On recurrent neural networks

Giulio Galvan

18th 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 distinguish a particular model in the family of artificial neural networks is how the perceptron units are arranged and conneceted,for example whether there are cycles or not, and how data inputs are *fed* to the network.

As you can see in figure 1.2 each neuron is *fed* with a set inputs which are the weighted outputs of other neurons and/or other external inputs. 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's worth noticing that in this formulation the input ϕ_l can be the outputs of other neurons or provided external inputs.

The activation function The σ function is called *activation* function and should determine wheter a perceptron unit is *active* or not. When artificial neural networks where first conceived, back in 80es, trying to mimic the brain structure, such function was a simple threshshold function, trying to reproduce the behaviour of brain neurons: a neuron is *active*, i.e it's output ϕ_j is 1, if the

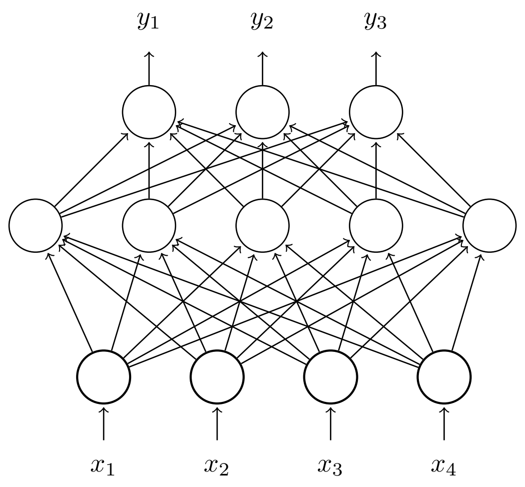

Figure 1.1: Artificial neural network example

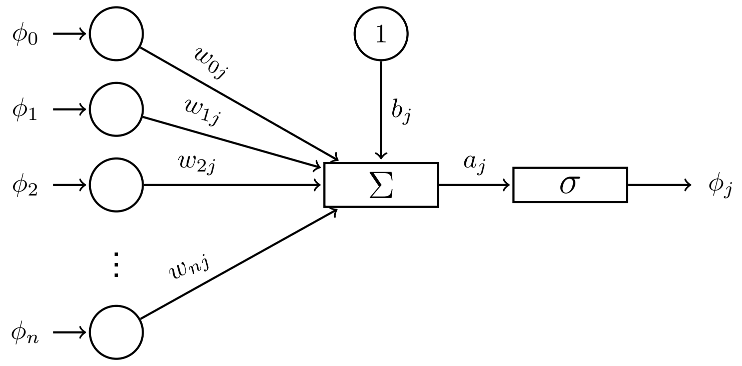

Figure 1.2: Neuron model

input stimuli $\sum_l w_{jl}\phi_l + b_j$ are greater than a given threshold τ.

$$\sigma_\tau(x) = \begin{cases} 1 & \text{if } x > \tau. \\ 0 & \text{otherwise.} \end{cases} \tag{1.3}$$

Such function, however, is problematic when we are to compute gradients because it's not continuoos, so one of the following function is usually choosen:

$$sigmoid(x) = \frac{1}{1+e^{-x}} \tag{1.4}$$

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.5}$$

These functions behave similarly to the threshold function, but, because of their *smoothness*, present no problems in computing gradients.

Another function which is becoming a very popular choice is the *rectified linear unit*:

$$ReLU(x) = \begin{cases} x & \text{if } x > 0. \\ 0 & \text{otherwise.} \end{cases} \tag{1.6}$$

ReLU activation function is rather different from previous activation functions, some of these difference, in particular with respect to gradients will be analyzied in later sections.

It's worth noticing that the activation function it's the only component which make artifical neuron networks a non linear model. Were we to choose a *linear* function as activation funciont we will end up with a simple linear model since the outputs of the network would be ultimately composed only of sums and products.

The bias term Let's consider the old threshold function σ_τ, and ask ourselves what the bias term is for, what does changing this term bring about. Suppose neuron j if fed with inputs $a_j = \sum_l w_{jl}\phi_l$; if $a_j > \tau$ that neuron is active otherwise it is not. Now, let's add the bias term to a_j; we obtain that neuron j is active if $a_j > \tau - b_j$. So the effect of the bias term is to change the activation threshold of a given neuron. Using bias terms in a neural network architecture gives us the ability to change the activation threshold for each neuron; that's particularly important considering that we can learn such bias terms. We can do these same considerations in an analogous way for all the other activation funcitons.

It's often 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

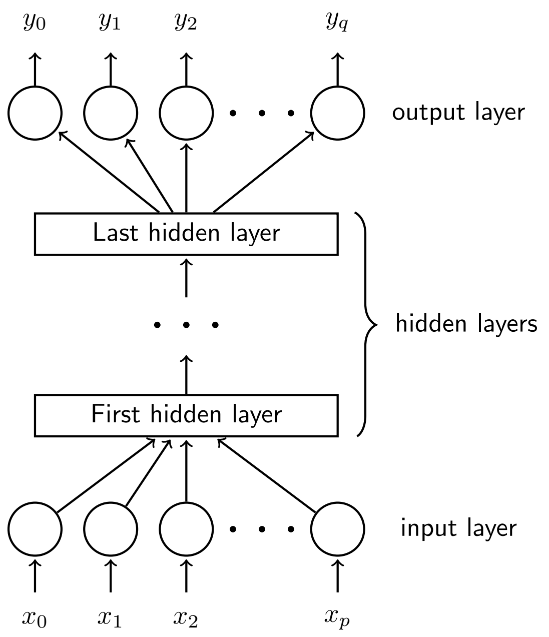

Figure 1.3: Layered structure of an artificial neural network

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].

1.2 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

[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*

- $\sigma(\cdot) : \mathbb{R} \to \mathbb{R}$ 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:

$$\boldsymbol{y} = F(\boldsymbol{a}^U) \tag{1.7}$$

$$\boldsymbol{\phi}^i \triangleq \sigma(\boldsymbol{a}_i), \qquad i = 1, ..., U \tag{1.8}$$

$$\boldsymbol{a}^i \triangleq W^i \cdot \boldsymbol{\phi}^{i-1} + \boldsymbol{b}^i \qquad i = 1, ..., U \tag{1.9}$$

$$\boldsymbol{\phi}^0 \triangleq \boldsymbol{x} \tag{1.10}$$

1.2.1 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

$$D \triangleq \{\overline{\boldsymbol{x}}^{(i)} \in \mathbb{R}^p, \overline{\boldsymbol{y}}^{(i)} \in \mathbb{R}^q, i = 1, ..., N\} \tag{1.11}$$

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

$$L_D(\mathcal{W}, \mathcal{B}) \triangleq \frac{1}{N} \sum_{i=1}^{N} L(\overline{\boldsymbol{x}}^{(i)}, \boldsymbol{y}^{(i)}) \tag{1.12}$$

$L(\boldsymbol{x}, \boldsymbol{y}) : \mathbb{R}^{p(U)} \to \mathbb{R}$ 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. 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:

$$\min_{\mathcal{W},\mathcal{B}} L_D(\mathcal{W},\mathcal{B}) \tag{1.13}$$

1.2.2 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

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

$$\frac{\partial g}{\partial W^i} = \nabla L \cdot J(F) \cdot \frac{\partial \boldsymbol{a}^U}{\partial W^i} \tag{1.14}$$
$$= \frac{\partial g}{\partial \boldsymbol{a}^U} \cdot \frac{\partial \boldsymbol{a}^U}{\partial W^i} \tag{1.15}$$

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.

$$\frac{\partial a_u}{\partial w_{lj}} = \frac{\partial a_u}{\partial a_l} \cdot \frac{\partial a_l}{\partial w_{lj}} \tag{1.16}$$
$$= \delta_{ul} \cdot \phi_j \tag{1.17}$$

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:

$$P(l) = \{k : \exists \text{ a link between } l \text{ and } k \text{ with weight } w_{lk}\} \tag{1.18}$$

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:

$$\delta_{ul} = \sum_{k \in P(l)} \delta_{uk} \cdot \sigma'(a_k) \cdot w_k l \tag{1.19}$$

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

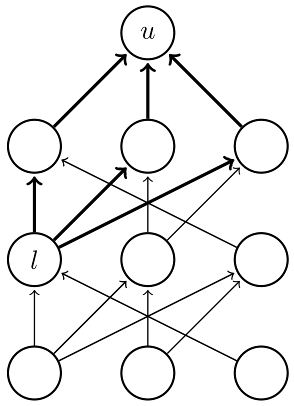

Figure 1.4: Nodes involved in $\frac{\partial a_u}{\partial a_l}$

$$\frac{\partial a_u}{\partial b_l} = \frac{\partial a_u}{\partial a_l} \cdot \frac{\partial a_l}{\partial b_l} \tag{1.20}$$

$$= \delta_{ul} \cdot 1 \tag{1.21}$$

In matrix notation we can rewrite the previous equations as:

$$\frac{\partial a^U}{\partial W^i} = \frac{\partial a^U}{\partial a^i} \cdot \frac{\partial a^i}{\partial W^i} \tag{1.22}$$

$$\frac{\partial a^i}{\partial W^i_j} = \begin{bmatrix} \phi_1^{i-1} & 0 & \cdots & \cdots & 0 \\ 0 & \phi_2^{i-1} & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & \phi_{p(i-1)}^{i-1} \end{bmatrix} \tag{1.23}$$

$$\frac{\partial a^U}{\partial a^i} \triangleq \Delta^i = \begin{cases} \Delta^{i+1} \cdot diag(\sigma'(\boldsymbol{a}^{i+1})) \cdot W^{i+1} & \text{if } i < U \\ Id & \text{if } i == U \\ 0 & \text{otherwise.} \end{cases} \tag{1.24}$$

where

$$diag(\sigma'(\boldsymbol{a}^i)) = \begin{bmatrix} \sigma'(a_1^i) & 0 & \cdots & \cdots & 0 \\ 0 & \sigma'(a_2^i) & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & \sigma'(a_{p(i)}^i) \end{bmatrix} \tag{1.25}$$

$$\frac{\partial a^U}{\partial b^i} = \frac{\partial a^U}{\partial a^i} \cdot \frac{\partial a^i}{\partial b^i} \tag{1.26}$$

$$= \Delta^i \cdot Id \tag{1.27}$$

Backpropagation Previous equations form the basis of the famous *backpropagation* algorithm which was first introduced by Rumelhart et al. [4]. The algorithm consists in two *passes*, a *foward pass* and a *backward pass* which give the name to the algorithm. The *foward pass* start from the first level, compute the hidden units value and the proceed to upper layers using the value of the hidden units $\boldsymbol{a^i}$ of previous layers which have been already computed. The *backward pass* instead, start from the upmost layer and computes Δ^i which we can compute, as we can see from equation 1.24 , once we know Δ^{i+1}, which we have computed in the previous step, and $\boldsymbol{a^i}$ which we have computed in the *foward pass*.

Backpropagation algorithm has time complexity $\mathcal{O}(\mathcal{N}(\mathcal{W}))$.

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.

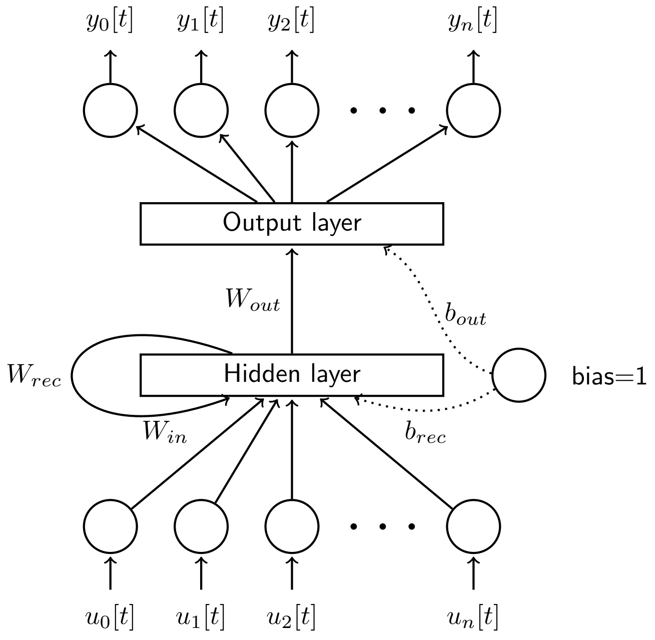

Figure 1.5: Rnn model

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 $\{\boldsymbol{x}\}_{t=1,\ldots,T}, \boldsymbol{x}_t \in \mathbb{R}^p$ the output sequence $\{\boldsymbol{y}\}_{t=1,\ldots,T}, \boldsymbol{y}_t \in \mathbb{R}^o$ of the net is defined by the following:

$$\boldsymbol{y}_t \triangleq F(W^{out} \cdot \boldsymbol{a}_t + \boldsymbol{b}^{out}) \tag{1.28}$$

$$\boldsymbol{a}_t \triangleq W^{rec} \cdot \boldsymbol{\phi}_{t-1} + W^{in} \cdot \boldsymbol{x}_t + \boldsymbol{b}^{rec} \tag{1.29}$$

$$\boldsymbol{\phi}_t \triangleq \sigma(\boldsymbol{a}_t) \tag{1.30}$$

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.

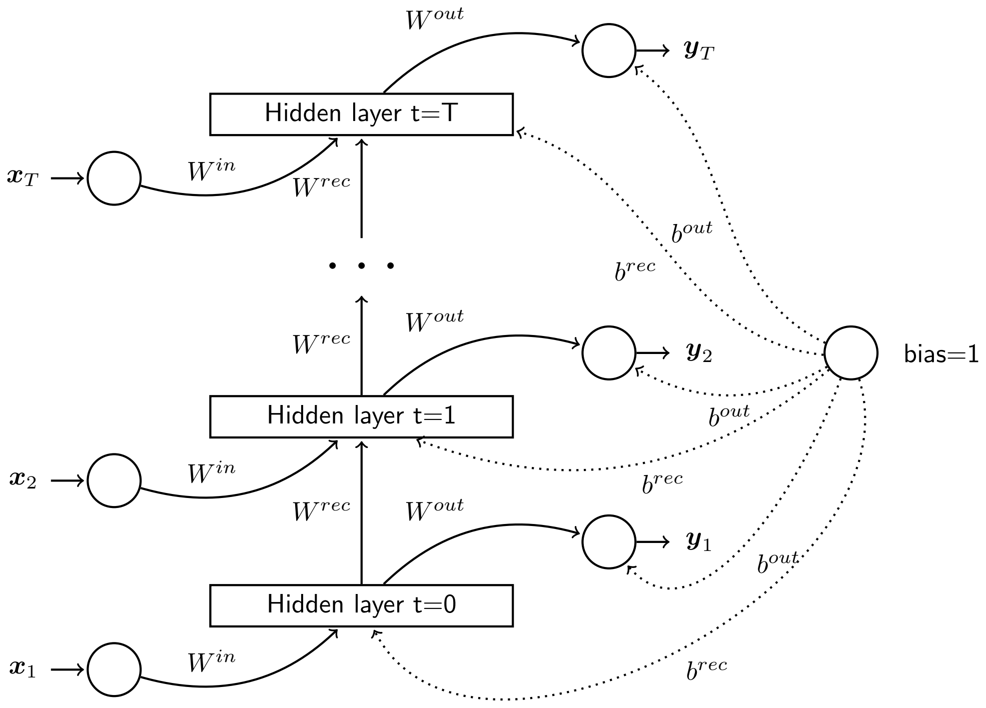

Figure 1.6: Unfolding of a rnn

1.3.1 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:

$$D \triangleq \{\{\overline{\boldsymbol{x}}\}_{t=1,...,T}, \overline{\boldsymbol{x}}_t \in \mathbb{R}^p, \{\overline{\boldsymbol{y}}\}_{t=1,...,T}, \overline{\boldsymbol{y}}_t \in \mathbb{R}^o; i = 1, ..., N\} \tag{1.31}$$

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

$$L_D(\mathcal{W}, \mathcal{B}) \triangleq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} L_t(\overline{\boldsymbol{x}}_t^{(i)}, \boldsymbol{y}_t^{(i)}) \tag{1.32}$$

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.

1.3.2 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

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

$$\frac{\partial g}{\partial W^{rec}} = \sum_{t=1}^{T} \nabla L_t \cdot J(F) \cdot \frac{\partial \boldsymbol{a}^t}{\partial W^{rec}} \tag{1.33}$$

$$= \sum_{t=1}^{T} \frac{\partial g_t}{\partial \boldsymbol{a}^t} \cdot \frac{\partial \boldsymbol{a}^t}{\partial W^{rec}} \tag{1.34}$$

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

$$\frac{\partial a_u^t}{\partial w_{lj}} = \sum_{k=1}^{t} \frac{\partial a_u^t}{\partial a_l^k} \cdot \frac{\partial a_l^k}{\partial w_{lj}} \tag{1.35}$$

$$= \sum_{k=1}^{t} \delta_{lu}^{tk} \cdot \phi_j^{t-1} \tag{1.36}$$

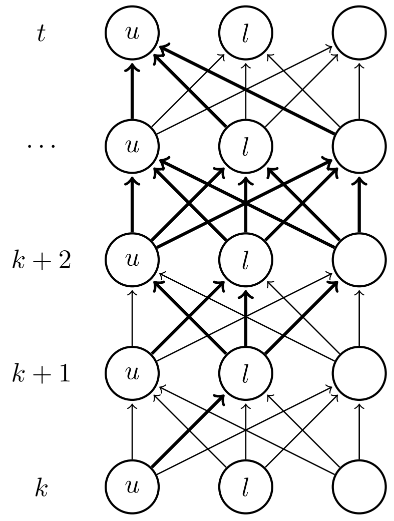

Figure 1.7: Nodes involved in $\frac{\partial a_u^t}{\partial a_l^k}$

where

$$\delta_{lu}^{tk} \triangleq \frac{\partial a_u^t}{\partial a_l^k} \tag{1.37}$$

Let's observe a first difference from ffnn case: since the weights are shared in each unfolded layer, in equation 1.35 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.

$$\delta_{lu}^{tk} = \sum_{h \in P(l)} \delta_{hu}^{tk} \cdot \sigma'(a_h^{t-1} \cdot w_{hl}) \tag{1.38}$$

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

In matrix notation we have:

$$\frac{\partial \boldsymbol{a}^t}{\partial W^{rec}} = \sum_{k=1}^{t} \frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k} \cdot \frac{\partial \boldsymbol{a}^k}{\partial W^{rec}} \tag{1.39}$$

$$\frac{\partial a^k}{\partial W_j^{rec}} = \begin{bmatrix} \phi_1^{k-1} & 0 & \cdots & \cdots & 0 \\ 0 & \phi_2^{k-1} & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & \phi_r^{k-1} \end{bmatrix} \tag{1.40}$$

$$\triangleq \Delta^{tk} \tag{1.41}$$

$$\Delta^{tk} = \Delta^{t(k+1)} \cdot diag(\sigma'(\boldsymbol{a}^k)) \cdot W^{rec} \tag{1.42}$$

$$= \prod_{i=t-1}^{k} diag(\sigma'(\boldsymbol{a}^i)) \cdot W^{rec} \tag{1.43}$$

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:

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

$$\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{b}^{out}} = \sum_{t=1}^{T} \frac{\partial g_t}{\partial \boldsymbol{y^t}} \cdot J(F) \cdot \frac{\partial \boldsymbol{y}^t}{\partial \boldsymbol{b}^{out}} \tag{1.45}$$

Backpropagation through time (BPTT) *Backpropagation through time* is an extension of the *backpropagation* algorithm we described for FNNs, we can think of BPTT simply as a standard BP in the unfolded network. The same consideration done for BP also apply for BPTT, the difference is of course in how derivatives are computed, equation 1.43. Time complexity is easily derived noticing that in the unfolded network there are $n{\cdot}T$ units, where n is the number of units of the RNN .This yields time complexity $\mathcal{O}(\mathcal{N}(\mathcal{W}) \cdot T)$. Please see [7] for more details.

1.3.3 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. 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.

Vanishing gradient means long term components approach zero value, this in turn leads to the fact that the output of the net won't depend on inputs of distant temporal steps, i.e the output sequence is determined only using recent temporal input: we say that the network doesn't have memory. Evidently this can have catastrofic effects on the classification error. Imagine we would like to classiy an input sequence as positive wheter or not it contains a zero character.

It would seem a rather easy task, a task other classification models wouldn't have difficulties with. If the neural network we are trainign has *vanishing* gradient issue, it means it perform classification only using the most recent temporal input. What if the zero character was at the beginning of the sequence? Of course the predicition would be wrong.

Exploding gradient seems to be a different kind of a problem, it does not affect the ability of the network to use information from distant temporal step, on the contrary we have very strong information about where to go. One could argue that some components, namely, the long term ones, have gradient norm exponentially bigger than short term ones. I fail to see why this could be a problem: of course output of the first time stepes influences all successive steps, so changing the output of a long term neuron do imply big changes in the output of more distant in time neurons. The only reason i see for considering this a problem is that big gradient norm, in general, is a problem if we are using algortithms like gradient descent with constant step. If we are to compute a step in the gradient direction with a fixed step and gradient has too big norm we may make a too big step.

Let's now return to the nature of the problem and try to explaining the mechanics of it.

We have seen in the previous section that

$$\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k} = \prod_{i=t-1}^{k} diag(\sigma'(\boldsymbol{a}^i)) \cdot W^{rec} \tag{1.46}$$

Intuitively we can understand why such problems arises, more evidently in *long term* components, just by looking at equation 1.46; 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.

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:

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

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

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 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.

Explaining the problem using network's graph Let's now dig a bit deeper and rewrite equation 1.46 with respect to a couple of neurons i and j

$$\frac{\partial \boldsymbol{a}_i^t}{\partial \boldsymbol{a}_j^k} = \sum_{q \in P(j)} \sum_{l \in P(q)} \cdots \sum_{h: i \in P(h)} w_{qj} \dots w_{jh} \cdot \sigma'(a_j^k)\sigma'(a_q^{k+1}) \dots \sigma'(a_i^{t-1}) \tag{1.52}$$

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 $\sigma'(a_j^k)\sigma'(a_q^k) \dots \sigma'(a_i^{t-1})$.

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

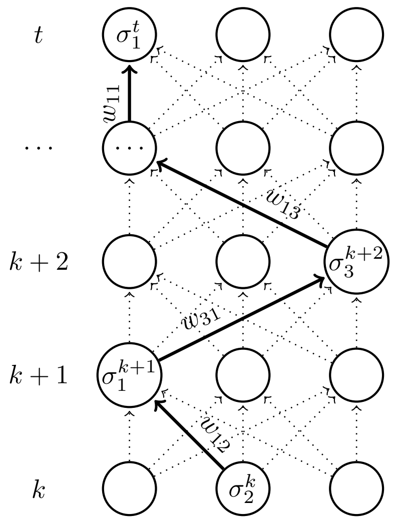

Figure 1.8: The cost for a path from neuron 2 at time k to neuron 1 at time t is $w_{12}w_{31}w_{13}\dots w_{11}\cdot\sigma_2^k\sigma_1^{k+1}\sigma_3^{k+2}\dots\sigma_1^{t-1}$

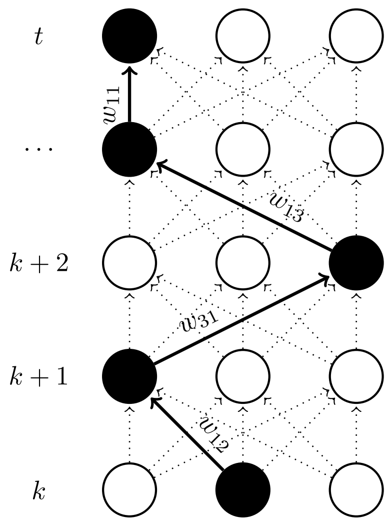

Figure 1.9: The cost for an enabled path from neuron 2 at time k to neuron 1 at time t is $w_{12}w_{31}w_{13}\dots w_{11}$

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 and all path cost have the same sign, which is consistent with what we found in Hochreiter analysis.

COLLEGARE MELIO Let's pretend we have found, with some learning tecnique, an assigment for all the weights which causes the gradient to have zero norm. We could be happy with it and claim to have 'solved' the problem. However, by chance, we discover that $\frac{\partial \boldsymbol{a}^T}{\partial \boldsymbol{a}^k}$ has zero norm for all time steps $k < \tau$. So, the output of the network doesn't depend on the inputs of the sequence for those time steps. In other words we have found a possibly optimal solution for the truncated sequence $x_{k=\tau:T}$. The solution we have found is an optimal candidate to be a bad local minimum.

As a final observation on this matter it's worth noticing how a bad initiliazation of W^{rec} can lead to poor solutions or extremely large convergence time just because such initialization imply $\frac{\partial \boldsymbol{a}^t}{\partial \boldsymbol{a}^k}$ approaching zero norm for $t >> k$. Moreover, even if we somehow provide an initialization matrix wich is unafflicted by this curse, it's certainy possible that we reach such a bad matrix during learning phase. Several techniques have been proposed to overcome this problem, they will be the topic of later chapters.

1.4 Activation functions and gradient

INTRO

Definition 5. A function $f(\cdot) : \mathbb{R} \to \mathbb{R}$ is said to be a *squashing* function if

$$\lim_{x \to +\infty} f(x) = b \tag{1.53}$$

$$\lim_{x \to -\infty} f(x) = a \tag{1.54}$$

where $b, a \in \mathbb{R}$ and $b > a$

Step function, ramp function and all sigmoidal functions are all examples of squashing functions.

Sigmoid

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{1.55}$$

$$sigmoid'(x) = sigmoid(x) \cdot (1 - sigmoid(x)) \tag{1.56}$$

As we can see from figure 1.10, the sigmoid derivative has only one maximum in 0 where it assume value 0.25. Receding from 0, in both direction leads to regions where the the derivative take zero value, such regions are called *saturation* regions. If we happen to be in such regions, for a given neuron, we cannot learn anything since that neuron doesn't contribute to the gradient.

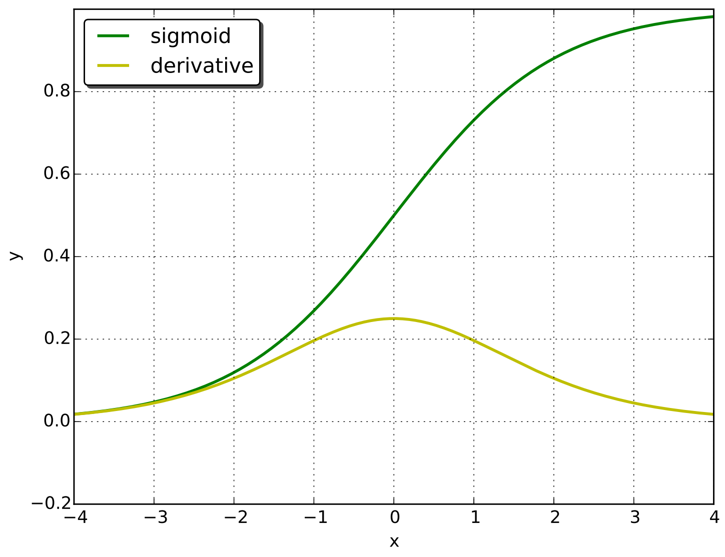

Figure 1.10: sigmoid and it's derivative

Tanh As we can see from figure 1.11 tanh (and it's derivative) have a behaviour similar to the sigmoid one; Again we have two saturation region towards infinity: that's typical of all squashing functions.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.57}$$

$$tanh'(x) = 1 - tanh^2(x) \tag{1.58}$$

ReLU

$$ReLU(x) = \begin{cases} x & \text{if } x > 0. \\ 0 & \text{otherwise.} \end{cases} \tag{1.59}$$

$$ReLU'(x) = \begin{cases} 1 & \text{if } x > 0. \\ 0 & \text{otherwise.} \end{cases} \tag{1.60}$$

ReLU is a bit different from other activation function seen so far: the main difference is that's it's not a squashing function. As we can see from figure 1.12, ReLU's derivative is the step function; it has only one *saturation* region $(-\inf, 0]$ and a region in which is always takes value one, $(0, +\inf]$ This leads to the fact that we cannot learn to 'turn on' a switched off neuron $(x < 0)$, but we have no *saturation* region toward $+\inf$.

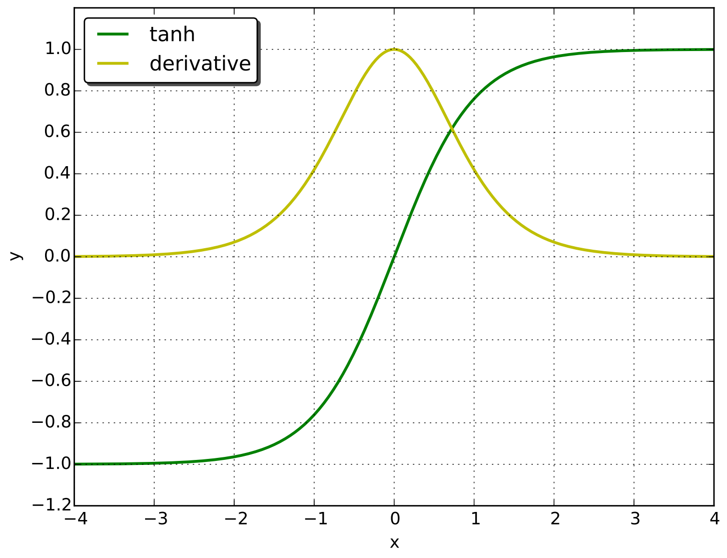

Figure 1.11: tanh and it's derivative

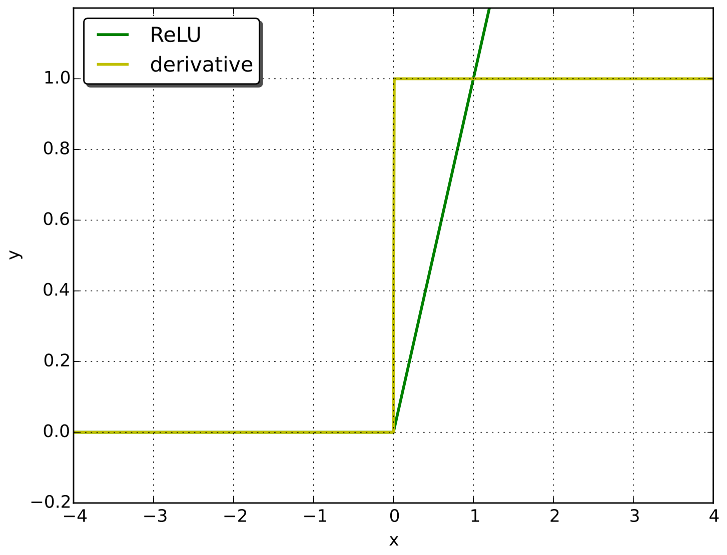

Figure 1.12: ReLU and it's derivative

1.5 On expressiveness

In this section we will investigate the expressive power of neural networks, presenting some results that motivate the use of neural networks as learning model.

ALTRO

One of the first import results regarding the expressive power of neural networks it's due to Hornik et al. [2] which basically states *'Multilayered feed foward networks with at least one hidden layer, using an arbitrary squashing function can approximate virtually any function of interest to any desired degree of accuracy provided sufficiently many hidden units are available'*.

To give a more formal result we need first to define what *approximate to any degree of accuracy means*, this concept is captured in definition 6

Definition 6. A subset S of $\mathbb{C}^n$ (continuoos functions in $\mathbb{R}^n$) is said to be *uniformly dense on compacta in* $\mathbb{C}^n$ if $\forall$ compact set $K \subset \mathbb{R}^n$ holds: $\forall \epsilon > 0$, $\forall g(\cdot) \in \mathbb{C}^n$ $\exists f(\cdot) \in S$ such that $\sup_{x \in K} \|f(x) - g(x)\| < \epsilon$

Hornik result is contained in theorem 1.

Theorem 1. For every squashing function σ, $\forall n \in \mathbb{N}$, feed foward neural networks with one hidden layer are a class of functions which is *uniformly dense on compacta in* $\mathbb{C}^n$.

Theorem 1 extends also to Borel measurable functions, please see [2] for more details.

A survey of other approches, some of which constructive, which achieve similar results can be found in [5] At the momement I dont know of any results concerning ReLU activation function.

This results implies that FNN are *universal approximators*, this is a strong argument for using such models in machine learning. It's important to notice, however, that the theorem holds if we have *sufficiently many* units. In practice the number of units will bounded by the machine capabilities and by computational time, of course greater the number of units greater will be the learning time. This will limit the expressiveness of the network to a subset of all measurable functions.

Let's now turn our attention to RNNs and see how the architectural changes, namely the addition of backward links, affect the expressive power of the model. It's suffice to say that RNNs are as powerfull as turing machine. Siegelman and Sontag [6] proved the existence of a finite neural network, with sigmoid activation function, which simulates a universal Turing machine. Hyötyniemi [3] proved, equivalently, that turing machine are recurrrent neural network showing how to build a network, using instead ReLU activation function, that performs step by step all the instruction of a computer program.

This seems extremely good news, since we could simulate turing machines, hence all algorithms we can think of, using a recurrent neural network with a finite number of units; recall that for FFNN we had to suppose infinitely many units to obtain the universal approximator property. Of course there is a pitfall: we can simulate any turing machine but we have to allow sufficiently many time steps.

SPACE-TIME STRUGGLE

IMPLICT REPRESENTATION

DISCORSI SULL UTILIZZO

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.

[2] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, July 1989.

[3] Heikki Hyotyniemi. Turing machines are recurrent neural networks, 1996.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[5] Franco Scarselli and Ah Chung Tsoi. Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results. *Neural Netw.*, 11(1):15–37, January 1998.

[6] Hava T. Siegelmann and Eduardo D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4:77–80, 1991.

[7] Ronald J. Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2:490–501, 1990.