3. Feed Forward Neural Networks

INTRODUCTION

As seen earlier, the MCP neuron represents an abstraction of the biological neuron to a simple input/output processing system. We have seen than the simplest MCP network uses step functions like the Heaviside or the signum. In the previous section, it has been reported that the activation of the MCP neuron can be extended to nonlinear sigmoidal functions like logistic or hyperbolic tangent models. There are three main advantages in using such activation functions:

1. avoiding the sensitivity of the output near zero;
2. exploiting the three-region shape;
3. harnessing the continuity of the derivative.

We have seen that the step functions suffer from noise on action potential when approaching the 0 value. Specifically, also very small shifts of the action potential of the neuron may lead to uncertainty in the output that may waving between 0 and 1. The smoothing feature of sigmoidal function overcome such an issue. Sigmoidal functions feature three different regions namely the saturation, the linear and the non-linear ones. Equivalently to the step function, the saturation allows to bound the output, emulating the neural biological concept of refractory period in the spike generation, reducing in principle the risk of uncontrolled grow of the signals within a NN which could lead to unstable behaviors. Then the linear range assures a strict proportionality between the action potential and the neuron output and therefore in such range the effect of the weight factors on the input variables is mapped unaltered into the output (constant sensitivity). In the non-linear region, the action potential is progressively and continuously smoothed up (decreasing/increasing sensitivity) to saturation avoiding discontinuity in the derivative reducing the sensibility of the response to the possible weight variations. As far as derivative is concerned, C^1 characteristic ensures the chance of using classical gradient-based optimization for training the network.

Likewise, it has been reported that single-layer perceptron may be evolved to multi-layer networks by forward linking together multiple MCPs, attaining the so called feed-forward neural network (FFNN). FFNNs (Fig. 1) with opportune neuron configuration and non-linear activation functions, have been proved to show general processing abilities, typical of complex non-linear systems, as for example the ability to classify input patterns into classes, to compress high dimensional input data into low dimensional output data, to approximate continuous functions to cite few. It is said that complex FFNNs show increasing emergent properties that are not easily predictable from the analysis of the behavior of a pool of neurons taken singularly. In 1986, Rumelhart evolved the Delta rule to train multi-layer FF networks based on the innovative concept of back-propagation of the error throughout the network, demonstrating a general learning ability.

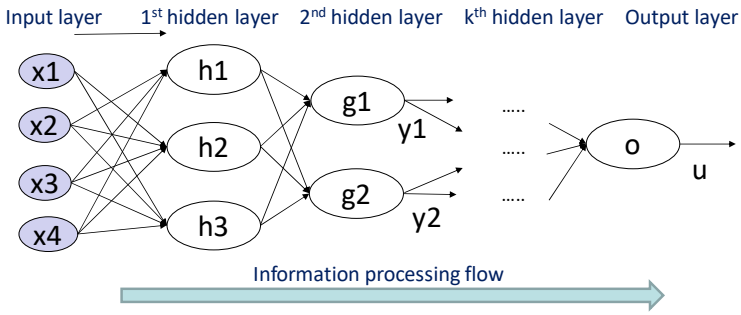


Figure 1. Feed-forward neural network.

APPLICATIONS OF MULTI-LAYER FEED-FORWARD ANN

ANN-based multiple label pattern classification of non-linearly separable distributions

Multi-label classication of non-linearly separable distributions is a complex problem that in general put traditional machine learning techniques to the test. Multi-layer ANN were demonstrated to show wide generality achievable by tailoring the number of hidden layer and the number of neurons. In general, multi-label output is allowed by setting the number of binary/sigmoidal neurons in the output so that with two

outputs, four different classes can be attained (Fig. 2). The geometrical consideration of using non-linear processing in the hidden layer consists of thinking to such a network as a way to bend/close decision boundaries to enclose patterns belonging to the same class.

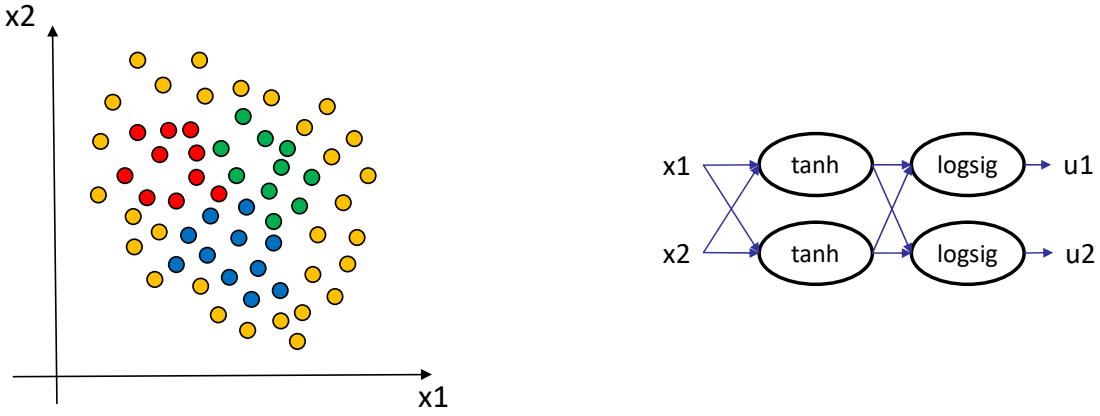


Figure 2. Example of complex distribution of (x1,x2) patterns. In principle, if opportunely trained, the depicted network might be able to solve the classification problem with high accuracy.

ANN-based function modeling

Another way of exploiting sigmoidal activation functions into a neural network is to reconstruct/approximate a function. In (Fig. 3), we report a three-layer feed-forward network that is mapping the input variable *x* into an output variable *u* by means of a linear combination of three logistic functions as:

$$u = v_1 * \log\mathrm{sig}(w_1 x - T_1) + v_2 * \log\mathrm{sig}(w_2 x - T_2) + v_3 * \log\mathrm{sig}(w_3 x - T_3) - T_4$$

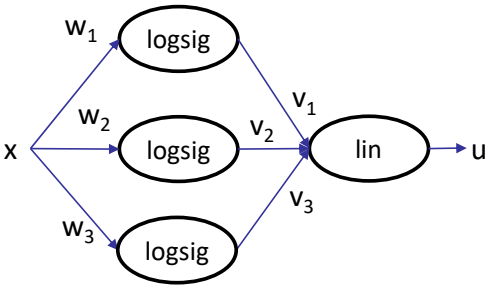
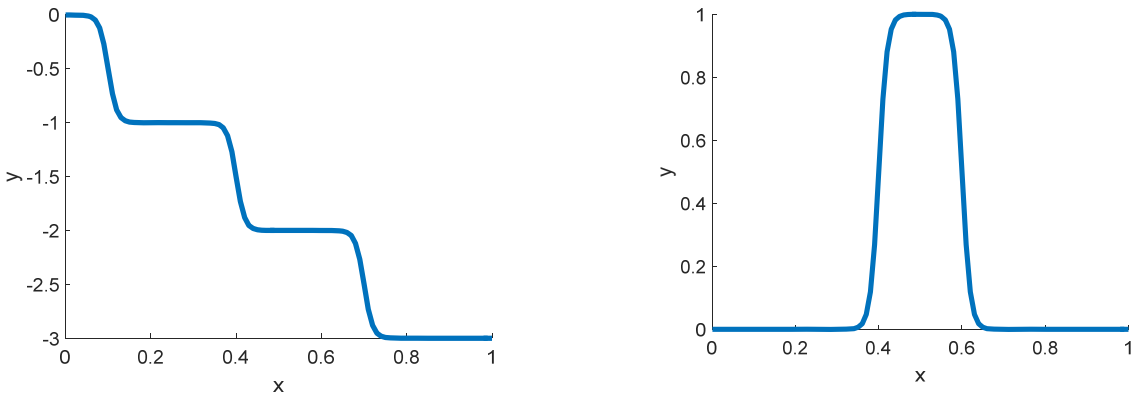


Figure 3. FFNN that models a function as a linear combination of 3 logistic functions.

where v_i and w_i are the weights of the output and the hidden neurons, respectively. T_i are the thresholds of the neurons. It is easily shown that by modifying the weights both in the hidden and in the output layers very different functions can be modeled. For instance by setting {w_1=-60, w_2=-60, w_3=-60, T_1=-10, T_2=-30, T_3=-50, v_1=1, v_2=1, v_3=1, T_4=1} we can obtain the function depicted in (Fig. 2a). Alternatively, by setting {w_1=20; w_2=20; w_3=20; T_1=20, T_2=60, v_1=1; v_2=-1; v_3=0}, the reconstructed function looks like a bell (Fig. 2b).



(a) (b)

Figure 3. Two functions modeled by the same FFNN in corespondence of two different parameter sets.

It can be demonstrated (principle of reconstruction universality) that, using a FFNN endowed with an arbitrary number of layers and neurons, featuring sigmoidal functions in the hidden layers and linear function in the output layer, you can approximate any function with an arbitrary resolution. This means that with a sufficient number of input/output pairs one can train the network to represent the target function in a continuous way.

ANN-based data processing

The use of linear-activation neurons in the network output discloses the opportunity the exploit the FF network as a tool to process data, namely applying linear and non-linear transforms to the inpout corresponding for instance to algebraic operations. Figure 4 reports two 1-D signals S1 and S2 evolving in time. The above network, by construction, processes one pair of two samples x1 and x2 at a time and produces in output two real values. One might want to compute sample by sample the product and sum of the two signals. Differently, the bottom network features an input pattern composed by a set of signal samples coming from S1 and S2, taken into a corresponding time window. In this case, the first *n* samples are taken from S1 signal and the next ones *n* are coming from S2. The processing result here consists into a single real value. For instance, one might aspire to compute time window by time window the norm of the difference between the two signals. Generalizing, according to the target task, one can define the optimal network topology to cope with the specific input/output transform.

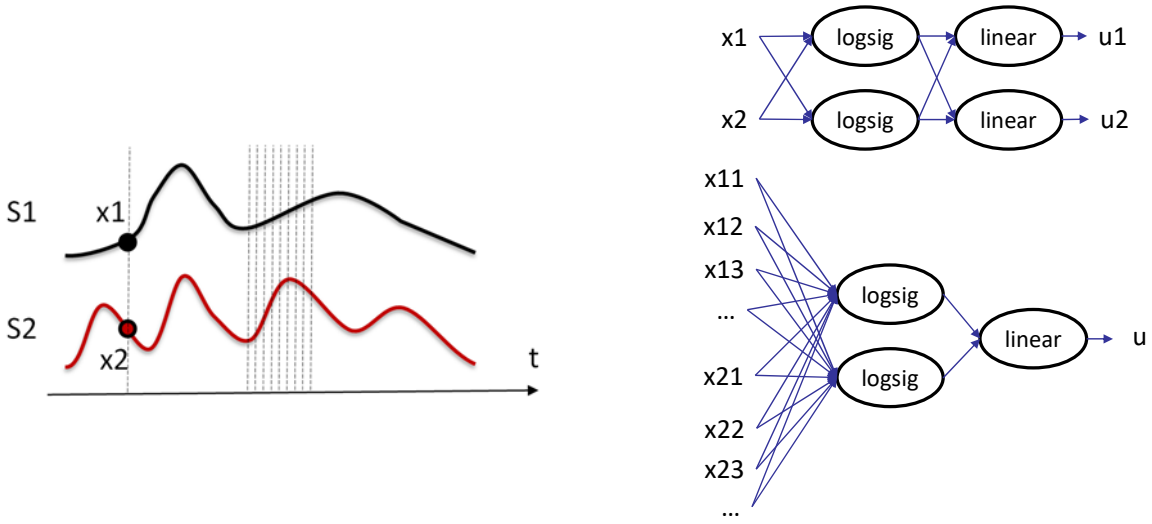


Figure 4. Signal processing performed through multi-layer FF networks.

COST/LOSS FUNTIONS

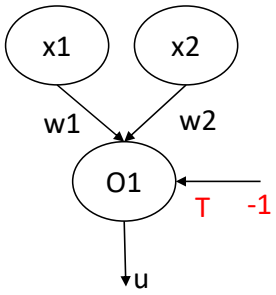
We have seen that the training process corresponds to an optimization of the network weights, starting from an initial guess. The optimization is driven by the cost function that is computed in an iterative procedure. In general, the key point of any supervised training is that we aspire at minimizing the error between the nominal and the predicted output so that the cost function can be regarded as a way to model the prediction error. In the domain of ANN, different cost functions can be used according to the problem tackled by the network. For regression problems, the best cost functions are the mean absolute, the mean squared and the logarithmic mean squared errors as follows:

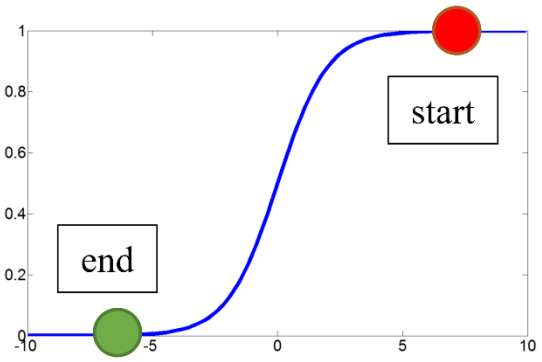
- MAE = $\sum(1/n*|t_i-u_i|)$
- RMSE = $\sqrt{(1/n * \sum (t_i-u_i)^2)}$
- RMSEL = $\sqrt{(1/n* \sum(\log(t_i+1)-\log(u_i+1))^2)}$

The most used is the RMSE while RMSEL is used when the effect of larger errors must be weighted less.

- Binary Cross-Entropy (usually for binary classification problems)
  - bCE = - (t*log(u) + (1-t)*log((1-u))
- Categorical Cross-Entropy (Usually for multi-class classification problems)
  - cCE = - Σ (t_i*log(u_i))

We have seen that the Delta rule with RMSE cost function problem makes the weight increment Δw dependent on the derivative of the activation function. When using logistic activation functions in the output for solving a binary classification makes the convergence speed strongly dependent on the starting point.





The above case shows that the initial guess of the weights predict an output u very far away (we expect to have 0 but we are starting from 1). In this case Delta rule being dependent on the derivative (almost 0) will move away very slow. We know that the learning rate acts only to shorten or enlarge the step (modulation of the gradient of the cost/loss function. Enlarging the learning rate may speed up the learning in general but, in the specific case, its effect may be ineffective even for very large learning rate as the derivative is almost 0. Let us now take into account the binary cross entropy bCE = - (t*log(u) + (1-t)*log(1-u)) and develop the Delta rule for that cost function.

It is easy to compute that d(bCE)/dw does not depends on the derivative if the output is a logistic function as:

$$-\left(\frac{d(t*\log(u))}{dw}+\frac{d((1-t)*\log(1-u))}{dw}\right)$$

Now applying traditional chaining

$$\frac{d(t*\log(u))}{du}*\frac{du}{dw}+\frac{d((1-t)*\log(1-u))}{du}*\frac{du}{dw}$$

$$\frac{t}{u}*\frac{du}{dw}-\frac{(1-t)}{(1-u)}*\frac{du}{dw}=\left(\frac{t}{u}-\frac{(1-t)}{(1-u)}\right)*\frac{du}{dw}=$$

$$\left(\frac{t}{u}-\frac{(1-t)}{(1-u)}\right)*\frac{du}{dP}*\frac{dP}{dw}=\left(\frac{t}{u}-\frac{(1-t)}{(1-u)}\right)*u'*x=\left(\frac{t(1-u)-u(1-t)}{u*(1-u)}\right)*u'*x=$$

$$\left(\frac{t-tu-u+tu}{u*(1-u)}\right)*u'*x=\left(\frac{t-u}{u*(1-u)}\right)*u'*x=\left(\frac{t-u}{u*(1-u)}\right)*u*(1-u)*x=$$

$$t-u*x$$

considering the initial minus sign then

$$\frac{dbCE}{dw}=-(t-u)*x=(u-t)*x$$

So it does not depends on the derivative of the activation function

It tells us that the rate at which the weight learns is controlled by (u-t), i.e., by the error in the output. The larger the error, the faster the neuron will learn.

TRAINING A MULTILAYER FFNN

As earlier stated, Delta rule can be applied to Perceptron networks with no hidden layers. We don't have basically any reference about the output of the neurons in the hidden layers so that we cannot directly compute the error signal for those neurons. Multi-layer Perceptron were demonstrated to overcome limitations intrinsic in the simple Perceptron thus different training methodologies were investigated in the past. In 1986, Rumelhart, Hinton, Williams (article in Nature magazine) proposed a generalization of the delta rule called back-propagation. Back-propagation, namely backward propagation of errors, is based on the propagation of the error signals from the output layer back to hidden layers by using the chain rule to iteratively compute gradients for each layer. Training through back-propagation is again an iterative supervised learning process requiring a training set of patterns along with the corresponding desired output set. The algorithm is basically divided into two different mechanisms working sequentially at each iteration step. The first mechanism, called direct computation, involves the computation of the neuron output, layer by layer, until a result is generated by the output layer. The actual output of the network is compared to expected output for that particular input as to attain error signal. The second mechanism involves the update of the weights layer by layer working backwards from the output layer, through the hidden layer, and to the input layer. This process is repeated until the correct output is produced. Fine tuning the weights in this way has the effect of teaching the network to produce the correct output for a particular input, i.e. the network *learns*. The main issue with updating the weights of the neurons in the hidden layers is that the supervisor does not provide the desired output signal for such neurons. The solution in the back-propagation is to solve the differential of the error function with respect the weights of hidden neurons in function of the weights updated in the next layer by chaining the differential.

BACK-PROPAGATION ALGORITHM

In order the shed light on this algorithm, let us consider a multi-layer FF network with 4 inputs, one hidden layer composed of 3 neurons and one output layer composed of 2 neurons (Fig. 5). The Delta rule can be applied both to weights w_{ij} and c_{jr} for the output and hidden neurons, respectively. The desired signal set $\{t\}$ of the neurons in the output layer is known by supervision. The delta rule can be applied to compute the weight increment Δw_{ij} for the neurons of the output layer. Technically, we can make use of the delta rule to compute the weight increment Δc_{jr} for the neurons in the hidden layer but the desired signal $\tilde{y}$ of the neurons in the hidden layer is unknown. The backpropagation algorithm allows to overcome such a shortcoming.

$$\Delta w_{ij} = \eta \sum_{k=1}^{R} \left(t_i^{(k)} - u_i^{(k)} \right) f'\left(P_i^{(k)} \right) y_j^{(k)}$$

$$\Delta c_{jr} = \eta \sum_{k=1}^{R} \left(\tilde{y}_j^{(k)} - y_j^{(k)} \right) f'\left(P_j^{(k)} \right) x_r^{(k)}$$

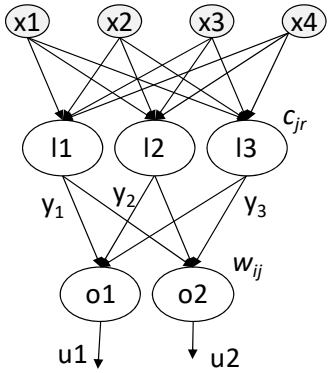


Figure 5. Multi-layer FF network with 4 input, one hidden layer composed of 3 neurons and one output layer composed of 2 neurons. The Delta rule can be applied both to weights w_{ij} and c_{jr} for the output and hidden neurons, respectively.

Let us consider a single pattern in the training set. According to the gradient descent principle, the increment of the weight c_{jr} of hidden neuron *j* with respect to the input *r* is linearly dependent of the differential of the error function with respect the weight itself as:

$$\Delta c_{jr} = f\left(\frac{\partial E}{\partial c_{jr}}\right)$$

By chaining the differential with the factor ∂P_j^H, namely the differential of the action potential of the hidden neuron j (H superscript stands for "hidden"), it can be expressed as:

$$\frac{\partial E}{\partial c_{jr}} = \frac{\partial E}{\partial P_j^H}\frac{\partial P_j^H}{\partial c_{jr}} =$$

which can be further chained to the factor ∂y_j (output signal of the hidden neuron j) as:

$$\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial P_j^H}\frac{\partial P_j^H}{\partial c_{jr}} =$$

Let now consider the term $\frac{\partial E}{\partial y_j}$. This term can be chained to ∂P_i^O which represent the differential of the action potential of the neuron i in the output layer. However, chaining ∂P_i^O makes Δc_{jr} dependent on i which is undue. This is solved by letting i varying over all the *N* neurons (*N*=2 in this specific case) in the output layer so that

$$\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial P_j^H}\frac{\partial P_j^H}{\partial c_{jr}} = \sum_i^N\left(\frac{\partial E}{\partial P_i^O}\frac{\partial P_i^O}{\partial y_j}\right)\frac{\partial y_j}{\partial P_j^H}\frac{\partial P_j^H}{\partial c_{jr}} =$$

Considering that signal y_j is an input signal to neuron i in the output layer, we can further chain ∂u_i (differential of the output signal of the neuron i in the output layer) obtaining:

$$\sum_{i=1}^N\left(\frac{\partial E}{\partial u_i}\frac{\partial u_i}{\partial P_i^O}\frac{\partial P_i^O}{\partial y_j}\right)\frac{\partial y_j}{\partial P_j^H}\frac{\partial P_j^H}{\partial c_{jr}} =$$

which can be simplified in:

$$\sum_{i=1}^N\left((t_i - u_i)f'\left(P_i^O\right)w_{ij}\right)\frac{\partial y_j}{\partial P_j^H}\frac{\partial P_j^H}{\partial c_{jr}} =$$

and in conclusion the final expression reads as:

$$\sum_{i=1}^N\left((t_i - u_i)f'\left(P_i^O\right)w_{ij}\right)f'\left(P_j^H\right)x_r$$

The training performed for all patterns R leads to define the increment Δc_{jr} as:

$$\Delta c_{jr} = \eta \sum_{k=1}^{R} \left(\bar{y}_j^{(k)} - y_j^{(k)} \right) f'\left(P_j^{H(k)}\right) x_r^{(k)} = \eta \sum_{k=1}^{R} \left(\sum_{i=1}^{N} \left(\left(t_i^{(k)} - u_i^{(k)} \right) f'\left(P_i^{O(k)}\right) w_{ij} \right) f'\left(P_j^{H(k)}\right) x_r^{(k)} \right)$$

The back-propagation can be generalized to any number of hidden layers. Let assume we have a multi-layer FFNN with *L* hidden layers, each layer having M_l neurons, and an output layer with N neurons. Let $w_{ij}^{(l)}$ be the generic weight of the *i*-th neuron in the layer *l* with respect to the output of the neuron *j* in the layer *l*-1. The factor $\Delta w_{ij}^{(l)}$ can be written then as:

$$\Delta w_{ij}^{(l)} = \eta \sum_{k=1}^{R} \delta_i^{(l),(k)} y_j^{(l-1),(k)}$$

where $y_j^{(l-1),(k)}$ is the output signal of the *j*-th neuron of the (*l*-1)-th layer in correspondence of the *k*-th pattern of the training set with:

$$\begin{aligned} \delta_i^{(l),(k)} &= \left(t_i^{(k)} - u_i^{(k)} \right) f'\left(P_i^{(k)} \right) && \text{if } \; l = L \\ \delta_i^{(l),(k)} &= \textstyle\sum_{r=1}^{M_{l+1}} \left(\delta_r^{(l+1),(k)} w_{ri}^{(l+1)} \right) f'\left(P_i^{(k)} \right) && \text{if } \; l < L \end{aligned}$$

(M_{l+1} is the number of neurons in the (*l*+1)-th layer). The overall procedure for a single iteration step can be then synthesized as follows:

1. Start from *l*=*L* (output layer) and compute $\delta_i^{(L),(k)}$ for each neuron i
2. Move to *l*=*L*-1 and compute $\delta_i^{(L-1),(k)}$ in function of $\delta_i^{(L),(k)}$
3. Continue $\delta_i^{(l),(k)}$ until *l*=2
4. Update all the weights

$$\Delta w_{ij}^{(l)} = \eta \sum_{k=1}^{R} \delta_i^{(l),(k)} y_j^{(l-1),(k)}$$

which determines the weight updates in the actual iteration step of the training procedure.

BACKPROGATION BOTTLENECK

From earlier considerations, the idea of using a multi-layer FFNN, endowed with several hidden layers, to learn a distributed representation of the input seems plausible in principle. In the supervised learning approach, the backpropagation of the output error, layer-by-layer backward up to the initial layer, is nominally the optimal strategy to overcome the limitation of the Delta rule. However, this leaning process was demonstrated to suffer from the so called “vanishing gradient problem”. This issue arises intrinsically in the backpropagation mechanism because of the progressive backward error correction layer-by-layer, which updates the weights starting from the last layer of the network. Two main effects are evident: 1) early hidden layers learn much more slowly than later hidden layers; 2) weights in the early hidden layers can even undergo erratic updating. This leads to expect that early layers can either become useless or even lead to instability of the learning.

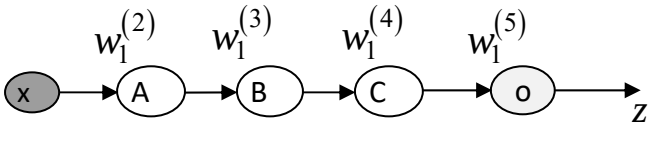


Figure 6. Simple multi-layer FFNN with 5 layers.

To get insight into why the vanishing gradient problem occurs, let us look at a simple multi-layer network with one input, three hidden layers (one neuron each) and an output neuron (Fig. 7). Here $w_1^{(k)}$ are the weights, *k* the layer. Let us remember that the output $y^{(k)}$ of the neuron is computed as:

$$y^{(k)} = \varphi\left(w_1^{(k)} y^{(k-1)} - T^{(k)}\right)$$

where φ and $T^{(k)}$ are the sigmoidal activation function and the neuron threshold, respectively. Let us try to analyze for instance the gradient $\frac{\partial E}{\partial T^{(2)}}$ associated to the first hidden neuron A, with E and $T^{(2)}$ the error function of the learning and the threshold of the neuron A, respectively (remember that by convention the first hidden layer is the layer of index 2 in the nework). Applying backpropagation leads to write:

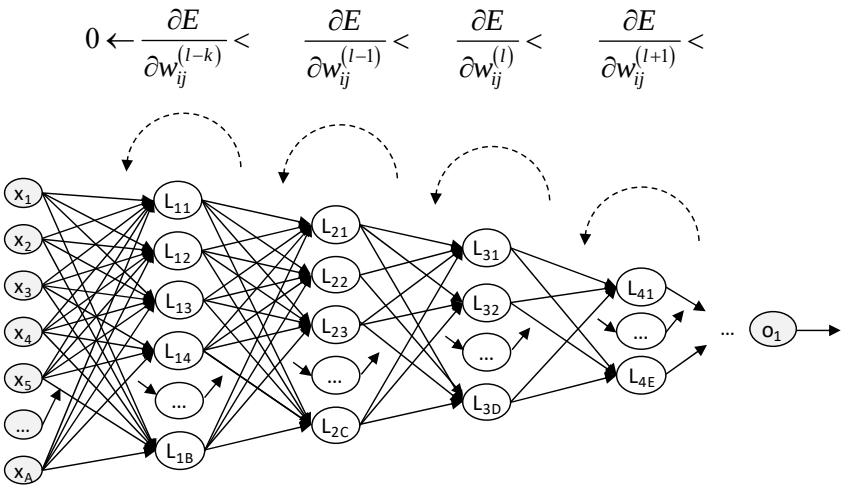


Figure 7. Example of multi-layer FFNN where the neurons in the hidden layers decrease in number as the depth in the network increases.

$$\frac{\partial E}{\partial T^{(2)}} = \varphi'\left(w_1^{(2)} x - T^{(2)}\right) \times w_1^{(3)} \times \varphi'\left(w_1^{(3)} y^{(2)} - T^{(3)}\right) \times w_1^{(4)} \times \varphi'\left(w_1^{(4)} y^{(3)} - T^{(4)}\right) \times w_1^{(5)} \times \varphi'\left(w_1^{(5)} y^{(4)} - T^{(5)}\right) \times \frac{\partial E}{\partial z}$$

The structure in the above expression is as follows: there is a φ' term in the product for each neuron in the network, a w_1 term for each weight in the network and the $\partial E / \partial z$ term, corresponding to the error function gradient, at the end. Considering that the derivative of the sigmoid function $\varphi'(x)$ reaches its maximum 0.25 at *x*=0 (Fig. 8) and assuming that the weights are initialized at small values ($|w_i| < 1$), we can expect that

$$\left|\varphi'\left(w_1^{(k)} y^{(k-1)} - T^{(k)}\right) \times w_1^{(k)}\right| < 1/4$$

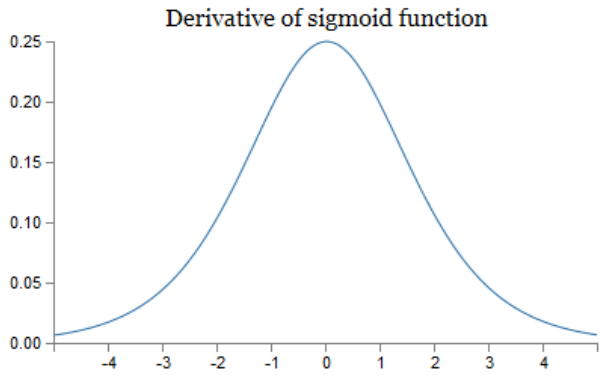


Figure 8. Derivative of the sigmoid function.

When we compute a product of many such terms, the product will have a tendency to exponentially decrease: the more terms, the smaller the product will be. Therefore, it is plausible that:

$$\frac{\partial E}{\partial T^{(2)}} = \underset{<1/4}{\varphi'\left(w_1^{(2)} x - T^{(2)}\right)} \times w_1^{(3)} \times \underset{<1/4}{\varphi'\left(w_1^{(3)} y^{(2)} - T^{(3)}\right)} \times w_1^{(4)} \times \underset{<1/4}{\varphi'\left(w_1^{(4)} y^{(3)} - T^{(4)}\right)} \times w_1^{(5)} \times \underset{<1/4}{\varphi'\left(w_1^{(5)} y^{(4)} - T^{(5)}\right)} \times \frac{\partial E}{\partial z}$$

stating that 4 layers beforehand the last layer in the network will get a weight update smaller by a factor of 16 (or more) than the weight update at the last layers. This option is under the hypothesis that $|w_i| < 1$ during learning. However, we know that whether the weights greatly overcome 1 then we will no longer

have a vanishing gradient problem but rather the exploding gradient problem (instability of the backpropagation). However, the fundamental problem here is not so much the vanishing gradient problem or the exploding gradient problem. It is that the gradient in early layers is the product of terms from all the later layers. When there are many layers, we can easily expect an unstable outcome of the learning process. The only way all layers can learn at close to the same speed is whether all those products of terms come close to balancing out. Without some mechanism or underlying reason for that balancing to occur, it is highly questionable to happen simply by chance. In short, the real problem here is that multilayer FFNN potentially suffer from an *unstable gradient problem*. In conclusion, if we use standard gradient-based learning techniques, different layers in the network will tend to learn at largely different speeds and degrees and especially earlier layers might not learn anything at all. We will see in the next three ways to overcome vanishing gradient problem:

1. removing fully connection exploiting local receptive field and convolutional neural processing
2. uses autoencoder units;
3. automatic weight control of neurons that are saturating during training by means of dropout.

MULTI-LAYER FFNN: FILNAL REMARKS AND TRAINING ISSUES

Some key elements are to be discussed to remark the issues of feed-forward multi-layer ANNs. First, one has to put the attention to the topology of the network. Any specific task (classification, mapping relations, function approximation, ..) has to carefully understood to select an opportune number of hidden layer and neurons. The performance of the network can dramatically modify even for small changes in such numbers. The network topology impacts the representation and processing of the information within the network. In principle, fully interconnected networks imply that all the neurons, layer-by-layer, process all the input coming from the previous layer so that the network internally does not represent locally specific subspaces of the input patterns. In contrast, the network can be configured to map a subspace of the input into a subset of the neurons or even map one single input element to a subset of neurons in the hidden layers. One can say that the information representation is super-distributed and distributed, respectively. A network features locality when a single neuron into in the hidden layers is connected to a subset of input neurons.

Second, the selection of the activation functions can impact on the overall network performance. When dealing with classification tasks for instance, the use of linear activation functions allows only to define linear decision boundaries preventing general classification properties (the network can even behave like a single layer Perceptron). Non-linearity of the activation function and hidden layers can model non-linear decision boundaries. However, the use of non-linear activation functions makes the error function E in the supervised training extremely complex generating an irregular surface error with potentially many local minima. This complicates the convergence and the opportune selection of learning rate η can be crucial. However, small values of η imply good approximation but slow convergence. Large values of η imply fast convergence but possible either local minima or oscillations.

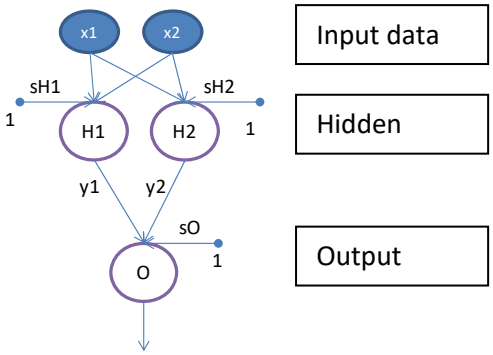
Third, the supervised training has to be carefully carried out by taking into account the selection of a consistent training set, the fitting quality, convergence and the stop criterion, and the extrapolation quality. As a main principle, the overall performance of the network will depend on the training set. For a classification task for instance, the patterns used to train the network should be representative of all the classes that have to be learned. Unbalancing the number of patterns belonging to a class with respect to the number of pattern into another class can lead to bias in the network. Differently, if one class of patterns is easy to learn, having a large number of patterns from that class in the training set will only slow down the overall learning process. For a function approximation task, the function domain is to be opportunely sampled to get a consistent description of the function co-domain. In general, systematic errors on the patterns of the training set can easily disrupt the learning. Another critical issue for the training is the fitting quality of the minimization. Under- and over-fitting can sensibly affect the network extrapolation performance. Under-fitting, meaning that the network is not enough complex to represent the information within the training patterns, can be dig by a increasing the number of hidden units. Alternatively, an error minimization check can constraint the optimization to be more adherent to the input data. Over-fitting, meaning that the network is too complex to represent the information within the training patterns, can be

dig by removing both hidden layers and neurons. Alternatively, a decrease of the fitting can be gained by superimposing random noise to the training patterns, preventing the exact learning of the original input data. Usually, the standard technique to decrease the fitting is to separate the training set into two subsets, namely the real training set, used to estimate the weight increment, and the validation set (10-20% of the overall dataset), used to check the prediction error which is used as an additional stop criterion. An additional set of patterns, called test set, is adopted to evaluated to extrapolation capability of the network after training. It is agreed that the network has good generalization capabilities if its performance on the test set is similar to that one obtained on the training set. However, small residual error on the training set does not guarantee good generalization properties. As we have previously mentioned, the use of the validation set to check during training is a valid countermeasure to avoid interpolation of the training set increasing extrapolation quality.

Appendix A

BUILDING A XOR BOOLEAN PORT USING A MULTI-LAYER PERCEPTRON

The following network represents a three-layer feed-forward network where the activation function of the neurons is a sigmoid.



Input (4 patterns)

Threshold bias	X1	X2	t
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Let assume
Neuron H1:
wH(1,1) = sH1 wH(1,2) = c11 wH(1,3) = c12

Neuron H2:
wH(2,1) = sH2 wH(2,2) = c21 wH(2,3) = c22

Neuron O:
wO(1) = sO wO(2) = w1 wO(3) = w2

PH1 action potential of the neuron H1
PH2 action potential of the neuron H2
PO action potential of the neuron O

Random initialization of the weights

%%Weights of the hidden layer
c(1,1) = 0.341232;
c(1,2) = 0.129952;
c(1,3) = -0.923123;

c(2,1) = -0.115223;
c(2,2) = 0.570345;
c(2,3) = -0.328932;

%%Weights of the output layer
w(1) = -0.993423;

w(2) = 0.164732;
w(3) = 0.752621;

%%learning rate
lrate = 0.25;

%%ITERATION 1
First pattern
Threshold Bias= 1
X1 = 0
X2 = 0
I=[Threshold Bias X1 X2]

Forward step
Compute all the neuron output

%%First layer (hidden)

Action potential
PH1 = c(1,1) * I(1) + c(1,2) * I(2) + c(1,3) * I(3) = 0.341232 * 1 + 0.129952 * 0 + -0.923123 * 0 =
Y1 = 1/(1 + exp(-PH1)) = 0.5845

PH2 = c(2,1) * I(1) + c(2,2) * I(2) + c(2,3) * I(3) = -0.115223* 1 + 0.570345* 0 + -0.328932* 0 =
Y2 = 1/(1 + exp(-PH2)) = 0.4712

%%Output layer
PO = w(1) * 1 + w(2) * Y1 + w(3) * Y2 = -0.993423 * 1 + 0.164732 * 0.5845 + 0.752621 * 0.4712 = -0.5425
U = 1/(1 + exp(-PO)) = 0.3676

Backward step

%%compute the signal error for the output neuron
errO = (t(1) – U) * f'(PO) = (t(1) – U) * U * (1 – U) = -0.3676 * 0.3676 * 0.6324 = -0.0855

%%compute the signal error for the hidden neurons
errH1 = Y1 * (1-Y1) * w(2) * errO = 0.5845 * 0.4155 * 0.164732 * -0.0855 = -0.0034
errH2 = Y2 * (1-Y2) * w(3) * errO = 0.4712 * 0.5288 * 0.752621 * -0.0855 = -0.0160

%%compute the deltaW
dW1 = lrate * 1 * errO = 0.25 * 1 * -0.0855 = -0.0214
dW2 = lrate * Y1 * errO = 0.25 * 0.5845 * -0.0855 = -0.0125
dW3 = lrate * Y2 * errO = 0.25 * 0.4712 * -0.0855 = -0.0101

%%Update W
w(1) = w(1) + dW1 = -1.0148
w(2) = w(2) + dW2 = 0.1522
w(3) = w(3) + dW3 = 0.7426

%%compute the deltaC

dC11 = lrate * I(1) * errH1 = -0.0009
dC12 = lrate * I(2) * errH1= 0
dC13 = lrate * I(3) * errH1= 0

dC21 = lrate * I(1) * errH2 = -0.0040
dC22 = lrate * I(2) * errH2 = 0
dC23 = lrate * I(3) * errH2 = 0

c(1,1) = c(1,1) + dC11 =0.341232 - -0.0009= 0.3404
c(1,2) = c(1,2) + dC12 =0.129952 + 0 = 0.129952
c(1,3) = c(1,3) + dC13 =-0.923123 + 0 = -0.923123

c(2,1) = c(2,1) + dC21 = -0.115223- 0.0040 = -0.1192
c(2,2) = c(2,2) + dC22 = 0.570345 + 0 = 0.570345
c(2,3) = c(2,3) + dC23 = -0.328932 + 0 = -0.328932

%%ITERATION 1
Second pattern

…..

%%ITERATION 2

…..

Stop after 10000 iteration

c(1,:) = 2.7028 5.5651 -5.3875
c(2,:) = -3.1941 5.8723 -6.0424

w = 3.9693 -8.4069 8.6434

Verification

pattern 0 0 -> u = 0.02 ok <0.5

pattern 1 0 -> u = 0.96 ok >0.5

pattern 0 1 -> u = 0.97 ok >0.5

pattern 1 1 -> u = 0.02 ok <0.5

```
%%Backpropagation
clear all
clc;
%% XOR with two layers
%% Two inputs
%% Hidden layer with two neurons
%%output layer with one neuron

%%table
In = 2; %%(number of inputs) %%(1 with the unit input for bias)
x = [1 0 0 ;1 0 1 ;1 1 0 ; 1 1 1;];
t = [ 0;  1;  1 ; 0;];
%% y output dell'hidden layer
%%Consider the threshold S as an additional weight corresponding to a unit
%%input

%%Weights of the hidden layer
Hn = 2;
c(1,1) = 0.341232; %%(2*rand -1);
c(1,2) = 0.129952; %%(2*rand -1);
c(1,3) = -0.923123; %%(2*rand -1);

c(2,1) = -0.115223; %%(2*rand -1);
c(2,2) = 0.570345; %%(2*rand -1);
c(2,3) = -0.328932; %%(2*rand -1);

%%Weights of the output layer
On = 1;
w(1) = -0.993423; %%(2*rand -1);
w(2) = 0.164732; %%(2*rand -1);
w(3) = 0.752621; %%(2*rand -1);

learning = 0.25;

for e = 1 : 10000

    for r = 1 : 4

       %% j = 1 x = [1(bias) 0 0]   t = 0
       %% j = 2 x = [1(bias) 1 0]   t = 1
       %% j = 3 x = [1(bias) 0 1]   t = 1
       %% j = 4 x = [1(bias) 1 1]   t = 0

       %%FORWARD STEP
       %%-------------------------
       %%Compute all the neuron output

            %%First layer (hidden)
              for j = 1 : Hn
                    Ph(j) = c(j,:) * [x(r,:)]';
                    y(j) = 1/(1 + exp(-Ph(j)));
              end
            %%..

            %%Last layer  (output) (On = 1)
              for j = 1 : On
                  Po(j) = w(j,:) * [1 y(j,:)]';
                  u(j) = 1/(1 + exp(-Po(j)));
              end

       %%-------------------------

       %%BACKWARD STEP
       %%-------------------------
       %%From output layer up to the first one

            %%Compute signal error (Delta rule)
            for j = 1 : On
                errO(j) = (t(r)-u(j)) * u(j) * (1 - u(j));
                %% Propagate the error backwards
                %%For each neuron in the hidden layer
                errH = zeros(Hn,1);
                for k = 1 : Hn
                    errH(k) = errH(k) + y(k) * (1 - y(k)) *  w(j,k+1) *  errO(j);
                end
            end

            %%Update the weights layer by layer
```

```
            %%-------------------------
            %%Output layer
            %%For each neuron in the output layer
            for  j = 1 : On
                %%Bias
                dw(j,1) = learning * 1 * errO(j);
                w(j,1) = w(j,1) + dw(j,1);
                for k = 1 : Hn
                    dw(j,k+1) = learning * errO(j) * y(k);
                    w(j,k+1) = w(j,k+1) + dw(j,k+1);
                end
            end
            %%Output layer
            %%------------------------- (end)

            %%-------------------------
            %% Hidden layer c(j,k)
            %%For each neuron in the hidden layer
            for j = 1 : Hn

                %%Bias
                dc(j,1) = learning * 1 * errH(j);
                c(j,1) = c(j,1) + dc(j,1);

                for k = 1 : In
                    dc(j,k+1) = learning * x(r,k+1) * errH(j);
                    c(j,k+1) = c(j,k+1) + dc(j,k+1)
                end

            end

       %%-------------------------

    end

     W(e,1) = w(1);
     W(e,2) = w(2);
     W(e,3) = w(3);


end


%                for  k = 1 : Hn
%
%                    dw(k) = err * y(k);
%                    w(k) = w(k) + dw(k);
%                end


%%Test
    for r = 1 : 4
              for j = 1 : Hn
                    Ph(j) = c(j,:) * [x(r,:)]';
                    y(j) = 1/(1 + exp(-Ph(j)));
              end
            %%..

            %%Last layer  (output)
              for j = 1 : On
                  Po(j) = w(j,:) * [1 y(j,:)]';
                  u(j) = 1/(1 + exp(-Po(j)))
              end
    end

figure;plot(W);
```