Notice of the *toyneuralnetwork* toolbox

Benoit Merlet*

March 16, 2021

We present a python toolbox for solving simple geometric classification problems by training a Neural Network. It is designed for experimenting with optimization methods. The toolbox is made of three classes: the class `ToyPb` deals with the classification problem to solve, the class `nD_data` deals with the training and test data and eventually the class `ToyNN` deals with the neural network. For simplicity, we skip some options in this presentation.

Before coming to the description of these classes, we give a mathematical description of the problem and explain how is computed the gradient of the loss function with respect to the parameters of the neural net work.

1 Mathematical description

1.1 Neural Networks

We consider a fully connected neural network with $N+1$ layers numbered $n = 0, \dots, N$. Each layer has card_n nodes. The input layer has two nodes by default ($\text{card}_0 = 2$) but more generally $\text{card}_0 \geq 1$ is valid. For the two classes classification problem that we study, the output layer has only one node ($\text{card}_N = 1$). The nodes of the layer n are noted $N_0^{(n)}, N_1^{(n)}, \dots, N_{\text{card}_n}^{(n)}$.

For $n \in \{0, \dots, N-1\}$, each node of the layer n is connected to each node of the layer $n+1$. A weight $W_{j,k}^{(n)}$ is associated to the connection $N_j^{(n)}$—$N_k^{(n+1)}$, so for $n \in \{0, \dots, N-1\}$, $W^{(n)}$ is a $\text{card}_n \times \text{card}_{n+1}$ real matrix.

For $n \in \{0, \dots, N-1\}$, and $j \in \{0, \dots, \text{card}_{n+1}\}$ a real number $\text{Bias}_j^{(n)}$ is defined which is the bias associated to the node $N_j^{(n+1)}$. For $n \in \{0, \dots, N-1\}$, $\text{Bias}^{(n)}$ is an array of length card_{n+1}. See Figure 1.

*Univ. Lille, CNRS, UMR 8524, Inria - Laboratoire Paul Painlevé, F-59000 Lille, email: benoit.merlet@univ-lille.fr

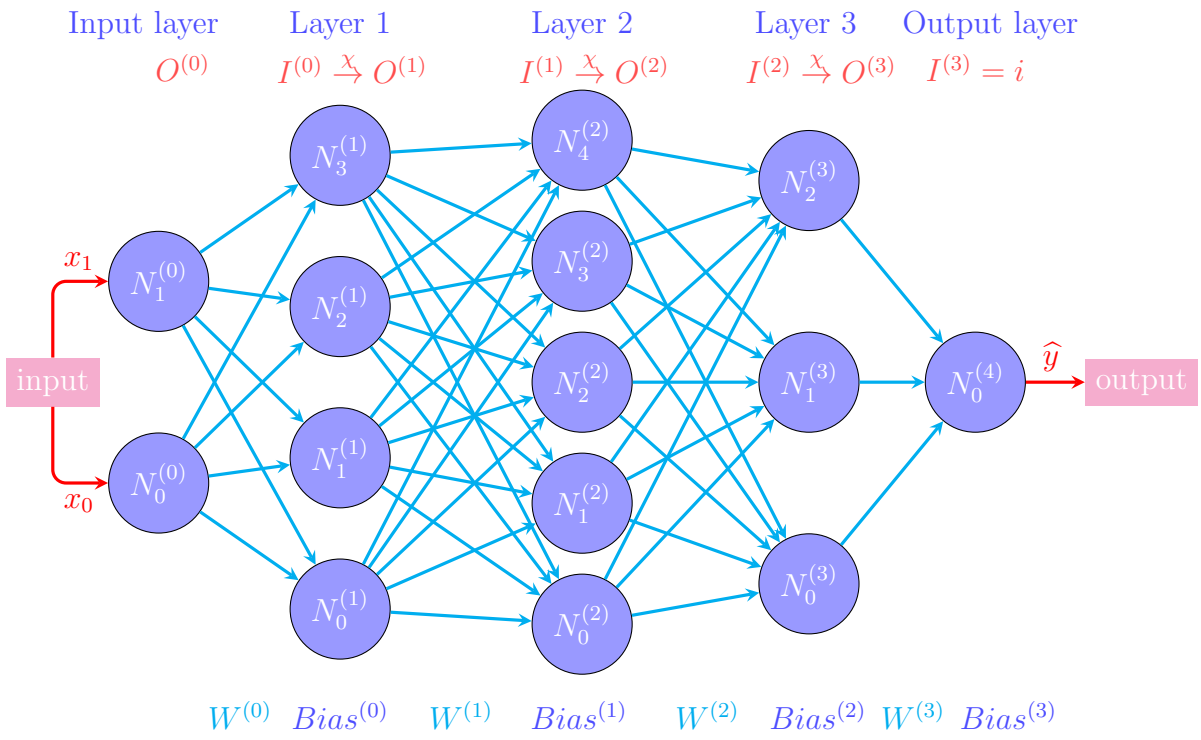


Figure 1: An example of Neural Network with $N = 4$ and card$= (2, 4, 5, 3, 1)$ (the input layer has two nodes, the output layer one node and the three hidden layers have four, five and three nodes respectively).

Eventually, an activation function $\chi : \mathbf{R} \to \mathbf{R}$ is given.

Given an input $X = (x_0, x_1, \ldots, x_{\text{card}_0 - 1})$, two values are associated to the nodes:

1. for $n \in \{1, \ldots, N\}$ and $k \in \{0, \ldots, \text{card}_{n-1} - 1\}$ an "input" value $I_k^{(n-1)}$ is associated to the node $N_k^{(n)}$;

2. for $n \in \{0, \ldots, N-1\}$ and $j \in \{0, \ldots, \text{card}_n - 1\}$ an "output" value $O_j^{(n)}$ is associated to the node $N_j^{(n)}$;

These quantities are defined as follows:

(0) First, the outputs at layer 0 are given by the components of the input X:

$$\text{for } j \in \{0, \ldots, \text{card}_0 - 1\}, \qquad O_j^{(0)} := x_j \,. \tag{1.1}$$

These quantities are transmitted to the next layer.

(1) The node $N_k^{(1)}$ of layer 1 receives the contribution $W^{(0)_{j,k}} O_j^{(0)}$ from each node j of layer 0. To form the input at node $N_k^{(1)}$, we sum these contributions and add the bias $\text{Bias}_k^{(0)}$. We obtain,

$$\text{for } k \in \{0, \ldots, \text{card}_1 - 1\}, \qquad I_k^{(0)} := \sum_{k=0}^{\text{card}_1 - 0} W_{j,k}^{(0)} O_j^{(0)} + \text{Bias}_k^{(0)}.$$

This can be written using linear algebra notation,

$$I^{(0)} = {W^{(0)}}^T O^{(0)} + \text{Bias}^{(0)}. \tag{1.2}$$

(2) The output at node $N_j^{(1)}$ is simply obtained by applying the activation function to the input at the same point: $O_j^{(1)} = \chi(I_j^{(0)})$. We note

$$O^{(1)} = \chi\left(I^{(0)}\right).$$

meaning that the function χ is applied to each component of the vector $I^{(0)}$. The process is then repeated from one layer to the next:

$$\text{for } n = 1, \dots, N-1, \quad O^{(n)} := \chi\left(I^{(n-1)}\right), \qquad I^{(n)} = {W^{(n)}}^T O^{(n)} + \text{Bias}^{(n)}. \tag{1.3}$$

The last computed value $i := I_0^{(N-1)}$ is the output produced by the neural network fed with the data X. We will note

$$\widehat{f}(X; \text{NN}) := \widehat{f}(X; W, \text{Bias}) := i. \tag{1.4}$$

When solving the classification problem, we want to minimize the quantity

$$E_{tr}(\text{NN}) := E_{tr}(W, \text{Bias}) := \sum_{j=0}^{n_{tr}-1} \ell(y_i \widehat{f}(X_i; W, \text{Bias})).$$

In the optimization process, we need to compute the gradient of E_{tr} with respect to the parameters W and Bias.

1.2 Computing the gradient of the loss function

We describe below how we can use the chain rule to obtain a practical algorithm for computing the partial derivatives of $\widehat{f}(X; W, \text{Bias})$ with respect to the entries of W and of Bias. Notice that the formulas (1.1)(1.2) and (1.3) provide an algorithm for the computation of the quantities $O_j^{(n)}$, $I_k^{(n)}$. This algorithm is *forward* in the sense that the quantities associated to the layer $n+1$ are computed from the quantities at layer n. The algorithm for the computation of the gradient is instead *backward*. We first introduce some notation before coming to its description.

The last layer contains only one node and we recall that

$$i = I_0^{(N-1)} = \widehat{f}(W, \text{Bias}; X),$$

is the "input" for this node. For $n \in \{0, \dots, N-1\}$, we note $\underset{W^{(n)}}{\nabla} i$ and $\underset{\text{Bias}^{(n)}}{\nabla} i$ the gradient of $\widehat{f}(X; \cdot)$ with respect to $W^{(n)}$ and Bias(n) all the other components (in $W^{(n')}$, Bias(n') for $n' \neq n$) being fixed. More precisely, for $0 \leq j \leq \text{card}_n - 1$, $0 \leq k \leq \text{card}_{n+1}$,

$$\left[\underset{W^{(n)}}{\nabla} i\right]_{j,k} := \frac{\partial \widehat{f}}{\partial W_{j,k}^{(n)}}(W, \text{Bias}; X),$$

and for $0 \leq k \leq \text{card}_{n+1}$,

$$\left[\underset{\text{Bias}^{(n)}}{\nabla} i\right]_k := \frac{\partial \widehat{f}}{\partial \text{Bias}_k^{(n)}}(f(W, \text{Bias}; X).$$

According to the forward algorithm, if the vector $I^{(n)}$ is known for some $n \in \{0, \cdots, N-1\}$ together with the parameters of the neural network, we can compute recursively $O^{(n+1)}$, $I^{(n+1)}$, $O^{(n+2)}, \ldots, O^{(N-1)}$ and eventually, $i = I_0^{(N-1)}$. From a mathematical point of view, this means that there exists a function $G_I^{(n)}$ such that

$$i = G_I^{(n)}(I^{(n)}, W, \text{Bias}).$$

We note $\underset{I^{(n)}}{\nabla} i$ its gradient with respect to the components of $I(n)$.

Similarly, if $O^{(n)}$, W and Bias are known, we can compute i. We write $i = G_O^{(n)}(O^{(n)}, W, \text{Bias})$ and note $\underset{O^{(n)}}{\nabla} i$ the gradient of $G_O^{(n)}$ with respect to $O(n)$

We are now ready to derive the backward algorithm.

(0) We obviously have

$$\frac{\partial\, i}{\partial\, I_0^{(N-1)}} = 1. \tag{1.5}$$

(1) According to the forward algorithm, i is completely determined by $O^{(N-1)}$, $W^{(N-1)}$ and $\text{Bias}^{(N-1)}$ and we can write $i = G^{(N-1)}(O^{(N-1)}, W^{(n)}, \text{Bias}^{(N)})$. We have by the chain rule formula,

$$\frac{\partial\, i}{\partial \text{Bias}_0^{(N-1)}} = \frac{\partial G^{(N-1)}}{\partial \text{Bias}_0^{(N-1)}}, \qquad \underset{W^{(N-1)}}{\nabla} i = \underset{W^{(N-1)}}{\nabla} G^{(N-1)}, \qquad \underset{O^{(N-1)}}{\nabla} i = \underset{O^{(N-1)}}{\nabla} G^{(N-1)}.$$

Next, the formula for $G^{(N-1)}$ in the forward algorithm is

$$G^{(N-1)}(I_0^{(N)}) = \sum_{j=0}^{\text{card}_{N-1}} W_{j,0}^{(N-1)} O_j^{(N-1)} + \text{Bias}_0^{(N-1)}.$$

Differentiating, we deduce,

$$\frac{\partial G^{(N-1)}}{\partial \text{Bias}_0^{(N-1)}} = 1, \qquad \underset{W^{(N-1)}}{\nabla} G^{(N-1)} = O^{(N-1)}, \qquad \frac{\partial G^{(N-1)}}{\partial O^{(N-1)}} = W^{(N-1)}.$$

Substituting in the above formula, we have

$$\frac{\partial\, i}{\partial \text{Bias}_0^{(N-1)}} = 1, \qquad \underset{W^{(N-1)}}{\nabla} i = O^{(N-1)}, \qquad \underset{O^{(N-1)}}{\nabla} i = W^{(N-1)}. \tag{1.6}$$

Since $\text{card}_N = 1$, $W^{(N-1)}$ is a column matrix of same size than $O^{(N-1)}$, this is no longer true in the next steps.

Eventually, we deduce from $O_k^{(N-1)} = \chi(I_k^{(N-2)})$, that

$$\frac{\partial\, i}{\partial I_k^{(N-2)}} = \chi'(I_k^{(N-1)}) \frac{\partial\, i}{\partial O_k^{(N-1)}} = \chi'(I_k^{(N-1)}) W_{k,0}^{(N-1)}. \tag{1.7}$$

At this stage we have computed, $\dfrac{\partial\, i}{\partial \text{Bias}_0^{(N-1)}}$, $\underset{W^{(N-1)}}{\nabla} i$, $\underset{O^{(N-1)}}{\nabla} i$ and $\underset{I^{(N-2)}}{\nabla} i$.

(2) Now, we assume that $\underset{I^{(n)}}{\nabla} i$ has been computed for some $0 \leq n \leq N-1$, let us show how to compute

$$\underset{\mathrm{Bias}^{(n)}}{\nabla} i, \quad \underset{W^{(n)}}{\nabla} i, \quad \underset{O^{(n)}}{\nabla} i \quad \text{and} \quad \underset{I^{(n-1)}}{\nabla} i.$$

(a) According to the forward algorithm (1.3), the vector $I^{(n)}$ is a functions of $O^{(n)}$, $W^{(n)}$ and $\mathrm{Bias}^{(n)}$ and we can write again $I^{(n)} = G^{(n)}(O^{(n)}, W^{(n)}, \mathrm{Bias}^{(n)})$ for some function $G^{(n)}$ with values into $\mathbf{R}^{\mathrm{card}_{n+1}}$. By the chain rule formula, for $j \in \{0, \ldots, \mathrm{card}_n - 1\}$,

$$\frac{\partial i}{\partial \mathrm{Bias}_j^{(n)}} = \sum_{k=1}^{\mathrm{card}_{n+1}-1} \frac{\partial i}{\partial I_k^{(n)}} \frac{\partial G_k^{(n)}}{\partial \mathrm{Bias}_j^{(n)}}.$$

Formula (1.3) reads

$$G_k^{(n)}\left(O^{(n)}, W^{(n)}, \mathrm{Bias}^{(n)}\right) = \sum_{j=0}^{\mathrm{card}_n-1} W_{j,k}^{(n)} O_j^{(n)} + B_k^{(n)}. \tag{1.8}$$

Differentiating with respect to $\mathrm{Bias}_j^{(n)}$, we see that $\dfrac{\partial G_k^{(n)}}{\partial \mathrm{Bias}_j^{(n)}} = 1$, so

$$\underset{\mathrm{Bias}^{(n)}}{\nabla} i = \underset{I^{(n)}}{\nabla} i. \tag{1.9}$$

(b) Similarly, for $j \in \{0, \ldots, \mathrm{card}_n - 1\}$, $k \in \{0, \ldots, \mathrm{card}_{n+1} - 1\}$,

$$\frac{\partial i}{\partial W_{j,k}^{(n)}} = \sum_{k'=1}^{\mathrm{card}_{n+1}-1} \frac{\partial i}{\partial I_{k'}^{(n)}} \frac{\partial G_{k'}^{(n)}}{W_{j,k}^{(n)}}.$$

Differentiating (1.8) with respect to $W_{j,k}^{(n)}$, we see that $\dfrac{\partial G_{k'}^{(n)}}{W_{j,k}^{(n)}} = 0$ if $k' \neq k$ and $\dfrac{\partial G_{k'}^{(n)}}{W_{j,k}^{(n)}} = O_j^{(n)}$ if $k' = k$. This leads to

$$\frac{\partial i}{\partial W_{j,k}^{(n)}} = O_j^{(n)} \frac{\partial i}{\partial I_k^{(n)}}.$$

Using tensor notation, we write

$$\underset{W^{(n)}}{\nabla} i = O^{(n)} \otimes \underset{I^{(n)}}{\nabla} i. \tag{1.10}$$

(c) Now, for $j \in \{0, \ldots, \mathrm{card}_n - 1\}$,

$$\frac{\partial i}{\partial O_j^{(n)}} = \sum_{k=1}^{\mathrm{card}_{n+1}-1} \frac{\partial i}{\partial I_k^{(n)}} \frac{\partial G_k^{(n)}}{O_j^{(n)}}.$$

We have $\dfrac{\partial G_k^{(n)}}{O_j^{(n)}} = W_{j,k}^{(n)}$, hence

$$\frac{\partial i}{\partial O_j^{(n)}} = \sum_{k=1}^{\text{card}_{n+1}-1} W_{j,k}^{(n)} \frac{\partial i}{\partial I_k^{(n)}}.$$

In matrix forms, this reads,

$$\underset{O^{(n)}}{\nabla} i = W^{(n)} \underset{I^{(n)}}{\nabla} i. \tag{1.11}$$

(d) If $n > 0$, $O_j(n) = \chi(I_j^{(n-1)})$ and since $W^{(n)}$ and $\text{Bias}^{(n)}$ does not depend on $I^{(n-1)}$, we have for $j \in \{0, \ldots, \text{card}_n - 1\}$,

$$\frac{\partial i}{\partial I_j^{(n-1)}} = \chi'(I_j^{(n-1)}) \frac{\partial i}{\partial O_j^{(n)}}.$$

Using (1.11), this yields

$$\underset{I^{(n-1)}}{\nabla} i = \text{diag}(\chi'(I^{(n-1)})) \cdot W^{(n)} \underset{I^{(n)}}{\nabla} i, \tag{1.12}$$

where $\text{diag}(\chi'(I^{(n-1)}))$ is the diagonal $\text{card}_n \times \text{card}_n$ square matrix with diagonal terms $\chi'(I_0^{(n-1)}), \ldots, \chi'(I_{\text{card}_n-1}^{(n-1)})$.

In conclusion, the backward algorithm runs as follows, we start from (1.5) and then we compute the gradients $\underset{I^n}{\nabla} i$, $\underset{\text{Bias}^n}{\nabla} i$, $\underset{\text{Bias}^n}{\nabla} i$ by using (1.9), (1.10) and (1.12) recursively backward for $n = N-1, N-2, \ldots, 0$.

Remark, that by (1.9), we always have $\underset{\text{Bias}^n}{\nabla} i = \underset{I^n}{\nabla} i$ so that we do not need to store both vectors in the implementation. Remark also that we do not need to compute the vectors $\underset{O^{(n)}}{\nabla} i$.

2 The tool box

2.1 Toy classification problems

We consider a subset E of a rectangle $[a, b] \times [c, d]$ defined as a sublevel set of a function f,

$$E = \{x \in [a, b] \times [c, d] : f(x) < 0\}.$$

For instance, the choice $f(x) = max(x_0, x_1) - 1/2$ leads to the square $E = (-1/2, 1/2) \times (-1/2, 1/2)$ and $f(x) = x_0^2 + x_1^2 - 1/4$ leads to the disc $E = B(0, 1/2)$.

We assume that we have a set of training points $x^j \in [a, b] \times [c, d]$ for $j \in \{0, \ldots, n_{tr} - 1\}$ for which we know whether $x^j \in E$ or not. We note $y^j := \text{sign}(x^j)$, that is:

$$y^j = \begin{cases} -1 & \text{if } x^j \in E, \\ 0 & \text{if } x^j \in \partial E, \\ 1 & \text{if } x^j \in [a, b] \times [c, d] \backslash \overline{E}. \end{cases}$$

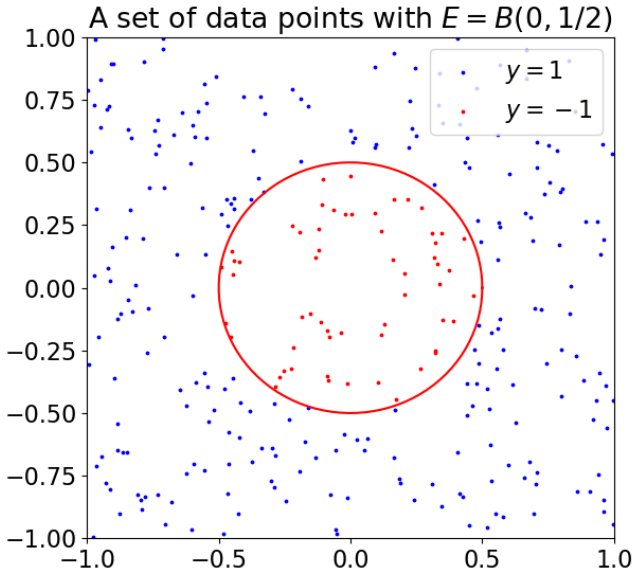


Figure 2:

More generally, we note $y(x) = \text{sign}(f(x))$. Figure 2 displays an example of a set of points randomly and independently chosen in the square $[-1, 1] \times [-1, 1]$ and classified according to whether they belong to $E = B(0, 1/2)$.

Given $S = \{(x^j, y^j) : j \in \{1, \dots, n_{tr} - 1\}\}$ and the bounds (a, b, c, d), we want to produce a prediction function

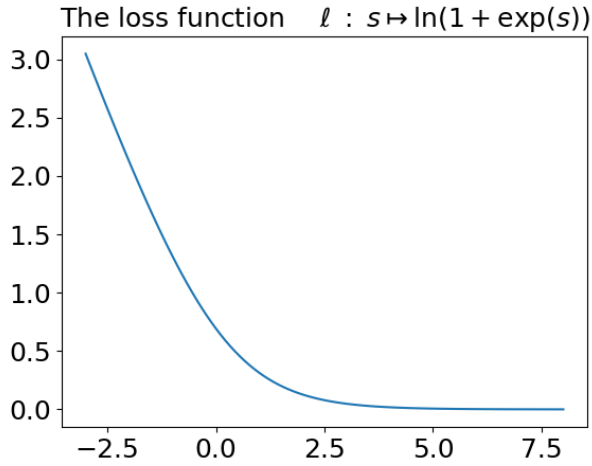
$$\widehat{f} : [a, b] \times [c, d] \to \mathbf{R}$$

such that

$$\widehat{f}(x) < 0 \quad \text{when } x \in E, \qquad \widehat{f}(x) > 0 \quad \text{when } x \in [a, b] \times [c, d] \backslash \overline{E}.$$

To measure the error we use a loss function $\ell : \mathbf{R} \to \mathbf{R}_+$ (that we assume to be convex and non-increasing). Below, we take a smooth version of the function $\max(-s, 0)$,

$$\ell(s) := \ln(1 + \exp(-s)) \quad \text{or} \quad \ell_0(s) := \sqrt{\varepsilon^2 + s^2} - s, \text{ with } \varepsilon^2 = 1/10. \tag{2.1}$$



The error on the instance $x \in [a,b] \times [c,d]$ is given by

$$err(x) = \ell(y(x)\widehat{f}(x)),$$

Therefore, if $x \in E$, $y(x) = -1$ and we want $\widehat{f}(x)$ to be large and positive in order to minimize $err(x)$. On the contrary, if $y(x) = 1$, we want $\widehat{f}(x)$ to be negative. On the total training data, the error is

$$E_{tr}(\widehat{f}) := \sum_{j=0}^{n_{tr}-1} \ell(y^j \widehat{f}(x^j)).$$

The class ToyPb

The python class `ToyPb` is designed to describe the classification problems above. The creation function is of the form:

```
pb = Toypb(name= , dim= ,  bounds= , f= ,  loss= , loss_prime= )
```

The attributes of an object of this class are :

1. `name`: a string containing the name of the problem. Available names are: `"affine"`, `"square"`, `"sin"`, `"disc"`, `"ring"`;

2. `dim`: an integer, number of free variables. The default value is 2;

3. `bounds`: a list of four floats (a,b,c,d) which define the rectangle $[a,b] \times [b,c]$ where the classification problem is set. The default value is `(-1,1)*dim`;

4. `f`: a numerical python function implementing the function f. `f` takes an array of two floats $([x_0, x_1])$ and returns a float. If `name` is not specified, then it is necessary to specify the value of `f` by `f=f1` where `f1` is a numerical function. It is also possible to define `pb.f` later by using the method `criteria`:
   `pb.criteria(name="string", f=f1,  bounds=(a,b,c,d))`;

5. `loss`: a numerical python function encoding the function ℓ. If `loss` is not specified, then `pb.loss` implements the function ℓ_0 of (2.1). If `loss="softplus"` then `pb.loss` implements the function ℓ of (2.1) and if `loss="demanding"`, `pb.loss` implements the function $s \mapsto \ell_0(s-1)$. For other choices, specify `loss=f1` and `loss_prime=f2` where `f1` and `f2` are python numerical functions (for consistancy `f2` should implement the mathematical derivative of `f1`).

6. `loss_prime`: a numerical python function implementing the derivative ℓ' of ℓ.

The method `show_border` plots ∂E (the curve $\{x \in [a,b] \times [c,d] : f(x) = 0\}$). This method requires `pb.name` to be in the list given above.

2.2 Data

As already said, our data sets are of the form $\{(x^j, y^j) : j \in \{0, \dots, n_{tr}\}\}$ with $x^j \in \mathbf{R}^2$ and $y^j \in \{-1, 1\}$ (in fact the class below allows for $x^j \in \mathbf{R}^{\dim}$, $\dim \geq 1$).

To assess a prediction function $\widehat{f}$, it is useful to have a similar (but often smaller) set of test data $\{(\tilde{x}^j, \tilde{y}^j) : j \in \{0, \dots, n_{test}\}\}$. The test error is defined as

$$E_{test}(\widehat{f}) := \sum_{j=0}^{n_{test}-1} \ell(\tilde{y}^j \widehat{f}(\tilde{x}^j)).$$

It will be convenient to store the predictions $\widehat{f}(\tilde{x}^j)$ as a sequence of values $\widehat{y}^j := \widehat{f}(\tilde{x}^j)$, $j = 0, \dots, n_{test}$.

The class nD_DATA

The creation function is of the form:

```
DATA= nD_data (n= , dim= , pb= , bounds= ,
                                    init_pred= , loss= , loss_prime=  )
```

The attributes of an object of the class `nD_data` are :

1. `n`: an integer, the number of data;

2. `dim`: an integer, the number of free variables. Default value is 2;

3. `bounds`: a tuple of $(2 \times \dim)$ floats $(a_0, b_0, \dots, a_{\dim} - 1, b_{\dim} - 1)$. Equals to `bounds` if `bounds` is specified or to `pb.bounds` if `pb` is specified with a value an object in the class `ToyPb`;

4. `X`: a numpy array of size `n` $\times$ `dim` which contains the coordinates of `n` points of $\mathbf{R}^{\dim}$. These coordinates are chosen randomly in $[a,b] \times [c,d]$ (uniformly and independently);

5. `Y`: a numpy array of length `n` which contains a list of `n` values in $\{-1, 1\}$. This array is defined by the constructor if the parameter `pb` is defined (as an object of the class `ToyPb`). In this case `Y[j]` is ± 1 if `pb.f[X[j]]`$\lessgtr 0$.

6. `Ypred`: a numpy array of length `n` which contains a list of `n` values. This array is defined by the constructor if the parameter `init_pred` is `True`. In this case it is initialized as a zero numpy array of length `n`. This initialization is necessary for creating predictions by a neural network. It can be performed later by using the method `DATA.init_pred()`.

Given an object `NN` in the class `ToyNN`, the method `DATA.prediction(NN)` updates the values `DATA.Ypred[j]` according to the outputs of the neural network `NN` corresponding to the input data `DATA.X[j]`.

The method `show_class(pred=None)` plots the points `DATA.X[j]` with two possible colors. If `Ypred` is `None` (or `False`, or 0 or the empty string, tuple or list) the colors depend on the sign of the respective values `DATA.Y[j]` (red for negative values, blue for positive ones, see the example Figure 2). If any other value is given to `pred`, the coloring is performed according to the sign of `DATA.Ypred[j]`.

2.3 The Neural netwrk

We describe eventually the class `ToyNN` whose objects are simple neural networks which are characterised by their number of layers, the number of nodes in each layer, the coefficients associated with the edges between nodes and the biasses associated with each node. The class comes with methods allowing the implementation of descent methods.

The class ToyNN

The attributes of an object of the class `ToyNN` are :

1. `N`: an integer, the number of interlayers;
2. `card`: a tuple of length $N+1$ containing the number of node in each layer;
3. `W`: a list of N 2D numpy arrays. `W[n]` has dimension `card[n]`$\times$`card[n+1]` and contains the weights associated to the edges betwen the nodes of layer n and the nodes of layer $n+1$;
4. `Bias`: a list of N 2 numpy arrays. `Bias[n]` has length `card[n+1]` and contains the bias at the nodes of layer $n+1$;
5. `DW`: a list of N 2D numpy arrays. `DW[n]` has dimension `card[-n]`$\times$`card[n+1]` and represents vector variation listed in reverse order `W`;
6. `DBias`: a list of N 2 numpy arrays. `DBias[n]` has length `card[-n+1]` and represents vector variation listed in reverse order `Bias`.
7. `chi`: a numerical function implementing the activation function of the neural network. It can be set by the option `chi`. The values `"tanh"`, `"sigmoide"` and `"RELU"` are avaliable. It is also possible to pick any other function `f` by `chi=f`. In this case, it is also necessary to specify `chi=g` where `g` is the dervative of `f`.
8. `chi_prime`: the mathematical derivative of `chi`;

9. `xx`, `yy`, `zz` : three 2D arrays for the graphic representation of the NN's output. They are crated if the option `grid` is given a tuple of the form `(xmin, xmax, ngrid)` or `(xmin, xmax, ymin, ymax, nxgrid, nygrid)`.

Given an array `X` of length `NN.card[0]`, the method `NN.output(X)` returns the output produced by the neural network for the input `X`.

Given the same array `X` of length `NN.card[0]`, a number `y`, a problem `pb` and a floating number `t`, the method `NN.descent(X=X,y=y, pb=pb,tau=t)` returns two lists `Desc_W` and `Desc_Bias` which contain the gradient with respect to the coefficients of the neural network `NN` of the cost associated to the output, the given correct value `y` and the loss function `pb.loss`. Beware that these lists are arranged in reverse order compared to the order of the lists `NN.W` and `NN.Bias`.

In the following methods, if the values of `dW` and `dBias` are not given, then `dW=self.DW` and `dBias=self.DBias`.

The method `NN.init_vector(dW,dBias)` initialize the vectors `dW`, `dBias` to 0.

The method `NN.add_to_vector(dW,dBias)` increment `self.DW` and `dBias` with `dW`, `dBias` respectively.

The method `mult_vector(dW,dBias,c)` multiplies `dW`, `dBias` by `c` and send it as an output or put the result `dW` ,`dBias` depending on the value of the input variable `'output'=True or False`.

The method `NN.add_to_coefs(dW,dB)` increment the degrees of freedom of the neural network with the values stored in the reverse ordered lists `dW,dB`.

There are other methods allowing computations on vectors.

Given an object `DATA` in the class `nD_data`, the method `NN.prediction(DATA)` updates the values `DATA.Ypred[j]` according to the outputs of the neural network `NN` corresponding to the input data `DATA.X[j]`.

The method `NN.show_pred()` displays the output produced by the neural network at the points of the grid with coordinates `NN.xx`, `NN.yy`.

The method `NN.show()` displays a graphic representation of the neural network. The width of each edge is proportional to the magnitude of the corresponding coefficient `NN.W[n][j,k]` and its color depends on the sign of this coefficient.