



Mathematics for Supply Chain

Msc Supply Chain & Purchasing (2023-2024)

Guillaume Metzler

Institut de Communication (ICOM)
Université de Lyon, Université Lumière Lyon 2
Laboratoire ERIC UR 3083, Lyon, France

guillaume.metzler@univ-lyon2.fr

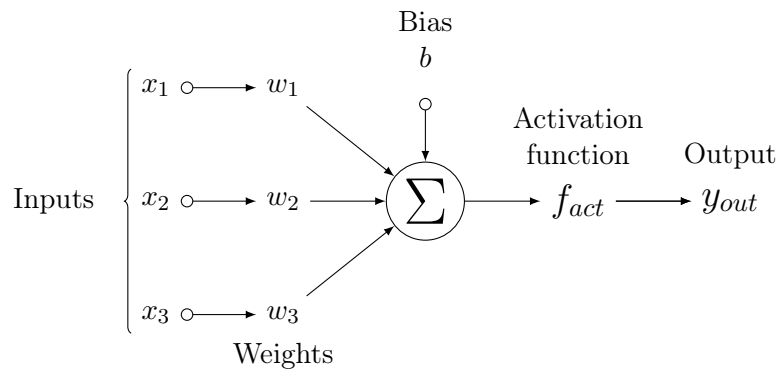


Figure 1: Illustration of a Perceptron with an input space of dimension 3. Each feature is multiplied by a parameter w and a bias b is added. When the sum is computed, the value pass through an activation function f to give the output.

1 Neural Networks : presentation and concepts

The model presented here is a model that can be used for both classification and regression but also in some unsupervised learning algorithms.

Foundations This algorithm is very freely inspired by nature and more precisely by neurosciences on synaptic models. The first traces of neural networks can be found in works dating from the middle of the 20th century [McCulloch and Pitts, 1943]. This is the first mathematical modeling of a neuron, which is currently known as a perceptron [Rosenblatt, 1958], a representation of which is given in Figure 1.

This type of model takes a vector $\mathbf{x} \in \mathbb{R}^d$ as an input and depends on one parameter $(\mathbf{w}, b) \in \mathbb{R}^{d+1}$. The output $h(\mathbf{x})$ is historically computed the as the sign of the inner product of vectors \mathbf{x} and \mathbf{w} to which a constant value b is added, *i.e.*

$$h(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b) = \text{sign} \left(\sum_{j=1}^d w_j x_j + b \right).$$

Thus, this first model was initially intended to do binary classification. The model that is used is affine (or linear if b is equal to 0) and an *activation function* is then applied to the output to return the predicted class:

$$h(\mathbf{x}) = \begin{cases} 0 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b < 0, \\ 1 & \text{otherwise.} \end{cases}$$

Note that the rule that is used to predict the class is the same as the one that we have seen with the SVM classifier. The activation function that is used here is called the *heavyside* function, illustrated in Figure 2 (left).

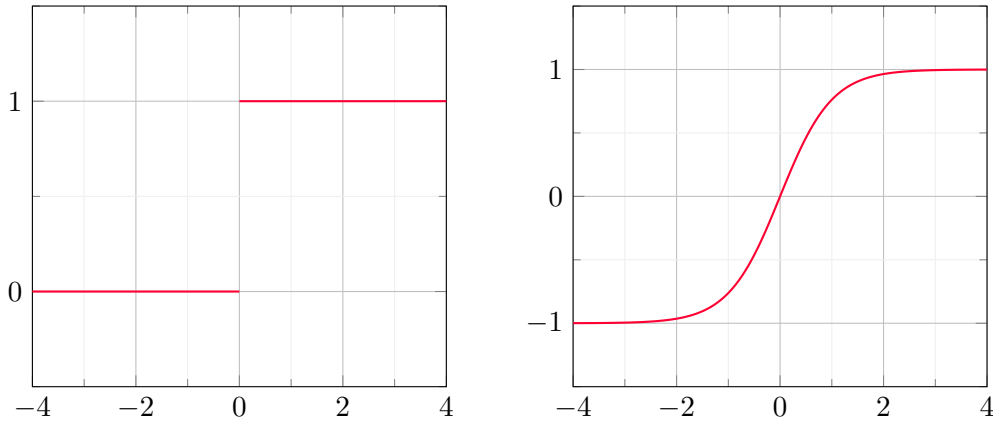


Figure 2: Illustration of two activation functions. On the left, the heavyside function, it takes the value 1 when x is positive and 0 otherwise. On the right, the function \tanh , defined by $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, takes its values in the range $[-1, 1]$.

The first algorithm that has been used to learn the parameters \mathbf{w} and b is known as the *Hebb algorithm* and the process is quite simple but it only converges for linearly separable data. The parameters are updated as follows: let us denote \mathcal{I} the set of missclassified instances. Then, for all (\mathbf{x}_i, y_i) such that $i \in \mathcal{I}$ compute:

$$\mathbf{w} = \mathbf{w} + \alpha y_i \mathbf{x}_i \quad \text{and} \quad b = b + \alpha y_i,$$

where α is the learning rate. The update rule is repeated as long as \mathcal{I} is not empty or it stops after a given number of iteration.

Another rule to learn the perceptron parameters, called *law of Widrow-Hoff*, works in a similar manner, but it also takes into account the error observed at the current state:

$$\mathbf{w} = \mathbf{w} + (y_i - h(\mathbf{x}_i)) \mathbf{x}_i \quad \text{and} \quad b = b + y_i - h(\mathbf{x}_i).$$

This second update rule can be used with the sign function but we usually use it with the \tanh function which can be seen as smoother version of the heavyside function with a little offset (see Figure 2 (right)), the sigmoid can also be used. The later has the advantage to be smooth compared the heavyside function where the derivative is equal to 0 almost everywhere and thus more suited for gradient descent algorithm.

Neural Networks The previously presented algorithms are interesting when we are dealing with problems that are linearly separable. But this situation is rarely met in

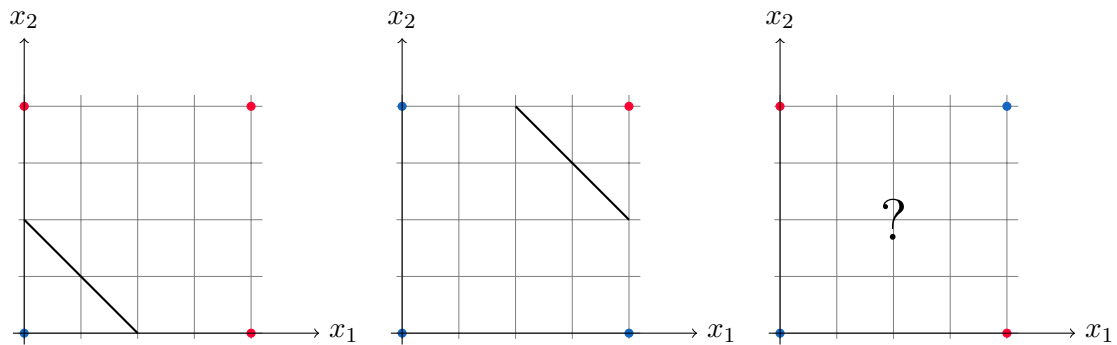


Figure 3: Representation, in a two dimensional space, of the **OR**, **AND** and **XOR** classification datasets respectively. A value x_i equal to 0 means that entry is *FALSE*, it is equal to 1 if it is *TRUE*. The color of the point is used to denote the label of the data which is determined by the logical operator: **TRUE** and **FALSE**.

practice and we need to develop more flexible and complex models to achieve performance for non linear classification problems.

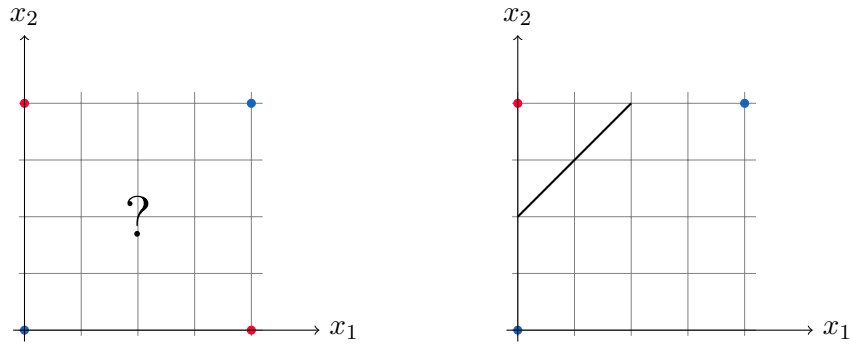
For instance, the perceptron algorithm is able to achieve good performances on the **OR** or **AND** dataset, *i.e.* a linear model is enough to classify the data, but it cannot solve the **XOR** problem. The problems are illustrated in Figure 3 using a two dimensional dataset and we effectively see that cannot separate the third dataset perfectly using a simple linear classifier, we need a more complex model, *i.e.* to learn another representation of data where the problem is linearly separable.

To do so, we are still inspired by neurosciences and the brain architecture where several neurons are connected between them, this what we call a *Neural Network*. But before going on with the architecture, let us go back to our example and see how can solve the **XOR** problem.

To solve this problem we can perform the following transformations and the x_1 and x_2 axes:

- $x_1 \leftarrow x_1 \wedge x_2$
- $x_2 \leftarrow x_1 \vee x_2$

to have the following new representation:



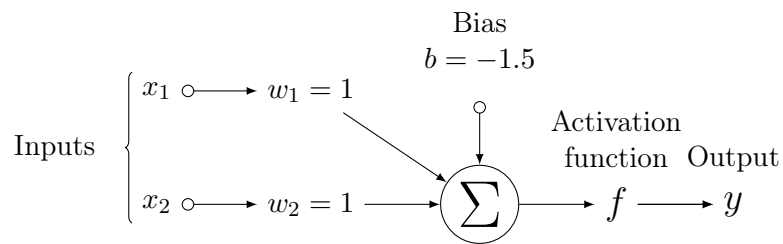
With this transformation, the two initial red points are projected at the same place in the new space because only one of their two entries was in the *TRUE* state. The representation of the blue points is the same since their two entries are the same. In this new representation, the problem is linearly separable and such a representation can be learned by a neural network, more precisely, using a multi layer perceptron.

To find the architecture, we first to do some logical reasoning and write the **XOR** function differently. This function is true if and only if exactly one of the input is true. In other words

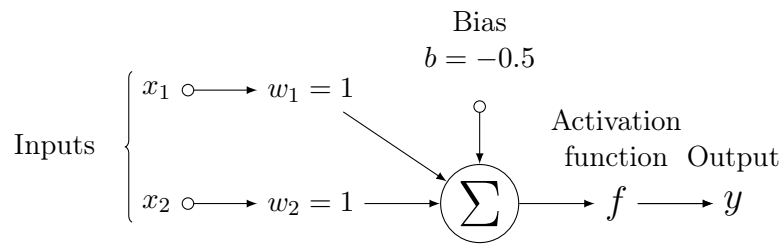
$$XOR(x_1, x_2) = (x_1 \vee x_2) \wedge (\overline{x_1 \wedge x_2}) .$$

It is now enough to translate this expression using several perceptron. For the **XOR** problem, we just how to combine the several perceptron given below. We leave it to the reader to represent the solution of the problem by combining these different perceptrons.

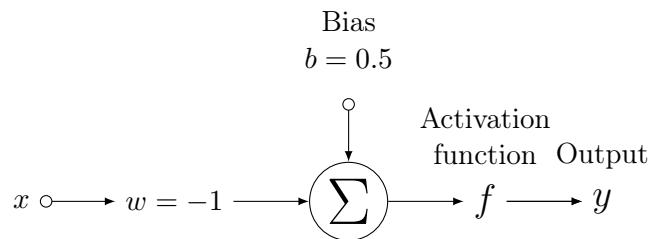
The **AND** perceptron:



The **OR** perceptron:



The **NOT** perceptron:



The activation function that is used is always the heavyside function.

A multiple layer perceptron is represented in Figure 4. The first layer is called *input layer*. Its size is equal to the dimension of the input space and we also add an other neuron for which the entry s always equal to 1 and which represents the bias term b .

The intermediate layers are called hidden layers. The number of layers and their sizes are defined by the user according to his needs and the problem he is facing. Again, to each hidden layer, we can associate a bias parameter that will be used to evaluate the output at the next layer.

The last layer is called the output layer, and its size also depends on the size of the output space. For a binary regression or classification problem, the output is of dimension 1. On the other hand, for a multi-class classification problem, the output layer will have as many neurons as there are classes in the data set.

The neural network presented in Figure 4 is said to be *fully connected*, *i.e.* all the inputs are connected to all of the outputs. However, it is possible to remove some of the connections, by cutting or setting the respective weights equal to 0. We could also imagine connections between two non successive layers.

We will finish this generalization on networks by evoking the number of parameters of a multi-layer neural network.

In the example given in Figure 4, we can see the number that the input is of dimension 3, the two hidden layers are respectively of dimension 3 and 2 and the output layer is of dimension 1. Further, at each hidden layers and at the input layer is associated

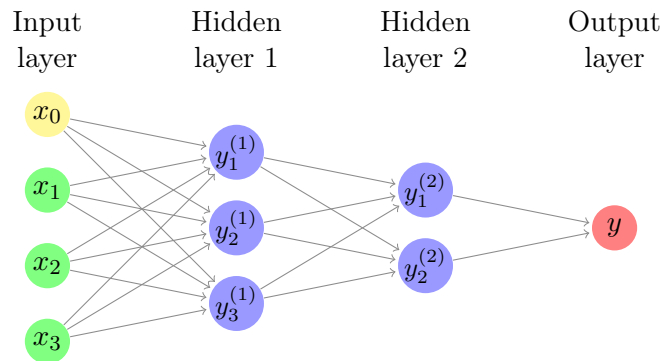


Figure 4: Representation of a multi layer perceptron. On this particular example, the input space is of dimension 3, there are two hidden layers of size 3 and 2 respectively. The output space is of dimension 1.

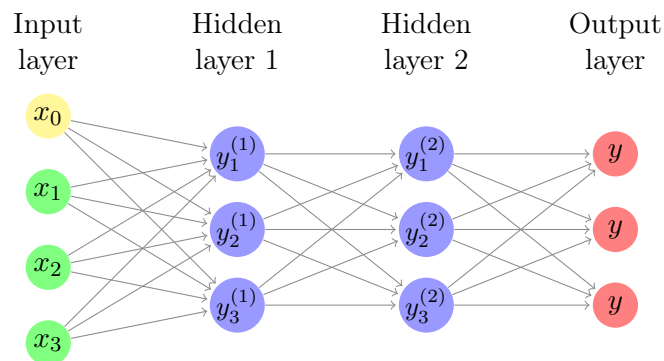
a bias term.

So, in our example, the number of links, which is equal to the number of parameter to learn, is the dimension of input layer plus one multiplied by the dimension of the hidden layer, thus 12 parameters. We also have 8 parameters between the second and the third layer and 3 parameters between the third and the fourth layer. Thus a total of 23 parameters for this network.

More generally, the number of parameters to learn in neural network with K hidden layers

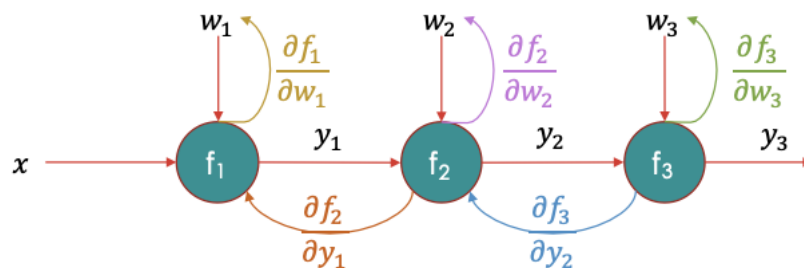
$$\sum_{k=1}^{K-1} \left(d^{(k+1)} \right) \times \left(d^{(k)} + 1 \right),$$

where the $d^{(k)}$ denotes the dimension/number of units the k -th layer. In the example below, the neural network has 48 parameters.



This last example shows a multi-layer perceptron than be used for *multi-class classification*. But we will give more precision about how it works when will present different losses we can use in neural networks and after providing information about how to learn the parameters in the next paragraph.

Training and losses (Back-Propagation algorithm) We will consider, for the sake of simplicity, the following network with only two hidden layers with a single unit, with a one dimensional input¹.



Where w_i, f_i denote respectively the parameters and the activation function of the i -th layer and y_i the output. The parameters of the network are updated standard gradient descent algorithm using a *Forward - Backward* procedure.

The *Forward* consists in giving data to the network one by one (stochastic) or using mini-batch (subset of the data) in order to compute the loss value. The second step is the *Backward* one, where the parameters update using the data and the loss value. The main difficulty will be to trace the error along the entire network in order to update all the parameters. Indeed, if the weights of the last layer are directly linked to the output value, the same cannot be said for the weights of the first hidden layer or the input layer. The gradient of the parameters of the first layers will directly depend on the gradient of the parameters of the layers further downstream.

2 Neural Network in practice

Training a neural network might be hard in practice since it requires to play with several parameters. In this section we will first try to have in insight of the power of neural networks (only for the multi-layer perceptron) for both classification and regression tasks. We will then use **R** to train our own network.

¹This example has been extracted from the Thesis of [Damien Fourure](#).


2.1 A first insight: complexity of networks

You can first study the following link to try to play a little with neural networks

[Link to an illustration with Tensorflow](#)

We will now have a look on how we can train a neural network using our software.

2.2 Training its own network

For regression task. We are going to work with the *Boston* dataset which is available on  directly to predict median value of owner-occupied homes in \$1000s (you can find more information on the dataset, and the meaning of each variable [here](#)).

```
# Import Required packages
set.seed(500)
library(neuralnet)
library(MASS)

# Boston dataset from MASS
data <- Boston
```

Before feeding the data into a neural network, it is good practice to perform normalization. There is a number of ways to perform normalization. We will use the min-max method and scale the data in the interval $[0,1]$. The data is then split into training (75%) and testing (25%) set.

```
# Normalize the data
maxs <- apply(data, 2, max)
mins <- apply(data, 2, min)
scaled <- as.data.frame(scale(data, center = mins,
                             scale = maxs - mins))

# Split the data into training and testing set
index <- sample(1:nrow(data), round(0.75 * nrow(data)))
train_ <- scaled[index,]
test_ <- scaled[-index,]
```

Now, we can create a neural network using the neuralnet library. Modify the parameters and calculate the mean squared error (MSE). Use the parameters with the least MSE. We will use two hidden layers having 5 and 3 neurons. The number of neurons should be between the input layer size and the output layer size, usually 2/3 of the input size. However, modifying and testing the neural network, again and again, is the best way to find the parameters that best fit your model. When this neural network is trained, it will perform gradient descent to find coefficients that fit the data until it arrives at the optimal weights (in this case regression coefficients) for the model.

```
# Build Neural Network
nn <- neuralnet(medv ~ crim + zn + indus + chas + nox
                + rm + age + dis + rad + tax +
                ptratio + black + lstat,
                data = train_, hidden = c(5, 3),
                linear.output = TRUE)

# Predict on test data
pr.nn <- compute(nn, test_[,1:13])

# Compute mean squared error
pr.nn_ <- pr.nn$net.result * (max(data$medv) - min(data$medv))

test.r <- (test_$medv) * (max(data$medv) - min(data$medv)) +

MSE.nn <- sum((test.r - pr.nn_)^2) / nrow(test_)
```

We can also have a look at the structure of our network.

```
# Plot the neural network
plot(nn)
```

We can finally try to detect study if the predictions are good using the following code which aims to plot a graph which compares predicted and observed values.

```
# Plot regression line
plot(test$medv, pr.nn_, col = "red",
      main = 'Real vs Predicted')
abline(0, 1, lwd = 2)
```

For classification task. We will first download the required packages as in the previous task. We also convert the type of flowers into factors.

```
library(tidyverse)
library(neuralnet)

iris <- iris %>% mutate_if(is.character, as.factor)
summary(iris)
```

We will set seed for reproducibility and split the data into train and test datasets for model training and evaluation.

```
set.seed(245)

sample(1:nrow(data), round(0.75 * nrow(data)))
train_indices <- sample(c(1:nrow(iris)), floor(0.75*nrow(iris)))
train_data <- iris[train_indices,]
test_data <- iris[-train_indices,]
```

We create two hidden layers, the first layer with four neurons and the second with two neurons.

```
model = neuralnet(
Species~Sepal.Length+Sepal.Width+Petal.Length+Petal.Width,
data=train_data,
hidden=c(4,2),
linear.output = FALSE
)
```

To view our model architecture, we will use the 'plot' function. It requires a model object and 'rep' argument.

```
plot(model,rep = "best")
```

We can now compute the performance of our model on the test data and compute the accuracy.

```
pred <- predict(model, test_data)
labels <- c("setosa", "versicolor", "virginica")
prediction_label <- data.frame(max.col(pred)) %>%
mutate(pred=labels[max.col.pred.]) %>%
select(2) %>%
unlist()

table(test_data$Species, prediction_label)
```

References

- [McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.
- [Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.