# On recurrent neural networks

Giulio Galvan

26th March 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, each arc which connects two neurons $i$ and $j$ is associated with a weight $wji$. 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 1.1 each neuron is *fed* with a set inputs which are the weighted outputs of other neurons. Formally the output of a perceptron $\phi_j$ 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.

A neural network looks like the one in figure 1.2

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*

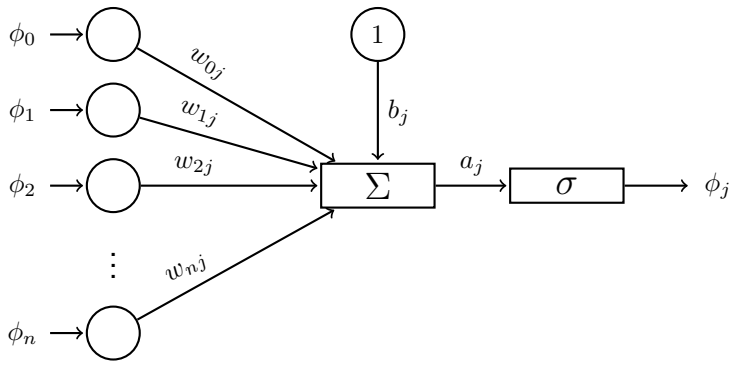


Figure 1.1: Neuron model

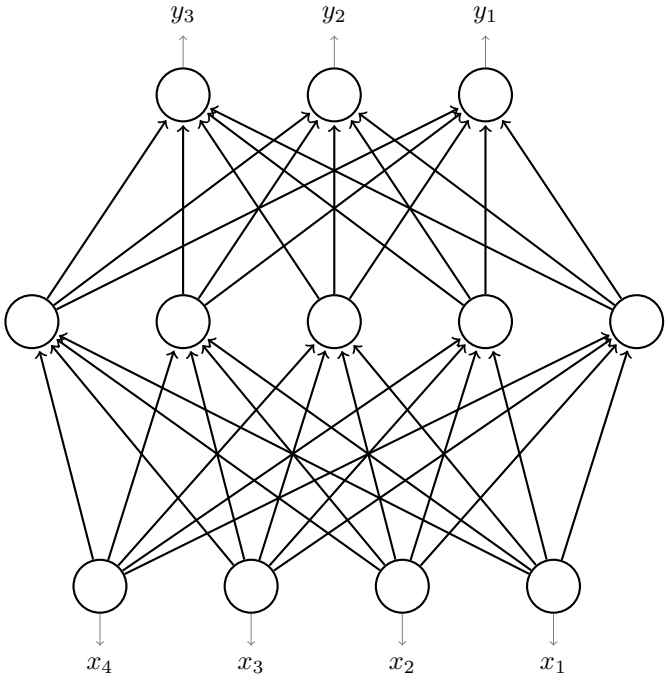


Figure 1.2: Artificial neural network example

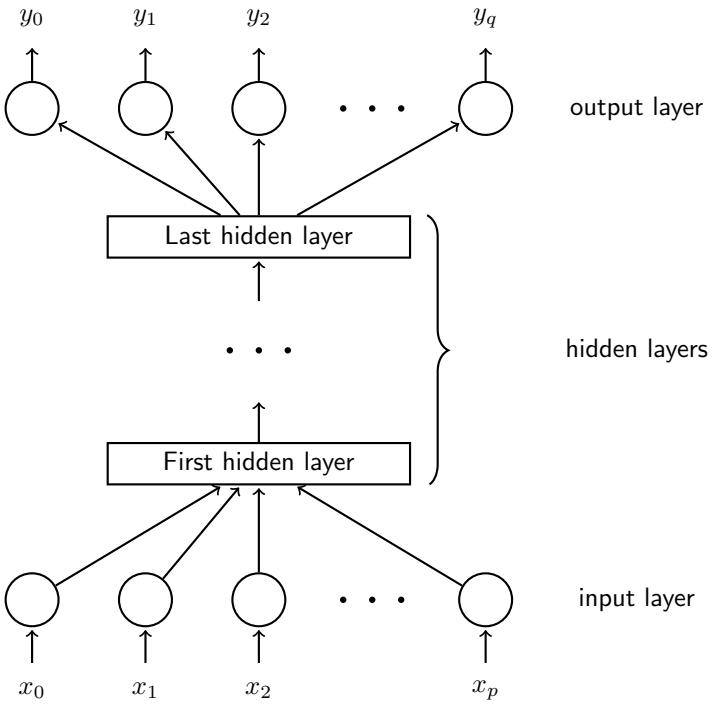


Figure 1.3: Layered structure of an artificial neural network

## 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

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

- $\boldsymbol{p} \in \mathbb{N}^U$ is the vector whose elements $p(k)$ are the number of neurons of layer $k$; $U$ is the number of layers
- $W \triangleq \{W^k_{p(k+1)\times p(k)}, k = 1, ..., U\}$ is the set of weight matrixes of each layer
- $b \triangleq \{\boldsymbol{b}^k \in \mathbb{R}^{p(k)}, k = 1, ..., U\}$ 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 $p(1)$
- The total number of weights is $\mathcal{N}(W) \triangleq \sum_{k=1}^{U} p(k+1)p(k)$
- The total number of biases is $\mathcal{N}(b) \triangleq \sum_{k=1}^{U} p(k)$

**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:

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

### 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

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

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

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

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

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:

$$\min_{W,b} \qquad L_D(W, b) \tag{1.12}$$

$$\tag{1.13}$$

### 1.2.3 Gradient

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

$$g(W, b) \triangleq L(F(a(W, b)))$$

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

$$= \frac{\partial g}{\partial \boldsymbol{a}} \cdot \frac{\partial \boldsymbol{a}^{U}}{\partial \boldsymbol{w}} \tag{1.15}$$

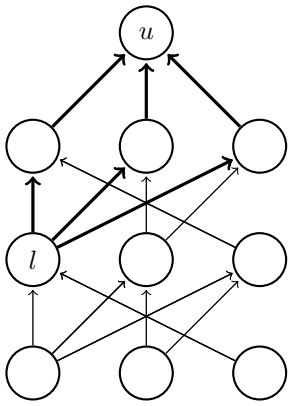


Figure 1.4: Nodes involved in $\frac{\partial a_u}{\partial a_l}$

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

$$\frac{\partial a_u}{\partial w_l j} = \frac{\partial a_u}{\partial a_l} \cdot \frac{\partial a_l}{\partial w_l j} \tag{1.16}$$

$$= \delta_l^u \cdot \phi_j \tag{1.17}$$

where we put

$$\delta_l^u \triangleq \frac{\partial a_u}{\partial a_l}$$

.

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 $\delta_l^u$ simply applying the chain rule, if we write it down in bottom-up style, as can be seen in figure 1.4, we obtain:

$$\delta_l^u = \sum_{k \in P(l)} \delta_k^u \cdot \sigma'(a_k) \cdot w_k l \tag{1.19}$$

In matrix notation we can rewrite the previous equations as:

$$\frac{\partial a^U}{\partial \boldsymbol{w^i}} = \frac{\partial a^U}{\partial a^i} \cdot \frac{\partial a^i}{\partial \boldsymbol{w^i}} \tag{1.20}$$

$$\frac{\partial a^i}{\partial \boldsymbol{w^i}} = \begin{bmatrix} \phi_1^{i-1} & 0 & \cdots & \cdots & 0 \\ 0 & \phi_2^{i-1} & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & \phi_{p(i-1)}^{i-1} \end{bmatrix} \tag{1.21}$$

$$\frac{\partial a^U}{\partial a^i} \triangleq \Delta^i = \Delta^{i+1} \cdot diag(\sigma'(\boldsymbol{a}^{i+i})) \cdot W^{i+1} \tag{1.22}$$

where

$$diag(\sigma'(\boldsymbol{a}^{i+i})) = \begin{bmatrix} \sigma'(a_1^{i+1}) & 0 & \cdots & \cdots & 0 \\ 0 & \sigma'(a_2^{i+1}) & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & \cdots & \sigma'(a_{p(i+1)}^{i+1}) \end{bmatrix} \tag{1.23}$$

## 1.3 Recursive 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 $j < i$. 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.

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.

FIGURE RNN

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

$$RNN \triangleq < W_{in}, W_{out}, W_{rec}, \boldsymbol{b}_{out}, \boldsymbol{b}_{rec}, \sigma(\cdot), f(\cdot) >$$

- $W^{in}$ 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
- $\boldsymbol{b}^{rec} \in \mathbb{R}^r$ 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

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

Given a $RNN$ and an input sequences $\{\boldsymbol{x}\}_t$ the output sequence $\{\boldsymbol{y}\}_t$ of the net is defined by the following:

$$\boldsymbol{y}_t \triangleq f(W^{out} \cdot \boldsymbol{\phi}_t + \boldsymbol{b}^{out}) \tag{1.24}$$

$$\boldsymbol{a}_t \triangleq W^{rec} \cdot \boldsymbol{\phi}_{t-1} + W^{in} \cdot \boldsymbol{x}_t + \boldsymbol{b}^{rec} \tag{1.25}$$

$$\boldsymbol{\phi}_t \triangleq \sigma(\boldsymbol{a}_t) \tag{1.26}$$

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 **??**, 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.

UNFOLDING FIGURE

### 1.3.1 On expressivness of rnn

Rnns are as powerful as turing machines

### 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 $D$:

$$D \triangleq \{\boldsymbol{x}_t^{(i)} \in \mathbb{R}^p, \boldsymbol{y}_t^{(i)} \in \mathbb{R}^q; t = 1, ..., T; i = 1, ..., N\} \tag{1.27}$$

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

$$L_D(W, b) \triangleq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} L_t^i(W, b) \tag{1.28}$$

$L_t^i$ is an arbitrary loss function for the $i^{th}$ example 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 vecotor, at time $T$, and adds 0 for each other time step.

# Bibliography

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