Instructor's notes on Recurrent Neural Networks

Michele Rossi

Department of Information Engineering, University of Padova

6/B, Via G. Gradenigo, I–35131 Padova (PD), Italy

E-mail: {rossi@dei.unipd.it}

Abstract

A few notes detailing how Recurrent Neural Networks are structured, why they are noticeable, how they are trained, via the so called *backpropagation through time* algorithm.

These notes are supplementary material to the lessons carried out by Prof. Rossi for the course Human Data Analytics, given yearly at the Department of Information Engineering (DEI), University of Padova.

I. Recurrent Neural Networks

As opposed to Feed Forward Neural Networks (FFNN), Recurrent Neural Networks (RNN) are able to model correlation in a data sequence. This is achieved by creating self-transitions inside the network, by which past information (hidden state vector) is propagated through time and used to compute future hidden states and output. A simple but instructive RNN is shown in Fig. 1. The discussion and the derivations that follow consider this diagram, showing how the network transforms an input sequence onto a corresponding output sequence, and how the network parameters (weights) can be trained through *backpropagation*.

Time is discrete $t = 1, 2, \dots$ and is not necessarily connected with the duration of the events, but rather on the sequential order by which they are inputted into the RNN. The input at time t is a vector $\boldsymbol{x}^{(t)}$ of fixed dimension $\texttt{dim}(\boldsymbol{x}^{(t)}) = K$, with $\boldsymbol{x}^{(t)} = [x_1^{(t)}\, x_2^{(t)}\, \dots\, x_K^{(t)}]^T$. At time t, an internal state $\boldsymbol{h}^{(t)}$, with size $\texttt{dim}(\boldsymbol{h}^{(t)}) = H$, is updated from the new input $\boldsymbol{x}^{(t)}$ and the previous value of $\boldsymbol{h}^{(t-1)}$, i.e., the network takes past states (inputs) into account for the computation at the current time step. An output vector $\boldsymbol{o}^{(t)}$, of size $\texttt{dim}(\boldsymbol{o}^{(t)}) = K$, is computed solely based on the internal state $\boldsymbol{h}^{(t)}$. Since, for now, we consider a *supervised learning* task, at each time t a label (or target output vector) $\boldsymbol{y}^{(t)}$, with $\texttt{dim}(\boldsymbol{y}^{(t)}) = K$, is provided and a *loss* (or error) function $L(t)$ is obtained from $\boldsymbol{x}^{(t)}$ and $\boldsymbol{o}^{(t)}$.

Specifically, the equations governing this process are:

$$\boxed{\begin{aligned} \boldsymbol{a}^{(t)} &= \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{b} && (1)\\ \boldsymbol{h}^{(t)} &= \tanh\left(\boldsymbol{a}^{(t)}\right) && (2)\\ \boldsymbol{o}^{(t)} &= \boldsymbol{V}\boldsymbol{h}^{(t)} + \boldsymbol{c} && (3)\\ \hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}\left(\boldsymbol{o}^{(t)}\right), && (4) \end{aligned}}$$

where the network parameters to be learned are $\boldsymbol{\theta} = \{\boldsymbol{U}, \boldsymbol{W}, \boldsymbol{V}, \boldsymbol{b}, \boldsymbol{c}\}$. $\boldsymbol{a}^{(t)}$, with $\texttt{dim}(\boldsymbol{a}^{(t)}) = H$, represents the vector of activations at time t, which is fed to a hyperbolic tangent nonlinear (activation) function, which is applied in an element-wise fashion, namely, $h_i^{(t)} = \tanh\left(a_i^{(t)}\right)$. The size of the weight matrices $\boldsymbol{U}, \boldsymbol{W}, \boldsymbol{V}$ and biases $\boldsymbol{b}, \boldsymbol{c}$ are: $\texttt{dim}(\boldsymbol{U}) = H \times K$, $\texttt{dim}(\boldsymbol{W}) = H \times H$, $\texttt{dim}(\boldsymbol{V}) = K \times H$, $\texttt{dim}(\boldsymbol{b}) = H$, and $\texttt{dim}(\boldsymbol{c}) = K$, respectively. To avoid any confusion, the softmax layer is also applied

element-wise as follows:

$$\hat{y}_j^{(t)} = \text{softmax}\left(o_j^{(t)}\right) = \frac{\exp\left(o_j^{(t)}\right)}{\sum_{k=1}^{K} \exp\left(o_k^{(t)}\right)}\,,\; j = 1, \ldots, K\,. \tag{5}$$

Moreover, note that the softmax operator is used as in the following we analyze RNN parameter learning for a *classification* task. For a *regression* task the softmax is no longer needed and vector $\boldsymbol{o}^{(t)}$ would be used directly, together with the target output vector $\boldsymbol{y}^{(t)}$ to compute the loss $L(t)$.

Input vectors: we deal with a classification task, where at the input we feed the network with objects from a discrete and finite set of size K, i.e., K objects. Each input $\boldsymbol{x}^{(t)}$ is thus a one-hot vector that contains a single element equal to one, whereas all the remaining elements are set to zero. The only non-zero element identifies the object that is inputted to the network at time t. In a language processing application, for instance, this construct can be used to identify the word inputted at time t.

Target vectors: the target vector $\boldsymbol{y}^{(t)}$ corresponds to a one-hot vector denoting the object (e.g., the word) that the network should learn to output at time t in response to the input sequence $\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)}$. In fact, due to the network's recurrent character, each output depends on the entire input sequence, up to the present time step.

Loss function: for a classification task, we use a cross entropy loss function, that at any time t leads to the loss $L(t)$:

$$L(t) = -\sum_{k=1}^{K} y_k^{(t)} \log\left(\hat{y}_k^{(t)}\right). \tag{6}$$

The **total loss** (or error, the two terms are here used interchangeably) is obtained accumulating $L(t)$ across all time steps, namely:

$$L = \sum_{t=1}^{T} L(t) = -\sum_{t=1}^{T}\sum_{k=1}^{K} y_k^{(t)} \log\left(\hat{y}_k^{(t)}\right), \tag{7}$$

where T is the last time step and 1 is the first.

Computation of the error gradient: parameter learning is carried out as in a FFNN, i.e., by computing the gradient of the total error L with respect to each network parameter. Once this gradient is obtained, it is utilized within a selected gradient descent algorithm to tune the network parameters towards the desired task. To efficiently compute the gradient, the RNN is unfolded as shown in Fig. 1

(diagram on the right). The unfolded diagram is an acyclic graph, just as the one that we have in any standard FFNN. This means that a standard backpropagation algorithm also applies to the unfolded RNN representation, and that this allows to efficiently compute the total error for each time t. This way of computing the gradient of an RNN is commonly known as Backpropagation Through Time (BTT). Note also that, for any time t, the input in *all* previous time steps $1, \ldots, t-1$ ideally concur in the computation of the current output $\boldsymbol{o}^{(t)}$, which means that they also influence the total error gradient. However, as t grows large it is impractical to unfold the whole RNN diagram from time 1, as this would take a huge amount of memory and computation resources to backpropagate the gradient. As a solution to this, in practice the diagram is unfolded up to a fixed (and small) number of time steps in the past. This is a compromise solution that leads to some degradation in the performance of the learned architecture, but makes learning its parameters practical.

Chain rule of calculus: we briefly review the chain rule of calculus. It is used to compute derivatives of functions constructed by composing other functions, whose derivatives are known. Backpropagation is an algorithm that iteratively uses this chain rule, making it possible to compute the gradient of complex error functions in a highly efficient fashion. To start with, let $x \in \mathbb{R}$ and let $f(\cdot)$ and $g(\cdot)$ be two functions that map a real number onto another real number. For instance, assume $y = g(x)$ and $z = f(g(x)) = f(y)$. The chain rule of calculus allows one to compute dz/dx as:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}\,. \tag{8}$$

This can be generalized to the case where $\boldsymbol{x} \in \mathbb{R}^m$ and $\boldsymbol{y} \in \mathbb{R}^n$, whereas $g(\cdot) : \mathbb{R}^m \to \mathbb{R}^n$ and $f(\cdot) : \mathbb{R}^n \to \mathbb{R}$, with $\boldsymbol{y} = g(\boldsymbol{x})$, $z = f(\boldsymbol{y})$. If $\boldsymbol{x} = [x_1\, x_2\, \ldots x_m]^T$ and $\boldsymbol{y} = [y_1\, y_2\, \ldots y_n]^T$, the gradient of z is obtained as:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}\,. \tag{9}$$

In vector notation, (9) is written as:

$$\boxed{\nabla_{\boldsymbol{x}} z = \begin{bmatrix} \dfrac{\partial z}{\partial x_1} & \dfrac{\partial z}{\partial x_2} & \cdots & \dfrac{\partial z}{\partial x_m} \end{bmatrix}^T = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z\,,} \tag{10}$$

where $\partial \boldsymbol{y}/\partial \boldsymbol{x}$ is the $n \times m$ Jacobian matrix of $g(\cdot)$ and

$$\nabla_{\boldsymbol{y}} z = \left[\frac{\partial z}{\partial y_1} \; \frac{\partial z}{\partial y_2} \; \cdots \; \frac{\partial z}{\partial y_n} \right]^T . \tag{11}$$

II. Parameter learning through backpropagation through time

Next, we detail the backpropagation algorithm for the RNN of Fig. 1.

1) The first quantity that we compute is the gradient[1] of the total error L (see equation (7)) with respect to the output vector $\boldsymbol{o}^{(t)}$ at any time $t = 1, \ldots, T$,

$$\nabla_{\boldsymbol{o}^{(t)}} L \triangleq \left[\frac{\partial L}{\partial o_1^{(t)}} \; \frac{\partial L}{\partial o_2^{(t)}} \; \cdots \; \frac{\partial L}{\partial o_K^{(t)}} \right]^T . \tag{12}$$

Applying the chain rule of calculus the i-th element of such gradient is obtained as:

$$\frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L(t)} \frac{\partial L(t)}{\partial o_i^{(t)}} . \tag{13}$$

From (7), we see that $\partial L/\partial L(t) = 1$, whereas the second term is obtained as follows:

$$\frac{\partial L(t)}{\partial o_i^{(t)}} = -\sum_{k=1}^{K} y_k^{(t)} \frac{\partial \log \hat{y}_k^{(t)}}{\partial o_i^{(t)}} = -\sum_{k=1}^{K} y_k^{(t)} \frac{1}{\hat{y}_k^{(t)}} \frac{\partial \hat{y}_k^{(t)}}{\partial o_i^{(t)}} \tag{14}$$

From (5) we obtain:

$$\frac{\partial \hat{y}_k^{(t)}}{\partial o_i^{(t)}} = \begin{cases} \hat{y}_i^{(t)}(1 - \hat{y}_i^{(t)}) & k = i \\ -\hat{y}_i^{(t)} \hat{y}_k^{(t)} & k \neq i \, . \end{cases} \tag{15}$$

Now, using (15) back into (14), we get:

$$\frac{\partial L(t)}{\partial o_i^{(t)}} = \underbrace{-y_i^{(t)}(1 - \hat{y}_i^{(t)})}_{k=i} + \sum_{k \neq i} y_k^{(t)} \hat{y}_i^{(t)} = -y_i^{(t)} + y_i^{(t)} \hat{y}_i^{(t)} + \sum_{k \neq i} y_k^{(t)} \hat{y}_i^{(t)} = \tag{16}$$

$$= -y_i^{(t)} + \hat{y}_i^{(t)} \underbrace{\sum_{k=1}^{K} y_k^{(t)}}_{1} = \hat{y}_i^{(t)} - y_i^{(t)} , \tag{17}$$

where $\sum_{k=1}^{K} y_k^{(t)} = 1$ as $\boldsymbol{y}^{(t)}$ is a one-hot vector. In vector notation, we have:

$$\boxed{\nabla_{\boldsymbol{o}^{(t)}} L = \hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)} ,} \tag{18}$$

[1] Expressed as a column vector.

that as noted above holds for all time steps $t = 1, \ldots, T$.

2) At the final time step T, the inner state $\boldsymbol{h}^{(T)}$ has $\boldsymbol{o}^{(T)}$ as its only descendant, the error gradient is defined as:

$$\nabla_{\boldsymbol{h}^{(T)}} L \triangleq \left[\frac{\partial L}{\partial h_1^{(T)}} \; \frac{\partial L}{\partial h_2^{(T)}} \; \cdots \; \frac{\partial L}{\partial h_H^{(T)}} \right]^T . \tag{19}$$

The i-th element of this gradient is:

$$\frac{\partial L}{\partial h_i^{(T)}} = \underbrace{\frac{\partial L}{\partial L(T)}}_{1} \frac{\partial L(T)}{\partial h_i^{(T)}} = \frac{\partial L(T)}{\partial h_i^{(T)}} = \tag{20}$$

$$= \sum_j \frac{\partial L(T)}{\partial o_j^{(T)}} \frac{\partial o_j^{(T)}}{\partial h_i^{(T)}} . \tag{21}$$

Now, from (10), the above equation (20) can be rexpressed in compact form as:

$$\boxed{\nabla_{\boldsymbol{h}^{(T)}} L = \left(\frac{\partial \boldsymbol{o}^{(T)}}{\partial \boldsymbol{h}^{(T)}} \right)^T \nabla_{\boldsymbol{o}^{(T)}} L = (\boldsymbol{V})^T \nabla_{\boldsymbol{o}^{(T)}} L \, ,} \tag{22}$$

where $(\cdot)^T$ is the transpose operator and the second equality follows from (3).

3) It is now time to back propagate the gradients through time, from time T to time 1 (no truncation of the unfolded graph is assumed, so we go all the way to the beginning of time). To this end, assume the current time is $t < T$. The inner state $\boldsymbol{h}^{(t)}$ has descendants $\boldsymbol{o}^{(t)}$ and $\boldsymbol{h}^{(t+1)}$. The gradient of the error L is thus derived as:

$$\frac{\partial L}{\partial h_i^{(t)}} = \sum_j \frac{\partial L}{\partial h_j^{(t+1)}} \frac{\partial h_j^{(t+1)}}{\partial h_i^{(t)}} + \sum_j \frac{\partial L}{\partial o_j^{(t)}} \frac{\partial o_j^{(t)}}{\partial h_i^{(t)}} , \tag{23}$$

that following the same line of reasoning as for (22) can be written in compact form as:

$$\boxed{\begin{aligned} \nabla_{\boldsymbol{h}^{(t)}} L &= \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} \right)^T (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \right)^T (\nabla_{\boldsymbol{o}^{(t)}} L) = \\ &= \boldsymbol{W}^T \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)} \right)^2 \right) (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \boldsymbol{V}^T (\nabla_{\boldsymbol{o}^{(t)}} L) \, , \end{aligned}} \tag{24}$$

where $\operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)} \right)^2 \right)$ is an $H \times H$ diagonal matrix with diagonal elements $1 - (h_i^{(t+1)})^2$ (we show this shortly below). From (24), we see that $\nabla_{\boldsymbol{h}^{(t)}} L$ is *recursively computed* from $\nabla_{\boldsymbol{h}^{(t+1)}} L$, using (22) as the starting point. $\nabla_{\boldsymbol{o}^{(t)}} L$ is instead a local quantity that is computed independently for each t according to (18). Before moving on, let us shed some light on the second line of

(24). From (3), it promptly follows that $\partial \boldsymbol{o}^{(t)}/\partial \boldsymbol{h}^{(t)} = \boldsymbol{V}$. The computation of the first term of the second line is slightly more involved. We recall that:

$$\boldsymbol{h}^{(t)} = [h_1^{(t)}\ h_2^{(t)}\ \dots\ h_H^{(t)}]^T \tag{25}$$

$$h_i^{(t+1)} = \tanh\left(a_i^{(t+1)}\right) \tag{26}$$

$$a_i^{(t+1)} = (\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t)} + \boldsymbol{U}\boldsymbol{x}^{(t+1)})_i \tag{27}$$

Applying the derivative of the hyperbolic tangent, from (26), we get

$$\frac{\partial h_i^{(t+1)}}{\partial a_k^{(t+1)}} = \begin{cases} 1 - (h_i^{(t+1)})^2 & i = k \\ 0 & i \neq k\,, \end{cases} \tag{28}$$

which is the Jacobian of the hyperbolic tangent associated with hidden unit i at time $t+1$. We are now ready to compute the Jacobian:

$$\frac{\partial h_i^{(t+1)}}{\partial h_j^{(t)}} = \sum_k \frac{\partial h_i^{(t+1)}}{\partial a_k^{(t+1)}} \frac{\partial a_k^{(t+1)}}{\partial h_j^{(t)}} = (1 - (h_i^{(t+1)})^2) \frac{\partial a_i^{(t+1)}}{\partial h_j^{(t)}} = (1 - (h_i^{(t+1)})^2) w_{ij}\,, \tag{29}$$

where the second equality follows from (28), whereas the last equality follows inspecting (27). It is not difficult to see that (29) can be compactly expressed as:

$$\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} = \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) \boldsymbol{W}\,, \tag{30}$$

that once transposed explains (24).

4) The previously obtained derivatives $\nabla_{\boldsymbol{o}^{(t)}} L$ and $\nabla_{\boldsymbol{h}^{(t)}} L$ are utilized to derive the derivatives for the network parameters $\boldsymbol{\theta}$. The gradient of the biases $\boldsymbol{b}, \boldsymbol{c}$ is:

$$\boxed{\nabla_{\boldsymbol{c}} L = \sum_{t=1}^{T} \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}}\right)^T \nabla_{\boldsymbol{o}^{(t)}} L \overset{(a)}{=} \sum_{t=1}^{T} \nabla_{\boldsymbol{o}^{(t)}} L} \tag{31}$$

$$\boxed{\nabla_{\boldsymbol{b}} L = \sum_{t=1}^{T} \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}}\right)^T \nabla_{\boldsymbol{h}^{(t)}} L \overset{(b)}{=} \sum_{t=1}^{T} \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L} \tag{32}$$

where (a) follows from (3), i.e., the Jacobian $\partial \boldsymbol{o}^{(t)}/\partial \boldsymbol{c}$ is the identity matrix, (b) descends from the same calculations done for $\partial \boldsymbol{h}^{(t+1)}/\partial \boldsymbol{h}^{(t)}$, by observing that the derivative $\partial a_i^{(t)}/\partial b_j = 1$ if $i = j$ and is zero otherwise.

Let y be a scalar and $\boldsymbol{X}$ be a matrix of size $m \times n$. We denote by $\nabla_{\boldsymbol{X}} y$ the matrix of derivatives of y with respect to the elements of x_{ij} of $\boldsymbol{X}$, formally:

$$\nabla_{\boldsymbol{X}} y = \begin{bmatrix} \frac{\partial y}{\partial x_{11}} & \frac{\partial y}{\partial x_{12}} & \cdots & \frac{\partial y}{\partial x_{1n}} \\ \frac{\partial y}{\partial x_{21}} & \frac{\partial y}{\partial x_{22}} & \cdots & \frac{\partial y}{\partial x_{2n}} \\ \vdots & \cdots & \ddots & \vdots \\ \frac{\partial y}{\partial x_{m1}} & \frac{\partial y}{\partial x_{m2}} & \cdots & \frac{\partial y}{\partial x_{mn}} \end{bmatrix} . \tag{33}$$

Armed with this definition, we can compute the derivatives of L with respect to the matrix parameters in $\boldsymbol{V}$, $\boldsymbol{U}$ and $\boldsymbol{W}$. For $\boldsymbol{V}$ we can write:

$$\boxed{\nabla_{\boldsymbol{V}} L \overset{(a)}{=} \sum_{t=1}^{T} \sum_{i=1}^{K} \left(\frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{\boldsymbol{V}} o_i^{(t)} \overset{(b)}{=} \sum_{t=1}^{T} (\nabla_{\boldsymbol{o}^{(t)}} L)(\boldsymbol{h}^{(t)})^T ,} \tag{34}$$

where (a) directly follows from the chain rule of calculus, whereas (2) follows from the structure of $\nabla_{\boldsymbol{V}} o_i^{(t)}$: a close inspection of (3) reveals that

$$\frac{\partial o_i^{(t)}}{\partial v_{kj}} = \begin{cases} h_j^{(t)} & k = i \\ 0 & k \neq i \, , \end{cases} \tag{35}$$

which means that $\nabla_{\boldsymbol{V}} o_i^{(t)}$ is a matrix with row i being equal to vector $(\boldsymbol{h}^{(t)})^T = [h_1^{(t)} \; h_2^{(t)} \; \ldots \; h_H^{(t)}]$, while all other elements are zero. The inner sum over i in (34) amounts to summing up K matrices (each of size $K \times H$), where for each the only non-zero row is row i, i.e., the non-zero elements are disjoint and the final result is:

$$\sum_{i=1}^{K} \left(\frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \begin{bmatrix} h_1^{(t)} \frac{\partial L}{\partial o_1^{(t)}} & h_2^{(t)} \frac{\partial L}{\partial o_1^{(t)}} & \cdots & h_H^{(t)} \frac{\partial L}{\partial o_1^{(t)}} \\ h_1^{(t)} \frac{\partial L}{\partial o_2^{(t)}} & h_2^{(t)} \frac{\partial L}{\partial o_2^{(t)}} & \cdots & h_H^{(t)} \frac{\partial L}{\partial o_2^{(t)}} \\ \vdots & \cdots & \ddots & \vdots \\ h_1^{(t)} \frac{\partial L}{\partial o_K^{(t)}} & h_2^{(t)} \frac{\partial L}{\partial o_K^{(t)}} & \cdots & h_H^{(t)} \frac{\partial L}{\partial o_K^{(t)}} \end{bmatrix} . \tag{36}$$

This explains (b) in equation (34). For matrix W the following observation is in order. The gradient operator $\nabla_{\boldsymbol{W}} f$ that is used in calculus takes into account the contribution of $\boldsymbol{W}$ to the value of f due to all edges in the unfolded computational graph. However, the equations that we want to implement use a recursive argument, which is the usual practice adopted in

backpropagation. According to this recursive view, we would like to compute the contribution of $\boldsymbol{W}$ to a single edge at a time of the computational graph. To cope with this, we define dummy variables $\boldsymbol{W}^{(t)}$ that are copies of $\boldsymbol{W}$ but with each $\boldsymbol{W}^{(t)}$ only used at time step t. Hence, $\nabla_{\boldsymbol{W}^{(t)}}$ denotes the (local) contribution of the weights at time t to the overall gradient. Below, we shall clarify this further.

$$\boxed{\nabla_{\boldsymbol{W}} L = \sum_{t=1}^{T} \sum_{i=1}^{H} \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \overset{(a)}{=} \sum_{t=1}^{T} \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \left(\boldsymbol{h}^{(t-1)}\right)^T .} \tag{37}$$

To explain (a) we take a close look at $\nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)}$. At time t, we have:

$$h_i^{(t)} = \tanh\left(a_i^{(t)}\right) \tag{38}$$

$$a_i^{(t)} = \left(\boldsymbol{b} + \boldsymbol{W}^{(t)} \boldsymbol{h}^{(t-1)} + \boldsymbol{U} \boldsymbol{x}^{(t)}\right)_i . \tag{39}$$

From which it follows that:

$$\frac{\partial a_i^{(t)}}{\partial w_{kj}^{(t)}} = \begin{cases} h_j^{(t-1)} & k = i \\ 0 & k \neq i \, . \end{cases} \tag{40}$$

Moreover, we can write:

$$\frac{\partial h_i^{(t)}}{\partial w_{kj}^{(t)}} = \frac{\partial h_i^{(t)}}{\partial a_i^{(t)}} \frac{\partial a_i^{(t)}}{\partial w_{kj}^{(t)}} = (1 - (h_i^{(t)})^2) \frac{\partial a_i^{(t)}}{\partial w_{kj}^{(t)}} = \begin{cases} (1 - (h_i^{(t)})^2) h_j^{(t-1)} & k = i \\ 0 & k \neq i \, , \end{cases} \tag{41}$$

which means that $\nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)}$ is a matrix where only row i in non-zero and is precisely given by (41). All the elements of this non-zero row are then multiplied by $\partial L / \partial h_i^{(t)}$. This explains (a) in (37). Note also that in (37) the internal state variables appear as $\boldsymbol{h}^{(t-1)}$. When $t = 1$, $\boldsymbol{h}^{(0)}$ refers to the initial state of the network.

The parameters of the last matrix $\boldsymbol{U}$ are obtained in a totally similar manner, leading to:

$$\boxed{\nabla_{\boldsymbol{U}} L = \sum_{t=1}^{T} \sum_{i=1}^{H} \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} = \sum_{t=1}^{T} \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \left(\boldsymbol{x}^{(t)}\right)^T .} \tag{42}$$

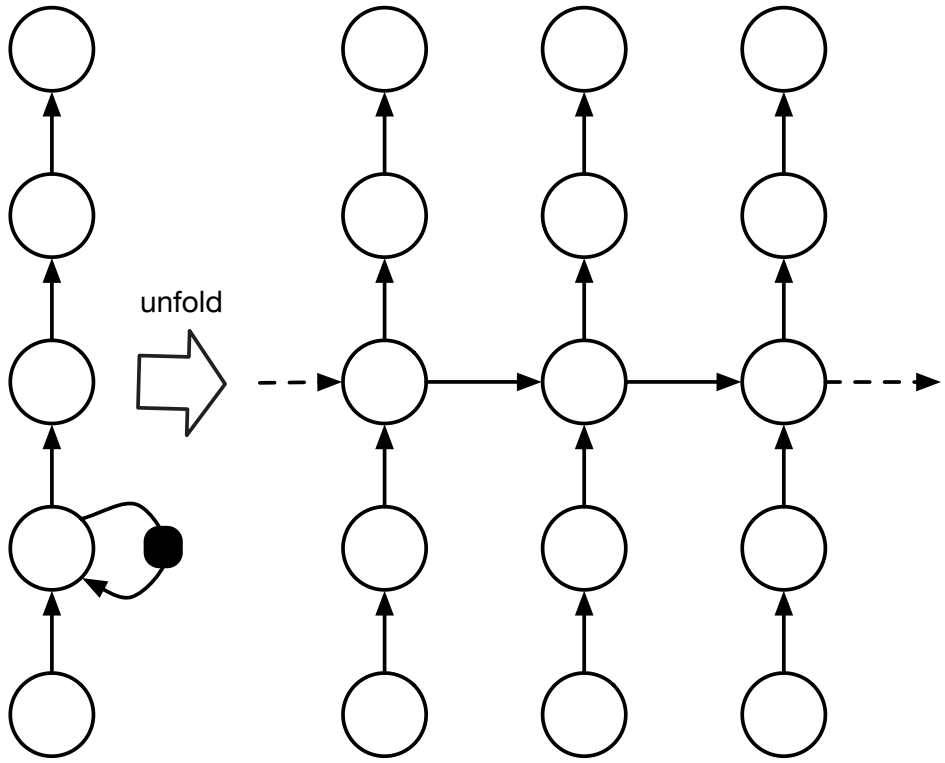

Fig. 1: Computational graph for a baseline RNN. **Left:** compact representation, with a time loop in the internal (inner) state. **Right:** unfolded representation, leading to a feed forward computational graph.