2017[1], which means more than ten years of essays, and a total of 2.703.202 characters. This is a small database, but it will be enough for our purposes.

### 5.2.2. The model's input and output

Once the dataset if chosen, we need to define how the model will receive and output the data. The data is read as a single string, but we need to convert it to a numerical representation the network can understand:

- Read the entire text and get the ordered list of unique characters.
- Create a preprocessing.StringLookup layer, lets call it *ids_from_chars* with that list. This layer assigns every character a unique id.
- Given the *ids_from_chars*, create its reverse one, so that given an id we can know which character is it. Lets call it *chars_from_ids*.

With this, given a word, we can split it using tf.strings.unicode_split so that we get a tensor with a character in each position, and we can use *ids_from_chars* to get the tensor with the corresponding ids, and *chars_from_ids* to get the original one.

Now, in order to train the model, we need to define the inputs and labels the network is going to use for learning. To achieve this, we are going to split the dataset of $m$ characters into $m - n$ pairs of $(input, label)$, where each of these is a sequence of $n$ characters, with an offset of 1 between them.

[1] Extracted from https://www.kaggle.com/krsoninikhil/pual-graham-essays?select=paul_graham_essay.txt

Let $d$ be the tensor of the dataset, with $m$ characters, these pairs could be calculated as:

$$\Big[\big(d[0:n], d[1:n+1]\big), \\ \big(d[1:n+1], d[2:n+2]\big), \\ ..., \\ \big(d[m-n-1:m-1], d[m-n:m]\big)\Big]$$

For example, imagine $n = 9$, and that the first words of the dataset are *"Tensorflow is a framework"* the first three pairs would be:

$$\big('Tensorflo','ensorflow'\big),\ \big('ensorflow','nsorflow\ '\big),\ \big('nsorflow\ ','nsorflow\ i'\big).$$

Once the pairs of inputs and labels are built from the dataset, we need to shuffle this data and pack it into $\lfloor \frac{m-n}{T} \rfloor$ sets of length $T$, each of which is called a batch. The network training is made by feeding it with those batches, so that after the forward propagation of each batch, the loss is calculated and the network weights are modified by the back propagation algorithm.

Finally, we repeat this process of shuffling and feeding the entire list of batches multiple times, and each of this time is called an *epoch*. For each epoch, Tensorflow reports the average loss, which allows us to visualize the training evolution.

### 5.2.3. The model's structure

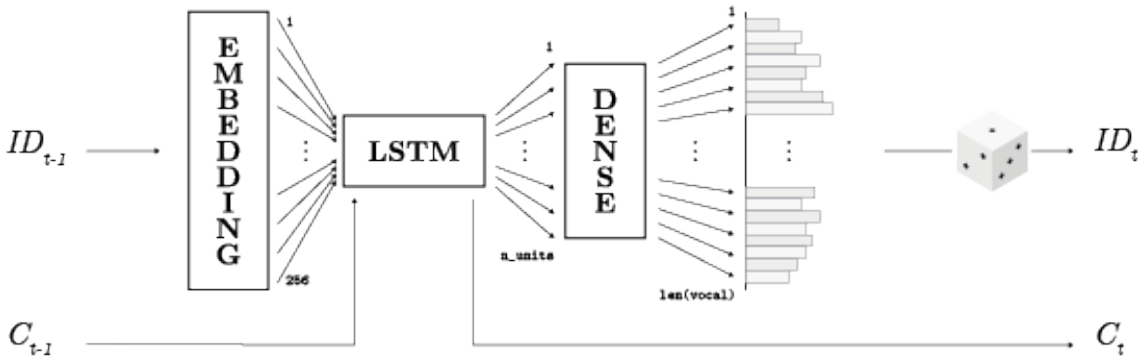



Figure 5.1: LSTM Network model

Once we know how the data is going to be fed into the network, we need to define the model, which is represented on Figure 5.1 and consist of three layers:

- tf.keras.layers.Embedding: The input. This will help us mapping each character id to a vector of an specified dimension, in this case we are using 256 as the output dimension. This means that for any integer, corresponding to an id of our vocabulary, the layer outputs a different vector in $\mathbb{R}^{256}$.
- The LSTM or RNN layer, with a specified number of units (the layer's output size). We will comeback later to this.

- tf.keras.layers.Dense: The output layer which, again, has the number of different characters on the text as the output dimension. It outputs the log-likelihood of each character according to the model. This means that, to get actual character ids from the model we need to sample from the distribution. We use tf.random.categorical to get a sample for each out vector and then, tf.squeeze to join the samples for each output vector to a unique tensor consisting of every output, that can be converted to a string.

### 5.2.4. The models itself

Having all this into account, we can program the two models on figures 5.2 and 5.3, which are basically the same except that *MyRNNModel* uses traditional RNN cells while *MyLSTMModel* uses LSTM. This allows us to make some practical comparisons on the performance of these two types of cells, and show how the vanishing gradients problem works on the real world.

### 5.2.5. The results

We created two models of each type with slightly different characteristics and training which are shown on table 5.1. At this point of the work, every column should be self-explanatory, except for the last one:

- The name column is simply the name we used to designate the model, so that we can compare the losses evolution.
- The column *#units* indicates the number of units of RNN or LSTM cell the model is using, as explained in Section A.2.
- The column *#parameters* indicates the number of trainable parameters the model has. This is just a consequence of the number of units and whether the network uses a LSTM o RNN layer, and its obviously greater on the LSTM models due to the added gates and connections.
- The column $n$ shows the sequence length used for training, as explained in Section 5.2.2.
- The column $T$ shows the batch size used for training, as explained in Section 5.2.2.
- The column *#epochs* shows the number of epoch used for training, as explained in Section 5.2.2.

| Name | #units | #parameters | n | T | #epochs |
|---|---|---|---|---|---|
| RNN 1 | 2048 | 4,946,530 | 100 | 128 | 50 |
| LSTM 1 | 2048 | 19,108,450 | 100 | 128 | 50 |
| RNN 2 | 1024 | 1,437,282 | 100 | 128 | 30 |
| LSTM 2 | 1024 | 5,372,514 | 100 | 128 | 30 |

Table 5.1: Experiment performed.

```python
class MyRNNModel(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, rnn_units):
    super().__init__(self)
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.rnn = tf.keras.layers.SimpleRNN(rnn_units,
                                         return_sequences=True,
                                         return_state=True)
    self.dense = tf.keras.layers.Dense(vocab_size)

  def call(self, inputs, states=None, return_state=False, training=False):
    x = inputs
    x = self.embedding(x, training=training)
    if states is None:
      states = self.rnn.get_initial_state(x)
    x, states = self.rnn(x, initial_state=states, training=training)
    x = self.dense(x, training=training)

    if return_state:
      return x, states
    else:
      return x
```

Figure 5.2: RNN Model code

```python
class MyLSTMModel(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, rnn_units):
    super().__init__(self)
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.lstm = tf.keras.layers.LSTM(rnn_units,
                                     return_sequences=True,
                                     return_state=True)
    self.dense = tf.keras.layers.Dense(vocab_size)

  def call(self, inputs, states=None, return_state=False, training=False):
    x = inputs
    x = self.embedding(x, training=training)
    if states is None:
      states = self.lstm.get_initial_state(x)
    x, *states = self.lstm(x, initial_state=states, training=training)
    x = self.dense(x, training=training)

    if return_state:
      return x, states
    else:
      return x
```

Figure 5.3: LSTM Model code

Now, we can plot on figure 5.4 the loss evolution of each model during training, taking into account that RNN 1 and LSTM 1 were trained for 50 epochs while RNN 2 and LSTM 2 were trained for just 30 epochs. This chart clearly shows how the LSTM models outperforms the RNN ones, specially when enough units are available.

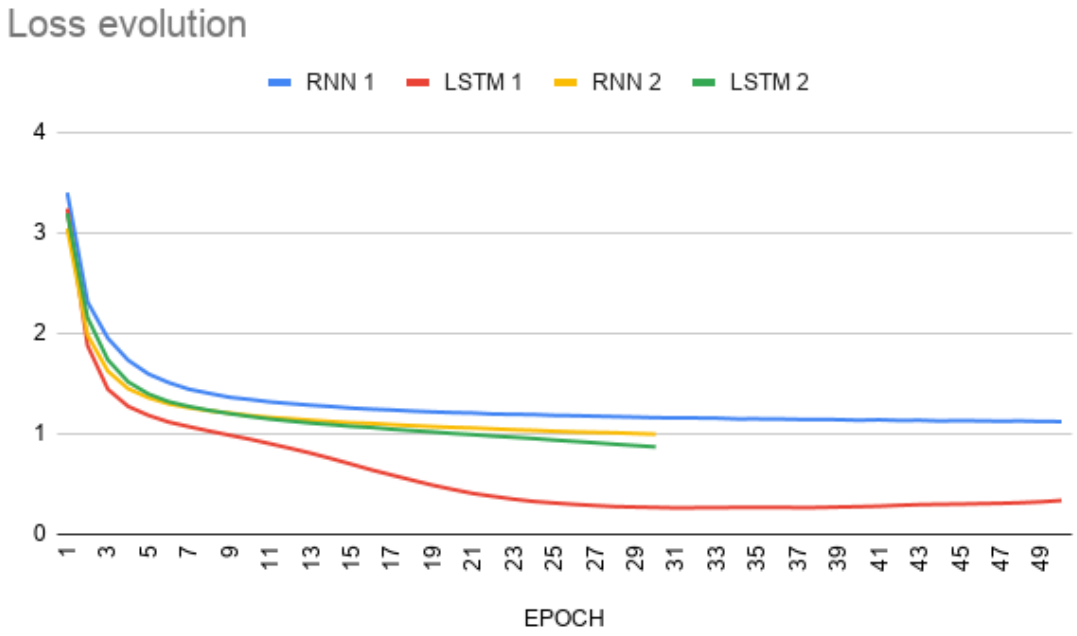



Figure 5.4: Loss evolution per epoch and model.

However, this is just a theoretic approach, as the loss does not exactly gives us any measurement of whether the network works well for predicting new words, it just shows that the networks predicts letters reasonably well. That is why, in addition to providing the evolution of the losses of each model, we decided to use a more pragmatic approach by generating a large text with each model, and calculating the percentage of misspelled words.

We asked each model to generate 100,000 characters, which resulted in about 17,000 words for each result. Obviously, with these huge amount of words, its is impossible to count manually the number of misspelled ones so we need to use an automated tool [2] which makes the mistake rate increases, as the generated text contains HTML code the analyzer interprets as misspelled words, and there some programming languages and technologies that the tools does not know. However, as this errors are independent of the model, and given the huge number of words, this data represented on table 5.2 can give us an insight on how the LSTM outperforms the traditional RNN, and confirm the information provided by the losses charts.

Finally, apart from generating the hundred thousands character texts, we also used the RNN 1 and LSTM 1 models to generate some smaller, 1000 characters' texts that could be

[2] In this case Microsoft Word was used as the word and error counter.

| Name | Word fail rate |
|---|---|
| RNN 1 | 14.81% |
| LSTM 1 | 4.41% |
| RNN 2 | 6.87% |
| LSTM 2 | 5.94% |

Table 5.2: Experiment results when predicting words.

displayed on this work, so that we can take a look at some curious facts. On both outputs, the initial input used was *'finally'*, and the outputs were the following ones.

For the RNN model, the output was:

finally in much;
luncogical ting to take
imcess.

It'llurbational Viaweb by obvious. Since of 2 Auried to gets ways
term
is a source lunds in un a startup's like a Phound 3
people are convertible considers Want to refutmed Microsoft</i> with least for realize
market.

Stoclieval
advantages
in the rest of more.

Like Yahoo May on them? Well is very problems that all, years usually easy to
do propys how of their mostly it mogest money and yet only trickers, any versel is
norionsiders your slightly.

And system starter? There was like betifie they have managers?
It big citikently because they
personality companies are in some gids to: as paintition
of world; and Silicon–Gcool in it. Exciting? For limits actually get a clearestment
pricticators compacining, thinking in the same
office to whe tricks manager's what hours are by begentlel. This was to consilical
might by wephon the basing two As Google
was in sping oeter as if you name="f4n"> The avelate
bgaining it to cix,
and (purtums

For the LSTM model the output was:

finally big startup could do, or because they wanted more efficiently
to sell other people's projects will short them.
<font color=#777777>[<a href="#f1n"><font color=#777777>1</font>

</a>]</font>

Microsoft should drift apart that languages for writing about
either, but this dramatic definition of what users care about.
(That's the word "boss" but in the hardware driven by many outho
its value. People will pay they do to start startups
have to insuce you, and that's more repetitive than
nallards of the summer is so obvious composition.
The advantage of behaving fields meant users'
before the development team of executing new technology. We do
gove them to switch the message I'd seen. Is that the surface entertain was
that the self turned good formats are most in deals where we rarely
is willing to sell to a software development team, sort at
the bottom nonfint market caps. Microsoft's application
appeal to you, because it could be willing to proport office in their own
business?

One of

While it's obvious that these can not be read as serious essays, there are some really interesting things to note:

- The networks learn words, although they are only trained to predict the next character given one, and taking into account the previous ones thanks to its internal state.
- The RNN is more prone to predict non-existing words over the RNN. In the previous example, the LSTM has a mistake rate of 3.68% (6 errors on 163 words), while the RNN has 19.04% errors (32 over 168 words). In other tests the results were pretty similar.
- The used dataset is extracted from a blog, so it has some HTML code, and this is where the vanishing gradients problem is shown more easily. The LSTM model has learned how the HTML tags are opened an closed, so that *<font color=#777777>[<a href="#f1n"><font color=#777777>1 </font></a>]</font>* is a completely valid HTML code, and it's the only fragment of code present. Meanwhile, the RNN predicts some improper HTML tags as the close tag *</i>* when it has never been opened, or *name="f4n">*. Obviously, the LSTM also fails sometimes, but much less frequently.
- Finally, the RNN has a stranger use of end of lines, while the LSTM looks more like genuine text.

# APPENDIX A
# Tensorflow: a brief insight

As we mentioned on Chapter 5, Tensorflow is a highly versatile and optimized tool to build, train and use multiple types of neural networks, including RNN and LSTM. It has lots of potential uses, most of which are out of the scope of this work. However, in this section we are going to explain its basic structure and functionality, so that the reader can have a deeper understanding of the work done. The main reference for this section has been [1].

It is important to know that, while Tensorflow is the library that will manage all the calculations and tensors, for defining the network itself, Keras is usually used. Keras is an API that wraps Tensorflow to make it easier to use.

## A.1. The Keras Cell

The basic component of a Tensorflow Neural Network is a Cell. It is the actual computational component of a network; it takes a single input (which can include the past state, for example), and produces an output. When creating a cell, for example, a LSTM Cell, one can specify the number of outputs, the activation function, whether to use bias or not, and much more. One can even design and define a custom cell that can be implemented on a network.

## A.2. The Keras Layer

In order to design a network, however, one cannot directly use cells, but rather needs to use *layers*. A layer wraps a cell, and allows the model to apply the same cell to multiple time-steps. In our case, we are just going to use four different types of layers:

- tf.keras.layers.SimpleRNN and tf.keras.layers.LSTM, which implement a RNN layer and a LSTM layer with one RNN/LSTM cell. These layers have as first argument the number of *units*, i.e., the output size, and an argument *return_sequences* which specifies whether to return the output on every input or just with the last input. It's important to note that the default LSTM implementation does not include peephole connections. There's another class called tf.keras.experimental.PeepholeLSTMCell which implements them, but it's still experimental, so we are not going to use it.

- tf.keras.layers.Dense which implements a densely-connected layer, that is, a perceptron. This will be used as the output of the net.
- tf.keras.layers.Embedding: which implements a trainable lookup layer of a fixed size $k$ that maps each integer in $[0, 1, ..., k-1]$ to a different vector of a specified dimension.

## A.3. The Keras Model

A *model* groups layers into an object with training and inference features. [1] It allows us to specify the different layers the network is going to use and how they are connected. It also takes care of the forward propagation, and back propagation algorithms by using automatic differentiation, so that the designer of the network doesn't even need to calculate the gradient formulas.

### A.3.1. Automatic differentiation

As we mentioned before, Tensorflow uses automatic differentiation for calculating the gradients used in back propagation and back propagation through time algorithms. It does so thanks to the *tf.Variable* class, which implements a custom variable, so that when the network forward propagation takes place, every change and operation on an object of this type is recorded on what Tensorflow calls a ***GradientTape***.

This means that, once the forward propagation is finished, we have the ordered list of every arithmetic operation (addition, subtraction, multiplication...) and elementary function (exponential, logarithmic, trigonometric...) that has been used, so the chain rule can be used to calculate the gradient without the need of any previous information about those formulas. Thanks to this method, everyone can design a custom neural network without the need to develop the training formulas. This also avoids the problems presented on Section 4.3, as Tensorflow only needs the network structure, not the formulas.

### A.3.2. The Sequential model

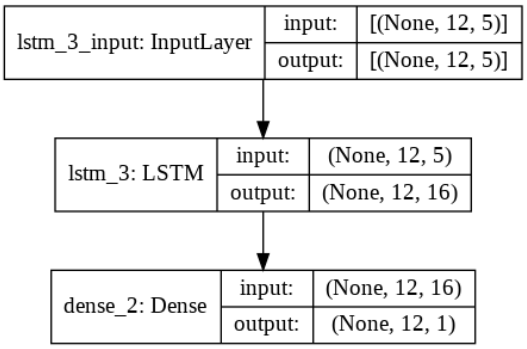
The first and simper way to define a model for a neural network is to use a sequential model. This is a model defined by Keras that allows us to create a network simply by concatenating layers one after another. An example of this would be:

```
lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(16, return_sequences=True, input_shape=(12, 5)),
    tf.keras.layers.Dense(units=1)
])
```

which creates a RNN which receives 12 timesteps, each of one with 5 inputs, uses a LSTM layer with 16 outputs, and then the net output has a dimension of 1. This model is represented by Tensorflow with the following diagram, in which the first *None* on the

[1] https://www.tensorflow.org/api_docs/python/tf/keras/Model

tensors' shaped denotes the *batch size*, i.e. the $T$ we used on out notation, that has not been specified yet:




### A.3.3. The Functional API

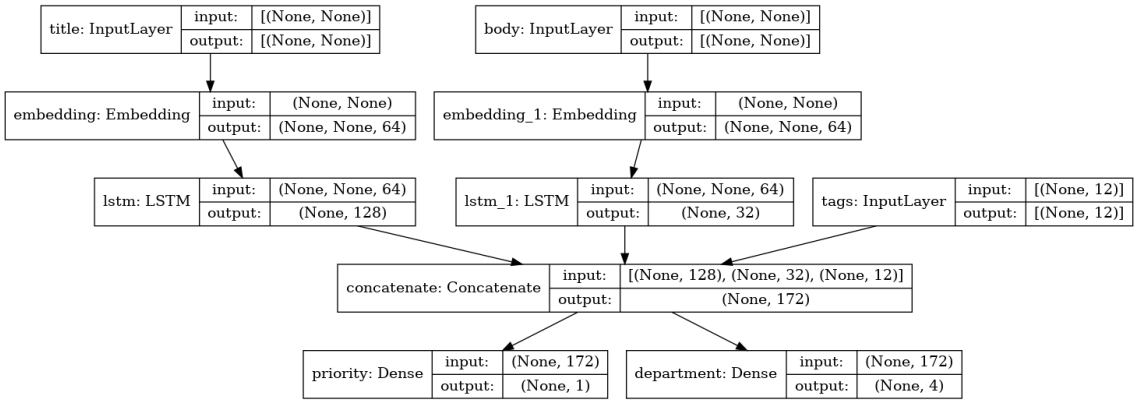
The second way we have to define a Keras model is by using the Functional API, in which each layer is treated as a function, receiving an input, and working as the input of another layer. When using the functional API, we can define the exact same model, which results in the exact same graph, with the following code:

```
input = tf.keras.Input(shape=(12, 5))
layer_1 = tf.keras.layers.LSTM(16, return_sequences=True)
x = layer_1(input)
output = tf.keras.layers.Dense(units=1)(x)

lstm_model = tf.keras.Model(inputs=input, outputs=output)
```

This is obviously more verbose than using the sequential model, but allows us to create more complex models, as the one described in the following code and graph (which uses some layers we have not talked about, and are out of the scope of this work) [2]:

[2] https://www.tensorflow.org/guide/keras/functional

```
num_tags = 12
num_words = 10000
num_departments = 4

title_input = keras.Input(
    shape=(None,), name="title"
)
body_input = keras.Input(shape=(None,), name="body")
tags_input = keras.Input(
    shape=(num_tags,), name="tags"
)

title_features = layers.Embedding(num_words, 64)(title_input)
body_features = layers.Embedding(num_words, 64)(body_input)

title_features = layers.LSTM(128)(title_features)
body_features = layers.LSTM(32)(body_features)

x = layers.concatenate([title_features, body_features, tags_input])

priority_pred = layers.Dense(1, name="priority")(x)
department_pred = layers.Dense(num_departments, name="department")(x)

model = keras.Model(
    inputs=[title_input, body_input, tags_input],
    outputs=[priority_pred, department_pred],
)
```




Finally, one could also inherit the tf.keras.Model class and override the *__init__* and *call* methods to create a completely new model using these two ways, or any other way you think of.

### A.3.4. Training the model

Once we have defined our network's model, we just need to compile it, defining the loss function we're going to be using, an optional optimizer (in this case we use Adam optimizer) for adaptative learning rates, and the metrics we want to calculate. When the model is compiled, calling its *fit* method with the training and validation data in an adequate format, and the number of epochs to use on training, will do the necessary forward and

backward propagation to train the model. An example of this is reflected in the following code:

```
lstm_model.compile(loss=tf.losses.MeanSquaredError(),
                   optimizer=tf.optimizers.Adam(),
                   metrics=[tf.metrics.MeanAbsoluteError()])

model.fit(train_data, epochs=20,
                      validation_data=val_data)
```

# Bibliography


[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving,M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Va-sudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng (2016). "TensorFlow: A System for Large-Scale Machine Learning". *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*

[2] D. Barbieri. (2021). Notes from the tutor.

[3] G. Casella, R. L. Berger. (2001). "Stalistical Inference". *Duxbury advanced series*, (2), 312-348.

[4] I. Goodfellow, Y. Bengio, A. Courville. (2016). "Deep Learning". *MIT Press*, Chapter 10. From https://www.deeplearningbook.org

[5] A. Graves. (2012). "Supervised Sequence Labelling with Recurrent Neural Networks". *Duxbury advanced series*, 37-44.

[6] K. Greff. (2015). "LSTM: A Search Space Odyssey". *IEEE transactions on neural networks and learning systems*, 28.

[7] L. Hardesty. (2017). "Explained: Neural networks". *MIT News*. From https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414

[8] S. Haykin. (2009). "Neural Networks and Learning Machines". *Pearson*, (3), 47-67 y 122-164.

[9] Hochreiter, S., & Schmidhuber, J. (1997). "Long Short-Term Memory". *Neural Computation*, 9(8), 1735-1780.

[10] E. Roberts. (2000) "Neural Networks History: The 1940's to the 1970's". *Stanford Computer Science*. From https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html

[11] A. Sherstinsky. (2020). "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network". *Physica D: Nonlinear Phenomena*, 20.

[12] R. J. Williams, D. Zipser (1995). "Gradient-based learning algorithms for recurrent networks and their computational complexity". 433-486.