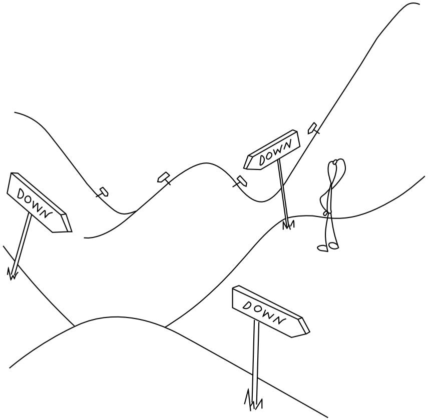
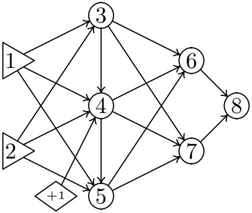
DEEP ARCHITECTURES

Marco Gori, Lisa Graziani



Any linear threshold units (LTU) can be regarded as vertexes of a graph which carries out a collective computation. If the neuron is a classic LTU the only consistent computational mechanism that can be constructed is based on Directed Acyclic Graph (DAG).

- A DAG $\mathcal{G}$ is a digraph that contains no oriented cycles.
- $\mathcal{G} = (\mathscr{V}, \mathscr{A})$, where $\mathscr{V}$ is the set of vertices and $\mathscr{A}$ is the set of arcs.

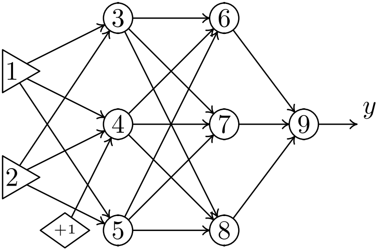


This data flow scheme can be expressed by the partially ordered set $\mathscr{S} = \{\{1, 2\}, \{3\}, \{4\}, \{5\}, \{6, 7\}, \{8\}\}$.

A *feedforward neural network* FFN is a DAG $\mathcal{G}$ with $\mathscr{V} = \mathscr{I} \cup \mathscr{H} \cup \mathscr{O}$, and the following computational structure:

$$x_p = v_p\,[p \in \mathscr{I}] + \sigma\left(\sum_{q \in \mathrm{pa}(p)} w_{pq} x_q + b_p\right)[p \in \mathscr{H} \cup \mathscr{O}].$$

- $\mathscr{I}$ is the input layer, $\mathscr{H}$ is the hidden layer and $\mathscr{O}$ is the output layer.
- p states a vertex.
- $w_{pq} \in \mathbb{R}$ is the weight attached to the arch $p \to q$.
- $b_p \in \mathbb{R}$ is the bias relative to p.
- $\mathrm{pa}(p) = \{q \in \mathscr{V} : q \to q \in \mathscr{A}\}$, is the set of the parents of p.
- The activation relative to the vertex p is $a_p = \sum_{q \in \mathrm{pa}(p)} w_{pq} x_q + b_p$.



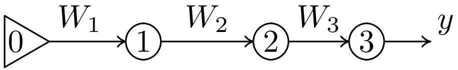
The neurons are organized into two *hidden layers* $(3, 4, 5)$ and $(6, 7, 8)$ and there is an output layer composed of neuron 9.

$$\mathscr{S} = \{\{1, 2\}, \{3, 4, 5\}, \{6, 7, 8\}, \{9\}\}.$$

Here, we have the total ordering $\{1, 2\} \prec \{3, 4, 5\} \prec \{6, 7, 8\} \prec \{9\}$, whereas there is no ordering inside the layers.

ARCHITECTURAL ISSUES

We can represent the previous multi-layered network in a compact way with layers and interconnection matrices.



W_l is the matrix associated with the pair of layers $l-1, l$.

The output is

$$y = \sigma(W_3\sigma(W_2\sigma(W_1 x))).$$

In general we have

$$x_l = \sigma(W_l x_{l-1})$$

with $x_0 := x$.

ARCHITECTURAL ISSUES

In case of linearity, a feedforward network of L layers collapses to a single layer. We have $\sigma(\cdot) := id(\cdot)$, therefore

$$y = \prod_{\ell=1}^{L} W_\ell x = Wx, \text{ where } W := \prod_{\ell=1}^{L} W_\ell.$$

But the computational collapsing of layers is a rare property. In general, there is no matrix W_3 such that

$$\sigma(W_2(\sigma(W_1(x))) = \sigma(W_3 x).$$

ARCHITECTURAL ISSUES

There are different kinds of neurons:

- *Ridge neurons* determine the output $y = g(w, b, x) = \sigma(w'x + b)$.
- *Radial basis function* neurons determine the output $y = g(w, b, x) = k(\|x - w\|/b)$, where k is usually a bell-shaped function.

REALIZATION OF BOOLEAN FUNCTIONS

We can realize cascade of LTU to represent boolean functions.
The *truth table* of the boolean function $f(x, y)$ is the sequence of the four values $f(0,0)f(0,1)f(1,0)f(1,1)$, where 1 corresponds to T and 0 to F.
Let us consider Heaviside linear-threshold units.

- AND function
  We want to realize the truth table 0001 by a linear-threshold function $x_1 \wedge x_2 = [w_1 x_1 + w_2 x_2 + b \geq 0]$.
  The solutions are the vectors $(w_1, w_2, b)' \in \mathbb{R}^3$ such that

  $$(b < 0) \wedge (w_2 + b < 0) \wedge (w_1 + b < 0) \wedge (w_1 + w_2 + b > 0).$$

  A possible solution is $(w_1, w_2, b) = (1, 1, -\frac{3}{2})$.
  The solution space $\mathscr{W}_\wedge$ is convex.

REALIZATION OF BOOLEAN FUNCTIONS

- OR function

  We want to realize the truth table 0111 by $[w_1 x_1 + w_2 x_2 + b \geq 0]$.

  The solutions are the vectors $(w_1, w_2, b)' \in \mathbb{R}^3$ such that

  $$(b < 0) \wedge (w_2 + b > 0) \wedge (w_1 + b > 0) \wedge (w_1 + w_2 + b > 0).$$

  A possible solution is $(w_1, w_2, b) = (1, 1, -\frac{1}{2})$.
  The solution space $\mathscr{W}_\vee$ is convex.

REALIZATION OF BOOLEAN FUNCTIONS

- XOR function

$$x_1 \oplus x_2 = \neg x_1 \wedge x_2 \vee x_1 \wedge \neg x_2.$$

Unlike the case of $\wedge$ and $\vee$, the set

$$\mathcal{L} = \{((0,0),0),((0,1),1),((1,0),1),((1,1),0)\}$$

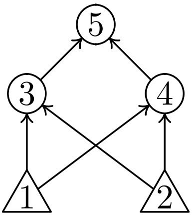
is not linearly separable. The equation

$$(b < 0) \wedge (w_2 + b) > 0 \wedge (w_1 + b) > 0 \wedge (w_1 + w_2 + b < 0)$$

has no solution ($\mathscr{W}_{\oplus} = \emptyset$).
We cannot compute the `XOR` function using a single LTU.

REALIZATION OF BOOLEAN FUNCTIONS

There are many ways to represent the XOR using a multilayered network.



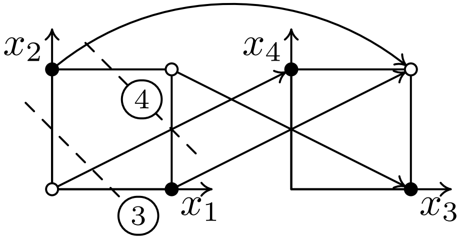
Input x_1 and x_2 must be mapped by the hidden layer to x_3 and x_4 such that it can be linearly separated by the neuron 5.

REALIZATION OF BOOLEAN FUNCTIONS

I) The inputs are mapped into a linearly separable configuration:

$x_3 = [x_1 + x_2 - 1/2 \geq 0]$

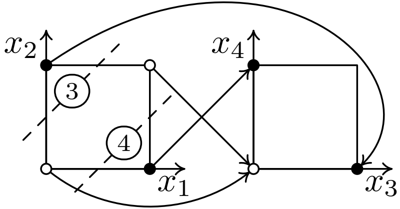
$x_4 = [-x_1 - x_2 + 3/2 \geq 0]$.



REALIZATION OF BOOLEAN FUNCTIONS

II) $x_1 \wedge x_2$ and $x_1 \wedge \neg x_2$ can be represented by LTU with the Heaviside function.
Units 3 and 4 detect $x_1 \wedge \neg x_2$ and $\neg x_1 \wedge x_2$, respectively, and then neuron 5 acts as an OR.

$$\neg x_1 \wedge x_2 = [-x_1 + x_2 - 1/2 \geq 0]$$
$$x_1 \wedge \neg x_2 = [x_1 - x_2 - 1/2 \geq 0]$$



REALIZATION OF BOOLEAN FUNCTIONS

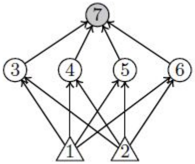
Any boolean function can be represented with two layers using the first canonical form

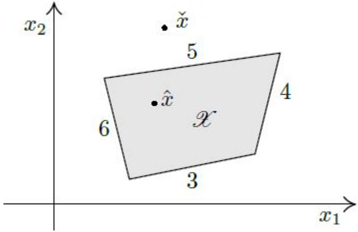
$$f(x) = \bigvee_{j=1}^{m} \bigwedge_{k=1}^{s_j} u_{jk} = (u_{11} \wedge \cdots \wedge u_{1s_1}) \vee \cdots \vee (u_{m1} \wedge \cdots \wedge u_{ms_m}),$$

where u_{ij} are literals, which means either a variable x_i or its complement.

REALIZATION OF REAL-VALUED FUNCTIONS

Real-valued functions can model both regression and classification problems.
Any neural network with one hidden layer and hard-limiting LTU characterizes convex domains.

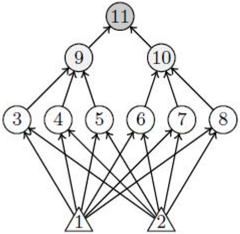


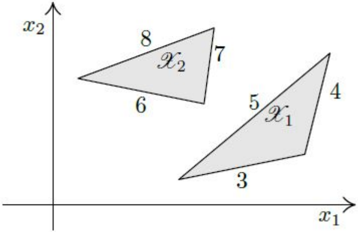


This is an example of classification in $\mathbb{R}^2$. The neural network with hard-limiting LTU returns $f(\mathscr{X}) = 1$. Each neuron in the hidden layer $(4, 5, 6, 7)$ is associated with a corresponding line, that define the convex domain $\mathscr{X}$.

REALIZATION OF REAL-VALUED FUNCTIONS

Through a neural network with two hidden layers we can characterize non-connected domains.

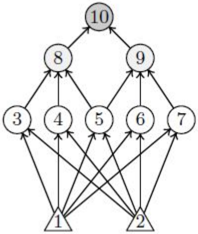


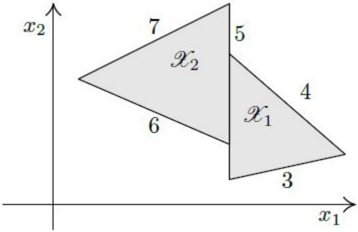


The neurons $4, 5, 6$ and $6, 7, 8$ characterize the convex sets $\mathscr{X}_1$ and $\mathscr{X}_2$, respectively. The neurons 9 and 10 establish if $x \in \mathscr{X}_1$ and $x \in \mathscr{X}_2$, respectively. Finally the output neuron 11 establishes if $x \in \mathscr{X} = \mathscr{X}_1 \bigcup \mathscr{X}_2$ through the `OR` operation.

REALIZATION OF REAL-VALUED FUNCTIONS

The construction shown for non-connected convex sets can be used to realize any concave set.
We can provide a partition of $\mathscr{X}$ by convex sets $\mathscr{X}_1$ and $\mathscr{X}_2$.





The neuron 5 participates to the construction of both the convex sets.

In general, given $\mathscr{X}$ concave we need to find a family of sets $\mathscr{F}_X = \{\mathscr{X}_i, \;\; i = 1, \ldots, m\}$ such that $\bigvee_{i=1}^{m} \mathscr{X}_i = \mathscr{X}$.

CONVOLUTIONAL NETWORKS

- Convolutional networks are mostly used in computer vision.
- They allow to extract *invariant features* and this is important when we consider spatiotemporal information.

Let $\mathscr{Z} \subset \mathbb{R}^2$ be the *retina*, where each pixel is identified by $z = (z_1, z_2)$. Let $v(z) \in \mathbb{R}^{\vdash m}$ be the brightness on pixel z, where $\vdash m = 1$ for black-white pictures and $\vdash m = 3$ for color pictures. We can take a compact representation of the contextual information associated with z:

$$y(z) := g(z, \cdot) * v(\cdot) = \int_{\mathscr{Z}} g(z, u) v(u) du, \tag{1}$$

where $g : \mathscr{Z}^2 \to \mathbb{R}^{m, \vdash m}$ is a *kernel-based filter*.

CONVOLUTIONAL NETWORKS

If $g(z,u) = h(z-u)$,

$$y(z) = \int_{\mathscr{Z}} h(z-u)v(u)du$$

is the *convolution* of filter $g(\cdot)$ with the video signal $v(\cdot)$.

- The convolution returns a feature vector $y(z) \in \mathbb{R}^m$ that depends on the pixel z on which we are focussing attention.
- Map $y(\cdot)$ represents contextual information, whereas map $v(\cdot)$ only expresses lighting properties of the single pixel, regardless of its neighbors.
- Convolution is associative and commutative.

We can use the eq. (1) to get a context in the processing of video streams. The video signal is represented by $v(t, z)$, where the retina domain $\mathscr{X}$ now becomes $\mathscr{V} = \mathscr{X} \times \mathscr{T}$, being $\mathscr{T} = [t_o, t_1]$ the temporal domain of the video.
If we define $\zeta := (t, z), \quad \mu := (\tau, u)$ then we have

$$y(\zeta) := g(\zeta, \cdot) * v(\cdot) = \int_{\mathscr{V}} g(\zeta, \mu) v(\mu) d\mu.$$

LEARNING IN FEEDFORWARD NETS

- Learning algorithms typically require to compute the gradient of the loss for any example v, that is ∇e, where $e(w, v, y) = V(y, f(w, v))$.
- The derivatives of a function can either be computed numerically or symbolically. For instance, if we want to compute $\sigma'(a)$, where $\sigma(a) = 1/(1 + e^{-a})$, the symbolic derivative is $\sigma'(a) = \sigma(a)(1 - \sigma(a))$.
- The numerical computation produces roundoff errors and is very expensive for high dimensional problems.

LEARNING IN FEEDFORWARD NETS

Forward propagation

Algorithm F *FORWARD*$(\mathcal{G}, w, m, v, x)$

- $\mathcal{N} = (\mathcal{G}, w)$ network based on the DAG $\mathcal{G}$ with weights w
- m vector used as a weight modifier
- v vector of inputs

For all $i \in \mathscr{V} \setminus \mathscr{I}$ it computes the state of vertex i and stores the value in the vector x_i (then in particular it computes the function function $f(w, v)$ that is specified by the values x_o with $o \in \mathscr{O}$).

$TOPSORT(\mathscr{S}, s)$ takes a set $\mathscr{S}$ and copies the elements of this set in the array s topologically sorted, so that for each i and j with $i < j$ we have $s_i \prec s_j$.

LEARNING IN FEEDFORWARD NETS

Forward propagation

F1. [Initialize] For all $i \in \mathscr{I}$ set $x_i \leftarrow v_i$ and initialize an integer variable $k \leftarrow 1$.

F2. [Topsort] Invoke TOPSORT on the set $\mathscr{V} \setminus \mathscr{I}$, so that the vector s contains the topological sorting of the nodes of the net. Set the variable l to the dimension of the vector s.

F3. [Finished yet?] If $k \leq l$ go on to step F4, otherwise the algorithm stops.

F4. [Compute the state x] If $m = (1, 1, \ldots, 1)$ set $x_{s_k} \leftarrow \sigma\big(\sum_{j \in \mathrm{pa}(s_k)} w_{s_k j} x_j\big)$ otherwise set $x_{s_k} \leftarrow m_{s_k} \sum_{j \in \mathrm{pa}(s_k)} w_{s_k j} x_j$. Increase k by one and go back to step F3.

LEARNING IN FEEDFORWARD NETS

- Numerical algorithms for the gradient computation are $\Theta(m^2)$, where m is the number of weights.
- FNNs are sometimes applied in problems where m is order of millions. The numerical computation of the gradient in those cases would require order of teraflops.
- Backpropagation is the best gradient computation algorithm: is $\Theta(m)$.
- We can write

$$\frac{\partial e}{\partial w} = \frac{\partial V}{\partial f} \cdot \frac{\partial f}{\partial w} = \sum_{o \in \mathscr{O}} \frac{\partial V}{\partial f_o} \frac{\partial f_o}{\partial w},$$

so whenever we are given a symbolic expression for $V(y, f(w, v))$, we can also give a corresponding symbolic expression to $\frac{\partial e}{\partial w}$.

LEARNING IN FEEDFORWARD NETS

Consider the derivative of $f_o(w, v) = x_o$ with respect to w_{ij}, and call this quantity g^o_{ij}; by using the chainrule, we get

$$g^o_{ij} = \frac{\partial x_o}{\partial w_{ij}} = \frac{\partial x_o}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial x_o}{\partial a_i}\frac{\partial}{\partial w_{ij}} \sum_{h \in \mathrm{pa}(i)} w_{ih} x_h = \delta^o_i x_j, \qquad (2)$$

where $\delta^o_i \equiv \partial x_o / \partial a_i$ is the **delta error**.

The delta error of an output neuron is

$$\delta^o_o = \sigma'(a_o). \qquad (3)$$

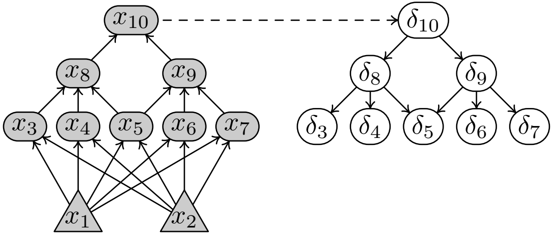
For example in the case of logistic function $\delta^o_o = x_o(1 - x_o)$.

By using the chain rule we have

$$\delta^o_i = \frac{\partial x_o}{\partial a_i} = \sum_{h \in \mathrm{ch}(i)} \frac{\partial x_o}{\partial a_h}\frac{\partial a_h}{\partial x_i}\frac{\partial x_i}{\partial a_i} = \sigma'(a_i) \sum_{h \in \mathrm{ch}(i)} w_{hi}\delta^o_h. \qquad (4)$$

LEARNING IN FEEDFORWARD NETS

Equations (3) and (4) allow us to determine δ_i^o by propagating backward the values δ_o^o throughout the hidden units $i \in \mathscr{H}$.



The backward step propagates recursively the delta error beginning from the output through its children. For example, $\delta_5 = \sigma'(a_5)(w_{85}\delta_8 + w_{95}\delta_9)$.

LEARNING IN FEEDFORWARD NETS

Now we want to calculate the derivative of the loss V with respect to the generic weight w_{ij}. We can follow the steps done in eq.(2):

$$\frac{\partial V}{\partial w_{ij}} = \frac{\partial V}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} = \delta_i x_j,$$

where $\delta_i = \partial V / \partial a_i$.

After the forward phase, we can immediately evaluate δ_o once we know the symbolic expression of V. For example for the quadratic loss $V(y, f) = \frac{1}{2}(y - f)^2$, $\delta_o = (y_o - \sigma(a_o))\sigma'(a_o)$.

Then we can recursively evaluate all the other δ_i using the analogous of eq.(4):

$$\delta_i = \sum_{h \in \mathrm{ch}(i)} \frac{\partial V}{\partial a_h}\frac{\partial a_h}{\partial x_i}\frac{\partial x_i}{\partial a_h} = \sigma'(a_i) \sum_{h \in \mathrm{ch}(i)} w_{hi}\delta_h.$$

Algorithm B *BACKWARD*$(\mathcal{G}, w, x, q, V)$

Algorithm that computes the derivatives either of the output or of the loss function of a general DAG with respect to the weights.

- $\mathcal{N} = (\mathcal{G}, w)$ network based on the DAG $\mathcal{G}$ with weights w
- x vector that contains the states of the vertices of $\mathcal{G}$
- q parameter
- V loss function

If $q > 0$ it returns the derivatives g_{ij}^{q}, otherwise returns the derivatives of the loss $\partial V / \partial w_{ij}$.

LEARNING IN FEEDFORWARD NETS

B1. [Loss or output?] If $q \leq 0$ go to step B2, otherwise jump to step B3.

B2. [Initialize for the loss] For all $o \in \mathscr{O}$ set $v_o \leftarrow \partial V / \partial a_o$ and go to step B4.

B3. [Initialize for x_q] For each $o \in \mathscr{O}$ if $o \neq q$ set $v_o \leftarrow 0$, otherwise $v_o \leftarrow \sigma'\big(\sum_{h \in \mathrm{pa}(o)} w_{oh} x_h\big)$.

B4. [Compute Backwards] For each $k \in \mathscr{V} \setminus \mathscr{I}$ set $m_k \leftarrow \sigma'\big(\sum_{h \in \mathrm{pa}(k)} w_{kh} x_h\big)$, then invoke $FORWARD((\mathcal{G} \setminus \mathscr{I})', w', m, v, \delta)$.

$(\mathcal{G} \setminus \mathscr{I})'$ is the graph obtained by reversing the direction of the arrows of $\mathcal{G}$ without the input nodes.

B5. [Output the gradient] For each $i \in \mathscr{V} \setminus \mathscr{I}$ and then for each $j \in \mathrm{pa}(i)$ set $g_{ij} \leftarrow \delta_i x_j$ and output g_{ij}. Terminate the algorithm.

Algorithm FB

Given a network $\mathcal{N} = (\mathcal{G}, w)$ based on the DAG $\mathcal{G}$, a vector of inputs v, and a loss function V, it returns the gradient of the loss with respect to w.

FB1. [Forward] Invoke $FORWARD(\mathcal{G}, w, (1, 1, \ldots, 1), v, x)$.

FB2. [Backward] Invoke $BACKWARD(\mathcal{G}, w, x, -1, V)$. Terminate the algorithm.

LEARNING IN FEEDFORWARD NETS

We can express the forward/backward equations using the tensor formalism.

Forward step:

$$\hat{X}_\ell = \sigma(\hat{X}_{\ell-1}\hat{W}_\ell), \quad \ell = 0, \dots, L. \tag{5}$$

If we have a structure with L layers

$$\hat{X}_L = \sigma(\dots\sigma(\sigma(\hat{X}_0\hat{W}_1)\hat{W}_2)\dots\hat{W}_L).$$

Backward step:

$$\Delta_{\ell-1} = \sigma' \odot (\Delta_\ell W_\ell) \tag{6}$$

$$G_\ell = \hat{X}'_{\ell-1}\Delta_\ell \tag{7}$$

where $\sigma' \in \mathbb{R}^{L,\ell-1}$ is the matrix with coordinates $\sigma'(a_{i,\kappa})$, $\odot$ is the Hadamard product, and $\Delta_\ell := (\delta_1, \dots, \delta_{n(\ell)}) \in \mathbb{R}^{\ell,n(\ell)}$, where $n(\ell)$ is the number of nodes in the layer ℓ.