Building a Binary Classifier Using Neural Networks

Emanuele Sansone*

January 12, 2017

1 Basics of statistical learning theory (binary classification problem)

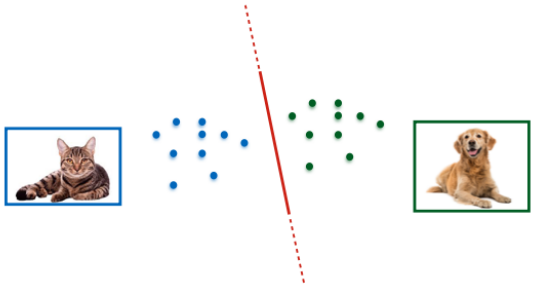


Figure 1: Example of binary classification problem. Each point corresponds to an image. Each color represents a class (the blue color is associated with class "cat" and the green color is associated with class "dog"). The goal is to learn the boundary (highlighted in red) between the two classes given the available images.

Let us consider the binary classification problem: we are given a training dataset $D_b = \{(\mathbf{x}_i, y_i) : \mathbf{x}_i \in X, y_i \in Y\}_{i=1}^m$, where $X \subseteq \mathbb{R}^d$, $Y = \{-1, 1\}$ and each pair of samples in D_b is drawn independently from the same joint distribution $\mathcal{P}$ defined over X and Y. The goal is to learn a function f that maps the input space X into the class set Y. In order to familiarize with this notation, let us consider the example in Figure 1, where the goal is to learn an image classifier discriminating between "cats" and "dogs". In this case, each point/image is represented by a two-dimensional vector denoted by $\mathbf{x}_i$, where i is used to identify an image in the training dataset, while y_i represents its respective class (either cat or dog). The boundary (e.g. linear separator) is denoted by function f.

How to learn f? Statistical learning theory [Vap99] provides us with a theoretically grounded answer. Assuming that $\mathcal{P}$ is known, the function f can be learnt by minimizing the *risk functional* $\mathcal{R}$, namely

$$\begin{aligned}\mathcal{R}(f) =& \sum_{y \in Y} \int \ell(f(\mathbf{x}), y) \mathcal{P}(\mathbf{x}, y) d\mathbf{x} \\ =& \pi \int \ell(f(\mathbf{x}), 1) \mathcal{P}(\mathbf{x}|y=1) d\mathbf{x} + (1-\pi) \int \ell(f(\mathbf{x}), -1) \mathcal{P}(\mathbf{x}|y=-1) d\mathbf{x} \end{aligned} \tag{1}$$

where π is the positive class prior, viz. $\mathcal{P}(y=1)$, and ℓ is a loss function measuring the disagreement between the prediction of our classifier and the ground truth for sample x, viz. $f(\mathbf{x})$ and y, respectively. For example, if ℓ is defined in the following way (equivalent to the definition of zero-one loss):

$$\ell(f(\boldsymbol{x}), y) = \begin{cases} 1 & f(\boldsymbol{x}) \neq y \\ 0 & f(\boldsymbol{x}) = y \end{cases} \tag{2}$$

*https://emsansone.github.io/

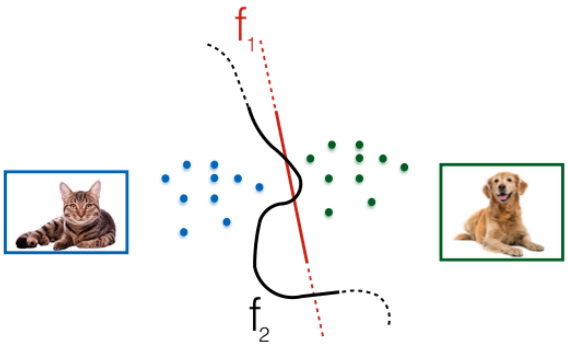


Figure 2: Example of binary classification problem. Two possible hypotheses are shown.

then the risk functional $\mathcal{R}$ is equivalent to the expected classification error rate.[1] More generally the risk functional is the expected value of the loss function, which is directly connected with the performance of the classifier. The higher is the number of misclassified samples, the higher is the risk/loss incurred by the classifier. Therefore, we need to find the function that minimizes it.

But $\mathcal{P}$ is unknown! This prevents us to compute the integrals in (1). If we cannot compute them exactly, then we can look for approximations. A good solution is to use the mean estimates over the available training data and this leads to the definition of the *empirical risk functional*, namely:

$$\mathcal{R}(f) \approx \mathcal{R}_{emp}(f) = \frac{\pi}{|D_b^+|} \sum_{\boldsymbol{x}_i \in D_b^+} \ell(f(\boldsymbol{x}_i), 1) + \frac{(1-\pi)}{|D_b^-|} \sum_{\boldsymbol{x}_i \in D_b^-} \ell(f(\boldsymbol{x}_i), -1) \tag{3}$$

where D_b^+ and D_b^- are the portions of dataset D_b belonging to the positive and negative class, respectively, while $|\cdot|$ is the cardinality operator. Unfortunately, minimizing (3) is an ill-posed problem, in the sense that multiple solutions may exist. This is because we are considering a finite number of training samples and thus many different classifiers can correctly discriminate them. For example, in Figure 1 there are multiple lines (in reality an infinite number of them) which separate correctly the two classes. The set of feasible solutions can be reduced to consider only "simple/smooth" cases. Let us consider the example in Figure 2, where there are two possible hypotheses for the family of polynomial functions. In this case, it would be more reasonable to choose the less complex function, namely the linear separator. The notion of simplicity/smoothness refers therefore to the complexity of the model. In order to make our problem well-posed, we can add a term (also called regularizer) to the empirical risk functional in order to penalize the complexity of solutions, namely:

$$\mathcal{R}^{\lambda}_{emp}(f) = \frac{\pi}{|D_b^+|} \sum_{\boldsymbol{x}_i \in D_b^+} \ell(f(\boldsymbol{x}_i), 1) + \frac{(1-\pi)}{|D_b^-|} \sum_{\boldsymbol{x}_i \in D_b^-} \ell(f(\boldsymbol{x}_i), -1) + \lambda\Omega(f) \tag{4}$$

where Ω is the regularizer and λ is a real-positive weight which is used to balance the relative importance of the complexity of the solution with respect to its expected loss (namely the first two terms in (4)). Therefore, the binary classification problem is defined as follows:

$$f^* = \arg\min_{f \in \mathcal{F}} \mathcal{R}^{\lambda}_{emp}(f) \tag{5}$$

where $\mathcal{F}$ is our solution space, called the **hypothesis space**. So far, we have considered examples with linear or polynomial functions. In the next section, we will consider more complex models.

We need to choose the loss function! The zero-one loss is the desired function, since it is directly connected with the classification error rate. Nevertheless, solving problem (4) with the zero-one loss is difficult firstly because it is NP-hard [FGRW12] and secondly because it is non-convex and it is therefore affected by the problem of local optima. What is usually done is

[1] Note that solving the binary classification problem with the zero-one loss is in general a NP-hard problem [FGRW12].

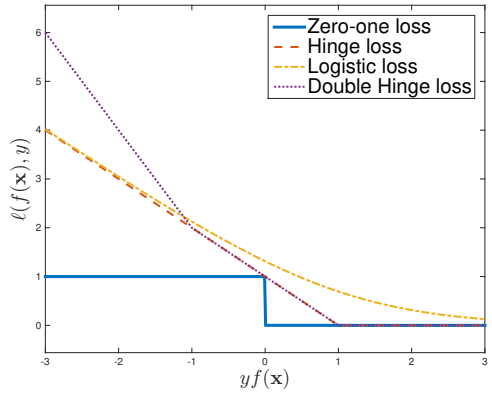


Figure 3: Visual comparison between different convex loss functions and the zero-one loss.

to use a convex loss function which approximates well the zero-one loss. The work in [RDVC+04] shows that such best choice corresponds to the Hinge loss function, namely:

$$\ell(f(\boldsymbol{x}), y) = \max\{0, 1 - yf(\boldsymbol{x})\} \tag{6}$$

Figure 3 provides a visual representation of different convex loss functions (compared against the zero-one loss). From that, it is quite evident that the Hinge loss is the function that best approximates the zero-one loss.

Let us now rewrite $\mathcal{R}^{\lambda}_{emp}$ using the Hinge loss function and denote the new regularized empirical risk as $\mathcal{R}^{\lambda,\mathcal{H}}_{emp}$, which allows to formulate the following optimization problem:

$$f^* = \arg\min_{f\in\mathcal{F}} \mathcal{R}^{\lambda,\mathcal{H}}_{emp}(f)$$
$$\arg\min_{f\in\mathcal{F}} \left\{ \frac{\pi}{|D_b^+|} \sum_{\boldsymbol{x}_i\in D_b^+} \max\{0, 1 - f(\boldsymbol{x}_i)\} + \frac{(1-\pi)}{|D_b^-|} \sum_{\boldsymbol{x}_i\in D_b^-} \max\{0, 1 + f(\boldsymbol{x}_i)\} + \lambda\Omega(f) \right\} \tag{7}$$

This is our final formulation of the binary classification problem.

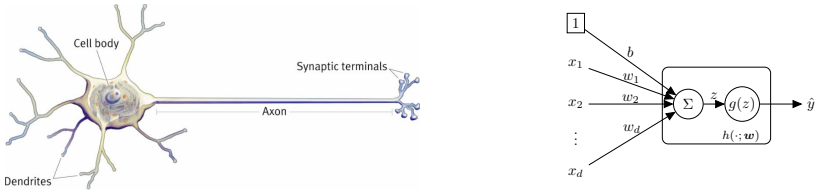
2 Basics of neural networks

In the previous section, we have seen how to formulate the binary classification problem as an optimization task. Here, we focus the attention on the family of functions (the hypothesis space), which can be described by parametric models, more specifically by neural networks.

The choice of using neural networks is dictated by the following reasons:

- **Biological inspiration**. Neural networks are frameworks whose aim is to reproduce the structure as well as the functionality of the human brain [MP43].
- **Representational power**. Neural networks can model large families of functions and therefore can be applied to a large variety of real-world problems [Cyb89].
- **Representation learning**. Neural networks allow to reduce the problem of feature engineering, because they learn new feature representations automatically from data [BCV13].

The computational unit of a neural network is the **artificial neuron**, which reproduces the functionality of the biological neuron in the human brain. Figures 4a-4b provide a visual analogy between the biological and artificial neurons. In the biological case, electrical signals are propagated through the dendrites and are accumulated in the body cell of the neuron. If the accumulated signals exceed a given level of charge, then the neuron is activated and an eletrical signal is sent through the axon. In the artificial case, the propagation of signals is modelled through the weighting of inputs, the accumulation of signals is performed through the summation operator and the output activation is controlled by function g, called the **activation function**. In literature, many different activation function are used. Some examples are given in Figure 5, namely:



(a) Biological neuron (b) Artificial neuron

Figure 4: Visual analogy between biological and artificial neurons.

- The binary threshold activation.

$$g(z) = \begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases}$$

- The linear activation.

$$g(z) = z$$

- The rectified linear activation.

$$g(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$$

- The sigmoid activation.

$$g(z) = \frac{1}{1 + e^{-z}}$$

Therefore, the response of the artificial neuron to an input vector $\boldsymbol{x} \in \mathcal{R}^d$ is given by $\hat{y} = h(\boldsymbol{x}; \boldsymbol{w}) = g(\boldsymbol{w}^T \boldsymbol{x} + b)$, which depends on the values of $\boldsymbol{w} = [w_1, \ldots, w_d]^T$ and b as well as on the kind of activation function employed.

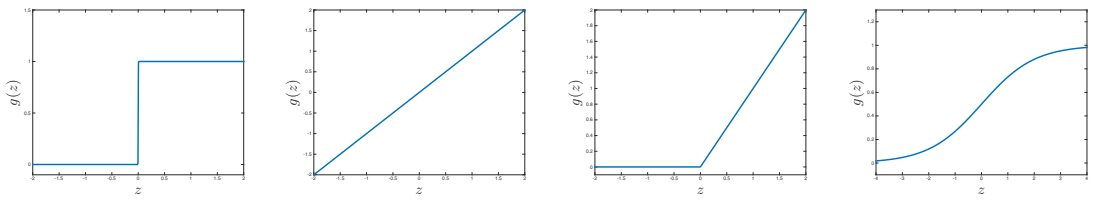
By combining multiple artificial neurons, it is possible to build more sophisticated models, which are able to describe more complex functions. Typical architectures of neural networks can be classified according to three main categories:

- **Feedforward** neural nets. They are organized in multiple layers stacked on top of each other (we distinguish between input, hidden and output layers). Each layer transforms the output of its previous layer and provides the result to the next layer. If we denote $\boldsymbol{h}^{(l)}$ as the transformation performed by layer l,[2] where $l \in \{1, \ldots, L, L+1\}$, then the global effect of a feedforward neural net is equivalent to a composition of functions, namely $\hat{y}^{(L+1)} = f(\boldsymbol{x}) = h^{(L+1)} \circ \boldsymbol{h}^{(L)} \circ \cdots \circ \boldsymbol{h}^{(1)}(\boldsymbol{x})$. See Figure 6a for a graphical representation.

  For the sake of notation compactness, we define $\mathbf{W}^{(l)} \in \mathcal{R}^{n_l \times n_{l-1}}$ as the matrix containing the weights of all neurons in a given layer l (included the bias). n_l is the number of neurons in layer l.

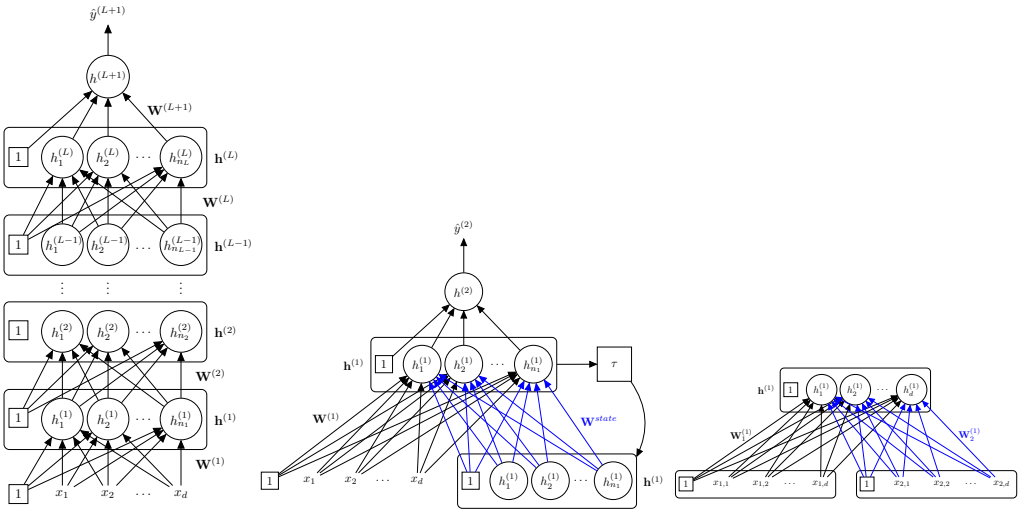
- **Recurrent** neural nets. They are simple neural networks with the addition of a memory mechanism, which keeps track about the state of the network. They are mainly used for data sream processing. The output of these networks is a function that depends on the current input data (at time t) and the previous state (at time $t - \tau$), namely $\hat{y}^{(2)} = f(\boldsymbol{x}) = h^{(2)}(\boldsymbol{W}^{(2)} \boldsymbol{h}_t^{(1)}(\boldsymbol{W}^{(1)} \boldsymbol{x} + \boldsymbol{W}^{state} \boldsymbol{h}_{t-\tau}^{(1)}))$. See Figure 6b for a graphical representation.

[2] Note the use of vector notation. Here we are considering the effect of all neurons in layer l.



(a) Binary threshold (b) Linear (c) Rectified linear (d) Sigmoid

Figure 5: Activation functions for artificial neurons.



(a) Feedforward (b) Recurrent (c) Recursive

Figure 6: Architectures of neural networks.

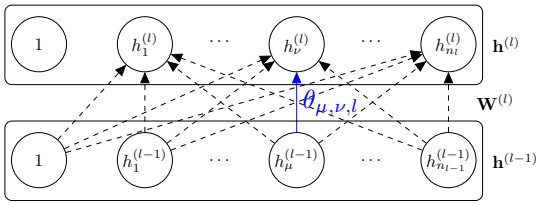


Figure 7: Notation used to identify the parameters in any layer.

- **Recursive** neural nets. The simplest architecture consists of many input layers (two or more) and one hidden layer. The dimensionality of the hidden layer is equal to the one of the input layers, thus allowing to reuse the basic network to create more complex architectures, like tree structures. They are mainly used to process/learn structured data (e.g. text). See Figure 6c for a graphical representation.

In this work, we focus only on feedforward neural networks, also because recurrent as well as recursive neural nets can be converted into the feedforward category (through **time** and/or **structure unfolding**).

It is important to make a distinction between the terms **parameters** and **hyperparameters** of a network (and more generally of any machine learning model), in order to avoid any source of confusion and misunderstanding. All unknown quantities of a model, which have to be learnt during the training stage, are called parameters, while all quantities introduced during the design stage are called hyperparameters. For example, the parameters of a feedforward neural networks are the weight matrices $\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(L)}, \boldsymbol{W}^{(L+1)}$, while the hyperparameters of that network are the number of hidden layers, called L, the number of neurons contained in each layer and all other design quantities (like λ and π in (7) for the binary classification problem).

Now, we focus on how to train neural networks, or equivalently on how to learn the parameters using the available training data.

2.1 Training neural networks: the backpropagation algorithm

Modify using delta notation SEE SOLUTION TO ASSIGNMENT

Here, we derive a general algorithm that allows to train any kind of feedforward neural network. The following three elements are the basic ingredients of any training algorithm (we recall them to introduce some more general notation):

- A **parametric model**, viz. a feedforward neural network, which is used to model function

$f \in \mathcal{F}_{\boldsymbol{\theta}}$, where $\boldsymbol{\theta} = (\theta_{0,1,1}, \ldots, \theta_{0,n_1,1}, \ldots, \theta_{\mu,\nu,l}, \ldots, \theta_{0,1,L+1}, \ldots, \theta_{n_L,1,L+1})^T \in \mathbb{R}^k$ is the vector containing all parameters.

In this case, $\theta_{\mu,\nu,l}$ is equal to the entry of matrix $\boldsymbol{W}^{(l)}$ located at row ν and column μ and it corresponds to the weight associated with the link going from neuron μ in the $(l-1)$-th layer to neuron ν in the l-th layer. Figure 7 provides a visual summary for this notation.

- A **training dataset** $D = \{(\boldsymbol{x}_i, y_i)_{i=1}^m\}$
- An **objective function** $J(\boldsymbol{\xi}(\boldsymbol{\theta}))$, which is used to train the neural network, where $J : \mathbb{R}^{m+1} \to \mathbb{R}$, $\Omega : \mathbb{R}^k \to \mathbb{R}$ (is the **regularizer function**) and

$$\boldsymbol{\xi}(\boldsymbol{\theta}) = \begin{bmatrix} \xi_1 \\ \vdots \\ \xi_m \\ \xi_{m+1} \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{x}_1; \boldsymbol{\theta}) \\ \vdots \\ f(\boldsymbol{x}_m; \boldsymbol{\theta}) \\ \Omega(\boldsymbol{\theta}) \end{bmatrix} \tag{8}$$

If the problem is minimizing $J(\boldsymbol{\xi}(\boldsymbol{\theta}))$, then we can use an iterative algorithm, like the gradient (subgradient) descent, to find a solution.[3] Computing the solution analytically is not generally feasible, since in the majority of cases the objective function is highly non-linear (i.e. by equating the gradient of the objective to zero, we would obtain a system of non-linear equations).

Therefore, a local optimal solution[4] can be obtained by repeatedly applying the following update rule (from the gradient descent algorithm):

$$\theta_{\mu,\nu,l}^{t+1} = \theta_{\mu,\nu,l}^{t} - \eta \frac{\partial J(\boldsymbol{\xi}(\boldsymbol{\theta}^t))}{\partial \theta_{\mu,\nu,l}} \quad \forall \mu, \nu, l \tag{9}$$

where $\eta \in \mathbb{R}^+$ is called the **learning rate** and $\frac{\partial J(\boldsymbol{\xi}(\boldsymbol{\theta}^t))}{\partial \theta_{\mu,\nu,l}}$ is computed in the following way:

$$\begin{aligned} \frac{\partial J(\boldsymbol{\xi}(\boldsymbol{\theta}))}{\partial \theta_{\mu,\nu,l}} &= \nabla J(\boldsymbol{\xi}) \cdot \left[\frac{\partial \xi_1}{\partial \theta_{\mu,\nu,l}}, \ldots, \frac{\partial \xi_{m+1}}{\partial \theta_{\mu,\nu,l}} \right]^T \\ &= \sum_{i=1}^{m+1} \frac{\partial J(\boldsymbol{\xi})}{\partial \xi_i} \frac{\partial \xi_i}{\partial \theta_{\mu,\nu,l}} \end{aligned} \tag{10}$$

In (10), the computation of $\nabla J(\boldsymbol{\xi}(\boldsymbol{\theta}^t))$ is straightforward once $\boldsymbol{\xi}(\boldsymbol{\theta}^t)$ is known, whereas the computation of $\frac{\partial \xi_i}{\partial \theta_{\mu,\nu,l}}$ is more challenging. In fact,

$$i = m+1 \quad \Rightarrow \quad \frac{\partial \xi_i(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}} = \frac{\partial \Omega(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}} \tag{11}$$

$$i = 1, \ldots, m \quad \Rightarrow \quad \frac{\partial \xi_i(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}} = \frac{\partial f(\boldsymbol{x}_i; \boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}} = \frac{\partial \hat{y}_i^{(L+1)}}{\partial \theta_{\mu,\nu,l}} = \begin{cases} \frac{\partial \hat{y}_i^{(l)}}{\partial \theta_{\mu,\nu,l}}, & l = L+1 \\ \frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\nu}^{(l)}} \frac{\partial \hat{y}_{i,\nu}^{(l)}}{\partial \theta_{\mu,\nu,l}}, & l \neq L+1 \end{cases} \tag{12}$$

which is obtained by applying the chain rule over the composition of functions modelled by the given network (see Figure 8).

It is important to mention that before computing (12), we need firstly to estimate all of its

[3] Assuming that $J(\boldsymbol{\xi}(\boldsymbol{\theta}))$ is continuous in $\boldsymbol{\theta}$.

[4] We are not able to achieve a global optimal solution, since in general the objective function is not convex.

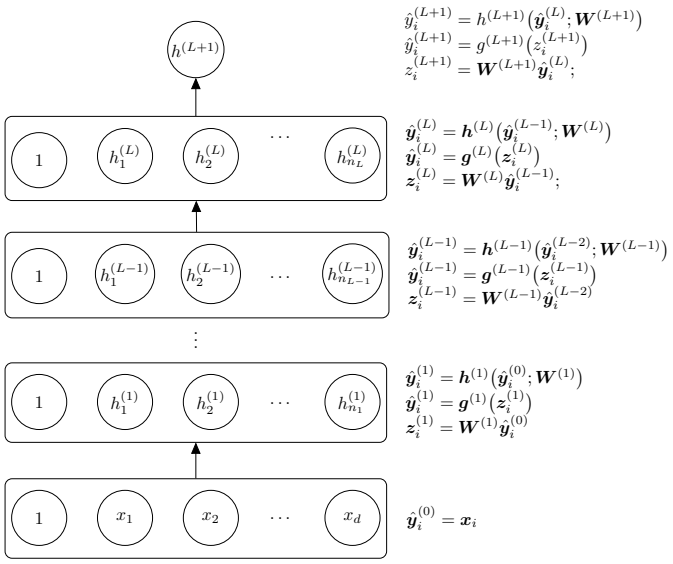


Figure 8: Composition of functions.

terms, namely:

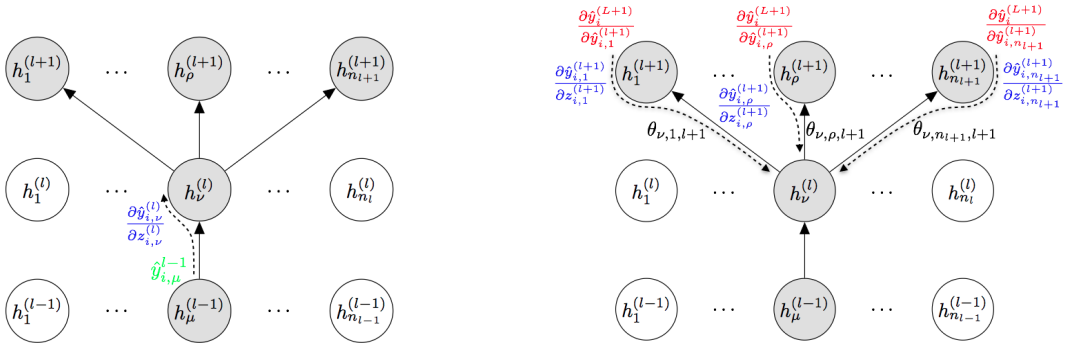
$$\begin{aligned}
\frac{\partial \hat{y}_{i,\nu}^{(l)}}{\partial \theta_{\mu,\nu,l}} &= \frac{d\hat{y}_{i,\nu}^{(l)}}{dz_{i,\nu}^{(l)}} \frac{\partial z_{i,\nu}^{(l)}}{\partial \theta_{\mu,\nu,l}} \\
&= \frac{d\hat{y}_{i,\nu}^{(l)}}{dz_{i,\nu}^{(l)}} \hat{y}_{i,\mu}^{(l-1)}, \qquad \forall l = 1, \dots, L+1 \qquad (13) \\
\frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\nu}^{(l)}} &= \sum_{\rho=1}^{n_{l+1}} \frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\rho}^{(l+1)}} \frac{\partial \hat{y}_{i,\rho}^{(l+1)}}{\partial \hat{y}_{i,\nu}^{(l)}} \\
&= \sum_{\rho=1}^{n_{l+1}} \frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\rho}^{(l+1)}} \frac{d\hat{y}_{i,\rho}^{(l+1)}}{dz_{i,\rho}^{(l+1)}} \frac{\partial z_{i,\rho}^{(l+1)}}{\partial \hat{y}_{i,\nu}^{(l)}} \\
&= \sum_{\rho=1}^{n_{l+1}} \frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\rho}^{(l+1)}} \frac{d\hat{y}_{i,\rho}^{(l+1)}}{dz_{i,\rho}^{(l+1)}} \theta_{\nu,\rho,l+1}, \qquad \forall l = 1, \dots, L \qquad (14)
\end{aligned}$$

where the terms corresponding to the **derivatives of the activation functions** are highlighted in blue, the terms measuring the total variation of the output of the whole network due to a little variation in the output of any hidden neuron, called **local-to-global variations**, are highlighted in red, while the outputs of previous layer are highlighted in green.[5]

Let us note that blue and green terms can be obtained only when $\hat{\boldsymbol{y}}_i^{(l-1)}$ is known, while red terms can be obtained only through recursive computation of derivatives from above layers, as it is shown in Figure 9a and Figure 9b. This is a fundamental aspect, because it tells us that an iteration of the gradient descent algorithm is equivalent to perform the following two stages:

- **Forward propagation.** Given a training sample $\boldsymbol{x}_i$, compute and store the derivatives of activation functions and layer outputs, namely $\frac{d\hat{\boldsymbol{y}}_i^{(l)}}{d\boldsymbol{z}_i^{(l)}}$ and $\hat{\boldsymbol{y}}_i^{(l)}$, starting from the lowest layer and going up to the output layer.
- **Backward propagation.** Update the weights associated to each network link starting from the output layer and going down to the lowest layer by applying (9), exploiting information stored during forward propagation and using equations (10),(11),(12),(13) and (14).

[5] Equation (13) can be derived by using the chain rule, while Equation (14) can be derived by using the multivariable chain rule.



(a) (b)

Figure 9: Computation of $\frac{\partial \xi_i(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}}$. On the left, computation of (13); On the right, computation of (14).

By putting together all considerations made so far, we obtain the so-called **backpropagation algorithm** [RHW86], which is summarized in Algorithm 1. It is important to mention that this derivation of the backpropagation algorithm is very general, therefore it can be used to train any feedforward neural network for a large variety of tasks, like clustering, dimensionality reduction as well as classification. In the next section, we consider the specific problem of binary classification.

3 Binary classification using neural networks

In the previous section, we have seen that any training algorithm is characterized by three basic ingredients, namely a parametric model, a training dataset and an objective function. In the binary classification problem, these three elements can be defined as follows:

- The output of the feedforward neural network consists of a linear neuron, while all other layers are characterized by rectifier linear activation units [GBB11].
- The training dataset D is simply the concatenation of D_b^+ and D_b^-, viz. the two portions of training data belonging to the positive and negative classes, respectively.
- The objective function is the same as the one defined in (7). By adopting the same notation of (8) and using the L_2 norm as regularizer $\Omega(\boldsymbol{\theta})$[6], we can rewrite (7) as

$$J(\boldsymbol{\xi}(\boldsymbol{\theta})) = \frac{\pi}{|D_b^+|} \sum_{\boldsymbol{x}_i \in D_b^+} \max\{0, 1-\xi_i\} + \frac{(1-\pi)}{|D_b^+|} \sum_{\boldsymbol{x}_i \in D_b^-} \max\{0, 1+\xi_i\} + \lambda \|\boldsymbol{\theta}\|_2^2 \tag{15}$$

Before applying the general backpropagation algorithm to the binary classification problem, we need to compute the gradient of the objective function $\nabla J(\boldsymbol{\xi})$ and the derivatives of the activation functions $\frac{\partial \hat{\boldsymbol{y}}_i^{(l)}}{\partial \boldsymbol{z}_i^{(l)}}$ for any layer l of the neural network. Regarding the computation of the gradient term, we have that for all positive samples $\boldsymbol{x}_i \in D_b^+$

$$\frac{\partial J(\boldsymbol{\xi})}{\partial \xi_i} = \begin{cases} 0, & \xi_i \geq 1 \\ -\frac{\pi}{|D_b^+|}, & \xi_i < 1 \end{cases} \tag{16}$$

for all negative samples $\boldsymbol{x}_i \in D_b^-$

$$\frac{\partial J(\boldsymbol{\xi})}{\partial \xi_i} = \begin{cases} 0, & \xi_i \leq -1 \\ \frac{1-\pi}{|D_b^-|}, & \xi_i > -1 \end{cases} \tag{17}$$

while for $i = m+1$

$$\frac{\partial J(\boldsymbol{\xi})}{\partial \xi_{m+1}} = \lambda \tag{18}$$

[6] to enhance the sparsity in the obtained solution

Algorithm 1 General Backpropagation Algorithm

Initialize the network hyperparameters.
$\hat{\boldsymbol{y}}_i^{(0)} = \boldsymbol{x}_i$.
$\frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_i^{(L+1)}} = 1$
for Repeat until convergence **do**
  Compute $\frac{\partial \xi_{m+1}(\boldsymbol{\theta}^t)}{\partial \boldsymbol{\theta}}$. (11)
  for $l = 1, \ldots, L+1$ **do** ▷ Forward propagation
    for $i = 1, \ldots, m$ **do**
      Compute and store $\frac{d\hat{\boldsymbol{y}}_i^{(l)}}{d\boldsymbol{z}_i^{(l)}}$.
      Compute and store $\hat{\boldsymbol{y}}_i^{(l)}$.
    end for
  end for
  for $l = L+1, \ldots, 1$ **do** ▷ Backward propagation
    for $\nu = 1, \ldots, n_l$ **do**
      if $l \neq L+1$ **then**
        for $i = 1, \ldots, m$ **do**
          Compute and store $\frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\nu}^{(l)}} = \sum_{\rho=1}^{n_{l+1}} \frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\rho}^{(l+1)}} \frac{d\hat{y}_{i,\rho}^{(l+1)}}{dz_{i,\rho}^{(l+1)}} \theta_{\nu,\rho,l+1}^t$. (14)
        end for
      end if
      for $\mu = 0, \ldots, n_{l-1}$ **do**
        for $i = 1, \ldots, m$ **do**
          Compute $\frac{\partial \xi_i(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}} = \frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\nu}^{(l)}} \frac{\partial \hat{y}_{i,\nu}^{(l)}}{\partial \theta_{\mu,\nu,l}} = \frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\nu}^{(l)}} \frac{d\hat{y}_{i,\nu}^{(l)}}{dz_{i,\nu}^{(l)}} \hat{y}_{i,\mu}^{(l-1)}$. (12)(13)
        end for
        Compute $\frac{\partial J(\boldsymbol{\xi}(\boldsymbol{\theta}^t))}{\partial \theta_{\mu,\nu,l}} = \sum_{i=1}^m \left[\frac{\partial J(\boldsymbol{\xi}(\boldsymbol{\theta}^t))}{\partial \xi_i} \frac{\partial \xi_i(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}}\right] + \frac{\partial J(\boldsymbol{\xi}(\boldsymbol{\theta}^t))}{\partial \xi_{m+1}} \frac{\partial \xi_{m+1}(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}}$. (10)
        $\theta_{\mu,\nu,l}^{t+1} = \theta_{\mu,\nu,l}^t - \eta \frac{\partial J(\boldsymbol{\xi}(\boldsymbol{\theta}^t))}{\partial \theta_{\mu,\nu,l}}$.
      end for
    end for
  end for
  $t = t + 1$.
end for

Regarding the computation of the partial derivatives, we have that for the output neuron

$$\frac{\partial \hat{y}_i^{(L+1)}}{\partial z_i^{(L+1)}} = 1 \tag{19}$$

while for all other neurons

$$\frac{\partial \hat{y}_{i,\rho}^{(l)}}{\partial z_{i,\rho}^{(l)}} = \begin{cases} 0, & z_{i,\rho}^{(l)} \leq 0 \\ 1, & z_{i,\rho}^{(l)} > 0 \end{cases} \tag{20}$$

All these relations allow us to modify lines 5, 16, 21 and 23 in Algorithm 1 in order to obtain the specific version for the binary classification problem (See Algorithm 2).

Algorithm 2 Backpropagation Algorithm for Binary Classification

1: Initialize the network hyperparameters.
2: $\hat{\boldsymbol{y}}_i^{(0)} = \boldsymbol{x}_i$.
3: $\frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_i^{(L+1)}} = 1$
4: **for** Repeat until convergence **do**
5: Compute $\frac{\partial \xi_{m+1}(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}} = 2\theta_{\mu,\nu,l}^t \quad \forall \mu, \nu, l.$ (11)
6: **for** $l = 1, \ldots, L+1$ **do** ▷ Forward propagation
7: **for** $i = 1, \ldots, m$ **do**
8: Compute and store $\hat{\boldsymbol{y}}_i^{(l)}$.
9: **end for**
10: **end for**
11: **for** $l = L+1, \ldots, 1$ **do** ▷ Backward propagation
12: **for** $\nu = 1, \ldots, n_l$ **do**
13: **if** $l \neq L+1$ **then**
14: **for** $i = 1, \ldots, m$ **do**
15: Compute and store $\frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\nu}^{(l)}} = \sum_{\rho^*} \frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\rho^*}^{(l+1)}} \theta_{\nu,\rho^*,l+1}^t$. (14)
16: where $\rho^* \in \{1, \ldots, n_l : z_{i,\rho^*}^{(l+1)} > 0\}$
17: **end for**
18: **end if**
19: **for** $\mu = 0, \ldots, n_{l-1}$ **do**
20: **for** $i = 1, \ldots, m$ **do**
21: **if** $l = L+1$ **then** $\frac{\partial \xi_i(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}} = \hat{y}_{i,\mu}^{(l-1)}$. (12)(13)
22: **else** $\frac{\partial \xi_i(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}} = \begin{cases} 0, & z_{i,\nu}^{(l)} \leq 0 \\ \frac{\partial \hat{y}_i^{(L+1)}}{\partial \hat{y}_{i,\nu}^{(l)}} \hat{y}_{i,\mu}^{(l-1)}, & z_{i,\nu}^{(l)} > 0 \end{cases}$. (12)(13)
23: **end if**
24: **end for**
25: Compute $\frac{\partial J(\boldsymbol{\xi}(\boldsymbol{\theta}^t))}{\partial \theta_{\mu,\nu,l}} = -\frac{\pi}{|D_b^+|} \sum_{i^+} \frac{\partial \xi_{i^+}(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}} + \frac{(1-\pi)}{|D_b^-|} \sum_{i^-} \frac{\partial \xi_{i^-}(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}} + \lambda \frac{\partial \xi_{m+1}(\boldsymbol{\theta}^t)}{\partial \theta_{\mu,\nu,l}}$. (10)
26: where $i^+ \in \{i : \boldsymbol{x}_i \in D_b^+, \xi_i < 1\}$ and $i^- \in \{i : \boldsymbol{x}_i \in D_b^-, \xi_i > -1\}$
27: $\theta_{\mu,\nu,l}^{t+1} = \theta_{\mu,\nu,l}^t - \eta \frac{\partial J(\boldsymbol{\xi}(\boldsymbol{\theta}^t))}{\partial \theta_{\mu,\nu,l}}$.
28: **end for**
29: **end for**
30: **end for**
31: $t = t + 1$.
32: **end for**

References

[BCV13] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[Cyb89] George Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[FGRW12] Vitaly Feldman, Venkatesan Guruswami, Prasad Raghavendra, and Yi Wu. Agnostic Learning of Monomials by Halfspaces is Hard. *SIAM Journal on Computing*, 41(6):1558–1590, 2012.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep Sparse Rectifier Neural Networks. In *AISTATS*, pages 315–323, 2011.

[MP43] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[RDVC+04] Lorenzo Rosasco, Ernesto De Vito, Andrea Caponnetto, Michele Piana, and Alessandro Verri. Are Loss Functions All the Same? *Neural Computation*, 16(5):1063–1076, 2004.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[Vap99] Vladimir N Vapnik. An Overview of Statistical Learning Theory. *Neural Networks, IEEE Transactions on*, pages 988–999, 1999.