From Machine Learning to Deep Learning

Pierre-Marc Jodoin, Nicolas Duchateau, Christian Desrosiers

DOI = https://doi.org/10.1007/978-3-031-05071-8_3

3 From Machine Learning to Deep Learning

Prof Pierre-Marc Jodoin[a,b,*]

Dr Nicolas Duchateau[c,d]

Prof Christian Desrosiers[e]

[a] *Université de Sherbrooke, Canada.*

[b] *Imeka Solutions Inc., Sherbrooke, Canada.*

[c] *Univ Lyon, Université Claude Bernard Lyon 1, INSA-Lyon, CNRS, Inserm, CREATIS UMR 5220, U1294, F-69621, Lyon, France.*

[d] *Institut Universitaire de France (IUF), France.*

[e] *École de Technologie Supérieure, Montréal, Canada.*

[*] *Corresponding author.*

Authors' contribution:

- Main chapter: PJ, CD.
- Tutorial: ND, CD, PJ.

Abstract

This chapter provides a thorough grounding in the fundamental mathematical concepts of deep learning. It is first shown how a simple linear classifier can be defined based on the equation for a straight line. A more general scheme for optimization of the parameters of classifiers is introduced, based on gradient descent and its variants. We then see how the basic classifier model can be extended to produce simple artificial neural networks such as the perceptron and logistic regression. Next, these models are taken further to show how multiclass classification problems and nonlinearly separable data can be handled. Finally, the idea of a convolutional neural network is introduced and we see how this leads to the idea of deep learning. A practical tutorial is provided to give the reader some practical experience of developing classification models using Python.

Keywords:

deep learning, machine learning, classification, training, neuron, neural network, gradient descent, perceptron, sigmoid function, softmax, logistic regression, convolutional neural networks

Learning Objectives:

At the end of this chapter you should be able to:

- *O3.A Explain how the equation for a straight line leads naturally to the formulation of a simple binary classifier*
- *O3.B Describe the differences between the main gradient descent optimization algorithms*
- *O3.C Explain the operation of simple artificial neural networks such as the perceptron and the logistic regression*
- *O3.D Decide which approach to use to extend a machine learning model to classify data which are not linearly separable*
- *O3.E Explain the basic principles of convolutional neural networks (CNNs), and summarize their relevance to analyse medical images and signals*

Introduction

As mentioned in the previous chapter, a very active area of AI is *machine learning*, which accounts for mathematical models whose behaviour adapts to the data they are faced with. This adaptation process is called *learning*, an obscure term that we ought first to disambiguate.

In this chapter, we lay the mathematical foundations of supervised machine learning through a simple medical example. This example will soon lead to the notions of *classification function*, *training*, *neuron*, *neural network*, and *deep neural networks* as well as fundamental concepts associated to these notions.

The example goes as follows: for a few days, patients are showing up at a clinic to adjust their medication. Based on their symptoms, some patients need to extend their treatment while others, who had a successful reaction to the medication, are now healthy and may stop their treatment. This example includes two (and only two) classes of patients: the ones which are *sick* and those that are *healthy*. In this example, we assume that the status of a patient can be determined by the inspection of two characteristics: the body temperature in degrees Celsius and the heart rate in cycles/minute.

In a retrospective analysis, N patients were analysed and their information stored in a dataset that we shall call D. These characteristics can be visualized for the whole population using a scatter plot as shown in Figure 3.1a, in which each patient is represented by a point in a 2-D feature space. In mathematical terms, this translates into a dataset $D = \{(\boldsymbol{x}_1, t_1), (\boldsymbol{x}_2, t_2), \ldots, (\boldsymbol{x}_N, t_N)\}$ where $\boldsymbol{x}_i \in \mathbb{R}^2$ is a vector containing the body temperature and the heart rate of patient i while $t_i \in \{healthy, sick\}$ stands for the patient's status.

Technical Note
In the equations of this chapter, standard letters like i or t designate scalar variables. A bold letter in lower case, such as $\boldsymbol{x}$ or $\boldsymbol{w}$, stands for a vector, while a bold variable in upper case, such as $\boldsymbol{W}$, stands for a matrix. Also, a variable with a subscript indicates that it is a

single element of a set (e.g. $\boldsymbol{x}_i$ and t_i are variables of the i^{th} patient).

As one can see from Figure 3.1a, the sick patients are those with a fever and/or a high heart rate and so stand away from the healthy ones in the plot.

The goal of supervised learning is to learn a function $f : \mathbb{R}^2 \to \{healthy, sick\}$ which can correctly identify the status from the characteristics according to the data contained in D, i.e.

$$f(\boldsymbol{x}_i) = t_i, \quad \forall i \in \{1, \ldots, N\}. \tag{3.1}$$

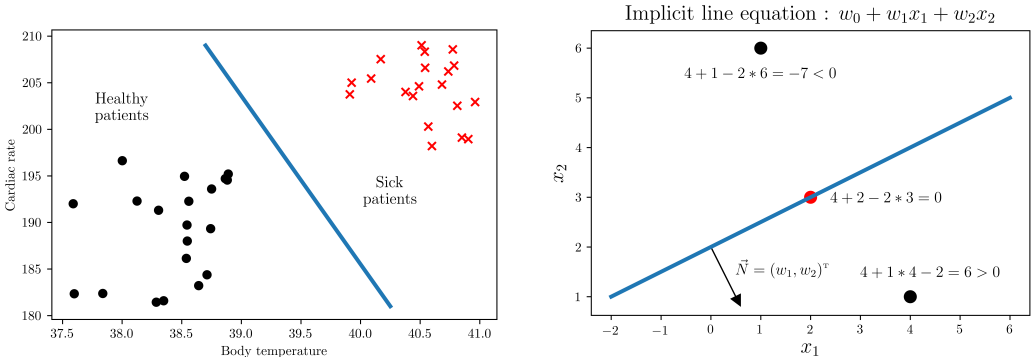
Put another way, for any given patient, the machine learning function f must be capable of converting a vector of characteristics $\boldsymbol{x}$ (in our example, *temperature* and *heart rate*) into a class label (in our example, *sick* or *healthy*). As such, the function $f(\boldsymbol{x})$ is called a *classification function*.

Technical Note
The notation $f : \mathbb{R}^2 \to \{healthy, sick\}$ means that we define a function f that *maps from* ($\to$) a vector of 2 real values ($\mathbb{R}^2$) to a single label which can take either of the values $\{healthy, sick\}$. We call $\mathbb{R}^2$ the *domain* of the function and $\{healthy, sick\}$ the *range*. The $\forall i \in \{1, \ldots, N\}$ in Eq. (3.1) means that this mapping should be correct *for all* ($\forall$) values of i between 1 and N.

Machine Learning and Neural Networks

If the distribution of the *sick* and *healthy* patients was known *a priori*[18], it would be easy for a programmer to write a deterministic algorithm that would satisfy Eq. (3.1). All one would have to do is determine which side of the line a patient lies in the feature space shown in Figure 3.1a to determine the status of that patient. However, for the sake of our example, we will

[18] i.e. before looking at the data



(a) (b)

Figure 3.1: (a) 2-D scatter plot of linearly separable *healthy* and *sick* patients. (b) Implicit equation of a line with its normal vector $\boldsymbol{\mathcal{N}}$, and examples of the line equation for three points.

assume that we do not know *a priori* that *sick* patients are those with a high body temperature and a high heart rate and the *healthy* patients are the other ones. Instead, we will implement a system for discovering automatically how to separate these patients by adjusting its parameters based on the content of D via a *training* procedure. Since D is at the core of the training procedure, it is usual to call it the *training set* (see Chapter 2, Model Validation, page 34).

But before we dive into the specifics of the training procedure, let us first consider what $f(\boldsymbol{x}_i)$ looks like mathematically.

Two-Class Prediction

Different machine learning approaches may lead to different types of classification function $f(\boldsymbol{x})$. However, one of the simplest and most widely used function is the *linear classifier*. In Figure 3.1a, we see that the sick and healthy patients can be separated by a straight line. That line is a symbolic representation of a linear classifier. To understand the mathematics behind a linear classifier, we shall go back to its very roots, i.e. the definition of a line.

One may recall from high school years that the equation of a line is given by

$$y = mx + b \tag{3.2}$$

where x and y are the horizontal and vertical coordinates of a 2-D point, m is the slope and b the intercept (i.e. the distance to the origin). This equation is known as the *explicit* formulation of a line (explicit because one variable is expressed in relation to the other).

One may also recall that the slope is given by a ratio: $m = \frac{\Delta y}{\Delta x}$. If we replace m by this ratio in Eq. (3.2) and then rearrange the terms, we get the following *implicit* formulation of a line,

$$0 = \Delta y \cdot x - \Delta x \cdot y + \Delta x \cdot b. \tag{3.3}$$

This formulation stipulates that each point (i.e. each pair of patient characteristics) located on the line satisfies this equation.

By renaming some variables, namely $\Delta y \to w_1, -\Delta x \to w_2$, and $\Delta x b \to w_0$, we get a less convoluted implicit formulation,

$$0 = w_1 x + w_2 y + w_0. \tag{3.4}$$

While x and y is a convenient naming convention for a two-dimensional space, it is far less convenient in a higher dimensional space. To illustrate this, another experiment could require more than two characteristics like cardiac EF, age, body mass index, blood sugar level, cholesterol level, etc. In that case, more characteristics would lead to more dimensions and more variables to name. As such, it is usual to rename x, y to x_1, x_2 where x_j stands for the j-th characteristic (in our case, x_1 is the *body temperature* and x_2 is the *heart rate*).

The resulting implicit line equation is as follows:

$$0 = w_1 x_1 + w_2 x_2 + w_0. \tag{3.5}$$

While different from $y = mx + b$, this equation is still the equation of a line. To convince ourselves, let us consider Figure 3.1b where the line parameters are $(w_0, w_1, w_2)^T = (4, 1, -2)^T$. If we take a point that falls on the line, say $(2, 3)$ and plug it into Eq. (3.5), we get $1 \times 2 - 2 \times 3 + 4$ which, indeed, equals zero.

Technical Note
The implicit equation of a line can be represented by the dot product between two vectors: the parameter vector $\boldsymbol{w} = (w_0, w_1, w_2)^T$ and the augmented characteristic vector $\boldsymbol{x}' = (1, x_1, x_2)^T$ where 1 is added to $\boldsymbol{x}$ to account for the bias. Assuming the column vector notation, the implicit line equation can be represented as

$$0 = \boldsymbol{w}^T \boldsymbol{x}' \tag{3.6}$$

where $.^T$ is the transpose operator. Please note that for the rest of this chapter, we will drop the prime and refer to $\boldsymbol{x}$ as an augmented vector.

Furthermore, one can prove that the vector $\boldsymbol{\mathcal{N}} = (w_1, w_2)^T$ is the normal of the line. Any point lying in the direction of the normal is said to be *in front* of the line while the other ones are said to be *behind* the line. As for w_0, it is the so-called *bias*, which is zero when the line crosses the origin.

Linear separation of a feature space: Things get interesting when we feed Eq. (3.5) with points that do not lay on the line. For example, if we take the point $(1, 6)$, located above the line in Figure 3.1b, we get $1 \times 1 - 2 \times 6 + 4 = -7$, i.e. a *negative* value. For the point $(4, 1)$, located below the line, we get $1 \times 4 - 2 \times 1 + 4 = 6$, a *positive* value.

This little experiment underlines an important fact. By its very nature, the implicit equation of a line separates the space into two regions: the region for which the line equation produces a *positive* value and the one for which it is *negative*.

Since our goal is to classify patients as being *healthy* or *sick*, we need a *binary classifier*. As such, we can use the following *sign* function,

$$sign(t) = \begin{cases} +1 & \text{if } t > 0 \\ -1 & \text{otherwise} \end{cases} \tag{3.7}$$

to convert the negative values into the -1 label and the positive scores into the $+1$ label (i.e. *healthy = +1* and *sick = -1* in our case). This leads to the following *binary classification function*:

$$f(\boldsymbol{x}) = sign(\boldsymbol{w}^T \boldsymbol{x}). \tag{3.8}$$

Training an AI model: At this point, the reader may contemplate the fact that a binary linear classification is no more than a dot product between a vector of parameters $\boldsymbol{w}$ and an augmented vector of characteristics $\boldsymbol{x}$. If the vector $\boldsymbol{w}$ has the right values (as in Figure 3.1a) the system can successfully separate the *sick* and the *healthy* patients by predicting the right positive and negative values.

In the context of our simple medical example, the process by which the parameter vector $\boldsymbol{w}$ is adjusted to the content of the dataset D (and thus fulfill the requirement of Eq. (3.1)) is called the *training* operation. The dataset D contains a series of pairs $(\boldsymbol{x}_i, t_i)$ and the goal of the classification function is to predict the right label t_i for a given feature vector $\boldsymbol{x}_i$.

In this section, we describe how $\boldsymbol{w}$ can be estimated from training data and how this process generalizes to more complex models such as the perceptron and logistic regression.

Training a model can only be done with the support of a function that can measure how *good* a set of parameters is at discriminating, for example between *healthy* and *sick* patients. This function is called the *loss function* and its value is typically proportional to the error rate of the model. As such, the goal of the training procedure is to estimate the parameter vector $\boldsymbol{w}$ which produces the lowest possible loss $\mathcal{L}(\boldsymbol{w})$. One way of illustrating this is through the plot of a 'loss landscape' as shown in Figure 3.2. This plot

shows the loss value (the vertical axis) for different values of parameters w_1 and w_2. The best pair of parameters is at the bottom of the trough in the blue area.

The training objective can thus be formulated as follows:

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) \tag{3.9}$$

which can be translated as: "out of every possible parameter vector $\boldsymbol{w}$, find one that minimizes the loss function $\mathcal{L}(\boldsymbol{w})$ and thus best classifies the data."

Technical Note
The goal of any machine learning algorithm is to find the parameter vector $\boldsymbol{w}$ that minimizes the loss $\mathcal{L}(\boldsymbol{w})$. The scientific field oriented towards the development of such algorithms is called *mathematical optimization*. While mathematical optimization is a broad field, we will focus on a specific optimization algorithm that is widely used to train neural networks: *gradient descent*.

A vector $\boldsymbol{w}$ minimizes the loss $\mathcal{L}(\boldsymbol{w})$ when its derivative with respect to its dimensions is zero:

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \mathbf{0} \tag{3.10}$$

The left-hand side of this equation is the *gradient of the loss* with respect to the parameter vector $\boldsymbol{w}$. Since we have two characteristics in our example (again, *body temperature* and *heart rate*), $\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w})$ is a vector containing two values and the previous equation can be reformulated as

$$\begin{pmatrix} \partial\mathcal{L}/\partial w_1 \\ \partial\mathcal{L}/\partial w_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{3.11}$$

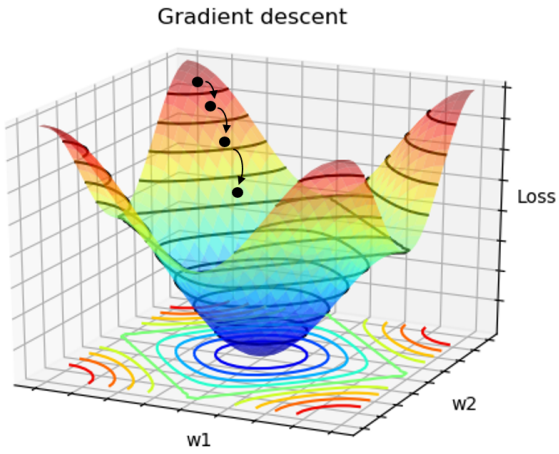


Figure 3.2: Illustration of a loss landscape of a space defined by two parameters w_1 and w_2. The black dots are the parameter values at 4 iterations of a gradient descent optimization. The arrows illustrate the opposite direction to the gradient.

Gradient descent is an iterative algorithm that minimizes the loss function by successively updating $\boldsymbol{w}$ in the opposite direction to the gradient. This is illustrated by the black dots in Fig 3.2 which iteratively go from a high loss down to a lower loss.

Since computing the exact gradient requires an infinite amount of pairs $(\boldsymbol{x}, t)$, one approach is to use the (entire) training dataset D to compute it, i.e.

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) \approx \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}; D) = \frac{1}{N} \sum_{(\boldsymbol{x}_i, t_i) \in D} \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, (\boldsymbol{x}_i, t_i)) \tag{3.12}$$

where N is the total number of patients in D, and $\sum_{(x_i,t_i)\in D}$ is a summation operation over every training data pair. Note that the notation $(w; D)$ is used to indicate that this version of the loss function is solely defined by the data contained in the training set, D. Using this gradient approximation based on the entire training set leads to the *batch gradient descent* algorithm (see Algorithm 1). In this case, we say that the parameter vector $\boldsymbol{w}$ is updated once every *epoch*.

Technical Note
In machine learning, the term *epoch* refers to the optimization algorithm seeing all data in the training set once. An *iteration* is one update of the learning parameters, and an iteration considers a *mini-batch* of the training data (which has a *batch size*). Therefore, the size of the training set is equal to the batch size multiplied by the number of iterations in an epoch. In *batch gradient descent*, the batch size is the entire training set, so there is one iteration in an epoch.

Unfortunately, for various technical reasons, batch gradient descent is prohibitively slow and memory intensive. An alternative strategy, known as *stochastic gradient descent*, (see Algorithm 2) is to approximate the gradient and update the parameters $\boldsymbol{w}$ for each *training sample*, i. Thus, there are N *iterations* per *epoch* for a stochastic gradient descent (where N is the training set size). Another slight variant use bundles of data pairs (*mini-matches*) to approximate the gradient. This would give rise to a *mini-batch stochastic gradient descent* algorithm.

In all cases, the change to the model parameters is weighted by a learning rate η, which is a predefined constant, typically between 0 and 1. η is a famous hyperparameter that one must determine, for example through a cross validation procedure[19].

Algorithm 1: Batch gradient descent algorithm

input : Training set D, learning rate η
output: Trained weights $\boldsymbol{w}$

Init $\boldsymbol{w}$ with [small] random values;
for *epoch = 1 to* $epoch_{MAX}$ **do**
  $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, D)$;
end

The first machine learning model - the perceptron:

[19] Recall our discussion of the terms 'hyperparameter' and 'cross validation' in Chapter 2, Model Validation, page 34.

Algorithm 2: Stochastic gradient descent algorithm

input : Training set D, learning rate η
output: Trained weights $\boldsymbol{w}$

Init $\boldsymbol{w}$ with [small] random values;
for *epoch = 1 to* $epoch_{MAX}$ **do**
  for $(\boldsymbol{x}_i, t_i) \in D$ **do**
    $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, (\boldsymbol{x}_i, t_i))$;
  end
end

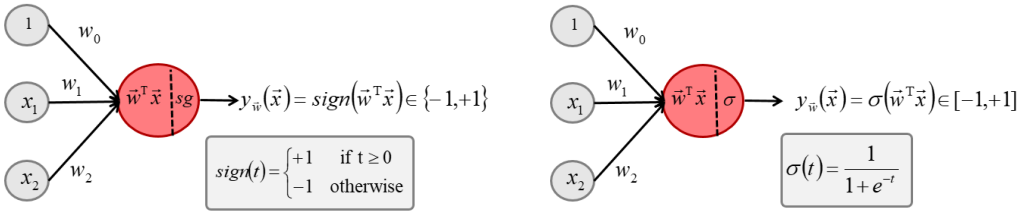
The perceptron is often cited as the oldest machine learning model [333]. It is a binary classifier that implements the classification function of Eq. (3.8). The perceptron is typically illustrated by a graph such as the one shown in Figure 3.3a. In this illustration, each grey circle on the left embodies an input variable. Since our example has two characteristics (*body temperature* and *heart rate*) the perceptron has two input variables and a 1 for the bias.

Each input variable has a connecting arrow showing the direction of the flow of information, and each arrow has an associated weight value $w_j \in \mathbb{R}$. These arrows connect to a red circle, which embodies two crucial operations:

1. The dot product between the input vector $\boldsymbol{x}$ and the weights $\boldsymbol{w}$. As mentioned before, this operation is the implicit formulation of a linear function (e.g. a line in 2-D or a plane in 3-D) which, by its very nature, linearly separates the feature space into two regions.
2. The *sign* function, which converts the result of the dot product to one of the two values $\{-1, 1\}$, as mentioned before. This non-linear function is referred to as an *activation function*.

Technical Note
The red circle in Figure 3.3a is a so-called *artificial neuron*. Such an artificial neuron is thus nothing more than a *dot product* followed by a *non-linear activation function*. This gives rise to the concept of an *artificial neural network*, which is a network of such artificial neurons.

Like any machine learning model, the perceptron has a loss function which is based on two properties of the implicit line equation (i.e. the dot product $\boldsymbol{w}^T\boldsymbol{x}$). First, considering that the target t can be either $+1$ or -1 (in our example : *sick* : $+1$ and *healthy* : -1), the dot product of a misclassified point has an opposite sign to its associated target value. In other words, a *sick* patient with target $t = +1$ is misclassified when $\boldsymbol{w}^T\boldsymbol{x} < 0$. Second, the more misclassified a point is, the larger will be the magnitude of its dot product. This is explained by the fact that the distance between a point and the line is related to the magnitude of the dot product.



(a) (b)

Figure 3.3: Graphical representation of (a) the perceptron, and (b) logistic regression. These are the simplest types of neural network and both are constructed using a single artificial neuron. On the left are the input variables (and 1 for the bias). The red circles represent artificial neurons that compute a dot product followed by an activation function (*sign* for the perceptron and σ for logistic regression).

With these two properties in mind, the *perceptron loss* is as follows,

$$\mathcal{L}(\boldsymbol{w}, D) = \frac{1}{M} \sum_{(\boldsymbol{x}_i, t_i) \in \mathcal{M}} -t_i \boldsymbol{w}^T \boldsymbol{x}_i \tag{3.13}$$

where $\mathcal{M}$ is the set of misclassified samples, of size M. Note that the perceptron loss increases with the amount of misclassified samples. On the other hand, the perceptron loss reaches zero when every point $(\boldsymbol{x}_i, t_i) \in D$ is correctly classified and hence $\mathcal{M}$ is empty.

Technical Note
The perceptron loss being linear with respect to $\boldsymbol{w}$, its batch gradient is given by

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, D) = \frac{1}{M} \sum_{(\boldsymbol{x}_i, t_i) \in \mathcal{M}} -t_i \boldsymbol{x}_i \tag{3.14}$$

and its gradient for one misclassified sample $(\boldsymbol{x}_i, t_i)$ is given by

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, (\boldsymbol{x}_i, t_i)) = -t_i \boldsymbol{x}_i. \tag{3.15}$$

One can therefore plug these two equations into Algorithms 1 and 2 to train the network.

Logistic regression: Like the perceptron, the logistic regression is a linear classifier. As shown in Figure 3.3b, it can be viewed as a simple artificial neural network with a dot product between the parameters $\boldsymbol{w}$ and the input vector $\boldsymbol{x}$. However, the logistic regression network has a different activation function called a *sigmoid*. The sigmoid $\sigma : \mathbb{R} \to [0, 1]$ is a mathematical function defined as follows:

$$\sigma(t) = \frac{1}{1 + e^{-t}} \tag{3.16}$$

where $\sigma(t = 0) = 0.5$, $\sigma(t \gg 0) \to 1$ and $\sigma(t \ll 0) \to 0$.[20] A plot of the sigmoid function is shown in Figure 3.4.

The sigmoid is an appealing activation function when considered in conjunction with the properties of the implicit line equation, i.e. the dot product of

[20] The right arrows ($\to$) here mean that the output of the sigmoid function *approaches* 1 when t is very positive and 0 when it is very negative. Note that this is not the same use of $\to$ that we saw earlier when defining the domain and range of functions (see Technical Note, page 54).

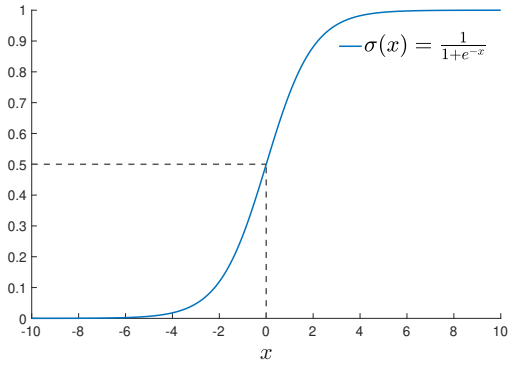


Figure 3.4: Plot of a sigmoid activation function. This function returns a value between 0 and 1 and a value of 0.5 at the origin $t = 0$.

Eq. (3.6). As mentioned before, while the dot product of a point lying on the line is zero, the same dot product of a point located *in front* of the line is positive and for a point *behind* the line it is negative. Thus, for a logistic regression, a point lying on the line will have a score of 0.5 whereas a point in front of the line will have a score larger than 0.5 and behind the line a score lower than 0.5. Moreover, a point located far in front of the line will have a score ≈ 1 while a point far behind the line will have a score ≈ 0.

Technical Note
While the use of a sigmoid activation function does not seem to bring much, it has nonetheless tremendous consequences. In fact, it turns the neural network into a machine capable of predicting the conditional probability of class C_1: $P(C_1|\boldsymbol{x})$. This conditional probability can be translated into English as: "the probability of being in class C_1 (the class *sick* in our example) given the input vector $\boldsymbol{x}$". Put another way, when properly trained, a logistic regression neural network has the sole property of predicting the probability of being in class C_1 (and by default class C_0 since $P(C_0|\boldsymbol{x}) + P(C_1|\boldsymbol{x}) = 1$) given the input vector $\boldsymbol{x}$ it receives as input.

In this way, a point lying in front of the line will have a large probability of being in class C_1, a point behind the line will have a low probability of

being in class C_1 (and thus a high probability of being in class C_0) and a point on the line will have a 50% chance of being in class C_1. This is why the sigmoid activation function (as well as the *softmax* function that we will soon introduce) is widely used at the end of classification and segmentation neural networks.

The loss function of the logistic regression network is the well-known *cross entropy* loss:

$$\mathcal{L}(\boldsymbol{w}, D) = -\frac{1}{N} \sum_{(\boldsymbol{x}_i, t_i) \in D} t_i \ln(y_{\boldsymbol{w}}(\boldsymbol{x}_i)) + (1 - t_i) \ln(1 - y_{\boldsymbol{w}}(\boldsymbol{x}_i)) \tag{3.17}$$

where N is the total number of patients in the training dataset D and $t_i = \{0, 1\}$ (instead of $\{-1, +1\}$ for the perceptron). According to this function, the loss is minimum when the output of the network $y_{\boldsymbol{w}}(\boldsymbol{x}_i) = t_i$. In other words, the cross entropy loss is close to zero when the network correctly classifies the samples, meaning a conditional probability close to 1 when $t_i = 1$ and close to 0 when $t_i = 0$.

Technical Note
Considering that $y_{\boldsymbol{w}}(\boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{x})$, one can prove that the batch gradient of the loss with respect to $\boldsymbol{w}$ is

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, D) = \frac{1}{N} \sum_{(\boldsymbol{x}_i, t_i) \in D} (y_{\boldsymbol{w}}(\boldsymbol{x}_i) - t_i) \boldsymbol{x}_i \tag{3.18}$$

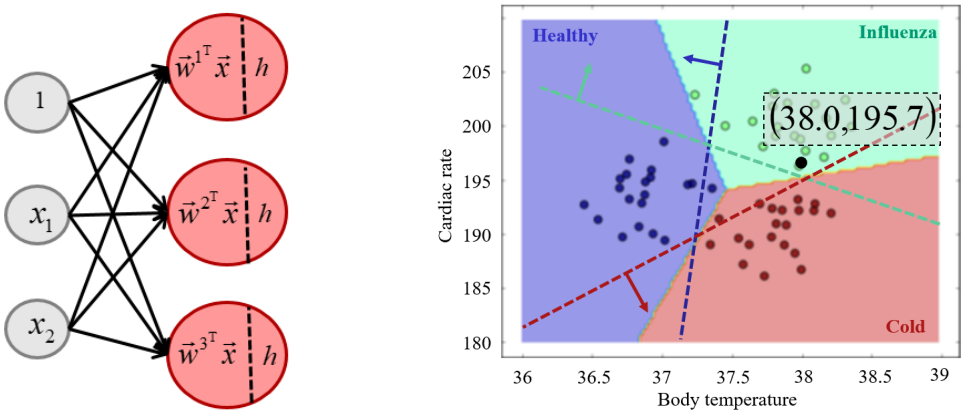
and the gradient for one data pair $(\boldsymbol{x}_i, t_i)$ is

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, (\boldsymbol{x}_i, t_i)) = (y_{\boldsymbol{w}}(\boldsymbol{x}_i) - t_i) \boldsymbol{x}_i. \tag{3.19}$$

Again, Algorithms 1 and 2 can be used with these gradient equations to train the logistic regression neural network.

K-Class Prediction

So far, we have studied a two-class example whose goal was to separate the *sick* patients from the *healthy* ones. Obviously, one can imagine classification problems with more than two classes, such as for example: *influenza*, *cold*, and *healthy*.



(a) (b)

Figure 3.5: (a) Three-class linear neural network. (b) Scatter plots of patients associated with three classes: *Healthy*, *Cold*, *Influenza*. The dotted lines are the linear functions of each class. Note that the activation function h will vary depending on the nature of the loss. The point $(38.0, 195.7)$ contains the body temperature and heart rate of a patient suffering from Influenza. Note that the values reported in this plot are for illustrative purposes only.

Fortunately, neural networks naturally scale to the number of classes. When the number of classes is larger than 2, one can can simply use K output neurons (the red neurons in Figure 3.5a), where K is the number of classes. This gives rise to the *multi-class perceptron* and *multi-class logistic regression networks*.

Like the neurons we have seen before, these output neurons perform a dot product on the input vector $\boldsymbol{x}$. Like any dot product, these neurons linearly separate the feature space.

Technical Note
The output of a three-class neural network is thus a vector of three dot products that can be expressed as a matrix-vector product:

$$\begin{bmatrix} \boldsymbol{w}^{1^T}\boldsymbol{x} \\ \boldsymbol{w}^{2^T}\boldsymbol{x} \\ \boldsymbol{w}^{3^T}\boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}^{1^T} \\ \boldsymbol{w}^{2^T} \\ \boldsymbol{w}^{3^T} \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} w_0^1 & w_1^1 & w_2^1 \\ w_0^2 & w_1^2 & w_2^2 \\ w_0^3 & w_1^3 & w_2^3 \end{bmatrix} \boldsymbol{x} = \boldsymbol{W}\boldsymbol{x} \qquad (3.20)$$

As can be seen, the i^{th} row of matrix $\boldsymbol{W}$ contains the parameters of the i^{th} classifier.

For the neural network shown in Figure 3.5a, the linear functions of the three output neurons are illustrated by the dotted lines in Figure 3.5b. For the multi-class perceptron, the output neurons have no activation function (thus h in Figure 3.5a is an identity function). Instead, the class predicted by the model is the one with the largest score. To illustrate this, let us consider the three-class example of Figure 3.5b. Here, we have a feature point $\boldsymbol{x} = (38, 195.7)^T$ which corresponds to a patient whose body temperature is 38 degrees Celsius and heart rate is 195.7 beats per minute. This point lies in the green section of the space, i.e. the area associated with class 3: *Influenza*. If we use the nine parameters of the system to form matrix $\boldsymbol{W}$ and multiply this by the augmented vector $\boldsymbol{x}$ we get

$$\begin{bmatrix} 1057.5 & -31 & 0.5 \\ -213 & 21 & -3 \\ -831.0 & 9 & 2.5 \end{bmatrix} \begin{bmatrix} 1 \\ 38 \\ 195.7 \end{bmatrix} = \begin{bmatrix} -22.7 \\ -2.1 \\ 0.25 \end{bmatrix} \qquad (3.21)$$

i.e. a *negative* value for classes 1 and 2 because $\boldsymbol{x}$ is located *behind* the blue and red dotted lines and a *positive* value for class 3 because it is located *in front* of the 3$^{\text{rd}}$ separation line.

The multi-class perceptron loss is given by,

$$\mathcal{L}(\boldsymbol{W}, D) = \frac{1}{M} \sum_{(x_i,t_i)\in\mathcal{M}} (\boldsymbol{w}^{j^T} \boldsymbol{x}_i - \boldsymbol{w}^{t_i^T} \boldsymbol{x}_i) \tag{3.22}$$

where j is the wrongly predicted class index and t_i the target class index. Here again, the loss reaches zero when every training sample is well classified.

Technical Note
The stochastic gradient of the multi-class perceptron loss for a pair of misclassified samples $(x_i, t_i) \in \mathcal{M}$ is given by

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, (\boldsymbol{x}_i, t_i)) : \begin{cases} \frac{\partial L}{\partial \boldsymbol{w}^j} = \boldsymbol{x}_i \\ \\ \frac{\partial L}{\partial \boldsymbol{w}^{t_i}} = -\boldsymbol{x}_i \end{cases} \tag{3.23}$$

and the batch gradient is obtained by averaging these partial derivatives across $\mathcal{M}$.

The multi-class logistic network is very similar to the multi-class perceptron in the sense that the output neurons embed a dot product. However, in place of the sigmoid activation function, the output layer is followed by a *softmax* operation which is a normalized exponential function. If we call f_i the output of the i^{th} neuron (in Eq. (3.21), $f_1 = -22.7, f_2 = -2.1, f_3 = 0.25$) the output of the softmax function for that neuron is,

$$S_i = \frac{e^{f_i}}{\sum_{k=1}^{K} e^{f_k}}. \tag{3.24}$$

If we apply the softmax operation to the output values of Eq. (3.21), we get $\boldsymbol{S} = (0.0, 0.087, 0.913)^T$.

As for the two-class logistic network, this output can be seen as the conditional probability $P(C_i|\boldsymbol{x})$. Put another way, according to the output $\boldsymbol{S}$, data

$\boldsymbol{x}$ has 91.3% chance of belonging to class 3 and a 8.3% chance of belonging to class 2.

The loss of the multi-class logistic network is also a cross entropy,

$$\mathcal{L}(W, D) = -\frac{1}{N} \sum_{(x_i, t_i) \in D} \ln S_{t_i}. \tag{3.25}$$

where S_{t_i} is the probability (or the softmax) of the correct class.

Technical Note
The batch gradient of the cross entropy loss with respect to the weights W is given by,

$$\nabla_W \mathcal{L}(W, D) = \frac{1}{N} \sum_{(x_i, t_i) \in D} (\boldsymbol{S}_i - \boldsymbol{t}_i) \boldsymbol{x}_i^T. \tag{3.26}$$

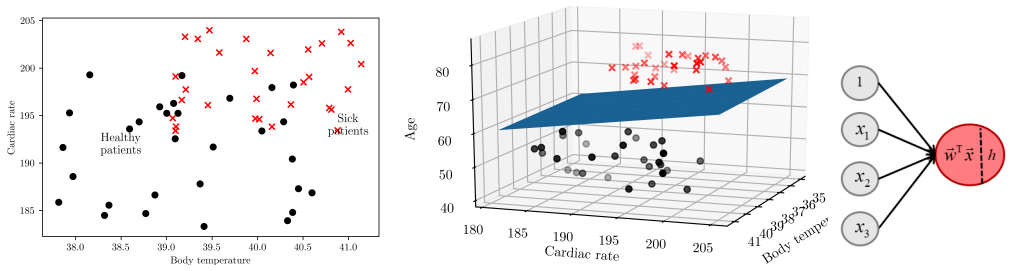
Handling Non-linearly Separable Data

Linear decision functions such as those we have seen so far work well for well separated subgroups. However, it often happens that subgroups cannot be separated by a linear function, as illustrated in Figure 3.6a. These problems require more sophisticated and complex solutions.

To tackle the problem of non-linearly separable data, three approaches are available:

1. Using a *non-linear decision function*.
2. Gathering *more information*.
3. *Transforming* the data.

While the first solution goes beyond the scope of this chapter, we will focus on the latter two solutions and underline how they fit within the scope of neural networks.



(a) (b) (c)

Figure 3.6: (a) Example of non-linearly separable 2-D data, and (b) its augmented version with a third dimension (age) with a 3-D plane separating the two groups of patients. (c) A 3-D plane can be mathematically represented by a neural network with four input variables.

Gather more information: In the context of our example, non-linearly separable data means that *body temperature* and *heart rate* measurements are not discriminative enough to separate the two classes with a linear classifier. As a solution, one might acquire a third measurement such as, for example the *age*. By doing so, $\boldsymbol{x} \in \mathbb{R}^3$ becomes a point in a 3-D space (see Figure 3.6b) and the classification function becomes a plane defined in this 3-D space. Interestingly, the implicit equation of a plane is a generalization of the equation of a line with a third dimension x_3,

$$0 = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_0 \tag{3.27}$$

where w_0 is still the bias. As for the implicit equation of a line, this equation can be represented by a dot product $0 = \boldsymbol{w}^T \boldsymbol{x}$ where $\boldsymbol{w}, \boldsymbol{x} \in \mathbb{R}^4$.

Without much surprise, we can further generalize the formulation to q measurements. In that case, a patient would become a point in a q-dimensional space where the *sick* and the *healthy* patients can be separated by a *hyperplane*. This hyperplane is again represented by a dot product $0 = \boldsymbol{w}\boldsymbol{x}$ where $\boldsymbol{w}, \boldsymbol{x} \in \mathbb{R}^{q+1}$. As for the implicit line equation, the implicit hyperplane equation has the sole property of splitting the feature space between a positive region located in front of the hyperplane and a negative region behind the hyperplane.

As shown in Figure 3.6c, the use of 3 input variables (and 1 for the bias) does not change the nature of the perceptron nor the logistic regression neural network as it only increases the size of the input layer.

Transform the data: Another well known solution to address the non-linearity issue is to project $\boldsymbol{x}$ into a new feature space where the data are linearly separable. The functions that perform this mapping are called *basis functions*, $\phi(\boldsymbol{x}) : \mathbb{R}^q \rightarrow \mathbb{R}^p$. Once the data are projected to the new feature space, a linear classifier can be used to separate the classes.

> **Technical Note**
> The use of a basis function also gave rise to the *kernel methods* (and the iconic *kernel SVM*[210]) which arguably were the most widely-used machine learning methods before the deep learning wave struck the scientific community [204].

One important limitation of basis functions is that not every function $\phi(.)$ (or its associated kernel $k(.)$ [210]) can successfully disambiguate two classes. As such, one often has to manually adjust $\phi(.)$ (or $k(.)$) to make it fit the training data distribution.

One great advantage of neural networks is their ability to simultaneously learn the basis function $\phi(.)$ as well as its associated classification function. This can be done by increasing the number of neurons. Figure 3.7 shows one such neural network organized into 4 layers, namely, the *input layer* (which corresponds to the grey circles on the left), the *hidden layers* (the yellow neurons in the middle) and the *output layer* (the red neurons). As before, the neurons encode a dot product between the output of the previous layer and an associated weight vector. This architecture is called a *multi-layer perceptron* (MLP).. The more hidden layers a MLP has, the more complex the overall neural network will be, i.e. the better it will be at estimating complex relationships between input samples $\boldsymbol{x}$ and target values t_i.

As before, the output layer of the MLP can be neurons without an activation function in which case the loss would the multi-class perceptron loss of Eq. (3.22). One could also add a softmax operation at the end of the network and get the cross entropy loss of Eq. (3.25). Note that the gradient of the loss

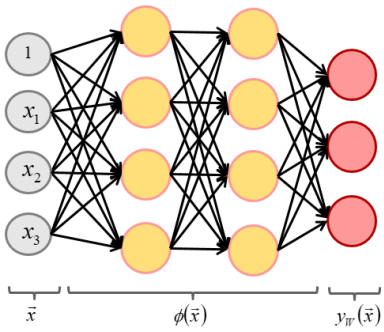


Figure 3.7: Multi-layer neural network made of an input layer (the four grey circles on the left) followed by two hidden layers (the yellow neurons) and an output layer (the red neurons). Mathematically, the purpose of the hidden layers is to act as a basis function $\phi(\boldsymbol{x})$ that projects the input data (here points in a 3-D space) into a linearly separable space.

with respect to the parameters of a multi-layer perceptron is done though an operation called *back propagation*. For more details on back propagation, please refer to [54].

The use of multiple layers of neurons leads to so-called *deep neural networks* and *deep learning*. There you have it! The more layers a neural network has, the deeper it gets.

Technical Note
As shown in Figure 3.7, a multi-layer neural network can be seen as a three-part machine:

1. An *input vector* $\boldsymbol{x}$.
2. A series of hidden layers, which act as a *basis function* $\phi(\boldsymbol{x})$ whose goal is to project $\boldsymbol{x}$ into a space where the data are linearly separable.
3. An output layer which is a *linear classifier*.

Since the weights $\boldsymbol{W}$ of the network are learned all together, we say

that deep neural networks are *end-to-end trainable* since $\phi(\boldsymbol{x})$ and the classification function are learned at the same time.

Convolutional Neural Networks

Multi-layer neural networks are not without their limitations. One of the most important limitations comes from the substantial increase in parameters when the size of the input vector increases. For example, if the input signal is a greyscale image containing 28×28 pixels (as for images from the iconic MNIST dataset [224]) each neuron of the first layer will be connected to a total of $28 \times 28 + 1$ input neurons (1 is for the bias). Therefore, if the first hidden layer has 100 neurons, the network will have $78,500$ parameters only in the first layer ($(28 \times 28 + 1) \times 100$). Even worse, if the input is, for example, a 3-D $256 \times 256 \times 256$ MR brain volume, the first layer will contain more than 16 million parameters. Without much surprise, very large neural networks pose important memory and computing challenges. Furthermore, it is empirically known that very large multi-layer networks are difficult to train, and often converge towards sub-optimal solutions.

The answer to this problem is to reduce the number of connections between two consecutive layers. While this is fundamentally difficult for an arbitrary input signal, there is an appealing solution when the input signal is temporally and/or spatially structured such as for an audio signal (1-D), a greyscale or color image (2-D) or a 3-D or 4-D medical image volume. In these cases, one can connect a neuron to a subset of neighboring neurons in the previous layer. This is illustrated in Figure 3.8 where each neuron in the first layer is connected to 3×3 grid of input nodes (here representing pixels). In this way, each neuron has a total of 9 weights instead of the very large number we would have with a fully-connected layer. The set of 3×3 weights that connects a neuron to the previous layer is called a *filter*. Furthermore, the "images" in the middle and on the right illustrate the output of each neuron of the first and second hidden layers. These "images" are called *feature maps*. As usual, these artificial neurons perform a dot product followed by an activation function.

One may reduce even more the number of parameters by forcing every filter of a layer to share the same set of weights. By doing so, the two hidden layers in Figure 3.8 would have a total of just 9 weights each. Even more

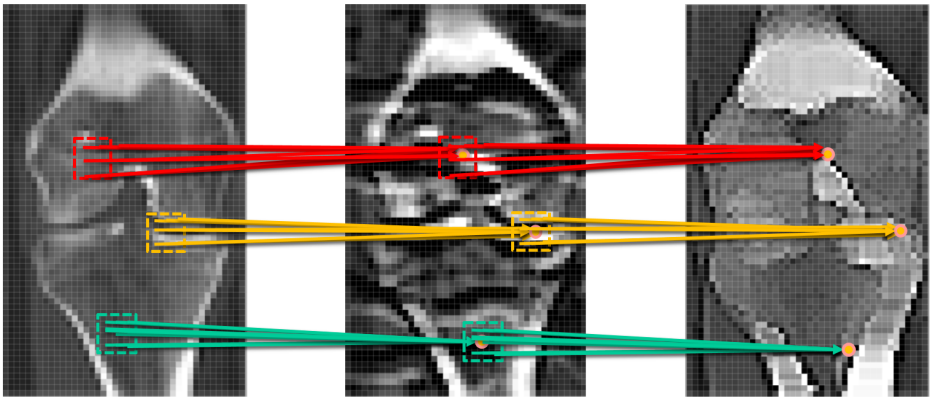


Figure 3.8: Illustration of two convolutional layers. On the left is the input layer containing a CT scan of a knee. In the middle and on the right are the 'feature maps' of the first and second hidden layers. Each element of these feature maps is an artificial neuron which embodies a dot product and a non-linear activation function. In this illustration, the feature maps show the neuron outputs. Each neuron is connected to a 3×3 grid of neurons in the previous layer and hence has 9 weights.

interesting, by doing so the number of weights is constant with respect to the size of the input signal.

> **Technical Note**
> By connecting the neurons as in Figure 3.8 and sharing the weights across a layer, the dot product computed for each neuron of a hidden layer is mathematically identical to that of a *convolution*, hence why we call these network layers *convolution layers*. These types of neural networks are called *convolutional neural networks* or CNNs for short.

Like the multi-layer neural networks that we have seen before, a CNN may have an arbitrary number of layers. Furthermore, $K > 1$ filters can be used at each layer which would produce K feature maps (Figure 3.8 shows one filter and one feature map per layer). Furthermore, like any other neural network, the number of neurons in the last layer corresponds to the number of classes (or variables in case of a regression) it should predict. CNNs are also trained with the same gradient descent and loss functions as any other neural network.

CNNs are the cornerstone of the machine learning revolution in medical imaging. Applications such as medical image reconstruction, denoising, disease recognition, tumor localization, and tissue segmentation to name a few are all intimately tied to CNNs. Further details on specific types of CNN are provided in Chapter 4.

Closing Remarks

In this chapter we have introduced the fundamental concepts of artificial neural networks, and seen how much of the formulation of such networks is based upon a simple linear algebra operation, i.e. the dot product. We have seen how neural networks can vary from the very simple (perceptron and logistic regression) to the more complex deep neural networks that are so widespread in cardiology and other fields of medicine today. Next, we include a set of self-assessment exercises to help you reinforce your knowledge of these fundamentals. Having built up our knowledge of the key concepts of machine and deep learning, the subsequent chapters provide more focused reviews of

specific topics in cardiology and the ways in which AI has, and will, impact these fields.

Exercises

Exercise 1.
Explain the meanings of the terms 'iteration' and 'epoch' in the content of machine learning optimization. How does the choice of batch size affect the relationship between epochs and iterations?

Exercise 2.
What are the main differences between the batch gradient descent, stochastic gradient descent and mini-batch stochastic gradient descent optimization algorithms? What is the main disadvantage of batch gradient descent?

Exercise 3.
What are the similarities and most fundamental difference between the perceptron and logistic regression artificial neural networks?

Exercise 4.
Describe three ways in which machine learning models can be extended from classifying linearly separable data to non-linearly separable data.

Exercise 5.
You have been asked to design a machine learning solution for analysing 3-D medical images and producing automated diagnoses. It is likely that the mapping from images to diagnoses is highly complex, but a large amount of training data are available to learn this mapping. Suggest which type of machine learning model might be appropriate for this application and justify your answer.

Tutorial - Classification From Linear to Non-linear Models

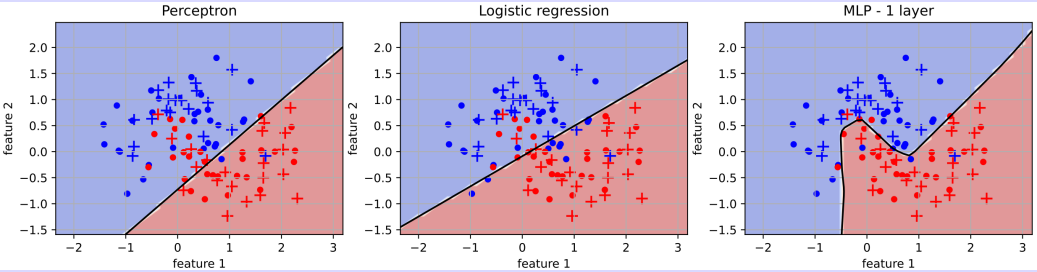
Tutorial 2.

As for the other notebooks, the contents of this notebook are accessible as Electronic Supplementary Material.

Overview

In this hands-on tutorial, you will test some of the concepts introduced in Chapter 3, in particular simple classifiers such as the perceptron, logistic regression and multi-layer perceptron. You will examine the performance of different classifiers and the effects of hyperparameters on two synthetic datasets, which are either linearly or non-linearly separable.

The figure below shows the output of three simple classifiers on non-linearly separable data, to be tested in this notebook:



Objectives

- *Become more familiar with Python and the essential tools for machine learning such as* `scikit-learn`.
- *Conduct a simple classification problem by testing progressively the contents described in Chapter 3.*

Computing Requirements

As for the other hands-on tutorials, this notebook starts with a brief "System setting" section, which imports the necessary packages, installs the potentially missing ones, and imports our own modules.

Acknowledgements

ND was supported by the French ANR (LABEX PRIMES of Univ. Lyon [ANR-11-LABX-0063] within the program "Investissements d'Avenir" [ANR-11-IDEX-0007], and the JCJC project "MIC-MAC" [ANR-19-CE45-0005]).