On recurrent neural networks

Giulio Galvan

10th April 2015

Contents

Chapter 1

Artificial neural networks

1.1 A family of models

An artificial neural network is a network of connected units called neurons or perceptrons, as can be seen in figure 1.1; each arc, which connects two neurons i and j, is associated with a weight w_{ji}. Perceptrons share the some structure for all models, what really define the model in the family is how the perceptrons are arrange and how are they conneceted, for example if there are cycles or not.

As you can see in figure 1.2 each neuron is *fed* with a set inputs which are the weighted outputs of other neurons. Formally the output of a perceptron ϕ_j is defined as:

$$\phi_j \triangleq \sigma(a_j) \tag{1.1}$$

$$a_j \triangleq \sum_l w_{jl}\phi_l + b_j \tag{1.2}$$

where w_{jl} is the weight of the connection between neuron l and neuron j, $\sigma(\cdot)$ is a non linear function and $b_j \in \mathbb{R}$ is called bias.

It sometime useful to think of a neurual network as series of layers, one on top of each other, as depicted in figure 1.3. The first layer is called the input layer and its units are *fed* with external inputs, the upper layers are called *hidden layers* because their's outputs are not observed from outside except the last one which is called *output layer* because it's output is the output of the net.

Whene we describe a network in this way is also useful to adopt a different notation: we describe the weights of the net with a set of matrixes W_i one for each layer, and neurons are no more linearly indexed, insted with refer to a neuron with a relative index with respect to the layer; this allows to write easier equations in matrix notation [1].

[1] In the rest of the book we will refer to the latter notation as *matrix notation* and to the previous one as *linear notation*

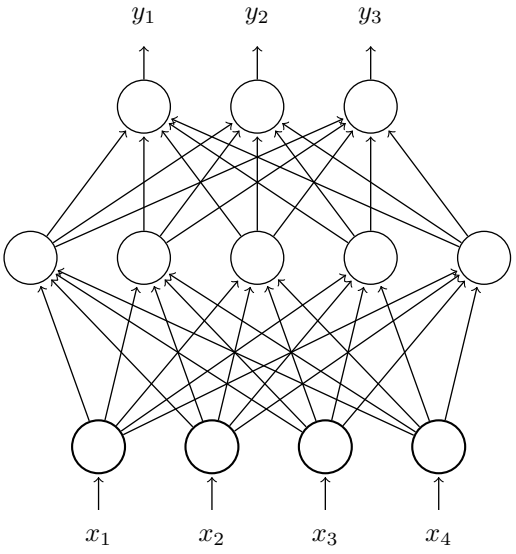

Figure 1.1: Artificial neural network example

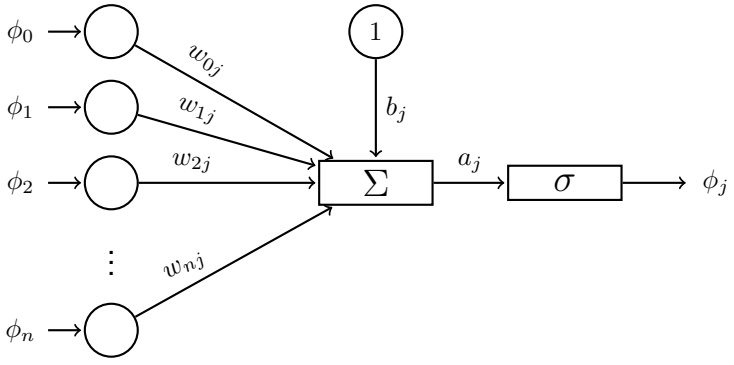

Figure 1.2: Neuron model

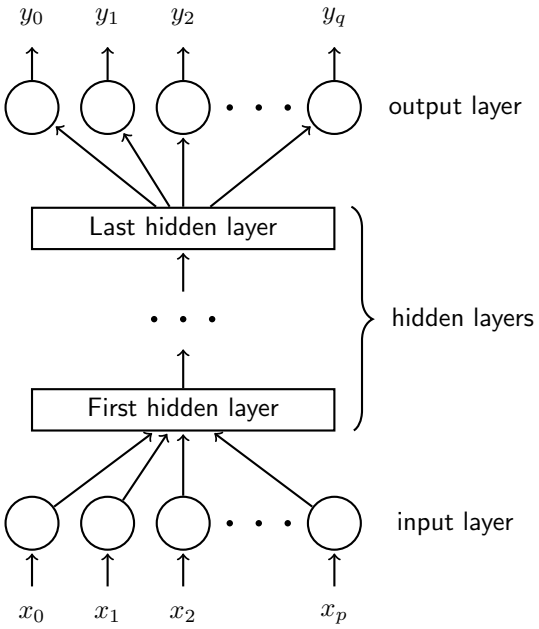

Figure 1.3: Layered structure of an artificial neural network

1.2 Feed foward neural networks

A feed foward neural network is an artificial neural network in which there are no cycles, that is to say each layer output is *fed* to the next one and connections to any earlier layer are not possible.

Definition 1 (Feed foward neural network)**.** A feed foward neural network is tuple

$$FFNN \triangleq < \boldsymbol{p}, \mathcal{W}, \mathcal{B}, \sigma(\cdot), F(\cdot) >$$

- $\boldsymbol{p} \in \mathbb{N}^U$ is the vector whose elements $p(k)$ are the number of neurons of layer k; U is the number of layers
- $\mathcal{W} \triangleq \{W^k_{p(k)\times p(k-1)}, k = 1, ..., U\}$ is the set of weight matrixes of each layer
- $\mathcal{B} \triangleq \{\boldsymbol{b}^k \in \mathbb{R}^{p(k)}, k = 1, ..., U\}$ is the set of bias vectors
- $\sigma(\cdot) : \mathbb{R} \to \mathbb{R}$ is the activation function
- $F(\cdot) : \mathbb{R}^{p(U)} \to \mathbb{R}^{p(U)}$ is the output function

Remark 1. Given a FFNN:

- The number of output units is $p(U)$

- The number of input units is $p(0)$
- The total number of weights is $\mathcal{N}(\mathcal{W}) \triangleq \sum_{k=1}^{U} p(k)p(k-1)$
- The total number of biases is $\mathcal{N}(\mathcal{B}) \triangleq \sum_{k=1}^{U} p(k)$

Definition 2 (Output of a FFNN)**.** Given a $FFNN$ and an input vector $\boldsymbol{x} \in \mathbb{R}^{p(1)}$ the output $\boldsymbol{y} \in \mathbb{R}^U$ of the net is defined by the following:

$$\boldsymbol{y} = F(\boldsymbol{a}^U) \tag{1.3}$$
$$\boldsymbol{\phi}^i \triangleq \sigma(\boldsymbol{a}_i), \qquad i = 1, ..., U \tag{1.4}$$
$$\boldsymbol{a}^i \triangleq W^i \cdot \boldsymbol{\phi}^{i-1} + \boldsymbol{b}^i \qquad i = 1, ..., U \tag{1.5}$$
$$\boldsymbol{\phi}^0 \triangleq \boldsymbol{x} \tag{1.6}$$

1.2.1 On expressivness of ffnn

1.2.2 Learning with ffnn

As we have seen in the previous section a model sush as $FFNN$ can approximate arbitrary well any smooth function, so a natural application of feed foward neural networks is machine learning. To model an optimization problem we first need to define a dataset D as

$$D \triangleq \{\overline{\boldsymbol{x}}^{(i)} \in \mathbb{R}^p, \overline{\boldsymbol{y}}^{(i)} \in \mathbb{R}^q, i = 1, ..., N\} \tag{1.7}$$

Then we need a loss function $L_D : \mathbb{R}^{\mathcal{N}(\mathcal{W})+\mathcal{N}(\mathcal{B})} \to \mathbb{R}_{\geq 0}$ over D defined as

$$L_D(\mathcal{W}, \mathcal{B}) \triangleq \frac{1}{N} \sum_{i=1}^{N} L(\overline{\boldsymbol{x}}^{(i)}, \boldsymbol{y}^{(i)}) \tag{1.8}$$

$L(\boldsymbol{x}, \boldsymbol{y}) : \mathbb{R}^{p(U)} \to \mathbb{R}$ is an arbitrary loss function computed on the i^{th} example. Note that $\boldsymbol{y}$ is the output of the networks, so it depends on $(\mathcal{W}, \mathcal{B})$

The problem is then fo find a $FFNN$ which minimize L_D. As we have seen feed foward neural network allow for large customization: the only variables in the optimization problem are the weights, the other parameters are said *hyper-parameters* and are determined *a priori*. Usually the output function is choosen depending on the output, for instance for multi-way classification is generally used the softmax function, for regression a simple identity function.

The activation function $\sigma(\cdot)$ is often choosen from:

$$sigmoid(x) = \frac{1}{1 - e^{-x}} \tag{1.9}$$

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.10}$$

$$ReLU(x) = \begin{cases} x & \text{if } x > 0. \\ 0 & \text{otherwise.} \end{cases} \tag{1.11}$$

For what concerns the number of layers and the number of units per layers they are choosen relying on experience or performing some kind of hyper-parameter tuning, which usually consists on training nets with some different configurations of such parameters and choosing the best one.

Once we have selected the values for all hyper-paramters the optimization problem becomes:

$$\min_{\mathcal{W},\mathcal{B}} L_D(\mathcal{W}, \mathcal{B}) \tag{1.12}$$

1.2.3 Gradient

Consider a $FFNN =< \boldsymbol{p}, \mathcal{W}, \mathcal{B}, \sigma(\cdot), F(\cdot) >$, let $L : \mathbb{R}^{p(U)} \to \mathbb{R}$ a loss function and $g(\cdot) : \mathbb{R}^{\mathcal{N}(\mathcal{W})+\mathcal{N}(\mathcal{B})} \to \mathbb{R}$ be the function defined by

$$g(\mathcal{W}, \mathcal{B}) \triangleq L(F(a^U(\mathcal{W}, \mathcal{B})))$$

$$\frac{\partial g}{\partial W^i} = \nabla L \cdot J(F) \cdot \frac{\partial \boldsymbol{a}^U}{\partial W^i} \tag{1.13}$$

$$= \frac{\partial g}{\partial \boldsymbol{a}^U} \cdot \frac{\partial \boldsymbol{a}^U}{\partial W^i} \tag{1.14}$$

We can easily compute $\frac{\partial g}{\partial \boldsymbol{a}^U}$ once we define $F(\cdot)$ and $L(\cdot)$, note that the weights are not involved in such computation. Let's derive an expression for $\frac{\partial \boldsymbol{a}^U}{\partial W^i}$. We will start deriving such derivative using linear notation. Let's consider a single output unit u and a weight w_{lj} linking neuron j to neuron l.

$$\frac{\partial a_u}{\partial w_{lj}} = \frac{\partial a_u}{\partial a_l} \cdot \frac{\partial a_l}{\partial w_{lj}} \tag{1.15}$$

$$= \delta_{ul} \cdot \phi_j \tag{1.16}$$

where we put

$$\delta_{ul} \triangleq \frac{\partial a_u}{\partial a_l}$$

.

Let $P(l)$ be the set of parents of neuron l, formally:

$$P(l) = \{k : \exists \text{ a link between } l \text{ and } k \text{ with weight } w_{lk}\} \tag{1.17}$$

We can compute δ_{ul} simply applying the chain rule, if we write it down in bottom-up style, as can be seen in figure 1.4, we obtain:

$$\delta_{ul} = \sum_{k \in P(l)} \delta_{uk} \cdot \sigma'(a_k) \cdot w_k l \tag{1.18}$$

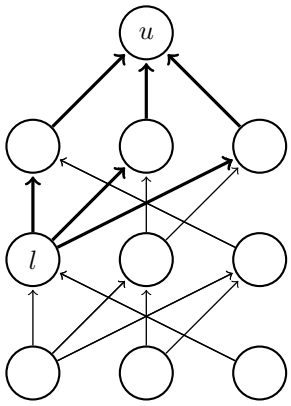

Figure 1.4: Nodes involved in $\frac{\partial a_u}{\partial a_l}$

The derivates with respect to biases are compute in the same way:

$$\frac{\partial a_u}{\partial b_l} = \frac{\partial a_u}{\partial a_l} \cdot \frac{\partial a_l}{\partial b_l} \tag{1.19}$$

$$= \delta_{ul} \cdot 1 \tag{1.20}$$

In matrix notation we can rewrite the previous equations as:

$$\frac{\partial a^U}{\partial W^i} = \frac{\partial a^U}{\partial a^i} \cdot \frac{\partial a^i}{\partial W^i} \tag{1.21}$$

$$\frac{\partial a^i}{\partial W^i_j} = \begin{bmatrix} \phi_1^{i-1} & 0 & \cdots & \cdots & 0 \\ 0 & \phi_2^{i-1} & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & \phi_{p(i-1)}^{i-1} \end{bmatrix} \tag{1.22}$$

$$\frac{\partial a^U}{\partial a^i} \triangleq \Delta^i = \begin{cases} \Delta^{i+1} \cdot diag(\sigma'(\boldsymbol{a}^{i+1})) \cdot W^{i+1} & \text{if } i < U \\ Id & \text{if } i == U \\ 0 & \text{otherwise.} \end{cases} \tag{1.23}$$

where

$$diag(\sigma'(\boldsymbol{a}^i)) = \begin{bmatrix} \sigma'(a_1^i) & 0 & \cdots & \cdots & 0 \\ 0 & \sigma'(a_2^i) & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & \sigma'(a_{p(i)}^i) \end{bmatrix} \tag{1.24}$$

$$\frac{\partial a^U}{\partial b^i} = \frac{\partial a^U}{\partial a^i} \cdot \frac{\partial a^i}{\partial b^i} \tag{1.25}$$

$$= \Delta^i \cdot Id \tag{1.26}$$

1.3 Recurrent neural networks

Recurrent neural networks differ from feed foward neural networks because of the presence of recurrent connections: at least one perceptron output at a given layer i is *fed* to another perceptron at a level $j < i$, as can be seen in figure 1.5. This is a key difference: as we will see in the next section, rnn are not only more powerfull than ffnn but as powerfull as turing machines.

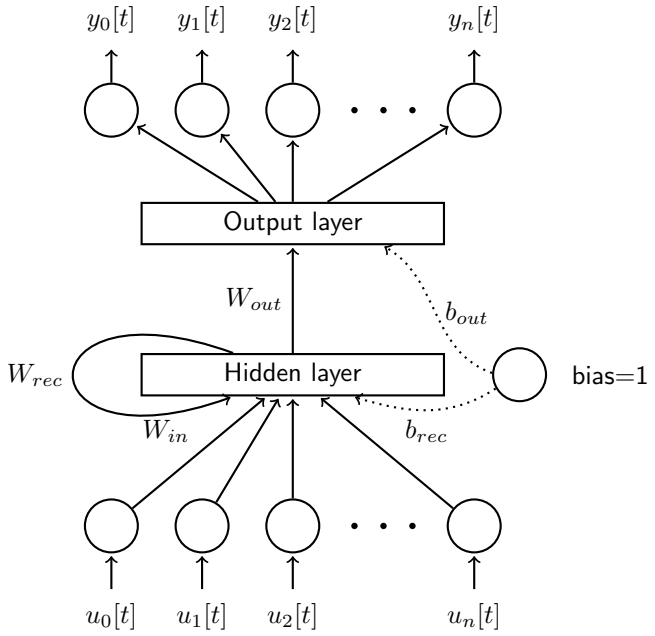

Figure 1.5: Rnn model

This difference in topology reflects also on the network's input and output domain, where in feed foward neural networks inputs and outputs were real valued vectors, recursive neural networks deal with sequences of vectors, that is to say that now time is also considered. One may argue that taking time (and sequences) into consideration is some sort of limitation because it restricts our model to deal only with a temporal inputs; that's not true, in fact we can apply rnn to non temporal data by considering space as the temporal dimension or we can feed the network with the same input for all time steps, or just providing no input after the first time step.

Definition 3 (Recurrent neural network)**.** A recurrent neural network is tuple

$$RNN \triangleq < \mathcal{W}, \mathcal{B}, \sigma(\cdot), F(\cdot) >$$

- $\mathcal{W} \triangleq \{W^{in}, W^{rec}, W^{out}\}$ where

  - W^{in} is the $r \times p$ input weight matrix
  - W^{rec} is the $r \times r$ recurrent weight matrix
  - W^{out} is the $o \times r$ output weight matrix
- $\mathcal{B} \triangleq \{\boldsymbol{b^{rec}}, \boldsymbol{b^{out}}\}$ where
  - $\boldsymbol{b}^{rec} \in \mathbb{R}^r$ is the bias vector for the recurrent layer
  - $\boldsymbol{b}^{out} \in \mathbb{R}^o$ is the bias vector for the output layer
- $\sigma(\cdot) : \mathbb{R} \to \mathbb{R}$ is the activation function
- $F(\cdot) : \mathbb{R}^o \to \mathbb{R}^o$ is the output function

Remark 2. Given a RNN

- The total number of weights si given by $\mathcal{N}(W) \triangleq rp + r^2 + ro$
- The number of biases by $\mathcal{N}(b) \triangleq r + o$
- p is the size of input vectors,
- r is the number of hidden units
- o is the size of output vectors.

Definition 4 (Output of a RNN)**.** Given a RNN and an input sequences $\{\boldsymbol{x}\}_{t=1,\dots,T}, \boldsymbol{x}_t \in \mathbb{R}^p$ the output sequence $\{\boldsymbol{y}\}_{t=1,\dots,T}, \boldsymbol{y}_t \in \mathbb{R}^o$ of the net is defined by the following:

$$\boldsymbol{y}_t \triangleq F(W^{out} \cdot \boldsymbol{a}_t + \boldsymbol{b}^{out}) \tag{1.27}$$

$$\boldsymbol{a}_t \triangleq W^{rec} \cdot \boldsymbol{\phi}_{t-1} + W^{in} \cdot \boldsymbol{x}_t + \boldsymbol{b}^{rec} \tag{1.28}$$

$$\boldsymbol{\phi}_t \triangleq \sigma(\boldsymbol{a}_t) \tag{1.29}$$

As we can understand from definition 3, there is only one recurrent layer, whose weights are the same for each time step, so one can asks where does the deepness of the network come from. The answer lies in the temporal unfolding of the network, in fact if we unfold the network step by step we obtain a structure similar to the structure of a feed foward neural network. As we can observe in figure 1.6, the unfolding of the network through time consist of putting identical version of the same reccurent layer on top of each other and linking the inputs of one layer to the next one. The key difference from feed foward neural networks if, as we have already observed, that the weights in each layer are identical, and of course the additional timed inputs which are different for each unfolded layer.

1.3.1 On expressivness of rnn

Rnns are as powerful as turing machines

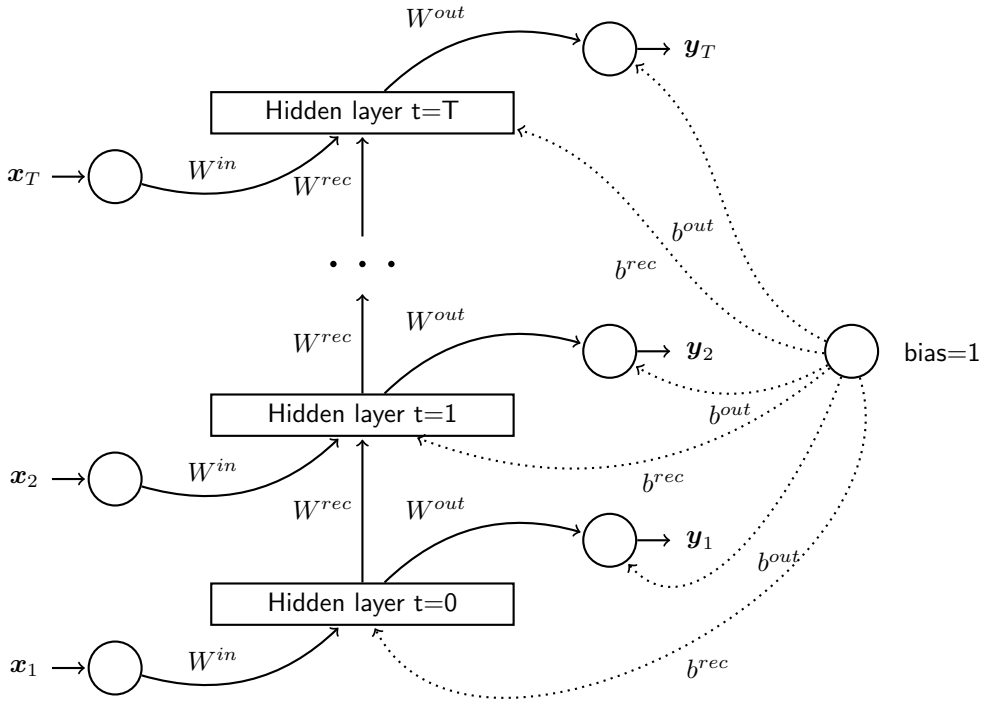

Figure 1.6: Unfolding of a rnn

1.3.2 Learning with ffnn

We can model an optimization problem in the same way we did for feed foward neural networks, the main difference is, again, that we now deal with temporal sequences so we need a slightly different loss function. Given a dataset D:

$$D \triangleq \{\{\overline{\boldsymbol{x}}\}_{t=1,...,T}, \overline{\boldsymbol{x}}_t \in \mathbb{R}^p, \{\overline{\boldsymbol{y}}\}_{t=1,...,T}, \overline{\boldsymbol{y}}_t \in \mathbb{R}^o; i = 1, ..., N\} \tag{1.30}$$

we define a loss function $L_D : \mathbb{R}^{\mathcal{N}(W)+\mathcal{N}(b)} \to \mathbb{R}_{\geq 0}$ over D as

$$L_D(\mathcal{W}, \mathcal{B}) \triangleq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} L_t(\overline{\boldsymbol{x}}_t^{(i)}, \boldsymbol{y}_t^{(i)}) \tag{1.31}$$

L_t is an arbitrary loss function at time step t.

The definition takes into account the output for each temporal step, depending on the task at hand, it could be relevant or not to consider intermediate

outputs; that's not a limitation, in fact we could define a loss which is computed only on the last output vector, at time T, and adds 0 for each other time step.

1.3.3 Gradient

Consider a $RNN =< \mathcal{W}, \mathcal{B}, \sigma(\cdot), F(\cdot) >$. Let $L : \mathbb{R}^o \to \mathbb{R}$ a loss function:

$$L(\mathcal{W}, \mathcal{B}) \triangleq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} L_t(\overline{\boldsymbol{x}}_t^{(i)}, \boldsymbol{y}_t^{(i)})$$

Let $g_t(\cdot) : \mathbb{R}^{\mathcal{N}(\mathcal{W})+\mathcal{N}(\mathcal{B})} \to \mathbb{R}$ be the function defined by

$$g_t(\mathcal{W}, \mathcal{B}) \triangleq L(F(a^t(\mathcal{W}, \mathcal{B})))$$

and

$$g(\mathcal{W}, \mathcal{B}) \triangleq \sum_{t=1}^{T} g_t(\mathcal{W}, \mathcal{B})$$

$$\frac{\partial g}{\partial W^{rec}} = \sum_{t=1}^{T} \nabla L_t \cdot J(F) \cdot \frac{\partial \boldsymbol{a}^t}{\partial W^{rec}} \tag{1.32}$$

$$= \sum_{t=1}^{T} \frac{\partial g_t}{\partial \boldsymbol{a}^t} \cdot \frac{\partial \boldsymbol{a}^t}{\partial W^{rec}} \tag{1.33}$$

As we noticed for ffnn it's easy to compute $\frac{\partial g_t}{\partial \boldsymbol{a}^t}$ once we define $F(\cdot)$ and $L(\cdot)$, note that the weights are not involved in such computation. Let's see how to compute $\frac{\partial \boldsymbol{a}^t}{\partial W^{rec}}$.

Let's consider a single output unit u, and a weight w_{lj}, we have

$$\frac{\partial a_u^t}{\partial w_{lj}} = \sum_{k=1}^{t} \frac{\partial a_u^t}{\partial a_l^k} \cdot \frac{\partial a_l^k}{\partial w_{lj}} \tag{1.34}$$

$$= \sum_{k=1}^{t} \delta_{lu}^{tk} \cdot \phi_j^{t-1} \tag{1.35}$$

where

$$\delta_{lu}^{tk} \triangleq \frac{\partial a_u^t}{\partial a_l^k} \tag{1.36}$$

Let's observe a first difference from ffnn case: since the weights are shared in each unfolded layer, in equation 1.34 we have to sum over time.

Let $P(l)$ be the set of parents of neuron l, defined as the set of parents in the unfolded network.

$$\delta_{lu}^{tk} = \sum_{h \in P(l)} \delta_{hu}^{tk} \cdot \sigma'(a_h^{t-1} \cdot w_{hl}) \tag{1.37}$$

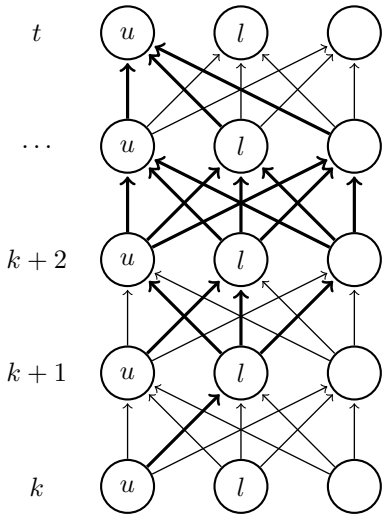

Figure 1.7: Nodes involved in $\frac{\partial a_u^t}{\partial a_l^k}$

In figure 1.7 we can see the arcs wich are involved in the derivatives in the unfolded network.

In matrix notation we have:

$$\frac{\partial \boldsymbol{a}^t}{\partial W^{rec}} = \sum_{k=1}^{t} \frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k} \cdot \frac{\partial \boldsymbol{a}^k}{\partial W^{rec}} \tag{1.38}$$

$$\frac{\partial a^k}{\partial W_j^{rec}} = \begin{bmatrix} \phi_1^{k-1} & 0 & \cdots & \cdots & 0 \\ 0 & \phi_2^{k-1} & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & \phi_r^{k-1} \end{bmatrix} \tag{1.39}$$

$$\triangleq \Delta^{tk} \tag{1.40}$$

$$\Delta^{tk} = \Delta^{t(k+1)} \cdot diag(\sigma'(\boldsymbol{a}^k)) \cdot W^{rec} \tag{1.41}$$

$$= \prod_{i=t-1}^{k} diag(\sigma'(\boldsymbol{a}^i)) \cdot W^{rec} \tag{1.42}$$

The derivatives with respect to W^{in} and $\boldsymbol{b}^{rec}$ have the same structure. The derivatives with respect to W^{out}, b^{out} are straightfoward:

$$\frac{\partial \boldsymbol{g}}{\partial W^{out}} = \sum_{t=1}^{T} \frac{\partial g_t}{\partial \boldsymbol{y^t}} \cdot J(F) \cdot \frac{\partial \boldsymbol{y}^t}{\partial W^{out}} \tag{1.43}$$

$$\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{b}^{out}} = \sum_{t=1}^{T} \frac{\partial g_t}{\partial \boldsymbol{y^t}} \cdot J(F) \cdot \frac{\partial \boldsymbol{y}^t}{\partial \boldsymbol{b}^{out}} \tag{1.44}$$

1.3.4 The vanishing and exploding gradient problem

The training of recurrent neural networks is afflicted by the so called *exploding* and *vanishing* gradient problem; such problem is directly linked to the notion of memory. When we talk about memory what are we really talking about is the dependecy of neuron output at a given time t from previous time steps, that is how ϕ^t depends on ϕ^k with $t > k$. This dependecy is captured by the expression $\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k}$. Obiuvsly when such expression equals zero it means neuron output at time t is not affected by output at time k, so if this situation occurs during training we have no hope to produce output sequence that take into consideration input sequence components previous to time k. The terms $\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k}$ are usually referred as *long term* contribution when $k << t$ or *short term* contributions otherwise.

We talk of *exploding* gradient problem when *long term* components grow exponentially, on the contrary of *short term* ones, causing the overall norm of the gradient to explode. We refer to *vanishing* gradient problem when, vice versa, *long terms* diminish exponentially.

We have seen in the previous section that

$$\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k} = \prod_{i=t-1}^{k} diag(\sigma'(\boldsymbol{a}^i)) \cdot W^{rec} \tag{1.45}$$

Intuitively we can understand why such problems arises, more evidently in *long term* components, just by looking at equation 1.45; We can notice each temporal contribution is the product of $l = t - k - 1$ jacobian matrix, so in *long term* components l is large and depening on eigenvalues of the matrixes in the product we can go exponentially fast towards 0 or infinty.

Let's now dig a bit deeper and rewrite equation 1.45 with respect to a couple of neurons i and j

$$\frac{\partial \boldsymbol{a}_i^t}{\partial \boldsymbol{a}_j^k} = \sum_{q \in P(j)} \sum_{l \in P(q)} \dots \sum_{h:i \in P(h)} w_{qj} \dots w_{jh} \cdot \sigma'(a_j^k)\sigma'(a_q^{k+1}) \dots \sigma'(a_i^{t-1}) \tag{1.46}$$

Observing the previous equation we can argue that each derivatives it's the sum of p^{t-k-1} terms; each term represents the path cost from neuron i to neuron j in the unfolded network, obviously there are p^{t-k-1} such paths. If we bind the cost $\sigma'(a_l^t)$ to neuron l in the t^{th} layer in the unfolded network we can read the path cost simply surfing the unfolded network multiply the weight of each arc we walk through and the cost of each neuron we cross, as we can see from figure 1.8.

We can further characterize each path cost noticing that we can separate two components, one that depends only on the weights $w_{qj} \dots w_{jh}$ and the other that depends both on the weights and the inputs $\sigma'(a_j^k)\sigma'(a_q^k) \dots \sigma'(a_i^{t-1})$.

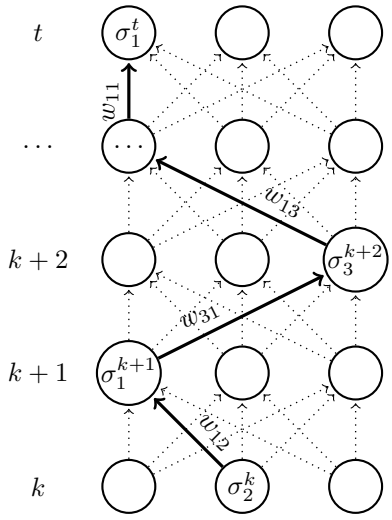

Figure 1.8: The cost for a path from neuron 2 at time k to neuron 1 at time t is $w_{12}w_{31}w_{13}\dots w_{11}\cdot\sigma_2^k\sigma_1^{k+1}\sigma_3^{k+2}\dots\sigma_1^{t-1}$

Hochreiter Analysis: Weak upper bound In this paragraph we report some useful consideration made by Hochreiter, please see [1] for more details.

Let's put:

$$\|A\|_{max} \triangleq \max_{i,j}|a_{ij}|$$

$$\sigma'_{max} \triangleq \max_{i=k,\dots,t-1}\{\left\|diag(\sigma'(a^i))\right\|_{max}\}$$

Since

$$\|A\cdot B\|_{max} <= p\cdot\|A\|\cdot\|B\|_{max} \qquad \forall A,B\in\mathbb{R}_{p\times p} \tag{1.47}$$

it holds:

$$\left\|\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k}\right\|_{max} = \left\|\prod_{i=t-1}^{k} diag(\sigma'(\boldsymbol{a}^i))\cdot W^{rec}\right\|_{max} \tag{1.48}$$

$$\leq \prod_{i=t-1}^{k} p\cdot\left\|diag(\sigma'(\boldsymbol{a}^i))\right\|_{max}\cdot\|W^{rec}\|_{max} \tag{1.49}$$

$$\leq \left(p\cdot\sigma'_{max}\cdot\|W^{rec}\|_{max}\right)^{t-k-1} \tag{1.50}$$

$$= \tau^{t-k-1} \tag{1.51}$$

where

$$\tau \triangleq p\cdot\sigma'_{max}\cdot\|W^{rec}\|_{max}$$

So we have exponential decay if $\tau < 1$; We can match this condition if $\|W^{rec}\|_{max} \leq \frac{1}{p\cdot\sigma'_{max}}$ As pointed out by Hochreiter in his work we can match this

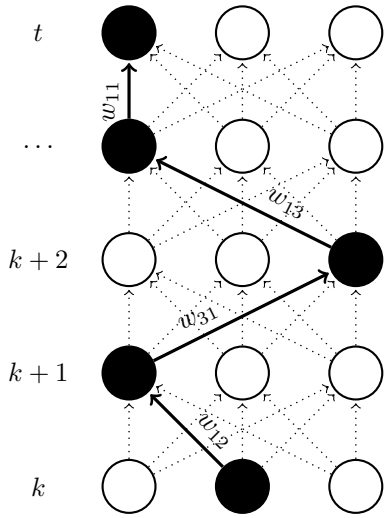

Figure 1.9: The cost for an enabled path from neuron 2 at time k to neuron 1 at time t is $w_{12}w_{31}w_{13}\dots w_{11}$

condition, in the case of sigmoid activation function by choosing $\|W^{rec}\|_{max} < \frac{4}{p}$.

Let's note that we would actually reach this upper bound for some i, j only if all the path cost have the same sign and the activations function take maximal value.

The ReLU case ReLU case is a bit special, because of it's derivative.

$$\sigma(x) = \begin{cases} x & \text{if } x > 0. \\ 0 & \text{otherwise.} \end{cases} \tag{1.52}$$

$$\sigma'(x) = \begin{cases} 1 & \text{if } x > 0. \\ 0 & \text{otherwise.} \end{cases} \tag{1.53}$$

ReLU's derivative is a step function, it can assume only two values: 1 when the neuron is active, 0 otherwise. Returning to the path graph we introduced earlier we can say that a path is *enabled* if each neuron in that path is active. In fact if we encounter a path wich cross a non active neuron it's path cost will be 0; on the contrary for an *enabled* path thw cost will be simply the product of weight of the arcs we went through, as we can see in figure 1.9

So $|(\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k})_{ij}|$ ranges from 0, when no path is enabled to, $|((W^{rec})^{t-k-1})_{ij}|$ when all paths are enabled, which is consistent with what we found in Hochreiter analysis.

Remark 3. We can see from ReLU's derivative expression that when a neuron is not active it doesn't contribute to the gradient, i.e the derivative with respct

to that neuron is zero. This leads to notice that we cannot learn to 'turn on' a neuron with ReLU activation function. This behaviour is somehow different from what we would have using sigmoid activation function, for instance, which have always derivative different from zero. Such activation function though exhibit a good behaviour near zero, they enter *saturation* regions when they are too distant from zero. In such region derivatives are close to zero, so again we cannot learn anything good from there. //DISEGNINO DELLE FUNZIONI

COMPORTAMENTO SBALIATO STRUTTURALE

Appendix A

Notation

Let $F : \mathbb{R}^N \to \mathbb{R}^{\mathbb{M}}$ be defined by

$$F(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \cdots, f_M(\boldsymbol{x}))) \text{ for some } f_i : \mathbb{R}^N \to \mathbb{R} \tag{A.1}$$

Derivative with respect to vector We define the derivative of $F(x(\boldsymbol{w}))$ with respect to a vector $\boldsymbol{w}$ of p elements as the $M \times p$ matrix

$$\frac{\partial F}{\partial \boldsymbol{w}} \triangleq \begin{bmatrix} \frac{\partial f_1}{\partial \boldsymbol{w}_1} & \frac{\partial f_1}{\partial \boldsymbol{w}_2} & \cdots & \cdots & \frac{\partial f_1}{\partial \boldsymbol{w}_p} \\ \frac{\partial f_2}{\partial \boldsymbol{w}_1} & \frac{\partial f_2}{\partial \boldsymbol{w}_2} & \cdots & \cdots & \frac{\partial f_2}{\partial \boldsymbol{w}_p} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_M}{\partial \boldsymbol{w}_1} & \frac{\partial f_M}{\partial \boldsymbol{w}_2} & \cdots & \cdots & \frac{\partial f_M}{\partial \boldsymbol{w}_p} \end{bmatrix} \tag{A.2}$$

Derivative with respect to matrix We define the derivative of $F(x(W))$ with respect to a vector W, being W_j the j^{th} column of a $p \times m$ matrix W as the $M \times (p \cdot m)$ matrix:

$$\frac{\partial F}{\partial W} \triangleq \left[\begin{array}{c|c|c|c} \frac{\partial F}{W_1} & \frac{\partial F}{W_2} & \cdots & \frac{\partial F}{W_m} \end{array} \right] \tag{A.3}$$

Bibliography

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory, 1995.