Contents

1. Introduction

1.1. Logistic regression

The way in which a multilayer perceptron approximates a given function bares striking similarity to the **method of least squares**, also known as **regression**, which is used to determine the polynomial function that best approximates the relationship between variables in a dataset.

Let $(X, Y) = \{(x_1, y_1), ..., (x_n, y_n)\}$ be a dataset of n points. Suppose that the relationship between X and Y can be approximated reasonably well by a straight line in the form $y = a + bx$, meaning that $y_i \approx a + bx_i$ for any (x_i, y_i). The straight line $y = a + bx$ is also called the **regression line**.

Let y_i be the true value for the Y variable for the i-th element, and let $\hat{y}_i = a + bx_i$ be the estimated value for the Y variable employing a straight line of parameters a and b. The error of approximation between y_i and $\hat{y}_i$ is given by the distance between the two points on the cartesian plane.

This distance can be quantified by the squared difference of the two quantities: $(\hat{y}_i - y_i)^2$. The interest is to have this distance minimized across the entire dataset, which means that the sum of all such distances:

$$F(a, b) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 = \sum_{i=1}^{n} (a + bx_i - y_i)^2$$

Should be as small as possible. Taking the partial derivative of $F(a, b)$ with respect to a:

$$\frac{\partial F}{\partial a} F(a, b) = \frac{\partial F}{\partial a} \sum_{i=1}^{n} (a + bx_i - y_i)^2 = \sum_{i=1}^{n} \frac{\partial F}{\partial a} (a + bx_i - y_i)^2 = \sum_{i=1}^{n} 2(a + bx_i - y_i) \frac{\partial F}{\partial a} (a + bx_i - y_i) =$$

$$= 2 \sum_{i=1}^{n} (a + bx_i - y_i) \left(\frac{\partial F}{\partial a} a + \frac{\partial F}{\partial a} bx_i - \frac{\partial F}{\partial a} y_i \right) = 2 \sum_{i=1}^{n} a + bx_i - y_i$$

And with respect to b:

$$\frac{\partial F}{\partial b} F(a, b) = \frac{\partial F}{\partial b} \sum_{i=1}^{n} (a + bx_i - y_i)^2 = \sum_{i=1}^{n} \frac{\partial F}{\partial b} (a + bx_i - y_i)^2 = \sum_{i=1}^{n} 2(a + bx_i - y_i) \frac{\partial F}{\partial b} (a + bx_i - y_i) =$$

$$= 2 \sum_{i=1}^{n} (a + bx_i - y_i) \left(\frac{\partial F}{\partial b} a + \frac{\partial F}{\partial b} bx_i - \frac{\partial F}{\partial b} y_i \right) = 2 \sum_{i=1}^{n} (a + bx_i - y_i) x_i$$

Setting them equal to 0 and rearranging the expressions:

$$\cancel{2} \sum_{i=1}^{n} a + bx_i - y_i = 0 \Rightarrow \sum_{i=1}^{n} a + \sum_{i=1}^{n} bx_i - \sum_{i=1}^{n} y_i = 0 \Rightarrow na + b \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$$

$$\cancel{2} \sum_{i=1}^{n} (a + bx_i - y_i) x_i = 0 \Rightarrow \sum_{i=1}^{n} ax_i + \sum_{i=1}^{n} bx_i^2 - \sum_{i=1}^{n} x_i y_i = 0 \Rightarrow a \sum_{i=1}^{n} x_i + b \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} x_i y_i$$

Allows one to retrieve the so-called **normal equations**, a linear equation system with two equations and two unknowns a and b:

$$na + b \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \qquad\qquad a \sum_{i=1}^{n} x_i + b \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} x_i y_i$$

The system has exactly one solution as long as all points do not lie on the same line.

The same approach can be extended to the case of approximating functions that aren't straight lines, but a polynomial of arbitrary degree m.

Suppose that a dataset $(X, Y) = \{(x_1, y_1), ..., (x_n, y_n)\}$ of n points has its relationship well approximated by a m degree **regression polynomial** $y = a_0 + a_1 x + ... + a_m x^m$, meaning that $y_i \approx a_0 + a_1 x_i + ... + a_m x_i^m$ for any (x_i, y_i). The error can be quantified as always by the sum of square differences:

$$F(a_0, a_1, ..., a_m) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 = \sum_{i=1}^{n} (a_0 + a_1 x_i + ... + a_m x_i^m - y_i)^2$$

Setting all partial derivatives equal to 0:

$$\frac{\partial F}{\partial a_0} F(a_0, a_1, ..., a_m) = 0 \qquad \frac{\partial F}{\partial a_1} F(a_0, a_1, ..., a_m) = 0 \qquad \cdots \qquad \frac{\partial F}{\partial a_m} F(a_0, a_1, ..., a_m) = 0$$

Rearranging, one obtains n equations in n unkowns:

$$\begin{aligned} n a_0 + a_1 \sum_{i=1}^{n} x_i + ... + a_m \sum_{i=1}^{n} x_i^m &= \sum_{i=1}^{n} y_i \\ n a_0 \sum_{i=1}^{n} x_i + a_1 \sum_{i=1}^{n} x_i^2 + ... + a_m \sum_{i=1}^{n} x_i^{m+1} &= \sum_{i=1}^{n} x_i y_i \\ &\vdots \\ n a_0 \sum_{i=1}^{n} x_i^m + a_1 \sum_{i=1}^{n} x_i^{m+1} + ... + a_m \sum_{i=1}^{n} x_i^{2m} &= \sum_{i=1}^{n} x_i^m y_i \end{aligned}$$

The system has exactly one solution as long as all points do not lie on the same polynomial of degree smaller or equal than m.

The approach can be extended to the case of finding a regression line for a function of any arity. Suppose that a dataset

$$(X_1, ..., X_m, Y) = \left\{ \left(x_{1,1}, x_{2,1}, ..., x_{m,1}, y_1\right), \left(x_{1,2}, x_{2,2}, ..., x_{m,2}, y_2\right), ..., \left(x_{1,n}, x_{2,n}, ..., x_{m,n}, y_n\right) \right\}$$

of n m-dimensional points has the relationship between $X_1, ..., X_m$ and Y well approximated by a m -dimensional linear function

$$y = a_0 + a_1 x_1 + a_2 x_2 + ... + a_m x^m = a_0 + \sum_{k=1}^{m} a_k x_k$$

The error is quantified by the function:

$$F(\vec{a}) = (\boldsymbol{X}\vec{a} - \vec{y})^T (\boldsymbol{X}\vec{a} - \vec{y}) \quad \text{with } \boldsymbol{X} = \begin{pmatrix} 1 & x_{1,1} & ... & x_{m,1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{1,n} & ... & x_{m,n} \end{pmatrix}, \vec{y} = \begin{pmatrix} y_1 \\ ... \\ y_n \end{pmatrix}, \vec{a} = \begin{pmatrix} a_0 \\ a_1 \\ ... \\ a_m \end{pmatrix}$$

Instead of minimizing the derivative, one has to minimize the gradient:

$$\nabla_{\vec{a}} F(\vec{a}) = \nabla_{\vec{a}} (\boldsymbol{X}\vec{a} - \vec{y})^T (\boldsymbol{X}\vec{a} - \vec{y}) = \vec{0}$$

Which gives a system of equations:

$$\boldsymbol{X}^T \boldsymbol{X} \vec{a} = \boldsymbol{X}^T \vec{y} \Rightarrow \vec{a} = \left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \boldsymbol{X}^T \vec{y}$$

That has solutions unless $\boldsymbol{X}^T\boldsymbol{X}$ is a singular matrix.

It should also be noted that it's possible to extend the techniques used to find linear relationships to non-linear ones. For example, suppose that the relationship between two variables X and Y of a dataset can be well approximated by a function $y = ax^b$. Taking the logarithm on both sides gives $\ln(y) = \ln(a) + b\ln(x)$. This means that, taking the logarithms of both x and y, one is in the situation of having to find a regression line.

This is particularly helpful for the tuning of the multilayer perceptron parameters, since the activation function that they use are non-linear. For example, suppose that the chosen activation function is the logistic function:

$$y = \frac{Y}{1 + e^{a+bx}}$$

Where Y, a, b are constants to be determined. If it's possible to "linearize" the function so that it's possible to apply the method of least squares to find the optimal values for these constants, then it's possible to find a **regression curve** carrying the optimal values for the original datapoints. If that's the case, it becomes possible to optimize the parameters of a two layer perceptron with a single input, since the value of a is the bias value of the output neuron and the value of b is the weight of the input.

The "linearization" can be performed as follows:

$$y = \frac{Y}{1 + e^{a+bx}} \Rightarrow \frac{1}{y} = \frac{1 + e^{a+bx}}{Y} \Rightarrow \frac{Y}{y} = 1 + e^{a+bx} \Rightarrow \frac{Y - y}{y} = e^{a+bx} \Rightarrow \ln\left(\frac{Y - y}{y}\right) = a + bx$$

This transformation is also known as **logit transformation**. By finding a regression line for the data points whose y variable is transformed according to left hand side of the equation, one (indirectly) obtains a regression curve for the original data points.

Exercise 1.1.1: Consider the dataset $\{(1, 0.4), (2, 1.0), (3, 3.0), (4, 5.0), (5, 5.6)\}$. Setting $Y = 6$, find the regression curve.

Solution: Each value of y is scaled as $\tilde{y} = \ln((Y - y)/y)$. This gives the new set of points $\{(1, 2.64), (2, 1.61), (3, 0.00), (4, -1.61), (5, -2.64)\}$. Noting that:

$$\sum_{i=1}^{n} x_i = 1 + 2 + 3 + 4 + 5 = 15 \qquad \sum_{i=1}^{n} \tilde{y}_i = 2.64 + 1.61 + 0.00 - 1.61 - 2.64 = 0$$

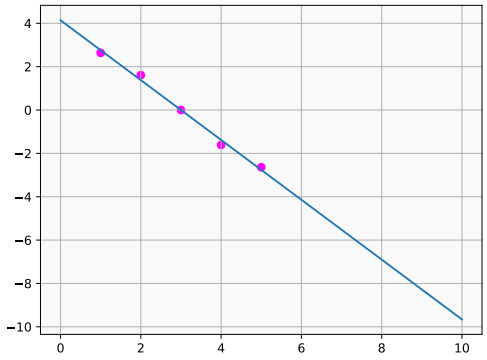
$$\sum_{i=1}^{n} x_i^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 55 \qquad \sum_{i=1}^{n} x_i\tilde{y}_i = 1 \cdot 2.64 + 2 \cdot 1.61 + 3 \cdot 0.00 - 4 \cdot 1.61 - 5 \cdot 2.64 \approx -13.78$$

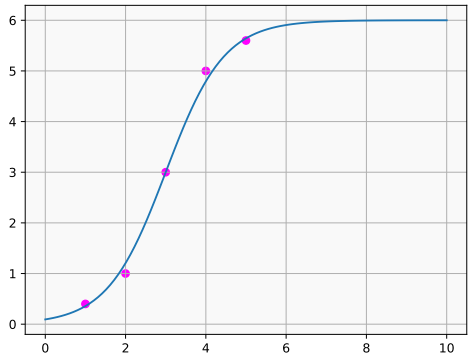
Leads to the following system of equations:

$$\begin{cases} 5a + 15b = 0 \\ 15a + 55b = -13.78 \end{cases} \Rightarrow \begin{cases} a = -3b \\ 15 \cdot (-3b) + 55b = -13.78 \end{cases} \Rightarrow \begin{cases} a = -3 \cdot (-1.38) \\ b = -1.38 \end{cases} \Rightarrow \begin{cases} a = 4.14 \\ b = -1.38 \end{cases}$$

Which gives the following regression line and, by extension, regression curve:

$$\tilde{y} = 4.14 - 1.38x \qquad \hat{y} = \frac{6}{1 + e^{4.14 - 1.38x}}$$





The resulting regression curve for the original data can be computed by a neuron with one input x having $f_{\text{net}}(x) = -1.38x$ as network input function, $f_{\text{act}}(\text{net}) = 1/\left(1 + e^{-(\text{net} \; -4.14)}\right)$ as activation function and $f_{\text{out}}(\text{act}) = 6 \; \text{act}$ as output function. □

Of course, the same approach can be used to find the optimized parameters of a two layer perceptron with more than one input. The problem with this approach is that the sum of square errors cannot be extended to multilayer perceptrons with more than two layers, because the layers in the middle cannot be taken into account.

1.2. Optimization problems

1.2.1. Optimization problems

An **optimization problem** is defined as a triple $(\Omega, f, \succ)$. $\Omega \subseteq \mathbb{R}^n$ is set called **search space**, $f : \Omega \mapsto \mathbb{R}$ is a function called **objective function** or **evaluation function** and $\succ$ stands for an inequality symbol (either $\geq$ or $\leq$).

Each member $\omega \in \Omega$ is said to be a **valid solution**, or simply a **solution**. In general, the search space is not the set of all real numbers, but a subset of reals that satisfy some conditions, or **constraints**. The elements of $\mathbb{R}^n$ that fall outside Ω (that is, those that do not abide by the constrains fixed by Ω) are called **invalid solutions**.

The value of $f(\omega)$ represents the "quality" or the "goodness" of ω. Two solutions ω_1 and ω_2 can be compared relying on the evaluation function: if $f(\omega_1) \geq f(\omega_2)$ and $\succ$ is $\geq$, then ω_1 is *better* than ω_2, and *worse* otherwise. On the other hand, if $f(\omega_1) \leq f(\omega_2)$ and $\succ$ is $\leq$, then ω_1 is *better* than ω_2, and *worse* otherwise.

A solution $\omega^* \in \Omega$ is said to be an **exact solution** of $(\Omega, f, \succ)$ if and only if it is an **optimum** for the evaluation function. That is, if $\succ$ is $\leq$, the optimum has to be a minimum ($\forall \omega \in \Omega, f(\omega^*) \leq f(\omega)$), while if $\succ$ is $\geq$ it has to be a maximum. ($\forall \omega \in \Omega, f(\omega^*) \geq f(\omega)$). **Solving** an optimization problem simply means finding its exact solution: if the search space contains more than one optima, solving the optimization problem means finding one (no matter which one) out of them.

If the symbol $\succ$ is $\geq$, an optimization problem $(\Omega, f, \geq)$ is also called a **maximization problem**; if it's $\leq$, it's called a **minimization problem**. Note that a maximization problem can be converted into a maximization problem or vice versa simply by changing the sign of the evaluation function: that is, $(\Omega, f, \geq) = (\Omega, -f, \leq)$ and $(\Omega, f, \leq) = (\Omega, -f, \geq)$.

Exercise 1.2.1.1: Consider the problem of finding the lengths of a tridimensional box with fixed surface area S such that its volume is as big as possible. How can it be formulated into an optimization problem? Does it have an exact solution?

Solution: The search space of the problem is the set of all triples of positive real numbers, representing all the possible values for the three lengths, constrained by forming a box having area equal to S. The evaluation function is simply the volume of the box:

$$(\Omega, f, >) = (\{(x, y, z) \in \mathbb{R}^+ \mid 2xy + 2xz + 2yz = S\}, f(x, y, z) = xyz)$$

The problem can be solved, for example using the method of Lagrange multipliers. Constructing the Lagrangian:

$$\mathcal{L} = f(x, y, z) + \lambda \cdot g(x, y, z) = xyz + 2\lambda xy + 2\lambda xz + 2\lambda yz - \lambda S$$

Computing its gradient:

$$\nabla(\mathcal{L}) = (yz + 2\lambda(y + z) \quad xz + 2\lambda(x + z) \quad xy + 2\lambda(x + y) \quad 2xy + 2xz + 2yz - S)^T$$

Setting it to 0 and solving[1] for x, y, z gives the exact solution $x = y = z = \sqrt{S/6}$. □

The approaches to solve optimization problems fall into four broad categories:

[1] Done automatically in Python using the `sympy` package.

- **Analytical Solution**: finding an optimum of the evaluation function by computing it directly, such as employing the *Method of Lagrange Multipliers*. This is the "obvious" way of solving an optimization problem, but it's hardly applicable, either because an analytical solution does not exist or because it's too computationally expensive to retrieve it;
- **Complete/Exhaustive Exploration**: finding the optimum of the evaluation function by trying every possible solution in the search space. Even though this guarantees to find an exact solution sooner or later, if the search space is too big the approach quickly becomes inefficient. Also, the approach is only applicable to search spaces that are discrete.
- **(Blind) Random Search**: finding the optimum of the evaluation function by trying random values of the search space, keeping track of the best solution found so far, and stopping when a "satisfactory" solution is found or when a given number of attempts is reached. The approach is hardly promising;
- **Guided (Random) Search**: finding the optimum of the evaluation function by trying out solutions in the search space, not randomly (like random search) but by "steering" the exploration of the search space by gathering information on the previous attempts. The idea is to start with a (mostly) random solution, observing how to change the solution in order to obtain a better one, and repeating the process until a "satisfactory" result is obtained.

1.2.2. Examples of optimization problems

Optimization problems are ubiquitous in fields where the goal is to maximize the efficiency/performance/return of a process. Examples include:

- Routing problems: finding the smallest route to take when moving from a start to a destination;
- Packing problems: finding out how to store as many objects as possible in a given container;
- Scheduling problems: determining how to arrange jobs or tasks in such a way that they do not overlap and they yield the best result (such as air traffic coordination).

A well-known example of routing problem is the **Travelling Salesman Problem** (**TSP**), that can be formulated informally as an analogy. Suppose that a traveller has a set of cities that they want to travel to, connected by roads, more or less distant from each other. How can they reach all cities of their planned trip, reaching each exactly once, such that the cumulative travelled distance is as small as possible?

Formally, the problem is understood in terms of graphs. Let $G = (V, E, W)$ be a weighted graph, with $V = \{v_1, ..., v_n\}$ a set of vertices, $E \subseteq V \times V - \{(v, v) \mid v \in V\}$ a set of edges (having no loops) and $W : E \to \mathbb{R}^+$ a function that assigns a (positive) weight to each edge. Each node represents a city, each edge represents a road that connects two cities and each weight represents the length of the road (or the time needed to travel it).

The Travelling Salesman Problem is then the optimization problem (Ω, f), where Ω is the set of all possible permutations of indices of the vertices that, two by two, have an edge that connects them:

$$\Omega = \left\{\pi(n) \mid \forall k \in [1, n], \left(v_{\pi(k)}, v_{\pi((k+1) \bmod n)}\right) \in E\right\}$$

Each representing one possible **Eulerian cycle** of the graph, a path that starts and ends in the same node and that reaches all of its nodes exactly once. The function f is the sum of all the weights of a member in Ω, sign-flipped:

$$f(\pi) = -\sum_{k=1}^{n} W\left(\left(v_{\pi(k)}, v_{\pi((k+1) \bmod n)}\right)\right)$$

The mod n is just to ensure that the last vertex "loops back" and connects to the first. The minus sign in front turns the original minimization problem into a maximization problem.

A solution of the TSP is then a solution $\pi^* \in \Omega$ that maximizes f. These represent an **Hamiltonian cycle** of the graph, an Eulerian cycle whose cumulative weight is as small as possible. Of course, a graph can have more than one Hamiltonian cycle.

The TSP is an NP-complete problem, therefore there is no way of computing a solution of the problem within an reasonable time bound, unless the dimension of the problem (the number of nodes in the graph) is very small.

For simplicity, the graph of the problem can be assumed to be complete, meaning that any node is connected to any other. If this is not the case, it is sufficient to add edges where are missing whose weight is so big that it's guaranteed to not be included in the solution. To simplify it further, the graph is assumed to be undirected, therefore the direction chosen to move from node to node is irrelevant.

1.2.3. Multi-criteria optimization problems

This simple model of optimization problem can be extended with more than one objective function, the so-called **multi-criteria optimization problems**, in the form $\left(\Omega, \left(f_1, \succ_1\right), ..., \left(f_k, \succ_k\right)\right)$.

Solving such problems does not simply entail finding a solution in Ω that maximizes/minimizes all functions at the same time, since some functions have to be minimized and others to be maximized. This means that a solution that has a very good quality with respect to some objective functions might have very bad quality with respect to others. Therefore, solving a multi-criteria optimization problem consists in finding the solution that yields the highest/lowest value for all objective functions at the same time *as much as possible*.

The simplest approach for solving a multi-criteria optimization problem is to combine all objective functions $(f_1, ..., f_k)$ into one, effectively reducing the problem to a standard optimization problem. This is done as follows:

$$f(\omega) = \sum_{i=1}^{k} w_i f_i(\omega) = w_1 f_1(\omega) + w_2 f_2(\omega) + ... + w_k f_k(\omega)$$

Where the weights $w_1, ..., w_k$ represent the "importance" of having a particular function maximized/minimized at the expence of the others. In other words, a great (absolute) value of w_i means that an exact solution to the problem should be as close as possible to an optima of f_i, whereas a low (absolute) value of w_i means that an exact solution to the problem doesn't have to strive to reach an optima of f_i. Also, the sign of the weights can be adjusted so that all functions have to be maximized/minimized.

This approach is not particularly effective for multiple reasons. First, note how this just shifts the goalpost: now solving the problem requires to find appropriate weights, which in general is not possible aside from employing some heuristics. Also, even if it were to possible to find them, this does not allow for the weights to be adapted based on the properties of the potential solutions to be obtained (unless the function is computed again). However, the real issue lies even deeper: each arrangement of weights defines a *preference order* of the solution candidates, and one has to aggregate these preference orders over the different arrangements to obtain an ordering of the solution candidates. This is also called the **problem of preference aggregation**.

Consider an multi-criteria optimization problem $(\Omega, f_1, ..., f_k, >)$, where all functions have already been tuned (changing their sign if needed) so that all functions have to be maximized. A solution $\omega_1 \in \Omega$ **dominates** another solution $\omega_2 \in \Omega$ if and only if, for any $1 \leq i \leq k$, $f(\omega_1) \geq f(\omega_2)$. A solution $\omega_1 \in \Omega$ **strictly dominates** another solution $\omega_2 \in \Omega$ if and only if ω_1 dominates ω_2 and there's at least one i in $\{1, ..., k\}$ such that $f(\omega_1) > f(\omega_2)$. If an element $\omega \in \Omega$ is not strictly dominated by any other element $\omega' \in \Omega$, it is said to be **Pareto-optimal**.

An alternative approach to combining all objective functions into one is to find one of these Pareto-optimal solutions. This is clearly a better approach, since now there's no need to specify the weights w_i and it becomes possible to change the priorities of the solutions based on the obtained result, without having to recompute everything again. The downside is that there is rarely just one Pareto-optimal solutions: in most cases, those form a set called **Pareto-frontier**.

A more interesting goal to set for solving multi-criteria optimization problems is to find not any solution in the Pareto-frontier, but the Pareto-frontier in its entirety. Clearly, this rules out the weighted combination method presented above, since it produces a single solution (although said solution *does* lie on the Pareto-frontier, or at least gets close). Even if one were to repeat the process r times, and returning r solutions, said solutions would still be in the same neighborhood of the frontier. This is because the combined objective functions act as a k-dimensional hyperplane that intersects the search space, and the solutions are either in said intersection (assuming that it exists) or close to it. Solving this form of the problem has to be done following different paths, such as employing evolutionary algorithms (discussed further).

1.3. Local search algorithms

Among guided search algorithms, particular relevance have the so-called **local search algorithms**. The ones hereby presented are concerned with finding a minimum or a maximum of a real-valued function, meaning that they tackle an optimization problem where the search space is the set of real numbers and the evaluation function is the function whose optima are of interest.

All local search algorithms work by starting from a random solution, probing the neighboring solutions and, if a better one is found, restarting from said point, repeating the process until a sufficiently satisfactory solution is found. Of course, such algorithms rely on a (to be justified) assumption: the result of evaluating similar elements of the search space must yield similar results. Otherwise, probing the neighborhood of a solution would be a pointless endeavour. This is also referred to as the **principle of small improvements**.

Of course, such problems could be solved analitically by computing the gradient of the evaluation function and setting it equal to 0, but this is possible only for very simple functions. This is because such calculation might be very hard to carry out or, as in the case of polynomial functions with degree greater than 5, impossible.

Also, since no local search algorithm is guaranteed to terminate with an exact result, there is the possibility of getting "stuck" in a neverending loop. For this reason, it is necessary to introduce a termination criteria that prevents the algorithm to run indefinitely. For example, the algorithm can be terminated when a maximum number of iterations has been reached or when the difference between the results yielded by the evaluation function for consecutive candidate solutions become negligible.

1.3.1. Gradient ascent/descent

Gradient ascent/descent is an obiquitous local search algorithm that explores the search space by relying on the gradient of the evaluation function. The idea is to start evaluating the gradient of the function in a random point of the search space, moving to a point in its neighborhood where the gradient is bigger (ascent) or smaller (descent) and repeating the process. Over many iterations, as long as the evaluation function is differentiable everywhere, there's a guarantee to get very close to an optimum.

To determine where to move after having computed the gradient in the current iteration, it possible to rely on the properties of the gradient. By definition, the gradient in a point is a vector that points in the direction where the function is the steepest. This means that the direction of the gradient (or the opposite direction) is the direction where the function, with respect to that point, increases or decreases the most, and therefore moves closer to an optimum as fast as possible.

GRADIENT-ASCENT/DESCENT($f : \mathbb{R}^n \to \mathbb{R}, \eta, \varepsilon$):

```
1  x = (x_1, ..., x_n) ← random initial solution
2  do
3      ∇ = (∇_1, ..., ∇_n) ← ( ∂/∂x_1 (f(x)), ..., ∂/∂x_n (f(x)) )   // Compute gradient
4      x ← x ± η∇                                                    // Update candidate solution
5  while (not(ε))
6  return x                                                          // Return best solution found
```

The parameter η of the algorithm denotes the size of the step taken in the direction of the gradient (or in the opposite direction). The choice of η is critical because a step size too small can result in a very slow procedure, whereas a step size too big can result in a process that oscillates back and forth in the

neighborhood of an optimum without reaching it. A possible refinement of the algorithm would be to adjust η in accord to the gradient: making long steps when the gradient is small, hence the function is almost linear, and making small steps when the gradient is big, to avoid overstepping.

One noticeable issue with gradient ascent/descent is that it does not distinguish between a local optimum and a global optimum, since in both cases the gradient is zero. Once an optimum has been found, gradient ascent/descent won't take alternative paths. This means that the starting point chosen to initiate the procedure can determine whether it will land in a global or local optimum, depending on which is the closest. The issue can be to some extent mitigated by repeating gradient descent multiple times with different starting points, and choosing the best result obtained among each trial

1.3.2. Hill climbing

For functions that are not differentiable everywhere it is still possible to get a rough estimate to where the function grows in a given point by simply trying random points in its neighborhood. If the new point yields an higher value for the function to be optimized, meaning that it's closer to a local optimum, the process restarts with this new point, otherwise another point in the neighborhood is tried. This approach, which can be thought of as a "naive" gradient descent, is called **hill climbing**.

HILL-CLIMBING($f : \mathbb{R}^n \to \mathbb{R}, \varepsilon$):

```
x = (x_1, ..., x_n) ← random initial solution
do
    x' = (x'_1, ..., x'_n) ← random point near x
    if (f(x') > f(x))
        x ← x'                          // Update if there is improvement
while (not(ε))
return x                                // Return best solution found
```

Even though this approach can be applied to more classes of functions, it inherits the same issues of gradient ascent/descent, mainly the tendency of getting stuck in local optima. Also, having to try random points means that some iterations will end up doing nothing, making it much more wasteful than gradient descent (where each iteration is a guaranteed improvement).

1.3.3. Simulated annealing

The issue of getting stuck in local optima can be overcome by allowing the search algorithm to consider solutions that are suboptimal in the short run, but that are deemed "promising" enough to lead to even better solutions in the long rung.

The idea is to start from a random point in the domain of the function and try points in its neighborhood for improvements, but deliberately accepting a worse solution as the current solution candidate under certain circumstances. More specifically, if the newly chosen point yields a better value for the evaluation function it is chosen as the new candidate solution, but if it yields a worse value it is accepted anyway as the new candidate solution with a certain probability, defined by the algorithm.

One local search algorithm that employs such strategy is **simulated annealing**. The algorithm works like hill climbing, but accepts as new candidate solution a worse solution with a probability dependent both on a parameter T, called **temperature**, on the difference between the current and the new candidate solutions and on the range of possible values $\Delta_{\max}$ that have been encountered so far. If the tolerance of accepting a worse solution is zero, simulated annealing is indistinguishable from hill climbing.

If the new solution candidate is worse than the current solution candidate but they yield similar values for the evaluation function, the algorithm will accept the new solution with higher probability. Also, the temperature parameter is decreased iteration by iteration, meaning that the algorithm will be more inclined to accept a worse solution in the early iterations and will become more and more "conservative" as the iterations go by.

```
SIMULATED-ANNEALING(f : ℝ^n → ℝ, T, δ, ε):
x = (x_1, ..., x_n) ← random initial solution
Δ_max ← 0                                   // Variability across solutions
do
    x' = (x'_1, ..., x'_n) ← random point near x
    Δ ← f(x') − f(x)                        // Improvement size
    if (|Δ| > Δ_max)
        Δ_max ← |Δ|
    p ← e^{Δ/Δ_max T}                       // Tolerance to worse solutions
    u ← a value sampled from U ∼ (0, 1)
    if (Δ > 0 ∨ p ≥ u)                      // New solution is better or tolerated
        x ← x'
    T ← T − δ(T)                            // Decrease temperature
while (not(ε))
return x                                    // Return best solution found
```

Simulated annealing has shown promising results when adapted to solve the Travelling Salesman Problem. To adapt simulated annealing to solve the TSP, it is necessary to define: what constitutes a solution, how to move in the search space (that is, how to construct a new candidate solution from the current candidate) and how to compute $\Delta_{\max}$.

Given a graph, the possible solutions of the TSP for said graph are all the possible permutations of the set of vertices of the graph. The evaluation function is the function that has a permutation as input and returns the sum of all edges that constitute the path described by the permutation.

Given a permutation π, a new solution can be constructed out of π as follows. Pick four distinct vertices in the graph, A, B, C, D, such that (A, B) and (C, D) are pairs of adjacent vertices in π. A new candidate solution can be obtained by swapping the order of B and C.

The range of qualities $\Delta_{\max}$ is impossible to compute, but can be reasonably estimated as follows:

$$\Delta_{\max} = \frac{t+1}{t}(Q_{\max,t} - Q_{\min,t})$$

With t being the number of the current iteration and $Q_{\max,t}$ and $Q_{\min,t}$ being the highest and lowest qualities found so far among candidate solutions.

1.3.4. Threshold accepting

The idea of **threshold accepting** is similar to simulated annealing: a worse solution can be accepted but only if it's sufficiently similar to the current one, meaning that their difference is smaller than a given threshold θ that decreases over time.

THRESHOLD-ACCEPTING($f : \mathbb{R}^n \to \mathbb{R}, \theta, \delta, \varepsilon$):

$x = (x_1, ..., x_n) \leftarrow$ random initial solution
do
    $x' = (x'_1, ..., x'_n) \leftarrow$ random point near x
    if $(f(x) - f(x') < \theta)$ // New solution is better or tolerated
        $x \leftarrow x'$
    $\theta \leftarrow \theta - \delta(\theta)$ // Decrease threshold
while (**not**(ε))
return x // Return best solution found

1.3.5. Great Deluge Algorithm

The **Great Deluge Algorithm** is similar to threshold accepting, but the tolerance of accepting worse solutions depends only on the initial choice of parameters for the threshold and on the number of the iteration, not on the current solution candidate. Such parameters are an initial threshold θ_0 and a scaling factor η.

GREAT-DELUGE($f : \mathbb{R}^n \to \mathbb{R}, \theta_0, \eta, \varepsilon$):

$x = (x_1, ..., x_n) \leftarrow$ random initial solution
$t \leftarrow 0$ // Initialize iteration counter
do
    $x' = (x'_1, ..., x'_n) \leftarrow$ random point near x
    if $(f(x') \geq \theta_0 + t \cdot \eta)$ // New solution is better or tolerated
        $x \leftarrow x'$
    $t \leftarrow t + 1$ // Increase iteration counter
while (**not**(ε))
return x // Return best solution found

1.3.6. Record-to-Record Travel

Record-To-Record Travel uses a lower bound for tolerating worse solutions, similar to Great Deluge, but such threshold also depends on the value yielded by the best solution found so far and, like threshold accepting, is decreased over time.

RECORD-TO-RECORD-TRAVEL($f : \mathbb{R}^n \to \mathbb{R}, \theta, \varepsilon$):

```
1  x = (x_1, ..., x_n) ← random initial solution
2  x_best ← x                                   // Initialize best solution
3  do
4      x' = (x'_1, ..., x'_n) ← random point near x
5      if (f(x') ≥ f(x_best) − θ)                // New solution is better or tolerated
6          x ← x'
7      if (f(x') > f(x_best))                    // New solution better than the best
8          x_best ← x'                           // Increase iteration counter
9  while (not(ε))
10 return x_best                                 // Return best solution found
```

2. Neural Networks

2.1. Biological foundations

The goal (or dream?) of artificial intelligence is to reproduce into silicon the way humans and animals think, their intelligence, their cognitive capabilities, their ability to learn from experience. To do so, it is necessary to analyse the apparatus responsible for cognition and decision making: the **nervous system**. The nervous system comprises the **sensory systems**, responsible for gathering information from the outside (smell, sight, sound, ecc...), the **motory system**, responsible for motion, and the **brain**, that gathers and centralises inputs from the nervous system to elaborate.

The fundamental component of the nervous system is the **neuron**, a cell that collects and transmits electrical activity. A neuron[2] is constituted by a central body called **soma** that contains the nucleus, many small ramifications called **dendrites** and a long extension called **axon**. Neurons are connected with each other in an intricate structure and communicate with each other. The axon of a neuron is almost attached (very close, but not touching) to the dendrites of its neighboring neurons. The point of (almost) contact between an axon and the dendrites is called a **synapse**.

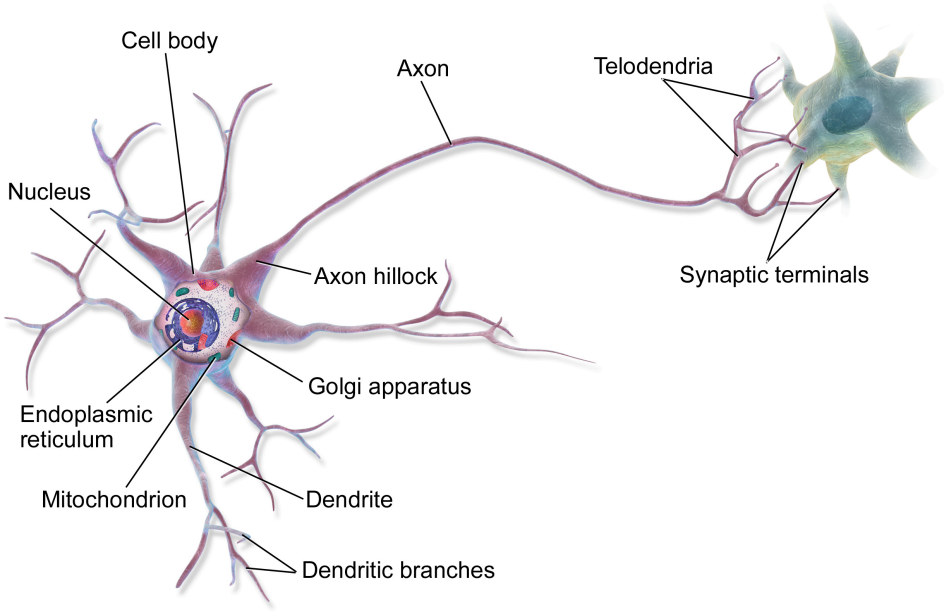


Figure 1: Schematic representation of a neuron. [Original image by BruceBlaus, licensed under CC BY 3.0.]

Neurons communicate with the ones they are attached with using electricity. More specifically, this is done by releasing through the synapse by the terminal section of the axon certain chemicals, called **neurotransmitters**. These act on the membrane of the receiving dendrite and change its polarization (there's a slight potential difference between the inside and the outside of the cell membrane of the neuron). Depending on the type of the released neurotransmitter, the potential difference may be reduced or increased on the side of the dendrite. Synapses that reduce the potential difference are called **excitatory**, those that increase it are called **inhibitory**.

The potential difference in a single neuron is close to be negligible, but the cumulative potential difference of thousands of neurons connected together is sufficient to stimulate the axon of a neuron to propagate the difference to its neighbors. Simply put: if a neuron receives sufficient amount of

[2]Not all neurons have the same, identical structure. This isn't particularly relevant, however: the interest isn't in describing the nervous system in detail, but to device an abstract representation of a neuron that can be implemented in a computer.

“stimulation”, meaning that a certain “stimulation threshold” is reached, then it will release said “stimulation” towards its neighbors and restore its equilibrium state. The number of impulses that a neuron transmits per second is also referred to as its **firing rate**.

The connections between neurons are fixed at one’s birth. Neurons are a class of cell that never undergoes division. That is, the number of neurons in the nervous system either decreases or remains the same, it’s (almost) impossible to replace a damaged or dead neuron with a new one. As a matter of fact, the neurons in the brain slowly and slowly degrade almost as one is born, until one by one they all die.

This (heavily) simplified description of the nervous system still hints at some advantages that an architecture built on this model could offer:

- The nervous system is *inherently parallel*: different sections of the system can work independently of each other and at the same time;
- The nervous system is *fault tolerant*: if a neuron stops working for some reason, its neighbors can still operate, and the overall system is still functional;
- The nervous system has *slow phase-out*: neurons slowly loose their capabilities, not abruptly. There is time for the architecture to accomodate the change.

2.2. Threshold logic units

2.2.1. Single threshold logic units

A **Threshold Logic Unit** (**TLU**), also known as **perceptron** or **McCulloch-Pitts neuron**[3] is a mathematical structure that models, in a very simplified way, how neurons operate.

A TLU has n inputs $x_1, x_2, ..., x_n$, each weighted by a weight $w_1, w_2, ..., w_n$, that generates a single binary output y. If the sum of all the inputs multiplied by their weights is a value greater or equal than a given threshold θ, the output y is equal to 1, to 0 otherwise:

$$y = H\left(\sum_{i=1}^{n} w_i x_i, \theta\right) = \begin{cases} 1 \text{ if } \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

This function is also referred to as the **Heaviside function**, or **stepwise function**.

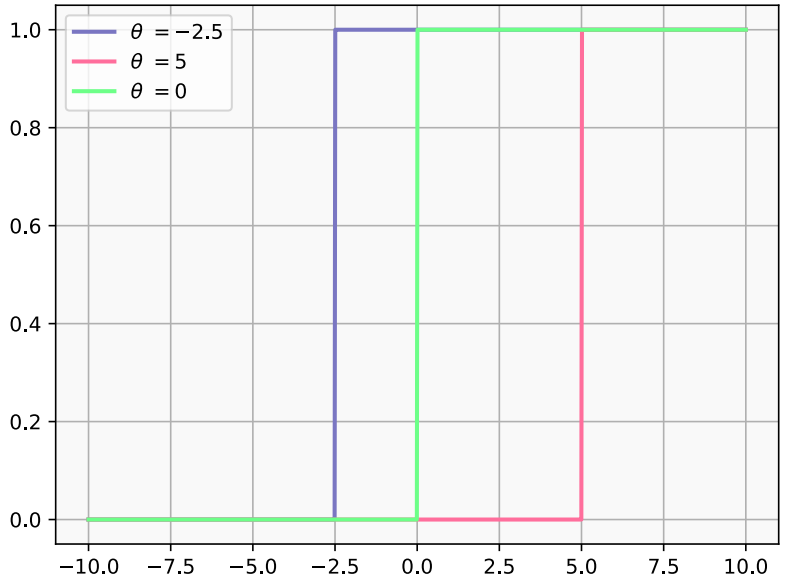


Figure 2: Plot of the Heaviside function, with three different choices of θ

The inputs can be collected into a single input vector $\boldsymbol{x} = (x_1, ..., x_n)$ and the weights into a weight vector $\boldsymbol{w} = (w_1, ..., w_n)$. This way, the output y is equal to 1 if the scalar product between $\boldsymbol{w}$ and $\boldsymbol{x}$ is greater or equal than θ:

$$y = \begin{cases} 1 \text{ if } \langle \boldsymbol{w}, \boldsymbol{x} \rangle \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

[3]The terminology is all over the place: the three mathematical objects have different degree of complexity. This is an oversimplification

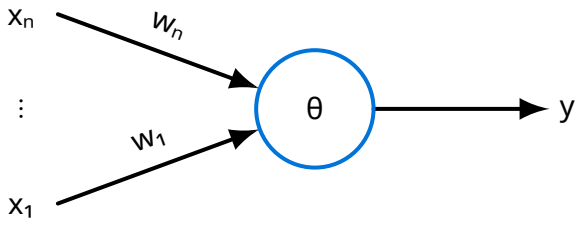


Figure 3: A common way of representing a TLU. The processing unit is drawn as a circle, with the threshold in its center. Inputs are drawn as arrows entering the TLU from the left, with their respective weights above. The output is an arrow exiting the TLU from the right.

The analogy between TLUs and biological neurons is straightforward. The output of a TLU is analogous to the firing of a neuron: an output equal to 1 corresponds to the firing of a neuron, whereas an output equal to 0 corresponds to a neuron insufficiently stimulated to be firing. A positive weight corresponds to an excitatory synapse, that increases the amount of stimulation received by the neuron, while a negative weight corresponds to an inhibitory synapse, that reduces the amount of stimulation.

Exercise 2.2.1.1: Construct a TLU with two inputs whose threshold is $+4$ and whose (two) weights are $+3$ and $+2$ respectively.

Solution:

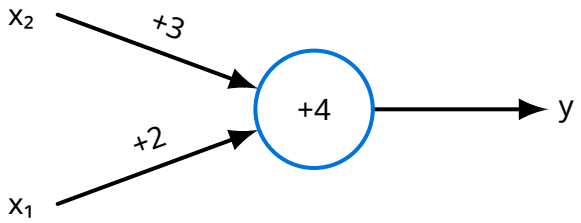


Figure 4: A TLU with x_1, x_2 as inputs, $w_1 = +3, w_2 = +2$ as weights and $\theta = +4$ as threshold.

□

It is easy to construct TLUs that can compute all four basic logical connectives: `AND`, `OR`, `NOT` and `IF...THEN`.

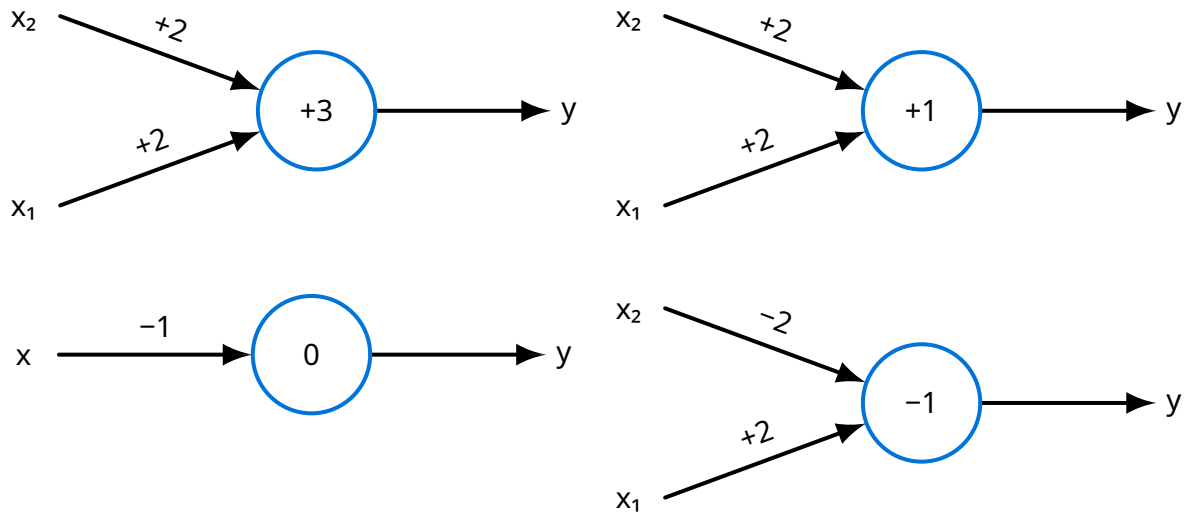


Figure 5: Going clockwise from top left: a TLU that computes conjunction, disjunction, implication and negation. In this context, the inputs x_1 and x_2 are assumed to be either 0 or 1. Of course, there are many other choices of weights and thresholds that encode the same functions.

Since the purpose of a TLU is to model the neurons in the brain, there is clear interest is in understanding if a TLU can encode any possible function or, if this were not to be the case, which functions can be encoded.

First of all, notice how the weighted summation that determines its output is very similar to an n-dimensional linear function. That is, by substituting $\geq$ with $=$, one obtains an n-dimensional straight line:

$$\sum_{i=1}^{n} w_i x_i = \theta \Rightarrow \sum_{i=1}^{n} w_i x_i - \theta = 0 \Rightarrow w_1 x_1 + w_2 x_2 + ... + w_n x_n - \theta = 0$$

As a matter of fact, the line $\sum_{i=1}^{n} w_i x_i - \theta = 0$ acts as a **decision border**, partitioning the n-dimensional Euclidean hyperplane into two half-hyperplanes: one containing n-dimensional points that give an output of 1 when fed to the TLU and the other containing points that give an output of 0.

To deduce which of the two regions of space is which, it suffices to inspect the coefficients of the line equation. By definition, the weights $w_1, ..., w_n$ are the coefficients of a vector that is perpendicular to $\sum_{i=1}^{n} w_i x_i - \theta = 0$. By traslating said vector onto any point on the line, one obtains which side of the hyperplane contains the points that give 1 when fed to the TLU: the one that the vector is pointing to.

This is easier to visualize if the number of dimensions (the number of inputs to the TLU) is 3 or less, since it's possible to graph the decision border on a cartesian plane.

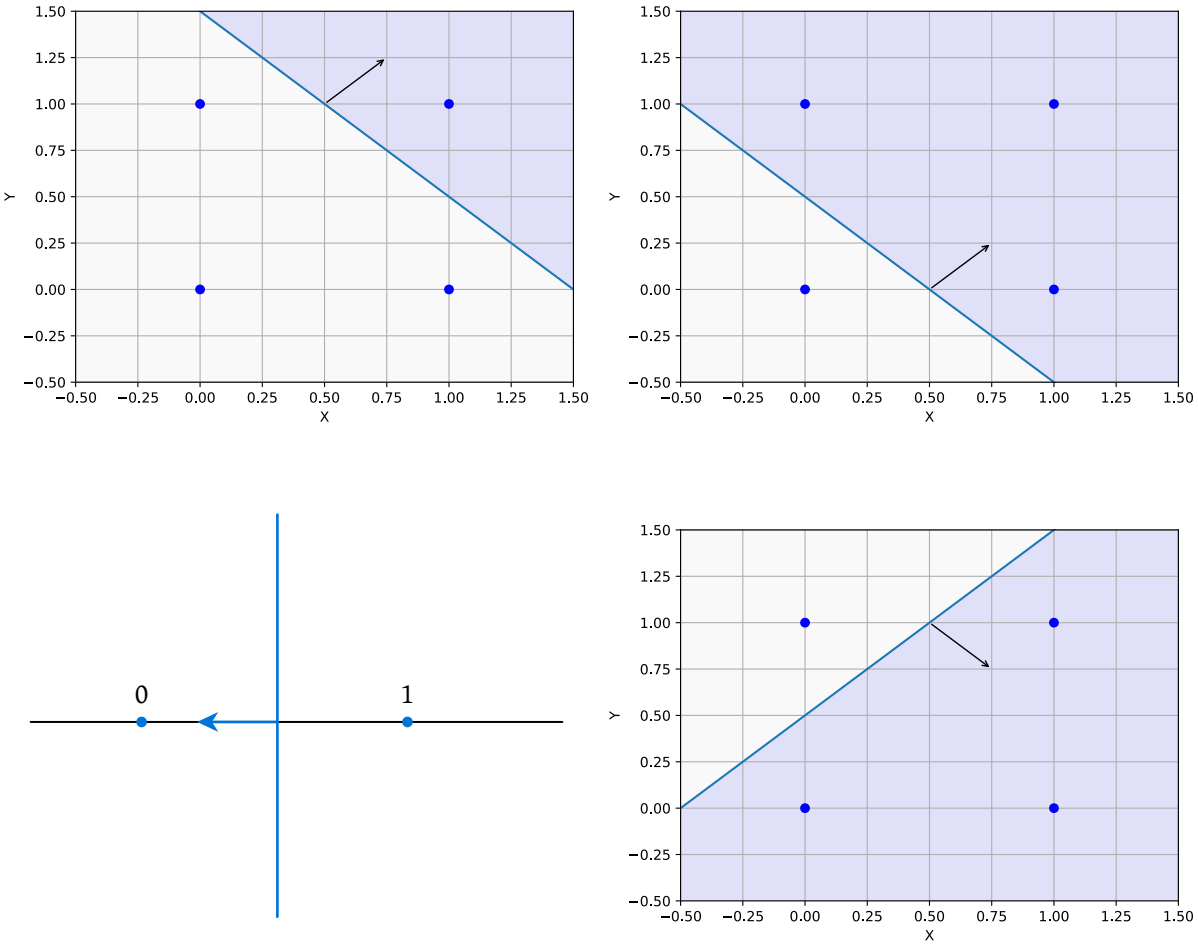


Figure 6: Going clockwise from top left: a cartesian plane partitioned by the TLU that computes the conjunction, disjunction, implication and negation.

More formally, two sets of points are called **linearly separable** if there exists a linear function, called **decision function**, that partitions the euclidean hyperplane into two half-planes, each containing one of the two sets. A set of points is called **convex** if it's possible to connect each point of the set with straight lines without crossing the boundaries of the set. The **convex hull** of a set of points is its the smallest superset that is convex. If two sets of points are both convex and disjoint, they are linearly separable.

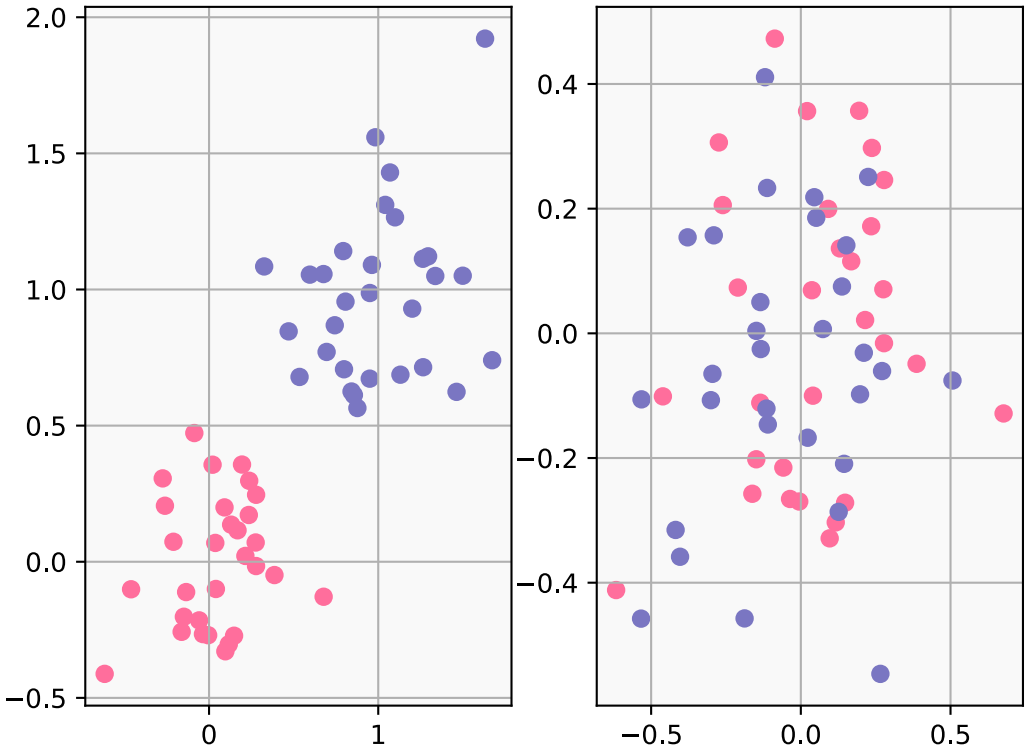


Figure 7: The two sets of points on the left (drawn with different colors) is linearly separable. The ones on the right are not

By extension, a binary function (a function that has two possible outputs) is said to be linearly separable if the set of inputs that give the first output and the set of inputs that give the second output are linearly separable sets.

Therefore, a single TLU can only encode linearly separable functions; indeed, `AND`, `OR`, `NOT` and `IF...THEN` are linearly separable functions. Unfortunately, not all (binary) functions are linearly separable: it is trivial to show, for example, that the biimplication and the logical `XOR` are not linearly separable. This means that a single TLU has (very) limited capabilities, because it can't encode all possible functions.

2.2.2. Training single threshold logic units

The TLUs described so far have all been constructed with the goal of embedding a function, fixing the parameters (thresholds and weights) "by hand". What would be more useful is having a TLU that can "deduce" the function without knowing its closed-form expression and simply by knowing how to map inputs to outputs. Even more interesting would be a TLU that can adjust its parameters "on its own", only based on the input-output pairs that it is presented with.

This would make a TLUs an even fitter model of human cognition: the TLU *learns* from its input-output pairs as a human learns from experience. Indeed, such a process exists, and can be described as such:

1. Start with a TLU having randomly generated values for the weights and the threshold;
2. Test the results given to see if this matches the expected outputs. If it doesn't, tune the TLU parameters in accord;
3. If the output of the TLU always matches the outputs in the data, stop. Otherwise, go back to from the previous point.

This process of tuning the parameters of a TLU one step at a time is also referred to as the **training** of the network.

The first step for constructing a learning process for a TLU is in quantifying "how much" the outputs of the TLU and the expected outputs differ. This quantification is given by an *error function* $e(w_1, ..., w_n, \theta)$, that taken in input the n weigths $w_1, ..., w_n$ of a TLU and the threshold θ and returns as output a weighted difference between the outputs of the TLU and the expected outputs. Clearly, when the error function results is 0, the expected and given outputs match perfectly. The goal is therefore to reduce step-by-step the value given by the error function until it reaches 0 (or gets sufficiently close).

The most intuitive way to construct this function would be to take the absolute value of the difference between the outputs of the function and the outputs of the TLU and summing all of them. However, this would create a stepwise error function, making it impossible to adjust the parameters automatically. A stepwise function is made of "plateaus", in which the function evaluates to the same value: aside from the points along a border, there is no way of inspecting the "neighborhood" of a point to know where to move to decrease the error.

A better way would be considering "how far" the threshold of the TLU is exceeded for each input. This way, it becomes possible to read "locally" where to follow along the shape of the error function by moving, at each step, in the direction of greatest descent, that is, with the direction of the highest slope, even when the overall shape of the function is unknown.

There are two main formulations of the training process. The first consists in going one input-output pair at a time, tuning the parameters where needed and restarting from the first if necessary. This is referred to as **online training**. The second consists in accumulating the tunings needed for each input and applying them all at once at the end of a training cycle. this is referred to as **batch training** and each training cycle is also referred to as an **epoch**. Both approaches are valid and have strengths and weaknesses on their own.

As for how to tune the parameters, one should start from this simple observation. If the output of the TLU is 1 and the output in the input-output pair is 0, it must mean that either the threshold has been set too low and/or the weights have been set too high. Therefore, if this is the case, one should raise the threshold and/or lower the weights in order for the TLU to match the expected output. On the other hand, if the output of the TLU is 0 and the expected output is 1, it must mean that the threshold has been set too high and or/ the weights have been set too low. If this is the case, one should lower the threshold and/or raise the weights. Of course, if the output of the TLU matches the expected output, there is no need for tuning its parameters.

What remains to be done is to define how exactly this parameter tuning should be performed. Consider a TLU having n weights $\boldsymbol{w} = (w_1, ..., w_n)$ and a threshold θ. Let $(\boldsymbol{x}, y) = ((x_1, ..., x_n), y)$ be an input-output pair that is to be presented to the TLU. Let $\hat{y}$ be output of the TLU when given $\boldsymbol{x}$ as input. If $\hat{y} \neq y$, the parameters of the TLU can be updated in accord to the following rule, called **delta rule**, or **Widrow-Hoff rule**:

$$\begin{cases} \theta & \leftarrow \theta - \eta(y - \hat{y}) \\ w_1 & \leftarrow w_1 + \eta(y - \hat{y})x_1 \\ w_2 & \leftarrow w_2 + \eta(y - \hat{y})x_2 \\ & ... \\ w_n & \leftarrow w_n + \eta(y - \hat{y})x_n \end{cases}$$

The parameter η is called **learning rate**, and determines how much the parameters are changed. At every step, the parameters are either increased or decreased by a factor of η. The choice of η is arbitrary: it shouldn't be set too low, because the training process would take too many steps, but shouldn't be set too high either, because the new value of the parameters might jump to another slope of the error function.

As stated, the update of the threshold and of the weights has to be done in opposite directions, which is why the delta rule treats them separately. However, it's possible to simplify the formula by turning the threshold into an extra weight and introducing an extra input.

To do so, recall that the output of a TLU is 1 if $\boldsymbol{w}, \boldsymbol{x}\rangle \geq \theta$ and 0 otherwise. This is the same as having output 1 if $\boldsymbol{w}, \boldsymbol{x}\rangle - \theta \geq 0$ and 0 otherwise by moving θ to the left hand side. Let x_0 an extra input that is always equal to 1: the expression can be rewritten as outputting 1 if $\boldsymbol{w}, \boldsymbol{x}\rangle - x_0\theta \geq 0$ and 0 otherwise, since x_0 has no influence on the update of θ. By renaming $\theta = w_0$, one has that the output of the TLU is 1 if $\sum_{i=0}^{n} w_i x_i \geq 0$ and 0 otherwise. In this formulation, the θ is always 0 and can be ignored.

It is now possible to restate the delta rule as follows. Consider a TLU having n weights $\boldsymbol{w} = (w_1, ..., w_n)$ and a threshold θ. Rewrite the weights as $\boldsymbol{w} = (w_0 = -\theta, w_1, ..., w_n)$ and set the threshold to 0. Let $(\boldsymbol{x}, y) = ((x_0 = 1, x_1, ..., x_n), y)$ be an input-output pair that is to be presented to the TLU, with an extra ficticious input x_0. Let $\hat{y}$ be output of the TLU when given $\boldsymbol{x}$ as input. If $\hat{y} \neq y$, the parameters of the TLU are updated as:

$$w_i \leftarrow w_i + \eta(y - \hat{y})x_i, \forall i \in \{0, 1, ..., n\}$$

Once the training process is over, if one so desires, it suffices to turn $-w_0$ back into θ and to remove the input x_0 to obtain the original formulation of the TLU.

The delta rule allows one to write out an algorithm for the training of TLU, both following the batch training paradigm and the online training paradigm. Let $L = ((X_1, y_1), ..., (X_m, y_m))$ be a set of examples used to train the TLU; each example is constituted by an array of binary inputs $X_j = \left(x_{1,j}, ..., x_{m,j}\right)$ and a binary output y_j. Let $W = (w_1, ..., w_n)$ be a set of randomly chosen initial weights and let θ be a randomly chosen initial threshold. The two algorithms are presented as follows:

TLU-TRAIN-ONLINE$(W = (w_1, ..., w_n), L = ((X_1, y_1), ..., (X_m, y_m)), \theta, \eta)$:

```
1  let e ← ∞                                          // Error
2  while (e ≠ 0)                                      // Continue until error vanishes
3      e ← 0
4      foreach l_i in L
5          let X, y ← l_{i,1}, l_{i,2}                // Unpack
6          let ŷ ← 0                                  // Evaluate scalar product
7          if (Σ_{j=1}^{|X|} X_j · W_j ≥ θ)
8              ŷ ← 1
9          if (ŷ ≠ y)                                 // Test for output mismatch
10             e ← e + |y − ŷ|                        // Update error
11             θ ← θ − η · (y − ŷ)                    // Update threshold
12             foreach w_j in W
13                 w_j ← w_j + η · (y − ŷ) · X_j      // Update weights
```

TLU-TRAIN-BATCH$(W = (w_1, ..., w_n), L = ((X_1, y_1), ..., (X_m, y_m)), \theta, \eta)$:

```
1  let e ← ∞                                          // Error
2  while (e ≠ 0)                                      // Continue until error vanishes
3      e ← 0
4      let θ* ← 0                                     // Partial threshold
5      let W* ← (0, ..., 0)                           // Partial weights
6      foreach l_i in L
7          let X, y ← l_{i,1}, l_{i,2}                // Unpack
8          let ŷ ← 0                                  // Evaluate scalar product
9          if (Σ_{j=1}^{|X|} X_j · W_j ≥ θ)
10             ŷ ← 1
11         if (ŷ ≠ y)                                 // Test for output mismatch
12             e ← e + |y − ŷ|                        // Update error
13             θ* ← θ* − η · (y − ŷ)                  // Partially update threshold
14             foreach w_j in W
15                 w*_j ← w*_j + η · (y − ŷ) · X_j     // Partially update weights
16         θ ← θ + θ*                                 // Update threshold
17         W ← W + W*                                 // Update weights
```

Both algorithms don't turn thresholds into weights just for clarity of explanation, but it would be trivial to edit them to introduce said variation.

Exercise 2.2.2.1: Using both online learning and batch learning, construct a TLU that computes the logical AND between two bits.

Solution:

Let $L = (((0,0),1),((0,1),0),((1,0),0),((1,1),1))$, $W = (0,0)$, $\theta = 0$ and $\eta = 1$. The tables on the left and on the right denote the training of the TLU, employing online learning and batch learning respectively[4].

Trial	Weights	θ	Error
0	$(0,0)$	0	∞
1	$(1,1)$	0	2
2	$(2,1)$	1	3
3	$(2,1)$	2	3
4	$(2,2)$	2	2
5	$(2,1)$	3	1
6	$(2,1)$	3	0

Trial	Weights	θ	Error
0	$(0,0)$	0	∞
1	$(-1,-1)$	3	3
2	$(0,0)$	2	1
3	$(1,1)$	1	1
4	$(0,0)$	3	2
5	$(1,1)$	2	1
6	$(1,1)$	2	0

□

The natural question to ask is whether the training process of a TLU always works, that is, if the function encoded in the TLU *converges* to the actual input-output mapping. As stated, threshold logic units can only encode functions that are linearly separable, therefore it is to be expected that the training process won't converge when presented with an input-output pair coming from non-linearly separable function. This is indeed the case, and what happens instead is that the error function oscillates without never reaching 0. However, if the function is linearly separable, the training process is guaranteed to converge.

Theorem 2.2.2.1 (Convergence Theorem for the Delta Rule): Let $L = ((X_1, y_1), ..., (X_m, y_m))$ be a set of training examples; each example is constituted by an array of binary inputs and a binary output y_j. Let:

$$L_0 = \{(X,y) \in L \mid y = 0\} \qquad L_1 = \{(X,y) \in L \mid y = 1\}$$

The subsets of L containing all the training examples having output equal to 0 and to 1 respectively. If both L_0 and L_1 are linearly separable, meaning that there exist a vector of weights $W = (w_1, ..., w_n) \in \mathbb{R}^n$ and a threshold $\theta \in \mathbb{R}$ such that:

$$\sum_{j=1}^{n} w_j X_j < \theta, \ \forall (X,0) \in L_0 \qquad \sum_{j=1}^{n} w_j X_j \geq \theta, \ \forall (X,1) \in L_1$$

Then, the training process (either batch or online) is guaranteed to terminate.

The delta rule can be improved upon even further by observing that inputs of 0 are problematic. Infact, note that the expression for updating the weights depends on the input, and if the input is 0 the entire expression is 0. This means that, effectively, if the input is 0 there is never an update, even if it necessary. This doesn't impact the correctness of the process (which will always converge, as stated in Theorem 2.2.2.1) but slows it down unnecessarely. The problem can be circumvented by using 1 and

[4]These have been computed with the Python snippet present in this folder.

-1 instead of 1 and 0 as boolean values, so that false inputs don't make the expression go to 0 and hence contribute meaningfully to the training. This is referred to as the **ADALINE model** (**ADAptive LINear Element**).

2.2.3. Multiple threshold logic units

Even though a single TLU can do little, it is possible to connects more TLUs together, creating a *network* of threshold logic units. This can be done by breaking down a complicated boolean function into approachable functions, each representable by a TLU, and using the outputs of TLUs as inputs of other TLUs. This way, each subsection of the network partitions the (hyper)plane into sub-(hyper)planes that are then recollected together. More formally: by applying a coordinate transformation, moving from the original domain to the image domain, the set of points become linearly separable again.

Exercise 2.2.3.1: Construct a network of threshold logic units that encodes biimplication.

Solution: One possibility is as follows:

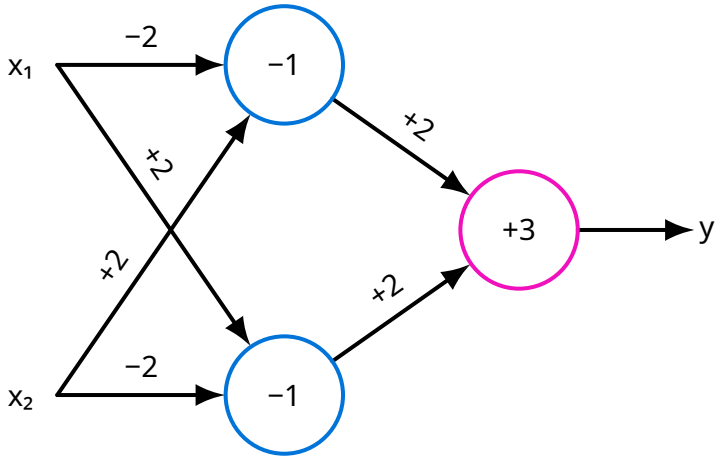


Figure 8: A network of TLUs that encodes the biimplication.

The idea is to deconstruct the biimplication $A \leftrightarrow B$ as $(A \rightarrow B) \wedge (B \rightarrow A)$. Each of the three functions (two single implications and one conjunction) is, when considered on its own, linearly separable.

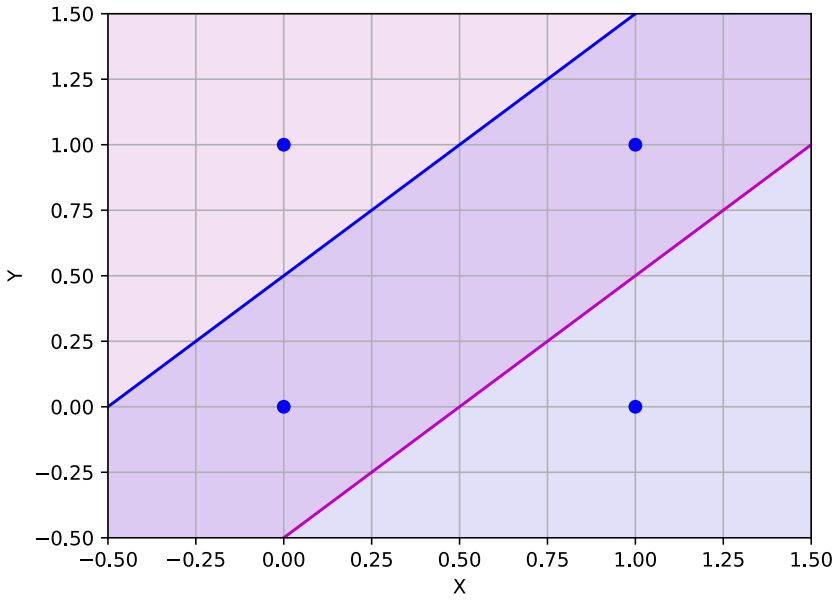


Figure 9: The euclidean plane partitioned by the two implications are in blue and in red. The strip in the middle (in purple) and the rest of the plane are now linearly separable.

□

Exercise 2.2.3.2: Construct a network of threshold logic units that encodes the exclusive or.

Solution: One possibility is as follows:

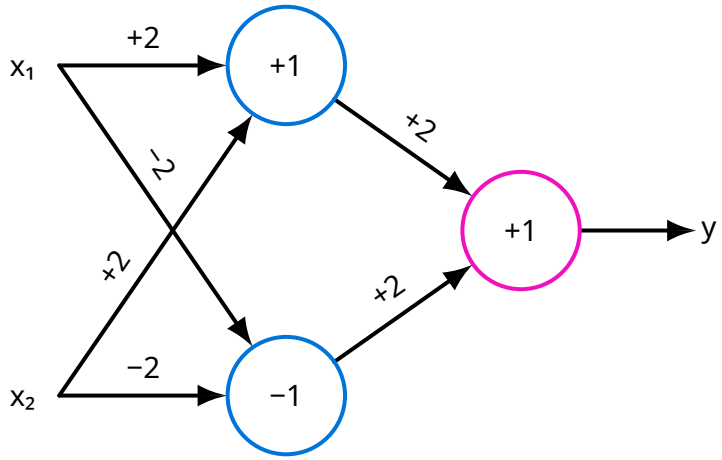


Figure 10: A network of TLUs that encodes the exclusive or.

This is done by rewriting $A \oplus B$ as $(A \vee B) \wedge (\neg A \vee \neg B)$.

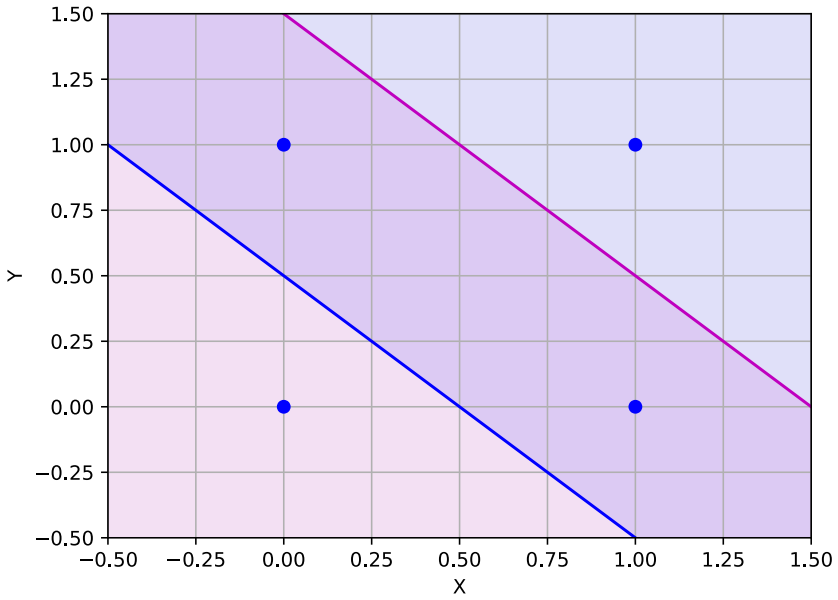


Figure 11: The euclidean plane partitioned by the two disjunctions are in blue and in red. The strip in the middle (in purple) and the rest of the plane are now linearly separable.

□

To further investigate the capabilities of a network of TLUs, recall that it's always possible to rewrite a boolean function in the so called **disjunctive normal form**, or **DNF** for short. A boolean function in DNF is constituted by a disjunction of many composite propositions; each of these propositions is a conjunction of (potentially negated) atomic propositions. That is:

$$K_1 \vee ... \vee K_m = (l_{1,1} \wedge ... \wedge l_{1,n}) \vee ... \vee (l_{m,1} \wedge ... \wedge l_{m,n}) = \bigvee_{j=1}^{m} \left(\bigwedge_{i=1}^{n} l_{j,i} \right)$$

Where each $l_{j,i}$ can be either non-negated (**positive literal**) or negated (**negative literal**).

There also exists the **conjunctive normal form**, or **CNF**, which is a conjunction of many disjunctions. The two forms are equivalent, since, any formula written in DNF can be converted into an equivalent expression in CNF, and vice versa.

Note that the individual components of a boolean expression in CNF or DNF are linearly separable. Therefore, any boolean expression in CNF or DNF can be encoded into a network of TLUs. Since any boolean expression can always be rewritten in such forms, this is tantamount to stating that a network of TLUs can encode any boolean expression.

Let $y = f(x_1, ..., x_n)$ be a boolean function of n variables. The following algorithm constructs a network of TLUs that encodes y:

1. Rewrite y in disjunctive normal form;
2. For each K_j construct a TLU having n inputs (one input for each variable) and the following weights and threshold:

$$w_{j,i} = \begin{cases} +2 \text{ if the j,i-th literal is a positive literal} \\ -2 \text{ if the j,i-th literal is a negative literal} \end{cases} \qquad \theta_j = n - 1 + \frac{1}{2} \sum_{i=1}^{n} w_{j,i}$$

3. Create one more TLU, whose m inputs are the outputs of the m TLUs so-far constructed. Its weights are all equal to $+2$ and its threshold to $+1$.

Exercise 2.2.3.3: Construct a TLU network for the boolean function:

$$F(A,B,C,D) = (A \wedge B \wedge C) \vee \left(\overline{A} \wedge B \wedge D\right) \vee \left(A \wedge B \wedge \overline{C} \wedge \overline{D}\right)$$

Solution:

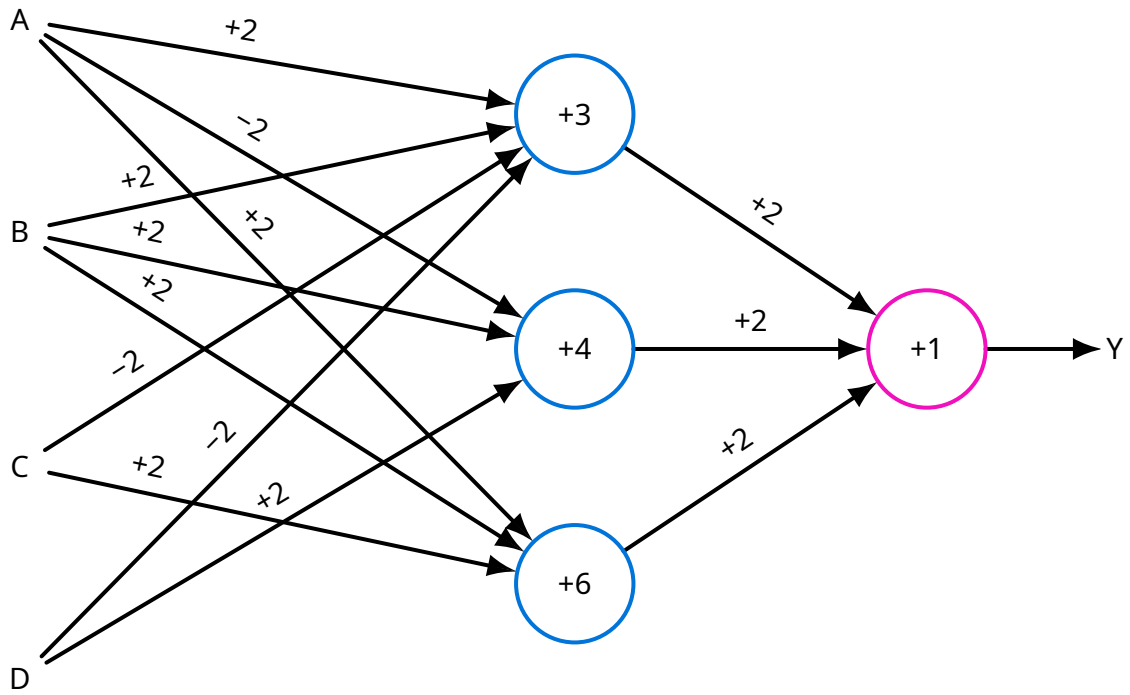


Figure 12: A network of TLUs that computes the given function, constructed employing the aforementioned algorithm.

□

The idea of the algorithm is to partition the space in which the output is expected to be 1 going one (hyper)side at a time. Each term each conjunction defines one area in which the output is 1 and then these disconnected areas are merged together with a disjunction. Different choices of weights and thresholds could be made: the one here given is simply one that results in integer values.

Having devised a training method for single TLUs, it would be interesting to engineer a training method for networks of TLUs. This would allow one to encode any kind of boolean function, not just linearly separable functions. Unfortunately, transferring the training process one-to-one from single TLUs to multiple TLUs is not possible. For example, the updates carried out by the delta rule are computed from the difference between the output of the original function and the output of the TLU. However, the tuned output becomes available only to the current TLU, while the others are oblivious to the changes. This means that, to train a network of TLUs, a completely different approach is required. Despite its shortcomings, this network structure points in the right direction.

2.3. Artificial neural networks

An **artificial neural network**, or simply **neural network**, is a directed graph $G = (U, C)$, whose vertices $u \in U$ are called **neurons** or **units** and whose edges $c \in C$ are called **connections**. Each connection $(v, u) \in C$ carries a **weight** $w_{u,v}$. The set of vertices is made up of three subsets: a set U_{in} of **input neurons**, a set U_{out} of **output neurons** and a set U_{hidden} of **hidden neurons**.

$$U = U_{\text{in}} \cap U_{\text{out}} \cap U_{\text{hidden}}$$

The set of input neurons and the set of output neurons cannot be empty and may not be disjoint. That is, a neuron can be both an input and an output neuron at the same time. The set of hidden neurons can be empty and must be disjoint from the sets of input and output neurons. That is, a neuron cannot be both a hidden neuron and an input/output neuron at the same time.

$$U_{\text{in}} \neq \emptyset \qquad U_{\text{out}} \neq \emptyset \qquad U_{\text{hidden}} \cap (U_{\text{in}} \cap U_{\text{out}}) = \emptyset$$

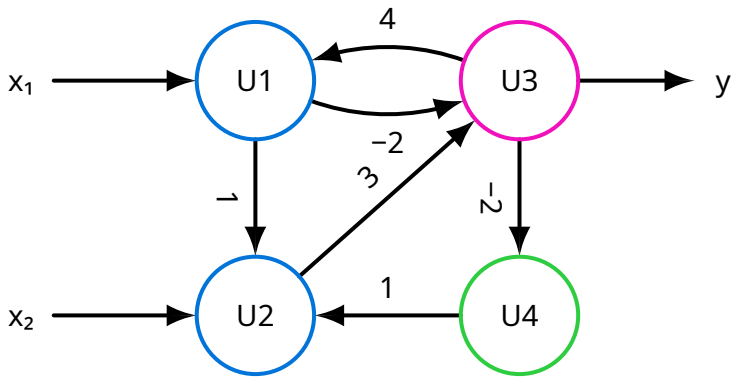
The input neurons receive information from the environment in the form of the external input, whereas the output neurons release the information processed by the network. The hidden neurons do not communicate with the environment directly, but only with other neurons, hence the name "hidden". By extension, the (external) input of a neural network is simply the external input fed to its input neurons, and the output of a neural network is the output of all of its output neurons.

Being a graph, it is natural to represent a neural network either with a pictorial representation (that is, circles and arrows) or as an adjacency matrix. These are also referred to as the **network structure** of the neural network.

In describing the network structure, it is customary to denote the ending node of the connection before the starting node, and not the other way around (how it's usually done). That is, a weight $w_{u,v}$ is carried by a connection *ending* in u and *starting* in v. This way, each row contains the weights of the connections leading to the same neuron, not those coming out of the same neuron.

Exercise 2.3.1: Let $G = (V, E)$ an artificial neural network, where $V = \{U_1, U_2, U_3, U_4\}$ and $E = \{(U_1, U_2, 1), (U_1, U_3, 4), (U_2, U_3, 3), (U_3, U_1, -2), (U_3, U_4, -2), (U_4, U_2, 1)\}$. U_1 and U_2 are input neurons with one input, x_1 and x_2 respectively, whereas U_3 is an output neuron. Represent its network structure.

Solution:



$$\begin{bmatrix} \infty & \infty & 4 & \infty \\ 1 & \infty & \infty & 1 \\ -2 & 3 & \infty & \infty \\ \infty & \infty & -2 & \infty \end{bmatrix}$$

Figure 13: On the left, the pictorial representation of the neural network. On the right, its adjacency matrix.

□

If the network structure of a neural network is acyclic graph (no loops and no directed cycles), the neural network is said to be a **feed forward neural network**. If instead it has at least one loop and/or one cycle, it is referred to as a **recurrent network**. The difference between the two is in how the information flows: in the former, information can only flow from the input neurons to the hidden neurons (if any) to the output neurons, meaning that it can only go "in one direction". In the latter, information can be fed back into the network from its own output to its own input (the neural network in Exercise 2.3.1 would be a recurrent neural network).

To understand how a neural network operates, it is necessary to start from single neurons. To each neuron $u \in U$ are assigned three real-valued quantities: the **network input** net_u, the **activation** act_u, and the **output** out_u. Each input neuron $u \in U_{\mathrm{in}}$ has a fourth quantity, the **external input** ext_u. Each neuron $u \in U$ also possesses three functions:

- The **network input function** $f_{\mathrm{net}}^{(u)} : \mathbb{R}^{2|\mathrm{pred}(u)| + p_l(u)} \to \mathbb{R}$;
- The **activation function** $f_{\mathrm{act}}^{(u)} : \mathbb{R}^{p_k(u)} \to \mathbb{R}$;
- The **output function** $f_{\mathrm{out}}^{(u)} : \mathbb{R} \to \mathbb{R}$.

Where $p_l(u) = \sigma_1, ..., \sigma_l$ and $p_k(u) = \theta_1, ..., \theta_k$ are two lists of (real) parameters that depend on the type and on the number of arguments of the function.

The network input function $f_{\mathrm{net}}^{(u)}$ computes the network input net_u from the inputs $\mathrm{in}_{u,v_1}, ..., \mathrm{in}_{u,v_n}$, the weights $w_{u,v_1}, ..., w_{u,v_n}$ and the parameters (if any) in $p_l(u)$. Each input is itself the output of a preceding neuron, meaning that $\mathrm{in}_{u,v_1} = \mathrm{out}_{v_1}, ..., \mathrm{in}_{u,v_n} = \mathrm{out}_{v_n}$. The network input function is often written with vector arguments:

$$f_{\mathrm{net}}^{(u)}(\boldsymbol{w}_u, \mathbf{in}_u) = f_{\mathrm{net}}^{(u)}\left(w_{u,v_1}, ..., w_{u,v_n}, \mathrm{in}_{u,v_1}, ..., \mathrm{in}_{u,v_n}\right) = f_{\mathrm{net}}^{(u)}\left(w_{u,v_1}, ..., w_{u,v_n}, \mathrm{out}_{v_1}, ..., \mathrm{out}_{v_n}\right)$$

The activation function $f_{\mathrm{act}}^{(u)}$ computes the new **activation** act_u of the neuron u, representing the "stimulation" that a neuron receives. The value of this computation depends on the network input, on the parameters (if any) in $p_k(u)$ and, in the case of a recurrent neural network, on a feedback of the current activation. If the neuron is an input neuron, its initial activation is given by the external input ext_u. The value of the activation is then used by the output function $f_{\mathrm{out}}^{(u)}$ to compute the output of the neuron.

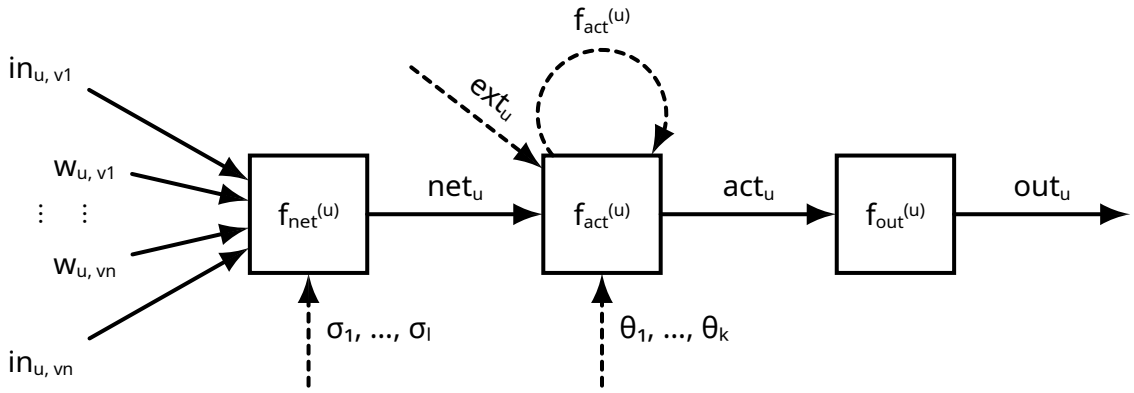


Figure 14: A very generic depiction of the structure of a single neuron.

This structure should hint at the fact that a neural network is simply a more generalized network of TLUs. In particular, it's possible to construct a TLU from a neural network with the following choices of network input function, activation function and output function for all of its neurons:

$$f_{\mathrm{net}}^{(u)}(\boldsymbol{w}_u, \mathbf{in}_u) = \sum_{v \in \mathrm{pred}(u)} w_{u,v}\mathrm{in}_{u,v} \qquad f_{\mathrm{act}}^{(u)}(\mathrm{net}_u, \theta) = H(\mathrm{net}_u, \theta) \qquad f_{\mathrm{out}}^{(u)}(\mathrm{act}_u) = \mathrm{act}_u$$

A single neuron can operate "in a vacuum", meaning that it can receive input and deliver output without interfering with the operation of other neurons. On the other hand, the neurons in a neural network depend on each other for their input and output. For this reason, it is important to distinguish the operational state of a neural network into an **input phase**, in which external input is fed into the neural network, and a **work phase**, in which the output of the neural network is computed.

In the input phase, neurons have their network input function bypassed completely: the activation of input neurons is entirely given by the external input, whereas other neurons have their activation set to an arbitrary value. In addition, the output function is applied to the activations, so that all neurons produce initial outputs, even if meaningless. The neural network does not transition from the input phase until all external input has been received by all input neurons.

In the work phase, the external inputs of the input neurons are ignored and the activations and outputs of the neurons are (re)computed, possibly multiple times, applying the network input function, the activation function and the output function. Input neurons that have no input from other neurons, but only from outside, simply maintain the value of their activation. The recomputations are terminated either if the network reaches a stable state, that is, if further recomputations do not change the outputs of the neurons anymore, or if a predetermined number of recomputations has been carried out.

The neural network architecture does not specify a precise order in which the computations are to be carried out. All neurons might recompute their outputs at the same time (**synchronous update**), drawing on the old outputs of their predecessors, or it might be possible to define an update order in which neurons compute their outputs one after another (**asynchronous update**), so that the new outputs of other neurons may already be used as inputs for subsequent computations.

For a feed forward network the computations usually follow a **topological ordering** of the neurons. A topological ordering is an enumeration of the vertices of a directed graph such that all edges are directed from a vertex with a lower number to a vertex with a higher number. A topological ordering is possible only if the graph is acyclic, which for feed forward networks is indeed the case. Updating a feed forward network employing topological ordering ensures that all inputs of a neuron have already been computed, before it (re)computes its own activation and output.

Note that for recurrent networks the final output may depend on the order in which the neurons recompute their outputs as well as on how many recomputations are carried out. This is because the number of recomputations influences how many times (and how much) of its output is fed back as input.

Unlike a network of TLUs, neural networks can be trained. The process is similar to the training of a single TLU: presenting the network a set of input-output pairs and tuning its weights and its parameters step-by-step while minimizing an error function. Most training tasks fall into two categories: fixed learning tasks and free learning tasks.

A **fixed learning task** L_{fixed} for a neural network with n input neurons $U_{\text{in}} = \{u_1, ..., u_n\}$ and m output neurons $U_{\text{out}} = \{v_1, ..., v_m\}$ is a set of training patterns $l = \left(\boldsymbol{i}^{(l)}, \boldsymbol{o}^{(l)}\right)$, each consisting of an **input vector** $\boldsymbol{i}^{(l)} = \text{ext}_{u_1}^{(l)}, ..., \text{ext}_{u_n}^{(l)}$ and an **output vector** $\boldsymbol{o}^{(l)} = o_{v_1}^{(l)}, ..., o_{v_m}^{(l)}$. *Solving* a fixed learning task means training a neural network such that, for all training patterns $l \in L_{\text{fixed}}$, when fed $\boldsymbol{i}^{(l)}$ as external input its output is $\boldsymbol{o}^{(l)}$, or at least as close as possible.

In order to determine how well a neural network solves a fixed learning task, an error function is employed, which measures how well the outputs of the network match the outputs in the training patterns. The cumulative error has the following expression:

$$e = \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{v \in U_{\text{out}}} e_v = \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} e_v^{(l)}$$

Which is the sum of the errors of each neuron with the corresponding output in a training pattern, summed over all training patterns.

As for the choice of the error function, simply taking the difference between the outputs of the network and the outputs in the patterns is not a good idea, since positive and negative errors may cancel out. A common choice of error function for fixed learning tasks is the **Mean Squared Error function** (**MSE**):

$$e = \sum_{l \in L_{\text{fixed}}} \left(o_{v_1}^{(l)} - \text{out}_{v_1}^{(l)}\right)^2 + ... + \left(o_{v_m}^{(l)} - \text{out}_{v_m}^{(l)}\right)^2 = \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \text{out}_v^{(l)}\right)^2$$

That is, the sum over all training examples of the squared difference between the outputs in the given patterns and the outputs of the network. This type of error function has the advantage of being differentiable everywhere, which means that it is easy to optimize (zeroing its gradient).

A fixed learning task is considered solved when the value of the error function is sufficiently small (or, ideally, 0). Note how the error function is not computed one pattern at a time, but instead after all patterns have been presented to the network. This is necessary to guarantee that the training process actually converges.

A **free learning task** L_{free} for a neural network with n input neurons $U_{\text{in}} = \{u_1, ..., u_n\}$ is a set of training patterns $l = \left(\boldsymbol{i}^{(l)}, \boldsymbol{o}^{(l)}\right)$, each consisting of an **input vector** $\boldsymbol{i}^{(l)} = \text{ext}_{u_1}^{(l)}, ..., \text{ext}_{u_n}^{(l)}$. Solving a free learning task is trickier, because there are no outputs in the training pattern to which the output of the neural network can be compared. Informally speaking, a free learning task is solved when similar inputs are consistently mapped to similar outputs. This is often done with the help of so-called **distance functions**.

Fixed learning tasks are also referred to as **supervised learning**, where the term "supervised" hints at the fact that the values of the weights and parameters of the neural network are tuned under the "guidance" of the output vector. On the other hand, free learning tasks are also referred to as **unsupervised learning**, because there is no counterexample (no "guidance") that can test the quality of the output of the neural network.

It is advisable to normalize the external inputs of a neural network to prevent biases during the training session. If the inputs have very different orders of magnitude, the training process will be skewed. Given the mean and standard deviation of the external input of each k-th (input) neuron:

$$\mu_k = \frac{1}{|L|} \sum_{l \in L} \text{ext}_{u_k}^{(l)} \qquad\qquad \sigma_k = \sqrt{\frac{1}{|L|} \sum_{l \in L} \left(\text{ext}_{\mu_k}^{(l)} - \mu_k\right)^2}$$

The normalized external input is given by:

$$\overline{\text{ext}}_{u_k}^{(l)} = \frac{\text{ext}_{u_k}^{(l)} - \mu_k}{\sigma_k}$$

This normalization can be carried out as a preprocessing step (before feeding it to the network) or, in a feed forward network, by the output function of the input neurons.

The artificial neural network architecture has been constructed with real numbers in mind (all three functions have the real numbers as both domain and codomain). However, it is possible to have neural networks manipulate nominal attributes as long as they can be encoded into numbers.

The most intuitive way to do so would be to take the different modes of the attribute and assign to each one of them an integer value. The problem of this approach is that it makes little sense to use an encoding implying an order when the attribute does not. A better solution is what is called **1-in-n encoding**, where to each mode of the attribute is assigned a binary string of length equal to the

number of possible modes. Such strings are all 0 except for a single 1, different from mode to mode. This way, all possible values are equally taken into account in a completely unbiased way.

2.4. Multilayer perceptrons

2.4.1. Structure of a multilayer perceptron

An **r-layer perceptron**, or **multilayer perceptron** (**MLP** for short) is the most well-known class of feed-forward neural networks, whose neurons can be arranged into r "stacks", or **layers**. A feed-forward neural network is classified as a multilayer perceptron if it possesses the following characteristics:

- Input neurons cannot also be output neurons, and vice versa: $U_{\text{in}} \cap U_{\text{out}} = \emptyset$;
- Each of the $r-2$ hidden layers must be disjointed with the others:

$$U_{\text{hidden}} = U_{\text{hidden}}^{(1)} \cup ... \cup U_{\text{hidden}}^{(r-2)} = \bigcup_{i=1}^{r-2} U_{\text{hidden}}^{(i)} \qquad U_{\text{hidden}}^{(i)} \cap U_{\text{hidden}}^{(j)} = \emptyset, \ \forall i,j \in \{1,...,r-2\}$$

  That is, no hidden neuron can belong to more than one hidden layer at the same time;
- Connections can only exist between nodes of subsequent layers. There cannot be connections between two layers that aren't adjacent and there cannot be connections between nodes of the same layer[5]:

$$C \subseteq \left(U_{\text{in}} \times U_{\text{hidden}}^{(1)}\right) \cup \left(\bigcup_{i=1}^{r-3} U_{\text{hidden}}^{(i)} \times U_{\text{hidden}}^{(i+1)}\right) \cup \left(U_{\text{hidden}}^{(r-2)} \times U_{\text{out}}\right)$$

- The output function of any neuron is the identity function;
- The network input function and the activation function of input neurons are the identity function (input neurons propagate the external input unchanged);
- The network input function of hidden neurons and output neurons is the weighted sum of their inputs and the corresponding weights:

$$\forall u \in U_{\text{hidden}} \cup U_{\text{out}}, \ f_{\text{net}}^{(u)}\left(w_{u_1}, ..., w_{u_n}, \text{in}_{u_1}, ..., \text{in}_{u_n}\right) = f_{\text{net}}^{(u)}(\boldsymbol{w}_u, \mathbf{in}_u) = \sum_{v \in \text{ pred}(u)} w_{u,v} \cdot \text{out}_v$$

- The activation function of hidden neurons is a **sigmoid function**, meaning any monotonical non-decreasing function in the form:

$$f : \mathbb{R} \mapsto [0,1], \quad \text{with} \lim_{x \to -\infty} f(x) = 0 \ \text{ and } \lim_{x \to +\infty} f(x) = 1$$

  These functions have a characteristic S-shape;
- The activation function of output neurons is either a sigmoid function or any linear function $f_{\text{act}}(\text{net}, \theta) = \alpha \ \text{net} - \theta$, with $\alpha \in \mathbb{R}$.

Note how a multilayer perceptron is very close to a network of TLUs. As a matter of fact, the only difference is that the activation function of a multilayer perceptron is not a binary function.

[5] Some sources also require that each node of a layer is connected to each node of the following layer.

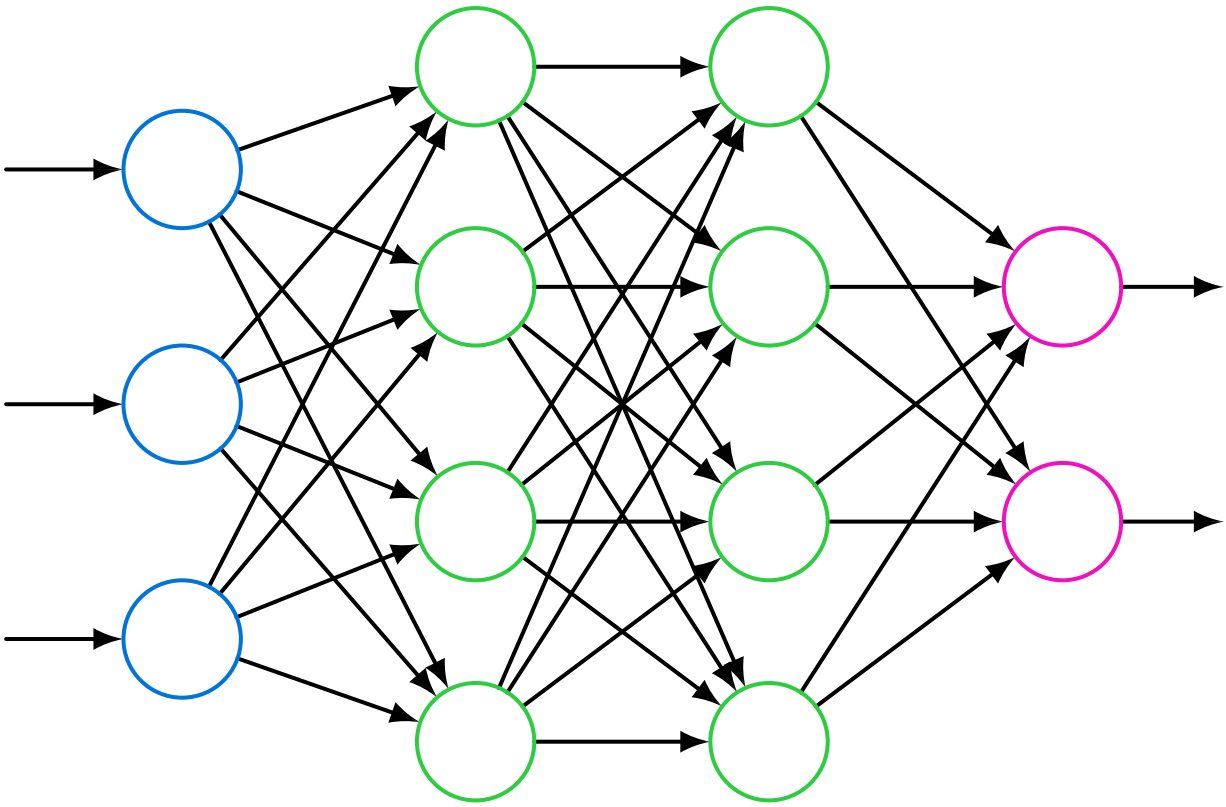


Figure 15: Structure of a generic multilayer perceptron

There are many functions that can be classified as sigmoids. The simplest one is the Heaviside function, already employed in TLUs:

$$f_{\text{act}}(\text{net}, \theta) = H(\text{net}, \theta) = \begin{cases} 1 \text{ if net} \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

This function is both very easy to conceptualize, and very efficient to implement in hardware. This is because, as it was done for the TLUs, it is possible to move the threshold into the weighted sum and obtain an equivalent function that outputs 0 if the weighted sum is negative (less than 0) and outputs 1 if positive (greater than 0), and this control can be performed simply by looking at the most significant bit of the result of the weighted sum[6]. In particular, since positive numbers are encoded in hardware with a most significant bit of 0 and negative number with a most significant bit of 1, the result is just the negation of the most significant bit of the weighted sum.

The issues of the function lie in its abrupt jump from one value to another. This is both from a mathematical standpoint, because it renders the function not differentiable everywhere, and from a logical standpoint, since it models neurons that either fire or not fire, without nuances in between. Also, since it's not injective, it is not invertible.

An improvement of the Heaviside function is the **semi-linear function**, that grows linearly inside an interval and remains constant outside of those boundaries:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1 & \text{if net} > \theta + \frac{1}{2} \\ 0 & \text{if net} < \theta - \frac{1}{2} \\ (\text{net} - \theta) + \frac{1}{2} & \text{otherwise} \end{cases}$$

[6]Weighted sums can be computed efficiently by GPUs, since they are specifically designed to efficiently compute convolutions.

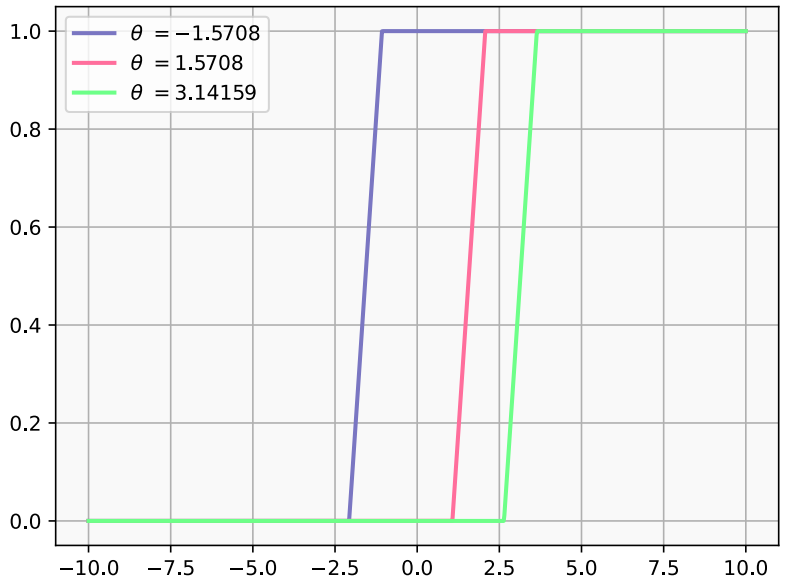


Figure 16: Plot of the semi-linear function, with three different choices of θ

The semi-linear function "smooths" the transition between the two extremes, giving a more nuanced and accurate model of a firing neuron. It is still problematic, however, since for example it is still not injective.

An even smoother function is the **sine up to saturation function**:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1 & \text{if net} > \theta + \frac{\pi}{2} \\ 0 & \text{if net} < \theta - \frac{\pi}{2} \\ \frac{1}{2}(\sin(\text{net} - \theta) + 1) & \text{otherwise} \end{cases}$$

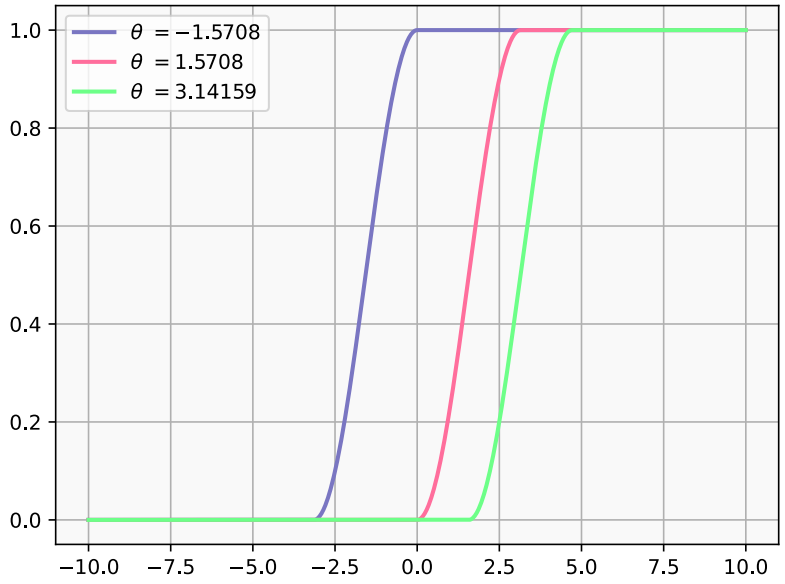


Figure 17: Plot of the sine up to saturation function, with three different choices of θ

The first historic example of a widely deployed activation function is the **logistic function**[7]:

[7]This function is sometimes referred to, improperly, as the *sigmoid* function. This is due to the fact that, out of all the sigmoids, the logistic function is the most known.

$$f_{\text{act}}(\text{net}, \theta) = \frac{1}{1 + e^{-(\text{net} - \theta)}}$$

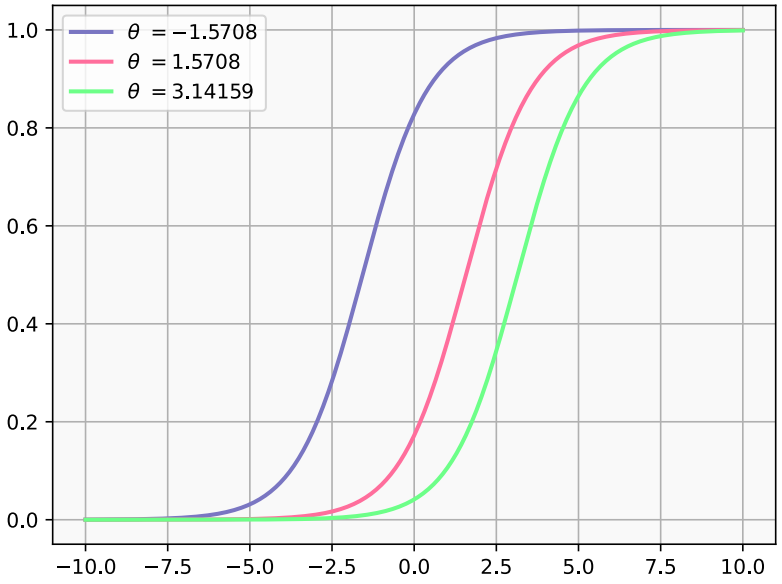


Figure 18: Plot of the logistic function, with three different choices of θ

This function is not only continuous everywhere, but also invertible and differentiable everywhere. Furthermore, its derivative is very easy to compute:

$$\begin{aligned}
\frac{d}{d\ \text{net}} f_{\text{act}}(\text{net}, \theta) &= \frac{d}{d\ \text{net}} \left(\frac{1}{1 + e^{-(\text{net} - \theta)}} \right) = \frac{d}{d\ \text{net}} \left(\left(1 + e^{-(\text{net} - \theta)}\right)^{-1} \right) = \\
&= -\left(1 + e^{-(\text{net} - \theta)}\right)^{-2} \frac{d}{d\ \text{net}} \left(1 + e^{-(\text{net} - \theta)}\right) = \\
&= -\frac{1}{\left(1 + e^{-(\text{net} - \theta)}\right)^2} \left(\frac{d}{d\ \text{net}} 1 + \frac{d}{d\ \text{net}} e^{-(\text{net} - \theta)} \right) = \\
&= -f_{\text{act}}^2(\text{net}, \theta) \left(0 - e^{-(\text{net} - \theta)} \frac{d}{d\ \text{net}} (\text{net} - \theta) \right) = \\
&= f_{\text{act}}^2(\text{net}, \theta) \left(e^{-(\text{net} - \theta)} \right) = f_{\text{act}}^2(\text{net}, \theta) \left(1 + e^{-(\text{net} - \theta)} - 1 \right) = \\
&= f_{\text{act}}^2(\text{net}, \theta) (f_{\text{act}}^{-1}(\text{net}, \theta) - 1) = f_{\text{act}}(\text{net}, \theta)(1 - f_{\text{act}}(\text{net}, \theta))
\end{aligned}$$

The successive derivatives are as well:

$$\begin{aligned}
\frac{d^2}{d\ \text{net}} (f_{\text{act}}(\text{net}, \theta) &= \frac{d}{d\ \text{net}} (f_{\text{act}}(\text{net}, \theta)(1 - f_{\text{act}}(\text{net}, \theta))) = \\
&= (1 - f_{\text{act}}(\text{net}, \theta)) \frac{d}{d\ \text{net}} f_{\text{act}}(\text{net}, \theta) + f_{\text{act}}(\text{net}, \theta) \frac{d}{d\ \text{net}} (1 - f_{\text{act}}(\text{net}, \theta)) = \\
&= (1 - f_{\text{act}}(\text{net}, \theta)) \frac{d}{d\ \text{net}} f_{\text{act}}(\text{net}, \theta) - f_{\text{act}}(\text{net}, \theta) \frac{d}{d\ \text{net}} f_{\text{act}}(\text{net}, \theta) = \\
&= (1 - f_{\text{act}}(\text{net}, \theta) - f_{\text{act}}(\text{net}, \theta)) \frac{d}{d\ \text{net}} f_{\text{act}}(\text{net}, \theta) = \\
&= (1 - 2 f_{\text{act}}(\text{net}, \theta)) \frac{d}{d\ \text{net}} f_{\text{act}}(\text{net}, \theta)
\end{aligned}$$

Sigmoid functions having $[0, 1]$ as codomain are called **unipolar sigmoid functions**. Functions having all the traits of a sigmoid function whose codomain is $[-1, 1]$ are called **bipolar sigmoid functions**.

Any bipolar sigmoid function can be converted into a (unipolar) sigmoid function by adding 1 and dividing by 2.

One example of bipolar sigmoid function is the **hyperbolic tangent**, closely resembling the logistic function:

$$f_{\text{act}}(\text{net}, \theta) = \tanh(\text{net})$$

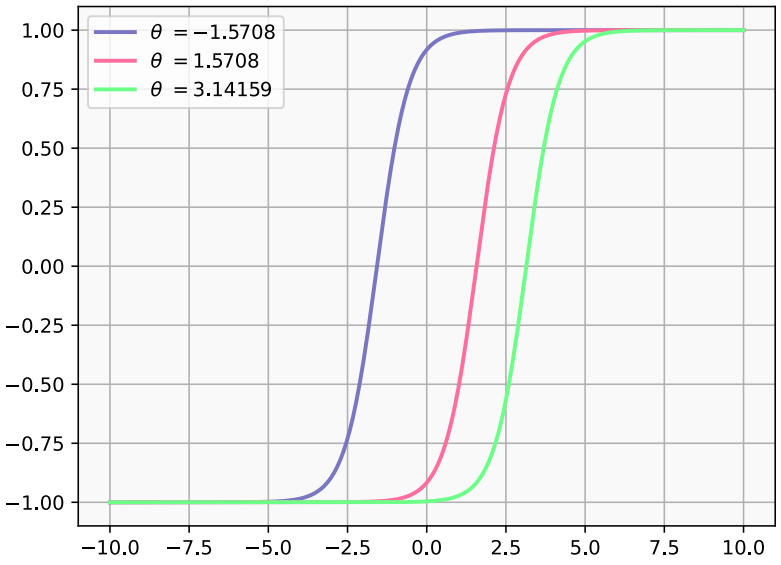


Figure 19: Plot of the hyperbolic tangent, with three different choices of θ

A clear advantage of having a weighted summation as the network input function of a multilayer perceptron is that it translates naturally into a matrix multiplication. Let $U_1 = (v_1, ..., v_m)$ and $U_2 = (u_1, ..., u_n)$ be two adjacent layers (assume U_2 is right after U_1) in a multilayer perceptron. The network input of the U_2 layer as a whole can be written as:

$$\mathbf{net}_{U_2} = \begin{pmatrix} \text{net}_{u_1} \\ \text{net}_{u_2} \\ \dots \\ \text{net}_{u_m} \end{pmatrix} = W_{U_2,U_1}\mathbf{in}_{U_2} = \begin{pmatrix} w_{u_1,v_1} & w_{u_1,v_2} & \dots & w_{u_1,v_m} \\ w_{u_2,v_1} & w_{u_2,v_2} & \dots & w_{u_2,v_m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{u_n,v_1} & w_{u_n,v_2} & \dots & w_{u_n,v_m} \end{pmatrix} \begin{pmatrix} \text{in}_{u_1} \\ \text{in}_{u_2} \\ \dots \\ \text{in}_{u_m} \end{pmatrix}$$

Where w_{u_i,v_j} is the weight of the connection coming into the i-th neuron of U_2 and going out of the j-th neuron of U_1.

Exercise 2.4.1.1: Rewrite the network of TLUs in Exercise 2.2.3.1 as a multilayer perceptron.

Solution: An equivalent construction is as follows:

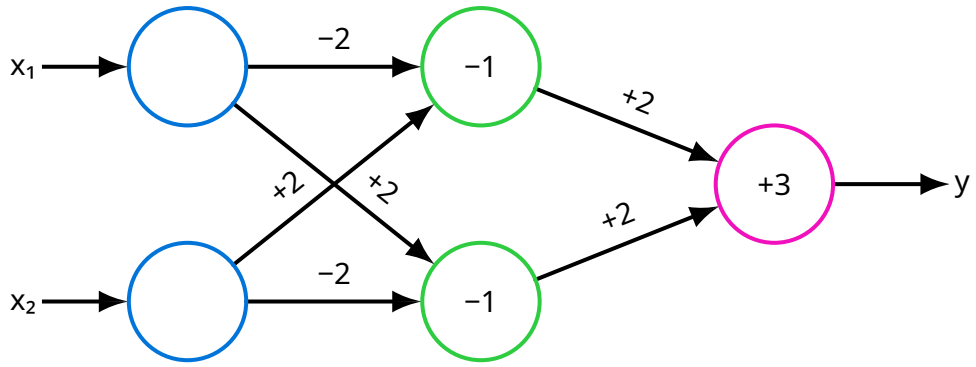


Figure 20: A multilayer perceptron equivalent to the network of TLUs in Exercise 2.2.3.1

Where two input neurons on the left are added. These accomodate the fact that the external input of the input layers of a multilayer perceptron cannot be weighted (a network of TLUs did not have this restriction). These receive the external input and transmit it unchanged.

The output function can be set to be the identity function for any neuron. The activation function is the Heaviside function, using as parameter θ the threshold of the TLU. The network input function for the hidden layer (in green) and the output layer (in red) can be written in matrix form:

$$W_{U_{\text{hidden}},U_{\text{in}}} = \begin{pmatrix} -2 & 2 \\ 2 & -2 \end{pmatrix} \qquad W_{U_{\text{out}},U_{\text{hidden}}} = \begin{pmatrix} 2 & 2 \end{pmatrix}$$

For example, suppose that the external input is $x_1 = 1$ and $x_2 = 1$. The input of the hidden layer is:

$$\mathbf{net}_{U_{\text{hidden}}} = W_{U_{\text{hidden}},U_{\text{in}}} \mathbf{in}_{U_{\text{hidden}}} = \begin{pmatrix} -2 & 2 \\ 2 & -2 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \cdot 1 + 2 \cdot 1 \\ 2 \cdot 1 + (-2) \cdot 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

The activation of the hidden layer is given by applying the Heaviside function to both entries of the vector, having -1 as the θ parameter:

$$\mathbf{act}_{U_{\text{hidden}}} = \begin{pmatrix} f_{\text{act}}(0,-1) \\ f_{\text{act}}(0,-1) \end{pmatrix} = \begin{pmatrix} H(0,-1) \\ H(0,-1) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

And the output is left unchanged by the output function. The input of the output layer is:

$$\mathbf{net}_{U_{\text{out}}} = W_{U_{\text{out}},U_{\text{hidden}}} \mathbf{in}_{U_{\text{out}}} = \begin{pmatrix} 2 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 2 \cdot 1 + 2 \cdot 1 = 4$$

Which means that the output of the multilayer perceptron is, as expected:

$$\mathbf{out}_{U_{\text{out}}} = \mathbf{act}_{U_{\text{out}}} = H(4,3) = 1$$

□

The matrix notation for multilayer perceptrons can shed light on why it's better to have non-linear activation functions. This is because a multilayer perceptron that has linear activation functions has very limited expressiveness.

Theorem 2.4.1.1: Any r-layer perceptron whose neurons employ linear activation functions and linear output functions can always be reduced to an equivalent 2-layer perceptron.

Proof: Consider a multilayer perceptron whose neurons have linear activation functions and linear output functions. Both functions are in the form $f_{\text{act}}(\text{net}, \theta) = \alpha \text{ net} - \theta$. Let U_a, U_b and U_c be three consecutive layers (U_c comes after U_b and U_b comes after U_a) of the perceptron.

The network input vector of the U_b layer is $\mathbf{net}_{U_b} = W_{U_b,U_a} \cdot \mathbf{in}_{U_b} = W_{U_b,U_a} \cdot \mathbf{out}_{U_a}$. The activation vector of the U_b layer is given by applying the (linear, in this case) function $f_{\text{act}}(\text{net}, \theta)$ to each entry:

$$\mathbf{act}_{U_b} = \begin{pmatrix} \text{act}_{u_1} \\ \text{act}_{u_2} \\ \vdots \\ \text{act}_{u_n} \end{pmatrix} = \boldsymbol{D}_{\text{act}} \cdot \mathbf{net}_{U_b} - \boldsymbol{\theta} = \begin{pmatrix} \alpha_{u_1} & 0 & \dots & 0 \\ 0 & \alpha_{u_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \alpha_{u_n} \end{pmatrix} W_{U_b,U_a} \begin{pmatrix} \text{in}_{u_1} \\ \text{in}_{u_2} \\ \vdots \\ \text{in}_{u_n} \end{pmatrix} - \begin{pmatrix} \theta_{u_1} \\ \theta_{u_2} \\ \vdots \\ \theta_{u_n} \end{pmatrix}$$

Where the $u_1, \dots, u_n$ neurons are neurons of the U_b layer. Since the output functions are also linear:

$$\mathbf{out}_{U_b} = \begin{pmatrix} \text{out}_{u_1} \\ \text{out}_{u_2} \\ \vdots \\ \text{out}_{u_n} \end{pmatrix} = \boldsymbol{D}_{\text{out}} \cdot \mathbf{act}_{U_b} - \boldsymbol{\xi} = \begin{pmatrix} \beta_{u_1} & 0 & \dots & 0 \\ 0 & \beta_{u_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \beta_{u_n} \end{pmatrix} \begin{pmatrix} \text{act}_{u_1} \\ \text{act}_{u_2} \\ \vdots \\ \text{act}_{u_n} \end{pmatrix} - \begin{pmatrix} \xi_{u_1} \\ \xi_{u_2} \\ \vdots \\ \xi_{u_n} \end{pmatrix}$$

Substituting the expression for $\mathbf{act}_{U_b}$ into the expression for $\mathbf{out}_{U_b}$:

$$\begin{aligned} \mathbf{out}_{U_b} &= \boldsymbol{D}_{\text{out}} \cdot \mathbf{act}_{U_b} - \boldsymbol{\xi} = \\ &= \boldsymbol{D}_{\text{out}} \cdot \left(\boldsymbol{D}_{\text{act}} \cdot \mathbf{net}_{U_b} - \boldsymbol{\theta} \right) - \boldsymbol{\xi} = \\ &= \boldsymbol{D}_{\text{out}} \cdot \left(\boldsymbol{D}_{\text{act}} \cdot \left(W_{U_b,U_a} \cdot \mathbf{in}_{U_b} \right) - \boldsymbol{\theta} \right) - \boldsymbol{\xi} = \\ &= \boldsymbol{D}_{\text{out}} \cdot \left(\boldsymbol{D}_{\text{act}} \cdot \left(W_{U_b,U_a} \cdot \mathbf{out}_{U_a} \right) - \boldsymbol{\theta} \right) - \boldsymbol{\xi} = \\ &= \left(\boldsymbol{D}_{\text{out}} \cdot \boldsymbol{D}_{\text{act}} \cdot W_{U_b,U_a} \right) \cdot \mathbf{out}_{U_a} + (-\boldsymbol{D}_{\text{out}} \cdot \boldsymbol{\theta} - \boldsymbol{\xi}) = \\ &= \boldsymbol{L}_{a,b} \cdot \mathbf{out}_{U_a} + \boldsymbol{R}_{a,b} \end{aligned}$$

If the same process were to be applied to the third layer, one would get:

$$\mathbf{out}_{U_c} = \boldsymbol{L}_{b,c} \cdot \mathbf{out}_{U_b} + \boldsymbol{R}_{b,c}$$

Substituting the expression for $\mathbf{out}_{U_b}$ in the expression for $\mathbf{out}_{U_c}$:

$$\begin{aligned} \mathbf{out}_{U_c} &= \boldsymbol{L}_{b,c} \cdot \mathbf{out}_{U_b} + \boldsymbol{R}_{b,c} = \boldsymbol{L}_{b,c} \cdot \left(\boldsymbol{L}_{a,b} \cdot \mathbf{out}_{U_a} + \boldsymbol{R}_{a,b} \right) + \boldsymbol{R}_{b,c} = \\ &= \left(\boldsymbol{L}_{b,c} \cdot \boldsymbol{L}_{a,b} \right) \cdot \mathbf{out}_{U_a} + \left(\boldsymbol{L}_{b,c} \cdot \boldsymbol{R}_{a,b} + \boldsymbol{R}_{b,c} \right) = \boldsymbol{L'}_{a,c} \cdot \mathbf{out}_{U_a} + \boldsymbol{R'}_{a,c} \end{aligned}$$

This means that the three layers U_a, U_b and U_c can be "compressed" into the two layers U_a and U_c without any loss of precision. Since U_a, U_b and U_c were chosen arbitrarely, it's always possible to choose any triplet of layers and "compress" them into two. The same process could be applied to the network as many times as desired, which means that the number of layers can be arbitrarely decreased.

□

2.4.2. Approximating functions using a multilayer perceptron

Multilayer perceptrons that use non-linear activation functions are more powerful than any network of TLUs, and can encode a much wider range of functions. As a matter of fact, it can be proven that almost all functions can be encoded into a multilayer perceptron with arbitrary accuracy, but this proof is very convoluted. Nonetheless, a "relaxed" version of this result, that concerns itself only with Riemann-integrable functions, is fairly approachable.

Let f be a function that is Riemann-integrable. To prove that there's at least one multilayer perceptron that can encode f, notice how it's always possible to approximate f with arbitrary precision as a series of **stepwise functions**. A single stepwise function of extremes a and b and height h is defined as:

$$S_{a,b,h}(x) = \begin{cases} h \text{ if } a \leq x \leq b \\ 0 \text{ otherwise} \end{cases}$$

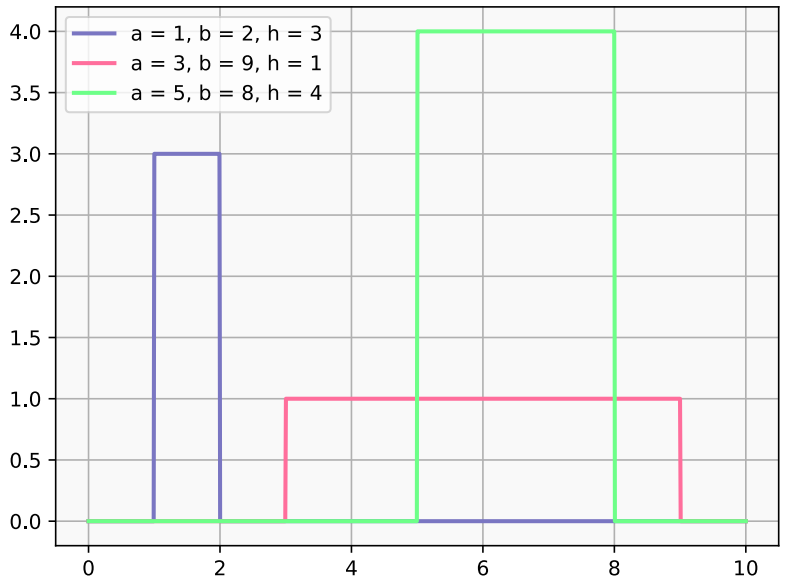


Figure 21: Plot of the stepwise function, with three different choices for the parameters

To approximate f this way, the domain of the function is first partitioned into n steps, delimited by the border points $x_1, x_2, ..., x_n$. f is then rewritten as a piecewise function, where each i-th piece has x_i and x_{i+1} as extremes and $f((x_i + x_{i+1})/2)$, the function evaluated at the midpoint of the two extremes, as height:

$$f(x) \approx \begin{cases} f\left(\frac{x_1 + x_2}{2}\right) & \text{if } x_1 \leq x < x_2 \\ f\left(\frac{x_2 + x_3}{2}\right) & \text{if } x_2 \leq x < x_3 \\ \vdots & \\ f\left(\frac{x_{n-1} + x_n}{2}\right) & \text{if } x_{n-1} \leq x < x_n \end{cases}$$

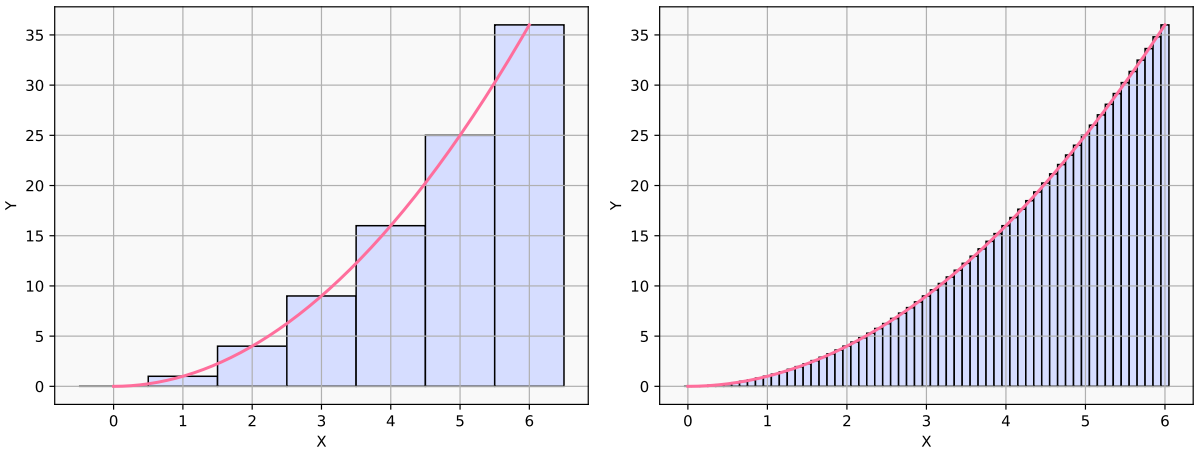


Figure 22: The function $f(x) = x^2$, whose domain is restriced to $[0, 6]$, approximated by a series of stepwise functions. On the left, a rough approximation. On the right, a refined approximation.

The idea is to first show that a multilayer perceptron can encode a single stepwise function, then show that it can also encode any piecewise approximation with arbitrary accuracy. This approach can be used not only for multilayer perceptrons, but for most neural networks in general.

Theorem 2.4.2.1: Any Riemann-integrable function can be encoded with arbitrary accuracy into a multilayer perceptron of four layers.

Proof: A single stepwise function $S_{a,b,h}(x)$ can be encoded into a multilayer perceptron in the following way:

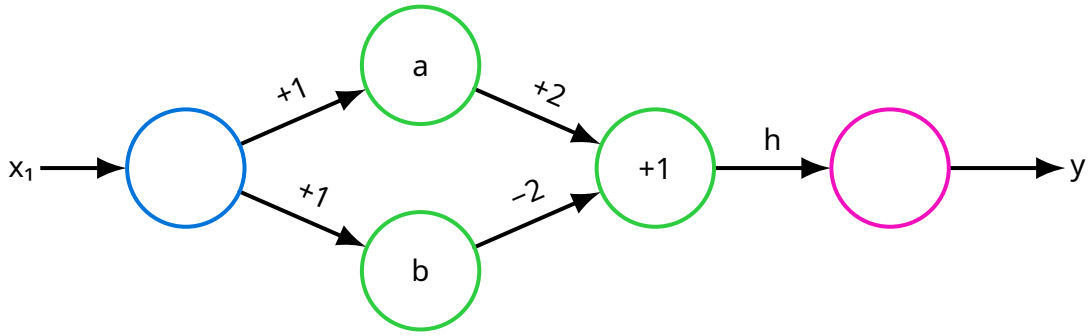


Figure 23: A multilayer perceptron that encodes a stepwise function of extremes a and b and of height h.

If the input is smaller than a, then both hidden layers output 0, which is fed into the second hidden layer outputting 0 as well. If the input lies between a and b, the hidden neuron at the top of the first hidden layer outputs 1, the bottom one 0. The hidden neuron in the second hidden layer receives as input $(2 \cdot 1) + (0 \cdot 1) = 2$, hence it outputs 1. The output neuron then outputs $1 \cdot h = h$. If the input is bigger than b, then the output of the two hidden layers cancel out, and the final output is 0.

The entire 4-layer perceptron is a generalized form of this perceptron. The input layer has a single neuron, whose external input is the point in the domain of the function that one wishes to approximate. The output layer is also single neuron, receiving the input and transmitting it unchanged. All hidden neurons have a step function as activation function, whereas the input and output neuron have the identity function.

A neuron in the first hidden layer of the perceptron is added for each border point $x_1, ..., x_n$. The weights of the incoming connections of said nodes are set to 1, and the parameter θ of these nodes is the border point itself. This way, only neurons whose θ parameter is smaller than the output of the input layer (the external input, that is) will output 1.

A neuron is added to the second hidden layer for each step $[x_1, x_2], [x_2, x_3], ..., [x_{n-1}, x_n]$. The output of each of these neurons will be 1 if the external input is greater than x_i, but less than x_{i+1}. This way, there is one and only neuron in the second hidden layer whose output is 1: the one that lies in the interval approximating its evaluation.

Let $\overline{x}$ be the external input fed into the network, and let the cutoff points of the steps be $x_1, x_2, ..., x_n$. Suppose that:

$$x_1 < x_2 < ... < x_i \leq \overline{x} \leq x_{i+1} < ... < x_n$$

The neurons of the first hidden layer that will output 1 are those having as θ parameter $x_1, x_2, ..., x_i$, whereas those having $x_{i+1}, ..., x_n$ will output 0. As for the second hidden layer, the first $i - 1$ neurons will output 0, because the incoming weight of a neuron is cancelled by the incoming weight of the following neuron. However, also the last $i + 1$ neurons will output 0, because their incoming inputs are all 0. The only neuron that will output 1 is the i-th, because it receives a positive weight with a positive output and a negative weight with a null output; it's total incoming input is $+2$ and $H(2, 1) = 1$.

The weights of the output layer are the approximated values of evaluating the function in each interval. Since only one input to the output neuron is not null, its output will precisely be the approximation of the value of the function with the given (external) input.

Increasing the number of neurons (of step functions) give a better and better approximation. If the step sizes were to become infinitesimally small, the approximation would be perfect. Since there is no (theoretical) limit on the number of neurons that could be added, the function can be encoded with any degree of approximation, and therefore the result is proven.

□

Exercise 2.4.2.1: Refer to Figure 22 and construct a 4-layer perceptron that can approximate $f(x) = x^2$.

Solution: Evaluating the function at the midpoints of the intervals gives $2.25, 6.25, 12.25, 20.25, 30.25$. Therefore:

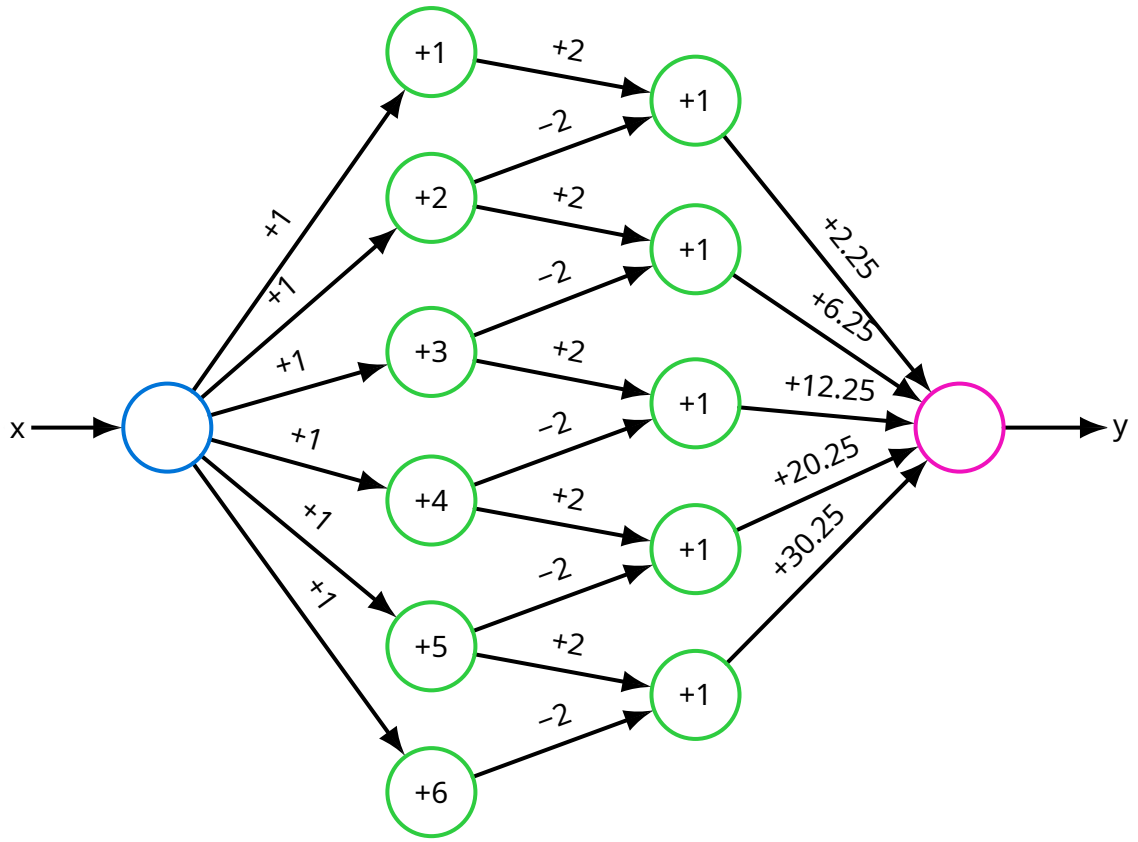


Figure 24: A 4-layer perceptron that approximates $f(x)$ in the $[0, 6]$ interval.

□

Theorem 2.4.2.1 does not restrict itself to continuous functions, since the definition of a Riemann-integrable does not require continuity[8]. However, if the function is continuous, the same result can be achieved with just 3 layers instead of 4 by using a slightly different approach.

Theorem 2.4.2.2: Any continuous Riemann-integrable function can be approximated with arbitrary accuracy by a multilayer perceptron of three layers.

Proof: This is done by taking into account not the absolute height of a step, but its relative height: the difference between the current step and the previous step.

A 3-layer perceptron of this kind is very similar to the previous 4-layer perceptron. The only differences are that the second hidden layer is removed and that the weights coming into the output layer are the relative step heights. This way, the first part of the computation behaves as in the previous case: only the neurons whose θ parameter is smaller than the external input will output 0. Now, however, the height differences of the steps are summed directly, "reconstructing" the desired height[9]. Since, again, the accuracy of the approximation can be increased at will, the result is proven. □

[8] Riemann-integrable but discontinuous functions are said to be *continuous almost everywhere*. This is because, despite not being continuous, they still behave "nicely enough" to be integrated.

[9] This "shortcut" would not work for non-continuous functions, since there is no guarantee that summing all relative heights up to a given point is actually the desired height

Exercise 2.4.2.2: Construct a 3-layer perceptron equivalent to the one in Exercise 2.4.2.1.

Solution: The relative heights of the step are $2.25 - 0 = 2.25, 6.25 - 2.25 = 4, 12.25 - 6.25 = 6, 20.25 - 12.25 = 8, 30.25 - 20.25 = 10$. This gives:

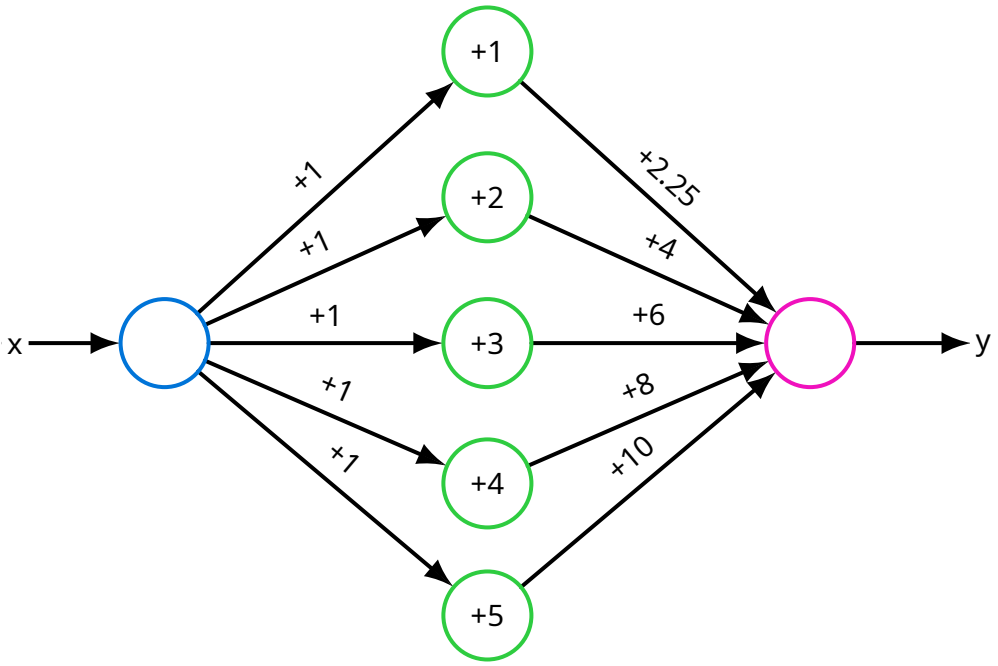


Figure 25: A 3-layer perceptron that approximates $f(x)$ in the $[0, 6]$ interval.

□

Theorem 2.4.2.1 has been stated for functions with a single argument, but can be readily extended to functions having arity $k > 1$. Instead of approximating the function with one-dimensional stepwise functions, it is approximated with k-dimensional "step functions" by partitioning the input space into k-dimensional hypercubes. This isn't transferred that easily to Theorem 2.4.2.2, because the dependency between the arguments has to be taken into account.

Note that the degree of approximation in Theorem 2.4.2.1 is given by the area between the function to approximate and the output of the multilayer perceptron. However, even though this area can be reduced at will, this does not mean that the difference between its output and the function to approximate is less than a certain error bound everywhere. That is, this area can only give an average measure of the quality of approximation.

For example, consider a case in which a function possesses a very thin spike (like a very steep gaussian curve) which is not captured by any stair step. In such a case the area between the function to represent and the output of the multilayer perceptron might be small (because the spike is thin), but at the location of the spike the deviation of the output from the true function value can nevertheless be considerable.

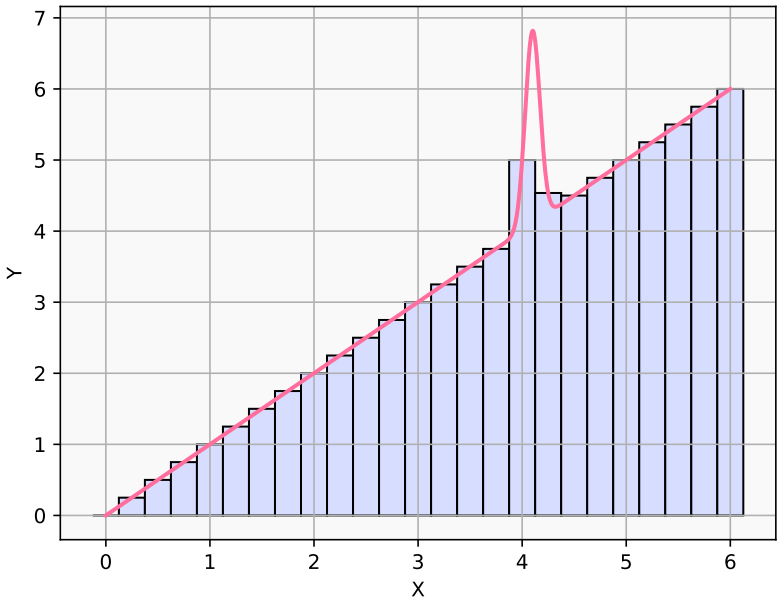


Figure 26: Plot of the function $x + \exp(1 - (10x - 41)^2)$, approximated with a series of stepwise functions of width $1/4$. The spike in the middle is not truly captured by any step, hence the local error is greater than the average error.

2.4.3. Training a multilayer perceptron

As it was the case for a single TLU, there is interest in having the neural network *learn* from examples which are the best choice of parameters and weights in order to approximate a given function, rather than fixing them by hand. TLUs employed a non-differentiable function (the stepwise function) as their "activation function", which meant that their error function had to be engineered in a very specific way. However, multilayer perceptrons don't abide by this restriction, and can have activation functions that are differentiable.

The central idea is to deduce the direction in which the weights and the parameters have to be changed at every step from the gradient of the error function. Since the gradient gives the direction of steepest descent, the correct direction is the opposite of the gradient (minimizing the error instead of maximising it). At each step, the gradient is computed, a small adjustment to the weights and the parameters is made and the process is repeated until the degree of approximation is satisfactory.

Consider a multilayer perceptron with r layers: let U_0 be the layer of input neurons, U_1 to U_{r-2} the layers of hidden neurons and U_{r-1} the layer of output neuron. Let e be the total error for a fixed learning task L_{fixed}. To understand how to the weights with respect to this function, it is necessary to explicitly rewrite the error in term of the weights. Assume that the multilayer perceptron has the logistic function as activation function for its neurons and the identity function as output function.

Since the input layer has no weights/parameters to be adapted, consider a single neuron u belonging either to an hidden layer or to the output layer, that is $u \in U_k$ with $0 < k \leq r - 1$. Its predecessors are given by $\text{pred}(u) = \{p_1, \ldots, p_n\} \in U_{k-1}$. The corresponding vector of weights, also embedding the θ parameter to ease calculations, is $\boldsymbol{w}_u = \left(-\theta_u, w_{u,p_1}, \ldots, w_{u,p_n}\right)$. The gradient of the total error function with respect to these weights is:

$$\nabla_{\boldsymbol{w}_u} e = \frac{\partial e}{\partial \boldsymbol{w}_u} = \left(-\frac{\partial e}{\partial \theta_u}, \frac{\partial e}{\partial w_{u,p_1}}, \ldots, \frac{\partial e}{\partial w_{u,p_n}} \right)$$

Explicitly substituting the expression for e:

$$\nabla_{\boldsymbol{w}_u} e = \frac{\partial e}{\partial \boldsymbol{w}_u} = \frac{\partial}{\partial \boldsymbol{w}_u} \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{l \in L_{\text{fixed}}} \frac{\partial e^{(l)}}{\partial \boldsymbol{w}_u} = \sum_{l \in L_{\text{fixed}}} \nabla_{\boldsymbol{w}_u} e^{(l)}$$

Consider a single training pattern l and its error $e^{(l)} = \sum_{v \in U_{\text{out}}} e_v^{(l)}$. The error depends on the value of the output of the layer, which in turn depends (also) on the network input. However, the network input depends on the weights:

$$\text{net}_u^{(l)} = \boldsymbol{w}_u \mathbf{in}_u^{(l)} = \boldsymbol{w}_u \left(1, \text{out}_{p_1}^{(l)}, ..., \text{out}_{p_n}^{(l)}\right)$$

Which means that there's a dependency between the error and the weights. Applying the chain rule:

$$\nabla_{\boldsymbol{w}_u} e^{(l)} = \frac{\partial e^{(l)}}{\partial \boldsymbol{w}_u} = \frac{\partial e^{(l)}}{\partial \text{net}_u^{(l)}} \frac{\partial \text{net}_u^{(l)}}{\partial \boldsymbol{w}_u} = \frac{\partial e^{(l)}}{\partial \text{net}_u^{(l)}} \frac{\partial \boldsymbol{w}_u \mathbf{in}_u^{(l)}}{\partial \boldsymbol{w}_u} = \frac{\partial e^{(l)}}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)}$$

Expanding the error $e^{(l)}$ in the first factor:

$$\begin{aligned}
\frac{\partial e^{(l)}}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)} &= \frac{\partial \sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \text{out}_v^{(l)}\right)^2}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)} = \sum_{v \in U_{\text{out}}} \frac{\partial \left(o_v^{(l)} - \text{out}_v^{(l)}\right)^2}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)} = \\
&= \sum_{v \in U_{\text{out}}} 2\left(o_v^{(l)} - \text{out}_v^{(l)}\right) \frac{\partial \left(o_v^{(l)} - \text{out}_v^{(l)}\right)}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)} = \\
&= 2 \sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \text{out}_v^{(l)}\right) \left(\cancel{\frac{\partial o_v^{(l)}}{\partial \text{net}_u^{(l)}}} - \frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_u^{(l)}}\right) \mathbf{in}_u^{(l)} = \\
&= -2 \underbrace{\sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \text{out}_v^{(l)}\right) \frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_u^{(l)}}}_{\delta_u^{(l)}} \mathbf{in}_u^{(l)} = -2\delta_u^{(l)} \mathbf{in}_u^{(l)}
\end{aligned}$$

Where the shorthand $\delta_u^{(l)}$ is introduced for clarity. Note that $\partial o_v^{(l)} / \partial \text{net}_u^{(l)}$ is 0 because $o_v^{(l)}$ is a known constant.

To compute $\delta_u^{(l)}$, first consider the particularly favourable case in which u is an output neuron. Since the neurons in the output layer (or in any layer, for that matter), are not connected to each other, there is no dependency between the output of one and the network input of another. This means that all terms of the sum except for $v = u$ are null, because $\partial \text{out}_v^{(l)} / \partial \text{net}_u^{(l)}$ is 0:

$$\nabla_{\boldsymbol{w}_u} e^{(l)} = \frac{\partial e^{(l)}}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)} = -2 \sum_{v=u} \left(o_v^{(l)} - \text{out}_v^{(l)}\right) \frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)} = -2\left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)}$$

This means that the weights coming into the output neuron u should be shifted by:

$$\Delta \boldsymbol{w}_u^{(l)} = -\frac{\eta}{2} \left(-2\left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)}\right) = \eta \left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)}$$

Where the minus signs cancel, because the interest is in going in the direction opposite to the gradient of the error function. The parameter η that absorbs the factor of 2 is called the **learning rate**, and represents the lenght of the step taken in one iteration of gradient descent. Popular choices for η are 0.1 and 0.2, but in general the best choice is domain specific.

The above expression was referring to the weight change that results from a single training pattern l. That is, this is how weights are adapted in online training, where the weights are adapted immediately

after each example is presented to the perceptron. In the case of batch learning the idea is the same, the difference being that one would have to sum all partial updates over all training patterns and then applying the resulting change at the end of each epoch, not immediately after each evaluation.

The derivative of $\partial \mathrm{out}_u^{(l)} / \partial \mathrm{net}_u^{(l)}$ cannot be calculated in the general case, since the output depends on the choice of the activation function. Suppose that the logistic function has been chosen, and that the output function is the identity function (as it is the case, in general). Then:

$$\frac{\partial \mathrm{out}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}} = \frac{\partial f_{\mathrm{out}}\left(\mathrm{act}_u^{(l)}\right)}{\partial \mathrm{net}_u^{(l)}} = \frac{\partial \mathrm{act}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}} = \frac{\partial f_{\mathrm{act}}\left(\mathrm{net}_u^{(l)}\right)}{\partial \mathrm{net}_u^{(l)}} = f'_{\mathrm{act}}\left(\mathrm{net}_u^{(l)}\right) =$$
$$= f_{\mathrm{act}}\left(\mathrm{net}_u^{(l)}\right)\left(1 - f_{\mathrm{act}}\left(\mathrm{net}_u^{(l)}\right)\right) = \mathrm{act}_u^{(l)}\left(1 - \mathrm{act}_u^{(l)}\right) = \mathrm{out}_u^{(l)}\left(1 - \mathrm{out}_u^{(l)}\right)$$

Which means that the adaptation should be:

$$\Delta \boldsymbol{w}_u^{(l)} = \eta\left(o_u^{(l)} - \mathrm{out}_u^{(l)}\right)\frac{\partial \mathrm{out}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}}\mathbf{in}_u^{(l)} = \eta\left(o_u^{(l)} - \mathrm{out}_u^{(l)}\right)\mathrm{out}_u^{(l)}\left(1 - \mathrm{out}_u^{(l)}\right)\mathbf{in}_u^{(l)}$$

Since the neuron under consideration was one belonging to the output layer, this adaptation is the one that ought to be performed on the weights coming into the output layer (or going out of the last hidden layer, that is). For a 2-layer perceptron, this would already be sufficient. However, the fact that the activation function is differentiable allows the same approach to be extended to all other weights, something that a network of TLUs couldn't.

Consider a neuron $u \in U$ than belongs to a hidden layer. That is $u \in U_k$ with $0 < k < r - 1$. In this case, the factor $\partial \mathrm{out}_v^{(l)} / \partial \mathrm{net}_u^{(l)}$ that appears in the expression for $\delta_u^{(l)}$ is not 0, because there is a dependency between the two functions. In particular, $\mathrm{out}_v^{(l)}$ depends on $\mathrm{net}_u^{(l)}$ indirectly via the successors of u, since they have $f_{\mathrm{out}}^{(u)}\left(f_{\mathrm{act}}^{(u)}\left(\mathrm{net}_u^{(l)}\right)\right)$ as one of their inputs.

Let $\mathrm{succ}(u) = \{s \in U \mid (u, s) \in C\} = \{s_1, ..., s_m\} \subseteq U_{k+1}$, and let $\mathrm{net}_s^{(l)}$ the network input (in the training pattern l) of the successor s of u. Since $\mathrm{net}_s^{(l)}$ depends (also) on $f_{\mathrm{out}}^{(u)}\left(f_{\mathrm{act}}^{(u)}\left(\mathrm{net}_u^{(l)}\right)\right)$, applying the chain rule to $\partial \mathrm{out}_v^{(l)} / \partial \mathrm{net}_u^{(l)}$ gives:

$$\delta_u^{(l)} = \sum_{v \in U_{\mathrm{out}}}\left(o_v^{(l)} - \mathrm{out}_v^{(l)}\right)\frac{\partial \mathrm{out}_v^{(l)}}{\partial \mathrm{net}_u^{(l)}} = \sum_{v \in U_{\mathrm{out}}}\left(o_v^{(l)} - \mathrm{out}_v^{(l)}\right)\sum_{s \in \mathrm{succ}(u)}\frac{\partial \mathrm{out}_v^{(l)}}{\partial \mathrm{net}_s^{(l)}}\frac{\partial \mathrm{net}_s^{(l)}}{\partial \mathrm{net}_u^{(l)}}$$

Since both sums are finite, they can be rearranged:

$$\delta_u^{(l)} = \sum_{v \in U_{\mathrm{out}}}\left(o_v^{(l)} - \mathrm{out}_v^{(l)}\right)\sum_{s \in \mathrm{succ}(u)}\frac{\partial \mathrm{out}_v^{(l)}}{\partial \mathrm{net}_s^{(l)}}\frac{\partial \mathrm{net}_s^{(l)}}{\partial \mathrm{net}_u^{(l)}} =$$
$$= \sum_{v \in U_{\mathrm{out}}}\sum_{s \in \mathrm{succ}(u)}\left(o_v^{(l)} - \mathrm{out}_v^{(l)}\right)\frac{\partial \mathrm{out}_v^{(l)}}{\partial \mathrm{net}_s^{(l)}}\frac{\partial \mathrm{net}_s^{(l)}}{\partial \mathrm{net}_u^{(l)}} =$$
$$= \sum_{s \in \mathrm{succ}(u)}\underbrace{\left(\sum_{v \in U_{\mathrm{out}}}\left(o_v^{(l)} - \mathrm{out}_v^{(l)}\right)\frac{\partial \mathrm{out}_v^{(l)}}{\partial \mathrm{net}_s^{(l)}}\right)}_{\delta_s^{(l)}}\frac{\partial \mathrm{net}_s^{(l)}}{\partial \mathrm{net}_u^{(l)}} = \sum_{s \in \mathrm{succ}(u)}\delta_s^{(l)}\frac{\partial \mathrm{net}_s^{(l)}}{\partial \mathrm{net}_u^{(l)}}$$

The network input $\mathrm{net}_s^{(l)}$ of the neuron s (for the training pattern l) can be written explicitly:

$$\frac{\partial \mathrm{net}_s^{(l)}}{\partial \mathrm{net}_u^{(l)}} = \frac{\partial}{\partial \mathrm{net}_u^{(l)}}\left(\boldsymbol{w}_s \mathbf{in}_s^{(l)}\right) = \frac{\partial}{\partial \mathrm{net}_u^{(l)}}\left(\left(\sum_{p \in \mathrm{pred}(s)} w_{s,p}\mathrm{out}_p^{(l)}\right) - \theta_s\right)$$

Out of all neurons $p \in \text{pred}(s)$ there is precisely the neuron u, since s is one of its successors. This means that the only term of $\partial \text{net}_s^{(l)} / \partial \text{net}_u^{(l)}$ that is not null is the one where $p = u$, because it's the only dependency between $\text{net}_s^{(l)}$ and $\text{net}_u^{(l)}$:

$$\frac{\partial \text{net}_s^{(l)}}{\partial \text{net}_u^{(l)}} = \frac{\partial}{\partial \text{net}_u^{(l)}} \left(\left(\sum_{p \in \text{ pred}(s)} w_{s,p} \text{out}_p^{(l)} \right) - \theta_s \right) = \frac{\partial}{\partial \text{net}_u^{(l)}} \left(\left(\sum_{p=u} w_{s,p} \text{out}_p^{(l)} \right) - \theta_s \right) =$$

$$= \frac{\partial \left(\left(w_{s,u} \text{out}_u^{(l)} \right) - \theta_s \right)}{\partial \text{net}_u^{(l)}} = \frac{\partial w_{s,u} \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} - \cancel{\frac{\partial \theta_s}{\partial \text{net}_u^{(l)}}} = w_{s,u} \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}}$$

Substituting in the previous expression:

$$\delta_u^{(l)} = \sum_{s \in \text{ succ}(u)} \delta_s^{(l)} \frac{\partial \text{net}_s^{(l)}}{\partial \text{net}_u^{(l)}} = \left(\sum_{s \in \text{ succ}(u)} \delta_s^{(l)} w_{s,u} \right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}}$$

Which gives the expression for $\nabla_{\boldsymbol{w}_u} e^{(l)}$ when the neuron u is an hidden layer:

$$\nabla_{\boldsymbol{w}_u} e^{(l)} = -2 \delta_u^{(l)} \mathbf{in}_u^{(l)} = -2 \left(\sum_{s \in \text{ succ}(u)} \delta_s^{(l)} w_{s,u} \right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)}$$

Confronting this expression with the gradient for an output neuron, the term $\sum_{s \in \text{ succ}(u)} \delta_s^{(l)} w_{s,u}$ plays the role of $o_u^{(l)} - \text{out}_u^{(l)}$, and indeed the expression for $\delta_s^{(l)}$ does contain $o_u^{(l)} - \text{out}_u^{(l)}$.

This is because $\delta_s^{(l)}$ is the error of the entire output layer multiplied by the partial derivative of the network input of s (a successor of u) with respect to the network input of u. In simpler terms, the error of a hidden neuron depends on the error of its successors, and can be computed from theirs. Even better, it could be stated that the error of the output neurons "ripples back" to the hidden neurons, layer by layer. This is why this method of weight adaptation is called **backpropagation**.

This gives a formula for the weight adaptation of hidden neurons:

$$\Delta \boldsymbol{w}_u^{(l)} = -\frac{\eta}{2} \nabla_{\boldsymbol{w}_u} e^{(l)} = \eta \left(\sum_{s \in \text{ succ}(u)} \delta_s^{(l)} w_{s,u} \right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)}$$

Assuming, again, that the activation function of choice is the logistic function and that the output function is the identity, the explicit expression is:

$$\Delta \boldsymbol{w}_u^{(l)} = \eta \left(\sum_{s \in \text{ succ}(u)} \delta_s^{(l)} w_{s,u} \right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)} = \eta \left(\sum_{s \in \text{ succ}(u)} \delta_s^{(l)} w_{s,u} \right) \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)} \right) \mathbf{in}_u^{(l)}$$

The backpropagation algorithm allows one to train a multilayer perceptron with training examples, but is oblivious to the "shape" of the network. That is, the number of hidden layers and the number of hidden neurons have to be fixed "by hand" before the training process. For a 3-layer perceptron, a known rule of thumb for determining how many hidden neurons there should be is:

$$\frac{\text{number of input neurons} + \text{number of output neurons}}{2}$$

The simplest approach to find the most suitable number of hidden neurons is to try out different configurations and see what works best. The dataset of examples is split into two (roughly) equally sized subsets: the *training data* and the *validation data*. Multilayer perceptrons with different number of neurons in the hidden layer are trained on the training data and evaluated on the validation data (on which the perceptrons are *not* trained on). The process is repeated with different splits, averaging

the results per number of hidden neurons. The perceptron with the number of hidden neuron that performs best on the validation data is chosen, and is then re-trained with the entire dataset. This approach is referred to as **cross-validation**.

If the number of hidden neurons is insufficient, the accuracy with which the perceptron maps inputs to output will also be insufficient. This happens because the error function cannot be lowered than an acceptable threshold. This is also referred to as **underfitting**.

Since increasing the number of neurons also increases the accuracy of the predictions, it might be tempting to naively construct multilayer perceptrons with as many neurons as possible. Even if this were to be computationally affordable (more neurons also means slower training, because more weights have to be updated), it's not advisable. This is because a perceptron that is "too focused" on a dataset might learn not only its input-output relationship, but also the inherent biases and errors that are present. These errors are unavoidable, because all data is sampled from larger populations and don't represent it as a whole, and all measurements have a finite precision.

When a multilayer perceptron is "too narrowed" on a dataset, what happens is that trying predicting inputs that are ever so slightly different from the training patterns result in considerable inaccuracy. This situation is also referred to as **overfitting**. Overfitting happens because different data have different inherent bias, and a multilayer perceptron that is too focused on a single dataset will be prone to incorporate also its bias.

Cross validation can effectively tame both overfitting and underfitting because the validation data and the training data will likely distorted in a different fashion, since the errors and deviations are random. Both underfitting and overfitting are unfavorable, but in general the former is worse: it is better to have a model with poor tolerance than a model with poor performance.

An extension of cross-validation is **n-fold cross-validation**. The technique consists in partitioning the dataset in n subsets of equal size, called **folds**. For n times, one of the n subsets is chosen to be the validation data and the remaining $n - 1$ subsets are merged into a single set, that will be the training data. Common choices of n are 5 (*5-fold cross-validation*) and 10 (*10-fold cross-validation*).

Another approach to prevent overfitting is what's called **early stopping**: during the training the accuracy of the multilayer perceptron is evaluated after each epoch (or every few epochs) on a validation data set. While the error on the training data set should always decrease with each epoch, the error on the validation data set should initially decrease and then increase again as soon as overfitting sets in. When this happens, training is terminated (hence the name early stopping) and the current choice of parameters for the perceptron are frozen.

2.4.4. Deep learning

Even though Theorem 2.4.2.1 proves that any Riemann-integrable function can be encoded into a multilayer perceptron. This shows how powerful such objects are in theory, but doesn't give much information on the practical side. Even though it's true that a 4-layer perceptron with an enormous amount of hidden neurons can encode any function with little to no error, this doesn't mean that it's actually feasible to implement a perceptron having that many hidden neurons.

Theorem 2.4.2.1 is an existence theorem, not an existence and uniqueness theorem: the fact that a function can be approximated with a 4-layer perceptron does not imply that said construction is the only possible. Moreover, it doesn't even imply that it's the "best" construction, meaning the one that requires the smallest number of neurons for reaching the same approximation error.

Technically speaking, it is possible to improve the approximation of a multilayer perceptron without increasing the number of hidden neurons. For example, an activation function that is not the Heaviside function might better model the shape of the function to encode. A complementary approach would

be to use step widths that aren't uniform, using smaller steps where the function is heavily skewed (thus a linear approximation is poor) and larger steps where it is almost linear.

Incorporating these improvements into a 4-layer perceptron can ameliorate the problem of the growing number of hidden neurons. It would then seem as the 4-layer model is always enough. However, it can be shown that a much greater approximating power can be obtained by abandoning the 4-layer model completely, constructing perceptrons that have 5 or even more layers.

An often cited example is in encoding the n-bit even parity function, the binary function defined as:

$$p_n(x_1, ..., x_n) = \begin{cases} 1 \text{ if an even number of inputs is } 1 \\ 0 \text{ if an odd number of inputs is } 1 \end{cases}$$

It is possible to construct a network of TLUs (which is just a particular declination of a multilayer perceptron) that can encode this function using the presented algorithm. However, the number of hidden neurons needed would be 2^{n-1}, because $p(n)$ written in DNF is a disjunction of 2^{n-1} disjunctions. This means that the total number of neurons would be $2^{n-1} + n + 1$.

Exercise 2.4.4.1: What would be the DNF for the n-bit parity function with $n = 3$?

Solution:

$$(x_1 \wedge x_2 \wedge (\neg x_3)) \vee (x_1 \wedge (\neg x_2) \wedge x_3) \vee ((\neg x_1) \wedge x_2 \wedge x_3) \vee ((\neg x_1) \wedge (\neg x_2) \wedge (\neg x_3))$$

□

There is, however, a cleverer approach to encoding the same function. The idea is to chain together $n - 1$ simple networks for computing the biimplication and the exclusive or. The first of these networks combines two inputs with a biimplication network. Each of the remaining $n - 2$ networks applies the exclusive or to the output of the preceding network and to another input. Since both the biimplication and the exclusive or require a total of three neurons, the number of hidden neurons is $3(n - 1) - 1$. This means that the total number of neurons would be $n + 3(n - 1) - 1 + 1 = 4n - 3$.

Exercise 2.4.4.2: What would be a multilayer perceptron that encodes the n-bit even parity function with $n = 3$?

Solution: The number of neurons needed is $4 \cdot 3 - 3 = 9$.

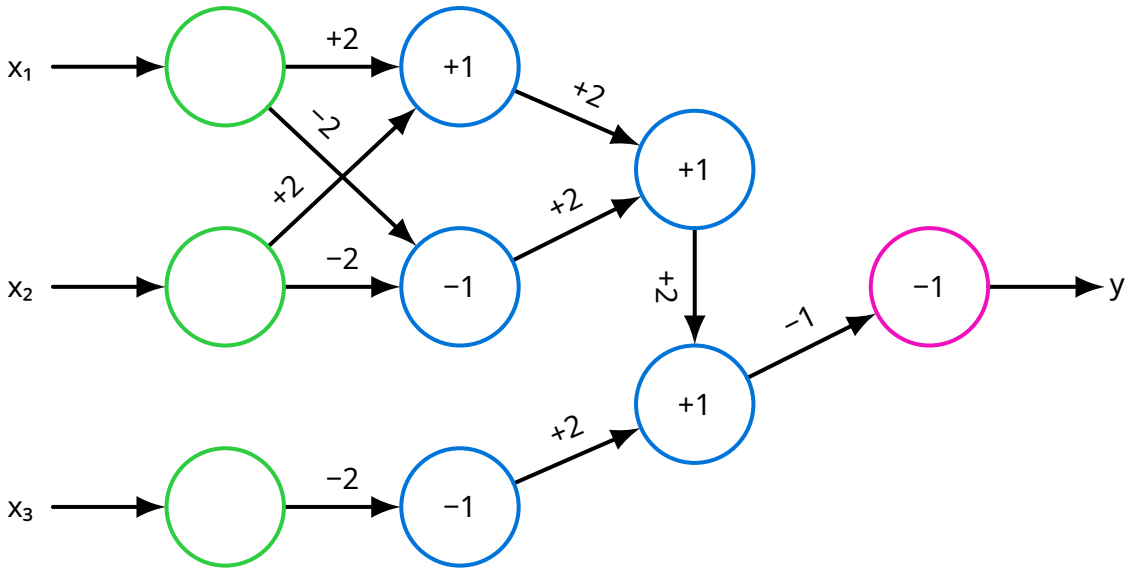


Figure 27: A multilayer perceptron that encodes the 3-bit even parity function.

□

In the first case, the multilayer perceptron always has 3 layers (one input, one hidden, one output) but the total number of neurons scales exponentially with the number of inputs. In the second case, both the number of layers and the total number of neurons scales linearly. For large values of n, the advantage is noticeable: with $n = 10$, a 3-layer perceptron would need $2^{10-1} + 10 + 1 = 523$ neurons, while a multilayer perceptron build with the second approach would need $4 \cdot 10 - 3 = 37$ neurons.

Multilayer perceptrons with many hidden layers are one of the interests of **deep learning**, the branch of neural network theory that studies neural networks that are *deep*. The "depth" of a neural network is the length of the longest path in the graph underlying the network. In the case of multilayer perceptrons, a feed-forward neural network where elements of the same layer cannot be connected, the depth is the number of hidden layers plus one (the output layer).

The first concern of deep learning is preventing overfitting. Overfitting is mainly caused from the fact that, having many hidden layers and hence many parameters, the network is much more vulnerable to picking up biases from the training examples. Overfitting can be addressed with **weight decay**, which prevents overly large weights and thus an overly precise adaptation to (accidental) properties of the data. Another solution is introducing sparsity constraints, either in the form of a restricted number of neurons in the hidden layers or by requiring that only few of the neurons in the hidden layers should be active (on average). The latter is usually achieved by adding a regularization term to the error function, which compares the observed number of activated neurons with the desired low number and pushes the adaptations into a direction that tries to match these numbers. A simple yet effective method for taming overfitting is the so-called **dropout training**, which consists in disabling some hidden neurons at random during each forward and backward propagation iteration.

Another concern of deep learning is preventing **vanishing gradients**. The phenomena of vanishing gradients occurr when the first hidden layers (the ones closer to the input layer) receive an adaptation from the backpropagation that is much smaller in magnitude than the adaptation received by the last hidden layers (the ones closer to the output layer), and hence remain mostly random.

This is a known problem especially when the employed activation function is the logistic function, since its derivative is bounded by $1/4$. Vanishing gradient happens because the logistic function is a *contracting* function, meaning that for any x, y, $|x - y| > |f_{\text{act}}(x) - f_{\text{act}}(y)|$. In words, the distance between the arguments of the logistic function is always bigger than the distance between the images. Since its derivative is itself times one minus itself, every time a new derivative is added to the chain of

multiplications, the maximum value that is can attain becomes smaller and smaller. This is precisely what happens in the latter stages of backpropagation, where the chain of multiplications is longer.

A drastic approach to the vanishing gradient problem is to use an activation function that does not suffer from the contraction problem as much. These functions aren't necessarely sigmoids, but this is not an issue as long as they are differentiable everywhere. One class of such functions is the **rectified linear units** (**ReLU** s), whose most notable representative is the **rectified maximum**:

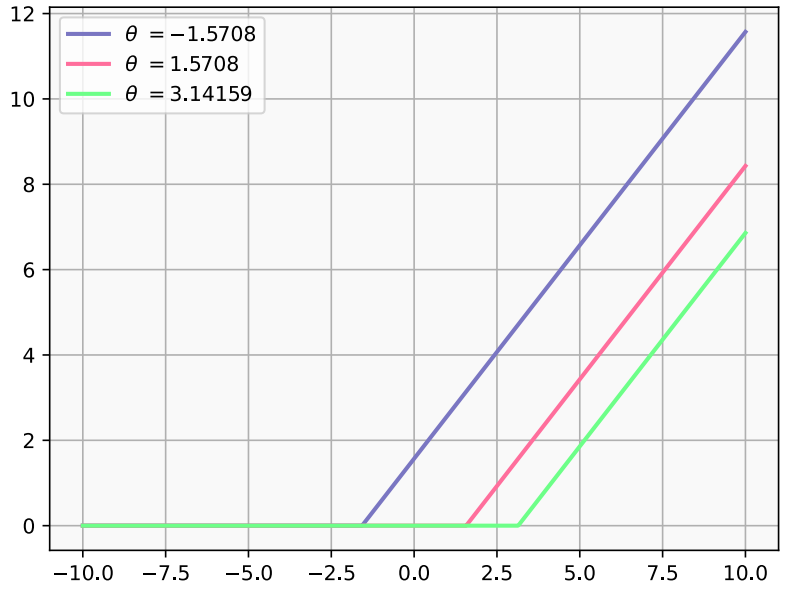


Figure 28: Plot of the rectified maximum, with three different choices of θ

Another ReLU that is often employed is the **softplus**:

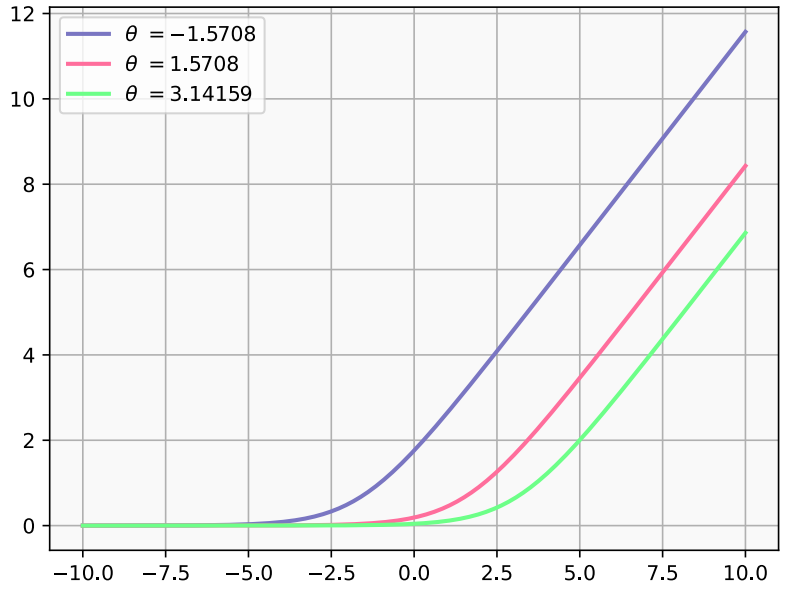


Figure 29: Plot of the softplus, with three different choices of θ

For many choices of inputs, the derivative of these functions are actually relatively close to 1, meaning that it's much more likely to "ripple back" a sizeable gradient. These can cause the opposite problem, called **exploding gradient**, meaning that the updates of the first hidden layers are much bigger in magnitude than the updates of the last hidden layers.

A completely different approach consists in building the multilayer perceptron one layer at a time, training only the newly added layer in each step. A very popular specific procedure for this is to build the network as **stacked autoencoders**.

An autoencoder is a 3-layer perceptron that maps its inputs to an approximation of those same inputs. The hidden layers *encodes* the data coming the input into a form internal representation, which is then *decoded* by the output layer. Autoencoders can be trained with standard backpropagation, since there is only one hidden layer and therefore the vanishing gradient problem does not arise.

The rationale of training an autoencoder is that the hidden layer is expected to construct features (that capture the information contained in the input data in a compressed form, so that the input can be well reconstructed from it. As a matter of fact, an autoencoder is analogous to applying a dimensionality reduction technique, having "raw" data as input and extracting the most relevant features from said input.

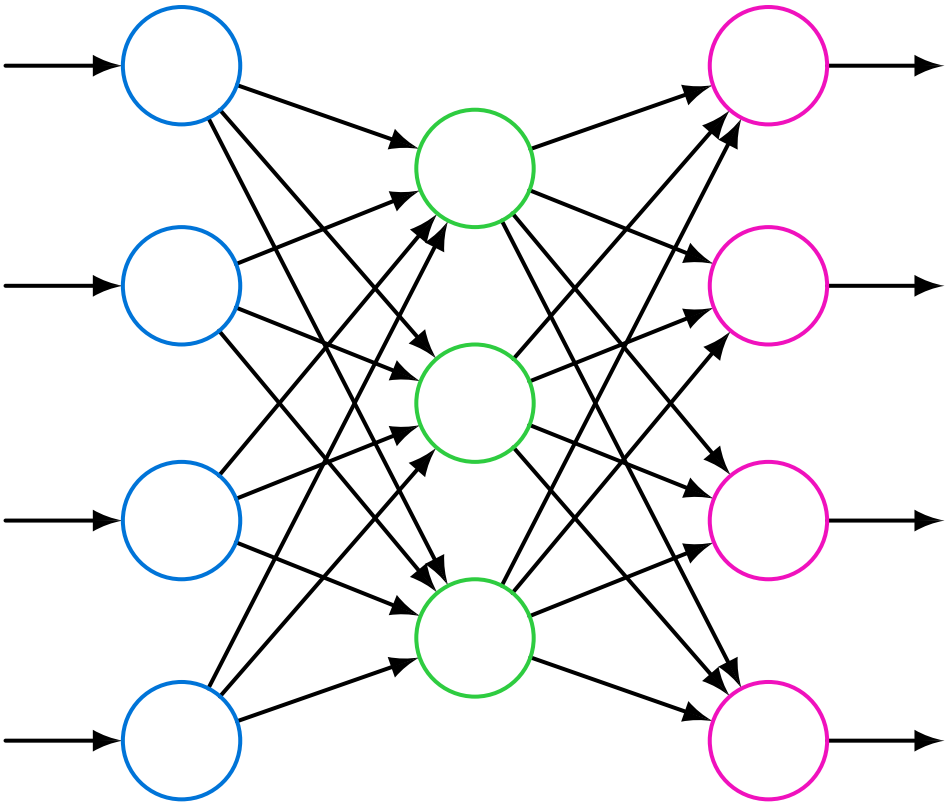


Figure 30: Structure of a generic autoencoder. The output is a reconstructed version of the input.

The number of neurons in the hidden layer of an autoencoder is a crucial choice for its effectiveness. If the number of neurons is equal or larger than the input neurons, it is much more likely that the reconstructed input won't be that different from the original input, because the entirety of the contained information is kept.

The first approach is simply to choose less hidden neurons than input neurons, in order to purposefuly loose part of the information and force the network to learn only what's relevant. This shifts the question to how many hidden neurons should be chosen. Using cross-validation to find the optimal number of neurons would be misleading, because the number of neurons that gives the smallest error is most likely the one that matches the number of inputs: this is precisely not what is desired, because a certain discrepancy between the encoded data and the decoded data is actually desireable.

A second commonly used approach is a sparse activation scheme, as it was done to avoid overfitting. The number of hidden neurons with a high activation is restricted to a small number, which is enforced

by either adding a regularization term to the error function that punishes a larger number of active hidden neurons or by explicitly deactivating all but a few neurons with the highest activations.

A third approach is to purposefully add noise to the input (only to the input of the autoencoder, not to the input used to evaluate its encoding and decoding at the end of the process). These are called **denoising autoencoders**, because the autoencoder to be trained is expected to map the input with noise to (a copy of) the input without noise, "cleaning" the data and only keeping what's needed. Noise can also help prevent overfitting, especially if the added noise is different from training epoch to training epoch, so that the network cannot become acquainted to a specific noise.

A multilayer perceptron made of stacked autoencoders is constructed as follows. First, an autoencoder is built and trained with standard backpropagation on the given training data (the one intended for the perceptron as a whole). The output layer is discarded and only the hidden layer is kept, together with its incoming weights. A new dataset is produced by feeding the input data through this layer and using the activation of the neurons as a the new data.

The process is then repeated, training a new autoencoder on this data, keeping only the hidden layer and the incoming weights, producing new data from its activation, ecc... at each step, the goal is for the autoencoders to extract more and more relevant and high-level features. Training these single autoencoders in isolation circumvents the vanishing gradient problem because they have only 3 layers.

Finally, all hidden layers and incoming weights constructed so far are chained together ("stacked", hence the name) in the same order as they were created. The final touch is to add an output layer and train the entire network on the original data: in this final training run the vanishing gradient problem is less severe, because only the output layer needs to be tuned. The earlier hidden layers have been trained already, and even if the gradient vanishes this is not a much of an issue because their optimal parameters have most likely been found already.

2.4.5. Sensitivity analysis

The knowledge that an artificial neural networks captures in the training process is difficult to understand. This is because it is effectively stored as matrices, that on their own don't have a particular semantic. Moreover, when the number of dimensions grows, it becomes impossible to give the neural network a geometrical interpretation.

To some extent, it is by design that a neural network is hard to inspect. After all, neural networks are often "fire and forget" computational devices, that perform complex computations without the need to carry it out by hand or to understand how it was done. However, it can sometimes be beneficial to get a glimpse inside the operations of a neural network.

Sensitivity analysis is a technique used to evaluate the influence that each input has on the output of a neural network. That is, how much the output changes if one of the inputs were to be changed by the same order of magnitude. For each input neuron u, it's possible to assign a sensitivity $s(u)$ as sum, over all training patterns, of the partial derivative of the output of each output neuron with respect to the external input of the input neuron u, divided by the size of the patterns. That is:

$$s(u) = \frac{1}{|L_{\text{fixed}}|} \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} \frac{\partial \text{out}_v^{(l)}}{\partial \text{ext}_u^{(l)}}$$

The rationale behind this quantity is that, since a derivative is by definition a rate of change, computing the sum of the partial derivatives of the outputs with respect to the inputs one has the rate of change between the latter and the former. Once this quantity is known, it is for example possible to

simplify the network by removing the inputs with the least influence, reducing the dimensionality of the problem.

To obtain an explicit expression for $s(u)$, it is first of all necessary to describe the dependency between out_v and ext_u. Clearly, since u comes before v, the output of v does depend to some extent to the output of u. Moreover, the output of v depends on the network input of v. This gives:

$$\frac{\partial \mathrm{out}_v}{\partial \mathrm{ext}_u} = \frac{\partial \mathrm{out}_v}{\partial \mathrm{out}_u}\frac{\partial \mathrm{out}_u}{\partial \mathrm{ext}_u} = \frac{\partial \mathrm{out}_v}{\partial \mathrm{net}_v}\frac{\partial \mathrm{net}_v}{\partial \mathrm{out}_u}\frac{\partial \mathrm{out}_u}{\partial \mathrm{ext}_u}$$

Since input neurons transmit their external input unchanged, the term $\partial \mathrm{out}_u / \partial \mathrm{ext}_u$ is just 1. As for the second term:

$$\frac{\partial \mathrm{net}_v}{\partial \mathrm{out}_u} = \frac{\partial}{\partial \mathrm{out}_u} \sum_{p \in \mathrm{pred}(v)} w_{v,p}\mathrm{out}_p = \sum_{p \in \mathrm{pred}(v)} w_{v,p}\frac{\partial \mathrm{out}_p}{\partial \mathrm{out}_u}$$

In the favourable case of v being a neuron in the first hidden layer, the formula can be simplified noticeably, since there is no dependency on the output of previous neurons, just on the weight that connects u to v:

$$\frac{\partial \mathrm{net}_v}{\partial \mathrm{out}_u} = \frac{\partial \mathrm{out}_v}{\partial \mathrm{net}_v}\sum_{p \in \mathrm{pred}(v)} w_{v,p}\frac{\partial \mathrm{out}_p}{\partial \mathrm{out}_u} = \frac{\partial \mathrm{out}_v}{\partial \mathrm{net}_v}\sum_{p=u} w_{v,u}\frac{\partial \mathrm{out}_u}{\partial \mathrm{out}_u} = \frac{\partial \mathrm{out}_v}{\partial \mathrm{net}_v}w_{v,u}$$

This serves as a starting point for the recursion. Substituting the general result in the previous expression:

$$\frac{\partial \mathrm{out}_v}{\partial \mathrm{ext}_u} = \frac{\partial \mathrm{out}_v}{\partial \mathrm{net}_v}\frac{\partial \mathrm{net}_v}{\partial \mathrm{out}_u}\cancel{\frac{\partial \mathrm{out}_u}{\partial \mathrm{ext}_u}} = \frac{\partial \mathrm{out}_v}{\partial \mathrm{net}_v}\sum_{p \in \mathrm{pred}(v)} w_{v,p}\frac{\partial \mathrm{out}_p}{\partial \mathrm{out}_u}$$

Assuming to use the logistic function as activation function, $\partial \mathrm{out}_v / \partial \mathrm{net}_v$ is just $\mathrm{out}_v(1 - \mathrm{out}_v)$. This gives:

$$\frac{\partial \mathrm{net}_v}{\partial \mathrm{out}_u} = \mathrm{out}_v(1 - \mathrm{out}_v)w_{v,u} \qquad \frac{\partial \mathrm{out}_v}{\partial \mathrm{ext}_u} = \mathrm{out}_v(1 - \mathrm{out}_v)\sum_{p \in \mathrm{pred}(v)} w_{v,p}\frac{\partial \mathrm{out}_p}{\partial \mathrm{out}_u}$$

Giving the recursive formula:

$$s(u) = \begin{cases} \dfrac{1}{|L_{\mathrm{fixed}}|}\displaystyle\sum_{l \in L_{\mathrm{fixed}}}\sum_{v \in U_{\mathrm{out}}} \mathrm{out}_v(1 - \mathrm{out}_v)w_{v,u} \\ \dfrac{1}{|L_{\mathrm{fixed}}|}\displaystyle\sum_{l \in L_{\mathrm{fixed}}}\sum_{v \in U_{\mathrm{out}}} \mathrm{out}_v(1 - \mathrm{out}_v)\sum_{p \in \mathrm{pred}(v)} w_{v,p}\frac{\partial \mathrm{out}_p^{(l)}}{\partial \mathrm{out}_u^{(l)}} \end{cases}$$

2.5. Radial basis function networks

2.5.1. Structure of a radial basis function network

A **radial basis function network** (**RBF** network) is a type of feed-forward neural network structured in layers, with the following characteristics:

- Input neurons cannot also be output neurons, and vice versa: $U_{\text{in}} \cap U_{\text{out}} = \emptyset$;
- It has exactly three layers, an input layer, an output layer and an hidden layer;
- The input layer and the hidden layer are fully connected: $C = (U_{\text{in}} \times U_{\text{hidden}}) \cup C', C' \subseteq (U_{\text{hidden}} \times U_{\text{out}})$;
- The network input function of hidden neurons is a **distance function**. Its arguments are the input(s) and the weight(s):

$$\forall u \in U_{\text{hidden}} \cup U_{\text{out}},\ f_{\text{net}}^{(u)}\left(w_{u_1}, ..., w_{u_n}, \text{in}_{u_1}, ..., \text{in}_{u_n}\right) = d\left(w_{u_1}, ..., w_{u_n}, \text{in}_{u_1}, ..., \text{in}_{u_n}\right)$$

- The activation function of hidden neurons is a **radial function**, that is a monotone nonincreasing function $f : \mathbb{R}_0^+ \mapsto [0, 1]$ with $f(0) = 1$ and $\lim_{x \to \infty} f(x) = 0$;
- The network input function of output neurons is the weighted sum of their inputs and the corresponding weights:

$$\forall u \in U_{\text{out}},\ f_{\text{net}}^{(u)}\left(w_{u_1}, ..., w_{u_n}, \text{in}_{u_1}, ..., \text{in}_{u_n}\right) = f_{\text{net}}^{(u)}(\boldsymbol{w}_u, \mathbf{in}_u) = \sum_{v \in \text{ pred}(u)} w_{u,v} \cdot \text{out}_v$$

- The activation function of output neurons is a linear function:

$$\forall u \in U_{\text{out}},\ f_{\text{act}}^{(u)}(\text{net}_u, \theta_u) = \text{net}_u - \theta_u$$

A distance function is any function $d : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}^+$ that possesses the following properties:

1. For any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$, $d(\boldsymbol{x}, \boldsymbol{y}) = 0$ implies $\boldsymbol{x} = \boldsymbol{y}$;
2. For any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$, $d(\boldsymbol{x}, \boldsymbol{y}) = d(\boldsymbol{y}, \boldsymbol{x})$ (**symmetry**);
3. For any $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathbb{R}^n$, $d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{y}, \boldsymbol{z}) \leq d(\boldsymbol{x}, \boldsymbol{z})$ (**triangle inequality**);

One notable family of distance functions is **Minkowski family**, defined as:

$$d_k(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_{i=1}^{n} |x_i - y_i|^k\right)^{\frac{1}{k}}$$

In particular:

- With $k = 1$, one has the **Manhattan distance**, or **city block distance**;
- With $k = 2$, one has the **Euclidean distance**;
- With $k \to \infty$, one has the **maximum distance**, that is $d_\infty(\boldsymbol{x}, \boldsymbol{y}) = \max_{i=1}^{n} |x_i - y_i|$

A simple way to visualize distances is to consider a two-dimensional plane and a reference point and plot surfaces of equal distance from said point. Each surface contains all the points of the plane that have the same distance from the reference point. Distances in the Minkowski family always result in regular shapes.

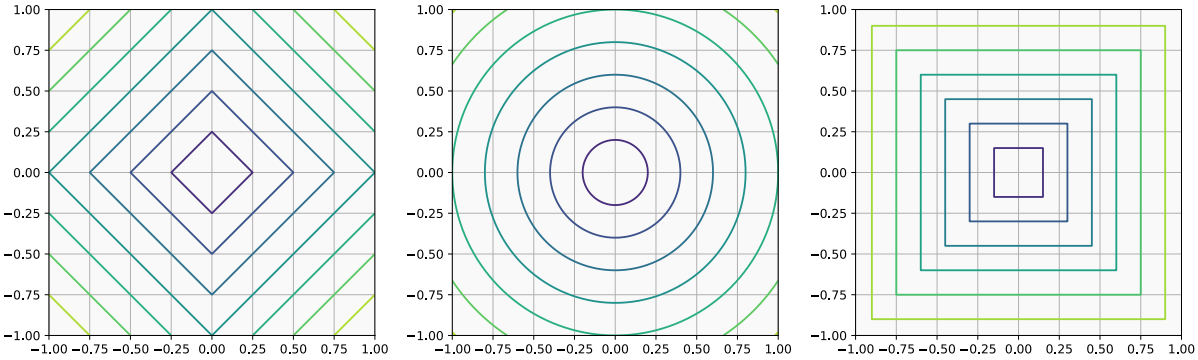


Figure 31: Surfaces of equal distance, with $(0, 0)$ being the reference point. From left to right, the Manhattan, Euclidean and maximum distance is employed. All shapes are regular: a diamond, a circle and a square respectively.

Each hidden neuron induces one of these hypersurfaces. The weights of the connections from the input layer to the hidden neuron determine the center of the region, while the σ parameter determines the radius of this surface. For this reason, the parameter σ is also called **reference radius**. Different choices of activation function give different shapes of the hypersurfaces; unless specified otherwise, it is assumed that the employed distance function is the Euclidean distance.

These surfaces form a sort of "capture region": a neuron that is "captured" by this surface, meaning that it falls inside the defined boundary, will give a non-zero activation. The closer the input is to the center of this region, the higher the activation. Indeed, the name radial functions comes from the fact that they are defined along a radius, the radius of length σ of the hypersurface centered in the incoming weights.

Each hidden neuron specifies its own capture region, and different neurons can specify potentially disjointed regions. These regions are then merged into one in the output layer, since the activation of its neurons is just the weighted sum of the activations in the hidden layer.

The simplest radial function is the **rectangular function**:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0 \text{ if net} > \sigma \\ 1 \text{ otherwise} \end{cases}$$

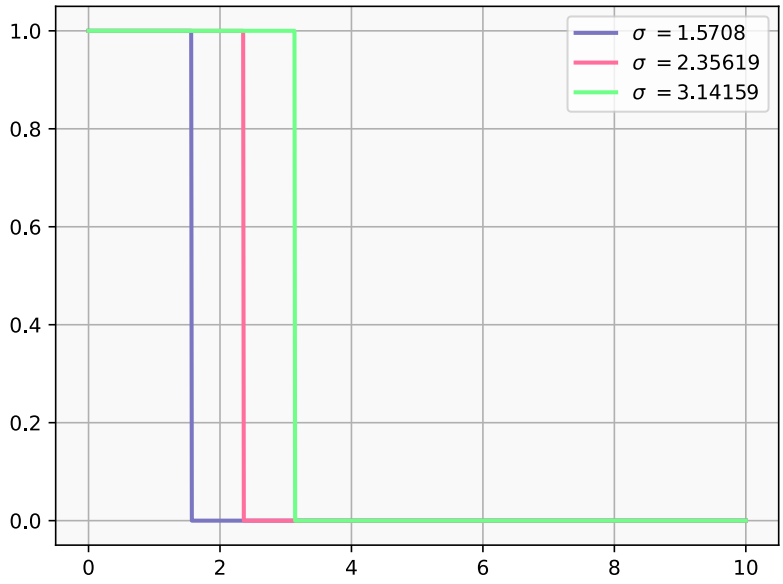


Figure 32: Plot of the rectangular function, with three different choices of σ

The activation inside the capture region is always 1 (no matter the distance from the center) and outside the capture region is always 0. Being a binary function, the rectangular function is well suited to construct radial basis function networks that encode Boolean functions.

For example, the conjunction should be encoded by a network whose circles capture the point of coordinates $(1, 1)$:

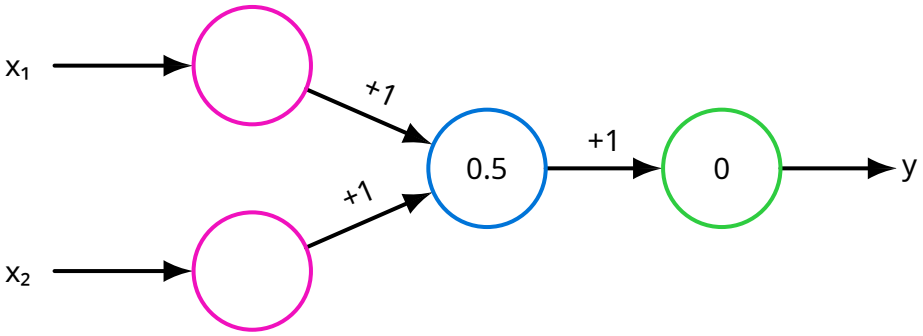


Figure 33: A radial basis function network for the conjunction.

The negation should capture 0 but not 1:

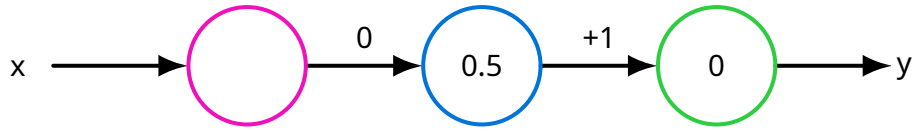


Figure 34: A radial basis function network for the negation.

If the conjunction should only capture $(1, 1)$, the disjunction should instead capture all points except for $(0, 0)$:

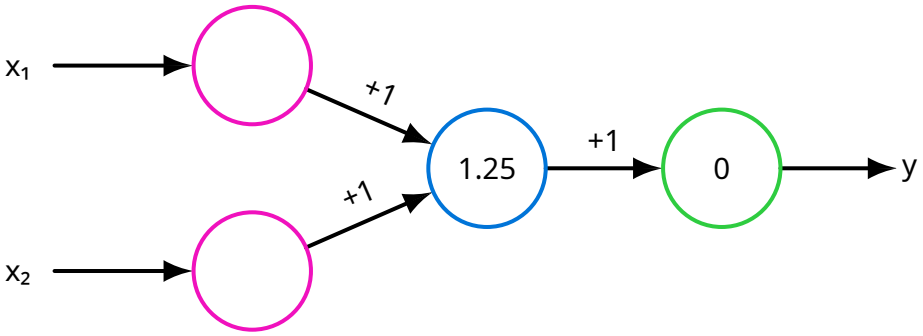


Figure 35: A radial basis function network for the disjunction.

Similarly, the implication should capture all points except for $(1, 0)$:

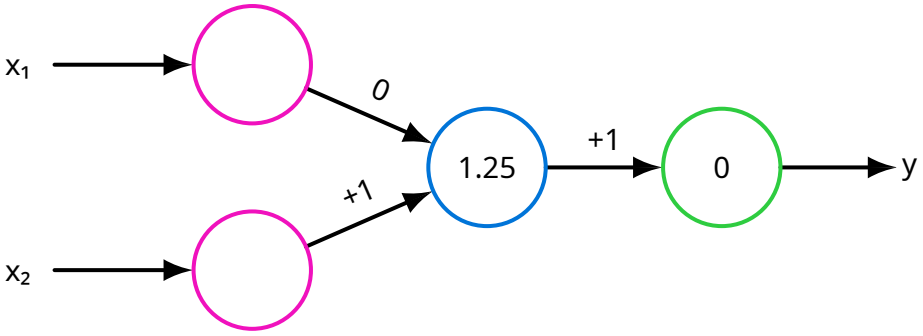


Figure 36: A radial basis function network for the implication.

The graphical representation, on its own, does not specify which activation function and network input function has been chosen for the hidden layer. It must either be stated explicitly or known from context.

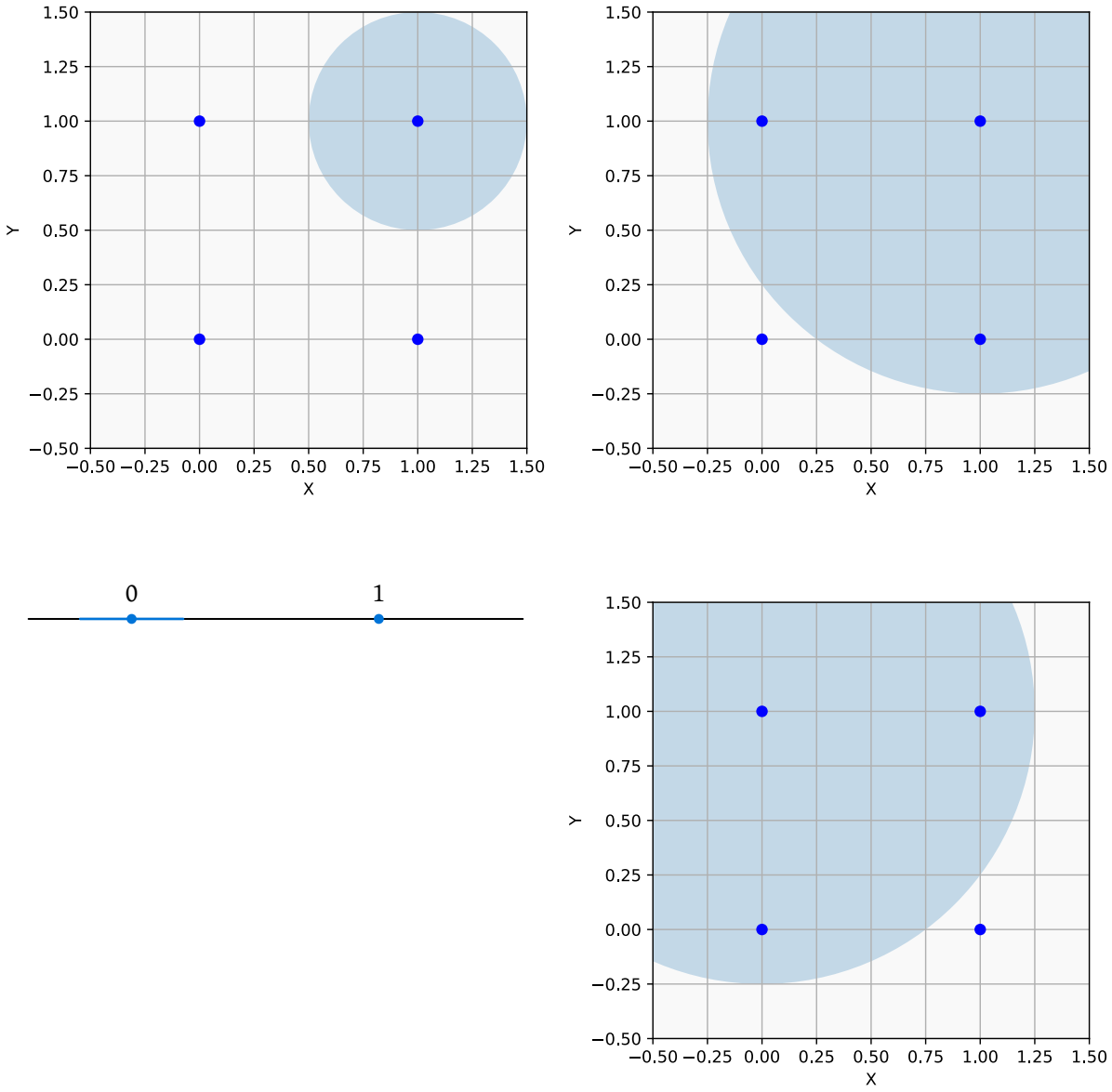


Figure 37: Going clockwise from top left: a cartesian plane partitioned by a radial basis function that computes the conjunction, disjunction, implication and negation.

Note that centering the circle precisely in the points of interest is just a matter of clarity. As long as the capture region only includes the points for which the output should be 1 any choice of coordinates for the center (that is, the weights coming into the hidden neuron) is valid. The same goes for the radius (the σ parameter of the hidden neuron); as long as the radius is small enough for the capture region to include the wrong points, but big enough for the capture region to enclose the correct points, any choice is valid.

These constructions are "additive", meaning that the area of interest is pieced together by summing smaller disjointed areas (only one, in these cases). It is also possible to work "by subtraction", that is by defining the area where the output should be 0 and then complementing it, by negating the σ parameter in the output layer.

For example, an alternative construction for the conjunction is the following:

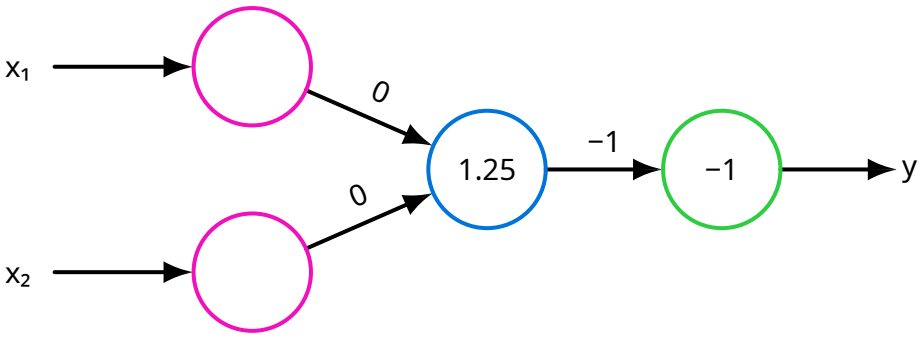


Figure 38: A different radial basis function network for the conjunction.

Where the circle envelopes the area that should *not* result in a 1; the output layer negates the input, effectively complementing the area.

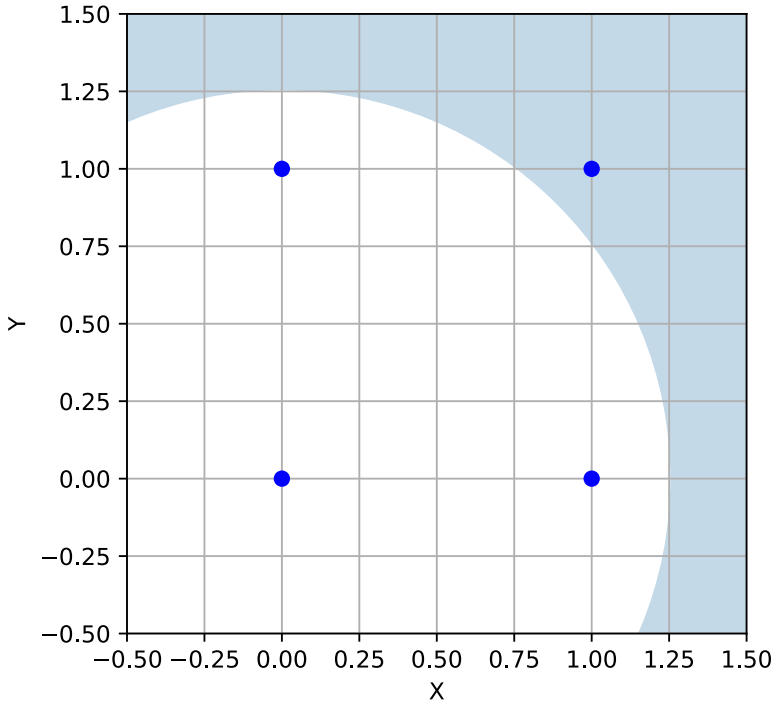


Figure 39: Geometrical representation of the computations the radial basis function network for the conjunction (new version).

More complex functions are constructed from these simple building blocks. For example, the biimplication is as follows:

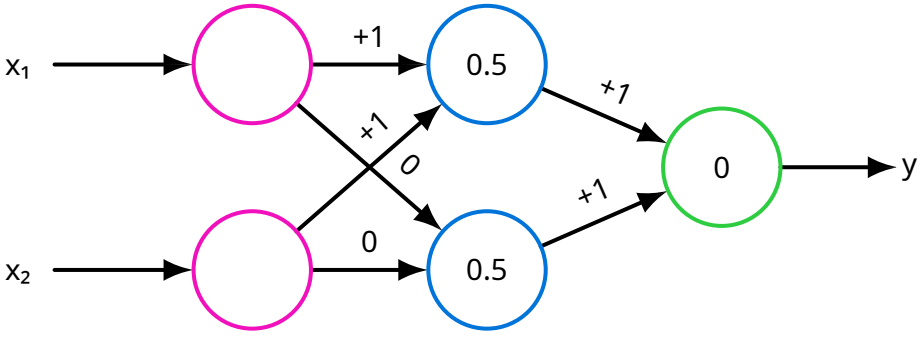


Figure 40: A radial basis function network for the biimplication.

An RBF network that encodes the exclusive or can be built directly or, since the exclusive or and the biimplication are the opposite of each other, simply by complementing the previous network:

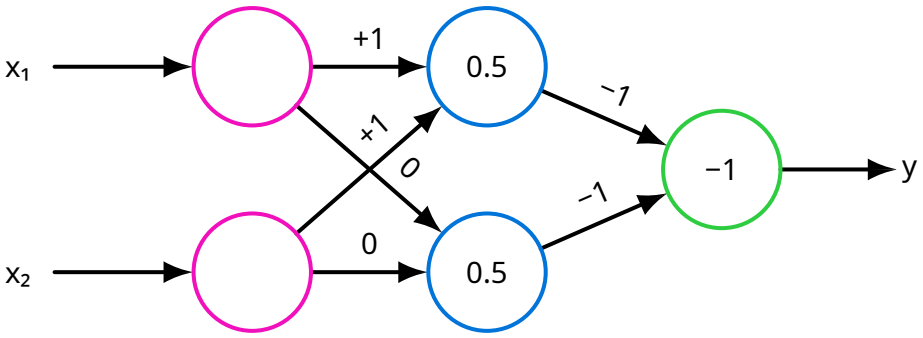


Figure 41: A radial basis function network for the exclusive or.

Note how the issue of separability is here present in a different form: there is no regular single shape that can enclose both $(0, 0)$ and $(1, 1)$, the only way is to sum two regular shapes. This is why the number of hidden neurons has to be two[10].

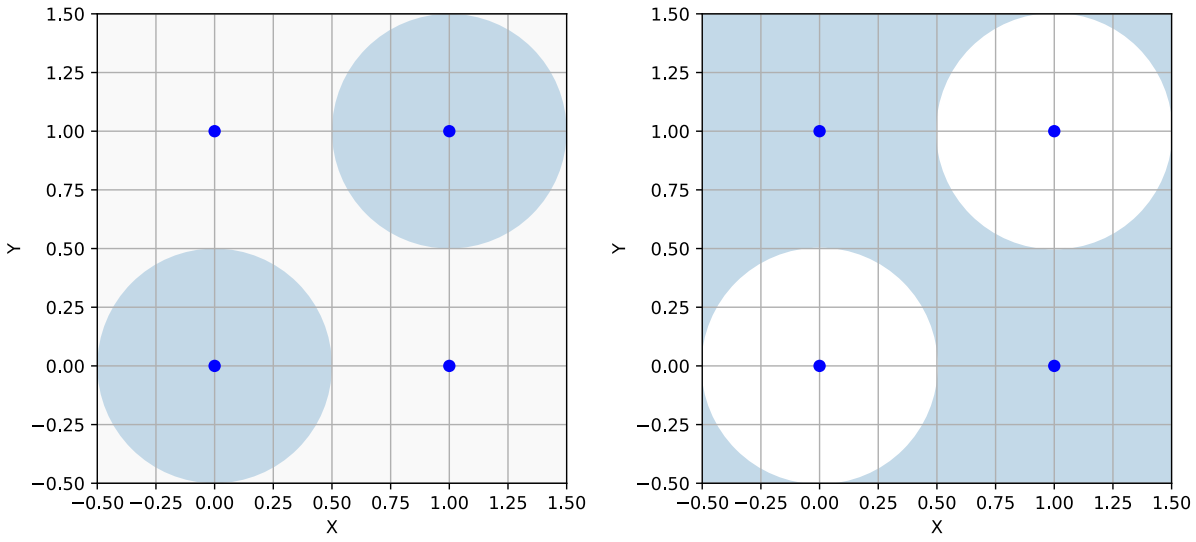


Figure 42: Geometrical representation of the computations the radial basis function network for the biimplication (left) and the exclusive or (right).

Different activation functions have their output scale with the input. An example is the **triangular function**:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0 & \text{if net} > \sigma \\ 1 - \dfrac{\text{net}}{\sigma} & \text{otherwise} \end{cases}$$

[10]The fact that the capture region is always a regular shape is a byproduct of using a distance function of the Minkowski family. There are distance functions that do not belong to this family, such as the **Mahalanobis distance**, that can create shapes that aren't regular. It is therefore possible to encode a biimplication into an RBF network with a single hidden neuron by employing distance functions such as these.

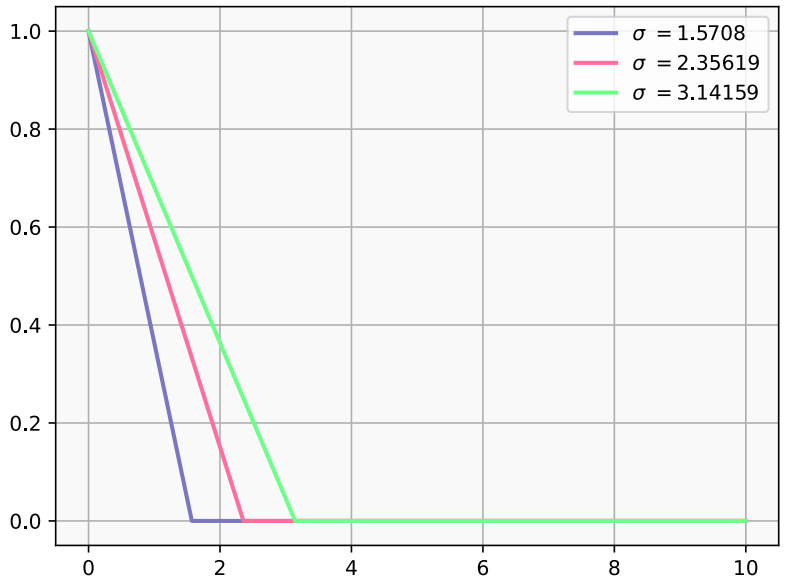


Figure 43: Plot of the triangular function, with three different choices of σ

A smoother alternative is the **cosine down to zero**:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0 & \text{if net} > 2\sigma \\ \frac{1}{2}\left(\cos\left(\frac{\pi}{2\sigma}\ \text{net}\right) + 1\right) & \text{otherwise} \end{cases}$$

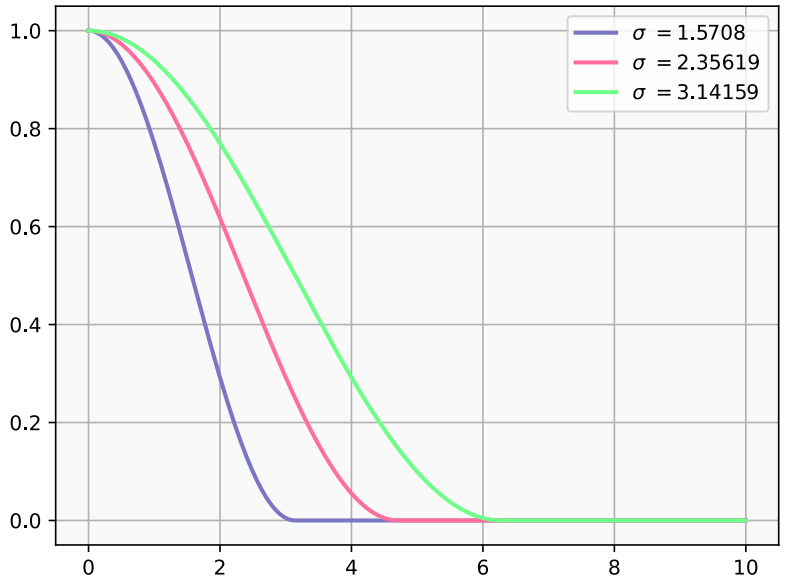


Figure 44: Plot of the cosine down to zero function, with three different choices of σ

The rectangular function, the triangular function and the cosine down to zero define a crisp boundary that separates an area where the activation is not 0 from an area where is 0. The difference is in how the activation scales with the distance from the center.

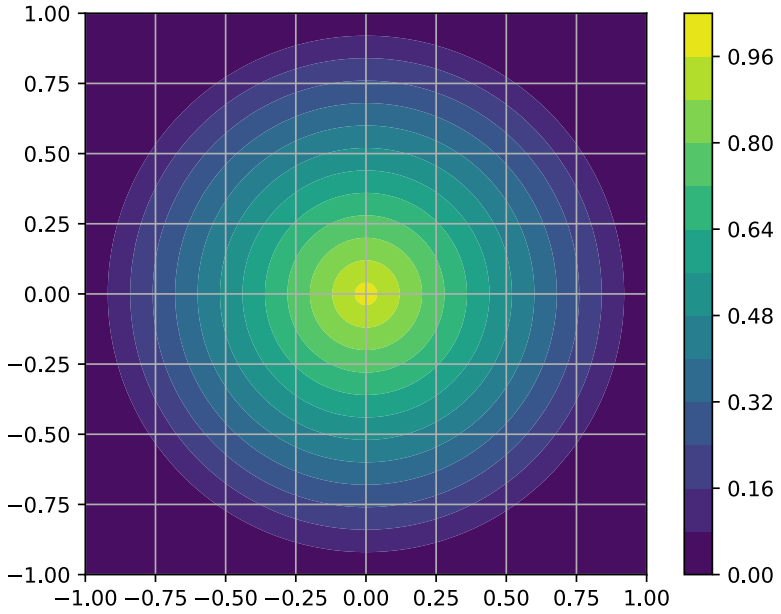


Figure 45: Plot of the value of the triangular function with $\sigma = 0.5$ for each point in the $[-1, 1] \times [-1, 1]$ plane, computed on the distance from the center $(0, 0)$. The value decreases as the point is further from the center, until becoming 0 when sufficiently far away

There are radial functions that have no boundary at all, meaning that they always assign a non-zero activation to any point of the domain. One example is the **Gaussian function**:

$$f_{\text{act}}(\text{net}, \sigma) = e^{-\frac{\text{net}^2}{2\sigma^2}}$$

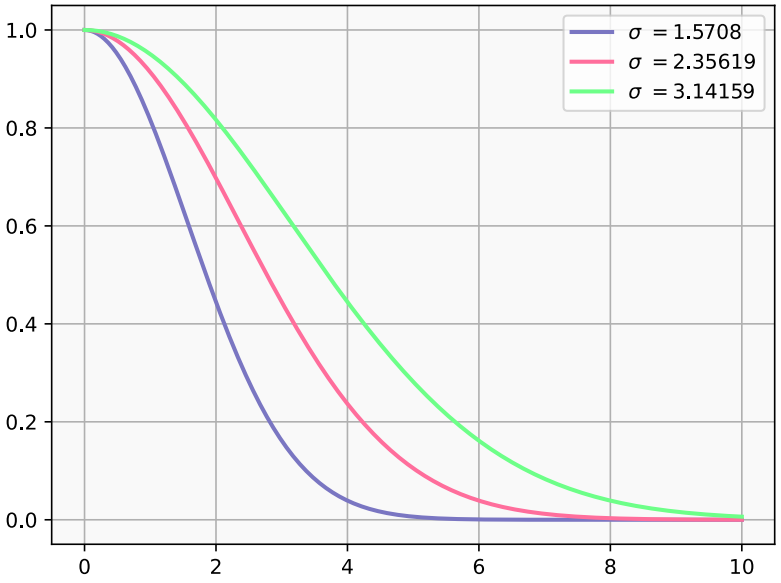


Figure 46: Plot of the gaussian function, with three different choices of σ

The activation is the highest in the center of the capture region and becomes smaller and smaller as the input is far away from the center, without ever going to 0. This makes it harder to give a visual representation of the function, since the capture region is, technically, the entire space, but has better approximation power.

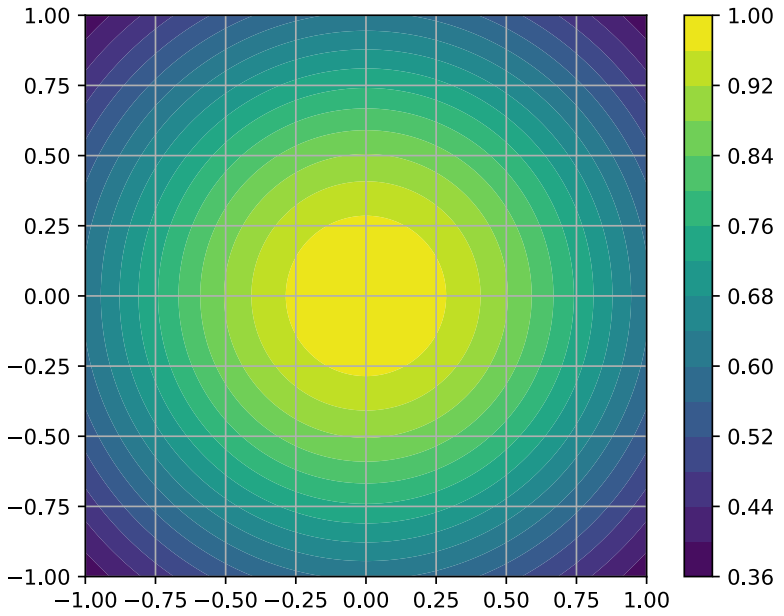


Figure 47: Plot of the value of the Gaussian function with $\sigma = 0.5$ for each point in the $[-1, 1] \times [-1, 1]$ plane, computed on the distance from the center $(0, 0)$. The value decreases as the point is further from the center, but never vanishes.

2.5.2. Approximating functions using a radial basis function network

Approximating functions using radial basis function networks follows the same logic as multilayer perceptrons. The idea is, again, to approximate a function as a stepwise function and encode them in a radial basis function network.

Theorem 2.5.2.1: Any Riemann-integrable function can be approximated with arbitrary accuracy by a radial basis function network.

Proof: The easiest way to do so is to use the rectangular function as radial basis function, since it has the natural shape of a pulse. In particular, a radial basis function network with one input layer and one output layer is constructed. The number of hidden layers is equal to the number of stepwise functions used for the approximation.

The weights coming into the hidden layer are the midpoints of the step, the σ parameter is half the width of the step and the weights coming into the output layer are the heights of the steps. This way, all hidden neurons output 0 except for a single neuron, the one in which the input lies. Its output, which is exactly 1, is then multiplied with which is then multiplied by the height of the corresponding step, resulting in the height itself. Since the degree of approximation can be increased at will, the result is proven. □

Exercise 2.5.2.1: Construct a 3-layer radial basis function network that encodes Figure 22.

Solution: Each step has width of 1, hence σ is always 0.5.

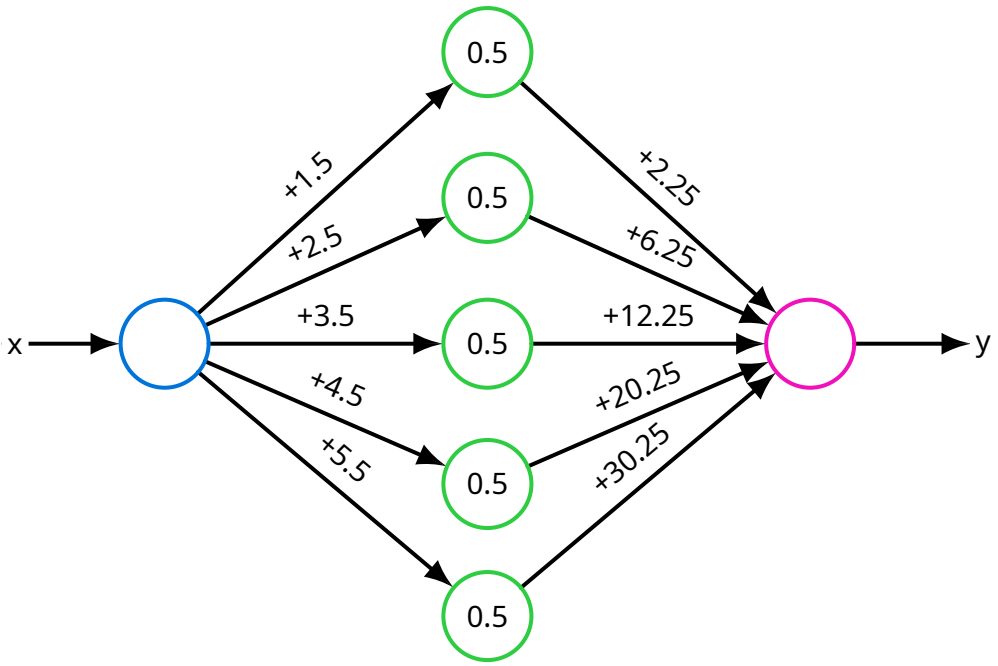


Figure 48: A radial basis function network that approximates $f(x)$ in the $[1, 6]$ interval.

□

As it was the case for multilayer perceptrons, Theorem 2.5.2.1 does not require the function to be approximated to be continuous. Also, it is trivial to extend the theorem to functions with any arity, since it is sufficient to consider multi-dimensional "step functions". In contrast to multilayer perceptrons, where discontinuous functions required 4 layers, an RBF network with 3 layer is always sufficient, because each neuron carries out its computations independently of the others.

The only caveat of using rectangular functions is that this independence is true only to some extent. That is, at the border points the value of the function is not computed correctly, because each region and the next are placed side-by-side. Note that this does not invalidate the theorem, because the approximation error is computed as the difference between the area under the curve and the total area of all the steps. This area is constituted by infinitely many points, while the "troublesome" points are finite (exactly two for each step), and therefore do not contribute to the error in any meaningful capacity.

However, if one desires to improve the accuracy of the model and remove this spuriousness, it is sufficient to substitute the rectangular function with another function. For example, using triangular functions or gaussian functions as radial basis functions gives an approximation that is both smoother and does not present the overlap problem.

Exercise 2.5.2.2: Approximate the following function with an RBF network, using Gaussian functions as activation functions:

$$g(x) = -\frac{1}{3}x^5 + \frac{5}{3}x^4 - \frac{1}{3}x^3 - \frac{23}{3}x^2 + \frac{56}{6}x - 2$$

Solution:

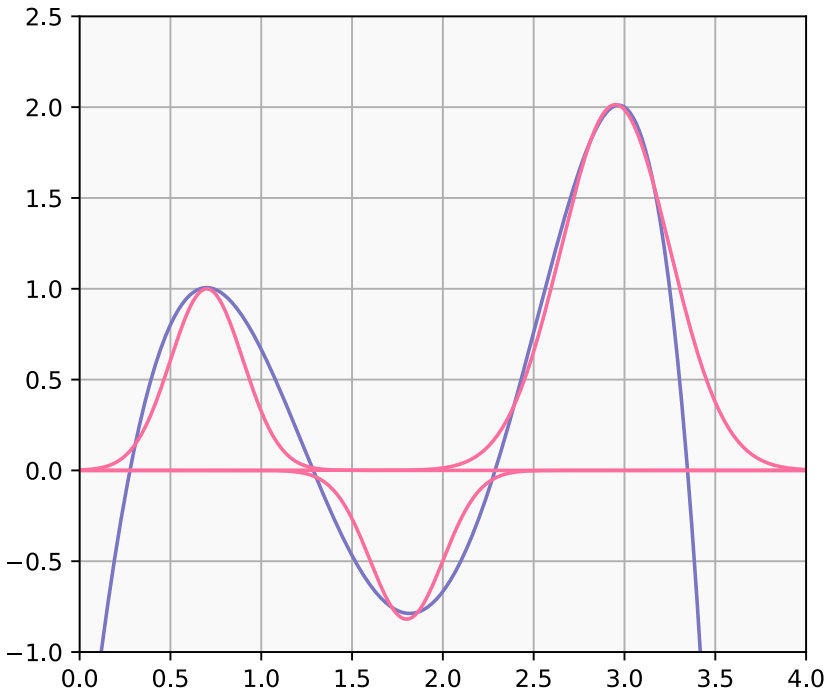


Figure 49: The given function, approximated by three Gaussian functions.

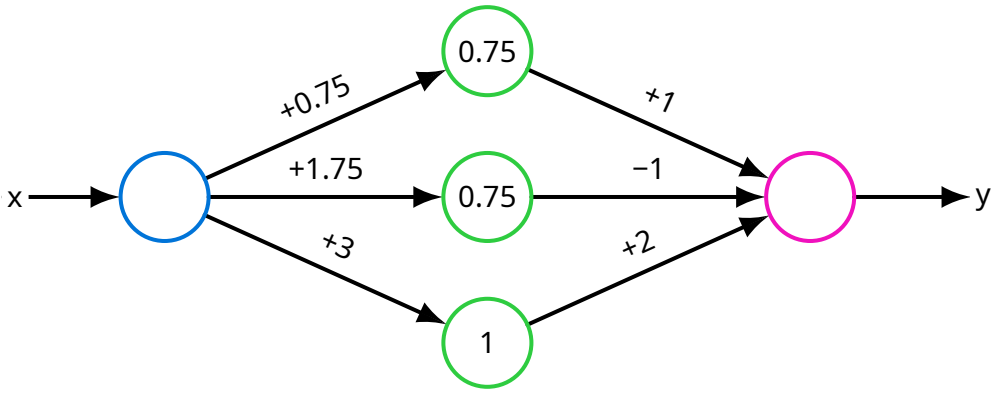


Figure 50: A radial basis function network that approximates $g(x)$ in the $[0, 4]$ interval.

□

2.5.3. Training a radial basis function network

Even though RBF networks have a more rigid structure than multilayer perceptron, their training has to be performed with care. This is because the neurons of each of the three layers have different functions, hence have to be treated separately. Even the way in which the parameters of an RBF are initialized is to be done with care.

Consider the case of a **simple radial basis function network**, where each training example is covered by its own radial basis function. That is, the number of hidden neurons is equal to the size of the training dataset and each hidden neuron "takes care" of representing a single training pattern. In particular, the weights coming into a hidden neuron are initialized with the values of the inputs of a single training pattern.

More formally, consider a fixed learning task $L_{\text{fixed}} = \{l_1, ..., l_m\}$ of m training patterns. The simple RBF network that is to learn L_{fixed} has exactly m hidden neurons $v_1, ..., v_m$. For each hidden neuron v_k,

the incoming weights $\boldsymbol{w}_{(v)_k}$ are initialized precisely to $\boldsymbol{i}^{(l_k)}$, the input contained in the k-th training pattern $l_k = \left(\boldsymbol{i}^{(l_k)}, \boldsymbol{o}^{(l_k)}\right)$.

If the Gaussian activation function is chosen, the σ parameter (the radius of the "circle") is often initialized according to the heuristic:

$$\sigma_k = \frac{d_{\max}}{\sqrt{2m}} = \frac{\max_{l_j, l_k \in L_{\text{fixed}}} \left\{ d\left(\boldsymbol{i}^{(l_j)}, \boldsymbol{i}^{(l_k)}\right) \right\}}{\sqrt{2m}} \ \forall k \in \{1, ..., m\}$$

Where $d_{\max}$ is the maximum distance that occurs between any two inputs of the training pattern (using the same distance as the one used in the network). This way, the Gaussian curves are neither too narrow nor too wide.

After having initialized the weights and the σ parameter of the hidden layer, it is necessary to initialize the weights and the θ parameter of the output layer. The idea is very simple: the output neurons have to be adapted in such a way that the output coming from the hidden neurons precisely results in the output of the training examples. This can be done easily because the weights coming into the hidden layer have been initialized precisely with the training inputs, hence their output is entirely known.

Since the network input function of the output neurons is a weighted sum of their inputs and their activation and output functions are both linear, each training example l yields for each output neuron u one linear equation:

$$\sum_{k=1}^{m} w_{u,v_k} \text{out}_{v_k}^{(l)} - \theta_u = w_{u,v_1} \text{out}_{v_1}^{(l)} + ... + w_{u,v_m} \text{out}_{v_m}^{(l)} - \theta_u = o_u^{(l)}$$

Where $v_1, ..., v_m$ are the hidden layers. All $o_u^{(l)}$ are known for all $l \in L_{\text{fixed}}$, since they come directly from the given training examples. All $\text{out}_{v_m}^{(l)}$ are also known for all $l \in L_{\text{fixed}}$, since they're the output of the hidden neurons, whose parameters have been fixed. On the other hand, θ_u and $\sum_{k=1}^{m} w_{u,v_k}$ are not known, and are the parameters to be determined.

Therefore, the expression results in a system of m equations, one for each training pattern, of $m + 1$ unkowns (θ_u and $\sum_{k=1}^{m} w_{u,v_k}$). Since there are more unknown than equations, to have a solvable system it is necessary to reduce the degrees of freedom by (at least) 1. This is easily done by setting $\theta_u = 0$. The resulting system of equations is:

$$\begin{cases} \sum_{k=1}^{m} \text{out}_{v_k}^{(l_1)} w_{u,v_k} = \text{out}_{v_1}^{(l_1)} w_{u,v_1} + ... + \text{out}_{v_m}^{(l_1)} w_{u,v_m} = o_u^{(l_1)} \\ \sum_{k=1}^{m} \text{out}_{v_k}^{(l_2)} w_{u,v_k} = \text{out}_{v_1}^{(l_2)} w_{u,v_1} + ... + \text{out}_{v_m}^{(l_2)} w_{u,v_m} = o_u^{(l_2)} \\ \vdots \\ \sum_{k=1}^{m} \text{out}_{v_k}^{(l_m)} w_{u,v_k} = \text{out}_{v_1}^{(l_m)} w_{u,v_1} + ... + \text{out}_{v_m}^{(l_m)} w_{u,v_m} = o_u^{(l_m)} \end{cases}$$

Or, in matrix form:

$$\begin{pmatrix} \text{out}_{v_1}^{(l_1)} & \text{out}_{v_2}^{(l_1)} & ... & \text{out}_{v_m}^{(l_1)} \\ \text{out}_{v_1}^{(l_2)} & \text{out}_{v_2}^{(l_2)} & ... & \text{out}_{v_m}^{(l_2)} \\ \vdots & \vdots & \ddots & \vdots \\ \text{out}_{v_1}^{(l_m)} & \text{out}_{v_2}^{(l_m)} & ... & \text{out}_{v_m}^{(l_m)} \end{pmatrix} \begin{pmatrix} w_{u,v_1} \\ w_{u,v_2} \\ \vdots \\ w_{u,v_m} \end{pmatrix} = \begin{pmatrix} o_u^{(l_1)} \\ o_u^{(l_2)} \\ \vdots \\ o_u^{(l_m)} \end{pmatrix} \Rightarrow \boldsymbol{A}\boldsymbol{w}_u = \boldsymbol{o}_u$$

Since it's actually $\boldsymbol{w}_u$ the vector of interest, it's convenient to rewrite the expression as:

$$\boldsymbol{w}_u = \boldsymbol{A}^{-1}\boldsymbol{o}_u$$

Which is then solved as a standard product between two matrices, assuming that $\boldsymbol{A}$ is invertible. If it's not, weights have to be chosen randomly until the remaining equation system is uniquely solvable.

Note how a simple RBF network requires no training at all: the weights coming into the output neuron are defined so that they exactly match the outputs, and it's always possible to compute them with the aforementioned formula, hence the error is always zero.

Exercise 2.5.3.1: Construct a simple RBF network for the biimplication, using the Euclidean distance as distance function and the Gaussian function as radial basis function.

Solution: The biimplication of two propositions is true if both are true or both are false. Hence, the training patterns are:

$$L_{\text{fixed}} = \{((0,0),1),((1,0),0),((0,1),0),((1,1),1)\}$$

Which gives a total of 4 hidden neurons. The maximum distance is between $(0,0)$ and $(1,1)$, which is $\sqrt{2}$, therefore the σ parameter is initialized to $\sqrt{2}/\sqrt{2 \cdot 4} = 1/2$.

Since the output function of the hidden neurons is the identity, their output is directly known from their activation. The output of the hidden neuron u for the training pattern l is:

$$\text{out}_u^{(l)} = f_{\text{out}}\left(\text{act}_u^{(l)}\right) = \text{act}_u^{(l)} = f_{\text{act}}\left(\text{net}_u^{(l)}\right) = e^{-\frac{\left(\text{net}_u^{(l)}\right)^2}{2(1/2)^2}} = e^{-2\left(\text{net}_u^{(l)}\right)^2}$$

Where $\text{net}_u^{(l)}$ is the distance between the two inputs. The distance between two identical inputs is 0, the distance between two inputs that have one input in common is 1, the distance between two inputs that have nothing in common is $\sqrt{2}$. This gives the following matrix:

$$\boldsymbol{A} = \begin{pmatrix} 1 & e^{-2} & e^{-2} & e^{-4} \\ e^{-2} & 1 & e^{-4} & e^{-2} \\ e^{-2} & e^{-4} & 1 & e^{-2} \\ e^{-4} & e^{-2} & e^{-2} & 1 \end{pmatrix} \qquad \boldsymbol{A}^{-1} = \begin{pmatrix} 1.0378 & -0.1404 & -0.1404 & 0.0190 \\ -0.1404 & 1.0378 & 0.0190 & -0.1404 \\ -0.1404 & 0.0190 & 1.0378 & -0.1404 \\ 0.0190 & -0.1404 & -0.1404 & 1.0378 \end{pmatrix}$$

The vector of weights for one and only output neuron is:

$$\boldsymbol{w}_u = \boldsymbol{A}^{-1}\boldsymbol{o}_u = \begin{pmatrix} 1.0378 & -0.1404 & -0.1404 & 0.0190 \\ -0.1404 & 1.0378 & 0.0190 & -0.1404 \\ -0.1404 & 0.0190 & 1.0378 & -0.1404 \\ 0.0190 & -0.1404 & -0.1404 & 1.0378 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \approx \begin{pmatrix} 1.0567 \\ -0.2809 \\ -0.2809 \\ 1.0567 \end{pmatrix}$$

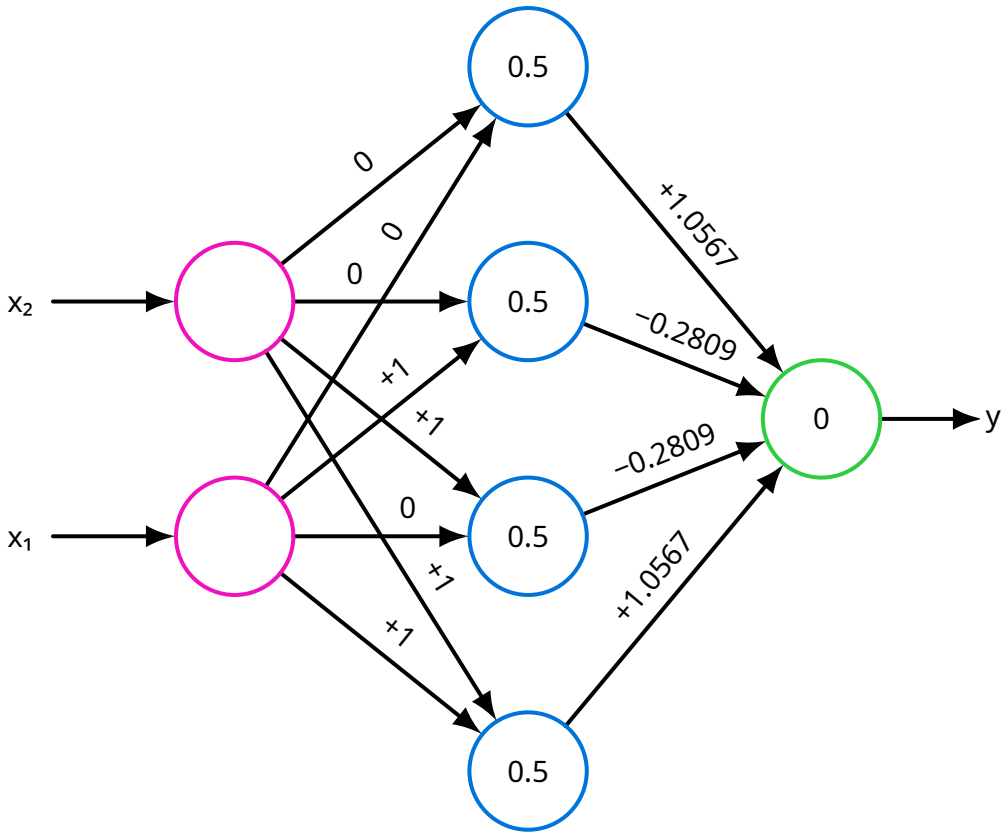


Figure 51: A simple radial basis function network for the biimplication.

□

Simple RBF networks have the advantage that don't need to be trained, since all parameters are fixed by hand and give solutions that will always perfectly match the examples. However, they cannot be used in practice, because in general the size of the training dataset is too large to feasibly assign a single hidden neuron to each example.

This is why (not simple) RBF networks are much more useful. To construct an RBF network, the idea is start from a simple RBF network that encodes a subset of the examples, initialize its parameters in the standard way and then train the network with the entire set of examples.

Consider a fixed learning task L_{fixed} of m examples. Take a subset of size $t < m$ of the training examples and construct a simple RBF network out of this subset. The patterns in the subset should capture the general trend of the data, in order to be as representative as possible. This can be done, for example, by performing k-means on the training dataset with $k = t$ and constructing the subset out of the t centroids.

As expected, the weights coming into said neurons are initialized with the values of the possible inputs in the subset, and the σ parameters as $d_{\max}/\sqrt{2m}$. The difference is that now the expression:

$$\sum_{k=1}^{t} w_{u,v_k} \text{out}_{v_k}^{(l)} - \theta_u = o_u^{(l)}$$

Leads to a system of equations that has m equations (one for each example of the whole set) and $t + 1$ unknowns (the weights of the t neurons and the θ parameter). Since $m < t$, the θ parameter(s) are not set to 0, but are instead embed into the weights, in order to obtain an extra degree of freedom. This gives:

$$\begin{pmatrix} 1 & \text{out}_{v_1}^{(l_1)} & \text{out}_{v_2}^{(l_1)} & \dots & \text{out}_{v_t}^{(l_1)} \\ 1 & \text{out}_{v_1}^{(l_2)} & \text{out}_{v_2}^{(l_2)} & \dots & \text{out}_{v_t}^{(l_2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \text{out}_{v_1}^{(l_m)} & \text{out}_{v_2}^{(l_m)} & \dots & \text{out}_{v_t}^{(l_m)} \end{pmatrix} \begin{pmatrix} -\theta_u \\ w_{u,v_1} \\ w_{u,v_2} \\ \vdots \\ w_{u,v_t} \end{pmatrix} = \begin{pmatrix} o_u^{(l_1)} \\ o_u^{(l_2)} \\ \vdots \\ o_u^{(l_m)} \end{pmatrix} \Rightarrow \boldsymbol{A}\boldsymbol{w}_u = \boldsymbol{o}_u$$

The problem is that, even with this embedding, m is still larger than the number of unknowns $(t+1)$. This means that A is not a square matrix, and therefore cannot be moved to the right hand side it was the case for simple RBF networks. Nevertheless, a satisfactory approximation is given by the pseudoinverse $\boldsymbol{A}^+ = \left(\boldsymbol{A}^T\boldsymbol{A}\right)^{-1}\boldsymbol{A}^T$. This gives:

$$\boldsymbol{w}_u \approx \boldsymbol{A}^+\boldsymbol{o}_u = \left(\boldsymbol{A}^T\boldsymbol{A}\right)^{-1}\boldsymbol{A}^T\boldsymbol{o}_u$$

Exercise 2.5.3.2: Consider Exercise 2.5.3.1. Write an equivalent RBF network.

Solution: Suppose that, out of all input combinations in L_{fixed}, only $((0,0),1)$ and $((1,1),1)$ are chosen. This gives the following 4×3 (two patterns plus one extra input to accomodate θ) matrix:

$$\boldsymbol{A} = \begin{pmatrix} 1 & 1 & e^{-4} \\ 1 & e^{-2} & e^{-2} \\ 1 & e^{-2} & e^{-2} \\ 1 & e^{-4} & 1 \end{pmatrix} \qquad \boldsymbol{A}^+ = \begin{pmatrix} -0.1810 & 0.6810 & 0.6810 & -0.1810 \\ 1.1780 & -0.6687 & -0.6687 & 0.1594 \\ 0.1594 & -0.6687 & -0.6687 & 1.1780 \end{pmatrix}$$

Where the distances and the σ parameter are taken directly from Exercise 2.5.3.1. This gives:

$$\boldsymbol{w}_u \approx \boldsymbol{A}^+\boldsymbol{o}_u = \begin{pmatrix} -0.1810 & 0.6810 & 0.6810 & -0.1810 \\ 1.1780 & -0.6687 & -0.6687 & 0.1594 \\ 0.1594 & -0.6687 & -0.6687 & 1.1780 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -0.3620 \\ 1.3375 \\ 1.3375 \end{pmatrix}$$

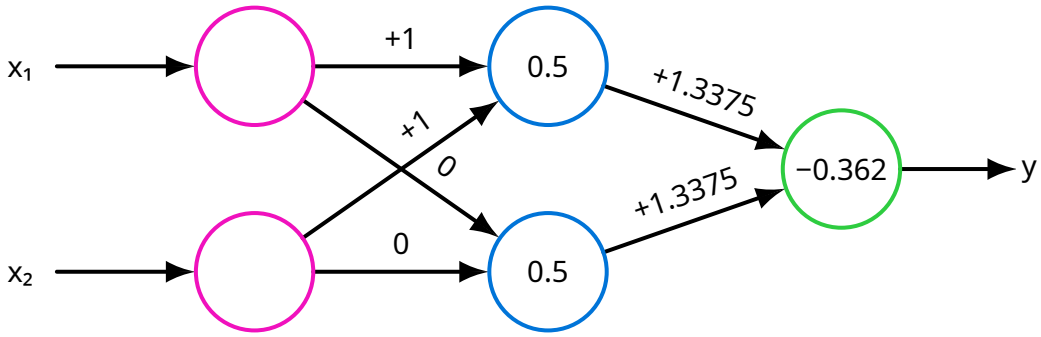


Figure 52: A radial basis function network for the biimplication.

□

Now that the RBF network has been initialized, it has to be trained. The training process follows the same approach for the training of a multilayer perceptron, applying the backpropagation algorithm.

Training an RBF network is actually easier than training a multilayer perceptron. This is because while the latter has to account for an arbitrary number of layers, an RBF network always has three. This means that each training run has exactly three steps: training the weights coming into the output layer, training the σ parameters of the hidden layer, training the weights coming into the hidden layer.

Training the weights coming into the hidden layer can be borrowed one-for-one from the training of a multilayer perceptron. This is because the network input function, activation function and output function of the output layer of an RBF are identical to those of a multilayer perceptron. The gradient for a single output neuron u and a single training pattern l is therefore:

$$\nabla \boldsymbol{w}_u e_u^{(l)} = \frac{\partial e_u^{(l)}}{\partial \boldsymbol{w}_u} = -2\Big(o_u^{(l)} - \mathrm{out}_u^{(l)}\Big)\frac{\partial \mathrm{out}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}}\mathbf{in}_u^{(l)}$$

For a fixed learning task $L_{\text{fixed}} = \big\{\big(\boldsymbol{i}^{(l)}, \boldsymbol{o}^{(l)}\big)\big\}$. Note that the θ parameter of the output layer, as it was the case for the multilayer perceptron, has been embed into the weights to simplify calculations.

Assuming, as it is generally the case, that the output function of the output layer is the identity:

$$\nabla \boldsymbol{w}_u e_u^{(l)} = -2\Big(o_u^{(l)} - \mathrm{out}_u^{(l)}\Big)\frac{\partial \mathrm{out}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}}\mathbf{in}_u^{(l)} = -2\Big(o_u^{(l)} - \mathrm{out}_u^{(l)}\Big)\frac{\partial \mathrm{act}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}}\mathbf{in}_u^{(l)} =$$

$$= -2\Big(o_u^{(l)} - \mathrm{out}_u^{(l)}\Big)\frac{\partial \mathrm{net}_u^{(l)}}{\partial \mathrm{net}_u^{(l)}}\mathbf{in}_u^{(l)} = -2\Big(o_u^{(l)} - \mathrm{out}_u^{(l)}\Big)\mathbf{in}_u^{(l)}$$

Which gives the adaptation rule:

$$\Delta \boldsymbol{w}_u^{(l)} = -\frac{\eta}{2}\nabla \boldsymbol{w}_u e_u^{(l)} = \eta_3\Big(o_u^{(l)} - \mathrm{out}_u^{(l)}\Big)\mathbf{in}_u^{(l)}$$

Note the presence of a pedix in the learning rate. This is because each of the three steps should use its own learning rate.

The training of the weights coming into the hidden layers cannot be borrowed from the training of multilayer perceptrons, because the are different types of functions at play. In particular, the network input function of the hidden layer is not a weighted sum, but a distance function. This means that it's not possible to embed the σ parameter into the weights, and the training of the weights and of the σ parameters has to be carried out separately.

Consider an neuron v belonging to the hidden layer U_{hidden} Its predecessors, which are just a subset of the input neurons, are given by $\mathrm{pred}(v) = \{p_1, ..., p_n\} \subseteq U_{\text{in}}$. The corresponding vector of weights is $\boldsymbol{w}_v = \big(w_{v,p_1}, ..., w_{v,p_n}\big)$. The gradient of the total error function with respect to these weights is:

$$\nabla_{\boldsymbol{w}_v} e = \frac{\partial e}{\partial \boldsymbol{w}_v} = \left(-\frac{\partial e}{\partial \theta_v}, \frac{\partial e}{\partial w_{v,p_1}}, ..., \frac{\partial e}{\partial w_{v,p_n}}\right)$$

Explicitly substituting the expression for e:

$$\nabla_{\boldsymbol{w}_v} e = \frac{\partial e}{\partial \boldsymbol{w}_v} = \frac{\partial}{\partial \boldsymbol{w}_v}\sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{l \in L_{\text{fixed}}}\frac{\partial e^{(l)}}{\partial \boldsymbol{w}_v} = \sum_{l \in L_{\text{fixed}}}\nabla_{\boldsymbol{w}_v} e^{(l)}$$

Consider a single training pattern l and its error $e^{(l)} = \sum_{u \in U_{\text{out}}} e_u^{(l)}$. The error depends on the value of the output of the layer, which in turn depends (also) on the network input. However, the network input depends on the weights:

$$\mathrm{net}_v^{(l)} = d\Big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\Big) = d\Big(\boldsymbol{w}_v, \big(\mathrm{out}_{p_1}^{(l)}, ..., \mathrm{out}_{p_n}^{(l)}\big)\Big)$$

Which means that there's a dependency between the error and the weights. Applying the chain rule:

$$\nabla_{\boldsymbol{w}_v} e^{(l)} = \frac{\partial e^{(l)}}{\partial \boldsymbol{w}_v} = \frac{\partial e^{(l)}}{\partial \mathrm{net}_v^{(l)}}\frac{\partial \mathrm{net}_v^{(l)}}{\partial \boldsymbol{w}_v} = \frac{\partial e^{(l)}}{\partial \mathrm{net}_v^{(l)}}\frac{\partial d\Big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\Big)}{\partial \boldsymbol{w}_v}$$

Expanding the error $e^{(l)}$ in the first factor:

$$\nabla_{\boldsymbol{w}_v} e^{(l)} = \frac{\partial e^{(l)}}{\partial \text{net}_v^{(l)}} \frac{\partial d\big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\big)}{\partial \boldsymbol{w}_v} = \frac{\partial \sum_{u \in U_{\text{out}}} \big(o_u^{(l)} - \text{out}_u^{(l)}\big)^2}{\partial \text{net}_v^{(l)}} \frac{\partial d\big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\big)}{\partial \boldsymbol{w}_v} =$$

$$= \sum_{u \in U_{\text{out}}} \frac{\partial \big(o_u^{(l)} - \text{out}_u^{(l)}\big)^2}{\partial \text{net}_v^{(l)}} \frac{\partial d\big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\big)}{\partial \boldsymbol{w}_v} =$$

$$= \sum_{u \in U_{\text{out}}} 2\big(o_u^{(l)} - \text{out}_u^{(l)}\big) \frac{\partial \big(o_u^{(l)} - \text{out}_u^{(l)}\big)}{\partial \text{net}_v^{(l)}} \frac{\partial d\big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\big)}{\partial \boldsymbol{w}_v} =$$

$$= \sum_{u \in U_{\text{out}}} 2\big(o_u^{(l)} - \text{out}_u^{(l)}\big) \left(\cancel{\frac{\partial o_u^{(l)}}{\partial \text{net}_v^{(l)}}} - \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_v^{(l)}} \right) \frac{\partial d\big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\big)}{\partial \boldsymbol{w}_v} =$$

$$= -2 \sum_{u \in U_{\text{out}}} \big(o_u^{(l)} - \text{out}_u^{(l)}\big) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_v^{(l)}} \frac{\partial d\big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\big)}{\partial \boldsymbol{w}_v}$$

Where $\partial o_u^{(l)} / \partial \text{net}_v^{(l)}$ is 0 since $o_u^{(l)}$ is a known constant.

Let $\text{succ}(v) = \{s_1, ..., s_n\} \subseteq U_{\text{out}}$ be the successors of the hidden neuron v, which are a subset of the output neurons. The output of an output neuron u depends on (the network input of) v only if there is a connection from v to u. If there is no connection, meaning that $u \notin \text{succ}(v)$, the term $\partial \text{out}_u^{(l)} / \partial \text{net}_v^{(l)}$ is just 0. Therefore, it is justified to rewrite the expression above as:

$$\nabla_{\boldsymbol{w}_v} e^{(l)} = -2 \sum_{s \in \text{ succ}(v)} \big(o_s^{(l)} - \text{out}_s^{(l)}\big) \frac{\partial \text{out}_s^{(l)}}{\partial \text{net}_v^{(l)}} \frac{\partial d\big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\big)}{\partial \boldsymbol{w}_v}$$

That is, by removing all the terms that give no contribution to the summation.

Since the neurons s and v are by definition adjacent, the output of of s is dependent on the network input of v only with respect to the network input of s. Applying the chain rule:

$$\nabla_{\boldsymbol{w}_v} e^{(l)} = -2 \sum_{s \in \text{ succ}(v)} \big(o_s^{(l)} - \text{out}_s^{(l)}\big) \frac{\partial \text{out}_s^{(l)}}{\partial \text{net}_v^{(l)}} \frac{\partial d\big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\big)}{\partial \boldsymbol{w}_v} =$$

$$= -2 \sum_{s \in \text{ succ}(v)} \big(o_s^{(l)} - \text{out}_s^{(l)}\big) \cancel{\frac{\partial \text{out}_s^{(l)}}{\partial \text{net}_s^{(l)}}} \frac{\partial \text{net}_s^{(l)}}{\partial \text{net}_v^{(l)}} \frac{\partial d\big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\big)}{\partial \boldsymbol{w}_v} =$$

$$= -2 \sum_{s \in \text{ succ}(v)} \big(o_s^{(l)} - \text{out}_s^{(l)}\big) \frac{\partial \text{net}_s^{(l)}}{\partial \text{net}_v^{(l)}} \frac{\partial d\big(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\big)}{\partial \boldsymbol{w}_v}$$

Where $\partial \text{out}_s^{(l)} / \partial \text{net}_s^{(l)}$ is 1 because s is an output neuron, and therefore its output function is the identity. As for $\partial \text{net}_s^{(l)} / \partial \text{net}_v^{(l)}$, since s is an output neuron, its network input functon is a weighted summation. In particular:

$$\frac{\partial \text{net}_s^{(l)}}{\partial \text{net}_v^{(l)}} = \frac{\partial\left(\boldsymbol{w}_s \mathbf{in}_s^{(l)} - \theta_s\right)}{\partial \text{net}_v^{(l)}} = \frac{\partial\left(-\theta_s + \sum_{p\in \text{ pred}(s)} w_{s,p}\text{out}_p^{(l)}\right)}{\partial \text{net}_v^{(l)}} =$$

$$= \frac{\partial \sum_{p\in \text{ pred}(s)} w_{s,p}\text{out}_p^{(l)}}{\partial \text{net}_v^{(l)}} - \frac{\partial \theta_s}{\partial \text{net}_v^{(l)}} = \sum_{p\in \text{ pred}(s)} \frac{\partial w_{s,p}\text{out}_p^{(l)}}{\partial \text{net}_v^{(l)}} - \cancel{\frac{\partial \theta_s}{\partial \text{net}_v^{(l)}}} =$$

$$= \sum_{p\in \text{ pred}(s)} w_{s,p}\frac{\partial \text{out}_p^{(l)}}{\partial \text{net}_v^{(l)}}$$

Out of all neurons $p \in \text{pred}(s)$ there is precisely the neuron v, since s is one of its successors. This means that the only term of $\partial \text{net}_s^{(l)}/\partial \text{net}_v^{(l)}$ that is not null is the one where $p = v$, because it's the only dependency between $\text{net}_s^{(l)}$ and $\text{net}_v^{(l)}$:

$$\frac{\partial \text{net}_s^{(l)}}{\partial \text{net}_v^{(l)}} = \sum_{p\in \text{ pred}(s)} w_{s,p}\frac{\partial \text{out}_p^{(l)}}{\partial \text{net}_v^{(l)}} = \sum_{p=v} w_{s,p}\frac{\partial \text{out}_p^{(l)}}{\partial \text{net}_v^{(l)}} = w_{s,v}\frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_v^{(l)}}$$

Substituting in the previous expression:

$$\nabla_{\boldsymbol{w}_v} e^{(l)} = -2 \sum_{s\in \text{ succ}(v)} \left(o_s^{(l)} - \text{out}_s^{(l)}\right) \frac{\partial \text{net}_s^{(l)}}{\partial \text{net}_v^{(l)}} \frac{\partial d\left(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\right)}{\partial \boldsymbol{w}_v} =$$

$$= -2 \sum_{s\in \text{ succ}(v)} \left(o_s^{(l)} - \text{out}_s^{(l)}\right) w_{s,v} \frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_v^{(l)}} \frac{\partial d\left(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\right)}{\partial \boldsymbol{w}_v}$$

Which gives the following adaptation rule:

$$\Delta \boldsymbol{w}_v^{(l)} = -\frac{\eta_1}{2}\nabla_{\boldsymbol{w}_v} e^{(l)} = \eta_1 \sum_{s\in \text{ succ}(v)} \left(o_s^{(l)} - \text{out}_s^{(l)}\right) w_{s,v} \frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_v^{(l)}} \frac{\partial d\left(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\right)}{\partial \boldsymbol{w}_v}$$

An explicit expression of the adaptation depends on the choice of the distance function and on the choice of the radial basis function. Assuming, as it is often the case, that the distance is the Euclidean distance, the last factor becomes:

$$\frac{\partial d\left(\boldsymbol{w}_v, \mathbf{in}_v^{(l)}\right)}{\partial \boldsymbol{w}_v} = \frac{\sqrt{\sum_{i=1}^{n}\left(w_{v,p_i} - \text{out}_{p_i}^{(l)}\right)^2}}{\partial \boldsymbol{w}_v} = \left(\sum_{i=1}^{n}\left(w_{v,p_i} - \text{out}_{p_i}^{(l)}\right)^2\right)^{-\frac{1}{2}}\left(\boldsymbol{w}_v - \mathbf{in}_v^{(l)}\right)$$

Assuming then (again, as it is often the case) that the radial basis function is the Gaussian function, the butlast factor becomes:

$$\frac{\partial \text{out}_v^{(l)}}{\partial \text{net}_v^{(l)}} = \frac{\partial f_{\text{act}}\left(\text{net}_v^{(l)}, \sigma_v\right)}{\partial \text{net}_v^{(l)}} = \frac{\partial e^{-\frac{\left(\text{net}_v^{(l)}\right)^2}{2\sigma_v^2}}}{\partial \text{net}_v^{(l)}} = e^{-\frac{\left(\text{net}_v^{(l)}\right)^2}{2\sigma_v^2}} \frac{\partial}{\partial \text{net}_v^{(l)}}\left(-\frac{\left(\text{net}_v^{(l)}\right)^2}{2\sigma_v^2}\right) =$$

$$= e^{-\frac{\left(\text{net}_v^{(l)}\right)^2}{2\sigma_v^2}} \frac{-1}{2\sigma_v^2}\frac{\partial\left(\text{net}_v^{(l)}\right)^2}{\partial \text{net}_v^{(l)}} = e^{-\frac{\left(\text{net}_v^{(l)}\right)^2}{2\sigma_v^2}} \frac{-\cancel{2}\text{net}_v^{(l)}}{\cancel{2}\sigma_v^2} = -\frac{\text{net}_v^{(l)}}{\sigma_v^2} e^{-\frac{\left(\text{net}_v^{(l)}\right)^2}{2\sigma_v^2}}$$

Substituting both factors:

$$\Delta \boldsymbol{w}_v^{(l)} = -\eta_1 \sum_{s \in \text{ succ}(v)} \left(o_s^{(l)} - \text{out}_s^{(l)}\right) w_{s,v} \frac{\frac{\text{net}_v^{(l)}}{\sigma_v^2} e^{-\frac{\left(\text{net}_v^{(l)}\right)^2}{2\sigma_v^2}}}{\sum_{i=1}^{n} \sqrt{\left(w_{v,p_i} - \text{out}_{p_i}^{(l)}\right)^2}} \left(\boldsymbol{w}_v - \mathbf{in}_v^{(l)}\right)$$

The last training step is the update of the σ parameters. The derivation follows the same steps as the derivation of the adaptation of the weights:

$$\frac{\partial e^{(l)}}{\partial \sigma_v} = -2 \sum_{s \in \text{ succ}(v)} \left(o_s^{(l)} - \text{out}_s^{(l)}\right) w_{s,v} \frac{\partial \text{out}_v^{(l)}}{\partial \sigma_v}$$

Which gives the adaptation:

$$\Delta \sigma_v^{(l)} = -\frac{\eta_2}{2} \frac{\partial e^{(l)}}{\partial \sigma_v} = \eta_2 \sum_{s \in \text{ succ}(v)} \left(o_s^{(l)} - \text{out}_s^{(l)}\right) w_{s,v} \frac{\partial \text{out}_v^{(l)}}{\partial \sigma_v}$$

As stated, three different learning rates are here at play: one for the weights coming into the output layer (η_3), one for the weights coming into the hidden layer (η_1) and one for the σ parameters of the hidden layer (η_2). This is because it has been shown empirically that the weights coming into the output layer have a much stronger influence on the final output than any other parameter of the network. For this reason, it should be adapted much more slowly than the rest.

2.6. Learning vector quantization networks

2.6.1. K-means clustering

The neural network models presented so far were designed to solve fixed learning tasks: approximating a function given input-output pairs of training examples. There are also neural networks designed to solve free learning tasks, also known as **clustering**. Clustering consists in dividing a set of data points into subgroups, or **clusters**, whose members share similar characteristics.

Among the different forms of clustering, **partitional clustering** attempts to divide the datapoints into partitions, creating a tessellation of the input space. If the interest is in assigning to each point its own partition (the number of clusters matches the size of the data), a visual representation of partitional clustering is given by the so-called **Voronoi diagram**:

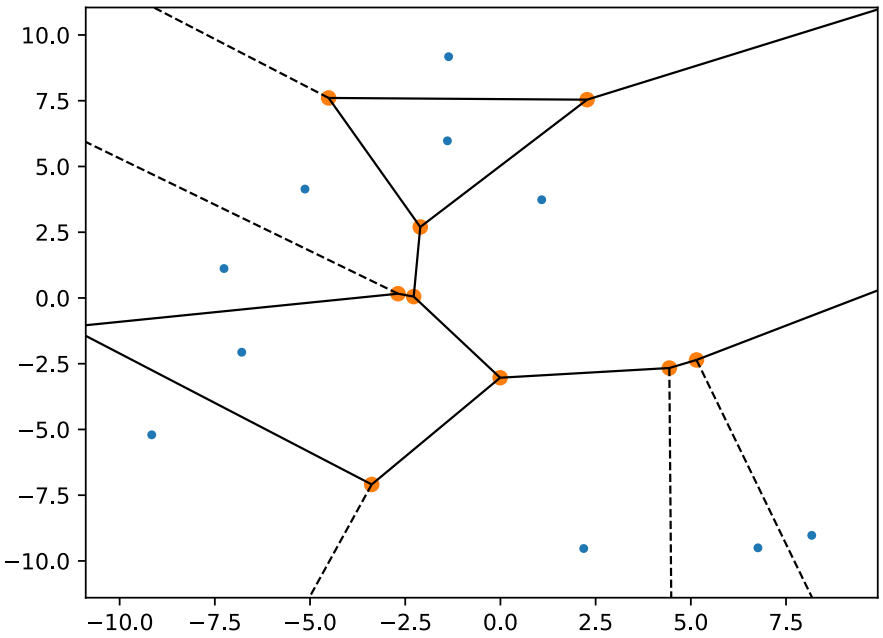


Figure 53: Example of a Voronoi diagram for 10 data points.

The simplest (and likely earliest) partitional clustering algorithm is **K-means**. The algorithm is as follows:

1. Fix the number of clusters k;
2. Choose k random points in the dataset (called **seeds**), as representatives of the initial clusters. These representatives, that act like a "center of gravity" of their respective cluster, are called **centroids**;
3. Compute the distance between each element of the dataset and each centroid. Each element is assigned to the cluster whose centroid is the closest;
4. The centroids of each cluster are computed as the arithmetic mean between all elements of the cluster;
5. If a termination criteria is met, stop. Otherwise, go back to 3.

The algorithm does not specify a termination criteria. One option is to fix a maximum number of iterations; once this number is reached, the algorithm stops. A slightly more refined option is fix a small value ε and inspect how much the new centroids differ from the previous ones: if all of them have a difference smaller than ε, the algorithm stops. These two criteria are not mutually exclusive, and can be employed at the same time.

The main selling point of K-means is that it's fast. The time bound for its execution is $O(tkn)$, with t being the number of iterations, n the size of the dataset and k the number of clusters. Since k is fixed and t is either capped at a maximum or just nowhere close to n, the execution time is almost linear.

However, K-means is not free from issues. Namely, it can only operate on data points that are real numbers, because distance functions are defined only on reals. Moreover, the number of clusters k is not a byproduct of the algorithm itself, but has to be fixed beforehand. For any (integer) choice of k the algorithm will work, but a poor choice of k might fail to capture an actual cluster structure of the dataset. Also, K-means is very sensitive to the choice of the initial seeds: two different choice of seeds for the same dataset can give completely different outcomes (K-means is said to be an **unstable algorithm**). Finally, K-means relies on the tacit assumption that the similarity between elements should be defined in terms of their reciprocal distance, but this isn't necessarely the case.

Despite its shortcomings, K-means (and its variants) remains one of the most widely employed clustering algorithms, due to its simplicity and efficiency. It can be used as a kind of "exploratory analysis" to get a big picture view of the cluster structure of the data, before employing algorithms that are more refined. It should be noted that there is no one-size fits-all clustering algorithm, therefore there are situations where K-means might actually turn out to be the best choice.

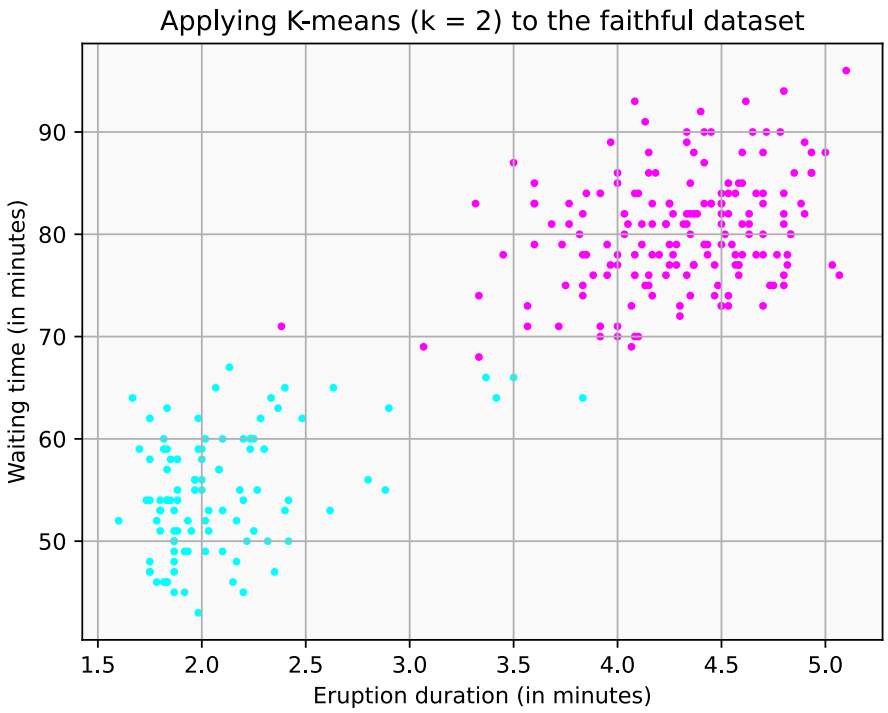


Figure 54: K-means, with the k parameter equal to 2, applied to the `faithful` dataset. This dataset records data related to the eruptions of the Old Faithful geyser in Yellowstone National Park. Each datum has two dimensions: the time for which each eruption lasted and the time elapsed between consequent eruptions. Choosing $k = 2$ properly captures the structure of the dataset, since there are two clear data clouds with few points in between.

2.6.2. Structure of a learning vector quantization network

A **learning vector quantization network** (**LQV** network) is a layered feed-forward neural network with the following characteristics:

- There are only input and output neurons, no hidden neurons: $U_{\text{hidden}} = \emptyset$;
- no input neuron can also be an output neuron, and vice versa: $U_{\text{in}} \cap U_{\text{out}} = \emptyset$;
- The input layer is fully connected to the output layer: $C = U_{\text{in}} \times U_{\text{out}}$;
- The network input function of the output neurons is a distance function;

- The activation function of the output neurons is a radial function;

In some sense, a self-organizing map is similar to a radial basis function network where the output layer and the hidden layer have been merged into one.

The weights coming into the output layer determine the coordinates of a center, from which the distance of the input is measured; in the context of LVQ networks, this center is called **reference vector**. The closer the input to a reference vector, the higher the activation of the corresponding neuron. In general, the σ parameter of all output neurons (the radius of the capture region) is always the same.

A notable difference with the previous neural network models is that the outputs neurons don't operate "in a vacuum", but instead depend upon each other. In particular, the output of the output neurons is computed following a "winner-takes-all" principle: the output neuron with the highest activation has output 1, any other neuron has output 0 (ties are broken at random):

$$f_{\text{out}}^{(u)}(\text{act}_u) = \begin{cases} 1 \text{ if } \text{act}_u = \max_{v \in U_{\text{out}}} \text{act}_v \\ 0 \text{ otherwise} \end{cases}$$

An alternative clustering algorithm is **learning vector quantization (LVQ)**. The idea behind LVQ is the same as k-means clustering: partitioning the space into clusters, each represented by a reference vector (or centroid). The difference between is that in k-means the two updates in each iteration (assigning points to clusters and recomputing the centroids) are performed by considering all points at the same time. In LVQ, each iteration performs the updates one point at a time.

This procedure is referred to as **competitive learning**. For each point, the reference vectors "compete" with the others to become the reference vector representing its cluster. The competition is "won" by the reference vector that is the closest to the point. Since each reference vector is the center of an output neuron, this is equivalent to stating that the competition is "won" by the output neuron with the highest activation. The reference vector/output neuron that won the competition is adapted, moving it slightly closer to the given point.

Let $\boldsymbol{p}$ be the training pattern, $\boldsymbol{r}$ the reference vector that "won" the "competition" for $\boldsymbol{p}$ and $\eta \in (0, 1)$ a learning rate. The adaptation rule is given by:

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + \eta(\boldsymbol{p} - \boldsymbol{r})$$

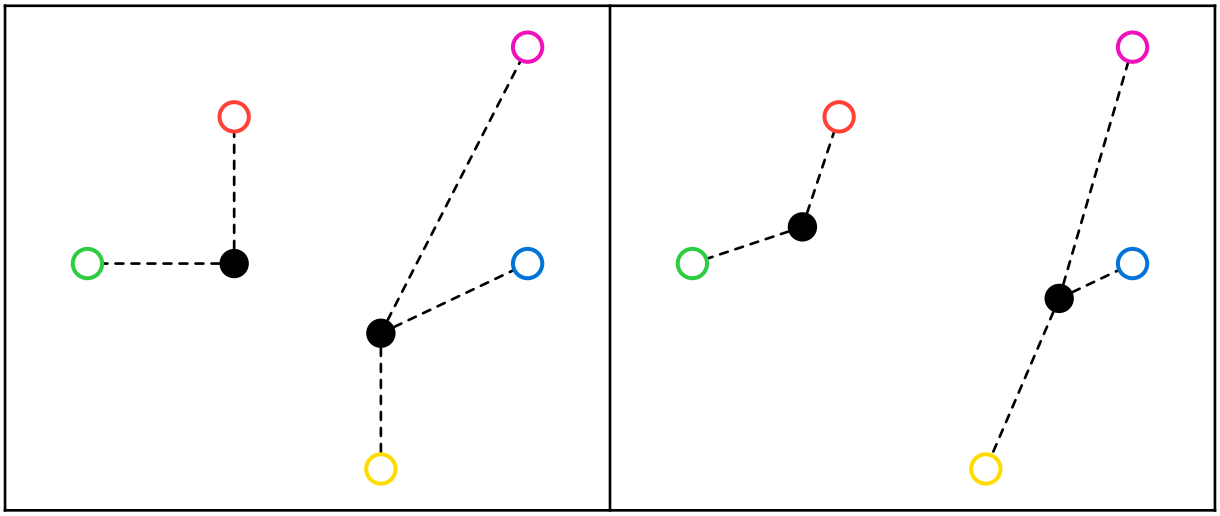


Figure 55: On the left, two reference vector and five training patterns for LVQ. The leftmost reference vector is closer to the green and red points, while the rightmost reference vector is closer to the yellow, purple and blue points. On the right, the two reference vectors have been moved closer to the points in their vicinity.

Learning vector quantization exists in both online learning and batch learning. In online learning, the position of the reference vector is updated immediately after each point is evaluated. In batch learning, the updates are "stashed" and then summed at the end of each training epoch, that is when each point has been evaluated once.

A constant learning rate can prevent, in some situations, the learning process to converge, especially with online learning. For this reason, it is actually preferable to have a learning rate that scales with time, becoming smaller and smaller as the number of iterations increases. For example:

$$\eta(t) = \eta_0 \alpha^t,\ 0 < \alpha < 1 \qquad\qquad \eta(t) = \eta_0 t^\kappa,\ \kappa < 0$$

With η_0 being the fixed initial rate. Even though a time-dependent learning rate improves convergence, attention should be paid to the speed of decrease. If the learning rate decreases too quickly, the step width might be so small that an optimal situation cannot be reached.

Learning vector quantization, as used so far, solved a free learning task (unsupervised learning): finding a pattern in a set of data without a counterexample. However, it is also possible to use learning vector quantization to solve a fixed learning task (supervised learning).

The idea is to assign a label to each training pattern, the class to which it belongs, as the output of the pattern. Phrased as such, learning vector quantization allows to predict, given an unknown point, which label should receive (to which class should belong). This is done by adding at least as many reference vectors as there are labels, with the goal of having each reference vector closer to the points in the training dataset sharing its label.

The learning process functions in the same way as before, the only difference is the adaptation rule. Instead of always updating a reference vector moving it closer to the closest points, two different updates can be performed depending on the label of the reference vector and on the label of the under consideration point. If the point and the closest reference vector have the same label, the reference vector is moved closer (as before). If the point and the closest reference vector have different labels, the reference vector is pushed away. These are referred to as the **attraction rule** and the **repulsion rule**, respectively:

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + \eta(\boldsymbol{p} - \boldsymbol{r}) \qquad\qquad \boldsymbol{r} \leftarrow \boldsymbol{r} - \eta(\boldsymbol{p} - \boldsymbol{r})$$

When the model is asked to predict a new (unlabelled) input, its result is the label of the reference vector closest to the input.

A simple improvement in the training process for fixed learning tasks relies on considering not the reference vector closest to the given point, but the two closest reference vectors. Let $\boldsymbol{r}$ and $\boldsymbol{r}'$ be these two reference vectors, and let $\boldsymbol{p}$ be the point under examination. If $\boldsymbol{r}$ and $\boldsymbol{r}'$ have different labels and $\boldsymbol{r}$ shares its label with $\boldsymbol{p}$, then $\boldsymbol{r}$ is moved closer to $\boldsymbol{p}$ and $\boldsymbol{r}'$ is pushed away from $\boldsymbol{p}$:

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + \eta(\boldsymbol{p} - \boldsymbol{r}) \qquad\qquad \boldsymbol{r}' \leftarrow \boldsymbol{r} - \eta(\boldsymbol{p} - \boldsymbol{r}')$$

If $\boldsymbol{r}$ and $\boldsymbol{r}'$ have different labels and $\boldsymbol{r}'$ shares its label with $\boldsymbol{p}$, then $\boldsymbol{r}$ is moved closer and $\boldsymbol{r}'$ is pushed away:

$$\boldsymbol{r} \leftarrow \boldsymbol{r} - \eta(\boldsymbol{p} - \boldsymbol{r}) \qquad\qquad \boldsymbol{r}' \leftarrow \boldsymbol{r} + \eta(\boldsymbol{p} - \boldsymbol{r}')$$

From empirical observations, it was observed how this version of learning vector quantization tends, in certain situations, to drive the reference vectors further and further apart, instead of having them converging. One solution is to introduce a so-called **window rule** into the adaptation formula: the reference vectors $\boldsymbol{r}$ and $\boldsymbol{r}'$ are adapted only if the data point $\boldsymbol{p}$ lies close to the hyper-surface that separates different clusters (the classification border). This "closeness" is quantified by requiring that:

$$\min\left(\frac{d(\boldsymbol{p},\boldsymbol{r})}{d(\boldsymbol{p},\boldsymbol{r}')},\frac{d(\boldsymbol{p},\boldsymbol{r}')}{d(\boldsymbol{p},\boldsymbol{r})}\right) > \frac{1-\xi}{1+\xi}$$

With ξ being a user-chosen parameter that specifies the "breadth" or "width" of the window adaptation. The idea is that a point lying too close to the classification border is poorly classified, hence it's reasonable to adapt the reference vector. On the other hand, if the point is already far from the border, then there's no need to adapt the reference vector. This way, an indefinite "pushing away" of reference vectors is prevented.

2.7. Self-organizing maps

2.7.1. Structure of a self-organizing map

A **self-organizing map** (SOM) or **Kohonen feature map** is a layered feed-forward neural network with the following characteristics:

- There are only input and output neurons, no hidden neurons: $U_{\text{hidden}} = \emptyset$;
- no input neuron can also be an output neuron, and vice versa: $U_{\text{in}} \cap U_{\text{out}} = \emptyset$;
- The input layer is fully connected to the output layer: $C = U_{\text{in}} \times U_{\text{out}}$;
- The network input function of the output neurons is a distance function;
- The activation function of the output neurons is a radial function;
- The output function of the output neurons is the identity function.

A self-organizing map is a generalization of the LVQ network, where the relationship between output neurons is more elaborate. In particular, the **neighborhood relationship** that exists between the output neurons is described by a distance function:

$$d_{\text{neurons}} : U_{\text{out}} \times U_{\text{out}} \mapsto \mathbb{R}_0^+$$

This is why it is common to represent the output layer of a self-organizing map as a (usually two-dimensional) grid.

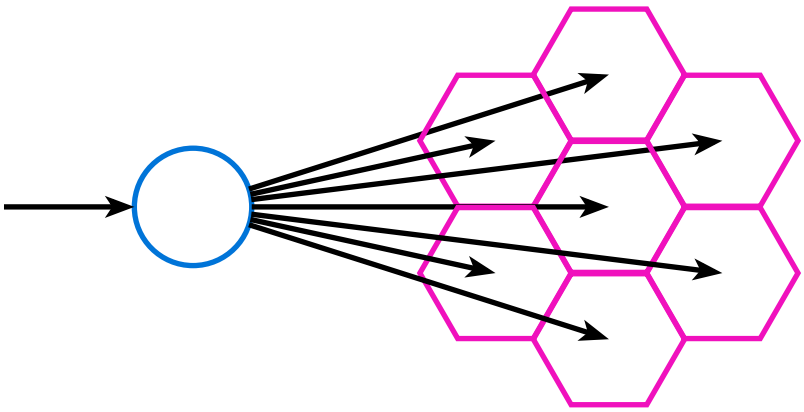


Figure 56: A self-organizing map with one input neuron and seven output neurons. The output layer is arranged into an honeycomb structure.

The goal of having a self-organizing map learn is to have reference vectors that are close to each other in the input space relate to output neurons that are close to each other as well. Even if the distance between the neurons is not identical to the distance between the inputs, the way in which the inputs are "arranged", their topology, is preserved. If this is the case, a self-organizing map describes a **topology preserving map**.

This is particularly enticing because, in general, the dimensionality of the ouputs of a self-organizing map is much lower than the dimensionality of its inputs. This means that it's possible to operate dimensionality reduction on a dataset, giving it a visual representation, while at the same time maintaining the arrangement of the inputs.

3. Fuzzy logic

3.1. Fuzzy sets

Standard set theory assumes that *elements* (facts and objects of the world) belong or do not belong to *sets* (collections of objects). That is, there is a clear and unambiguous rule that asserts whether or not an element is part of a set.

Mathematically, this is represented with a so-called **characteristic function**: let U be the set that, by definition, contains every element of potential interest (also called the **universe set**, or just the **universe**). Let A be a subset of U; the characteristic function $\chi_A : U \mapsto \{0, 1\}$ is defined as:

$$\chi_A(x) = \begin{cases} 1 \text{ if } x \in A \\ 0 \text{ if } x \notin A \end{cases}$$

This means that each set can be identified unambiguously with a function.

Exercise 3.1.1: Consider the set of natural numbers $\mathbb{N}$. Let $\mathbb{N}_{\text{even}} \subseteq \mathbb{N}$ be the set of even natural numbers. What would be its characteristic function?

Solution: Dividing an even integer by 2 gives a remainder of 0, while dividing an odd integer by 2 gives a remainder of 1. Adding 1 to the number gives the opposite result, because the successor of an even number is odd and the successor of an odd number is even. Therefore, one possibility is $\chi_{\mathbb{N}_{\text{even}}}(x) = (x + 1) \bmod 2$, with $x \in \mathbb{N}$. □

Standard set theory and propositional/predicative logic are better suited to model mathematical statements: a triangle *is* a three-sided shape, seven *is not* divisible by three, π *belongs* to $\mathbb{R}$, ecc... This is possible because assertions in math are, indeed, unambiguously true or false. Also, it is assumed that one is given "perfect" and "unbound" knowledge about the statements at hand. Finally, logic and set theory do not rely on natural language, but on a very restricted abstract formalism.

However, this approach is ill-suited to tackle most real-world phenomena and, most importantly, to model human reasoning. Even though there are subsets in real life where a characteristic function can be defined with no ambiguity (the subset of people that are married, the subset of animals that are dogs, ecc...), this is most likely not the case.

Human reasoning has to be carried out through natural language, which is spurious and context-dependent: adverbs such as *somewhat*, *likely*, *almost*, ecc... blur the line between truthness and falsehood. Whether or not an element should belong to a set or not is often not agreed upon, varying from context to context and/or from person to person. Also, being human cognition limited, reasoning has to rely on imperfect knowledge. However, it's still possible to make (imperfect) logical deductions even in the face of partial knowledge, assigning approximate truth values to propositions.

Exercise 3.1.2: Let U be the set of all people. Let X be the set of people that are *tall*. What would be its characteristic function?

Solution: Using classical set theory, the only way to model "tallness" would be through the following indicator function:

$$\chi_{\text{tall}}(h) = \begin{cases} 1 \text{ if } h \geq H \\ 0 \text{ if } h < H \end{cases}$$

Where H is a fixed threshold that discriminates between people that are tall and people that are not tall. However, this would poorly model reality for at least three reasons:

- The choice of H is not agreed upon, and would have to be chosen rather arbitrarely;
- Any person whose height is in the range $[0, H)$ is "just as" short, whereas any person whose height is in the range $[H, +\infty)$ is "just as" tall. This oversimplifies a more nuanced and realistic definition of "tallness";
- Even if H were to be determined unambiguously, it would still make the classification impractical. In general, no one actually measures the height of a person before concluding of whether they are tall or not, mostly relying on instinct instead.

□

These considerations suggest the need for subsets whose membership is spurious. Instead of having a characteristic function whose outputs are either 1 (complete membership) or 0 (no membership), a function whose output is a number that quantifies "how much" an element belongs to a set. That is, introducing a notion of *partial membership* to sets.

More formally, let U be a universe set. A **fuzzy subset** μ of U, or simply a **fuzzy set** μ of U, is a set of pairs defined as:

$$\mu = \{(x, \mu(x)) \mid x \in U\}$$

Where $\mu(x) : U \mapsto [0, 1]$ is a function, called **membership function**, that assigns to each member $x \in U$ a **degree of membership** $\mu(x)$ to the set μ. The set of all fuzzy sets that can be constructed out of a universe U (the "power set" of fuzzy sets) is denoted as $\mathcal{F}(U)$.

Given a certain $x \in U$, if $\mu(x) \in (0, 1)$ it means that x belongs to the fuzzy set "only to some extent": the closer to 1, the "more" it belongs, the closer to 0, the "less" it belongs. In particular, if $\mu(x) = 1$ then x has absolute and unambiguous membership, whereas if $\mu(x) = 0$ then x has no membership at all.

A "standard" set (a set that is not fuzzy, that is) can therefore be seen as a special case of a fuzzy set where its membership function (the characteristic function) is always either 0 or 1. It is customary to use the term **crisp set** to refer to sets that are not fuzzy.

Exercise 3.1.3: What is a possible membership function for the fuzzy set of tall people?

Solution: It is reasonable to consider a person to be unambiguously tall if their height is 2 metres or more, and to consider a person to be unambiguously short if their height is 1.6 metres or less. The issue arises for heights in the $(1.6, 2)$ interval; are people with an height of 1.8 metres tall? The simplest choice would be a straight line:

$$\mu(x) = \begin{cases} 1 & \text{if } x > 2 \\ \frac{5}{2}x - 4 & \text{if } x \leq 1.6 \leq 2 \\ 0 & \text{if } x < 1.6 \end{cases}$$

□

Note that fuzzy sets, even though they better model imprecise knowledge, aren't enough on their own. That is, there's still the need to find a membership function for the fuzzy set that adequately addresses the task at hand. The choice of the membership function is subjective and context dependent, and in general is not an empirical finding. As a matter of fact, the choice of membership function in Exercise 3.1.3 was as good as any. However, the flexibility of a membership function allows one to "tune" it to improve the model and make incremental adjustments.

If the universe set $U = \{x_1, ..., x_n\}$ is a discrete set, to represent a fuzzy set μ it is sufficient to list each member x_i of U together with its degree of membership $\mu(x_i)$. That is, $\mu = \{(x_1, \mu(x_1)), (x_2, \mu(x_2)), ..., (x_n\mu(x_n))\}$.

Another way to represent a fuzzy set, particularly useful if the universe set is continuous, is simply graphing its membership function. This representation is also referred to as the **vertical representation** of the fuzzy set.

Exercise 3.1.4: What would be the vertical representation of Exercise 3.1.3?

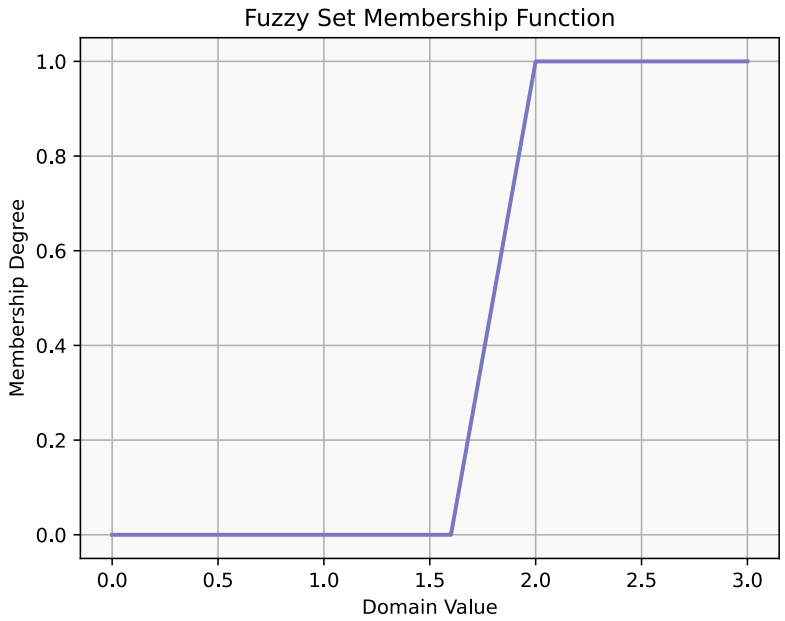


Figure 57: The vertical representation of the fuzzy set.

In general, the explicit expression of a fuzzy set (as a set of pairs) is hardly ever used, because it is cumbersome to manipulate and poorly integrates with crisp sets. Even worse, different sources use different constructions for defining a fuzzy set. Some refer to the membership function as the fuzzy set itself, some define a fuzzy set as a pair (U, μ), with U being the universe and μ being the membership function. The simplest and least ambiguous way to refer and to use a fuzzy set is to refer to its membership function.

Crisp sets A and B can be manipulated with the known set operators of **union**, **intersection** and **complement**:

$$A \cup B = \{x \in U \mid x \in A \vee x \in B\} \qquad \chi_{A \cup B}(x) = \max\{\chi_A(x), \chi_B(x)\}$$

$$A \cap B = \{x \in U \mid x \in A \wedge x \in B\} \qquad \chi_{A \cap B}(x) = \min\{\chi_A(x), \chi_B(x)\}$$

$$\overline{A} = \{x \in U \mid x \notin A\} \qquad \chi_{\overline{A}}(x) = 1 - \chi_A(x)$$

Fuzzy sets $\mathcal{A}$ and $\mathcal{B}$ can be manipulated in the exact same way, since these operators extend naturally. The membership functions of $\mathcal{A} \cup \mathcal{B}$, $\mathcal{A} \cap \mathcal{B}$ and $\overline{\mathcal{A}}$ are given by:

$$\mu_{\mathcal{A}\cup\mathcal{B}}(x) = \max\{\mu_{\mathcal{A}}(x), \mu_{\mathcal{B}}(x)\} \qquad \mu_{\mathcal{A}\cap\mathcal{B}}(x) = \min\{\mu_{\mathcal{A}}(x), \mu_{\mathcal{B}}(x)\} \qquad \mu_{\overline{\mathcal{A}}}(x) = 1 - \mu_{\mathcal{A}}(x)$$

This is just one (the simplest) of the many possible ways to extend set operators to fuzzy sets. Other choices, better suited for different models, will be explored later on.

Out of all continuous fuzzy sets, the main interest lies in **convex fuzzy sets**, that best model natural language. Convex fuzzy sets are fuzzy sets having a degree of membership function that is monotonically increasing up to a certain point and monotonically decreasing after said point. Ironically, the membership function of a convex fuzzy set is a concave function, not a convex function.

Especially when building applications, one should consider convex fuzzy sets whose membership function can be specified uniquely by a few parameters, so that they are easy to tune. One example are **triangular functions**:

$$\Lambda_{a,b,c} : \mathbb{R} \mapsto [0,1] \text{ with } a < b < c \text{ and } a,b,c \in \mathbb{R}$$

$$\Lambda_{a,b,c}(x) = \begin{cases} \frac{x-a}{b-a} & \text{if } a \leq x < b \\ \frac{c-x}{c-b} & \text{if } b \leq x \leq c \\ 0 & \text{otherwise} \end{cases}$$

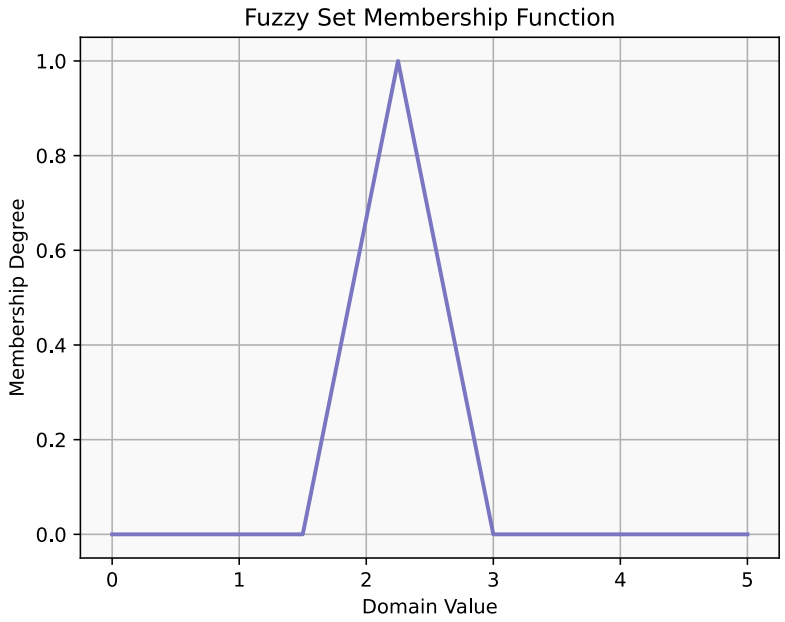


Figure 58: Plot of the triangular function $\Lambda_{1.5,2.25,3}$.

Triangular functions are a special case of **trapezoidal functions**:

$$\Pi_{a,b,c,d} : \mathbb{R} \mapsto [0,1] \text{ with } a < b \leq c < d \wedge a,b,c,d \in \mathbb{R}$$

$a = b = -\infty$ and $c = d = +\infty$ are also valid

$$\Pi_{a,b,c,d}(x) = \begin{cases} \frac{x-a}{b-a} & \text{if } a \leq x < b \\ 1 & \text{if } b \leq x < c \\ \frac{d-x}{d-c} & \text{if } c \leq x \leq d \\ 0 & \text{otherwise} \end{cases}$$

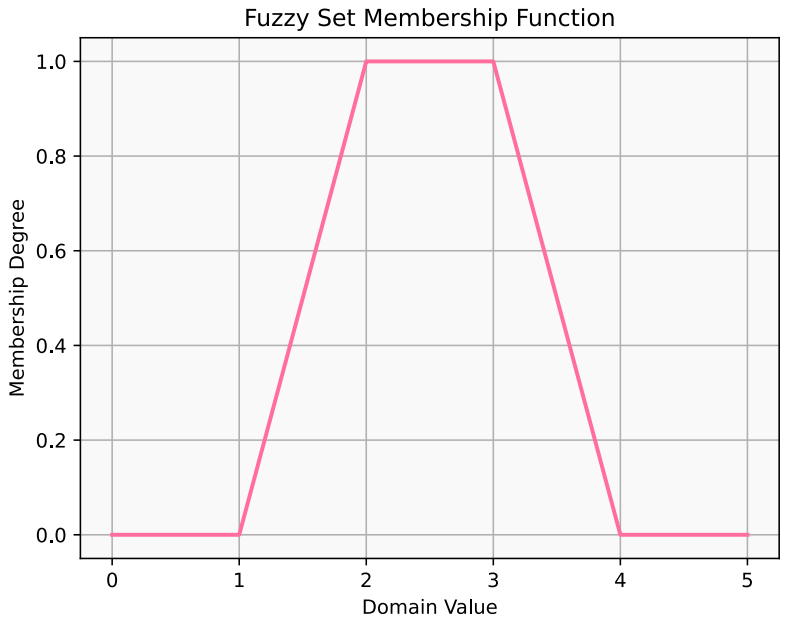


Figure 59: Plot of the trapezoidal function $\Pi_{1,2,3,4}$.

Another class of functions are the **bell-shaped functions**:

$$\Omega_{a,b} : \mathbb{R} \mapsto [0,1] \text{ with } a, b \in \mathbb{R} \qquad \Omega_{a,b}(x) = e^{-\left(\frac{x-a}{b}\right)^2}$$

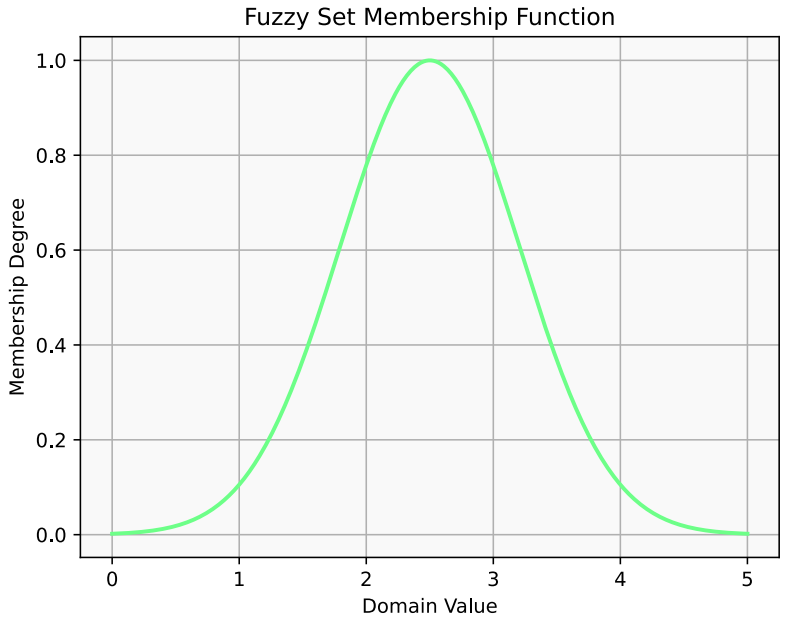


Figure 60: Plot of the bell-shaped function $\Omega_{2.5,1}$.

The **support** of a fuzzy set $\mu \in \mathcal{F}(U)$ is the (crisp) set that contains all the elements of U having non-zero degree of membership in μ:

$$S(\mu) = \{x \in U \mid \mu(x) > 0\}$$

The **core** of a fuzzy set $\mu \in \mathcal{F}(U)$ is the (crisp) set that contains all the elements of U having membership equal to 1:

$$C(\mu) = \{x \in U \mid \mu(x) = 1\}$$

The **height** of a fuzzy set $\mu \in \mathcal{F}(U)$ is the highest degree of membership obtained by any element of said set:

$$h(\mu) = \sup\{x \in U \mid \mu(x)\}$$

Even though the $[0, 1]$ codomain could suggest so, fuzzy membership has little to do with probability theory. That is, the value $\mu(x)$ does not represent the probability that the element $x \in U$ belongs to μ, or that a random sample of U results in an element of μ. Fuzzy sets model how closely a property or a statement is satisfied, whereas probability models the certainty of an event to happen or not[11]. Specifically, fuzzy sets have three main onthological interpretations:

- **Similarity**. A fuzzy set represents the degree of proximity between a an element and another, used to set the scale. Given a reference object that certainly and unambiguously belongs to the fuzzy set, the degree of membership between a given object and a reference object is therefore reduced to the similarity between the two: the greater the similarity, the higher the membership degree. If the similarity can be expressed mathematically, it can be formulated as a distance. Popular with fuzzy clustering and fuzzy control.
- **Preference**. A fuzzy set represents the degree of preference in favour of one object over another, or the feasibility of choosing one over another. Preference can be formulated as a utility function or as a cost function, leading one to choose the member of the fuzzy set with the lowest cost or the highest utility. Popular with fuzzy decision making theory and fuzzy optimization
- **Possibility**. A fuzzy set represents how reasonable is for an event to happen based on the current knowledge state, ranging from completely implausible ($\mu(x) = 0$) to completely reasonable ($\mu(x) = 1$). Note the difference between this formulation and probability theory: $\mu(x) = 0$ does not mean that x will never happen, as $\mu(x) = 1$ does not mean that x will always happen. What they really represent is the degree of "surprise" if they were to happen. Popular with fuzzy artificial intelligence.

Exercise 3.1.5 clarifies the difference between probability and fuzzy set membership.

Exercise 3.1.5: Let B be the set of all bottles of 1 litre capacity. Consider the two following statements:

- Choosing a random bottle out of B, the probability of finding a bottle of water is 0.999, while the probability of finding a bottle of (liquified) caffeine is 0.001;
- The degree of membership of a bottle in B with respect to the fuzzy set of water bottles is 0.999, while the degree of membership with respect to the fuzzy set of (liquified) caffeine bottles is 0.001.

Supposing that one bottle has to be drunk, and that one of the two statements must be true. Which one would be the safest?

Solution: Having a probability of 0.001 of being a bottle of caffeine could be interpreted as, out of 1000 identically sampled bottles, roughly 1 contains only and exclusively caffeine (and roughly 999 contain only water). Having a degree of membership of 0.001 with respect to the fuzzy set of caffeine bottles can be interpreted as a bottle in B matching the definition "this bottle contains only caffeine" with a degree of 0.001. That is, a bottle in B would be 0.001 litres (one gram) of caffeine and 0.999 litres of water. Knowing that the lethal dose of caffeine for an adult individual is about ten grams[12], the safest choice would be the second. This is because in the first case

[11] It should also be noted that $\mu(x)$ does not necessarely comply with the three Kolmogorov axioms for probability. For example, $\int_{-\infty}^{+\infty} \mu(x)dx$ might not be equal to 1.

[12] https://www.fda.gov/consumers/consumer-updates/spilling-beans-how-much-caffeine-too-much

there's a 0.1% chance of drinking one litre of caffeine (far more than ten grams) while in the second case there's a 100% chance of drinking one gram of caffeine. That is, one chance out of a thousand of dying against no chance at all. □

3.2. α-cuts

The vertical representation of fuzzy sets presents some issues, mainly that storing fuzzy sets encoded as functions in a computer would be both impractical and inefficient. Also, manipulating fuzzy sets (computing intersections, unions, ecc...) becomes cumbersome. For these reasons, the vertical representation should be limited to illustration purposes.

Given a universe set U and a fuzzy set $\mu \in \mathcal{F}(U)$, let α be any number between 0 and 1. The subset $[\mu]_\alpha = \{x \in U \mid \mu(x) \geq \alpha\}$ is referred to as the **α-level set** of μ, or **α-cut** of μ. That is, the α-cut of a fuzzy set is the subset that contains all the elements having degree of membership greater or equal than α. By definition, $[\mu]_0$ is just the entire universe.

In a similar fashion, the subset $[\mu]_{\underline{\alpha}} = \{x \in U \mid \mu(x) > \alpha\}$ is referred to as the **strict α-level set** of μ, or **strict α-cut** of μ.

α-cuts are used in the **horizontal representation** of fuzzy sets, that consists in picking a subset of α -cuts and drawing only those, instead of the entire function. Each α-cut is represented as a straight line of height $[\mu]_\alpha$.

Exercise 3.2.1: What would be the horizontal representation of the fuzzy set with the following membership function?

$$\mu(x) = \begin{cases} x - 1 & \text{if } 1 \leq x < 2 \\ -\frac{3}{8}x + \frac{7}{4} & \text{if } 2 \leq x < 4 \\ \frac{1}{8}x - \frac{1}{4} & \text{if } 4 \leq x < 6 \\ \frac{1}{2} & \text{if } 6 \leq x < 7 \\ -\frac{1}{2}x + 4 & \text{if } 7 \leq x < 8 \\ 0 & \text{otherwise} \end{cases}$$

As α use $0, 0.15, 0.30, 0.45, 0.60, 0.75, 0.90$.

Solution:

Figure 61: The α-cuts of the given fuzzy set. The membership function is drawn as a dashed line.

□

α-cuts of convex fuzzy sets are peculiar because they are always convex sets. On the other hand, α-cuts of non-convex fuzzy sets may be a union of more than one disjointed interval. It is also possible to use this property as a definition: a fuzzy set is convex if and only if all of its α-cuts are convex sets.

α-cuts can uniquely identify fuzzy sets. By definition, the degree of membership $\mu(x)$ of an element x is the highest possible value of α giving an α-cut that contains x:

$$\mu(x) = \sup\{\alpha \in [0,1] \mid x \in [\mu]_\alpha\}$$

Of course, actually storing all possible α-cuts of a fuzzy set would be no improvement over storing the expression of the entire membership function. However, discretizing the set of all α-cuts into a subset $\left\{[\mu]_{\alpha_1}, [\mu]_{\alpha_2}, ..., [\mu]_{\alpha_k}\right\}$ is sufficient to approximate the entire fuzzy set:

$$\mu(x) \approx \max\{\alpha \in \{\alpha_1, \alpha_2, ..., \alpha_k\} \mid x \in [\mu]_\alpha\}$$

Formally speaking, given a certain number of α-cuts, the original membership function can be reconstructed by taking the *upper envelope* of said cuts. Both the number of α values and which values to choose are decided arbitrarely (more values, better precision). A common choice is $0.25, 0.50, 0.75, 1$.

α-cuts can be stored in real memory in the form of linked lists. Each disjointed interval of each α-cut is stored in a separate node of the list, and the nodes are linked together in ascending order. Each list (each α-cut) has also a pointer to following list (following α-cut).

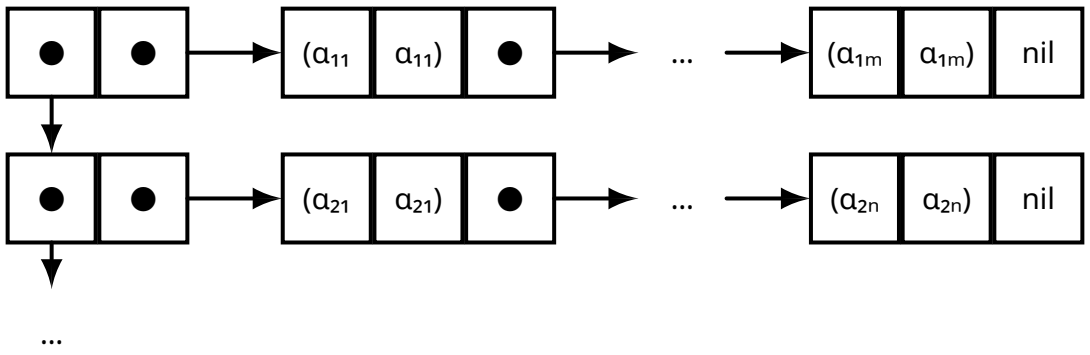


Figure 62: Linked list representation of α-cuts

Theorem 3.2.1 (Representation theorem): Given $\mu \in \mathcal{F}(U)$ a fuzzy set over a universe set U:

$$[\mu]_0 = \sup_{\alpha \in [0,1]} \left\{\min\left(\alpha, \chi_{[\mu]_\alpha}(x)\right)\right\}, \quad \text{where } \chi_{[\mu]_\alpha}(x) = \begin{cases} 1 \text{ if } x \in [\mu]_\alpha \\ 0 \text{ otherwise} \end{cases}$$

Lemma 3.2.1: Let $\mu \in \mathcal{F}(U)$ be a fuzzy set over a universe set U, and let $\alpha, \beta \in [0,1]$. If $\alpha < \beta$, then $[\mu]_\alpha \supseteq [\mu]_\beta$.

Lemma 3.2.2: Let $\mu \in \mathcal{F}(U)$ be a fuzzy set over a universe set U. For any $\alpha, \beta \in [0,1]$, $\bigcap_{\alpha:\alpha<\beta} [\mu]_\alpha = [\mu]_\beta$.

3.3. Relevant classes of fuzzy sets

Some classes of fuzzy sets are more important than others. For example:

- A fuzzy set $\mu \in \mathcal{F}(X)$ is said to be **normal** if and only if its height is equal to 1. The set of all normal fuzzy sets is given by:

  $$\mathcal{F}_N(X) = \{\mu \in \mathcal{F}(X) \mid \exists x \in X : \mu(x) = 1\}$$

  A fuzzy set that is not normal is said to be **subnormal**. Subnormal fuzzy sets possess no members having complete set membership;
- A fuzzy set $\mu \in \mathcal{F}(X)$ is called a **fuzzy number** if μ is normal and $[\mu]_\alpha$ is bounded, closed, and convex $\forall \alpha \in (0, 1]$. They are used to represent values that are "somewhat close" to a given number;
- A fuzzy set $\mu \in \mathcal{F}(X)$ is said to be **upper semi-continuous** if it's normal and all of its α-cuts are compact intervals. The set of all upper semi-continuous fuzzy sets is given by:

  $$\mathcal{F}_C(X) = \{\mu \in \mathcal{F}_N(X) \mid [\mu(x)]_\alpha \text{ is compact } \forall \alpha \in (0, 1]\}$$

  The definition recalls the one of upper semi-continuous functions. A function f is upper semi-continuous at point x_0 if and only if:

  $$\lim_{x \to x_0} \sup f(x) \leq f(x_0)$$

  That is, if values near to x_0 are either close to $f(x_0)$ or smaller than $f(x_0)$;
- A fuzzy set $\mu \in \mathcal{F}(X)$ is said to be a **fuzzy interval** if it's normal and, for any $a, b, c \in X$ such that $c \in [a, b]$, $\mu(c)$ is bigger than the minimum between $\mu(a)$ and $\mu(b)$. The set of all fuzzy intervals is given by:

  $$\mathcal{F}_I(X) = \{\mu \in \mathcal{F}_N(X) \mid \mu(c) \geq \min\{\mu(a), \mu(b)\} \forall a, b, c \in X : c \in [a, b]\}$$

  The definition implies that such sets are also convex and that their core is a classical interval. They are used to represent intervals that are "somewhat close" to a given range.

The concept of fuzzy number plays fundamental role in formulating **quantitative fuzzy variables**: those are (mathematical) variables whose possible states are fuzzy numbers. In particular, fuzzy variables that represent linguistic concepts (*small*, *tall*, *hot*, ecc...) are also referred to as **linguistic variables**.

A linguistic variable is a mathematical variable defined in terms of a base variable, which is a variable in classical sense (temperature, pressure, age, ec...) but whose possible values are fuzzy (about 10 degrees, roughly 20 years, ecc...), also called **linguistic terms**. More formally, a linguistic variable is defined by a tuple (ν, T, X, g, m):

- ν is the name of the variable;
- T is the set of linguistic terms of ν, the set of possible fuzzy numbers for ν;
- X is the base set, assumed in general to be a subset of real numbers. Those are the possible actual values of T (and of ν);
- g is the grammar (the syntactic rules) that generates the linguistic terms;
- m is the set of semantic rules that assigns a meaning to each linguistic term.

Exercise 3.3.1: Consider the following vague concepts: *freezing*, *cold*, *mild*, *warm*, *hot*. Suppose that such concepts can be defined by the following range of temperatures, in order: $[4, 10]$, $[9, 18]$, $[15, 27]$, $[22, 34]$, $[32, 38]$. Represent each with a fuzzy number.

Solution:

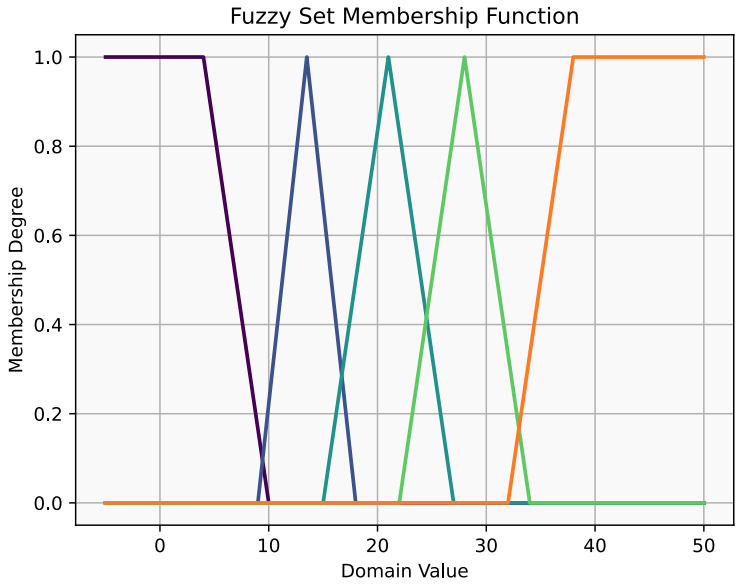


Figure 63: The five temperature ranges represented as fuzzy numbers.

□

3.4. Fuzzy logic

Propositional logic deals with statements that can have only two possible truth values: true (1) or false (0). Given a statement α, its truth value is denoted by $[\![\alpha]\!]$.

Propositions are combined with each other using logical connectives. The most fundamental connectives are **conjuction** (denoted as $\wedge$), **disjunction** (denoted as $\vee$), **implication** (denoted as $\rightarrow$) and **negation** (denoted as $\neg$). The first three are binary operators, the last one is unary. The set $\{\wedge, \vee, \rightarrow, \neg\}$ is sufficient to construct all possible composite propositions.

Let $u_\wedge, u_\vee, u_\rightarrow, u_\neg$ be the functions that have truth values as input and the truth value of applying the respective logical connective as output. the first three functions map the set $\{0,1\}^2$ to $\{0,1\}$, the last one maps the set $\{0,1\}$ to $\{0,1\}$:

$$u_\wedge([\![\alpha]\!],[\![\beta]\!]) : \{0,1\}^2 \mapsto \{0,1\} = [\![\alpha \wedge \beta]\!] \qquad u_\vee([\![\alpha]\!],[\![\beta]\!]) : \{0,1\}^2 \mapsto \{0,1\} = [\![\alpha \vee \beta]\!]$$
$$u_\rightarrow([\![\alpha]\!],[\![\beta]\!]) : \{0,1\}^2 \mapsto \{0,1\} = [\![\alpha \rightarrow \beta]\!] \qquad u_\neg([\![\alpha]\!]) : \{0,1\} \mapsto \{0,1\} = [\![\neg\alpha]\!]$$

Since all four functions operate on discrete inputs, they can be computed simply by enumeration. This is done with the help of **truth tables**:

$$\left[\begin{array}{cc|c} [\![\alpha]\!] & [\![\beta]\!] & [\![\alpha \wedge \beta]\!] \\ \hline 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{array}\right] \qquad \left[\begin{array}{cc|c} [\![\alpha]\!] & [\![\beta]\!] & [\![\alpha \vee \beta]\!] \\ \hline 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{array}\right] \qquad \left[\begin{array}{cc|c} [\![\alpha]\!] & [\![\beta]\!] & [\![\alpha \rightarrow \beta]\!] \\ \hline 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{array}\right] \qquad \left[\begin{array}{c|c} [\![\alpha]\!] & [\![\neg\alpha]\!] \\ \hline 0 & 1 \\ 1 & 0 \end{array}\right]$$

Set-belonging is readily translated into formal logic, where statements are either (absolutely) true or (absolutely) false, with nothing in between. Given a universe U, the proposition "x belongs to A" with $x \in U$ and $A \subseteq U$ is true if $\chi_A(x) = 1$, and false if $\chi_A(x) = 0$. That is, $\chi_A(x)$ is the *truth value* of the proposition "x belongs to A".

The problem that arose in using set theory when modelling spurious memberships manifests itself in logic as well. That is, propositional logic asserts that propositions ought to be either unambiguously true or unambiguously false, but for most real-world propositions a more nuanced notion of truth is necessary. For example, **ternary logic** is a form of logic where statements can either be true, false or *undefined* (neither true nor false).

In general, logics where there are more than two truth values are called **multi-valued logics**. The multi-valued logic of interest here is **fuzzy logic**, where the possible truth values are all real numbers in the interval $[0,1]$. That is, the closer a truth value is to 1 "the more true" the statement is, the closer a truth value is to 0 "the more false" the statement is. If the truth value of a statement is 1, then the statement is unambiguously true, if the truth value is 0 then it is unambiguously false.

Fuzzy propositions, on their own, are insufficient to build a logic. It is also necessary to extend logical operators of "standard" (non fuzzy) logic to fuzzy logic. More specifically, it is necessary to extend the way truth values are assigned in non fuzzy logic to fuzzy logic. Let $w_\wedge, w_\vee, w_\rightarrow, w_\neg$ be the functions that have fuzzy propositions as input and the truth value of applying the respective logical connective as output. the first three functions map the interval $[0,1]^2$ to $[0,1]$, the last one maps the interval $[0,1]$ to $[0,1]$:

$$w_\wedge([\![\alpha]\!],[\![\beta]\!]) : [0,1]^2 \mapsto [0,1] = [\![\alpha \wedge \beta]\!] \qquad w_\vee([\![\alpha]\!],[\![\beta]\!]) : [0,1]^2 \mapsto [0,1] = [\![\alpha \vee \beta]\!]$$
$$w_\rightarrow([\![\alpha]\!],[\![\beta]\!]) : [0,1]^2 \mapsto [0,1] = [\![\alpha \rightarrow \beta]\!] \qquad w_\neg([\![\alpha]\!]) : [0,1] \mapsto [0,1] = [\![\neg\alpha]\!]$$

The simplest way to extend the truth function of logical operators to fuzzy logic follows the same lines as extending set operators to fuzzy sets. Observe how, for any pair of propositions α and β, in propositional/predicative logic the truth value of $\alpha \wedge \beta$ is just the minimum between the truth value of α and β. Similarly, $[\![\alpha \wedge \beta]\!]$ is just the minimum between $[\![\alpha]\!]$ and $[\![\beta]\!]$. Finally, the truth value of the complement of a proposition is just 1 minus said value. This gives:

$$w_{\wedge}([\![\alpha]\!], [\![\beta]\!]) = u_{\wedge}([\![\alpha]\!], [\![\beta]\!]) = [\![\alpha \wedge \beta]\!] = \min\{[\![\alpha]\!], [\![\beta]\!]\}$$
$$w_{\vee}([\![\alpha]\!], [\![\beta]\!]) = u_{\vee}([\![\alpha]\!], [\![\beta]\!]) = [\![\alpha \vee \beta]\!] = \max\{[\![\alpha]\!], [\![\beta]\!]\}$$
$$w_{\neg}([\![\alpha]\!]) = u_{\neg}([\![\alpha]\!]) = [\![\neg\alpha]\!] = 1 - [\![\alpha]\!]$$

This is a reasonable choice, but not the only choice. Different choices for the truth value functions are better suited for different scenarios. To discover other candidate functions, the best approach is to lay out which requirements such functions should possess and restrict the exploration to those. In other words, the idea is to determine which properties the truth value of the classical logical operators possess and which of said properties are worth preserving.

A requirement that should be transversal to all truth functions is that they ensure compatibility between fuzzy logical operators and classical logical operators. That is, "fuzzy" conjunction, "fuzzy" disjunction, "fuzzy" implication and "fuzzy" negation, when given (exactly) 0 or (exactly) 1 as input, should return the same outputs as "classical" conjunction, "classical" disjunction, "classical" implication and "classical" negation, respectively.

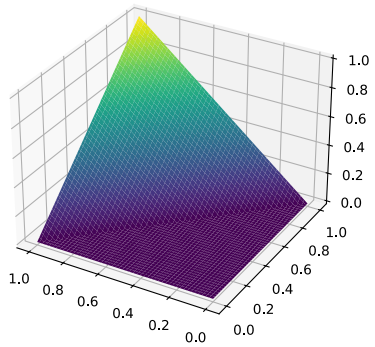
First, consider possible candidates for implementing the truth function for fuzzy conjunction. Any function $t : [0, 1]^2 \mapsto [0, 1]$ is said to be a t-**norm**, or **triangular norm**, if it possesses the following properties:

- **Commutativity**: for any x, y, $t(x, y) = t(y, x)$;
- **Associativity**: for any x, y, z, $t(t(x, y), z) = t(x, t(y, z))$;
- **Monotonicity**: for any x, y, z, if $y \leq z$ then $t(x, y) \leq t(x, z)$;
- **Boundedness**: for any x, $t(x, 1) = x$.

It is advisable to choose a t-norm as the truth function for the logical conjunction, since these properties are those that any reasonable implementation should possess. Indeed, $\min\{[\![\alpha]\!], [\![\beta]\!]\}$, the truth function of non-fuzzy conjunction, possesses all these four properties, which made it the first-class choice for the truth function for fuzzy conjunction. Other examples of t-norms are the:

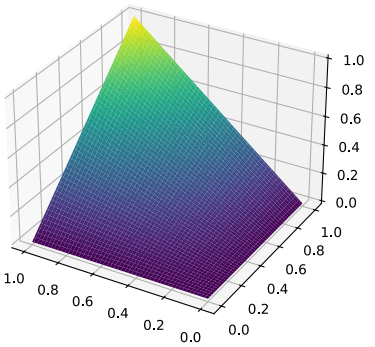
Łukasiewicz t-norm:

$$t(x, y) = \max\{x + y - 1, 0\}$$



Algebraic product:

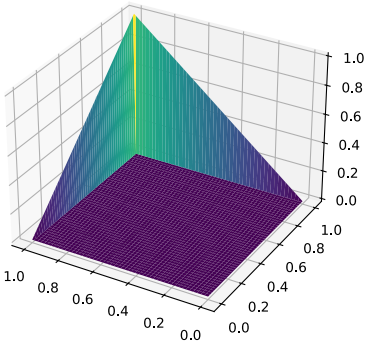
$$t(x, y) = x \cdot y$$

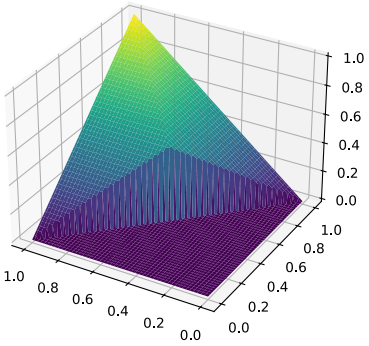


Drastic product:

Nilpotent minimum:

$$t(x,y) = \begin{cases} 0 & \text{if } 1 \notin \{x,y\} \\ \min\{x,y\} & \text{otherwise} \end{cases} \qquad t(x,y) = \begin{cases} \min\{x,y\} & \text{if } x+y > 1 \\ 0 & \text{otherwise} \end{cases}$$





Also, from the boundedness property, it follows that $t(1,1) = 1$ and $t(0,1) = 0$ for any t-norm. Applying the commutative property to $t(0,1) = 0$ one obtains $t(1,0) = 0$. Applying the monotonic property to $t(0,1) = 0$ gives $t(0,0) = 0$. Therefore, any t-norm behaves in the exact same way as the truth function of the logical conjunction when giving 0 and/or 1 as input.

The family of t-norms is very broad. $\min\{x,y\}$ stands out among the other t-norms because it is **idempotent**, meaning that $t(x,x) = x$ for all $x \in [0,1]$. This might seem a mandatory requirement, but it is not. There are scenarios where idempotency is actually undesireable.

A function $s : [0,1]^2 \mapsto [0,1]$ is said to be a t-**conorm**, or **triangular conorm**, if it possesses the first three properties of a t-norm (commutativity, associativity, monotonicity) and, for any x, $s(x,0) = x$. Similarly to how it was done for the t-norm, it is advisable to choose a t-conorm as a logical disjunction, and the truth function $\max\{[\![\alpha]\!], [\![\beta]\!]\}$ for the non-fuzzy disjunction is indeed a t-conorm.

t-norms and t-conorms possess a form of duality: from any t-norm t it is possible to induce a dual t-conorm s, and from any t-conorm s it is possible to induce a dual t-norm t. This is done as follows:

$$s(x,y) = 1 - t(1-x, 1-y) \qquad t(x,y) = 1 - s(1-x, 1-y)$$

These relations are a generalization of the **De Morgan's Laws** for classical logic:

$$[\![\alpha \vee \beta]\!] = [\![\neg(\neg\alpha \wedge \neg\beta)]\!] \qquad [\![\alpha \wedge \beta]\!] = [\![\neg(\neg\alpha \vee \neg\beta)]\!]$$

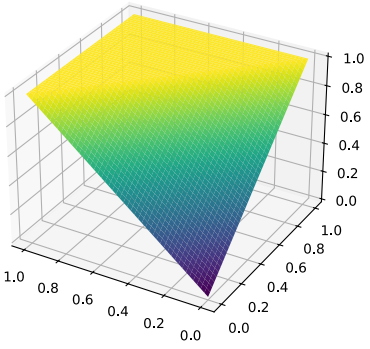
Applying this duality relation to $\max\{x,y\}$ gives $\min\{x,y\}$. This means that $\max\{[\![\alpha]\!], [\![\beta]\!]\}$, the truth function of the non-fuzzy conjunction, and $\min\{[\![\alpha]\!], [\![\beta]\!]\}$, the truth function of the non-fuzzy disjunction, are each other's dual. Applying the relation to the Łukasiewicz t-norm, the algebraic product, the drastic product and the nilpotent minimum respectively, one obtains the following conorms:

Łukasiewicz t-conorm:

$$s(x,y) = \max\{x+y, 1\}$$

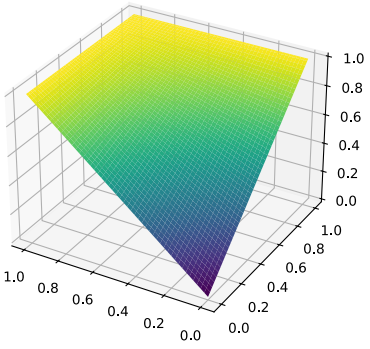
Algebraic sum:

$$s(x,y) = x + y - x \cdot y$$



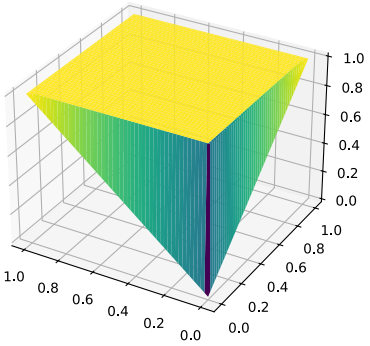
Drastic sum:

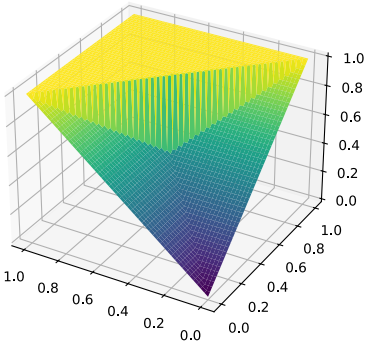
$$s(x,y) = \begin{cases} 1 & \text{if } 0 \notin \{x,y\} \\ \max\{x,y\} & \text{otherwise} \end{cases}$$



Nilpotent maximum:

$$t(x,y) = \begin{cases} \max\{x,y\} & \text{if } x+y<1 \\ 1 & \text{otherwise} \end{cases}$$





Analogously to $\min\{x,y\}$, $\max\{x,y\}$ is the only t-conorm that is idempotent. Also, $\min\{x,y\}$ and $\max\{x,y\}$ are the only t-norm and t-conorm that are the dual of each other possessing the distributive property.

In addition to the connection between t-norms and t-conorms, there exist a connection between t-norms and implications. A continuous t-norm t induces a **residuated implication** as:

$$\vec{t}(x,y) = \sup\{z \in [0,1] \mid t(x,z) \leq y\}$$

Substituting the Łukasiewicz t-norm in the aforementioned formula gives the **Łukasiewicz implication** (left), while substituting the minimum gives the **Gödel implication** (right):

$$x \to y = \min\{1-x+y, 1\}$$

$$x \to y = \begin{cases} 1 \text{ if } x \leq y \\ 0 \text{ otherwise} \end{cases}$$

From the residuated implication one can construct the **biimplication** $\vec{t}$ as:

$$\overleftrightarrow{t} = \vec{t}(\max\{x,y\}, \min\{x,y\}) = t\big(\vec{t}(x,y), \vec{t}(y,x)\big) = \min\big\{\vec{t}(x,y), \vec{t}(y,x)\big\}$$

This formula is motivated from the fact that, in classical logic, $[\![\alpha \leftrightarrow \beta]\!]$ is just $[\![\alpha \to \beta \wedge \beta \to \alpha]\!]$.

The remaining element whose truth function are to be generalized are the **universal quantifier** $\forall$ (forall) and the **existential quantifier** $\exists$ (exists). To perform this generalization, it is possible to exploit the link between $\forall$ and t-norms and between $\exists$ and t-conorms.

Consider a universe U and a predicate $P(x)$; the statement "for any element of U, P holds" is equivalent to the statement "P holds for the first element of U *and* for the second element of U *and* for the third element of U ecc...". More formally:

$$[\![\forall x \in U : P(x)]\!] \equiv [\![P(x_1) \land P(x_2) \land P(x_3) \land ...]\!]$$

Using, for example, min as t-norm gives:

$$[\![\forall x \in U : P(x)]\!] = \min\{[\![P(x_1)]\!], [\![P(x_2)]\!], [\![P(x_3)]\!], ...\} = \min\{[\![P(x)]\!] \mid x \in U\}$$

In the more general case of a universe U that is uncountably infinite:

$$[\![\forall x \in U : P(x)]\!] = \inf\{[\![P(x)]\!] \mid x \in U\}$$

In a similar fashion, the statement "there's at least an element of U for which P holds" is equivalent to the statement "P holds for the first element of U *or* for the second element of U *or* for the third element of U ecc...":

$$[\![\exists x \in U : P(x)]\!] \equiv [\![P(x_1) \lor P(x_2) \lor P(x_3) \lor ...]\!]$$

Using, max as t-conorm gives:

$$[\![\exists x \in U : P(x)]\!] = \sup\{[\![P(x)]\!] \mid x \in U\}$$

Even though min and max could technically be replaced with any other t-norm and t-conorm respectively, this is rarely the case. This is because, in general, t-norms and t-conorms are not idempotent, which easily leads to "degenerate" truth values in the case of uncountably infinite universes.

3.5. Extending set operators to fuzzy sets

As the name suggests, there exist a relationship between fuzzy sets and fuzzy logic. If the degree of membership to a fuzzy set describes "how much" an element possesses a certain property, the truth value of a fuzzy proposition describes "how truthful" it would be to classifiy said element as a member of the set.

In other words, given an element $x \in U$ and a fuzzy set $\mu \in \mathcal{F}(U)$, the degree of membership $\mu(x)$ corresponds to the truth value of the fuzzy proposition "x belongs to μ". That is, $\mu(x) = [\![x \in \mu]\!]$. Note that the notation $x \in \mu$ is somewhat misleading, since it should not be interpreted as set membership in the standard sense. It's just a proposition like any other.

Exercise 3.5.1: Refer to the fuzzy set and membership function of Exercise 3.1.3. Let $P(x)$ be the statement "x is tall" and let $\mu(x)$ be the truth value of said statement, that is $\mu(x) = [\![P(x)]\!]$. What would be the truth value of the statement "people whose height is between 1.70 and 1.80 metres are tall"?

Solution: The statement can be written as:

$$[\![\forall x \in [1.70, 1.80] : P(x)]\!] = \inf\{[\![P(x)]\!] \mid x \in [1.70, 1.80]\}$$

Substituting $[\![P(x)]\!]$ with $\mu(x)$:

$$\inf\{[\![P(x)]\!] \mid x \in [1.70, 1.80]\} = \inf\{\mu(x) \mid x \in [1.70, 1.80]\} = \mu(1.70) = 0.25$$

Since $\mu(x)$ is monotonically non-decreasing. □

The link between fuzzy sets and fuzzy logic can shed light on why intersection, union and complement of fuzzy sets were defined the way they were. In general, it can provide a framework to extend many more concepts of classical set theory, like mappings and the cartesian product, to fuzzy sets.

3.5.1. Intersection

Consider the classical intersection between two sets M_1 and M_2: an element x belongs to $M_1 \cap M_2$ if and only if it belongs to both M_1 and M_2 at the same time. It does not depend on the membership of any other element $y \neq x$ to M_1 or M_2.

In the case of fuzzy sets, it is reasonable to assume the same. That is, $(\mu_1 \cap \mu_2)(x)$, the degree of membership of an element x with respect the intersection between the fuzzy sets μ_1 and μ_2, should only depend on $\mu_1(x)$ and $\mu_2(x)$, the degrees of membership of x with respect to the two sets taken separately.

The proposition of interest is "x belongs to μ_1 *and* x belongs to μ_2", which is a conjunction of two atomic propositions. Since the truth function of a conjunction is best modeled by t-norms, let t be the t-norm of choice. The intersection of two fuzzy sets μ_1 and μ_2 is the set $\mu_1 \cap \mu_2$, whose membership function is defined as:

$$(\mu_1 \cap_t \mu_2)(x) = t(\mu_1(x), \mu_2(x))$$

Where the pedix t specifies that the explicit expression of $(\mu_1 \cap \mu_2)(x)$ depends on which t-norm has been chosen. Since $\mu(x) = [\![x \in \mu]\!]$, the truth value of the fuzzy proposition "x belongs to μ_1 *and* x belongs to μ_2" is given by:

$$[\![x \in (\mu_1 \cap_t \mu_2)]\!] = [\![x \in \mu_1 \wedge x \in \mu_2]\!]$$

Choosing for example min as the t-norm t, one obtains:

$$(\mu_1 \cap \mu_2)(x) = \min\{\mu_1(x), \mu_2(x)\}$$

This is the default choice, unless stated otherwise.

Employing a t-norm in the definition of the intersection between fuzzy sets implies that fuzzy set intersection inherits the four properties of a t-norm:

- Crisp set intersection is commutative, so is fuzzy set intersection;
- Crisp set intersection is associative, so is fuzzy set intersection;
- Given three crisp sets A, B, C, if $A \subseteq B$ then $(A \cap C) \subseteq (B \cap C)$. This is mirrored in the monotonicity property;
- The intersection of a fuzzy set with (the characteristic function of) an crisp set results in the original fuzzy set limited to the crisp set with which it was intersected. More formally, if $M \subseteq U$ is a crisp subset of U and $\mu \in \mathcal{F}(U)$ is a fuzzy set of U:

$$(\mu \cap \chi_M)(x) = \begin{cases} \mu(x) & \text{if } x \in M \\ 0 & \text{otherwise} \end{cases}$$

This is relevant because these properties are the properties of (crisp) set intersection that are worth preserving.

Exercise 3.5.1.1: Consider the fuzzy set in Exercise 3.1.3, labeled $\mu_1(x)$, and the fuzzy set μ_2:

$$\mu_1(x) = \begin{cases} 1 & \text{if } x > 2 \\ \frac{5}{2}x - 4 & \text{if } x \leq 1.6 \leq 2 \\ 0 & \text{if } x < 1.6 \end{cases} \qquad \mu_2(x) = \begin{cases} 20x - 35 & \text{if } 1.75 \leq x \leq 1.80 \\ 1 & \text{if } 1.80 \leq x \leq 1.90 \\ 39 - 20x & \text{if } 1.90 \leq x \leq 1.95 \\ 0 & \text{otherwise} \end{cases}$$

What would be their intersection? Assume that the t norm is min

Solution:

- In the range $(-\infty, 1.60)$, both membership functions are equal to 0, hence the minimum is 0;
- In the range $[1.60, 1.75)$, $\mu_1(x)$ is equal to $\frac{5}{2}x - 4$, while $\mu_2(x)$ is equal to 0. Since $\mu_1(x)$ is always positive, the minimum is 0;
- In the range $[1.75, 1.80)$, $\mu_1(x)$ is equal to $\frac{5}{2}x - 4$, while $\mu_2(x)$ is equal to $20x - 35$. Up to $62/35$ (about 1.7714) $\mu_2(x)$ is smaller than $\mu_1(x)$, larger on the other side;
- In the range $[1.80, 1.90)$, $\mu_1(x)$ is equal to $\frac{5}{2}x - 4$, while $\mu_2(x)$ is equal to 1. The minimum is $\frac{5}{2}x - 4$;
- In the range $[1.90, 1.95)$, $\mu_1(x)$ is equal to $\frac{5}{2}x - 4$, while $\mu_2(x)$ is equal to $39x - 20$. Up to $86/45$ (about 1.9111) $\mu_1(x)$ is smaller than $\mu_2(x)$, larger on the other side;
- In the range $[1.95, 2)$, $\mu_1(x)$ is equal to $\frac{5}{2}x - 4$, while $\mu_2(x)$ is equal to 0. The minimum is 0;
- In the range $[2, +\infty)$, $\mu_1(x)$ is equal to 1, while $\mu_2(x)$ is equal to 0. The minimum is 0.

Grouping the results into a membership function:

$$(\mu_1 \cap \mu_2)(x) = \begin{cases} 20x - 35 & \text{if } 1.75 \leq x < 1.7714 \\ \frac{5}{2}x - 4 & \text{if } 1.7714 \leq x < 1.9111 \\ 39x - 20 & \text{if } 1.9111 \leq x < 1.95 \\ 0 & \text{otherwise} \end{cases}$$

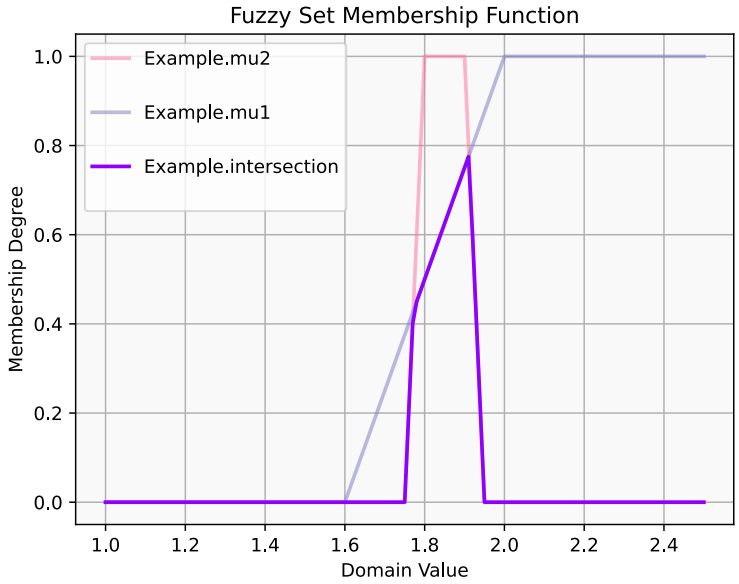


Figure 64: Vertical representation of the intersection of the two fuzzy sets.

□

3.5.2. Union

In the same fashion, the union operator can be extended to the realm of fuzzy sets. The proposition "x belongs to μ_1 *or* x belongs to μ_2", which is a disjunction of two atomic propositions. Let s be the t-conorm that models the truth function of the disjunction. The union of two fuzzy sets $\mu_1(x)$ and $\mu_2(x)$ is $\mu_1 \cup \mu_2$, with membership function:

$$(\mu_1 \cup_t \mu_2)(x) = s(\mu_1(x), \mu_2(x))$$

The truth value of the fuzzy proposition "x belongs to μ_1 *or* x belongs to μ_2" is given by:

$$[\![x \in (\mu_1 \cup_t \mu_2)]\!] = [\![x \in \mu_1 \vee x \in \mu_2]\!]$$

Choosing the default $\max$ as t-conorm:

$$(\mu_1 \cup \mu_2)(x) = \max\{\mu_1(x), \mu_2(x)\}$$

Choosing $\max$ and $\min$ for defining of the fuzzy union and the fuzzy intersection respectively has the added benefit of playing well with α-cuts. For any $\alpha \in [0, 1]$ and any fuzzy set μ_1 and μ_2:

$$[\mu_1 \cap \mu_2]_\alpha = [\mu_1]_\alpha \cap [\mu_2]_\alpha \qquad [\mu_1 \cup \mu_2]_\alpha = [\mu_1]_\alpha \cup [\mu_2]_\alpha$$

3.5.3. Complement

In classical logic, the proposition "x belongs to the complement of M" is equivalent to the proposition "x does not belong to M", That is, $[\![x \in \overline{M}]\!] \equiv [\![\neg(x \in M)]\!]$. It is sensible to stick to this definition in fuzzy logic as well: by using the standard truth function for the negation $w_\neg([\![\alpha]\!]) = 1 - [\![\alpha]\!]$, one obtains $\overline{\mu}(x) = 1 - \mu(x)$.

Fuzzy set complement, like non-fuzzy set complement, is **involutory**, meaning that applying it twice is the same as not applying it at all: $\overline{\overline{\mu}} = \mu$ for any fuzzy set μ.

For crisp sets, the **law of excluded middle** holds: the intersection of a crisp set with its complement always results in the universe set. With fuzzy sets, this is not the case. What happens instead is that $(\mu \cap \overline{\mu})(x) \leq 0.5$ and $(\mu \cup \overline{\mu})(x) \geq 0.5$ for any fuzzy set μ and any element x.

3.5.4. Functions with arity 1

Consider two universes X and Y and a crisp set $M \subseteq X$. Let $f : X \mapsto Y$ be a function of arity 1, having X as domain and Y as codomain. The image $f[M]$ of M under f is the subset of Y containing the images of all the elements of M to whom f is applied. That is:

$$f[M] = \{y \in Y \mid \exists x \in X : x \in M \wedge f(x) = y\}$$

Or equivalently:

$$y \in f[M] \iff (\exists x \in X)(x \in M \wedge f(x) = y)$$

Consider a fuzzy set $\mu \in \mathcal{F}(X)$, and let $f[\mu]$ be the image of μ under f. The truth value of the proposition "y belongs to $f[\mu]$" is given by:

$$[\![y \in f[\mu]]\!] = [\![\exists x \in X : x \in \mu \wedge f(x) = y]\!]$$

$f[\mu]$ is itself a fuzzy set, and $f[\mu](y)$ is the membership value of the element y to the image of μ under f. The existential quantifier has been extended as a supremum and using a t-norm as the truth function for the conjunction. This gives:

$$\begin{aligned} f[\mu](y) &= \sup\{[\![x \in \mu \wedge f(x) = y]\!] \mid x \in X\} = \\ &= \sup\{t([\![x \in \mu]\!], [\![f(x) = y]\!]) \mid x \in X\} = \\ &= \sup\{t(\mu(x), [\![f(x) = y]\!]) \mid x \in X\} \end{aligned}$$

Where "$x \in \mu \wedge f(x) = y$" plays the role of the proposition $P(x)$.

Notice how the choice of t in this expression is irrelevant. This is because the proposition $f(x) = y$ is not fuzzy: y either is or is not the image of x under f. That is, $[\![f(x) = y]\!] \in \{0, 1\}$. Recall that, for any t-norm and any $x \in \mathbb{R}$, $t(x, 0) = 0$ and $t(x, 1) = x$. Applying the property to the t-norm in the previous expression:

$$t(\mu(x), [\![f(x) = y]\!]) = \begin{cases} \mu(x) & \text{if } [\![f(x) = y]\!] = 1 \\ 0 & \text{otherwise} \end{cases} = \begin{cases} \mu(x) & \text{if } f(x) = y \\ 0 & \text{otherwise} \end{cases}$$

Thus $f[\mu](y)$ can be reduced to:

$$f[\mu](y) = \sup\left\{ \begin{cases} \mu(x) & \text{if } f(x) = y \\ 0 & \text{otherwise} \end{cases} \mid x \in X \right\} = \sup\{\mu(x) \mid f(x) = y\}$$

Stated otherwise, this means that the degree of membership of an element $y \in Y$ to the image of the fuzzy set $\mu \in \mathcal{F}(X)$ under f is the highest degree of membership that an element of X has to μ such that said element $x \in X$ has y as image under f. This extension of a mapping to fuzzy sets is called **extension principle** (for single-valued functions).

Exercise 3.5.4.1: Consider the fuzzy number with membership function $\Lambda_{-1.5,-0.5,2.5}$, representing the imprecise concept "about -0.5". What would be the fuzzy set representing "the absolute value of about -0.5"?

Solution: The explicit expression for $\Lambda_{-1.5,-0.5,2.5}$ is:

$$\Lambda_{-1.5,-0.5,2.5}(x) = \begin{cases} \frac{x+1.5}{-0.5+1.5} & \text{if} -1.5 \leq x < -0.5 \\ \frac{2.5-x}{2.5+0.5} & \text{if} -0.5 \leq x \leq 2.5 \\ 0 & \text{otherwise} \end{cases} = \begin{cases} x + \frac{3}{2} & \text{if} -\frac{3}{2} \leq x < -\frac{1}{2} \\ -\frac{1}{3}x + \frac{5}{6} & \text{if} -\frac{1}{2} \leq x \leq \frac{5}{2} \\ 0 & \text{otherwise} \end{cases}$$

The fuzzy set of interest is:

$$f\left[\Lambda_{-1.5,-0.5,2.5}\right](y) = \sup\left\{\Lambda_{-1.5,-0.5,2.5}(x) \mid |x| = y\right\}$$

Obviously, for any non-positive value of y, there is no x such that $|x| = y$, since $|x|$ cannot be negative. Therefore, with $y \in (-\infty, 0)$:

$$f_{y\in(-\infty,0)}\left[\Lambda_{-1.5,-0.5,2.5}\right](y) = \sup\left\{\Lambda_{-1.5,-0.5,2.5}(x) \mid |x| = y, y \in (-\infty, 0)\right\} = \sup\{\emptyset\} = 0$$

Since the supremum of the empty set is the smallest element. The same happens when y is greater or equal than 2.5, since $\Lambda_{-1.5,-0.5,2.5}(x) = 0$ for any $x \in [2.5, +\infty)$:

$$f_{y\in[2.5,+\infty)}\left[\Lambda_{-1.5,-0.5,2.5}\right](y) = \sup\left\{\Lambda_{-1.5,-0.5,2.5}(x) \mid |x| = y, y \in [2.5, +\infty)\right\} = \sup\{0\} = 0$$

For any value of y between 1.5 and 2.5, the image behaves like the original function, since $\Lambda_{-1.5,-0.5,2.5}(x) = 0$ for any $(-\infty, -1.5]$:

$$\begin{aligned} f_{y\in[1.5,2.5)}\left[\Lambda_{-1.5,-0.5,2.5}\right](y) &= \sup\left\{\Lambda_{-1.5,-0.5,2.5}(x) \mid |x| = y, y \in [1.5, 2.5)\right\} = \\ &= \sup\left\{\Lambda_{-1.5,-0.5,2.5}(-x), \Lambda_{-1.5,-0.5,2.5}(x)\right\} = \\ &= \sup\left\{0, -\frac{1}{3}x + \frac{5}{6}\right\} = -\frac{1}{3}x + \frac{5}{6} \end{aligned}$$

The same happens for y between 1 and 1.5, because $x + \frac{3}{2}$ with a negative x is always smaller than $-\frac{1}{3}x + \frac{5}{6}$ with a positive x when $|x| > 1$:

$$\begin{aligned} f_{y\in[1,1.5)}\left[\Lambda_{-1.5,-0.5,2.5}\right](y) &= \sup\left\{\Lambda_{-1.5,-0.5,2.5}(x) \mid |x| = y, y \in [1, 1.5)\right\} = \\ &= \sup\left\{\Lambda_{-1.5,-0.5,2.5}(-x), \Lambda_{-1.5,-0.5,2.5}(x)\right\} = \\ &= \sup\left\{-x + \frac{3}{2}, -\frac{1}{3}x + \frac{5}{6}\right\} = -\frac{1}{3}x + \frac{5}{6} \end{aligned}$$

With y between 0 and 0.5, the image behaves like the original, except for a minus sign:

$$\begin{aligned} f_{y\in[0,0.5)}\left[\Lambda_{-1.5,-0.5,2.5}\right](y) &= \sup\left\{\Lambda_{-1.5,-0.5,2.5}(x) \mid |x| = y, y \in [0, 0.5)\right\} = \\ &= \sup\left\{\Lambda_{-1.5,-0.5,2.5}(-x), \Lambda_{-1.5,-0.5,2.5}(x)\right\} = \\ &= \sup\left\{\frac{1}{3}x + \frac{5}{6}, -\frac{1}{3}x + \frac{5}{6}\right\} = \frac{1}{3}x + \frac{5}{6} \end{aligned}$$

The same happens for y between 0.5 and 1, because $x + \frac{3}{2}$ with a negative x is always bigger than $-\frac{1}{3}x + \frac{5}{6}$ with a positive x when $|x| < 1$:

$$f_{y\in[0.5,1)}\left[\Lambda_{-1.5,-0.5,2.5}\right](y) = \sup\left\{\Lambda_{-1.5,-0.5,2.5}(x) \mid |x| = y, y \in [0.5,1)\right\} =$$
$$= \sup\left\{\Lambda_{-1.5,-0.5,2.5}(-x), \Lambda_{-1.5,-0.5,2.5}(x)\right\} =$$
$$= \sup\left\{-x+\frac{3}{2}, -\frac{1}{3}x+\frac{5}{6}\right\} = -\frac{1}{3}x+\frac{3}{2}$$

Summing up:

$$f\left[\Lambda_{-1.5,-0.5,2.5}\right](y) = \begin{cases} \frac{1}{3}y+\frac{5}{6} & \text{if } 0 \leq y < 0.5 \\ -y+\frac{3}{2} & \text{if } 0.5 \leq y < 1 \\ -\frac{1}{3}y+\frac{5}{6} & \text{if } 1 \leq y < 2.5 \\ 0 & \text{otherwise} \end{cases}$$

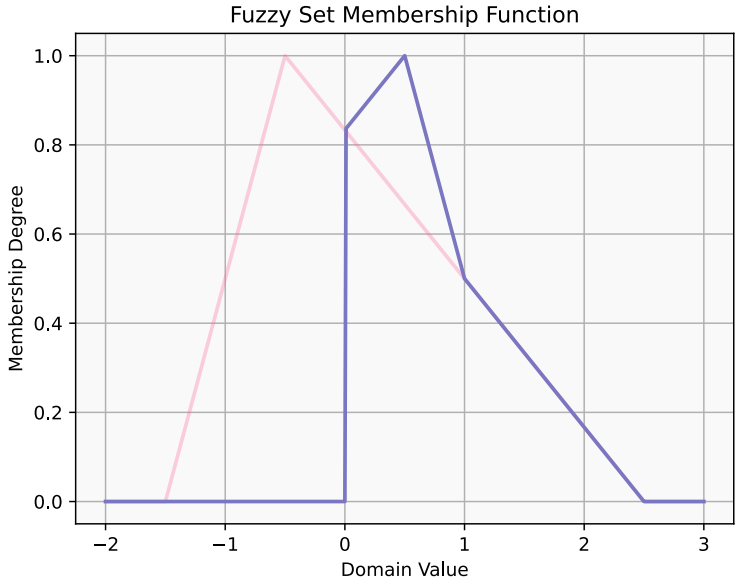


Figure 65: In blue, the vertical representation of the fuzzy set representing "the absolute value of about -0.5". In red, the vertical representation of the fuzzy set representing "about -0.5"

□

3.5.5. Cartesian product

Let M_i with $i = 1, ..., n$ be a family of n crisp sets. The cartesian product of said sets is the set:

$$M_1 \times M_2 \times ... \times M_n = \{(x_1, x_2, ..., x_n) \mid x_1 \in M_1, x_2 \in M_2, ..., x_n \in M_n\}$$

Which is the set of all possible ordered tuples whose i-th element comes from the i-th set.

A tuple $(x_1, ..., x_n)$ is a member of the cartesian product of crisp sets $M_1 \times ... \times M_n$ if and only if each i-th element of the tuple is a member to the i-th set in the product:

$$(x_1, x_2, ..., x_n) \in M_1 \times M_2 \times ... \times M_n \iff x_1 \in M_1 \land x_2 \in M_2 \land ... \land x_n \in M_n$$

Let $U_1, U_2, ..., U_n$ be n universes of discourse. Consider a family of n fuzzy sets $\mu_1, \mu_2, ..., \mu_n$, where each μ_i with $i = 1, ..., n$ is a member of $\mathcal{F}(U_i)$. The cartesian product of the fuzzy sets is a member of the $\mathcal{F}$ set of the cartesian product of the universes:

$$\mu_1 \times \mu_2 \times ... \times \mu_n \in \mathcal{F}(U_1 \times U_2 \times ... \times U_n)$$

To extend the cartesian product to fuzzy sets, refer to the definition of membership to the cartesian product for crisp sets. This gives:

$$[\![(x_1, x_2, ..., x_n) \in \mu_1 \times \mu_2 \times ... \times \mu_n]\!] = [\![x_1 \in \mu_1 \wedge x_2 \in \mu_2 \wedge ... \wedge x_n \in \mu_n]\!]$$

The truth value of the fuzzy proposition "$(x_1, x_2, ..., x_n)$ belongs to $\mu_1 \times \mu_2 \times ... \times \mu_n$" is equivalent to the membership value $(\mu_1 \times \mu_2 \times ... \times \mu_n)(x_1, x_2, ..., x_n)$, which is exactly the cartesian product of the n fuzzy sets. The cartesian product can therefore be extended as:

$$\begin{aligned}(\mu_1 \times \mu_2 \times ... \times \mu_n)(x_1, x_2, ..., x_n) &= [\![(x_1, x_2, ..., x_n) \in \mu_1 \times \mu_2 \times ... \times \mu_n]\!] = \\ &= [\![x_1 \in \mu_1 \wedge x_2 \in \mu_2 \wedge ... \wedge x_n \in \mu_n]\!] = \\ &= \min\{[\![x_1 \in \mu_1]\!], [\![x_2 \in \mu_2]\!], ..., [\![x_n \in \mu_n]\!]\} = \\ &= \min\{\mu_1(x_1), \mu_2(x_2), ..., \mu_n(x_n)\}\end{aligned}$$

Which means that the membership value of a n-tuple of elements to the cartesian product of n fuzzy sets is, out of all the membership values that each i-th element of the tuple has to the i-th fuzzy set, the smallest.

Note that the membership function $(\mu_1 \times ... \times \mu_n)(x_1, ..., x_n)$ of the cartesian product of n fuzzy sets has arity n, and therefore its vertical representation would have to be $(n+1)$-dimensional. This means that the vertical representation of the cartesian product of two fuzzy sets is a tridimensional plot (two universes and the $[0, 1]$ interval on the z axis) and for any cartesian product of 3 or more fuzzy sets no vertical representation is possible.

Exercise 3.5.5.1: Let $X = [-5, 5]$ and $Y = [-10, 10]$ be two universes. Consider the two fuzzy sets $\mu_1 \in \mathcal{F}(X)$ and $\mu_2 \in \mathcal{F}(Y)$, having membership function $\mu_1(x) = \Pi_{-1,0,1,2}(x)$ and $\mu_2(y) = \Lambda_{-5,0,5}(y)$, respectively. What would be $\mu_1 \times \mu_2$?

Solution: By writing:

$$\Pi_{-1,0,1,2}(x) = \begin{cases} x + 1 & \text{if } -1 \leq x < 0 \\ 1 & \text{if } 0 \leq x < 1 \\ 2 - x & \text{if } 1 \leq x < 2 \\ 0 & \text{otherwise} \end{cases} \qquad \Lambda_{-5,0,5}(y) = \begin{cases} 1 - \frac{1}{5}|y| & \text{if } |y| \leq 5 \\ 0 & \text{otherwise} \end{cases}$$

The expression for $(\mu_1 \times \mu_2)(x, y)$ can be written explicitly by going over all possible combinations:

$$(\mu_1 \times \mu_2)(x,y) = \min\{\Pi_{-1,0,1,2}(x), \Lambda_{-5,0,5}(y)\} =$$

$$= \begin{cases} \min\left(x+1, 1-\frac{1}{5}|y|\right) & \text{if } -1 \leq x < 0 \wedge |y| \leq 5 \\ \min(x+1, 0) & \text{if } -1 \leq x < 0 \wedge |y| > 5 \\ \min\left(1, 1-\frac{1}{5}|y|\right) & \text{if } 0 \leq x < 1 \wedge |y| \leq 5 \\ \min(0, 1) & \text{if } 0 \leq x < 1 \wedge |y| > 5 \\ \min\left(2-x, 1-\frac{1}{5}|y|\right) & \text{if } 1 \leq x < 2 \wedge |y| \leq 5 \\ \min(2-x, 0) & \text{if } 1 \leq x < 2 \wedge |y| > 5 \\ \min\left(0, 1-\frac{1}{5}|y|\right) & \text{if } (x < -1 \vee x \geq 2) \wedge |y| \leq 5 \\ \min(0, 0) & \text{if } (x < -1 \vee x \geq 2) \wedge |y| > 5 \end{cases} =$$

$$= \begin{cases} \min\left(x+1, 1-\frac{1}{5}|y|\right) & \text{if } -1 \leq x < 0 \wedge |y| \leq 5 \\ 1-\frac{1}{5}|y| & \text{if } 0 \leq x < 1 \wedge |y| \leq 5 \\ \min\left(2-x, 1-\frac{1}{5}|y|\right) & \text{if } 1 \leq x < 2 \wedge |y| \leq 5 \\ 0 & \text{otherwise} \end{cases}$$

$$\mu_1 \times \mu_2(x,y) = min\{\Pi_{-1,0,1,2}(x), \Lambda_{-5,0,5}(y)\}$$

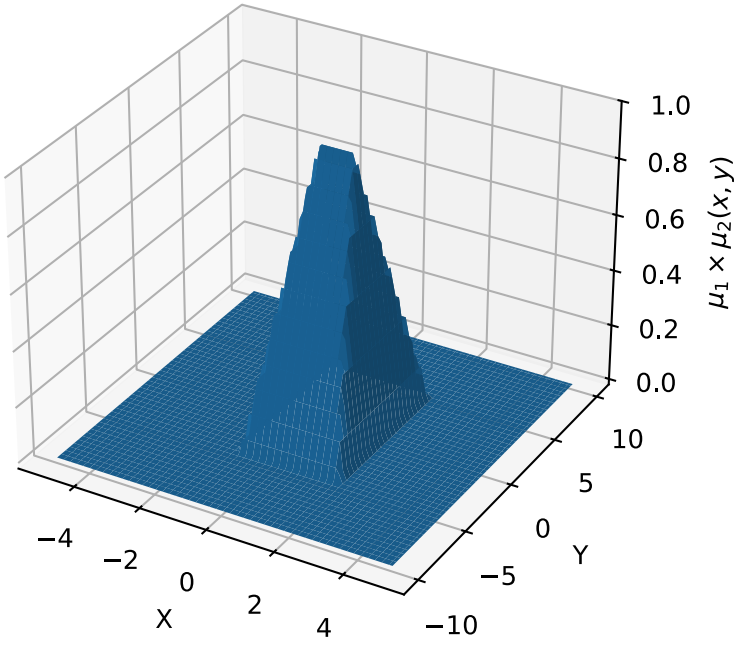


Figure 66: The vertical representation of the fuzzy set $\mu_1 \times \mu_2$

□

3.5.6. Cylindrical extension

When analyzing fuzzy sets, there are situations where there's interest in "artificially" increasing the number of dimensions (this is particularly useful when dealing with fuzzy relations, discussed further). An operation that can increase the number of dimensions is the **cylindrical extension**.

Let $U_1 \times ... \times U_n$ be a cartesian product of n universes, and let $\mu \in \mathcal{F}(U_i)$ be a fuzzy set. The cylindrical extension of μ with respect to $U_1 \times ... \times U_n$ is the fuzzy set $\hat{\pi}_i(\mu)$, with membership function:

$$\hat{\pi}_i(\mu)(x_1, ..., x_n) = \mu(x_i)$$

The idea behind the cylindrical extension can be better understood when considering only two universes. Let $\mu \in \mathcal{F}(X)$ be a fuzzy set and let X, Y be two universes. The cylindrical extension of μ with respect to $X \times Y$ is:

$$\hat{\pi}_X(\mu) = \{((x, y), \mu(x)) \mid x \in X, y \in Y \qquad \hat{\pi}_X(\mu)(x, y) = \mu(x)$$

That is, the cylindrical extension "inflates" the domain with extra dimensions, on which the membership function of the original fuzzy set has no effect. As a matter of fact, the cylindrical extension is a special case of the cartesian product where the membership function of all the other fuzzy sets is always 1:

$$\hat{\pi}_i(\mu)(x_1, ..., x_n) = \min\{1, ..., 1, \mu(x_i), 1, ..., 1\} = \mu(x_i)$$

Exercise 3.5.6.1: Consider the two universes $X = [-10, 10]$ and $Y = [1, 2]$ and the fuzzy set $\mu \in \mathcal{F}(X)$ with $\mu(x) = \Omega_{1,2}(x)$. What is the cylindrical extension of μ with respect to $X \times Y$?

Solution:

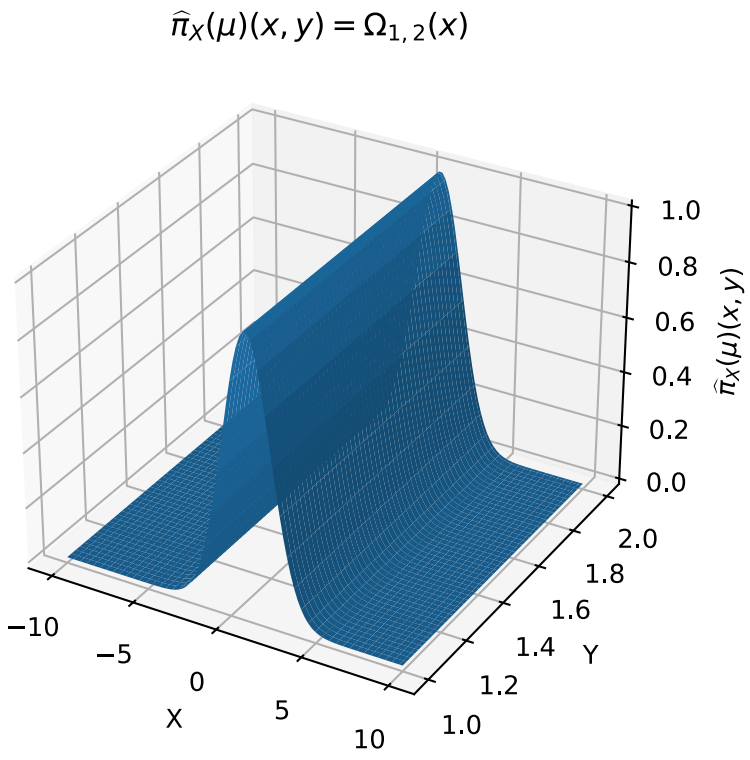


Figure 67: The vertical representation of the fuzzy set $\hat{\pi}_X(\mu)$

□

3.5.7. Projection

Let $U_1 \times ... \times U_n$ with $i \in \{1, ..., n\}$ be a cartesian product of n universe sets. Consider the mapping π_i that has as input an n-tuple of the cartesian product and returns as output the i-th entry of the tuple (a member of U_i):

$$\pi_i : U_1 \times ... \times U_n \mapsto U_i \qquad\qquad \pi_i(x_1, ..., x_n) = x_i$$

This is the **projection** of the cartesian product $U_1 \times ... \times U_n$ onto the coordinate space U_i. The projection of a fuzzy set $\mu \in \mathcal{F}(U_1 \times ... \times U_n)$ onto the space U_i is the fuzzy set $\pi_i[\mu]$, whose membership functions is given by applying the extension principle:

$$\pi_i[\mu](x) = \sup\big\{\mu(x_1, ..., x_{i-1}, x, x_{i+1}, ..., x_n) \mid x_j \in U_j \ \forall j \in \{1, ..., i-1, i+1, ..., n\}\big\}$$

A projection does the opposite of the cylindrical extension, reducing the number of dimensions instead of increasing them. Furthermore, as long as the universes $U_1, ..., U_n$ are nonempty, projecting a cylindrical extension results in the original fuzzy set: $\pi_i[\hat{\pi}_i(\mu)](x) = \mu(x)$.

Exercise 3.5.7.1: Consider Exercise 3.5.5.1. What are the projections of $\mu_1 \times \mu_2$ onto X and Y?

Solution: The projection onto the X space is given by:

$$\pi_x[\mu](x) = \sup_{y \in Y} \left\{ \begin{cases} \min\big(x + 1, 1 - \frac{1}{5}|y|\big) & \text{if } -1 \leq x < 0 \wedge |y| \leq 5 \\ 1 - \frac{1}{5}|y| & \text{if } 0 \leq x < 1 \wedge |y| \leq 5 \\ \min\big(2 - x, 1 - \frac{1}{5}|y|\big) & \text{if } 1 \leq x < 2 \wedge |y| \leq 5 \\ 0 & \text{otherwise} \end{cases} \right\}$$

If $x \in [-1, 0)$, then $x + 1 \in [0, 1]$. The choice of y that maximises $\min\big(x + 1, 1 - \frac{1}{5}|y|\big)$ is $y = 0$, since $1 - \frac{1}{5}|y| \in [0, 1]$ and $1 - \frac{1}{5}|0| = 1$:

$$\pi_{x, x \in [-1, 0)}[\mu](x) = \min(x + 1, 1) = x + 1$$

If $x \in [0, 1)$, the choice of y that maximises $1 - \frac{1}{5}|y|$ is $y = 0$:

$$\pi_{x, x \in [0, 1)}[\mu](x) = \sup_{y \in Y} \left\{ 1 - \frac{1}{5}|y| \right\} = 1 - \frac{1}{5}|0| = 1$$

If $x \in [1, 2)$, then $2 - x \in [1, 2]$. The choice of y that maximises $\min\big(2 - x, 1 - \frac{1}{5}|y|\big)$ is $y = 0$, as shown before:

$$\pi_{x, x \in [1, 2)}[\mu](x) = \min(2 - x, 1) = 2 - x$$

Finally, any choice of $x \in (-\infty, -1) \cup [2, +\infty)$ results in 0. This means that $\pi_x[\mu](x)$ "reconstructs" the membership function of the first fuzzy set of the cartesian product.

The projection onto the Y space is given by:

$$\pi_y[\mu](y) = \sup_{x \in X} \left\{ \begin{cases} \min\big(x + 1, 1 - \frac{1}{5}|y|\big) & \text{if } -1 \leq x < 0 \wedge |y| \leq 5 \\ 1 - \frac{1}{5}|y| & \text{if } 0 \leq x < 1 \wedge |y| \leq 5 \\ \min\big(2 - x, 1 - \frac{1}{5}|y|\big) & \text{if } 1 \leq x < 2 \wedge |y| \leq 5 \\ 0 & \text{otherwise} \end{cases} \right\}$$

Any choice of $y \in (-\infty, -5) \cup [5, +\infty)$ results in 0. If $y \in [-5, +5)$, then the choices of x that give the supremum are any value in the interval $[0, 1]$. In any case, $\pi_y[\mu](y)$ results in $1 - |y|/5$. Therefore, $\pi_y[\mu](y)$ "reconstructs" the membership function of the second fuzzy set of the cartesian product.

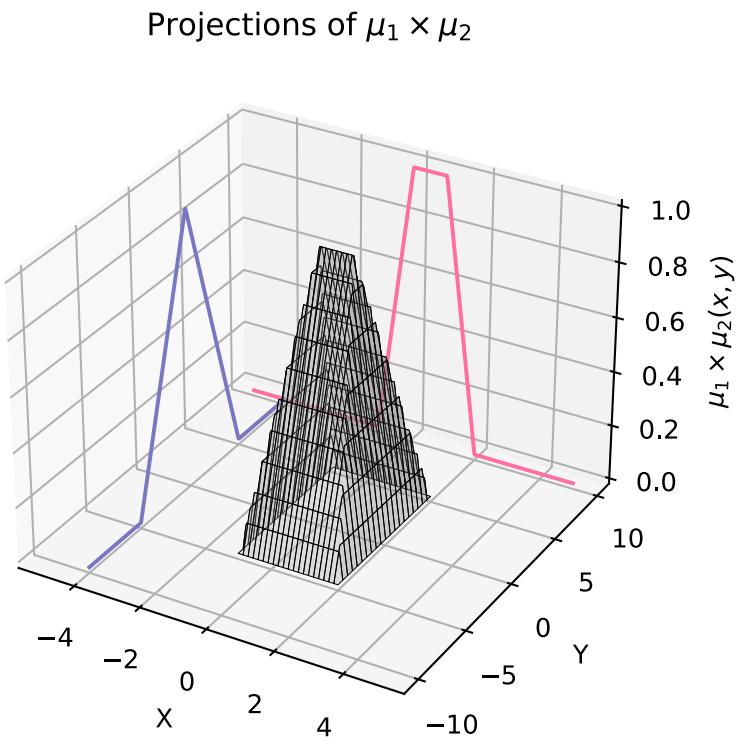


Figure 68: The vertical representation of the projections of $\mu_1 \times \mu_2$ onto X (in red) and onto Y (in blue). The original membership function is in the middle (in black).

□

If the fuzzy sets $\mu_1, ..., \mu_{i-1}, \mu_{i+1}, ..., \mu_n$ are normal, $\pi_i[\mu_1 \times ... \times \mu_n](x) = \mu_i(x)$ holds.

3.5.8. Function with arbitrarely many arguments

Extensions of functions with one argument can be generalized to functions with many arguments from the results obtained on the cartesian product.

Consider a mapping $f : X_1 \times ... \times X_n \mapsto Y$. The image of the tuple $(\mu_1, ..., \mu_n) \in \mathcal{F}(X_1) \times ... \times \mathcal{F}(X_n)$ of n fuzzy sets under the mapping f is the fuzzy set $f[\mu_1, ..., \mu_n]$ evaluated over the entire set Y. That means:

$$f[\mu_1, ..., \mu_n](y) = \sup_{(x_1,...,x_n) \in X_1 \times ... \times X_n} \{(\mu_1 \times ... \times \mu_n)(x_1, ..., x_n) \mid f(x_1, ..., x_n) = y\} =$$
$$= \sup_{(x_1,...,x_n) \in X_1 \times ... \times X_n} \{\min\{\mu_1(x_1), ..., \mu_n(x_n)\} \mid f(x_1, ..., x_n) = y\}$$

Which is the most general form of the extension principle. This allows one to define operations between fuzzy sets such as addition and multiplication. For example, addition between two fuzzy sets $\mu_1 \in \mathcal{F}(X_1)$ and $\mu_2 \in \mathcal{F}(X_2)$ would be:

$$f[\mu_1, \mu_2](y) = \sup_{(x_1,x_2)\in X_1\times X_2} \{\min\{\mu_1(x_1), \mu_2(x_2)\} \mid x_1 + x_2 = y\}$$

Exercise 3.5.8.1: What is $\Lambda_{0,1,2} + \Lambda_{1,2,3}$?

Solution: Recall that:

$$\Lambda_{0,1,2}(x) = \begin{cases} x & \text{if } 0 \leq x < 1 \\ 2 - x & \text{if } 1 \leq x \leq 2 \\ 0 & \text{otherwise} \end{cases} \qquad \Lambda_{1,2,3}(x) = \begin{cases} x - 1 & \text{if } 1 \leq x < 2 \\ 3 - x & \text{if } 2 \leq x \leq 3 \\ 0 & \text{otherwise} \end{cases}$$

The expression of interest is:

$$f\left[\Lambda_{0,1,2}, \Lambda_{1,2,3}\right](y) = \sup_{(x_1,x_2)\in X_1\times X_2} \left\{\min\left\{\Lambda_{0,1,2}(x_1), \Lambda_{1,2,3}(x_2)\right\} \mid x_1 + x_2 = y\right\}$$

Clearly, when $y \in (-\infty, 1)$, $f\left[\Lambda_{0,1,2}, \Lambda_{1,2,3}\right](y)$ is also 0. This is because $x_1 + x_2 = y$ requires either x_1 or x_2 to take a value that results in 0 membership for the corresponding set. The same happens for $y \in [5, +\infty)$:

$$\begin{aligned} f_{y\in(-\infty,1)}\left[\Lambda_{0,1,2}, \Lambda_{1,2,3}\right](y) = f_{y\in(5,+\infty)}\left[\Lambda_{0,1,2}, \Lambda_{1,2,3}\right](y) = \\ = \sup_{(x_1,x_2)\in X_1\times X_2} \left\{\min\left\{\Lambda_{0,1,2}(x_1), \Lambda_{1,2,3}(x_2)\right\} \mid x_1 + x_2 = y\right\} = \\ = \sup_{(x_1,x_2)\in X_1\times X_2} \left\{\min\left\{\Lambda_{0,1,2}(x_1), 0\right\} \mid x_1 + x_2 = y\right\} = \\ = \sup_{(x_1,x_2)\in X_1\times X_2} \left\{\min\left\{0, \Lambda_{1,2,3}(x_2)\right\} \mid x_1 + x_2 = y\right\} = \\ = \sup_{(x_1,x_2)\in X_1\times X_2} \{0\} = 0 \end{aligned}$$

It is also easy to show that $f\left[\Lambda_{0,1,2}, \Lambda_{1,2,3}\right](3) = 1$. This is because the choice of x_1, x_2 that maximises $\min\left\{\Lambda_{0,1,2}(x_1), \Lambda_{1,2,3}(x_2)\right\}$ such that $x_1 + x_2 = 3$ is $x_1 = 1, x_2 = 2$, returning precisely 1 for both membership functions:

$$\begin{aligned} f\left[\Lambda_{0,1,2}, \Lambda_{1,2,3}\right](3) = \sup_{(x_1,x_2)\in X_1\times X_2} \left\{\min\left\{\Lambda_{0,1,2}(x_1), \Lambda_{1,2,3}(x_2)\right\} \mid x_1 + x_2 = 3\right\} = \\ = \min\left\{\Lambda_{0,1,2}(1), \Lambda_{1,2,3}(2)\right\} = \min\{1, 1\} = 1 \end{aligned}$$

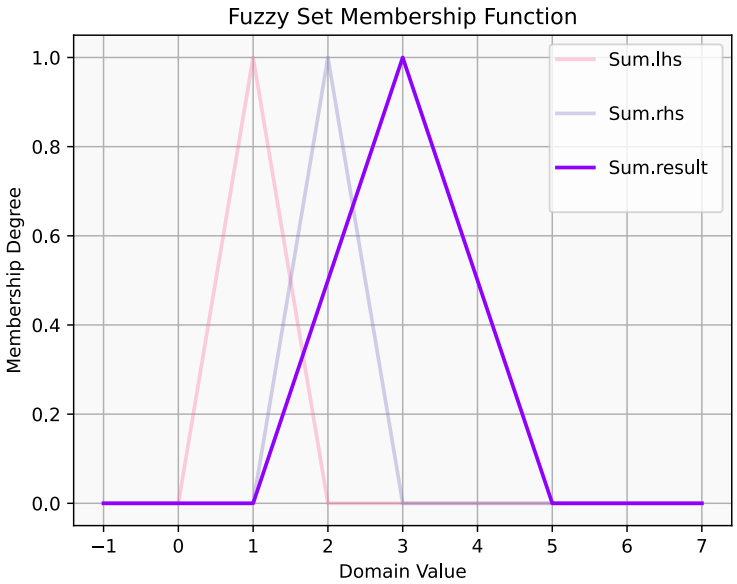


Figure 69: The vertical representation of $\Lambda_{0,1,2} + \Lambda_{1,2,3}$.

□

3.6. Fuzzy reasoning

A (binary) **relation** over the universes X and Y is any subset R of the cartesian product between X and Y. The pairs $(x, y) \in X \times Y$ that belong to the relation R are linked by a semantic connection specified by R.

If a relation R is a discrete set, it can be represented mainly in two ways: by enumerating its pairs or with a binary matrix, where each i, j-th entry is 1 if $(x_i, y_j) \in R$ and 0 otherwise.

Exercise 3.6.1: A house has six doors and each of them has a lock which can be unlocked by certain keys. Let the set of doors be $T = \{t_1, ..., t_6\}$, the set of keys $S = \{s_1, ..., s_5\}$. Key s_5 can open all doors. Key s_1 fits only to door t_1, s_2 to t_1 and t_2, s_3 to t_3 and t_4, s_4 to t_5. How would "key s opens door t" be represented as a relation?

Solution:

$$R \subseteq (S \times T) = \{(s_1, t_1), (s_2, t_1), (s_2, t_2), (s_3, t_3), (s_3, t_4), (s_4, t_5), (s_5, t_1), (s_5, t_2), (s_5, t_3), (s_5, t_4), (s_5, t_5), (s_5, t_6)\}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

□

Functions are a special case of relations: if the function $f : X \mapsto Y$ maps X to Y, the graph of f (the set of all input-output pairs of X and Y mediated by f) is the relation:

$$\text{graph}(f) = \{(x, f(x) \mid x \in X)\}$$

As long as, for any x, there is one and only $y \in Y$ such that $f(x) = y$.

As functions, a relation can be applied to an entire set. If $R \subseteq X \times Y$ is a relation between the universes X and Y and $M \subseteq X$ is a subset of X, the image of M under R is the set:

$$R[M] = \{y \in Y \mid \exists x \in X : (x, y) \in R \wedge x \in M\} = \pi_Y[R \cap \hat{\pi}_X(M)]$$

That is, $R[M]$ contains the elements from Y that appear in R paired with an element of M at least once. The image of a subset under a relation corresponds to the "inversion" of the semantical link between the pairs.

If $f : X \to Y$ is a function, then applying the relation $\text{graph}(f)$ to a singleton set $\{x\} \subseteq X$ gives:

$$\text{graph}[\{x\}] = \{y \in Y \mid \exists x \in X : (x, y) \in R \wedge x \in \{x\}\} = \{f(x)\}$$

Which is the singleton set containing the image of x under the function f. In general:

$$\text{graph}(f)[M] = f[M] = \{y \in Y \mid \exists x \in X : x \in M \wedge f(x) = y\}$$

For any subset $M \subseteq X$.

Exercise 3.6.2: Consider Exercise 3.6.1. Let $M = \{s_1, s_2, s_3, s_4\}$. What would be the set of doors that can be opened with the keys in M at one's disposal?

Solution: The expression of interest is:

$$R[M] = R[\{s_1, s_2, s_3, s_4\}] = \pi_Y[R \cap \hat{\pi}_X(\{s_1, s_2, s_3, s_4\})]$$

Since:

$$\begin{aligned}\hat{\pi}_X(\{s_1, s_2, s_3, s_4\}) = \{&(s_1, t_1), (s_1, t_2), (s_1, t_3), (s_1, t_4), (s_1, t_5), (s_1, t_6),\\ &(s_2, t_1), (s_2, t_2), (s_2, t_3), (s_2, t_4), (s_2, t_5), (s_2, t_6),\\ &(s_3, t_1), (s_3, t_2), (s_3, t_3), (s_3, t_4), (s_3, t_5), (s_3, t_6),\\ &(s_4, t_1), (s_4, t_2), (s_4, t_3), (s_4, t_4), (s_4, t_5), (s_4, t_6)\}\end{aligned}$$

This gives:

$$\begin{aligned}R[\{s_1, s_2, s_3, s_4\}] &= \pi_Y[R \cap \hat{\pi}_X(\{s_1, s_2, s_3, s_4\})] =\\ &= \pi_Y[\{(s_1, t_1), (s_2, t_1), (s_2, t_2), (s_3, t_3), (s_3, t_4), (s_4, t_5)\}] =\\ &= \{t_1, t_2, t_3, t_4, t_5\}\end{aligned}$$

□

Relations are useful because they can model logical inferences. Consider a logical deduction based on an implication of the form $x \in A \to y \in B$, with $A \subseteq X$ and $B \subseteq Y$ crisp sets. The statement "if x belongs to A then y belongs to B" can be encoded into a relation in the following way:

$$\begin{aligned}R(x, y) &= \{(x, y) \in X \times Y \mid x \in A \to y \in B\} =\\ &= (A \times B) \cup \left(\overline{A} \times \overline{B}\right) \cup \left(\overline{A} \times B\right) =\\ &= (A \times B) \cup \left(\overline{A} \cup \overline{A} \times \overline{B} \cup B\right) =\\ &= (A \times B) \cup \left(\overline{A} \times Y\right)\end{aligned}$$

Inferring new facts from rules and known facts usually means dealing with chained deduction steps in the form of $\varphi_1 \to \varphi_2$, $\varphi_2 \to \varphi_3$ from which one can derive $\varphi_1 \to \varphi_3$. A similar principle can be formulated in the context of relations.

Consider the relations $R_1 \subseteq X \times Y$ and $R_2 \subseteq Y \times Z$. An element $x \in X$ is indirectly related to an element $z \in Z$ if there exists an element $y \in Y$ such that x and y are in the relation R_1 and y and z are in the relation R_2. In this way, the composition of the relations R_1 and R_2 can be defined as the relation:

$$R_2 \circ R_1 = \{(x, z) \in X \times Z \mid \exists y \in Y : (x, y) \in R_1 \wedge (y, z) \in R_2\}$$

Then, for all $M \subseteq X$:

$$R_2[R_1[M]] = (R_2 \circ R_1)[M]$$

Exercise 3.6.3: Consider Exercise 3.6.1. Suppose that $P = \{p_1, p_2, p_3\}$ is the set of people owning the keys. The relation "person p is the owner of key s" is:

$$R' \subseteq (P \times S) = \{(p_1, s_1), (p_1, s_2), (p_2, s_3), (p_2, s_4), (p_3, s_5)\}$$

What would be $R \circ R'$, the relation "person p can open door t"? What would be $(R \circ R')[\{p_1, p_2\}]$?

Solution: Since $R \subseteq (S \times T)$:

$$\begin{aligned} R \circ R' &= \{(p,t) \in P \times T \mid \exists s \in S : (p,s) \in R' \wedge (s,t) \in R\} = \\ &= \{(p_1,t_1),(p_1,t_2),(p_2,t_3),(p_2,t_4),(p_2,t_5),(p_3,t_1), \\ &\quad (p_3,t_2),(p_3,t_3),(p_3,t_4),(p_3,t_5),(p_3,t_6)\} \end{aligned}$$

Which gives:

$$(R \circ R')[\{p_1,p_2\}] = \pi_T[(R \circ R') \cap \hat{\pi}_P(\{p_1,p_2\})]$$

Since:

$$\begin{aligned} \hat{\pi}_P(\{p_1,p_2\}) = \{&(p_1,t_1),(p_1,t_2),(p_1,t_3),(p_1,t_4),(p_1,t_5), \\ &(p_2,t_1),(p_2,t_2),(p_2,t_3),(p_2,t_4),(p_2,t_5)\} \end{aligned}$$

One has:

$$\begin{aligned} (R \circ R')[\{p_1,p_2\}] &= \pi_T[(R \circ R') \cap \hat{\pi}_P(\{p_1,p_2\})] = \\ &= \pi_T[\{(p_1,t_1),(p_1,t_2),(p_2,t_3),(p_2,t_4),(p_2,t_5)\}] = \\ &= \{t_1,t_2,t_3,t_4,t_5\} \end{aligned}$$

□

Relations can also be extended to fuzzy sets. A fuzzy set $\rho \in \mathcal{F}(X \times Y)$ is called a (binary) **fuzzy relation** between the universe sets X and Y. A fuzzy relation is a generalization of a "standard" relation where, instead of having elements of X and Y that are either paired or not paired, have a degree of "pairedness" quantified by $\rho(x,y)$.

Exercise 3.6.4: Suppose that a caliber has a precision of 0.1. That is, if a value of x millimetres is measured with the caliber, then the "real" value of the measurement lies in the interval $[x - 0.1, x + 0.1]$. Represent the relationship "the measured value x corresponds to the real value y" with a fuzzy relation.

Solution: Let X be the universe of all possible values that can be measured, and let Y be the universe of all "real" values of a measurement. For simplicity, both X and Y can be assumed to be $\mathbb{R}$. A crisp relation that connects X and Y would be:

$$R = \{(x,y) \in X \times Y \mid |x - y| \leq 0.1\}$$

The issue is that, phrased this way, it is equally reasonable to associate x to any value in $[x + 0.1, x - 0.1]$. This makes little sense, because a value of y that is closer to x should obviously more likely correspond to the real value than a value of y that is far away from x.

The fuzzy relation $\rho \in \mathcal{F}(\mathbb{R} \times \mathbb{R})$ that connects the measured values with the actual values can be given, for example, the following membership function:

$$\rho(x,y) = 1 - \min\{10|x - y|, 1\}$$

This way, the closer the real value is to the measurement, the more likely it is for it to be the real value for the measurement. In particular, if $x = y$, then $\rho(x,y) = 1$, meaning that if the real value is x then the measurement perfectly overlaps. Also, if $|x - y| \geq 0.1$ then $\rho(x,y) = 0$,

because a value of y that dists more than 0.1 from x is outside the precision range, and cannot correspond to its true value. □

A discrete fuzzy relation can still be represented in matrix form, where each i, j-th entry is not either 0 or 1 but $\rho(x, y)$.

The extention of the image of a relation to fuzzy sets follows from applying the extention principle:

$$\begin{aligned}\rho[\mu](y) &= [\![y \in \rho[\mu]]\!] = [\![\exists x \in X : (x, y) \in \rho \wedge x \in \mu]\!] = \\ &= \sup_{x \in X}\{[\![(x, y) \in \rho \wedge x \in \mu]\!]\} = \\ &= \sup_{x \in X}\{\min\{[\![(x, y) \in \rho]\!], [\![x \in \mu]\!]\}\} = \\ &= \sup_{x \in X}\{\min\{\rho(x, y), \mu(x)\}\}\end{aligned}$$

This membership value of a certain y_0 should be interpreted as how much one can believe that it is possible that the variable y attains the value y_0.

Exercise 3.6.5: Consider Exercise 3.6.4. Suppose that $\mu(x) = \Lambda_{0.2,0.3,0.4}(x)$. What is $\rho[\mu](y)$?

Solution:

$$\begin{aligned}\rho[\mu](y) &= \sup_{x \in X}\{\min\{\rho(x, y), \mu(x)\}\} = \\ &= \sup_{x \in X}\big\{\min\big\{1 - \min\{10|x - y|, 1\}, \Lambda_{0.2,0.3,0.4}(x)\big\}\big\} = \\ &= \sup_{x \in X}\left\{\min\left\{1 - \min\{10|x - y|, 1\}, \begin{cases} 10x - 2 & \text{if } 0.2 \leq x < 0.3 \\ 4 - 10x & \text{if } 0.3 \leq x < 0.4 \\ 0 & \text{otherwise} \end{cases}\right\}\right\} = \\ &= 1 - \min\{5|y - 0.3|, 1\}\end{aligned}$$

□

As stated, an implication of the form $x \in A \to y \in B$ can be represented using a relation. Replacing A and B with the fuzzy sets μ and ν and using the Gödel implication as truth function for the implication, one has:

$$\rho(x, y) = [\![(x, y) \in \rho]\!] = [\![x \in \mu \to y \in \nu]\!] = \begin{cases} 1 & \text{if } \mu(x) \leq \nu(y) \\ \nu(y) & \text{otherwise} \end{cases}$$

Exercise 3.6.6: Model the implication "If x is about 2, then y is about 3" as a fuzzy implication.

Solution:

$$\rho(x, y) = \begin{cases} 1 & \text{if} \min\{|3 - y|, 1\} \leq |2 - x| \\ 1 - \min\{|3 - y|, 1\} & \text{otherwise} \end{cases}$$

□

It's possible to extend relation compositions to fuzzy sets. Given two fuzzy relations $\rho_1 \in \mathcal{F}(X \times Y)$ and $\rho_2 \in \mathcal{F}(Y \times Z)$, their composition is the fuzzy relation:

$$\begin{aligned}(\rho_2 \circ \rho_1)(x,z) &= [\![(x,z) \in (\rho_2 \circ \rho_1)]\!] = [\![\exists y \in Y : (x,y) \in \rho_1 \wedge (y,z) \in \rho_2]\!] = \\ &= \sup_{y \in Y}\{[\![(x,y) \in \rho_1 \wedge (y,z) \in \rho_2]\!]\} = \\ &= \sup_{y \in Y}\{\min\{[\![(x,y) \in \rho_1]\!], [\![(y,z) \in \rho_2]\!]\}\} = \\ &= \sup_{y \in Y}\{\min\{\rho_1(x,y), \rho_2(y,z)\}\}\end{aligned}$$

As it was the case for crisp relations, for all $\mu \in \mathcal{F}(X)$:

$$\rho_2[\rho_1[\mu]] = (\rho_2 \circ \rho_1)[\mu]$$

Exercise 3.6.7: Consider Exercise 3.6.4. Suppose that the caliber is digital and the display that shows the measurement is scratched, giving unclear results that have to be interpreted. Let $\rho'(a,x) = 1 - \min\{5|a-x|, 1\}$ be the relation that models how far is the value measured by the caliber from the value read on the display. What is the relation that connects the values read from the display to the real measured values?

Solution:

$$\begin{aligned}(\rho \circ \rho')(a,y) &= \sup_{x \in X}\{\min\{\rho'(a,x), \rho(x,y)\}\} = \\ &= \sup_{x \in X}\{\min\{1 - \min\{5|a-x|, 1\}, 1 - \min\{10|x-y|, 1\}\}\} = \\ &= 1 - \min\left\{\frac{10}{3}|a-y|, 1\right\}\end{aligned}$$

□

4. Evolutionary computing

4.1. Evolutionary algorithms

Metaheuristics are computational techniques typically used to solve numerical and combinatorial optimization problems with a satisfactory degree of approximation. They are often employed where an analytical solution is either impossible or impractical to compute. These techiques are very general, offering a blueprint that has to be adapted to the specific problem at hand. Aside from the fact that all such algorithms apply some form of guided search, their methodologies and inspirations vary greatly.

The oldest and widely employed form of metaheuristics is **evolutionary computing**. Evolutionary computing seeks to solve optimization problems constructing algorithms, called **evolutionary algorithms**, that draw inspiration from nature-driven and biological processes, **biological evolution** in particular. Biological evolution explains why the living beings on Earth look as they do and where do they come from.

The existence of biological evolution is the result of the **Theory of Evolution** by Charles Darwin. The most important underpinning of the Theory is the presence in nature of a "driving force", **natural selection**: with respect to a given environment, one or more traits that can appear randomly in species may be favoured or disfavoured by natural selection. Species with favoured traits tend to thrive and reproduce, passing the acquired traits onto their offspring, whereas species with unfavoured traits tend to die out.

New or modified traits may be created by various processes. It can happen by chance in a single individual, for example from exposure to radiation or from an error in DNA duplication. It can also happen during reproduction, where the offspring inherits half set of chromosomes from each parent, therefore creating a new unique combination of traits. It can also happen during the **meiosis** process, when **crossing over** recombines homologous chromosomes.

The improvements carried out by these modification may vary: allowing an individual to find more and/or better food, better fend off predators, increase its reproductive capabilities, ecc... It should be noted, that such modified traits are not beneficial or harmful in themselves, only with respect to the environment in which species live. A desirable trait in one environment might turn out to be a setback in a different one.

The vast majority of random (genetic) modifications are actually harmful for the individual, either limiting its capabilities or even making it unfit to live. Each variation is immediately put to the test with respect to an environment and only the beneficial ones, a very tiny proportion, is kept. This is why biological evolution is so slow: most modifications are detrimental and become lost in time. To observe a substantial change, many small improvements have to accumulate over many generations.

A more detailed list of the principles underpinning the Theory of Evolution is the following:

- **Diversity**: all forms of life, even organisms of the same species, differ from each other, both genetically and physically. Nevertheless, the currently existing life forms are only a tiny fraction of the theoretically possible ones;
- **Variation**: mutation and genetic recombination continuously create new variants. These can result in a new combination of already existing traits or may introduce a new trait that has never been seen before;
- **Inheritance**: genetic variations are passed onto the offspring, physical variations are not;
- **Speciation**: a new species if formed when two or more subgroups of life forms from the same species have acquired so many cumulated variations that cannot crossbreed anymore;

- **Birth surplus/Overproduction**: (nearly) all life forms produce more offspring than can ever become mature enough to procreate themselves. That is, quantity is often favoured over quality;
- **Natural Selection**: on average, the survivors of a population exhibit such hereditary variations which increase their adaptation to the local environment;
- **Randomness/Blind Variation**: variations are random, both in cause and in intent. No variation is preprogrammed to "push" evolution in one direction;
- **Gradualism**: variations happen in small steps, thus appreciable changes are gradual and very slow;
- **Evolution/Transmutation/Inheritance with Modification**: having to adapt to the environment, species are not immutable, evolving instead in the course of time;
- **Discrete Genetic Units**: the genetic information is stored in continuous units, but in discrete ones, the genes;
- **Opportunism**: the evolution process builds upon the living beings as they are in the present, cannot create variations out of nothing;
- **Evolution-strategic Principles**: not just organisms are optimized for their environment, but also the *mechanisms* of evolution itself, such as reproduction rates, mortality rates, life spans, evolutionary speed, etc... become optimized;
- **Ecological Niches**: species that compete with each other can avoid coming into conflict only if they occupy different ecological niches. Otherwise, one would prevail over the others;
- **Irreversibility**: the course of evolution is irreversible. That is, a species cannot "evolve backwards";
- **Unpredictability**: the evolution process has no direction and no purpose, therefore it cannot be predicted;
- **Increasing Complexity**: biological evolution has led to increasingly more complex living beings, from cells to animals, over bilions of years of small changes.

The idea behind evolutionary algorithms is therefore to encode an optimization problem as a simulated evolutionary process. That is, representing the possible solutions as if they were living beings in a certain environment, and the desireable traits are those that represent an optimal solution. As biological evolution favours individuals fit for an environment, simulated evolution should favour solutions that are closer to an optimum.

To construct such an encoding, it is first necessary to introduce some terminology. Terms borrowed by biology ought to be translated into their computer science counterpart.

Individual is the term used in biology to refer to a single living being of a given species. Individuals have a higher or lesser degree of adaptation to their environment and are subject to natural selection. In computer science, an individual corresponds to a potential solution to the given optimization problem.

A **chromosome** is a string of DNA enveloped in proteins, that stores the genetic information of an individual, its "blueprint" that encodes its traits. In computer science, a chromosome would be the information related to a certain solution, stored as bits. Note that most living organisms don't have a single chromosome. Instead, they have more than one chromosome, and the genetic information is distributed (unequally) among them. Moreover, each chromosome exists in two copies for each individual, one inherited from each parent. In computer science there is no need to model this aspect[13], and all genetic information can be packed in a single chromosome.

A **gene** is the fundamental unit of inheritance as it determines (a part of) a trait or characteristic of an individual. The location of a gene in a DNA strand, called **locus** is (almost) fixed, meaning that it's (almost) possible to refer to a genetic trait with respect to a position on the strand. In computer science, a gene is one of the properties of a solution.

[13]This wouldn't be the case if the intent was to model an actual, really existing organism.

The possible ways or modes in which a gene can exist are called **alleles**. In computer science, an allele is one of the possible choice of values that can be made for a given property. Note that, in biology, each individual actually has two alleles, one for each chromosome in a pair. Again, in computer science there is no need to model this aspect, and each individual is assumed to have a single allele for each gene.

The **genotype** is the genetic configuration of an organism, which alleles are present for each of its genes, whereas the **phenotype** is the physical appearance of an organism, the way the genotype manifests itself. Since it's the phenotype that actually interacts with the environment, not the genotype, it's the phenotype that determines how much an individual is adapted to their environment. In computer science, the genotype corresponds to the way a potential solution is encoded, while whereas the phenotype is the implementation or application of a candidate solution, from which the fitness of the corresponding individual can be read.

A **population** is a set of individuals, usually understood to be of the same species. A **generation** is a "snapshot" of a population at a certain point in time. In computer science, a population is a *multiset* (a set where elements can appear more than once) of individuals, and a generation is the population at a given iteration of the algorithm. In biology, no two individuals from the same population can be an exact genetic copy of one another[14], since the number of possible combinations of genes is too big for this to happen. In computer science, in general, there is no need to impose this restriction, therefore identical individuals can coexist.

A new generation is created by **reproduction**, that is, by the generation of offspring from existing organisms, in which genetic material of the parent individuals may be recombined. The same holds for computer science, where a new individual of the population is created by recombining and shuffling information between individuals. In biology, reproduction occurs either between two individuals or just one, in computer science it is possible to have reproduction between more than two individuals.

The **fitness** of an individual is a measure of its degree of adaptation to the environment, that is, how high its chances of survival and reproduction are. In biology, a precise and quantifiable definition of fitness does not exist. In computer science, it's just the value of the fitness function of the optimization problem at hand when evaluating a given potential solution.

Summing up, to construct an evolutionary algorithm the following building blocks are required:

- An encoding that represents potential solutions as individuals. The encoding of choice is highly dependent on the problem at hand, and there is no one-size fits-all method for doing so;
- A method to create an initial population. Since computer science chromosomes are mostly just strings, the initial population is often random sequences. If the solution candidates have to satisfy certain constraints, a more refined approach might be needed;
- A fitness function that assigns a fitness to each individual. In many cases, this is just the evaluation function of the optimization problem, but it can also include additional constraints that an individual must possess;
- A selection method that simulates natural selection. The selection method should assign greater odds of reproducting to individuals with higher fitness and smaller odds to individual with lower fitness. In the most straightforward case, this means assigning a probability of reproducing to each individual that is proportional to their fitness;
- A set of operators that create new individuals from the existing ones, simulating reproduction and genetic mutation. For example, randomly changing a gene in each individual with a different allele or merging part of the genotype of two individuals into a new individual;
- A termination criterion, to prevent the algorithm from continuing indefinitely. For example, stopping the algorithm after a fixed amount of maximum iterations. This is necessary because,

[14] Not even homozygous twins.

by definition, it's not possible to solve an optimization problem with guided search with perfect accuracy, and an error bound has to be set;
- Values for various tunable parameters, like for example the number of individuals in the population, or maximum number of iterations.

A generic evolutionary algorithm can be written as such:

GENERIC-EVOLUTIONARY-ALGORITHM(ε):

```
1  t ← 0                                          // Starting time
2  INITIALIZE (pop(t))                            // Create the initial population
3  EVALUATE (pop(t))                              // Compute fitness
4  while (not(ε))
5      t ← t + 1
6      pop(t) ← SELECT-FROM (pop(t − 1))          // Select individuals based on fitness
7      ALTER (pop(t))                             // Apply genetic operators
8      EVALUATE (pop(t))                          // Evaluate the new population
```

After having created an initial population of solution candidates, each individual is evaluated. A new generation is created by selecting individuals based on their fitness, and then entirely new individuals are created by mutating existing individuals or having two or more individuals "reproduce". Each individual in the new population is evaluated, and the process is repeated. The algorithm stops when the chosen termination criterion ε is fulfilled.

4.2. Choosing a solution encoding

A good choice of the solution encoding can significantly speed up the process of finding an optimum, restricting the search space excluding unneeded subdomains. On the other hand, a poor choice of the encoding can result in an algorithm that has to navigate through many unfruitful solutions or, even worse, that does not find an optimum at all.

First off, it is important to pay attention to the interplay between the chosen encoding and the genetic operators. If a certain encoding reduces the search space but becomes hard to find genetic operators that are closed under said space, it is necessary to handle such edge cases. If this is not possible, it may be better to fall back to a looser encoding (incorporating fewer constraints) but that allows for simpler choices of genetic operators.

In general, there are three desirable properties for an encoding to have. The first one can be summarized as: *similar phenotypes should be represented by similar genotypes*. A very intuitive way to quantify the similarity between two genotypes is through **edit distance**, that is, the minimum number of mutations necessary to completely convert the first genotype into the second (or vice versa): the more mutations are needed, the less similar they are.

However, what is evaluated by the fitness function is the phenotype of the individual, not the genotype. It is reasonable to assume that similar phenotypes will yield similar values of the evaluation function, since this allows the search space to be explored using said fitness as a guidance. Since evolutionary algorithms only modify the genotype of individuals, not their phenotype, similar modifications of the genotype should be reflected in similar modifications of the phenotype, because otherwise it might be impossible to obtain a similar phenotype by small genetic modifications.

Even though this might seem unlikely at first hand, there are "problematic" encodings where completely different genotypes, under the effect of the same genetic operators, will yield similar phenotypes. Such encodings ought to be avoided.

To give an instructive example, suppose there's the need to encode the numbers inside a real-valued interval $[a, b]$ as binary numbers. Since such interval is not discrete, it is impossible to have a one-to-one mapping between $[a, b]$ and the set of binary numbers.

A possible way to circumvent the problem would be to fix a certain precision ε, to partition $[a, b]$ into smaller intervals of size greater or equal than ε and then map a binary number to each of these intervals. That is, one creates 2^k smaller intervals out of $[a, b]$, with $k = \lceil \log_2(\frac{b-a}{\varepsilon}) \rceil$, mapped to the (binary) numbers $0_2, 1_2, \left(2^k - 1\right)_2$. Therefore, the binary number:

$$z = \lfloor \frac{x - a}{b - a}(2^k - 1) \rfloor$$

Differs from x by at most ε. The opposite operation can also be performed, reconstructing an approximation x' of the original value x as:

$$x' = a + z\left(\frac{b - a}{2^k - 1}\right)$$

The difference between two binary numbers is given by their **Hamming distance**, the number of bits (digits) of the two in the same position having different values. It is easy to see that close numbers might be encoded into binary numbers that aren't close at all, meaning that they have a large Hamming distance.

Exercise 4.2.1: Suppose one wants to encode the real numbers in $[0, 1]$ onto 4 bits. What is the Hamming distance between the encoding of 0.5326 and 0.5400?

Solution: This results into the encoding:

$$z = \lfloor \frac{x - 0}{1 - 0}(2^4 - 1) \rfloor = \lfloor x(16 - 1) \rfloor = \lfloor 15x \rfloor$$

This gives:

$$\lfloor 15 \cdot 0.5326 \rfloor = \lfloor 7.9890 \rfloor = 7_{10} = 0111_2 \qquad \lfloor 15 \cdot 0.5400 \rfloor = \lfloor 8.1000 \rfloor = 8_{10} = 1000_2$$

Even though the two original numbers are very close, their encoding have completely different bits, hence their Hamming distance (4) is maximal. □

This isn't necessarely a problem *per se*, but it might very well be if such encoding were to be employed in a genetic algorithm. Even if the "phenotypical distance" between two individuals (the original numbers) might be small, their "genotypical distance" (the Hamming distance of their encodings) might as well be huge. This means that even if the algorithm were to find a solution with high fitness, it would have little use for it, since manipulating its genotype will most likely result in individuals whose encoding is completely different, hence having completely different fitness.

Conversely, to actually obtain a phenotype similar to the one of the current solution, it might be necessary to employ a huge amount of mutations (and luck) in order to cope with the very large Hamming distance between the two. For this reason, these distances are also called **Hamming cliffs**, stressing the difficulty in "climbing" such obstacles in the path of a better solution.

This problem can be solved by using a different encoding, for example **Gray codes**, an encoding where the representations of numbers that are next to each other always have Hamming distance equal to 1. This way, a genotypical difference would be better reflected into a phenotypical difference, and vice versa. The most common form of Gray encoding and decoding is, respectively:

$$g = z \oplus \lfloor \frac{z}{2} \rfloor \qquad\qquad z = \bigoplus_{i=0}^{k-1} \lfloor \frac{g}{2^i} \rfloor$$

Where $\oplus$ is the bitwise XOR and z is the number encoded as it was done previously. Note that dividing a binary number by 2 amounts to shifting its digits by one position to the right (inserting a 0 as most significant digit).

Exercise 4.2.2: What would be the Gray encodings of the two numbers in Exercise 4.2.1?

Solution:

$$0111_2 \oplus \frac{0111_2}{2} = 0111_2 \oplus 0011_2 = 0100_2 \qquad 1000_2 \oplus \frac{1000_2}{2} = 1000_2 \oplus 0100_2 = 1100_2$$

Their Hamming distance is indeed 1. □

Another principle of evolutionary algorithm design can be syntetized as: *similarly encoded candidate solutions should have a similar fitness*. In biology, the term **epistasis** refers to alleles of genes that are capable of shutting down completely the expression of other alleles of the same gene. In the context of

evolutionary computing, epistasis refers to how much influence a gene has on the value of the fitness function.

A solution encoding is said to manifest high epistasis if a small modification on a gene produces a considerable difference in the fitness of the solution. On the other hand, an encoding manifests low epistasis if a small modification on a gene produces differences of comparable orders of magnitude.

Exercise 4.2.3: Consider the Travelling Salesman Problem, and two possible encodings for a solution:

- A permutation of the nodes, meaning that the i-th node of the permutation is the i-th visited;
- A list of numbers, each specifying the node to be visited in that time frame as the index of a sorted list of all non visited nodes. That is, if a gene has value i, it means that the node to be visited in that time frame is the i-th not yet visited node.

Which one has the smallest epistasis?

Solution: The first one (and is also both simpler an more intuitive). This is because introducing a mutation, such as swapping the values of two genes, will produce a comparable result, no matter which genes are chosen. On the other hand, if in the second encoding two genes are exchanged, the sequence of visited nodes can change drastically, as the value of the fitness. □

Having high epistasis is undesirable, since it becomes very hard to use an evolutionary algorithm effectively. One reason is that if small phenotypical variations result in huge fitness variations destroys the assumption that a small "nudge" to the genotype is reflected in an equally small "nudge" to the genotype, hence making it impossible to use the fitness function to aid the search.

The third staple of solution encoding is: *the search space should be closed under the used genetic operators*. Indeed, if a genetic operator mutates an individual into an new individual that is not a member of the search space, by definition it cannot be a solution.

In the best case scenario, this just results in wasting computational time. This happens when a genetic operator produces individuals that aren't valid solutions and also having very poor fitness, hence they will be (hopefully) discarded in the next generations. However, such individuals may pollute the seach space with their offspring, preventing the evolution process to converge. In the worst case scenario, non conforming individuals might actually have very good fitness, tricking the algorithm into choosing them as solution, hence rendering it incorrect.

Generally, an individual created by a genetic operator is said to lie outside of the search space if:

- Its chromosome cannot be meaningfully interpreted or decoded;
- The represented candidate solution does not fulfill certain basic requirements;
- The represented candidate solution is evaluated incorrectly by the fitness function.

It is clearly better to not have to deal with such unwanted individuals in the first place. However, if a very promising choice of genetic operators and/or solution encoding has the side effect of potentially producing individuals outside of the search space, those can be tolerated as long as their presence is properly addressed. The main not mutually exclusive options to do so are:

- Choosing a different solution encoding (at the potential cost of enlarging the search space);
- Choosing a different set of genetic operators such that they are closed under the search space;
- Introduce repair mechanisms that "patch" individuals that fall out of the search space so that they are brought back in;

- Introduce a fitness penalty for non conforming individuals, so that they are guaranteed to be discarded in the evolution process.

Exercise 4.2.4: Suppose that a new encoding is chosen for the n-queens problem. Instead of n numbers from 0 to $n-1$, the encoding consists of a permutation of the numbers in the interval $\{0, 1, ..., n-1\}$, where the i-th element still represents the position in the row for the i-th queen. What would happen if the same genetic operators (one-point crossover, standard mutation) are chosen?

Solution: It is evident that both operators can produce individuals that are not solutions anymore, since resulting individuals might have duplicates. This issue can be addressed in the following ways:

- Reverting back to the previous encoding, which does not have this problem but results in a larger search space;
- Using pairwise swaps as genetic operator, which is actually close under this search space;
- Check individuals that contain duplicates and substitute such duplicate with the missing numbers (very expensive);
- Set all individuals that contain duplicates as having infinite fitness, so that they will be discarded in the upcoming generation (quite expensive).

Clearly, the second choice is the best choice, since it keeps the benefit of having a smaller search space while also preventing, with little cost, unwanted individuals to appear in the first place.□

It should be noted that, in certain cases, encoding-specific genetic operators or repair mechanism may actually complicate the search. This happens, for example, if the search space is *disconnected*, meaning that it's a union of dijoint subsets. Suppose that an algorithm is exploring one of these subdomains, but an optimal solution is in another subdomain. For the algorithm to reach it, it might be necessary to explore parts of such "forbidden" areas to go from one subdomain to the other.

However, if a repair mechanism is introduced, an individual that falls into one such forbidden area might be "brought back" to the subdomain of its parents, hence making it impossible for the algorithm to cross subdomains (unless making very long "jumps"). In such cases, it would be better to employ fitness penalties instead of repair mechanisms, so that an algorithm can tolerate individuals in "forbidden" regions as long as they can lead to unexplored search space subdomains.

4.3. Choosing a selection method

The basic principle of biological selection is that fit individuals have a better chance to procreate, hence passing their traits to their offspring. The "willingness" of the evolution process to tolerate sub-optimal individuals is called **selective pressure**. High selective pressure means that only individuals with a very high fitness will be (most likely) able to procreate, whereas low selective pressure means that even individuals with an average or subaverage fitness will (most likely) manage to procreate.

Of course, if the selective pressure is zero, then there is no evolution at all, since any individual can procreate regardless of their fitness. On the other hand, a selective pressure that is as high as possible will prioritize individuals that are optimals with respect to the initial population, at the expense of individuals underrepresented in the initial population which could be more promising.

When designing an evolutionary algorithm and its selection method, one must balance the **exploration** of the search space (finding promising individuals across an area as wide as possible) and the **exploitation** of the fittest individuals (operating on their genotype to yield even fitter offspring). Clearly, low selective pressure favour exploration, since many individuals can procreate and, through random mutations, can reach an area of the search space as wide as possible. On the other hand, high selective pressure favours exploitation, since only very fit individuals will be taken into account, restricting the search space to their neighborhood.

The best strategy to achieve this balance is to tune the selective pressure with respect to time, starting with low selective pressure (in order to cover the wides area possible) and then increasing it further and further along the iterations to restrict the focus on the most promising candidate solutions.

A very intuitive approach to the selection of individuals is what is called **Roulette-Wheel Selection** or **Fitness-Proportionate Selection**. Given a population P, for each individual s a relative fitness is computed as:

$$f_{\text{rel}}(s) = \frac{f_{\text{abs}}(s)}{\sum_{s' \in P} f_{\text{abs}}(s')}$$

That is, as its absolute fitness (its "standard" fitness) weighted by the fitness of all the other individuals. This relative fitness is then interpreted as the probability of obtaining said individual when sampling a random individual in the population to construct the next generation. This way, the probability of choosing any individual as a member of the next generation is directly proportional to its fitness (hence the name fitness-proportionate selection).

Note that this approach requires the problem at hand to be a maximization problem (higher fitness means better individual), because in a minimization problem (higher fitness means worse individual) the sampling would pick unfit individuals with high probability and fit individuals with low probability. In general this is not an issue, since minimization problems can be converted into maximization problems. Also, the fitness function must be non negative, otherwise one might incur in negative probabilities.

A drawback of Roulette-Wheel Selection is that some individuals may dominate the selection process. If an individual has much higher fitness that the others, it has a greater chance of being selected, which in turn means that its offspring will have higher chances, which leads to them being selected, and so on.

What might happen is what is called **crowding**, that is when the population is composed of individuals coming from a very small subset of the search space and the remaining space is underrepresented. Crowding might lead to **early convergence**, meaning that an optimum can actually be reached, but such optimum is local with respect to the (narrow) subset represented by the population.

Another pathological situation as a result of a poor choice of the selection method is the so-called **vanishing selective pressure**. This happens when the relative fitnesses of all individuals are too close to one another, therefore the probabilities of each individual to be chosen in the selection process are almost identical. In turn, this results in no individual being able to emerge over the others, essentially turning the evolutionary algorithm into no better than a random search.

Since an evolutionary algorithm tends to increase the average fitness of individuals from one generation to the next, as better individuals are selected with higher probability, it may even create the problem of vanishing selective pressure itself. As generations go by and the average fitness is increased, it might have the side effect of reducing the range of fitnesses too much (concentrating on the same optimal neighborhood), creating a bottleneck that lowers the selective pressure.

This issue ought to be mitigated, since an ideal evolutionary algorithm should do the exact opposite (start with low selective pressure and increase it over time). A popular solution to preventing both crowding and vanishing selective pressure is, before computing the relative fitness function, scaling the (absolute) fitness function. One simple approach is **linear dynamic scaling**:

$$f_{\text{lds}}(s) = \alpha \cdot f(s) - \min\{f(s') \mid s' \in P\}$$

Where $\alpha > 0$ is a tunable parameter than determines the strength of the scaling and P is the current population. Another approach is **σ-scaling**:

$$f_\sigma(s) = f(s) - \mu_f(t) + \beta\sigma_f(t) \qquad \text{with } \mu_f(t) = \frac{\sum_{s\in P(t)} f(s)}{|P|} \text{ and } \sigma_f(t) = \sqrt{\frac{\sum_{s\in P(t)} \big(f(s) - \mu_f(t)\big)^2}{|P| - 1}}$$

Where $\beta > 0$ is another tunable parameter, t is the index of the current generation, μ_f is the mean value of the distribution of the fitness functions and σ_f is its standard deviation.

Obviously, these formulas beg the question of how to find suitable values for α and β. One way to do so is to refer to the so-called **coefficient of variation** ν, defined as:

$$\nu = \frac{\sigma_f}{\mu_f} = \frac{|\Omega|\sqrt{\sum_{s'\in|\Omega|}\left(f(s') - \frac{1}{|\Omega|}\sum_{s\in|\Omega|} f(s)\right)^2}}{\sqrt{|\Omega| - 1}\sum_{s\in\Omega} f(s)} \approx \frac{\sigma_f(t)}{\mu_f(t)}$$

As denoted by the $\approx$ sign, this coefficient is rarely computed in its proper form (that is, by considering the entire search space Ω), because it would be too computationally expensive. Instead, it is estimated from the population at hand, assuming that the value of such coefficient is roughly constant across all generations.

Empirical analysis has found that a value of $\nu \approx 0.1$ yields a good tradeoff between exploration and exploitation. Therefore, if the value of ν for the population under consideration deviates significantly from 0.1, the fitness function should be tuned so that the coefficient of variation approaches 0.1.

An alternative to scaling the fitness function before sampling is to introduce a **time dependence**. That is, computing the relative fitness values not from $f(s)$ but from $g(s) = (f(s))^{k(t)}$, with $k(t)$ being an exponent (dependent on t) that modulates the selective pressure in order to obtain a value of the coefficient of variation close to 0.1. One possible choice is:

$$k(t) = \left(\frac{\nu^*}{\nu}\right)^{\beta_1} \left(\tan\left(\frac{t}{T+1} \cdot \frac{\pi}{2}\right)\right)^{\beta_2 \left(\frac{\nu}{\nu^*}\right)^\alpha}$$

Where $\nu^*, \alpha, \beta_1, \beta_2$ are tunable parameters, t is the index of the current generation and T is the maximum number of generations. To achieve $\nu \approx 0.1$, many combinations of parameters can be used: one such combination is known to be $\nu^* = \alpha = \beta_2 = 0.1$ and $\beta_1 = 0.05$.

$$k(t) = \left(\frac{0.1}{\nu}\right)^{0.05} \left(\tan\left(\frac{t}{T+1} \cdot \frac{\pi}{2}\right)\right)^{0.1\left(\frac{\nu}{0.1}\right)^{0.1}} = \sqrt[20]{\frac{1}{10\nu}} \left(\tan\left(\frac{t}{T+1} \cdot \frac{\pi}{2}\right)\right)^{\frac{\sqrt[10]{10\nu}}{10}}$$

An alternative choice of time-dependent fitness function is **Boltzmann selection**, where the relative fitness is computed from $g(s) = \exp\left(\frac{f(s)}{kT}\right)$, with k being a normalization constant and T the temperature parameter that tunes the selective pressure. The idea (similarly to simulated annealing) is to start from a high value of T in the early iterations, yielding small values of $g(s)$ (being at the denominator) and hence lowering the selective pressure. The more iterations are carried out, the more T is lowered, resulting in more appreciable fitness differences and therefore higher selective pressure.

Even though roulette-wheel selection is vary simple and strives to be fair, it presents an obvious problem: just because an individual has higher fitness (meaning an higher chance to be selected to produce the next generation), it does not mean that it will *certainly* be selected, just that it will *more likely* be selected. Hence, if each individual is allowed to reproduce the same number of times, it does not matter how fit an individual is: there will always be a chance that its genetic traits will be lost in the generations. This is also known as the **variance problem**.

A solution that is as simple as questionable to address the variance problem is to discretize the range of fitness values. Based on the mean $\mu_f(t)$ and the standard deviation $\sigma_f(t)$ of the fitness values in the population, offspring is created according to this rule:

- If $f(s) < \mu_f(t) - \sigma_f(t)$, then s will have no offspring;
- If $\mu_f(t) - \sigma_f(t) \leq f(s) \leq \mu_f(t) + \sigma_f(t)$, then s will have one descendant;
- If $f(s) > \mu_f(t) + \sigma_f(t)$, then s will have two descendants.

A better approach would be the **expected value model**, generating as many descendants for a given individual as its expected relative frequency. That is, for any individual s, the size of its offspring is $\lfloor f_{\text{rel}} \cdot |P| \rfloor$, with $|P|$ being the size of the population (of course this value has to be rounded, since individuals are discrete). The problem is that the number of individuals generated will most likely be quite small, since $\lfloor f_{\text{rel}} \cdot |P| \rfloor$ will often be 0. To compensate, one would need to apply the expected value model to obtain a first batch of individuals and then use other selection methods (like roulette-wheel selection itself) to enlarge the population size sufficiently.

A very elegant implementation of the expected value model is the **stochastic universal sampling**, which can be seen as a variant of roulette-wheel selection. It can still be seen conceptually as a roulette wheel, but with as many markers as there are individuals in the population, equally spaced around the wheel. Instead of turning the roulette wheel once for each individual to be selected (as in standard roulette-wheel selection), the roulette wheel is turned only once and each marker gives rise to one selected individual. This way, individuals with good fitness will certainly have at least one child (if not more), whereas individual with poor fitness will get no more than one child (or none at all).

An alternative approach is to employ roulette-wheel selection but decreasing the fitness of the chosen individual at each extraction by a certain amount Δf. If the fitness of an individual becomes negative, it is discarded and forbidden to have offspring for the upcoming generation. Methods for computing Δf are:

$$\Delta f = \frac{\sum_{s \in P(t)} f(s)}{|P(t)|} \qquad \Delta f = \frac{1}{k} \max\{f(s) \mid s \in P(t)\}$$

Rank-Based Selection selects individuals not with respect to their (relative) fitnesses, but with respect to their *ranks*. The individuals are sorted with respect to their fitness and thus a rank is assigned to each. Then, to each rank is assigned a probability, which is then used to select individuals using roulette-wheel selection (or one of its variants).

This way, the probability of choosing an individual is not directly related to the absolute value of their fitness, allowing one to assign probabilities to the ranks of the individuals in a more "standardized" way, without having a dependence on the fitnesses. It has the obvious disadvantage of having to sort the individuals meaning an overhead of (at best) $|P| \log(|P|)$.

In **Tournament Selection**, a subset of k individuals from the population is sampled, and then the fittest individual of the subset (the one that "wins" the "tournament") is chosen; if there were to be a tie between two individuals, one of them is chosen at random. The "winning" individual is allowed to have a descendant in the next generation, whereas the "losing" individual are shuffled back in the current population. This process is repeated until the next generation has reached the same size as the current.

In this method, the fitnesses of the individuals are also not directly coupled to the probability of the individuals to be selected, addressing the dominance problem. Since all individuals have the same probability of being chosen, the fitness of an individual only determines the chances of winning a tournament, not wheter it will be drawn into a tournament in the first place. This means that even the fittest individual has a reasonable chance of never reproducing, simply by never coming up in the samples.

Manipulating parameter k, the size of the tournament, allows one to tune the selective pressure. A large tournament size will increase the selective pressure, since there's an higher chance that very fit individuals will take part and hence kicking out unfit ones, whereas a small tournament size will increase the chances of average individuals to be drawn into tournaments with even worse ones.

It should be noted, however, that even if an fit individual manages to produce offspring, there is no guarantee that there will always be an improvement. That is, if all of its descendants (generated from applying genetic operators) have a worse fitness than their parent, then it would be little to no different than to having chosen unfit individuals in the first place.

Since evolution in evolutionary computing can at least be "tamed" (unlike biological evolution, which is unpredictable), it is possible to introduce some "protections" for fit individuals, guaranteeing that their fitness is not worsened by mutations. A very simple yet effective way to do so is what's called **elitism**: when a new generation is created, some of the fittest individuals in the current generation are transferred unchanged. This way, one has both gradual improvements that don't get lost (the best individuals found so far) and genetic diversity (the offspring of the previous population).

This form of elitism is what's called **global elitism**, to distinguish it from another form of elitism called **local elitism**. In non-elitist evolutionary algorithms, individuals in a generation are always superseded by their offspring, no matter their fitness. In local elitism, what enters in the next generation depends on the fitness of the resulting individual. That is, if the descendant(s) has better fitness that their parent(s), the descendant(s) is/are kept, whereas if the descendant(s) has worse fitness than its parent(s), the parent(s) is/are kept.

Elitism (local or glocal) ensures that an optimal solution is approached gradually and consistently, but has a downside: achieving a global optimum might actually require to discard the current population entirely and start from a completely different gene pool, but since some individuals are always kept, this might actually reduce the variability and result in early convergence to a local optimum.

Another approach for tackling crowding is given by the umbrella of **niche techniques**. One such example is **deterministic crowding**, the idea that generated offspring should always replace those

individuals in the population that are most similar. As a consequence, the local density of individuals in the search space can be limited. Of course, calculating the degree of similarity between two individuals requires a notion of distance.

A variant of deterministic crowding, which includes ideas of elitism is the following approach: in a crossover, the two parents and two children are grouped into two pairs, each consisting of one parent and one child. The guiding principle is that a child is assigned to the parent to which it is more similar. If both children happen to be assigned to the same parent, the child that is less similar is reassigned to the other parent. Ties are broken arbitrarily. From each pair the better individual is selected and passed on into the next generation. The advantage of this variant is that much fewer similarity computations are needed than in a global approach that finds the most similar individuals in the population as a whole.

Another approach is what is called **sharing**. The idea of sharing is to reduce the fitness of an individual if there are other individuals in its neighborhood. Intuitively, the individuals share the resources of a niche, that is, a region in the search space, which has a negative effect on their fitness. A possible choice for the fitness reduction is:

$$f_{\text{share}}(s) = \frac{f(s)}{\sum_{s' \in P(t)} g(d(s, s'))}$$

where d is a distance measure between individuals and g is a function that defines the shape of the niche. One such example is the so-called power law sharing:

$$g(x) = \begin{cases} 1 - \frac{x}{\rho}^{\alpha} & \text{if } x < \rho \\ 0 & \text{otherwise} \end{cases}$$

where ρ is the radius of the niche and α controls the strength of the influence that individuals in the niche have on each other.

The traits of selections methods are grouped below:

- **Static** or **Dynamic**. In the first case, the probability of selection remains constant across the generations. In the second case, the probability of selection changes (ideally, increasing) from generation to generation;
- **Extinguishing** or **Preservative**. In the first case, probabilities of selection may be 0. In the second case, all probabilities of selection must be greater than 0 (note that this does not mean that no individual can go exstinct, just that there's at least a chance of it not happening);
- **Pure-bred** or **Under-bred**. In the first case, individuals can only have offspring in one generation (hence lower crowding). In the second case, individuals are allowed to have offspring in more than one generation (hence higher crowding);
- **Right** or **Left**. In the first case, all individuals of a population may reproduce (higher exploitation). In the second case, the best individuals of a population may not reproduce (higher exploration, since premature converge is mitigated);
- **Generational** or **On the fly**. In the first case, each generation is created in batches. In the second case, offspring continuously replaces individuals in the population as long as they are created.

4.4. Choosing a genetic operator

Genetic operators introduce variability into the genetic pool of the population, with individuals with a genotype that (potentially) has never been seen so far that allow the algorithm to move around in the search space. The simplest genetic operators are **mutation operators** (also called **variation operators**), that generate a new individual from a single individual.

Mutation operators that rely on substituting one or more alleles of an individual with random (compatible) alleles are called **standard mutations**. The simplest standard mutation operator is the **bit mutation**, that operates on solution encodings of binary strings (arrays of 0s and 1s). For each bit of the encoding, the bit mutation flips its value (from 0 to 1 or from 1 to 0) with a given probability p_m:

BIT-MUTATION(S, p_m):
1 **for** $i = 1$ **to** $|S|$
2 $\quad u \leftarrow$ a value sampled from $U \sim (0, 1)$
3 $\quad$ **if** $(u \leq p_m)$
4 $\quad\quad S_i \leftarrow 1 - S_i$

Empirically, choosing $p_m = 1/|s|$ has been shown to give the most promising results.

A variant of bit mutation is **n-bit mutation**, where instead of bit flipping each value with a certain probability, $1 \leq n \leq |s|$ bits are chosen at random and flipped:

N-BIT-MUTATION(S, n):
1 $X \leftarrow$ empty array
2 **for** $i = 1$ **to** $|s|$
3 $\quad X_i = i$
4 $X \leftarrow$ the first n elements of X
5 $X \leftarrow$ random permutation of X
6 **foreach** x_i **in** X
7 $\quad S_{x_i} \leftarrow 1 - S_{x_i}$

In particular, when $n = 1$, the operator is referred to as **one-bit mutation**.

When the encoding of the solution is an array of real numbers instead of bits, **Gaussian mutation** is often employed. In Gaussian mutation, each element of the array (of the solution) is shifted by a random different value sampled from a normal distribution $N \sim (0, \sigma)$, with σ parameter to be chosen:

GAUSSIAN-MUTATION(S, σ):
1 **for** $i = 1$ **to** $|S|$
2 $\quad \nu \leftarrow$ a value sampled from $N \sim (0, \sigma)$
3 $\quad S_i \leftarrow S_i + \nu$
4 $\quad S_i \leftarrow \max\{S_i, l_i\}$
5 $\quad S_i \leftarrow \min\{S_i, h_i\}$

Where l_i and h_i are, respectively, the lower and higher bound (if they exist) of the allowed range of values for S_i. These ensure that Gaussian mutation is closed under the search space.

Gaussian mutation employs the same parameter σ for all chromosomes of the search space. A more refined variant of Gaussian mutation is **Self-adaptive Gaussian mutation**, where each chromosome S has its own standard deviation parameter S_σ and, with each mutation, the parameter itself is tuned:

SELF-ADAPTIVE-GAUSSIAN-MUTATION(S, σ_S):
1 $u \leftarrow$ a value sampled from $U \sim (0, 1)$
2 $\sigma_S \leftarrow \sigma_S \cdot \exp\left(u/\sqrt{|S|}\right)$
3 **for** $i = 1$ **to** $|S|$
4 $\quad \nu \leftarrow$ a value sampled from $N \sim (0, \sigma_S)$
5 $\quad S_i \leftarrow S_i + \nu$
6 $\quad S_i \leftarrow \max\{S_i, l_i\}$
7 $\quad S_i \leftarrow \min\{S_i, h_i\}$

In this way, the parameter σ_S is itself subject to evolutionary pressure. In other words, the individuals with a "good" value of σ_S, meaning a value that causes suitable "jumps" across the search space, will outmatch those with a "bad" value of σ_S, and the distance travelled in the search space adapts itself (hence the name).

A different class of mutation operators are the so-called **transposition operators**, that rely not on substituting alleles with new values, but instead on rearranging the position of the alleles without changing their values. Among those are:

- The **swap operator**, that exchanges the position of two alleles (placing the value of the first as the value of the second and vice versa);
- The **inversion operator**, that reverses the order of a subset of contiguous alleles;
- The **shift operator**, that moves an entire list of genes into an insertion point;
- The **arbitray permutation**, where a subset of alleles are shuffled at random.

Clearly, such operators can be applied safely only if the exchanged alleles can have the same values, otherwise the resulting individual would fall outside the search space and appropriate countermeasures would have to be taken. In particular, they should be considered when the solution encodings are permutations of numbers (like the Travelling Salesman Problem), since rearranging any permutation still gives a permutation, and therefore said operators will certainly be closed under the search space.

Genetic operators that involve two parents are referred to as **crossover operators**. The simplest crossover operator is **one-point crossover**, where a random cut point is chosen and the first section of the two operators are exchanged. That is, given two chromosomes S_1 and S_2, the first c alleles of S_1 are swapped with the first n alleles of S_2, with c chosen randomly:

ONE-POINT-CROSSOVER(S_1, S_2):
1 $c \leftarrow$ a random value **in** $\{1, 2, ..., |S_1| - 1\}$
2 **for** $i = 0$ **to** $c - 1$
3 $\quad t \leftarrow S_{1,i}$
4 $\quad S_{1,i} \leftarrow S_{2,i}$
5 $\quad S_{2,i} \leftarrow t$

One-point crossover is an example of a genetic operator that suffers from what's referred to as **positional bias**. A genetic operator is said to possess positional bias if the way that the genes are arranged in the chromosome influences the probability of them being inherited by the offspring. That is, even if single genes have a random chance of being inherited, groups of genes may be more or less likely to be inherited "in batch" depending on the position that they occupy in the chromosome. Positional bias is problematic because particular combinations of genes that could be valuable can be lost in the generations simply due to their reciprocal position.

The reason why one-point crossover exhibits positional bias is obvious: even though the cutting point is chosen at random, hence all genes taken by themselves have the same probability to be exchanged, the probability of two or more genes to be exchanged together depends on how close they are. This is because for two or more genes to undergo exchange together the cutoff point must not be between them, but to the left or to the right of both, and this depends on how much far apart they are. In the extreme case of two genes being at the opposite side of the chromosome, it is guaranteed that they will never undergo exchange together, since any cutoff point will separate them. On the other hand, two neighboring genes will undergo exchange together for all choices of cutoff points except one, making such event very likely.

A straightforward extension of one-point crossover is **two-point crossover**, where the section between two points is exchanged. That is, given two chromosomes S_1 and S_2 and two random cutoff points a and b (with $a < b$), the alleles $S_{1,a}, S_{1,a+1}, ..., S_{1,b}$ are swapped with the alleles $S_{2,a}, S_{2,a+1}, ..., S_{2,b}$; the first $a - 1$ and the last $b + 1$ alleles are left intact.

Even more generally, one-point crossover is extended to **n-point crossover**, where n cutoff points are chosen and the $n - 1$ subsequences are alternatingly exchanged and not exchanged. That is, given n cutoff points $c_1, c_2, ..., c_n$, the first c_1 alleles are exchanged, the alleles between c_1 and c_2 are kept intact, the alleles between c_2 and c_3 are exchanged, ecc...

Instead of randomly choosing cutoff points, **uniform crossover** follows another approach: each gene x of the pair of chromosomes is swapped with a probability p_x.

```
UNIFORM-CROSSOVER(S_1, S_2, (p_1, ..., p_n)):
for i = 0 to |S|
    u ← a value sampled from U ~ (0, 1)
    if (u < p_i)
        t ← S_{1,i}
        S_{1,i} ← S_{2,i}
        S_{2,i} ← t
```

Uniform crossover suffers from what's called **distributional bias**, which means that the probability that a certain number of genes will undergo exchange depends on the number itself. Distributional bias is an undesirable property (even though not as much as positional bias) because it means that subchromosomes of certain lengths will undergo exchange more or less likely.

Uniform crossover exhibits distributional bias because each gene is exchanged with a given probability p_x (dependent on the gene) and each choice is independent of the others. This means that the number of exchanged genes is binomially distributed, and a binomial distribution has a probability mass function that yields higher values for low and high inputs. That is, under uniform crossover, it is much more likely that either very small or very large portions of the chromosome(s) undergo exchange, whereas exchanges of moderate length are less likely.

Interestingly, even though it suffers from positional bias, one-point crossover does not suffer from distributional bias. This is because the choice of any cutoff point is equally likely and the entire subchromosome is exchanged, so all lengths are equally likely.

A slightly different operator is **shuffle crossover**, where the two chromosomes are shuffled at random, any crossover operator is applied and then they are shuffled again. The difference between the two lies in the fact that, while in uniform crossover the number of exchanged genes is binomially distributed (depending on p_x), in shuffle crossover the choice of any number of exchanged genes is equally likely. Shuffle crossover is an interesting choice, since it exhibits neither positional bias nor distributional bias.

All of the crossover operators presented so far could not be employed if, for example, the solution is encoded as a permutation, since merging two permutations does not guarantee to result in a permutation. There are crossover operators that are indeed closed under the search space of permutations, such as **uniform order-based crossover**.

This operator determines, like uniform crossover, for each allele whether it should be exchanged with a given probability p_x. However, instead of exchanging the designated alleles with their counterpart in the other chromosome, the designated alleles in a chromosome are exchanged with the alleles in the other chromosome that in the first chromosome are missing, and vice versa:

$$\underline{\text{UNIFORM-ORDER-BASED-CROSSOVER}}(S_1, S_2, (p_1, ..., p_n)):$$

As the name hints, uniform-order-based-crossover is order-preserving, since the ordering in which the values of the original alleles are found is the same. Specifically, the alleles that are not exchanged remain in the same place (hence trivially preserving their order) whereas the new values for the exchanged alleles are ordered in the same way as in the original chromosome.

A different permutation-preserving crossover operator is the so-called **edge recombination**. In this method, designed specifically for tackling the Travelling Salesman Problem, the alleles are interpreted as a graph, where each allele is connected to its neighbors by an edge, including the first and the last.

The first step in applying edge recombination is constructing a table, called **edge table**. The i-th entry of the table contains the neighbors of the i-th allele (the first and last allele are connected, so they do count as neighbors), taking both chromosomes into account. If a value in an entry happens to appear twice, meaning that the allele has the same neighbor in both chromosome, it is listed only once, but "marked" to denote that is has to be treated specially. The order of the neighbors in each entry is not relevant, but they are often sorted for readability.

The second step is to employ the edge table to construct a new individual out of the original two. This is done as follows:

1. If this is the first iteration, pick the value of the first allele in any of the two parents;
2. If this is not the first iteration, pick an allele in one of the following methods out of the neighbor list at hand, ordered by preference:
   - Marked neighbors;
   - Neighbors with the shortest neighborhood list;
   - Any neighbor;
   - Any allele that hasn't been chosen yet.

   If there are more candidates in the same tier, choose one at random;
3. Delete the chosen allele in all entries of the edge table;
4. Append the deleted allele to the chromosome of the new individual;

5. If the table does not contain any entry, stop. Otherwise, choose as new neighbor list the one of the allele that has been just deleted and restart the algorithm.

Exercise 4.4.1: Apply edge ricombination to the chromosomes $A = (6, 3, 1, 5, 2, 7, 4)$ and $B = (3, 7, 2, 5, 6, 1, 4)$.

Solution: The entries of the edge table are as follows:

1	2	3	4	5	6	7
$3, 4, 5, 6$	$5^*, 7^*$	$1, 4, 6, 7$	$1, 3, 6, 7$	$1, 2^*, 6$	$1, 3, 4, 5$	$2^*, 3, 4$

Where the neighbors marked with $*$ are those that appear in both chromosomes for the same allele.

Allele	Iter. 0	Iter. 1	Iter. 2	Iter. 3	Iter. 4	Iter. 5	Iter. 6	Iter. 7
1	$3, 4, 5, 6$	$3, 4, 5$	$3, 4$	$3, 4$	$3, 4$	3		
2	$5^*, 7^*$	$5^*, 7^*$	7^*	7^*				
3	$1, 4, 6, 7$	$1, 4, 7$	$1, 4, 7$	$1, 4, 7$	$1, 4$	1	1	
4	$1, 3, 6, 7$	$1, 3, 7$	$1, 3, 7$	$1, 3, 7$	$1, 3$	$1, 3$	1	
5	$1, 2^*, 6$	$1, 2^*$	$1, 2^*$	1	1	1	1	
6	$1, 3, 4, 5$	$1, 3, 4, 5$	$1, 3, 4$	$1, 3, 4$	$1, 3, 4$	$1, 3$	1	
7	$2^*, 3, 4$	$2^*, 3, 4$	$2^*, 3, 4$	$3, 4$	$3, 4$	3		

1. In the first iteration, 6 is chosen, since it's the first allele of the first chromosome;
2. In the second iteration, the choice is among $(1, 3, 4, 5)$, the neighbors of 6. None of them is marked, so the choice is done based on neighborhood length: 5 is chosen, since it has the shortest neighborhood list;
3. In the third iteration, the choice is among $(1, 2^*)$, the neighbors of 5. 2 is chosen, since it's marked;
4. In the fourth iteration, the choice is 7, since it's marked (and it's the only member of the neighbor list of 2);
5. In the fifth iteration, the choice is among $(3, 4)$, the neighbors of 7. None of them is marked, so the choice is done based on neighborhood length: since both 3 and 4 have two neighbors, 4 is randomly chosen as tie breaker;
6. In the sixth iteration, the choice is among $(1, 3)$, the neighbors of 4. None of them is marked, so the choice is done based on neighborhood length: since both 1 and 3 have two neighbors, 3 is randomly chosen as tie breaker;
7. In the seventh iteration, the choice is 1, since it's the only allele left.

Which means that the resulting individual is $C = (6, 5, 2, 7, 4, 3, 1)$. □

The precedence rules for the choice of the next allele guarantees that, whenever possible, an allele present in the neighborhood of both parents are favoured. Alleles with short neighbor lists are preferred over alleles with long neighbor list in order to delay the use of the two remaining choices as long as possible. The rationale is very simple: short neighbor lists run a higher risk of becoming empty due to allele selections, so one should choose from them earlier than from longer lists. Introducing new edges is discouraged since the principle of small improvements is lost.

Crossover operators can be extended from two parents to three or more parents. One such operator is **diagonal crossover**, that can be understood as a generalization of one-point crossover. Given a set of k parents, arranged in some order, $k - 1$ distinct cutoff points are chosen. Then, each i-th section of the chromosome is shifted cyclically across the chromosomes. That is, for each chromosome: the first section is not shifted (shifted to itself), the second section is shifted to the next chromosome, the third section is shifted to the next next chromosome, ecc...

As stated already, recombination operators merge the traits of two individuals into one, obtaining a new potential solution for exploring the search space. However, they cannot create alleles that are not present in either parents (that's why mutation operators are used). Therefore, they can be used effectively on their own only if the starting population has sufficient genetic diversity to ensure that as much of the search space can be covered without introducing new alleles but just by combining the existing ones.

There are, however, recombination operators that combine the traits of the parents in such a way that the resulting individual, even though having a genotype determined by its parents, possesses new alleles that neither parent had. An example of such an operator is **interpolating recombination**, which blends alleles of the parents with a randomly chosen mixing parameter. A more concrete example for chromosomes that are real-valued arrays is **arithmetic crossover**, which can be seen as interpolating between the n-dimensional points that are represented by the parent chromosomes.

ARITHMETIC-CROSSOVER(S_1, S_2):
1 $R \leftarrow$ empty array
2 $u \leftarrow$ a value sampled from $U \sim (0, 1)$
3 **for** $i = 1$ **to** $|S_1|$
4 $\quad R_i \leftarrow u \cdot S_{2,i} + (1 - u) \cdot S_{1,i}$

It should be noted that abusing such blending operators might result in a loss of genetic diversity, since they tend to "even out" the differences and converge to a genetic mean value (the so-called **Jenkins nightmare**). Therefore, arithmetic crossover should be balanced by a mutation operator that ensures genetic diversity.

4.5. Improving performance through parallelization

4.5.1. Parallelizing creation, selection and mutation

Evolutionary algorithms, despite their promising results, are computationally expensive, since at each iteration they have to handle not just one solution, but entire populations of considerable size. Hence, to have them being useful, it is mandatory to address the problem of high computational cost.

Unlike neural networks, where the GPU could be exploited to offload part of the calculations, with evolutionary algorithms this is hardly of any help. This is because GPUs are optimized for operations such as weighted sums and convolutions, which coincidentally are used both for rendering and for neural network training. On the other hand, evolutionary algorithms require much more sofisticated computations.

Slow execution time can be mitigated to some extent by parallelization, meaning that on the computer the inner workings of each step (selection, mutation, crossover) are run in parallel on different processors. Of course, it is not possible to run the three steps themselves in parallel, since they depend on each other.

Creating the initial population is a trivially-parallelizable task, since each individual is randomly generated independently of each other. There could be an issue of overrepresentation, since if each parallel computation generates a subset of the population independently of the others, the same individual may appear in more than one subset, giving them an edge. However, if the population is very large, this is hardly relevant. Also, since creating the initial population is done only once, introducing extra checks to ensure that no duplicated individuals is most-likely an over-engineered solution.

Computing the fitness of individuals is also trivially-parallelizable for the same reason. Genetic operators are trivially-parallelizable too, not just those that involve a single parent (one-bit-mutation, ecc...), but also those that involve two or more parents, since the original parent is discarded and no race condition arises.

The same cannot be said for selection. This is because most selection methods rely on the relative fitness of the population, which to be computed requires the absolutive fitness of the entire population. Selection methods such as roulette-wheel selection and rank-based selection can be parallelized after the relative fitness is computed (in a non-parallel way), since each extraction can be done independently of the others.

Expected value model and elitism are much harder to parallelize, since the selection process itself has to take into account the entire population. A compromise solution consists in partitioning the population and sorting each partition in a different parallel unit; the result of each partial sorting is then itself sorted by a central unit. In any case, both selection methods cannot be trivially-parallelizable in any way.

Tournament selection, on the other hand, is trivially parallelizable in its entirety, since each tournament can be carried out independently of the others. For this reason, many real-world implementations of evolutionary algorithms use tournament selection as selection method.

Some termination criteria can be trivially-parallelized. For example, the criterion “stop after a given number of iterations” is not problematic. However, criteria such as “stop when the fitness of an individual has reached a boundary” or “stop when the improvement from the previous generation is negligible” are not, since in both cases the fitnesses of each parallelized run have to be merged to be inspected by a central agent.

4.5.2. The island model

Even though some selection criteria can be troublesome to be parallelized, this does not mean that they have no use. Indeed, some selection methods are better suited than others depending on the problem at hand. Also, it's still possible to "mimic" parallelization simply by running the evolutionary algorithm with many populations at the same time, each on its dedicated processor, then merging the results selecting the best individual among all populations. Each population can be thought of as inhabiting its own "island", which explains the name **island model** for such an architecture.

Running many instances of the algorithm at the same time is the simplest form of island model, also called **pure island model**. As a matter of fact, aside for the performance improvement, this is conceptually no different than running the same algorithm many times in a serial fashion. Also, running many instances of the algorithm with smaller population sizes actually yields worse performance than running the algorithm just once but with a larger population size.

The island model can be extended much further, however. Instead of just having the islands as separate and independent instances of the algorithm, a variant of the island model suggests transfering individuals from one island to another every k iterations (with k tunable parameter). Again drawing on an obvious analogy from nature, such an approach is commonly called **migration**. Migration allows for the increase in genetic variety of the islands, since they will most likely be different from each other, leading to a better collective exploration of the search space.

Regarding the method used to choose which islands should have a migration, the simplest one is the **random model**, where any too islands are choosen at random, no matter their characteristic. A more restrictive approach is the **network model**, where islands are arranged in a graph or a lattice, usually in a squared or hexagonal grid, and migration happens only between two neighboring islands. A completely different model is the **contest model**, where islands do not cooperate transfering individuals but instead compete: each island has its own choice of parameters for the algorithms, and the population of each island is either increased or decreased every k iterations based on the average fitness of their individuals. It is advisable to introduce a lower bound on the island's size, so that an island cannot become empty.

4.5.3. Cellular evolutionary algorithms

Related to the island model, **cellular evolutionary algorithms** (**cEAs**) are a form of parallelization that is also called "isolation by distance". cEAs work with a large number of (virtual) processors, each handling a single individual (or a small number of individuals). The processors are arranged in a rectangular grid, usually in the shape of a torus in order to avoid boundary effects. Selection and crossover can happen only between neighbors, that is, with processors connected by an edge of the grid. Selection means that a processor chooses the best chromosome of the (four) processors adjacent to it (or one of these chromosomes randomly based on their fitness). The processor then performs crossover of the selected chromosome with its own. The better child resulting from such a crossover replaces the chromosome of the processor (local elite principle). A processor may also mutate its chromosome, the result of which, however, replaces the old chromosome only if it is better (local elite principle again). In such an architecture, groups of adjacent processors are created that maintain similar chromosomes. This mitigates the usually destructive effect of crossover.

4.6. Classes of evolutionary algorithms: evolutionary local search

Some local search algorithms are directly influenced by evolutionary algorithmic approaches. That is, they combine the elements of local search (search space, moving from a candidate solution to its neighbors, ecc...) and elements of evolutionary algorithms (populations, random mutations, ecc...)

4.6.1. Tabu search

Tabu search[15] is a variant of local search where the creation of a new solution candidate depends on the history of previous candidates. That is, if the new candidate solution is identical to a previous candidate that is known to be suboptimal, it is discarded immediately. This way, the algorithm avoids entering paths that were already attempted and is incentivized to try out new ways.

Tabu search does not consider one solution at a time, working instead on a population of individuals. The previous candidates are stored in a list, called **tabu-list**. This list is a FIFO (first in first out) and has a fixed lenght. Whenever a new solution candidate is chosen among the population, it is added to the list; if the list has reached its maximum capacity, the first element is removed from the list. This way, a solution candidate that has been evaluated recently will be in the list, hence unavailable, but after enough iterations a solution previously added to the list can become available again.

The algorithm, presented as follows, requires a a termination criteria and a parameter λ, which controls the size of the population:

```
TABU-SEARCH(f : ℝ^n → ℝ, λ, ε):
 1  A ← RANDOM-INDIVIDUAL ()
 2  A_max ← A
 3  T ← INIT-TABU-LIST ()
 4  do
 5      P ← ∅
 6      do
 7          B ← MUTATE (A)
 8          if ((A, B) ∉ T ∨ f(B) > f(A_max))
 9              P ← P ∪ {B}
10      while (|P| < λ)
11      A_old ← A
12      A ← max_f{P}
13      if (f(A) > f(A_max))
14          A_max ← A
15      if (MAXIMUM-CAPACITY (T))
16          POP (T)
17      PUSH ((A_old, A), T)
18  while (not(ε))
19  return A_max
```

[15] The name "tabu search" comes from the word "taboo", meaning "forbidden".

4.6.2. Memetic algorithms

Memetic algorithms try to take advantage of the benefits of evolutionary algorithms (many solutions explored at the same time) and those of local search algorithms (speed) but trying to mitigate the first (wastefulness) and the second's (susceptibility to local optima) downsides. The name comes from the biological concept of *meme*, an element of the behaviour that can be acquired from experience (in contrast to genes, that are hereditary).

The algorithm starts with a randomly generated population, then applies a local search algorithm to each individual of the population to find the candidates that are most promising. Then, at each step of the main loop a subset of the population that is deemed fit is chosen to repopulate, substituting the old population with the offring of said chosen few after having applied genetic operators to improve their fitness. This way, over time, individuals that are already fit will be superseded by even fitter ones.

Even though this approach is almost guaranteed to be very fast, has the issue of potentially limiting the search space too much. Also, the choice of the starting solution has a huge impact on the outcome, since each generation is heavily dependent on the previous.

4.6.3. Differential evolution

Differential evolution tries to exploit the relationships that intercur between solutions.

4.6.4. Scatter search

Scatter search employs populations of selection candidates that are subject to evolutionary pressure exploring their neighborhood, but as the name suggests tries to "jump" around the search space trying to cover an area as wide as possible.

4.6.5. Cultural algorithm

4.7. Classes of evolutionary algorithms: swarm intelligence

All the algorithms presented up to now relied on the performances and fitnesses of single individuals, figthing for survival in their environment. That is, each individual acts alone in its best interest, there is no form of communication or "workload sharing" between them. However, a different, more "collegial" approach for solving optimization problems is possible, relying on the biological phenomenon called **swarm intelligence**.

Swarm intelligence is a phenomenon present in many species of social animals, like fishes, birds or ants, where a group of animals (the school, the flock, the colony, ecc...), even if constituted by individuals having limited intelligence or capabilities, considered as a whole exhibits a remarkably complex behaviour. This "distributed" intelligence is not mediated by a central coordinator: the individuals regulate themseleves simply by communicating between neighbors. This "whole being greater than the sum of its parts" aspect is also referred to as *emergence*.

Genetic algorithms inspired by swarm intelligence are still constituted by populations of individuals, with the goal being solving an optimization problem, but don't take into account the fitness of the single individuals. Instead, rely on such individuals to cooperate and share information in order to achieve a common objective.

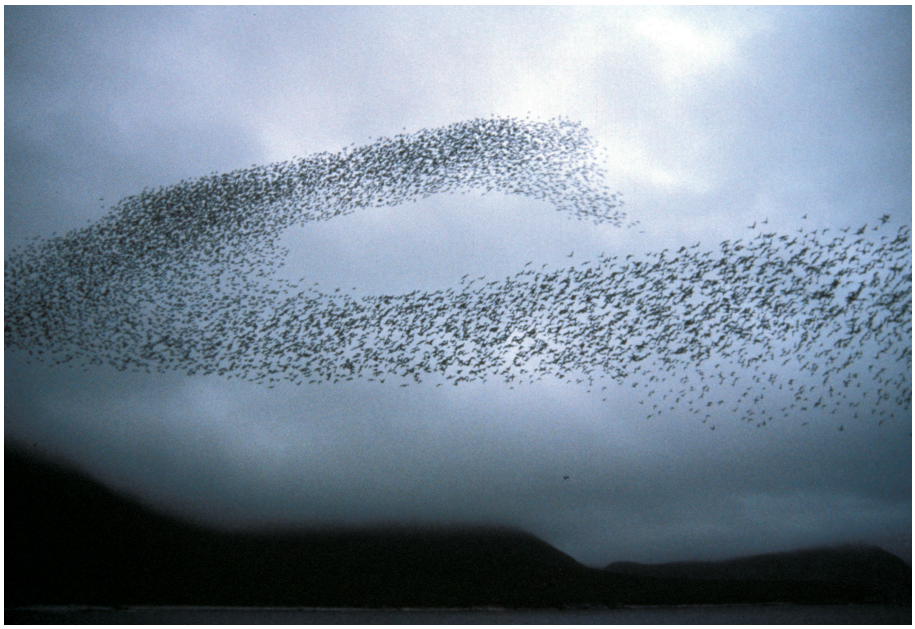


Figure 70: The complex patterns and unison movements of flock of birds is an example of swarm intelligence. [Original image under public domain.]

Such algorithms can be grouped into two broad categories: those that achieve information sharing between direct communication and those that do so by manipulating the (ficticious) environment of the individuals. An example of the former is **Particle Swarm Optimization**, an example of the latter is **Ant Colony Optimization**.

4.7.1. Particle Swarm Optimization

A very generic model of swarm intelligence using direct communication between individuals to achieve the swarm behaviour is as follows. Consider a population of N individuals, moving in a n-dimensional search space $\Omega \in \mathbb{R}^n$. Their position and velocity depends both on a time variable, that can be assumed to be discrete, and on the reciprocal position between individuals.

Ω represents the possible positions in which an individual can be found: the position of the i-th individual at time t is given by $\big(x_{1,i}, x_{2,i}, ..., x_{n,i}\big)(t) = \boldsymbol{x}_i(t) \in \Omega$. The velocity of the i-th individual at time t is given by $\big(v_{1,i}, v_{2,i}, ..., v_{n,i}\big)(t) = \boldsymbol{v}_i(t)$.

Individuals travel across the search space, adjusting their position and/or velocity at each time frame following three behavioural patterns:

- **Cohesion**. If an individual is too far away to its neighbors, will try to reach them by moving close by. To accomplish it, the individual computes the distance between itself and its neighbors; if this distance is greater than a given threshold d, it adjusts its position and/or velocity to lessen the distance;
- **Separation**. If an individual is too close to its neighbors, will try to keep its distance. If the distance between the individual and its neighbors is smaller than d, it adjusts its position and/or velocity to widen the distance;
- **Alignment**. Each individual will try to adjust its direction in order to move in the same average direction as its neighbors.

Of course, cohesion and separation are possible only if there's a way to define a distance for the search space (that is, if Ω is a metric space);

For simplicity, exclude the alignment behaviour and focus on cohesion and separation. The equations for position and velocity of the i-th individual of the population are defined as:

$$\boldsymbol{x}_i(t) = \boldsymbol{x}_i(t-1) + \boldsymbol{v}_i(t) \qquad\qquad \boldsymbol{v}_i(t) = w(t) \cdot \boldsymbol{v}_i(t-1) + \boldsymbol{g}_i$$

Where w and $\boldsymbol{g}_i$ are two constants: w indicates the so-called *inertia weight*, whereas $\boldsymbol{g}_i$ is the (positive or negative) contribution of the cohesion and separation behaviour of the individual. While w is the same for all individuals of the population, $\boldsymbol{g}_i$ can be different for each individual. Many choices of $\boldsymbol{g}_i$ are possible: for each individual, $\boldsymbol{g}_i$ is given by the sum of all contributions of the function for each other individual in the swarm.

Particle Swarm Optimization (PSO) is a particular application of swarm intelligence, where the swarm is instructed to move in the search space trying to optimize a given function. The approach is inspired by the collective behaviour of groups of animals, that share information on the location of food in their vicinities. It combines gradient-based search with population-based search; for this reason, the function to be optimized must be $f : \mathbb{R}^n \to \mathbb{R}$.

PARTICLE-SWARM-OPTIMIZATION$(f : \mathbb{R}^n \to \mathbb{R}, N, w, C_1, C_2, \varepsilon)$:

```
1  for i = 1 to N
2      v_i ← 0                                              // Initial velocity
3      x_i ← a random element of the search space Ω         // Initial position
4      x*_i ← x_i                                           // Locally known optimum
5  opt ← x_1                                                // Globally known optimum
6  for i = 2 to N
7      if f(x_i) ≥ f(opt) then
8          opt ← x_i
9  do
10     for i = 1 in to N
11         if f(x_i) ≥ f(x*_i) then
12             x*_i ← x_i                                   // Update local optimum
13         if f(x_i) ≥ f(opt) then
14             opt ← x_i                                    // Update global optimum
15     for i = 1 to N
16         φ_1 ← a value sampled from U ∼ (0, 1)
17         φ_2 ← a value sampled from U ∼ (0, 1)
18         v_i ← w v_i + φ_1 C_1 (x*_i − x_i) + φ_2 C_2 (opt − x_i)   // Update local velocity
19         x_i ← x_i + v_i                                  // Update local position
20 while (not(ε))
21 return opt
```

The idea is to have a population (a "swarm") of individuals that move in the search space and collaborate in finding an optimal solution by sharing information. Indeed, each i-th individual knows both its current position $\boldsymbol{x}_i$ and its current velocity $\boldsymbol{v}_i$, but also keeps track of the best solution $\boldsymbol{x}^*$ that he has found so far. Every member of the population also knows about the global solution opt, which is the best solution among the best local solution recorded by each individual:

$$\text{opt} = \boldsymbol{x}_M \quad \text{with} \quad M = \arg\max_{i=1}^{N} f(\boldsymbol{x}_i^*)$$

The global solution denotes the position of the particle that is the closest, in the current time frame, to the optimal solution. The global solution represents the "social" part of the swarm, since to be computed each individual has to share its local knowledge of the optimality with the other individuals. In the simplest case (the one presented above) each individual is able to communicate to all other individuals, but one could device a more constrained model where, for example, each individual can communicate with no more than $k < N$ individuals.

The parameters θ_1 and θ_2 are chosen at random at every iteration, and represent the strength of the influence that, respectively, the personal and global memory exert on the velocity of a particle. The parameters C_1 and C_2, also called **learning factors**, define the amount of linear attraction and have to be hand-picked instead. The inertia weight determines the size of the step iterations: small values induce a quick convergence to a local optima, whereas high values slow down convergence (or make it impossible). To tackle this exploration versus exploitation tradeoff, the same approach can be used:

starting with large values of w so that the space is thoroughly explored and slowly decreasing it over time to close in on an optimum.

PSO can be tuned further by introducing a **turbulence factor**, a mechanism that attempts to prevent a premature convergence to suboptimal solutions by forcibly introducing randomness. A very simple turbulence factor consists in choosing a random particle at each iteration and changing its position to a random one. A more refined turbulence factor is to change the position of slow particles (with velocity lower than a given threshold) with a random position, in the hope that this will speed them up and contribute more consistently in the exploration of the search space.

4.7.2. Ant Colony Optimization

Ant Colony Optimization is a mathematical optimization technique inspired by the behaviour of certain species of ants. These ants are blind, but are still able to communicate with each other if they happen to find a source of food close to their nest. This is possible because, on their way forward and backward to the food source, they deposit **pheromones**: when an ant smells another ant's pheromones, they will be inclined to follow the same trail, since it's deemed to be "safe". This creates a virtuous cycle: more ants smell each other's pheronomones, which leads them to follow this path, depositing even more pheromones, ecc... This behaviour is also called **stimergy**.

This behaviour can be quickly adapted to solve a mathematical optimization problem by observing how stimergy allows ants to find the shortest path in an environment without having any model of it. This is because each ant deposits pheronomones both on the way to the food source and on the way back from the food source. Assuming that all ants move at the same speed, in a given time interval the ants that have gone back and forth from the nest to the food source following the shortest path (known to them) will have done so more times than ants that have followed a longer path. But this means that the shortest path will have more pheromone layed onto it, which in turn will prompt other hands to follow this path, laying more pheromones, ecc... This means that, after a given time, the shortest path from the nest to the food source is the one where the ants have deposited the greatest amount of pheromone.

Note that the shortest path is found only if the ants deposit pheromone in both directions, from the nest to the source and from the source to the nest. This is because all ants move one after the other and the amount of pheromone deposited in all paths is mostly the same, equalizing each other. This means that the choice of a path will eventually converge, but not necessarely to the shortest one. Also, all paths must exist from the beginning, the behaviour of the ants leads them to find a path that "incidentally" also happens to be the shortest among the existing ones. If a new path is added, even if shorter than any existing path, the ants have no interest in trying it out, since the path that they have traced so far "just works".

Simulating ants and their stimergy can be used to solve graph-related problems, like the problem of finding the shortest path in a graph: each simulated ant traverses the graph and increases an attribute (the amount of pheromone) of the edge they traverse; the probability with which a certain path is traversed is proportional to the amount of pheromone that has been deposited on the path so far.

It should be noted, however, that this approach has to take into account the presence of cycles. A cycle, from the point of view of an ant traversing the graph, is no different than traversing the entire graph. This issue can be circumvented by depositing the pheromone only after the entire graph has been traversed. In addition, before pheromone is deposited, any cycles that a path may contain are removed.

Another issue, already hinted at before, is that the ants will try to stick the first (or one of the first) solution that they find, converging prematurely to a sub-optimal solution. This can be tackled by

letting the amount of pheromone "decay", or "evaporate"[16], over time, so that the ants are incentivised to try other solutions as well. An even more refined approach could entail tuning the amount of deposited pheromone with respect to the quality of the solution, or on the weight of the edge.

Ant Colony Optimization can be used to solve the Travelling Salesman Problem. The two main data structures are the adjacency matrix D of the graph and a matrix Φ, where each entry $\Phi_{i,j}$ contains the amount of pheromone on the (i, j) edge. By default, all $\Phi_{i,i}$ are set to 0 and all other entries are initialized with an arbitrary starting value.

The graph is traversed by the ants one by one, leaving pheromones on the path traversed when all nodes have been reached. To avoid having an ant traverse the same node twice, each ant is endowed with a memory C that contains all the nodes that have been reached so far[17]. Each ant starts in a random node, then moves from node to node with a certain probability that (also) depends on the amount of pheromone in the edges at each until all nodes are reached. After the entire Hamiltonian cycle is constructed, the pheromone matrix is updated with the newly deposited pheromone. Each time μ ants have traversed the graph, the pheromone matrix is evaporated employing an *evaporation factor* η.

[16]Even though, as any chemical marker, pheromones do evaporate over time, this has little influence in the real world behaviour of ants.

[17]This has also no real world counterpart.

ANT-COLONY-OPTIMIZATION$(Q, W, \mu, \eta, c, \varepsilon)$:

```
1  n ← |W|
2  Φ ← empty n × n matrix
3  s ← a generic starting value
4  for i = 1 to n
5      for j = 1 to n
6          Φ_{i,j} ← s
7  π* ← (1, ..., n)
8  Q(π*) = c · (Σ_{i=1}^{n} W_{π*(i),π*((i mod n)+1)})^{-1}
9  iteration ← 0
10 do
11     iteration ← iteration +1
12     C ← {1, ..., n}
13     t ← a random element in C
14     π ← (t)
15     C ← C \ {t}
16     while C ≠ ∅
17         P ← empty array of |C| elements
18         for i = 1 to n
19             P_i ← Φ_{t,i} / Σ_{j∈C} Φ_{t,j}
20         t' ← next node chosen at random weighted by P
21         π.append(t')
22         C ← C \ {t'}
23         t ← t'
24     Q(π) = c · (Σ_{i=1}^{n} W_{π(i),π((i mod n)+1)})^{-1}
25     for i = 1 to n
26         Φ_{π(i),π((i mod n)+1)} ← Φ_{π(i),π((i mod n)+1)} + Q(π)
27     if (Q(π) > Q(π*))
28         π* ← π
29         Q(π*) ← Q(π)
30     if (iteration mod μ = 0)
31         for i = 1 to n
32             for j = 1 to n
33                 Φ_{i,j} ← (1 − η) · Φ_{i,j}
34 while (not (ε))
35 return π*
```

The basic algorithm can be extended in several ways. For example, the probabilities with which the next node is selected could be weighted by the weight of the edge leading to said node, using the weight as a heuristic. Furthermore, one could employ a form of elitism augmenting the amount of

pheromone layed on the path that was found to be the most promising in the current μ iterations (in addition to the normal update).

Further variants include rank-based updating, in which pheromone is deposited only on the edges of the best m solution candidates of the last iteration (consisting of the runs of μ ants), and maybe also on the best solution candidate found so far. This approach can be seen as analogous to rank-based selection whereas the standard approach is analogous to fitness proportionate selection.

Strict elite principles are extreme forms of rank-based updating: pheromone is deposited only on the best solution candidate of the last iteration or even only on the best solution found so far. However, this approach carries the risk of premature convergence and thus of getting stuck in a local optimum.

In order to avoid extreme values of the pheromone deposits, it can be advisable to introduce lower and upper bounds for the amount of pheromone on an edge. They correspond to lower and upper bounds for the probability of selecting an edge and thus help to enforce a better exploration of the search space (though at the price of slower convergence). A similar effect can be achieved by restricted evaporation: pheromone evaporates only from edges that have been traversed in the last iteration.

Improvements of the standard approach, which are meant to lead to better solution candidates, are local improvements of the round trip (like removing edge crossings, which obviously cannot be optimal). More generally, we may consider simple operations as they could be used in a hill climbing approach and thus try (in a limited number of steps) to optimize solution candidates locally. Among such operations are: exchange of cities that are visited in consecutive steps, permutation of adjacent triplets, "inverting" a part of a round trip, etc. More costly local optimization should only be considered to improve the best solution candidate before it is returned from the search procedure.

In order to apply ant colony optimization to other optimization problems, the problem has to be formulated as a search in a graph. In particular, it must be possible to describe a solution candidate as a set of edges. However, these edges need not form a path. As long as there is an iterative procedure with which the edges of the set can be chosen, ant colony optimization is applicable. Even more generally, ant colony optimization is applicable if solution candidates are constructed with the help of a series of (random) decisions, where every decision extends a (partial) solution. The reason is that the sequence of decisions can be interpreted as a path in a **decision graph** (also called **construction graph**). The ants explore paths in this decision graph and try to find the best (shortest, cheapest) path, which yields a best set or sequence of decisions.

4.8. Classes of evolutionary algorithms: genetic algorithms

Genetic algorithms are a class of evolutionary algorithms having solutions encoded as binary strings and where the solutions lack any high-level structure or semantic. That is, what the binary strings actually represent and what constraint such representation were to entail are irrelevant[18]. The term genetic algorithm bears its name from the structure of DNA, the most basic component of life, since it's described entirely by just four nucleotides: A, C, G, T.

```
GENERIC-GENETIC-ALGORITHM(f : ℝ^n → ℝ, μ, p_x, p_m, ε):
pop ← a random sequence of μ bit strings
for i = 1 to μ
    pop[i].fitness ← f(pop[i])
do
    chosen ← select μ individuals from pop with roulette wheel selection
    newpop ← ∅
    for i = 1 to μ/2
        u ← a value sampled from U ~ (0, 1)
        if (u ≤ p_x)
            ONE-POINT-CROSSOVER (chosen[2i - 1], chosen[2i])
        BIT-MUTATION (chosen[2i - 1], p_m)
        BIT-MUTATION (chosen[2i], p_m)
        newpop ← newpop ∪ {chosen[2i - 1], chosen[2i]}
    pop ← newpop
    for i = 1 to μ
        pop[i].fitness ← f(pop[i])
while(not(ε))
best ← pop[0]
for i = 1 to μ
    if (pop[i].fitness > best.fitness)
        best ← pop[i]
return best
```

Genetic algorithms are much simpler than evolutionary algorithms, infact so approachable that it's relatively straightforward to prove their convergence. That is, it is possible to give a mathematical proof of the fact that, after an arbitrary number of iterations and for a certain set of chosen parameters, a genetic algorithm will certainly yield an optimal solution.

Without loss of generality, it can be assumed that the genetic operators and the selection method used in any genetic algorithms are the ones found above, roulette-wheel selection and bit-mutation. To simplify even further, it can be assumed that the population size μ is always an even number, so that

[18]Technically speaking, any evolutionary algorithm, when run on a computer, must be encoded as binary objects, since computers store information in binary format. The difference between any evolutionary algorithm and a genetic algorithm is that the former manipulates bit strings directly as part of its definition.

the population can be split in two without one remaining, and that the chromosomes have a fixed length L.

A **schema** h is a string of L characters over the alphabet $\{0, 1, *\}$, that is $h \in \{0, 1, *\}^L$. A binary chromosome $c \in \{0, 1\}^L$ is said to **match** a schema h, written as $c \lhd h$, if each of its characters is equal to the character in h occupying the same position. A chromosome c that does not match a schema h is written as $c \not\lhd h$. The $*$ character, called the **wildcard symbol**, is always equal to both 0 and 1.

Exercise 4.8.1: Consider the schema $h =$ `**0*11*10*` and the two chromosomes $c_1 =$ `1100111100` and $c_2 =$ `1111111111`. Do the two match the schema?

Solution: c_1 matches h, since all 0s and 1s are in the same position in both. c_2 does not match h since, for example, the third bit of c_2 is 1 while the third bit of h is 0. □

The number of all possible (binary) chromosomes is 2^L, whereas the number of all possible schemata is 3^L. Every chromosome matches:

$$\sum_{i=0}^{L} \binom{L}{i} = \sum_{i=0}^{L} \frac{L!}{(L-i)!i!} = \frac{L!}{(L-0)!0!} + \frac{L!}{(L-1)!1!} + ... + \frac{L!}{(L-L)!L!} = 2^L$$

distinct schemata. This is because taking any schemata that matches the chromosome and exchanging an arbitrary number of bits with '$*$' returns a schemata that is still matched. This means that observing a single chromosome corresponds to observing many more schemata at the same time.

From the schemata matches by a single chromosome, the interest is in finding how many schemata are matched, on average, by and entire population. In principle, a population of size μ could match $\mu 2^L$ distinct schemata, but the actual number of matched schemata is much smaller, since many chromosomes might actually be duplicates.

In order to carry out our plan of tracking the evolution of chromosomes that match a schema, one has to examine how selection and applying genetic operators influence these chromosomes. This is done in three steps. In the first step, the analysis focuses on the effect of selection, in the second step the effect of one-point crossover, and in the third step the effect of bit mutation. The transition from a population at time t to the next generation at time $t + 1$ can be split into four steps:

- The starting population itself, at time t;
- The starting population after applying selection, at time $t + \Delta t_s$;
- The starting population after applying selection and crossover, at time $t + \Delta t_s + \Delta t_x$;
- The starting population after applying selection, crossover and mutation, at time $t + \Delta t_s + \Delta t_x + \Delta t_m$. After applying selection, crossover and mutation the starting population has become the new generation, therefore $t + \Delta t_s + \Delta t_x + \Delta t_m = t + 1$.

The expected number of chromosomes by a population at time t that match a schema h is denoted as $N(h, t)$. The interest is in finding the relationship between $N(h, t)$ and $N(h, t + 1)$, that is, how evolving a population across the generations changes (on average) the number of chromosomes that match a certain schema.

The first step involves considering the effect of selection on the number of chromosomes that match a schema. Recall that the expected number of offspring generated by a chromosome s in a population μ with roulette-wheel selection as selection method is $\mu \cdot f_{\text{rel}}(s)$. In particular, this also means that each chromosome s that matches h will have an average offspring size of $\mu \cdot f_{\text{rel}}(s)$. Therefore:

$$N(h, t + \Delta t_s) = \sum_{s \in P(t), s \lhd h} \mu \cdot f_{\text{rel}}^{(t)}(s)$$

Where the apex t denotes that the number of offspring of s depends on which iteration is considered.

Expressing this number as a sum of contributions of the individual chromosomes can be misleading. For this reason, it is preferable to rewrite the expression as:

$$N(h, t + \Delta t_s) = \mu \sum_{s \in P(t), s \lhd h} f_{\text{rel}}^{(t)}(s) = (\mu N(h, t)) \frac{\sum_{s \in P(t), s \lhd h} f_{\text{rel}}^{(t)}(s)}{N(h, t)} = \mu N(h, t) f_{\text{rel}}^{(t)}(h)$$

Where $f_{\text{rel}}^{(t)}(h)$ is called **mean relative fitness**. Substituting the explicit expression for the relative fitness in $\mu f_{\text{rel}}^{(t)}(h)$ gives:

$$\mu f_{\text{rel}}^{(t)}(h) = \mu \left(\frac{\sum_{s \in P(t), s \lhd h} f_{\text{rel}}^{(t)}(s)}{N(h, t)} \right) = \mu \left(\frac{\sum_{s \in P(t), s \lhd h} \frac{f_{\text{abs}}(s)}{\sum_{s' \in P(t)} f_{\text{abs}}(s')}}{N(h, t)} \right) = \frac{\mu \left(\sum_{s \in P(t), s \lhd h} f_{\text{abs}}(s) \right)}{N(h, t) \left(\sum_{s' \in P(t)} f_{\text{abs}}(s') \right)} =$$

$$= \frac{\frac{\sum_{s \in P(t), s \lhd h} f_{\text{abs}}(s)}{N(h,t)}}{\frac{\sum_{s' \in P(t)} f_{\text{abs}}(s')}{\mu}} = \frac{\overline{f_t(h)}}{\overline{f_t}}$$

$\overline{f_t(h)}$, the ratio of all the fitness contributions of the chromosome that match h in generation t and the expected number of chromosomes that match h in generation t is the mean fitness of the chromosomes that match h in generation t. $\overline{f_t}$, the ratio of all the fitness contributions of the chromosome of generation t and the size of the population is simply the mean fitness of all chromosomes in generation t.

The expected number of chromosomes that match h in generation t can therefore be written as:

$$N(h, t + \Delta t_s) = N(h, t) \mu f_{\text{rel}}^{(t)}(h) = N(h, t) \frac{\overline{f_t(h)}}{\overline{f_t}}$$

The second step involves considering the effect of genetic operators on the number of chromosomes that match a schema. Since the genetic operator under consideration is one-point-crossover, this entails finding the probability that applying one-point-crossover to chromosomes that match h results in chromosomes that still match h.

More specifically, consider two chromosomes c_1 and c_2 and a schema h. Suppose that c_1 matches h, and that one-point-crossover is applied obtaining two new chromosomes $c_{(1)'}$ and $c_{(2)'}$. If $c_{(1)'}$ has inherited from c_1 the exact same bits that correspond to non-wildcard bits in h, then there's guarantee that $c_{(1)'}$ will also match h, no matter which bits are inherited from c_2.

Exercise 4.8.2: Consider the schema $h =$ `***0*1*1**` and the two chromosomes $c_1 =$ `0000011111` and $c_2 =$ `1111100000`. Do they match the schema? If one-point-crossover is applied with cutoff point 5, do the resulting chromosomes match the schema? And with cutoff point 3?

Solution: c_1 matches the schema while c_2 does not (due to a mismatch in the fourth position). With cutoff point 5, the resulting chromosomes are $c_{(1)'} =$ `1111111111` and $c_{(2)'} =$ `0000000000`; neither match the schema. With cutoff point 3, the resulting chromosomes are $c_{(1)'} =$ `1110011111` and $c_{(2)'} =$ `0001100000`; the first matches the schema, the second doesn't. Notice

how choosing cutoff point 3 preserves all the bits in c_1 that correspond to non-wildcard bits in h, hence $c_{(1)'}$ was guaranteed to match h. □

On the other hand, if $c_{(1)'}$ does not inherit from c_1 the exact same bits corresponding to non-wildcards in h, then there's still a chance for $c_{(1)'}$ to match h, even though there's no guarantee. In particular, this happens if the bits inherited from c_2 are equal to the corresponding bits in h. It is also entirely possible for two chromosome that don't match a schema to generate a chromosome that matches the schema, if the two parent chromosomes partially match a schema and the two partial matches are combined into a complete match.

Exercise 4.8.3: Consider the schema $h =$ `***0*1*1**` and the two chromosomes $c_1 =$ `0001011111` and $c_2 =$ `1110100100`. Do they match the schema? If one-point-crossover is applied with cutoff point 5, do the resulting chromosomes match the schema? Consider the two chromosomes c_2 and $c_3 =$ `0000011111`. Do they match the schema? If one-point-crossover is applied with cutoff point 5, do the resulting chromosomes match the schema?

Solution: Neither c_1 or c_2 match the schema. With cutoff point 5, the resulting chromosomes are $c_{(1)'} =$ `1110111111` and $c_{(2)'} =$ `0001000100`; the first matches the schema, the second doesn't. On the other hand, c_3 matches the schema, while c_2 does not. With cutoff point 5, the resulting chromosomes are $c_{(3)'} =$ `1110111111` and $c_{(2)'} =$ `0000000100`; the first matches the schema, the second doesn't. □

For this reason, it becomes evident that the choice of the cutoff point has great influence on whether the resulting chromosomes will match or not match the schema matched by their parents. In particular, the critical subsection of a chromosome is the one that does not correspond to wildcard bits in the schema. It is then useful to introduce a quantity called **defining length** of a schema h, denoted as $\text{deflen}(h)$, defined as the difference between the position of the last and the first non-wilcard element of h.

Exercise 4.8.4: What is the defining length of $h =$ `**0*11*10*`?

Solution: The first non-wilcard bit of h is a 0 in the third position, whereas the last non-wildcard bit of h is a 0 in the ninth position. Therefore $\text{deflen}(h) = 9 - 3 = 6$. □

The defining lenght of a schema determines which choices of the cutoff points are "safe", meaning that they guarantee that a matching parent will produce a matching childer. In particular, any choice of the cutoff point between 1 and $\text{deflen}(h) - 1$ and between $\text{deflen}(h)$ and $L - 1$ is safe, since they are constituted exclusively by wildcards that will match anyway. Since in one-point-crossover all choices of cutoff point are equally likely, the probability that the chosen cutoff point is not "safe" is $\frac{\text{deflen}(h)}{L-1}$, and the probability that the chosen cutoff point is "safe" is $1 - \frac{\text{deflen}(h)}{L-1}$.

Summing up, to derive an expression of $N(h, t + \Delta t_s + \Delta t_x)$ it is necessary to consider four possibilities:

- A chromosome does not undergo crossover, therefore its matching status with respect to the schema is the same;
- A chromosome does undergo crossover, and the cutoff point lies in such a way that all non-wildcard characters are transfered to the offspring, hence the matching status of the parent is itself inherited by the offspring;

- A chromosome does undergo crossover, and the cutoff point lies in such a way that only some non-wildcard characters are transfered to the offspring, but not all of them. However, the resulting chromosome still matches the schema, because the subchromosome inherited by the other parent also happens to match the subschema;
- A chromosome does undergo crossover, and even though both parents do not match a schema their offspring does because their partial matches are combined.

Let p_x be the probability that a chromosome will undergo crossover (the same for all chromosomes), and let p_{loss} be the probability that a chromosome that matches h undergoes crossover and turns into a chromosome that does not match h anymore. The four quantities above are combined as follows:

$$N(h, t + \Delta t_s + \Delta t_x) = A + B + C = \underbrace{(1 - p_x) N(h, t + \Delta t_s)}_{A} + \underbrace{p_x N(h, t + \Delta t_s)(1 - p_{\text{loss}})}_{B} + C$$

Where:

- A is the expected number of chromosomes that matched h and do not undergo crossover, hence will certainly still match h;
- B is the expected number of chromosomes that matched h, underwent crossover and whose resulting offspring manages to match h;
- C is the expected number of chromosomes that did not match h before crossover but whose offspring does.

Of course, it is impossible to know C with the data at hand. Therefore, a closed form expression for $N(h, t + \Delta t_s + \Delta t_x)$ cannot be obtained, but it's still possible to find a lower bound (being C positive, $A + B + C \leq A + B$).

The next step is finding an expression for p_{loss}. As stated above, a chromosome can result in a mismatch after applying one-point-crossover either if the cutoff point is not "safe", even though the opposite is not necessarely true, since an "unsafe" cut can still result in chromosomes that match. Therefore, p_{loss} is given by the difference between two probabilities, p_{unsafe} and p_{manage}, representing the two hereby described situations. Since p_{manage} is close to impossible to estimate, it is again necessary to resort to an upper bound of p_{loss} and compute just p_{unsafe}.

A loss of a match is possible if two conditions are satisfied at the same time: the cut is "unsafe" (obviously) and the second parent does not match the schema. The upper bound for p_{loss} is therefore:

$$p_{\text{loss}} = p_{\text{unsafe}} - p_{\text{manage}} \Rightarrow p_{\text{loss}} \leq p_{\text{unsafe}} \Rightarrow p_{\text{loss}} \leq \frac{\text{deflen}(h)}{L - 1} \cdot \left(1 - \frac{N(h, t + \Delta t_s)}{\mu}\right)$$

Where the two product terms correspond to the probabilities of the two conditions discussed above. Plugging this expression in the previous equation:

$$
\begin{aligned}
N(h, t+\Delta t_s+\Delta t_x) &\geq (1-p_x)N(h, t+\Delta t_s) + p_x N(h, t+\Delta t_s)(1-p_{\text{loss}}) \\
&= (1-p_x)N(h, t+\Delta t_s) + p_x N(h, t+\Delta t_s)\left(1 - \frac{\text{deflen}(h)}{L-1}\left(1 - \frac{N(h, t+\Delta t_s)}{\mu}\right)\right) \\
&= N(h, t+\Delta t_s)\left(1 - p_x + p_x\left(1 - \frac{\text{deflen}(h)}{L-1}\left(1 - \frac{N(h, t+\Delta t_s)}{\mu}\right)\right)\right) \\
&= N(h, t+\Delta t_s)\left(1 - \cancel{p_x} + \cancel{p_x} - p_x\frac{\text{deflen}(h)}{L-1}\left(1 - \frac{N(h, t+\Delta t_s)}{\mu}\right)\right) \\
&= N(h, t)\frac{\overline{f_t(h)}}{\overline{f_t}}\left(1 - p_x\frac{\text{deflen}(h)}{L-1}\left(1 - \frac{N(h, t)\frac{\overline{f_t(h)}}{\overline{f_t}}}{\mu}\right)\right) \\
&= N(h, t)\frac{\overline{f_t(h)}}{\overline{f_t}}\left(1 - p_x\frac{\text{deflen}(h)}{L-1}\left(1 - \frac{N(h, t)}{\mu}\frac{\overline{f_t(h)}}{\overline{f_t}}\right)\right)
\end{aligned}
$$

Note the presence of an inequality instead of an equality, since the C term was neglected and for p_{loss} an upper bound was computed.

The third step involves considering the effect of mutations on the number of chromosomes that match a schema. Clearly, flipping a bit that is paired to a non-wildcard character in a schema reverses the matching status. On the other hand, flipping a bit that is paired to a wildcard character in a schema has no effect on the matching status.

Therefore, a chromosome matching a schema that undergoes mutation will maintain its mutation status if and only if all of its flipped bits are those paired to wildcard characters of the schema. Equivalently, a chromosome preserves its matching status after mutation if and only if no bits that are paired to non-wildcard characters are flipped. Since all bit flips happen independently of the others, the probability of preserving matching status after mutation is $(1-p_m)^{\text{ord}(h)}$, where p_m is the probability of one bit to be flipped and $\text{ord}(h)$, called the **order** of the schema, is the number of non-wildcard characters of h.

The expression of $N(h, t+\Delta t_s+\Delta t_x+\Delta t_m)$ can therefore be written as:

$$
N(h, t+\Delta t_s+\Delta t_x+\Delta t_m) = N(h, t+1) = N(h, t+\Delta t_s+\Delta t_x)(1-p_m)^{\text{ord}(h)}
$$

Substituting the explicit expression for $N(h, t+\Delta t_s+\Delta t_x)$, one gets the so-called **schema theorem**:

$$
N(h, t+1) \geq N(h, t)\underbrace{\frac{\overline{f_t(h)}}{\overline{f_t}}\left(1 - p_x\frac{\text{deflen}(h)}{L-1}\left(1 - \frac{N(h, t)}{\mu}\frac{\overline{f_t(h)}}{\overline{f_t}}\right)\right)(1-p_m)^{\text{ord}(h)}}_{g(h,t)}
$$

The schema theorem states that the average number of chromosomes that matches a certain schema h is multiplied by a factor of $g(h, t)$ at every generation. If $g(h, t)$ is greater than 1, the average number of matching chromosomes will increase, if $g(h, t)$ is smaller than 1, the average number of matching chromosomes will decrease. Being the population size constant, the number of matching chromosomes cannot decrease for all schemata, since at least one schema must be matched by an individual of the population. Therefore, there's at least one schema that increases the number of matching chromosomes after each generation.

The importance of the schema theorem lies in the fact that it can be exploited to find for which schemata $g(h, t)$ grows particularly quickly. This is because the size of $g(h, t)$ corresponds to the

amount of chromosome "chunks" that are inspected, and in turn an high value of $g(h,t)$ corresponds to an effective exploration of the search space.

The expression of $g(h,t)$ is constituted by a product of three terms, therefore an high value of $g(h,t)$ is obtained when all three terms are high as well. In particular:

- The term on the left is a fraction, and the dependence on h is in the numerator. Therefore, the numerator should be high;
- The term in the middle is a polynomial, where the dependence on h is found as numerator of three fractions. Therefore, the numerators should be high;
- The term on the right is an exponentiation of a number between 0 and 1 and the dependence on h is in the exponent, therefore the exponent should be small.

Summing up, valuable schemata should have: high mean fitness, small defining length and low order. Such schemata are also called **building blocks**, due to which the schema theorem is sometimes also referred to as the **building block hypothesis**: the evolutionary search focuses on promising building blocks of solution candidates.

It should be noted that the schema theorem is an oversimplification, since many details are not taken into account. For example, it does not consider epistasis, where a gene expression is blocked by another, but assumes all genes to be mostly independent. Also, $N(h,t)$ refers to the average number of matching chromosomes, meaning that the theorem is valid only if the population size is very large (close to infinity, that is). Finally, $N(h,t)$ does not depend only on the schema h, but also on the time t, hence drawing conclusions from a single generation shift to any generation shift is questionable.

4.9. Classes of evolutionary algorithms: genetic programming

Evolutionary algorithms operate on chromosomes made up of strings. Logical predicates are nothing but strings with a semantic attached to it. Therefore, it is possible to generate logical predicates using evolutionary techniques.

Such techniques encompass what is called **genetic programming**: applying the principles of evolution to functional terms or entire computer programs to find, through an evolutionary-like algorithm, the one that addresses a particular purpose. In general, this entails starting from a set of inputs and outputs, and trying different possible functional terms or programs that map each inputs to its output. The program of interest is the one that is capable of matching every single input to the respective output.

The chromosomes of genetic programming are called **genetic programs**. Each genetic program is a functional term or a program. Since computer programs and logical statements with the same semantic can have more or less components, genetic programs are allowed (and expected) to have different lenghts. This is different from most evolutionary algorithms, where the length of a chromosome is fixed.

Each genetic program is expressed in a **formal language**, whose elements are constructed from two sets: a set $\mathcal{F}$ of **function symbols and operators** and a set $\mathcal{T}$ of **terminal symbols** (constants and variables). The choice of $\mathcal{T}$ and $\mathcal{F}$ is program-dependent. Parenthesis can also be introduced to specify the order in which functions are to be applied.

Exercise 4.9.1: What would be the set of terms and functions for the formal language of zeroeth-order logic?

Solution:

$$\mathcal{F} = \{\wedge, \vee, \neg, \Rightarrow\} \qquad \mathcal{T} = \{a, b, c, ..., 0, 1\}$$

□

Each element of a language, called **symbolic expression** is therefore a member of $\mathcal{G} \subseteq (\mathcal{F} \cup \mathcal{T} \cup \{(,)\})^*$. Out of all possible combinations of terms, functions and parenthesis, the symbolic expressions of interest are the so-called **well-formed formulas** (**WFFs**), that abide by a set of rules defined in a **grammar**. Well-formed formulas are defined recursively as follows:

- A single constant is a well-formed formula;
- A single variable is a well-formed formula;
- If $w_1, ..., w_n$ are n WFFs and f is a n-ary function in $\mathcal{F}$, then $f(w_1, ..., w_n)$ is a well-formed formula;
- Nothing else is a well-formed formula.

Notice how this way of writing logical formulas is somewhat different than the usual notation, especially for operators having arity equal to 2, of terminal-function-terminal. That is, instead of having something like $3 + 5$ or $a \Rightarrow b$ one has $+(3, 5)$ and $\Rightarrow (a, b)$. The notation of well-formed formulas in genetic programming is what's called **prefix notation**: even though it may be less readable, it is much easier to manipulate (terms are arranged into a stack, their number is predictable). Also, any expression written in prefix notation can be converted into an equivalent expression in "standard" notation, therefore there's no loss of expressiveness.

Exercise 4.9.2: What would be the well-formed formulas for the formal language of zeroeth-order logic?

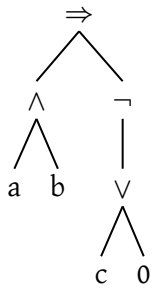
Solution:
- 0 is a well-formed formula;
- 1 is a well-formed formula;
- Any variable ($a, b, c, ...$) is a well-formed formula;
- If X and Y are two well-formed formulas, then $\wedge\,(X, Y)$ is a well-formed formula;
- If X and Y are two well-formed formulas, then $\vee\,(X, Y)$ is a well-formed formula;
- If X and Y are two well-formed formulas, then $\Rightarrow\,(X, Y)$ is a well-formed formula;
- If X is a well-formed formula, then $\neg(X)$ is a well-formed formula;
- Nothing else is a well-formed formula.

□

Symbolic expressions are represented using **parse trees**. A parse tree is a tree data structure where each node encodes one and only component (a terminal or a function) of a given symbolic expression. Terminal symbols are the leaves, functions are the inner nodes and each edge connects a function to one of its arguments. The root of a parse tree is, in general, a function, even though one could have a parse tree with a single terminal and nothing else, which would be the root (such degenerate cases are not considered, however). A subtree with a given root represents a subexpression having such root as the operator and its children as its arguments. The height of the node in the tree represents the order of preference in which the expression is to be evaluated: the higher, the earlier.

Exercise 4.9.3: Consider the symbolic expressions $\Rightarrow\,(\wedge\,(a, b), \neg(\vee\,(c, 0)))$ and $\wedge\,(\neg(a), \vee)$ of the formal language of zeroeth-order logic. Are they well-formed formulas? Draw their parse tree.

Solution: The first symbolic expression is a well-formed formula, the second is not.



□

A desirable property of $\mathcal{F}$ is that all the functions that it contains are total functions, meaning that they accept any possible input value of their domain. Examples of functions that are not total functions are the division (which is undefined when the second operand is 0) and the logarithm (which is undefined for negative numbers). If this is not the case, genetic programs might not be able to complete their execution. The issue can either be solved by:

- Restricting the domain introducing additional constraints, so that the function won't ever have an input whose output is undefined. For example, preventing 0 for being the input of a division;
- Introducing a penalty factor, as it was done in evolutionary computing, so that faulty chromosomes die out;
- Employing a **protected** version of function that return "nice" values in the presence of troublesome inputs. For example, modifying the logarithm so that it returns $\log(x)$ if x is positive (as expected) and 0 if x is negative.

Along the same line, if the function is supposed to accept values from multiple distinct subdomains, it can be adapted by changing the meaning of the data types as needed. One known example is the convention used by `C` and `C++` when dealing with booleans. A `C`/`C++` function that accepts a boolean parameter that receives a non-boolean parameter interprets it as follows: the number 0 is converted into `false`, anything else (a `char`, a non-zero number, ecc...) into `true`.

Another important property is the **completeness** of the sets $\mathcal{F}$ and $\mathcal{T}$ with respect to the expressions they represent. That is, the two sets should contain a sufficient number of elements to be able to generate every possible expression of the language. This is because genetic programming is a mere ricombination of "building blocks", not the creation of those "blocks" themselves. If the building blocks at hand cannot construct an expression with a given semantic, such expression will never come to life.

Finding the smallest set of functions and terminals that can generate every expression of a language is an NP-hard problem. Therefore, $\mathcal{F}$ will most likely be bigger than necessary, with expressions with the same semantic that have more than one syntactic representation. This is not an issue, however, since introducing more functions can simplify expressions and increase their readability.

Exercise 4.9.4: Is it possible to find a smaller set of functions than $\mathcal{F} = \{\wedge, \vee, \Rightarrow, \neg\}$ for the formal language of zeroeth-order logic?

Solution: Yes. A set such as $\mathcal{F}' = \{\wedge, \neg\}$ would be sufficient. This is because both $\vee$ and $\Rightarrow$ can be rewritten only using functions from $\mathcal{F}'$:

$$P \vee Q \equiv \neg((\neg P) \wedge (\neg Q)) \qquad\qquad P \Rightarrow Q \equiv \neg(P \wedge (\neg Q))$$

However, this would make manipulating expressions much more cumbersome, which is why $\mathcal{F} = \{\wedge, \vee, \Rightarrow, \neg\}$ is a much more convenient choice. □

A good symbolic expression that solves the problem at hand is no different than applying an evolutionary algorithm, constructing a random initial population of symbolic expressions. Such expressions are encoded as parse trees, since from an algorithmic standpoint they are much easier to manipulate than, say, bare strings, especially for computing its fitness.

The fitness of each parse tree (of each symbolic expression) is evaluated by a fitness function. The fitness of a symbolic expression represents how well the genetic program maps the inputs to the expected output, or how many inputs are mapped correctly. The symbolic expressions with higher fitness will (tend to) mutate and produce offspring applying genetic operators, the symbolic expressions with lower fitness will (tend to) die out in the generations.

Randomly generating chromosomes for genetic programming has to be done with caution, since it must abide by the rules defined in the grammar. An ill-defined parse tree is a waste at best and a source of defective offspring at worst. The best and simplest course of action to follow the (naturally) recursive definition for well-formed formulas. To make sure that the procedure terminates, one could

set a maximum number of nodes of the parse tree and/or a maximum depth; when such threshold is reached, all remaining branches are closed with terminal symbols, and aren't expanded further.

GP-INITIALIZE-GROW$(d, d_{\max})$:

```
1  if (d = 0)
2  |   n ← a random function sampled from F        // Avoid single-term expressions
3  else if (d ≥ d_max)
4  |   n ← a random term sampled from T            // Close branch when
5  |                                               // maximum size is reached
6  else
7  |   n ← a random element sampled from T ∪ F     // Open branch
8  foreach c in arguments of n
9  |   c ← GP-Initialize-Grow (d + 1, d_max)       // Expand branches recursively
10 return n
```

A common extention of the algorithm would be not to assign the same probability to each element drawn from $\mathcal{T}$ and $\mathcal{F}$, but to weight the probabilities differently. This way, both the complexity and the size of the parse tree can be controlled to some extent.

Another slight variation is what's called "full" initialization, where the maximum tree height $d_{\max}$ is always reached, whereas in the previous algorithm a branch could close long before $d_{\max}$ is reached (this is because a terminal symbol could be drawn at any step).

GP-INITIALIZE-FULL$(d, d_{\max})$:

```
1  if (d ≥ d_max)
2  |   n ← a random term sampled from T            // Close branch when
3  |                                               // maximum size is reached
4  else
5  |   n ← a random element sampled from F         // Always start with a function
6  |                                               // So that max size is always reached
7  foreach c in arguments of n
8  |   c ← GP-Initialize-Full (d + 1, d_max)       // Expand the branches recursively
9  return n
```

A more refined method of initialization is what's called "ramped half-and-half" initialization, where the maximum tree depth varies between iteration combining both the "grow" and the "full" initialization. This way, the population becomes representative of as many tree shapes, complexities and depths as possible.

GP-Initialize-Half-And-Half(μ, $d_{\max}$):

```
P ← ∅                                          // Population starts empty
for i = 1 to d_max do
    for j = 1 to 2 · d_max do
        P ← P ∪ GP-Initialize-Grow (0, i)
        P ← P ∪ GP-Initialize-Full (0, i)
return P
```

As of any evolutionary algorithm, a population evolves and increases its fitness by applying genetic operators. For example, applying crossover to two symbolic expressions means exchanging one subexpression (that is, one parse subtree) of the first with a subexpression of the second, and vice versa.

Applying mutation to a symbolic expression entails replacing a subexpression of a symbolic expression with a randomly generated subexpression. Again, one should be careful in applying mutation to genetic programs, since blindly substituting a subexpression with another can render the symbolic expression not a well-formed formula anymore. This can be prevented by randomly choosing a node in the parse tree, delete all nodes descending from said node and expanding the truncated branch applying an initalization algorithm like presented above. The algorithm can also be tuned to set a very small maximum depth, so that the newly formed branch has length comparable to the original, and does not introduce too much randomness.

However, if the population is sufficiently large and diversified, there's no need to apply both mutation and crossover: crossover alone is guaranteed to generate arbitrarely different individuals. This is because genetic programs come in very different sizes and depths (evolutionary algorithms deal with chromosomes of fixed lenght), hence recombining two genetic programs can generate individuals that are completely different from their parents, even if said parents are similar or, in extreme cases, identical.

Genetic programming has a tendency to "clutter" the solutions, even optimal solutions, with needless subexpressions. For example, in the case of Boolean formulas, a subexpression such as $\vee\ (x, \neg(x))$ is tautological, thererore inserting it in any Boolean formula won't change its meaning, but for the same reason it is also completely useless. (sub)Expressions such as these are called **introns**, in analogy to the sections of DNA that don't encode any information, either because they can never be activated (most likely a leftover of a previous stage of evolution) or because they are actually meaningless (also called *junk DNA*).

Introns can be introduced in a genetic program by a mutation or a ricombination. They are particularly problematic because, since adding one does not alter the semantics of a symbolic expression, it also means that is does not alter its fitness. Therefore, from the point of view of a "naive" algorithm, adding an intron does not worsen the candidate solution[19]. The simplest countermeasure would be lowering the fitness of genetic programs (of parse trees) that are particularly long, but this could also penalize valid solutions.

A better approach is what's called **editing**. Editing is a special genetic operator that, instead of introducing new genes in an individual, "prunes" and simplifies the already existing chromosome. Editing is divided into **general editing** and **special editing**. General editing constists in substituting a subexpression with its result, either because it's only made up of constants or because it depends on

[19] In nature this wouldn't be the case, since a DNA more "cluttered" with introns also makes it harder to replicate. Therefore, the presence of extra introns in the DNA of a living being could entail a loss of fitness.

variables that are aren't in the subexpression. Special editing consists in applying equivalences that don't change the meaning of the subexpression, but make it shorter and/or more readable.

Exercise 4.9.5: What would be an example of general and special editing for the formal language of zeroeth-order logic?

Solution: One example of general editing would be substituting $\vee\,(x, \neg(x))$, the expression hinted at before, with 1, since it's a tautology. One example of special editing would be applying De Morgan's Law(s), such as rewriting $(\neg(\vee\,(a, b)))$ as $(\wedge\,(\neg a), (\neg b))$ (or viceversa). □

Editing can be incorporated in the application of the genetic operators themselves (mutation and/or crossover), in order to obtain genetic operators that don't generate introns, hence reducing the number of wasterful computations. One example is **brood recombination**: brood recombination creates many children from the same two parents by applying a crossover operator with different parameters. Only the best child of the brood enters the next generation. This method is particularly useful if combined with a fitness penalty, because then it favors children that achieve the same result with less complex chromosomes.

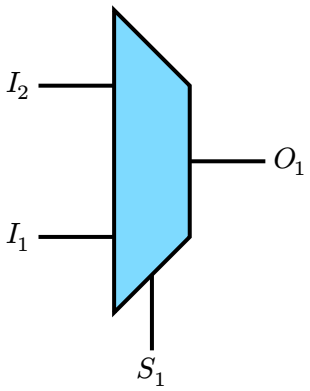
Another operator is **intelligent recombination**, a form of recombination that chooses the crossover points selectively in such a way to prevent, or at least to mitigate, the creation of introns. A third method consists in introducing slight changes in the evaluation function such that what are considered introns now can be "reactived", in the sense that the newly modified fitness function assigns a nonzero weight to that subtree, potentially leading to its elimination.

All these forms of intron prevention are still quite expensive, and rely on using ad-hoc genetic operators. In general, the cost of introducing intron prevention outweights its benefits, and is not worth it. It is actually much more common to keep the introns in the parse trees of the population, even if this reduces performance, and only prune the parse tree of the best solution found at the end of the process.

4.9.1. Applying genetic programming: the $n \times 1$ multiplexor problem

An example of genetic programming is solving the $n \times 1$ **multiplexor problem**. A multiplexor is a device that has n data inputs, $\log_2(n)$ address lines inputs and 1 output. The output of the multiplexor is given by the data input "chosen" by the address lines: if the value of the selectors is i (in binary), then the output of the multiplexor is the value of the i-th data input. The possible number of input combinations is $2^{n+\log_2(n)}$. Solving the problem entails having an $n \times 1$ multiplexor given as a "black box" and finding a symbolic expression of the Boolean function that describes it[20].

[20]The multiplexor problem can actually be solved analytically with little effort (when n is small, at least), hence it should be seen just as a paradigmatic example.



Selector	Input 1	Input 2	Output
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 3: On the left, a schematic representation of a 2×1 multiplexor. On the right, the corresponding truth table.

The set of symbols chosen for solving the problem is $\mathcal{T} = \left\{d_0, d_1, ..., d_n, s_0, s_1, ..., s_{\log_2(n)}\right\}$, where the d_i symbols are the n data inputs and the s_i symbols are the $\log_2(n)$ address lines inputs. The set of functions is $\mathcal{F} = \{\text{and}, \text{or}, \text{not}, \text{if}\}$: and and or have two arguments, not has one argument and if has three arguments (the condition, what to do if the condition is true, what to do otherwise). Since it is assumed that the inputs will always be Boolean values, the functions are domain complete. Also, the chosen set of functions is sufficient to generate all possible Boolean functions.

With $M = 2^{n+\log_2(n)}$, a simple choice of the fitness function can be $M - \sum_{i=0}^{M} e_i$, where e_i is the error for the i-th input. That is, $e_i = 0$ if the computed output for the i-th matches the desired output for the configuration and $e_i = 1$ otherwise. A termination criteria would stopping the procedure when an individual with fitness equal to M is found, meaning that is perfectly matches each configuration with its expected input.

4.10. Classes of evolutionary algorithms: evolutionary strategies

Evolutionary Strategies (**ES**) are a subclass of evolutionary algorithms (and the oldest) that are focused on solving numerical optimization problems. The chromosomes of an evolutionary strategies are therefore always arrays of real numbers, and the evaluation function is the function for which to find an optimum of.

Selection in evolutionary strategies is always carried out by strict elitism, meaning that only the fittest individuals are transfered to the next generation (instead of just being *more likely* to be transfered). Selection in strict elitism takes two forms: **plus strategy** and **comma strategy**. In the former, selection is applied considering both the parent individuals and the offspring individuals; in the latter, selection is applied considering only the offspring.

Let μ be the parent population size and let λ be the offspring population size. Evolution strategues employing plus strategy are often denoted as $(\mu + \lambda)$-ES, while those employing comma strategy are often denoted as (μ, λ)-ES. In both cases, the size of the population after applying selection is μ, therefore λ has to be at least equal to μ.

Actually, in comma strategy λ is much (much) greater than μ, in order to guarantee sufficient genetical diversity. λ should be high in plus strategies as well, since strict elitism makes the algorithm prone to getting stuck in local optima: an empirical argument states that λ should be at least 7 times larger than μ.

The tendency of the plus strategy to get stuck in local optima can be mitigated by substituting it with the comma strategy for some generations, in order to increase the diversity. On the other hand, in the comma strategy the best individuals of the original individuals are always lost, no matter their fitness, and this could be undesirable. Therefore, to improve the chances of converging, it is sensible to "relax" comma strategy and allow at least the best individual so far encountered to be kept, even if it's absent from the current or new population.

Since the goal of evolutionary strategies is to optimize a function, mutations consist in adding to the chromosome a random vector $\boldsymbol{r}$ of n elements, with n being the length of the chromosome. Each element r_i is the realization of a Gaussian random variable with mean 0 and standard deviation σ_i. The standard deviation may or may not depend on the index (that is, a different variance for each entry of the chromosome) and may or may not depend on the generation.

Another notable feature of evolutionary strategies is that the standard deviations of the chromosome entries are parameters themselves, meaning that they can be adapted together with the individuals. In the simplest case, there is just one standard deviation σ to be adapted, shared in common for all the entries of the chromosome.

A known heuristic for adapting σ is the so-called *one-fifth success rule*: if at least one-fifth of the offspring has better fitness than their parents, σ is increased; if less than one-fifth of the offspring has better fitness than their parents, σ is decreased. The most straigthforward way to increase/decrease σ is to multiply/divide it by a user-specified factor α.

Note how the ratio of offspring fitter than their parents and offspring less fit than their parents, called the **success rate**, may also be seen as a measure for the balance of exploration and exploitation. If the success rate is too large, exploitation of good individuals dominates, which can lead to effects of premature convergence. On the other hand, if the success rate is too low, exploration dominates, which can lead to slow convergence.

Also note how the one-fifth rule tends to be ill-suited for large populations, because it's clue that is too optimistic. For this reason, similar to how it is done in simulated annealing, a more refined approach would be to fix a threshold θ (which may be initialized at $1/5$) that is slowly increased over the iterations.

ES-GLOBAL-ADAPTATION-COMMA-STRATEGY($f : \mathbb{R}^n \to \mathbb{R}$, μ, λ, k, θ, α, ε):

```
1  s ← 0
2  σ ← an initial standard deviation
3  pop ← a random sequence of μ arrays of reals
4  foreach i in {1, ..., μ}
5  |   pop[i].fitness ← f(pop[i])
6  do
7  |   newpop ← ∅
8  |   foreach i in {1, ..., λ}
9  |   |   x ← a random element sampled from pop
10 |   |   y ← GAUSSIAN-MUTATION (x, σ)
11 |   |   if (x.fitness > y.fitness)
12 |   |   |   s ← s + 1
13 |   |   newpop ← newpop ∪ {y}
14 |   foreach i in {1, ..., μ}
15 |   |   newpop[i].fitness ← f(newpop[i])
16 pop ← the best μ individuals from newpop
17 if (t mod k = 0)
18 |   p_s ← fraction of the s individuals fitter than their parents
19 |   if (p_s > θ)
20 |   |   θ ← θ · α
21 |   else
22 |   |   θ ← θ/α
23 |   s ← 0
24 while (not (ε))
25 best ← pop[0]
26 foreach i in {1, ..., μ}
27 |   if (pop[i].fitness > best.fitness)
28 |   |   best ← pop[i]
29 return best
```

The algorithm assumes comma strategy. A version with plus strategy would differ only in the instruction pop' ← ∅ (destroy the current population completely), substituted by pop' ← pop (include the current population in the selection pool).

In contrast to global variance adaptation, local variance adaptation has a specific standard deviation for each chromosome, or even a specific standard deviation for each single gene of each chromosome. Each single standard deviation is optimized in turn with the individuals: the rationale behind it is

that chromosomes with "bad" standard deviations will create unfit individuals, whereas chromosomes with "good" standard deviations will create fitter individuals. Therefore, even though the two does not influence each other directly, "good" genes and "good" standard deviations should go hand in hand.

ES-Local-Adaptation-Comma-Strategy($f : \mathbb{R}^n \to \mathbb{R}$, μ, λ, ε):

```
1  pop ← a random sequence of μ arrays of reals
2  for i in {1, ..., μ}
3      pop[i].fitness ← f(pop[i])
4  do
5      newpop ← ∅
6      for i in {1, ..., λ}
7          (x, σ_x) ← a random element sampled from pop
8          (y, σ_y) ← Self-Adaptive-Gaussian-Mutation (x, σ)
9          newpop ← newpop ∪ {(y, σ_y)}
10     for i in {1, ..., μ}
11         newpop[i].fitness ← f(newpop[i])
12     pop ← the best μ individuals from newpop
13 while (not (ε))
14 best ← pop[0]
15 for i in {1, ..., μ}
16     if (pop[i].fitness > best.fitness)
17         best ← pop[i]
18 return best
```

The algorithm assumes, again, comma strategy, and can be adapted to work with plus strategy as done before.

A version of the algorithm where each single gene of each chromosome (and not simply each chromosome) is adapted is more convoluted. A commonly used rule for adapting single gene standard deviations is the following:

$$\sigma_i' = \sigma_i \exp(r_1 u_0 + r_2 u_i)$$

Where r_1 and r_2 are tunable parameters, $u_0, ..., u_i, ..., u_n)$ are n values sampled from a standard normal distribution and n is the length of the chromosome. Common empirically-driven choices for r_1 and r_2 are $\left(1/\sqrt{2n}, 1/\sqrt{2\sqrt{n}}\right)$ and $(0.1, 0.2)$. Also, a lower bound for σ_i (greater than the trivial lower bound of 0) is often introduced.

In the standard form of variance adaption, the variances of different vector elements are independent of each other. That is,the covariance matrix of the mutation operator is a diagonal matrix. As a consequence, the mutation operator is only able to prefer directions in the search space that are parallel to the coordinate axes of the search space. An oblique direction is impossible to reach, even if it were to be better.

This entails that the variances of the chromosome must either all increase or all decrease; it's impossible to have some standard deviations to increase and some to decrease. The issue can be solved by introducing, together with a variance for each gene, also their covariances. This way, it becomes possible (although not guaranteed) for each gene-specific variance to vary in its own direction.

As stated before, a recombination operator is used close to never in evolutionary strategies. If it is, it either takes the form of uniform crossover (randomly choosing genes from the two parents and merging the result) or blending, for example as implemented by arithmetic crossover. One should always keep in mind that crossover operators carry the danger of the Jenkins nightmare: the "evening out" of all variance in the population.

4.11. Classes of evolutionary algorithms: finding Pareto-frontiers

Since evolutionary algorithms operate with entire populations of individuals, it is no surprise that they are well-suited for tackling multi-criteria optimization problems. This is because solving such classes of problems requires to find an entire set of solutions (the Pareto-frontier), not just one. The basic idea of any evolutionary algorithm that solves multi-criteria optimization problems is to find a population that covers, or at least gets close to, the entire Pareto-frontier of the problem.

The simplest evolutionary algorithm that solves multi-criteria optimization problems is the **Vector Evaluated Genetic Algorithm** (**VEGA**) which works as follows: for each objective function $f_1, ..., f_k$ of the problem, the selection method is carried out k times, once with respect to each objective function. That is, if the population size is μ, selection (using roulette-wheel selection, tournament selection, elitism, ecc...) is applied once with respect to f_1, obtaining a batch of μ/k individuals, then it is applied once with respect to f_2, obtaining another batch of μ/k individuals, ecc... until $(\mu/k) \cdot k = \mu$ individuals are selected. In some sense, each subpopulation of size k evolves with respect to a certain function.

Even though the approach is simple and computationally inexpensive, it has the disadvantage of not being able to have individuals emerge over the others. That is, a solution that is a valid solution for all the objective functions at the same time struggles to be selected consistently across the iterations. This may have the unintended effect of "watering down" the solution to a single neighborhood of the Pareto-frontier, which defeats the purpose of the algorithm.

A better approach is to exploit the *dominance* of some solutions over others. The idea is to progressively remove from the population the solutions that are not dominated by any other, until the population becomes empty, ranking them by how early they were found:

1. Start from the highest rank and the entire population;
2. Find all solutions in the population that are not dominated by any other;
3. Assign a rank to this subset of solutions and remove them from the population;
4. Decrease the rank;
5. If the population is empty, stop. Otherwise, restart from the second step.

Once the partitioning is complete, it then becomes possible to perform rank-based selection, guaranteeing that the Pareto-frontier is covered as thoroughly as possible. For example, one could employ power law sharing: the fitness assigned to an individual is the lower, the more individuals in the population have similar function values.

An even more refined approach is **Non-dominated Sorting Genetic Algorithm-II**, based on the same dominance approach but substituting rank-based selection with tournament selection. That is, the algorithm employs a variant of tournament selection where the winner of the tournament is not decided based on its fitness, but based on its non-being-dominated rank. Also, a crowding distance mechanism is relied upon to ensure that the solutions are properly spread out along the Pareto-frontier.

The first building block of the algorithm is computing the fitness of each individual in the population with respect to all k activation functions:

```
UPDATE-ALL-FITNESSES((f_1, ..., f_k), P):
1  for i = 1 to k)
2      for j = 1 to μ)
3          P[j].fitness-i ← f_i(P[j])
```

The second building block is constructing the Pareto-frontiers. Given a population P, from the set are repeatedly extracted the Pareto-optimal solutions, each constituting a frontier. The procedure is repeated until all frontiers are built:

```
CONSTRUCT-PARETO-FRONTIERS(P):
1  i ← 1
2  F ← ∅
3  while (P ≠ ∅)
4      F_i ← the subset of P containing Pareto-optimal solutions
5      i ← i + 1
6      P ← P − F_i
7      APPEND (F, F_i)
8  return F                                   // F = (F_1, F_2, ...)
```

Note the use of a `while` loop instead of a `for` loop, since the number of frontiers is unknown a priori.

Then, each frontier is added to the new population one by one, starting from the first. Since the population size is $\mu = |P|$ and the frontiers $\mathcal{F}_1, \mathcal{F}_2, ...$ come from $P \cup Q$, there are clearly more individuals than the maximum capacity. Therefore, some of the frontiers have to be discared.

In particular, there will be a frontier $\mathcal{F}_i$ that, when added to the population, would exceed the maximum capacity, and has to be "truncated" to reach a size of exactly μ. This is done employing a selection mechanism called *crowding distance mechanism*. As the solutions in one front have the same quality, one can differentiate between them by using the so called crowding distance values (cd). The solutions in the crowded areas (in the objective space) get a low chance (small crowding distance) to survive the selection. Another approach is called **strength Pareto evolutionary algorithm 2 (SPEA2)**, which is a standard evolutionary algorithm extended to work with more than one evaluation function. The algorithm stores the non-dominated individuals separately in an *archive* of finite size, where elements can be added only if older elements are removed. The algorithm strives to fill it with as many non-dominated individuals as possible; if there aren't enough, it resorts to the best dominated individuals.

The idea of the archive is similar to the tabu list, but the admission criteria works the other way around: instead of filling the archive with the already attempted and therefore "banished" solutions, it is filled with the promising solutions. The last example is **Pareto-archived evolutionary strategy (PAES)**. This approach is based on a (1 + 1)-ES an also relies on an archive in the form of a multi-dimensional table. Unless the archive is full, new solution candidates are added to it. If it is full, all dominated solution candidates are removed from it. If there are no dominated solution candidates, one of the individuals in the hash entry with the most members is removed.

4.12. Classes of evolutionary algorithms: solving behaviour simulations

The applications of optimization problems are not confined to strictly numerical problems. They can be employed to study social interactions and the behaviours and choices of individuals. The framework under which **rational agents** are studied is **game theory**, where the decisions of two or more agents and the impact that they have are modeled as a game that they play. The problem of finding the set of decisions that give the best outcome for an agent is translated into finding a winning strategy for the game.

A paradigmatic example of problem in game theory is the **Prisoner's Dilemma**[21], where two agents are compelled to make a choice between cooperating with each other or betraying one another. The interest of the problem is finding out (in the context of the problem) which of the two strategies is the most effective, either with respect to their self-interest or with respect to the interest of both.

The problem can be phrased as follows. Two individuals found in possession of illegal fire arms are taken into custody. Not only that, but they are also suspected of being involved in a bank heist. Since there isn't sufficient evidence for framing them for robbery, the sheriff offers them a choice. If one of the two testifies as key witness and confesses that they both are indeed bank robbers, he will be acquainted of all charges for both crimes (0 years jail time, that is), whereas the other suspect will be jailed for 10 years. However, if they both plead guilty, the key witness priviledge won't apply, and they will both be jailed for 5 years. If both do not confess, since the evidence for illegal firearm possession is undisputed, they will both be in jail for 1 year. What would be the best strategy that the two prisoners should choose to get as little jail time as possible? To confess or to refrain?

Two-player problems such as these are often represented as matrices, called **payoff matrices**: the two sides (up-down and left-right) represent the two players, the rows/columns represent the choices that the two can make and the entries are the *payoffs*, the material advantage gained by the player for each combination of choices.

Since the Prisoner's Dilemma has two choices (refrain/confess) that each player can make, the matrix has four entries, corresponding to the four possibilities: both refrain, both confess, first refrains and second confesses, first confesses and second refrains. The values is the entries are the jail time served for each player for a given choice of actions. Since the desider outcome in game theory problems is often intended as the *maximum* advantage, whereas the suspects are interested in *minimizing* jail time, these values are written as negative numbers, so that the highest value (the most sought after) is 0.

		Suspect 2	
		Refrain	Confess
Suspect 1	Refrain	$-1, -1$	$-10, 0$
	Confess	$0, -10$	$-5, -5$

Figure 71: The payoff matrix for the Prisoner's Dilemma.

From a global perspective, it is clear that the best choice would be to refrain, since they would both get a feasible 1 year jail time. However, if both prisoners were to unapologetically behave to maximize their best interest, they would both choose to confess, since it's the option that gives the highest payoff. Yet, if both prisoners confess, hence they both behave "rationally" from their own perspective, the result is sub-optimal, with both having to serve 5 years in prison.

[21] The name "Prisoner's Dilemma" is somewhat misleading, since the two agents in the problem are suspects, not inmates. A more appropriate name would the "the Suspects' Dilemma".

Technically speaking, a double confession is the so-called **Nash equilibrium** of this payoff matrix: neither agent can improve its payoff by changing its action, while the other agent maintains the same action. If both players confess and only one of them could theoretically change its action retroactively, it would actually just worsen the situation (from -5 to 0, that is). An improvement is only possible if both agents change their action.

The Prisoner's Dilemma can be generalized to an abstract two-player game where both players, wanting to maximize their gain, can choose one move out of two in each iteration. These two moves are *cooperating* and *betraying*: in the first, they both obtain a little reward, in the second, one obtains a great reward and the other obtains nothing. A payoff matrix can encode the four possible outcomes (T, R, P, S), where:

- T: the player has betrayed and the other tried to cooperate (*temptation to defect*);
- R: both prisoners have cooperated (*reward for mutual cooperation*);
- P: prisoners betrayed each other (*punishment for mutual defection*);
- S: the player tried to cooperate, but the other betrayed (*sucker's payoff*).

		B	
		Cooperate	Betray
A	Cooperate	R, R	S, T
	Betray	T, S	P, P

Figure 72: The generalized payoff matrix for the Prisoner's Dilemma.

The values of T, R, P, S can be any quadruplet that satisfies the two following constraints:

$$T > R > P > S \qquad\qquad 2R > T + S$$

The left inequality states that looking after oneself should yield an higher payoff than cooperating, and that being betrayed results in a unfavorable outcome, otherwise there would be no point in being self-interested. It also states that being altruistic is better than being betrayed, otherwise there would be no point in cooperating. The right inequality states that making ongoing cooperation preferable to alternating exploitation. With these conditions, mutual defection is a Nash equilibrium of the payoff matrix.

The most interesting aspect of the Prisoner's Dilemma is that it models many real-world social interactions where two agents (not necessarely two humans) have to choose between helping each other out towards a common goal or being selfish and trying to take advantage of one another. From the point of view of the Dilemma, it would seem that the second choice is better, since, again, exploitation allows for a greater potential gain than collaboration. But if this is the case, it begs the question: why do most living beings (humans, animals, ecc...) favour altruism over egoism? If cooperating is worse than competing, should have evolution ruled it out?

First, it is clear that, despite its wide range of applicability, the Dilemma is a very limited model. For example, most real-world social interactions are episodic, meaning that after interacting with someone there's a good chance that many more other interactions with the same person/agent will happen in the near future. Also, it assumes perfect transfer of information, that is, both agents know with exact certainty which action the other agent has taken.

An extension of the Prisoner's Dilemma in this sense is the **Iterated Prisoner's Dilemma**, where the two parties have to take the same actions (cooperating/betraying) in multiple iterations. The rationale behind the Iterated Prisoner's Dilemma is that cooperating could be more enticing in the long run, since now actions have consequences: if one of the two players starts being selfish, the other might seek revenge in the following iterations, also acting self-interested. In the original formulation of the

problem this form of retaliation was not possible, since each player could only choose their action once.

In a known experiment, many strategies of varying complexity were tested against one another, to see which one, on average, managed to secure the highest number of points to the user employing it. Each strategy would compete in a round robin tournament, meaning that each would have to be paired once with each other strategy. Each strategy had to compete against the other for 200 rounds for each opponent. The strategies had access to the history of games that they played against their opponent, in order to get the upper hand by exploiting previously intercepted weaknesses. The "fitness" of each strategy was defined as the cumulative payoff obtained by the player in the entirety of the tournament.

Out of all the strategies ("always betray", "always cooperate", "only cooperate every n games", "choose randomly", ecc...) the winner was a very simple strategy that came to be known as *tit for tat*. The strategy was as follows: in the first game, always cooperate; in the following games, copy the move of the other player in the previous game. Even after repeating the experiment a second time, with revised and improved strategies, the winner was still *tit for tat*.

This is interesting, because *tit for tat* is not a strategy that will win in every game. For example, playing *tit for tat* against the *always betray* strategy results in a guaranteed loss, because in the first game *tit for tat* will cooperate and always defect in the following games, resulting in a net payoff loss and $n - 1$ ties. Also, if two *tit for tat* strategies are playing against each other, if by chance one of the two betrays by mistake, they would keep on betraying and cooperating changing they their roles back and forth, resulting in a very low quality.

An alternative strategy that does not fall to this issue could be *tit for two tats*, that starts betraying only after two betrayal in a row by the opponent. Note that this strategy is vulnerable to an opponent that already knows one will employ it, since it just needs to alternate back and forth between betraying and cooperating to win by a large margin.

The problem of finding out which strategy is the best can be solved in a more formal and substanciated way by employing a genetic algorithm, where each chromosome represents a strategy. Each gene of the chromosome corresponds to which move should be played in response to a specific history of matches. More specifically, each gene represented the action to take (cooperate, 0, or betray, 1) with respect to the actions taken in the previous three games with the same opponent. For example, the first allele could represent the action to take if the three previous games were $((0,0),(0,0),(0,0))$, the second allele the action take in response to $((1,0),(0,0),(0,0))$, ecc... all the way to $((1,1),(1,1),(1,1))$. Since each triplet of games involves one out of two choices for each player, six in total, the number of all possible triplets (and hence of genes) is $2^6 = 64$. Each chromosome was also endowed with 6 extra bits that contained the "zero" game, a starting condition so that a strategy could be engineered even in the first game, for a total of 70 bits.

The initial population is created by randomly generating sequences of 70 bits. The individuals of a population are evaluated by pairing each individual with sufficiently many opponents that are randomly selected from the population. In each pairing, 200 matches were played. The fitness of an individual is the average payoff it gained per pairing. Individuals are selected for the next generation according to the simplified expectation value model: let $\mu_f(t)$ be the average fitness of the individuals in the population at time t and $\sigma_f(t)$ its standard deviation. Individuals whose fitness was lower than $\mu_f(t) - \sigma_f(t)$ do not receive offspring; individuals whose fitness lied between $\mu_f(t) - \sigma_f(t)$ and $\mu_f(t) + \sigma_f(t)$ got one children and individuals whose fitness was greater then $\mu_f(t) + \sigma_f(t)$ got two. The genetic operators of choice were standard mutation and one-point crossover.

The algorithm was then run for a certain number of generations and the best individuals of the final population were examined. The patterns that most of such fittest individuals exhibited were the following:

- **Don't rock the boat**. If all three games ended up with both of you cooperating, keep going: $(0,0),(0,0),(0,0) \to 0$;
- **Be provocable**. If you both cooperated in the first and second games but the opponent betrayed you in the third, don't be naive and retaliate: $(0,0),(0,0),(0,1) \to 1$;
- **Accept an apology**. If you started cooperating, the opponent exploited you in the second game (you cooperated and they betrayed) and then you exploited them in the third (you betrayed and they cooperated), start cooperating again, since it would seem they are willing to make amend: $(0,0),(0,1),(1,0) \to 0$;
- **Forget**. If you cooperated in the first and third game but the opponent betrayed you in the second, don't hold a grudge and keep cooperating: $(0,0),(0,1),(0,0) \to 0$;
- **Accept a rut**. If you both always defected, keep going, since the opponent is most likely self-interested $(1,1),(1,1),(1,1) \to 1$.

The *tit for tat* strategy clearly possesses all five traits; the *tit for two tats* strategy had four out of five, lacking only the "be provocable" trait, since it started betraying only after two betrayals by the opponent in a row. This makes it vulnerable to players who know of this strategy and of its weaknesses, as stated above, hence it makes sense for this strategy to not be the top contender.

Note that this result should still not be taken as an argument that *tit for tat* is generally the best strategy. Again, a single individual employing the *tit for tat* strategy playing in a population that employs the *always betray* strategy will always loose. However, if there's a sufficiently large niche of individuals that employ *tit for tat*, they will eventually rule out the selfish individuals and take over the population. This is facilitated if the *tit for tat* players can choose their adversaries instead of being paired randomly, since they will naturally prefer to play against each other and thriving.