Contents

1. Neural Networks 3
   1.1. Biological foundations 3
   1.2. Threshold Logic Units 4
   1.3. Training TLUs 7
   1.4. Artificial neural networks 11
   1.5. Multilayer perceptrons 15
   1.6. Logistic regression 23
   1.7. Gradient descent 26
2. Fuzzy logic 29
   2.1. Fuzzy sets 29
   2.2. Fuzzy logic 34
   2.3. Extending operators to fuzzy sets 37
      2.3.1. Intersection, union, complement 37
      2.3.2. Universal and existential quantifiers 38
      2.3.3. Functions with one argument 38
      2.3.4. Cartesian product, projection, cylindrical extension 39
      2.3.5. Function with arbitrarely many arguments 40
   2.4. Linguistic variables 40
   2.5. Fuzzy reasoning 42
3. Evolutionary computing 44
   3.1. Optimization problems 44
      3.1.1. Optimization problems 44
      3.1.2. Examples of optimization problems 45
      3.1.3. Multi-criteria optimization problems 46
   3.2. Local search algorithms 47
      3.2.1. Gradient ascent/descent 47
      3.2.2. Hill climbing 48
      3.2.3. Simulated annealing 49
      3.2.4. Threshold accepting 50
      3.2.5. Great Deluge Algorithm 50
      3.2.6. Record-to-Record Travel 51
   3.3. Evolutionary algorithms 51
   3.4. Choosing a solution encoding 55
   3.5. Choosing a selection method 59
   3.6. Choosing a genetic operator 64
   3.7. Improving performance through parallelization 69
      3.7.1. Parallelizing creation, selection and mutation 69
      3.7.2. The island model 70
      3.7.3. Cellular evolutionary algorithms 71
   3.8. Classes of evolutionary algorithms: evolutionary local search 71
      3.8.1. Tabu search 71
      3.8.2. Memetic algorithms 72
      3.8.3. Differential evolution 72
      3.8.4. Scatter search 73
      3.8.5. Cultural algorithm 73
   3.9. Classes of evolutionary algorithms: swarm intelligence 73
      3.9.1. Particle Swarm Optimization 74
      3.9.2. Ant Colony Optimization 76

3.10. Classes of evolutionary algorithms: genetic algorithms 79
3.11. Classes of evolutionary algorithms: genetic programming 86
3.11.1. Applying genetic programming: the $n \times 1$ multiplexor problem 91
3.12. Classes of evolutionary algorithms: evolutionary strategies 92
3.13. Classes of evolutionary algorithms: finding Pareto-frontiers 96
3.14. Classes of evolutionary algorithms: solving behaviour simulations 97

1. Neural Networks

1.1. Biological foundations

Neurons are first and foremost studied by neurobiology and neurophyisiology. The interest of artificial intelligence is to mimic the way biological neurons work, so that the same model can be applied to non-living beings. In particular, the interest is to study the way living beings collect information through senses, the way they process this collected information and the way they learn from experience.

Neurons have a a core in the form of the nucleus that receives information from other neurons collected information. When the nucleus receives a sufficient aumount of stimulation, it releases back information on nearby neurons. The connection between the simulated neuron and the stimulating one is called **synapsis**; an excited neuron induces the synapsis to release chemicals called **neurotransmitters**, received from the **dendrites** of the receiving neuron.

If a neuron receives enough stimulation from its dendrites, it decides to send in turn a signal to other neurons through an electric signal. The **axon** propagate the electric stimulus from the dendrites to the nucleus. When a neuron sends an electric signal, we say that the neuron *fired*.

A real computer cannot, as is, completely capture the complexity of a real brain.

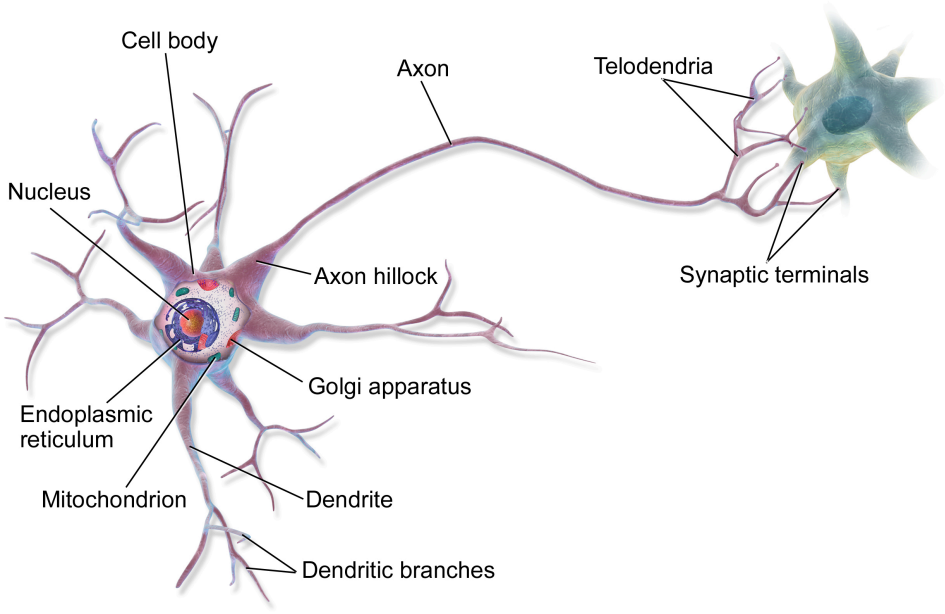


Figure 1: Schematic image of a neuron. By BruceBlaus, CC BY 3.0, via Wikimedia Commons. original image

Advantages of neural networks:

- High parallelism, which entails speedup;
- Fault tolerance, even if a large part of the network is failing the overall network might still work (not always, but close to);
- If some neurons get degraded, we slowly loose our capabilities, but never abruptly. Failing nodes can be phased slowly.

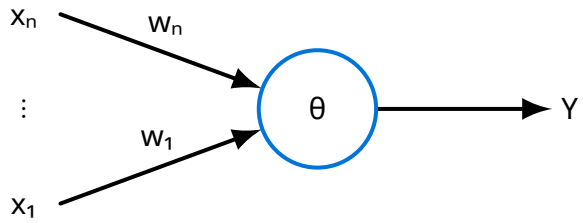
In first approximation, any living being has an input facility (smell, touch, taste), which deliver information to a neuron pool connected to an output. The idea is to have a model that approximates this structure but without the “living being” part.

1.2. Threshold Logic Units

A **Threshold Logic Unit** (**TLU**), also known as **perceptron**[1] or **McCulloch-Pitts neuron** is a mathematical structure that models, in a very simplified way, how neurons operate.

A TLU has n binary 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, otherwise is equal to 0.

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.



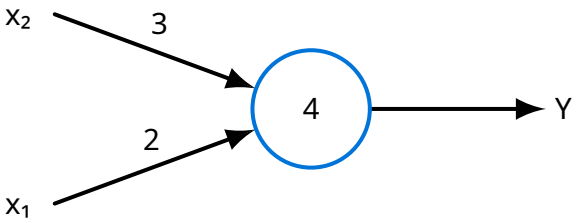
$$y = \begin{cases} 1 \text{ if } \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

Figure 2: 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 inputs can be collected into a single input vector $\boldsymbol{x} = (x_1, ..., x_n)$ and the weights into a weight vector $\boldsymbol{w} = (w_1, ..., w_n)$. With this formalism, the output y is equal to 1 if $\langle \boldsymbol{w}, \boldsymbol{x} \rangle \geq \theta$, where $\langle\rangle$ denotes the scalar product.

Exercise 1.2.1: Construct a TLU with two inputs whose threshold is 4 and whose weights are $w_1 = 3$ and $w_2 = 2$.

Solution:



x_1	x_2	$3x_1 + 2x_2$	y
0	0	0	0
1	0	3	0
0	1	2	0
1	1	5	1

□

Intuitively, a negative weight corresponds to an inhibitory synapse: if the corresponding input becomes active (that is, equal to 1), it gives a negative contribution to the overall excitation. On the other hand, a positive weight corresponds to an excitatory synapse: if the corresponding input becomes active (that is, equal to 1), it gives a positive contribution to the overall excitation.

Note how the weighted summation that discriminates whether the output of a TLU is 1 or 0 is very similar to an n-dimensional linear function. That is, by substituting the $\geq$ sign with a $=$ sign, it effectively turns into an n dimensional straight line:

[1] The original definition of perceptron was more refined than a TLU, but the two terms are often used interchangeably.

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

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-planes: one containing n-dimensional points that give an output of 1 when fed 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. Indeed, the coefficients $x_1, ..., x_n$ are the elements of a normal vector of the line: the half-plane that contains points that give the TLU an output of 1 is the one to which this vector points to.

Unfortunately, not all linear functions can be represented by a TLU. 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.

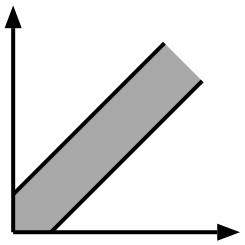
A set of points in the plane is called **convex** if connecting each point of the set with straight lines does not require to go outside of the set. The **convex hull** of a set of points is its the smallest superset that is convex. If two sets of points are both convex and disjoint, they are linearly separable.

A TLU is capable of representing only functions such as these, but for two sets of points a decision function might not exist, and therefore not all sets of points are linearly separable.

Exercise 1.2.2: Is the function $A \leftrightarrow B$ linearly separable?

Solution: No, and it can be proven.

x_1	x_2	y
0	0	1
1	0	0
0	1	0
1	1	1

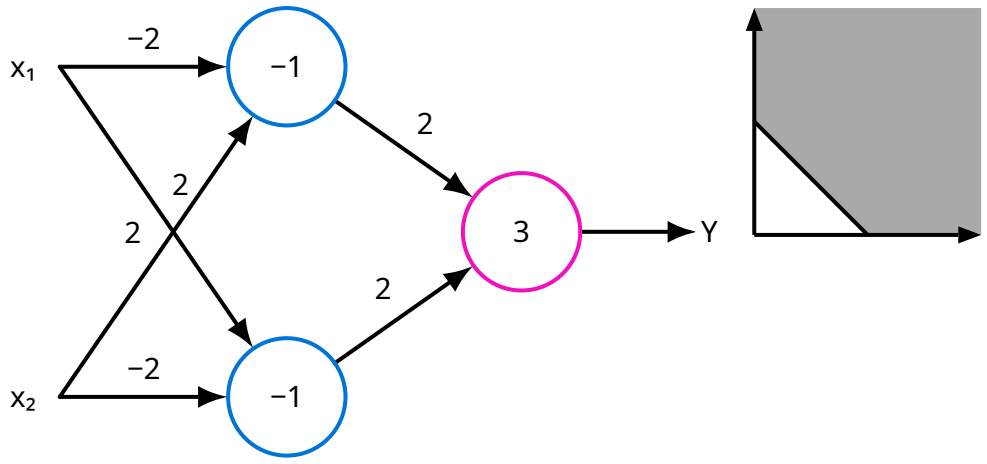


□

Even though single TLPs are fairly limited, it is possible to chain more TLPs 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. Since both the inputs and the outputs of a TLP are binary values, this doesn't pose a problem. By applying a coordinate transformation, moving from the original domain to the image domain, the set of points become linearly separable.

Exercise 1.2.3: Is it possible to construct a network of threshold logic units that can represent $A \leftrightarrow B$?

Solution: Yes. Note how $A \leftrightarrow B$ can be rewritten as $(A \rightarrow B) \wedge (B \rightarrow A)$. Each of the three functions (two single implications and one logical conjunction) is linearly separable.



□

It can be shown that all Boolean functions with an arbitrary number of inputs can be computed by networks of TLUs, since any Boolean function can be rearranged in the disjunctive normal form (or conjunctive normal form). A Boolean function in disjunctive normal form is only constituted by a streak of `or` each constituted by `and` (potentially negated), which are all linearly separable.

In particular, a TLU network of two layers will suffice: let $y = f(x_1, ..., x_n)$ be a Boolean function of n variables. It is possible to construct a network of threshold logic units that represents y applying this algorithm:

1. Rewrite the function y in disjunctive normal form:

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

   Where each $l_{j,i}$ can be either non-negated (positive literal) or negated (negative literal)
2. For each K_j construct a TLU having n inputs (one input for each variable) and the following weights and threshold:

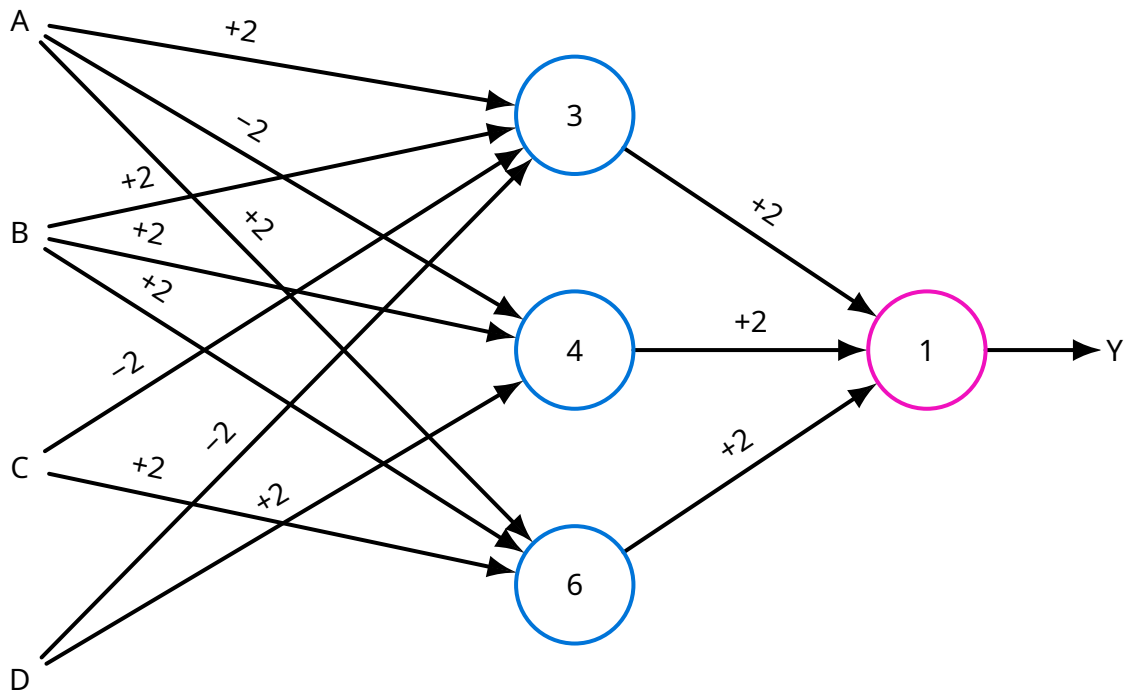
$$w_{j,i} = \begin{cases} +2 \text{ if } l_{j,i} \text{ is a positive literal} \\ -2 \text{ if } l_{j,i} \text{ is a negative literal} \end{cases} \qquad \theta_j = n - 1 + \frac{1}{2} \sum_{i=1}^{n} w_{j,i}$$

3. Create one output neuron, having m inputs (equal to the number of TLUs created in the previous steps), threshold equal to 1 and all weights equal to 2.

Exercise 1.2.4: 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:



□

1.3. Training TLUs

The aforementioned method for constructing a TLU consists in finding an n-dimensional hyperplane that separates a convex set into two subsets, one containing values for which the TLU outputs 1 and one containing values for which the TLU outputs 0. However, this method is feasible only if the dimension of the sets is small.

First of all, if the dimension of the sets is greater than 3, it's impossible to give it a visual representation. Secondly, this method requires a "visual inspection" of the set to identify the chosen line/plane, meaning that it is hardly possible to encode the process into an algorithm to be fed to a computer, and has to be carried out "by hand" instead. Finally, even if the number of dimensions is small, finding a linear separation for a set can still be tedious.

To construct an automated process that is capable of generate a TLU given a boolean function, a different approach is needed. The idea is to start with randomly generated values for the weights and the threshold of the TLU, trying out the TLU with input data to see if its outputs match the expected outputs, tuning the TLU parameters in accord if this isn't the case and repeating the process until the output of the TLU matches the output of the function. This process of stepwise tuning of the TLU is also referred to as the **training** of the TLU.

To achieve the goal of training a TLU, it is first necessary to quantify "how much" the outputs of the TLU and the outputs of the function to encode 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 outputs of the function. Clearly, when the output of the error function is 0, the original function and the encoded function of the TLU match perfectly. The goal is therefore to reduce the output of the function at any training step of the TLU until it becomes 0.

The most natural way to construct an error 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 them up. However, this approach would not be feasible, because it would create a stepwise error function,

meaning that, again, only visual inspection would be able to determine how to tune the weights and the threshold of the TLU so that the outputs match. This is due to the fact that stepwise functions are not minimizable, since they are not differentiable everywhere. One could try at random possible combinations of inputs and weights until one is found that zeros the error function, but in general this is not a feasible approach.

A better way to define such a function is to consider instead "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 formulations of the training process. The first consists in tuning the TLU with respect to the first input, then tuning the TLU with respect to the second input, and so on until a traning process is undergone for all inputs, then repeating from the first input if necessary: this is referred to as **online training**. The second consists in accumulating all the tunings 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**.

It is now possible to explicitly formulate an algorithm for the training process of the TLU. First, one should start from this observation: if the output of the TLU is 1 whereas the output of the function is 0, it must mean that the threshold of the TLU is too low and/or the weights of the TLU are too high. Therefore, if this happens, one should raise the threshold and lower the weights. On the other hand, if the output of the TLU is 0 whereas the output of the function is 1, it must mean that the threshold of the TLU is too high and/or the weights of the TLU are too low, and those should be tuned accordingly.

A single training step can be formulated as follows. Let $\boldsymbol{x} = (x_1, ..., x_n)$ be an input vector of a TLU, y the output of the function with $\boldsymbol{x}$ as input and $\hat{y}$ the output of the TLU with $\boldsymbol{x}$ as input. If $\hat{y} \neq y$, then the threshold θ and the weights $\boldsymbol{w} = (w_1, ..., w_n)$ of the TLU can be updated in accord to the following rule, called **delta rule**, or **Widrow-Hoff rule**:

$$\begin{cases} \theta & \leftarrow \theta - \eta(y - \hat{y}) \\ w_i & \leftarrow w_i + \eta(y - \hat{y})x_i, \forall i \in \{1, ..., n\} \end{cases}$$

The parameter η is called **learning rate**, and determines how much the threshold and weights are changed: at every step, they are increased or reduced by a factor of η. It shouldn't be set either too low, because the updates would be very slow, but should be too high either, because the new value of the parameters might jump to another slope of the error function.

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 $X_j = (x_{1,j}, ..., x_{m,j})$ 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
```

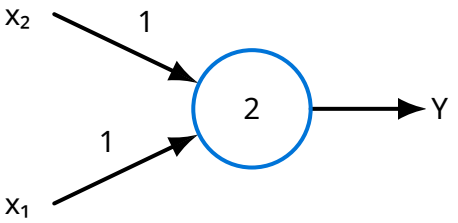
Exercise 1.3.1: Construct a TLU that computes the logical AND between two bits.

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 middle denote the training of the TLU, employing online learning and batch learning respectively. On the right, the graphical representation of the TLU obtained from batch learning.

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 function. Clearly, if the function to be encoded is not lineraly separable, the training process will never converge, since the error function will keep oscillating and never going to 0. However, if the function is linearly separable, the training process does always converge.

Theorem 1.3.1 (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:

$$\sum_{j=1}^{n} w_j X_j < \theta, \ \forall (X = (X_1, ..., X_n), 0) \in L_0 \quad \sum_{j=1}^{n} w_j X_j \geq \theta, \ \forall (X = (X_1, ..., X_n), 1) \in L_1$$

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

From this basic formulation, it is possible to look for improvements. First, note how the threshold tuning and the weights tuning are treated separately by the delta rule, since the two updates have opposite signs (negative and positive respectively). However, it is possible to simplify the formula by merging the two expressions into one, turning the threshold into an extra, "special" weight.

To do so, recall that the TLU outputs 1 if $\sum_{i=1}^{n} w_i x_i \geq \theta$ and 0 otherwise. However, this is equivalent to stating that the TLU outputs 1 if $\sum_{i=1}^{n} w_i x_i - \theta \geq 0$ and 0 otherwise. This, in turn, is equivalent to stating that the TLU outputs 1 if $\sum_{i=0}^{n} w_i x_i \geq 0$ and 0 otherwise, where the threshold is now 0 and θ was turned into $w_0 x_0$, a "ficticious" input and a corresponding weight. For the new and old expressions to be equivalent, it suffices to have x_0 always equal to 1 and w_0 equal to $-\theta$ or, equivalently, $x_0 = -1$ and $w_0 = \theta$.

It is now possible to restate the delta rule as follows. Let $\boldsymbol{x} = (x_0 = 1, x_1, ..., x_n)$ be an input vector of a TLU, y the output of the function with $\boldsymbol{x}$ as input and $\hat{y}$ the output of the TLU with $\boldsymbol{x}$ as input. If $\hat{y} \neq y$, then the weights $\boldsymbol{w} = (w_0 = -\theta, w_1, ..., w_n)$ of the TLU can be updated as follows:

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

Once the training process is over, it suffices to turn back w_0 into θ and to remove the input x_0 to obtain the actual formulation of the TLU.

A second improvement deals with the way Boolean functions are encoded. In the original formulation, the value of false is encoded as 0 and the value of true is encoded as 1. The problem of this encoding is that false inputs cannot influence the tuning of the weights, because the sum between weights and zero inputs is zero, slowing the training down. The problem can be circumvented by encoding true as $+1$ and false as -1, so that false inputs also contribute to the training. This is called the **ADALINE model** (**ADAptive LINear Element**).

Having devised a training method for single TLUs, it would be interesting to extend training to networks of TLUs. This would allow one to encode any kind of functions, not just linearly separable functions. Unfortunately, transferring the training process one-to-one from single TLUs to networks of 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, whereas the other TLUs are oblivious to the changes. This means that, to train a network of TLUs, a completely different approach is required.

1.4. 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 U of vertices is partitioned into three: a set U_{in} of **input neurons**, a set U_{out} of **output neurons** and a set U_{hidden} of **hidden neurons**. The set of hidden neurons can be empty, whereas the set of input and output neurons cannot. The set of input and output neurons may not be disjoint.

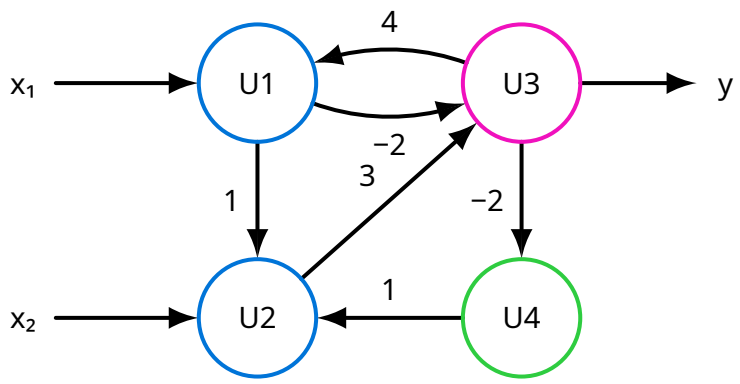
$$U = U_{\text{in}} \cap U_{\text{out}} \cap U_{\text{hidden}} \qquad U_{\text{in}} \neq \emptyset, U_{\text{out}} \neq \emptyset, U_{\text{hidden}} \cap (U_{\text{in}} \cap U_{\text{out}}) = \emptyset$$

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. Similarly, the output of a neural network is the output of all of its output neurons.

It is customary to denote the ending node of the connection before the starting node, and not vice versa. That is, a weight $w_{u,v}$ is carried by a connection ending in u and starting in v, not the other way around. The weights of a neural network are collected into a matrix where all the weights of connections that lead to the same neuron are arranged into the same row. This way, the neurons and their outgoing connections are to be read entrywise. The matrix and the corresponding weighted graph are called the **network structure**.

Exercise 1.4.1: 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 it both as matrix and as graph.

Solution:



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

□

If the graph describing a neural network is acyclic (has no loops and no directed cycles), it is referred to as a **feed forward neural network**. If, on the other hand, it is cyclic, it is referred to as a **recurrent network**. The difference between the two is the flow of information: in a feed forward neural network, the information can only flow from the input neurons to the hidden neurons (if any) to the output neurons, meaning that it can only go "forward", whereas in a recurrent network the information can be fed back into the network.

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:

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

Where $\sigma(u)$ and $\theta(u)$ are generic (real) parameters that depend on the type and on the number of arguments of the function.

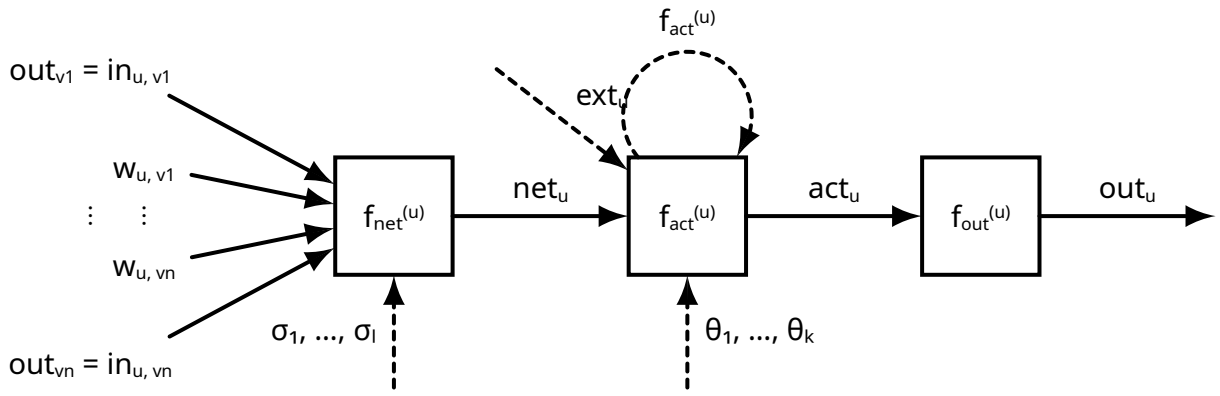


Figure 3: Structure of a neuron

The network input function process the inputs $\mathrm{in}_{u,v_1}, \ldots, \mathrm{in}_{u,v_n}$ of the neuron u, which are themselves the output $\mathrm{out}_{v_1}, \ldots, \mathrm{out}_{v_n}$ of other neurons, and the weights $w_{u,v_1}, \ldots, w_{u,v_n}$, merging the result into the network input net_u. The simplest formulation of the network input function is a weighted summation of the products of each weight and each input.

The network input is then fed into the activation function, that processes the "raw" network input into a degree of solicitation of the neuron. In some models of neurons, the activation is fed back to the

activation function itself. In the case of input neurons, the external input is merged with the activation. A notable example of parameter for activation functions is, as is the case for TLUs, a threshold.

The output function decides, based on the activation value it has been fed, what the output will be (whether the neuron will fire or not). In general, functions of this sort "quash" the network input in a "nicer" interval, and many functions with these traits exist (stepwise functions, logaritmic functions, ecc...). The simplest formulation of an output function is the identity function.

Exercise 1.4.2: Consider Exercise 1.4.1. Write a network input function, an activation function and an output function for all neurons.

Solution: Using the weighted sum of the output of their predecessors as inputs, the network input function can be written as:

$$f_{\text{net}}^{(u)}(w_{u,v1}, ..., w_{u,vn}, \text{in}_{u,v1}, ..., \text{in}_{u,vn}) = w_{u,v1} \cdot \text{in}_{u,v1} + ... + w_{u,vn} \cdot \text{in}_{u,vn} = \sum_{v \in \text{ pred}(u)} w_{u,v} \cdot \text{in}_{u,v}$$

Given a threshold θ, the activation function can be written as:

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

Using the identity function as output function:

$$f_{\text{out}}^{(u)}(\text{act}_u) = \text{act}_u$$

□

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 fed from outside, 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 not necessarely meaningful. The neural network does not move 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 blocked and the activations and outputs of the neurons are (re)computed, applying the network input function, the activation function and the output function in the described order. 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 order is which recomputations are carried out varies from neural network to neural network. 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, as no redundant computations are carried out in this way. 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.

Exercise 1.4.3: Consider Exercise 1.4.2. Let the initial output be $x_1 = 1$, $x_2 = 0$. Does the neural network reach a stable state if employing the ordering u_4, u_3, u_1, u_2? And how about the ordering u_4, u_3, u_2, u_1?

Like TLUs, neural networks can also be trained, by tuning its weights and its parameters so that a certain criterion is optimized (that is, an error function of sort is minimized). The way a neural network is trained depends on the optimization criteria and on the type of the training data, but all training tasks can be distinguished into two types: 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 $l = \boldsymbol{i}^{(l)}, \boldsymbol{o}^{(l)}$, each consisting of an **input vector** $\boldsymbol{i}^{(l)} = \text{ext}_{u1}^{(l)}, ..., \text{ext}_{un}^{(l)}$ and an **output vector** $\boldsymbol{o}^{(l)} = o_{v1}^{(l)}, ..., o_{vm}^{(l)}$.

A fixed learning task prescribes training a neural network such that its output (the output of its output neurons) is, for all training patterns $l \in L_{\text{fixed}}$, as close as possible to the output vector $\boldsymbol{o}^{(l)}$ when fed $\boldsymbol{i}^{(l)}$ as external input.

Unlike TLUs, training neural networks almost surely necessitates some degree of approximation. This is quantified by an error function, an estimate of the average deviation between the outputs of the network (the "estimated" outputs) and the outputs from the data (the "actual" outputs). The error function should not be computed from pattern to pattern, but instead after all the patterns are presented to the network, so that the result takes all of them into account and the result does actually converge.

A fixed learning task is considered complete when the value of the error function is sufficiently small. This is done by repeating the input and work phase of the neural network over and over. 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.

Of course, simply taking the difference between the outputs of the network and the outputs from the data does not make a good error function, since positive and negative errors may even each other out. A common choice for the error function for fixed learning tasks is the **Mean Squared Error function (MSE)**:

$$e = \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{l \in L_{\text{fixed}}} \left(o_{v1}^{(l)} - \text{out}_{v1}^{(l)}\right)^2 + ... + \left(o_{vm}^{(l)} - \text{out}_{vm}^{(l)}\right)^2 = \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \text{out}_v^{(l)}\right)^2$$

That is, the sum over all training examples of the squared difference between the outputs in the given data 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 (computing its derivative and setting it to 0).

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 $l = \boldsymbol{i}^{(l)}, \boldsymbol{o}^{(l)}$, each consisting of an **input vector** $\boldsymbol{i}^{(l)} = \text{ext}_{u1}^{(l)}, ..., \text{ext}_{un}^{(l)}$.

In free learning tasks, the network does not have a output vector to compare its output with, and has some degree of freedom (hence the name "free") in choosing its outputs. However, this does not

means that said outputs should be random; instead, a neural network should strive to produce similar outputs for similar inputs. Ideally, similar outputs should be clustered into highly coese groups, with little distance between its members.

Free learning tasks are also referred to as **unsupervised learning** since, unlike supervised learning, there is no counterexample (no "guidance") to test whether the output of the neural network is desireable or not.

It is advisable to normalize the inputs of a neural network, especially with respect to the way neural networks are trained: if some of the inputs are order of magnitude bigger than the others, those inputs will skew the training of the network in their favour. Normalizing the inputs of a neural network entails, as expected, dividing each input by the mean of the input and dividing the result by the standard deviation:

$$\mathrm{ext}_{uk}^{(l)(\mathrm{new})} = \frac{\mathrm{ext}_{uk}^{(l)(\mathrm{old})} - \mu_k}{\sigma_k} = \frac{\mathrm{ext}_{uk}^{(l)(\mathrm{old})} - \frac{1}{|L|}\sum_{l \in L} \mathrm{ext}_{uk}^{(l)}}{\sqrt{\frac{1}{|L|}\sum_{l \in L}\left(\mathrm{ext}_{\mu_k}^{(l)} - \mu_k\right)}}$$

This way, the arithmetic mean of the input will be 1 and the variance will be 0. This normalization can be carried out as a preprocessing step or (in a feed forward network) by the output function of the input neurons.

It is reasonable to deal with neural networks having (real) numbers as input and output. However, it is possible to have neural networks manipulate nominal attributes. A reasonable assumption would be to associate an integer to each possible value of the attribute, but this is a poor choice, because it makes little sense to use an encoding implying an order when the attribute does not. A better approach is what is called **1-in-n encoding**, where each value of the attribute is assigned a binary string of length equal to the number of possible attributes constituted of all 0 except for a single 1. This way, all possible values are equally taken into account.

1.5. Multilayer perceptrons

A **multilayer perceptron** (**MLP**) is a particular type of feed-forward neural network $G = (U, C)$ whose neurons can be partitioned into r layers. An input neuron of a multilayer percepteron cannot also be an output neurons, and vice versa. That is, the two sets are disjoint:

$$U_{\mathrm{in}} \cap U_{\mathrm{out}} = \emptyset$$

Hidden neurons of an MLP can be partitioned into $r - 2$ layers, disjointed with each other:

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

Connections in an MLP can only exist between nodes of subsequent layers, not even between nodes of the same layer. The maximum number of connections is as many connections that can be formed by connecting each neuron with all the neurons of the subsequent layer:

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

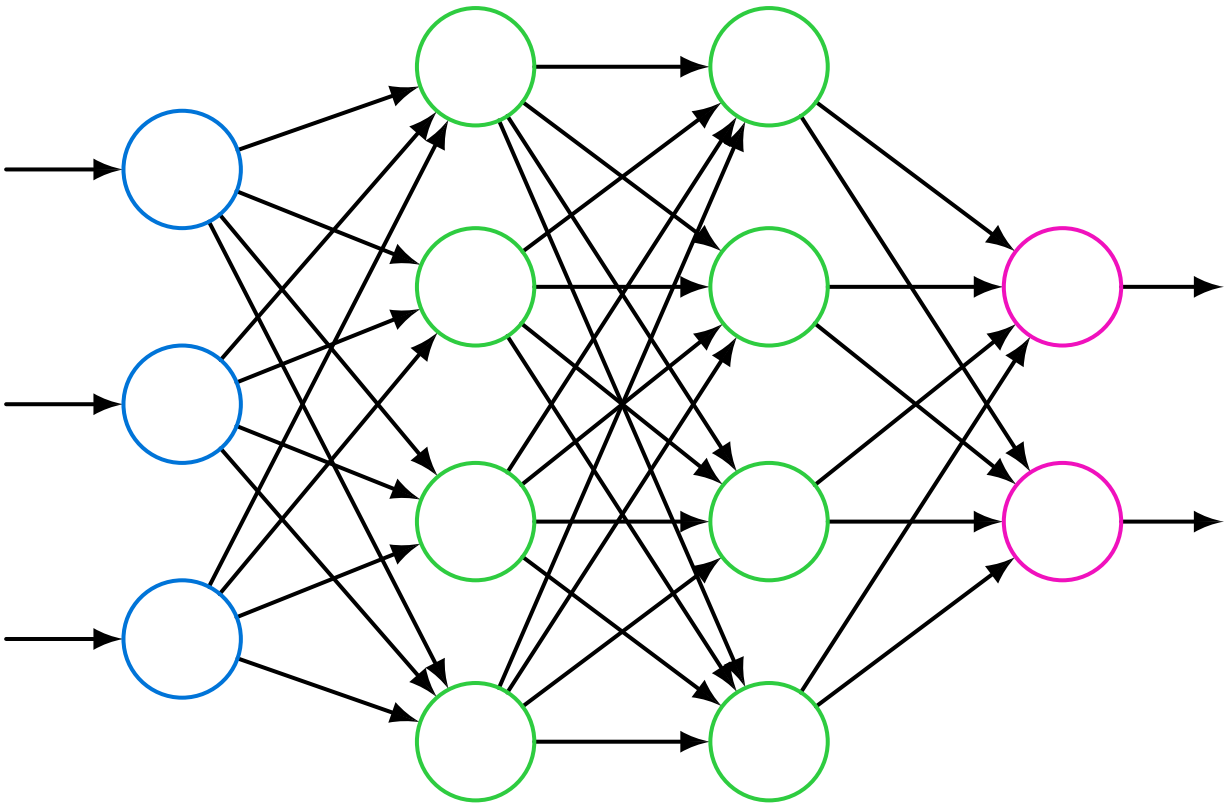


Figure 4: Structure of a generic multilayer perceptron

Input neurons have their input entirely specified by the external input; no input comes from other neurons. Their only purpose is to propagate unchanged the external input to the first hidden layer. In other words, the network function, the activation function and the output function of input neurons are the identity function.

Hidden neurons and output neurons have, as network input function, the weighted sum of their inputs and the corresponding weights:

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

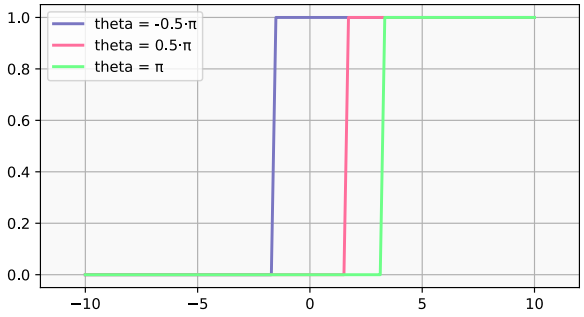
The activation function of hidden neurons is any **sigmoid function**, meaning a monotonic non-decreasing function of the form:

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

Function of this sort have a characteristic S-shape. Examples of this function are:

- The **Heaviside function**, or **step function**, that returns 1 for all values greater than a given argument θ and 0 otherwise. It has the advantage of being very easy to conceptualize, and it is also very efficient to implement it in hardware:

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

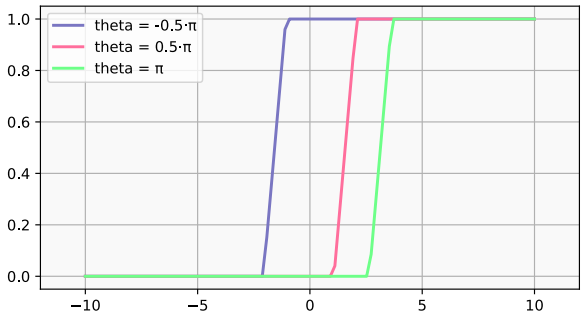


This is because, as it was done for the TLUs, it is possible to move the threshold into the weighted sum and obtain an equivalent function that output 0 if the weighted sum is negative (less than 0) and outputs 1 if positive (greater than 0), and this check can be done by simply looking at the most significant bit of the result of the weighted sum[2]. 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, it is sufficient to perform a negation on the most significant bit of the weighted sum and read the result.

The problems of the function line in its abrupt jump, both from a mathematical standpoint, since the step renders the function not differentiable, and from a logical standpoint, since it models neurons that either fire or not fire, without nuances in between. Also, this function is not invertible, since it is not injective.

- The **semi-linear function**, that grows linearly inside an interval and remains constant outside of those boundaries:

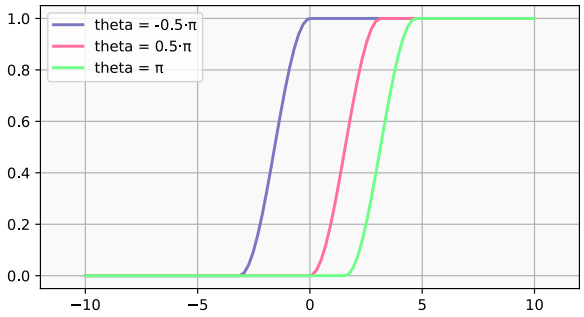
$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1 & \text{if net} > \theta + \frac{1}{2} \\ 0 & \text{if net} < \theta - \frac{1}{2} \\ (\text{net} - \theta) + \frac{1}{2} & \text{otherwise} \end{cases}$$



This function improves the Heaviside function "smoothing" the transition between the two extremes, increasing the expressing power of the model, but still presents problems. For example, it is still not injective, and therefore not invertible.

- The **sine up to saturation function**, that grows trigonometrically inside an interval and remains constant outside of those boundaries:

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

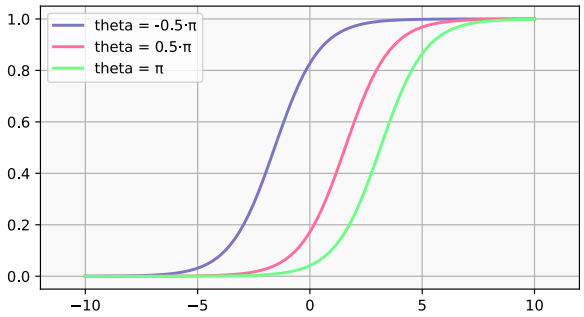


[2]Weighted sums can be computed efficiently by GPUs, since they are specifically designed to efficiently compute convolutions.

The growth of the function is even smoother, and the derivative grows smoothly as well, but it is still not invertible.

- The **logistic function**[3], which was the first historic example of a widely deployed activation function:

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



This function is not only continuous everywhere, but also differentiable everywhere. In particular, its derivative is particularly easy to compute:

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

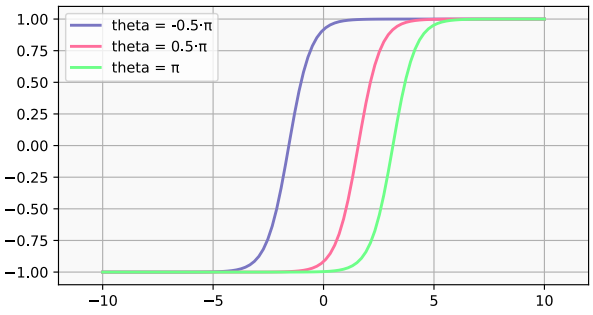
That is, it is just itself minus itself squared. Being injective, it is also invertible:

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

Sigmoid functions having $[0, 1]$ as codomain are called **unipolar sigmoid functions**. Functions having all the traits of a sigmoid function but having codomain $[-1, 1]$ instead are still considered sigmoids, and are called **bipolar sigmoid functions**. One notable example is the **hyperbolic tangent**, conceptually similar to the logistic function:

[3] 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.

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



Any unipolar function can be converted into a bipolar functions simply by multiplying by 2 and subtracting 1. As a matter fact, the codomain can be shifted and scaled as will, as long as its extremes are finite and as longs as the weights are tuned in accord. The only thing that matters is modelling a threshold that, until reached, blocks the stimulation of the neuron.

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 $\alpha \in \mathbb{R}$.

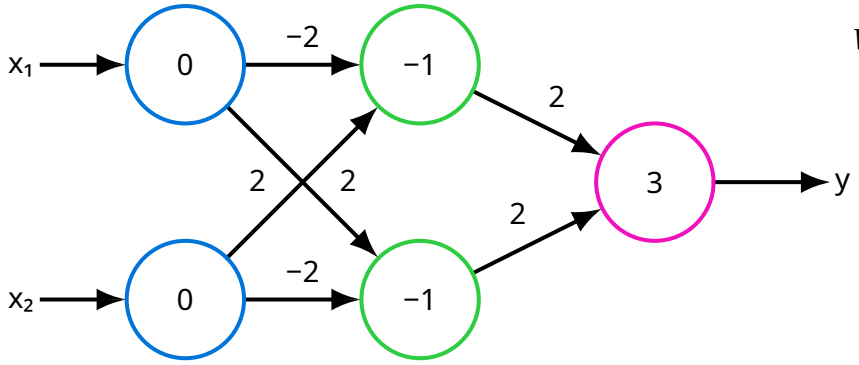
A clear advantage of having a weighted summation as the network input function of a multilayer perceptron is that it translates naturally to matrix multiplication. Let $U_1 = (v_1, ..., v_m)$ and $U_2 = (u_1, ..., u_n)$ be two subsequent layers (U_2 is right after U_1). It is possible to write the network input function for this layer as:

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

Where w_{u_i,v_j} is the weight of the connection between the j-th node of U_1 and the i-th node of U_2. If such connection does not exist, $w_{u_i,v_j} = 0$.

Exercise 1.5.1: Construct a multilayer perceptron that computes the Boolean expression $A \iff B$, rewriting the network of threshold logic units.

Solution: It is sufficient to write the activation function as the identity function.



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

$$W_{U_{\text{out}},U_{\text{hidden}}} = (2 \;\; 2)$$

□

Multilayer perceptrons allow one to approximate functions that aren't binary, but are real-valued. In particular:

Theorem 1.5.1: Any Riemann-integrable function can be approximated with arbitrary accuracy by a multilayer perceptron of four layers.

A perceptron of this kind can be constructed as follows. The four layers are the input layer, the output layer and two hidden layers. The first layer (the input layer) is a layer consisting of a single neuron, receiving the point of the function that one wishes to approximate. The fourth layer (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.

Consider an arbitrary function f. It is possible to partition its domain into n steps, delimited by the values $x_1, x_2, ..., x_n$ along the x axis. For each of these cutoff points, a node in the first layer of the perceptron is added. The weights of the incoming connections of said nodes are set to 1, and the threshold of these nodes is the cutoff point itself.

This way, only neurons having as threshold the cutoff points that are smaller than the given input will fire. Suppose $\overline{x}$ is fed into the network, and suppose that $x_1 \leq x_2 \leq ... \leq x_i \leq \overline{x}$. The neurons of the first layer that will fire are the ones having as threshold $x_1, x_2, ..., x_i$.

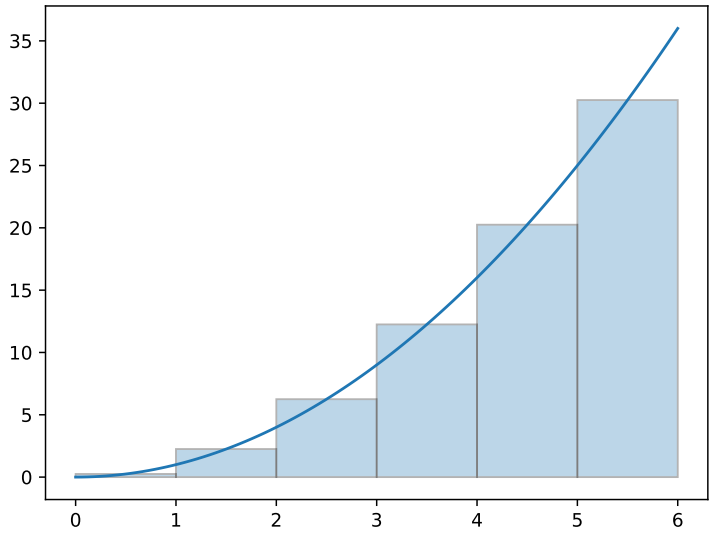
Each pair of adjacent cutoff points induces $n - 1$ intervals $[x_1, x_2], [x_2, x_3], ..., [x_{n-1}, x_n]$. Each of these intervals will (more or less accurately) give an approximation for all the values of the true function in said range (the most natural choice of this approximation is the middle point of the interval). For each of these intervals, the second hidden layers contains a neuron; the incoming weights and their thresholds are chosen so that a single neuron of the layer will be firing.

This neuron will be the one associated to the interval that contains the given input to approximate. Suppose $\overline{x}$ is fed as input, and the firing neurons of the first hidden layer are the ones having as threshold $x_1, x_2, ..., x_i$. The first, second, ..., up to $i - 1$-th neuron of the second hidden layer will not fire, because the incoming weights cancel out. The $i + 1$-th up to $n - 1$-th neuron of the second hidden layer will also not fire, since their inputs is 0. The only neuron that will fire is the i-th, because the neuron of the first hidden layer having x_i as threshold will give a positive contribution, whereas the neuron of the first hidden layer having x_{i+1} as threshold will not give any contribution.

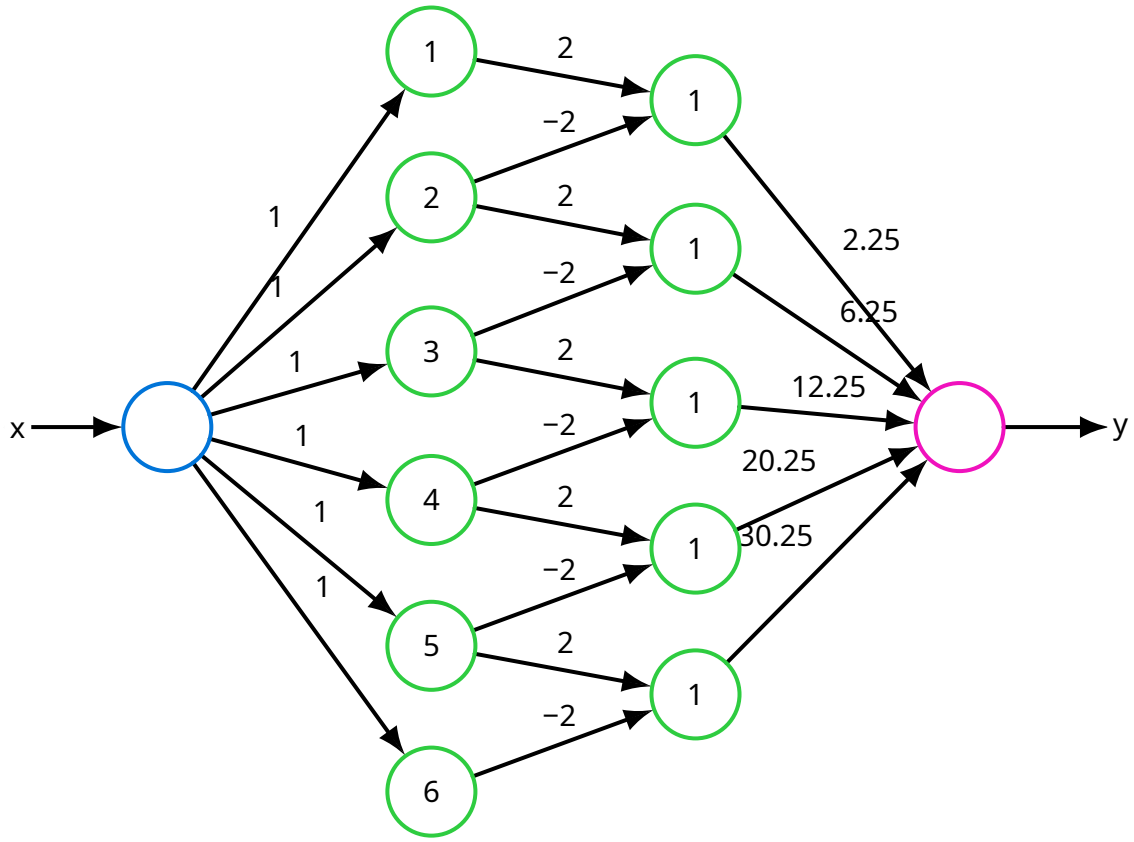
From the output of the network it is possible to know which is the best approximation for a given input, since each of the incoming weights of the input neuron is set to the chosen approximations of the function evaluated at the given input, and only one neuron of the second hidden layer will fire.

Exercise 1.5.2: Consider the function $f(x) = x^2$. Construct a multilayer perceptron that can approximate said function.

Solution: Suppose 6 steps going from 0 to 6 of uniform size. Evaluating the function at the midpoints gives: $2.25, 6.25, 12.25, 20.25, 30.25$.



Which is equivalent to the following multilayer perceptron:



□

Note that Theorem 1.5.1 does not restrict itself to continuous functions; there exist Riemann-integrable functions that present discontinuities[4], but a multilayer perceptron will still be able to approximate it. However, a continuous function is easier to approximate:

[4]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.

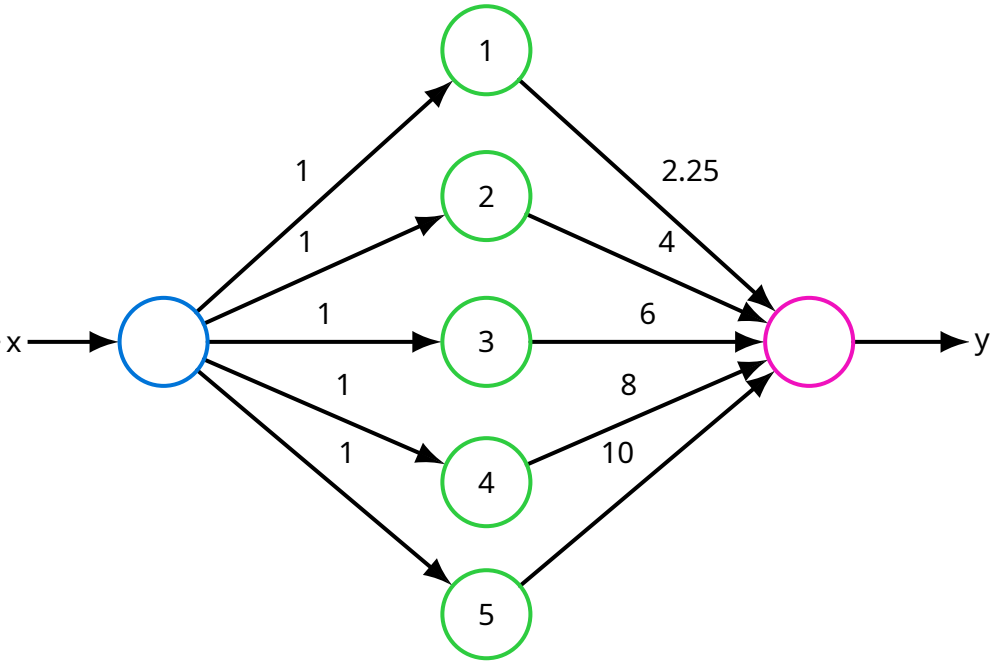
Theorem 1.5.2: Any continuous Riemann-integrable function can be approximated with arbitrary accuracy by a multilayer perceptron of three layers.

This can be done by encoding into the multilayer perceptron not the absolute height of a step, but the relative height: the difference between the current step and the previous step. This perceptron is analogous to the previous one, except for the second hidden layer, which is removed, connecting the hidden layer directly to the output neuron. The ouputs of the hidden layer are the relative height of the steps.

This way, the first part of the computation behaves just as in the previous case, only the neurons having as threshold a value smaller than the given input will fire. But now, the differences in height are added directly, reconstructing the height of the correct step. Clearly, applying this shortcut to non-continuous functions would not work, because there is no guarantee that the relative height at a certain step is actually the sum of the previous relative heights.

Exercise 1.5.3: Consider Exercise 1.5.2 and construct an equivalent three layer perceptron.

Solution: Computing the relative heights of the steps gives: $2.25 - 0 = 2.25, 6.25 - 2.25 = 4, 12.25 - 6.25 = 6, 20.25 - 12.25 = 8, 30.25 - 20.25 = 10$.



□

Even though Theorem 1.5.1 guarantees that any function can be approximated by a multilayer perceptron, the theorem itself isn't really useful. Clearly, the accuracy of the prediction of can be increased arbitrarely by increasing the number of neurons (that is, the number of steps) used in the hidden layers. The issue is that, to get a satisfying degree of approximation, it is necessary to construct a multilayer perceptron with many neurons (which means, choosing many steps), and this effort might outvalue the purpose.

There are ways, however, to improve the degree of approximation without resorting exclusively to reducing the step size. For example, choosing an activation function for the hidden layers that is not the Heaviside function (like, say, the logistic function) might better model the shape of the function at hand. A complementary approach would be to use step widths that aren't uniform, but that scale with the skewdness of the function. That is, using many steps where the function is heavily curved (and thus a linear approximation is poor) and little steps where it is almost linear.

Note that the degree of approximation in Theorem 1.5.1 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 as stated, 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.

1.6. 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:

$$\not{2}\sum_{i=1}^{n} a + bx_i - y_i = 0 \Rightarrow \sum_{i=1}^{n} a + \sum_{i=1}^{n} bx_i - \sum_{i=1}^{n} y_i = 0 \Rightarrow na + b\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$$

$$\not{2}\sum_{i=1}^{n} (a + bx_i - y_i)x_i = 0 \Rightarrow \sum_{i=1}^{n} ax_i + \sum_{i=1}^{n} bx_i^2 - \sum_{i=1}^{n} x_i y_i = 0 \Rightarrow a\sum_{i=1}^{n} x_i + b\sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} x_i y_i$$

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:

$$\begin{aligned} na_0 + a_1\sum_{i=1}^{n} x_i + ... + a_m\sum_{i=1}^{n} x_i^m &= \sum_{i=1}^{n} y_i \\ na_0\sum_{i=1}^{n} x_i + a_1\sum_{i=1}^{n} x_i^2 + ... + a_m\sum_{i=1}^{n} x_i^{m+1} &= \sum_{i=1}^{n} x_i y_i \\ &\vdots \\ na_0\sum_{i=1}^{n} x_i^m + a_1\sum_{i=1}^{n} x_i^{m+1} + ... + a_m\sum_{i=1}^{n} x_i^{2m} &= \sum_{i=1}^{n} x_i^m y_i \end{aligned}$$

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

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

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:

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

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 1.6.1: 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:

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

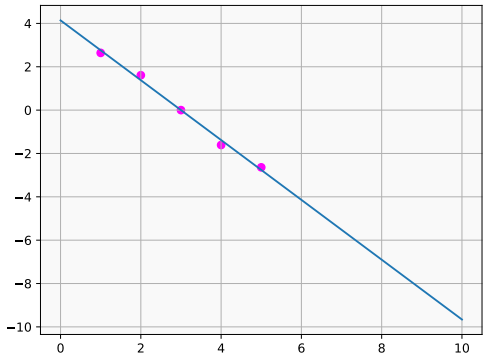
$$\sum_{i=1}^{n} x_i^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 55 \qquad \sum_{i=1}^{n} x_i \tilde{y}_i = 1 \cdot 2.64 + 2 \cdot 1.61 + 3 \cdot 0.00 - 4 \cdot 1.61 - 5 \cdot 2.64 \approx -13.78$$

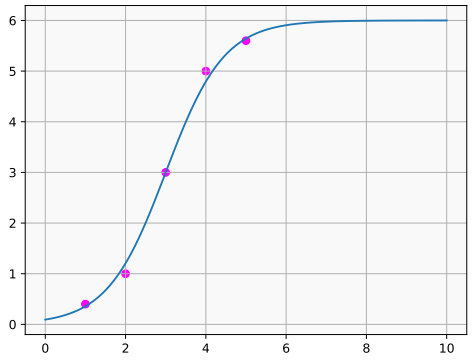
Leads to the following system of equations:

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

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.

1.7. Gradient descent

A more general approach for tuning the parameters of the multilayer perceptron is through the mathematical technique of **gradient descent**. As it was the case for TLUs, the idea is to compute the error function after a training task, zeroing the derivative of the error function and tune the parameters of the multilayer perceptron in accord to the result.

Note that the error function is likely to have arity greater than one, meaning that one should be computing the gradient, not the derivative. However, it might not be possible to zero the gradient of the error function, because it might not be solvable analytically. For this reason, the method of gradient descent is used, computing the gradient in one point, moving a tiny step in the opposing direction (the direction of the gradient is the direction of growth of the function) and repeating the process until a sufficient approximation is reached.

In the case of TLUs this wasn't possible, because the error function was not differentiable (it consisted of plateaus). However, the error function of a multilayer perceptron, as long as its neurons use a differentiable activation function, is itself differentiable, meaning that this does not constitue a problem.

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. The total error for a fixed learning task L_{fixed} is given by:

$$e = \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{l \in L_{\text{fixed}}} \left(o_{v1}^{(l)} - \text{out}_{v1}^{(l)}\right)^2 + ... + \left(o_{vm}^{(l)} - \text{out}_{vm}^{(l)}\right)^2 = \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \text{out}_v^{(l)}\right)^2$$

To understand how one should update 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.

Consider a single neuron u belonging to either an hidden or the output layer, that is $u \in U_k$ with $0 < k < r$. Its predecessors are given by $\text{pred}(u) = \{p_1, ..., p_n\} \in U_{k-1}$. The corresponding vector of weights, threshold embedded, is $\boldsymbol{w}_u = \left(-\theta_u, w_{u,p_1}, ..., w_{u,p_n}\right)$. The gradient of the total error function with respect to these weights is:

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

Substituting the expression for e gives:

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

Consider a single training pattern l and its error $e^{(l)}$. This error depends on the weights in $\boldsymbol{w}_u$ only via the network input

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

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. To compute $\delta_u^{(l)}$, a distinction between the output layer and the hidden layers ought to be made.

Computing $\delta_u^{(l)}$ for the output layer is particularly easy, because clearly the outputs of the output neurons are, by definition, independent of each other. This means that all the terms of the sum having $v \neq u$ vanish, giving:

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

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

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

Where the minus sign means that one should move in the opposite direction of the gradient in order to minimize it. The parameter η 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 the "best" choice problem domain specific.

Recall that, as stated before, this is only the weight change that results from a single training pattern l. In other words, this is how weights are adapted in online training, where the weights are adapted immediately after each training pattern. For batch training, one has to sum the changes described by the formula over all training patterns rather than changing the parameters immediately, since the weights are adapted only at the end of an epoch.

Also note that the derivative of $\text{out}_u^{(l)}$ with respect to $\text{net}_u^{(l)}$ cannot be calculated in the general case, since the output is computed from the activation function, which in turn is computed from the network input function, and the shape of those functions can vary.

Exercise 1.7.1: Consider a neuron u of the output layer of a multilayer perceptron in a given training example l. Suppose that the activation function of choice is the logistic function with parameter $\theta = 0$ and the output function is the identity. What would be the explicit expression for $\Delta_{w_u^{(l)}}$?

Solution: Recall that the derivative of the logistic function is equal to itself times one minus itself:

$$\begin{aligned} \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} &= \frac{\partial f_{\text{out}} \left(\text{act}_u^{(l)} \right)}{\partial \text{net}_u^{(l)}} = \frac{\partial \text{act}_u^{(l)}}{\partial \text{net}_u^{(l)}} = \frac{\partial f_{\text{act}} \left(\text{net}_u^{(l)} \right)}{\partial \text{net}_u^{(l)}} = f_{\text{act}} \left(\text{net}_u^{(l)} \right) \left(1 - f_{\text{act}} \left(\text{net}_u^{(l)} \right) \right) = \\ &= \text{act}_u^{(l)} \left(1 - \text{act}_u^{(l)} \right) = \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)} \right) \end{aligned}$$

Which gives:

$$\Delta_{w_u^{(l)}} = \eta \left(o_u^{(l)} - \text{out}_u^{(l)} \right) \frac{\partial \text{out}_u^{(l)}}{\partial \text{net}_u^{(l)}} \mathbf{in}_u^{(l)} = \eta \left(o_u^{(l)} - \text{out}_u^{(l)} \right) \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)} \right) \mathbf{in}_u^{(l)}$$

□

2. Fuzzy logic

2.1. Fuzzy sets

Boolean logic assumes that any proposition can be given a truth value of either *true* or *false*, or 1 or 0, with no room for ambiguity in between. Technically speaking, given a set X of propositions, the subset $M \subseteq X$ of true propositions is constructed from X by its **characteristic function** I_M:

$$I_M : X \mapsto [0,1], I_M = \begin{cases} 1 \text{ if } x \in M \\ 0 \text{ otherwise} \end{cases} \qquad M = \{x \mid x \in X, I_M(x) = 1\}$$

Exercise 2.1.1: Consider the set of natural numbers $\mathbb{N}$. Let $M \subseteq \mathbb{N}_0$ be the set of even natural numbers. What would be its characteristic function?

Solution: It can be written as $I_M(x) = (x+1) \bmod 2$, since the remainder of the division between an even number and 2 is 0 whereas the remainder of the division between an odd number and 2 is 1. □

However, modelling everyday life human propositions this way will most likely be unfruitful. Fixing thresholds for determining whether a certain proposition is true or false is most likely impossible, both because they hardly exist and, even if they do, they are not agreed upon, and depend either from context to context or from person to person. Also, natural language widely employs adverbs such as somewhat, likely, almost, ecc... that aren't really reflected in binary logic and that induce a blurred line between truth and falseness. Finally, it should be noted that it would make little sense, in everyday life, to treat truth values are threshold overcomings: people just have "hunches", yet this doesn't (and shouldn't) stop one from giving nuanced definitions of true and false.

Exercise 2.1.2: Is there an example of natural language propositions for whom is hard to assign a binary truth value?

Solution: Consider the definition of "hot". It is hard to state if the proposition *the weather is hot* is either true or false, since there is no set definition of what it means for the wheather to be hot. That is, there really isn't an indicator function for "hotness".

The problem could seemingly be sidestep by fixing a certain threshold and formulating the indicator function of "hotness" as, say:

$$I_H(w) = \begin{cases} 1 \text{ if the temperature is greater or equal than 25 degrees} \\ 0 \text{ otherwise} \end{cases}$$

However, this approach would poorly model reality for at least three reasons:

- The cutoff point of 25 degrees, or any cutoff point for that matter, is chosen completely arbitrarely. In reality, every person has its own way of determining if the weather is or isn't hot;
- This would mean that if the temperature is 24.9 degrees the weather should be considered just as cold as it would be if the temperature was 0 degrees, or any temperature below 25 degrees;
- Even if it were possible to unambiguously agree upon a cutoff point, it would still be impractical, since no one states that the weather is hot by checking the temperature, it is just an intuitive feeling.

□

The idea behind fuzzy sets is to introduce the idea of "partial" membership. That is, in contrast to classical logic where an element either is or is not a member of a set, elements of a fuzzy set have a number assigned that quantifies "how much" they belong to said set.

More formally, let X be a set. A **fuzzy subset** μ of X, or simply a **fuzzy set** μ of X, is a mapping $\mu : X \mapsto [0, 1]$ that assigns to each member $x \in X$ a **degree of membership** $\mu(x)$ to the fuzzy set μ. The set of all fuzzy sets for a given set X (the "power set" of fuzzy sets) is denoted as $\mathcal{F}(X)$.

When $\mu(x) = 1$, it means that x has complete membership with respect to μ, whereas if $\mu(x) = 0$ it means that x has complete non-membership with respect to μ. Characteristic functions can therefore be considered as special cases of fuzzy sets, having only 0 and 1 as possible outputs.

Even though values of 0 and 1 for $\mu(x)$ have a reasonable ontological interpretation, a value $\mu(x) \in (0, 1)$ begs the question: what does it mean, exactly, for an element to be partially a member of a set? It might seem natural to interpret $\mu(x)$ as a probability value, since the range of possible values is $[0, 1]$. That is, to interpret $\mu(x)$ as a probability distribution that assigns a probability $\mu(x)$ to each $x \in X$ of finding x in the set μ.

However, giving such interpretation would be wrong both from a mathematical perspective and an onthological perspective. First of all, $\mu(x)$ does not validate (in general) all the axioms of probability, for example $\int_{-\infty}^{+\infty} \mu(x)dx$ might not be equal to 1. Even if fuzzy sets were to be restricted to only consider functions that satisfy the Kolmogorov axioms, it would still be ill-advised to interpret fuzzy sets as probabilities: fuzzy sets model how closely a property or a statement is satisfied, whereas probability models the certainty of an event to happen or not.

The most used interpretations of fuzzy sets are the following three:

- **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 2.1.3: Consider two bottles of water, A and B. Bottle A has a 0.0004 probability of actually being a bottle of chlorine, whereas bottle B has a degree of membership of 0.0004 with respect to the fuzzy set of chlorine bottles. Are the two the same?

Solution: No. Having a probability of 0.0004 of being a bottle of chlorine could be interpreted as, out of 10000 identically sampled bottles, 4 contain (only) chlorine and 9996 contain (only)

water. Having a degree of membership of 0.0004 with respect to the fuzzy set of chlorine bottles could be interpreted as the statement "bottle B contains chlorine" matching the definition "a bottle containing chlorine" with a degree of 0.0004, meaning that bottle B has almost nothing in common with a bottle of chlorine, containing a lot of water and an infinitesimal amount of chlorine. □

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

On the other hand, continuous fuzzy sets are tricky. A continuous fuzzy set μ is identified by a degree of membership $\mu(x)$ that is a continuous function, having codomain $[0, 1] \subset \mathbb{R}$. The most intuitive representation of a continuous fuzzy set consists in drawing the graph of its degree of membership function: this representation is also referred to as its **vertical representation**.

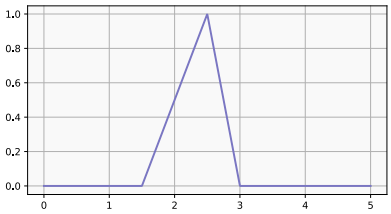
Out of all continuous fuzzy sets, the main interest lies in **convex fuzzy sets**, that best model natural language. Convex fuzzy sets are fuzzy sets having a degree of membership function that is monotonically increasing up to a certain point and monotonically decreasing after said point. Note that a fuzzy set being convex does not entail that its degree of membership function is also a convex functions; indeed, such functions are concave functions. Examples of functions with these characteristics are:

- **Triangular functions**

  $$\Lambda_{a,b,c} : \mathbb{R} \mapsto [0, 1]$$

  with $a < b < c$ and $a, b, c \in \mathbb{R}$.

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

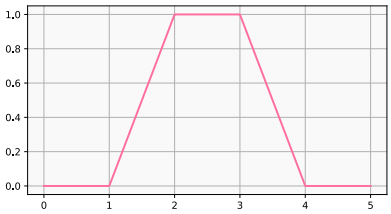


- **Trapezoidal functions**

  $$\Pi_{a,b,c,d} : \mathbb{R} \mapsto [0, 1]$$

  with $a < b \leq c < d$ and $a, b, c, d \in \mathbb{R}$. $a = b = -\infty$ and $c = d = +\infty$ are also valid.

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

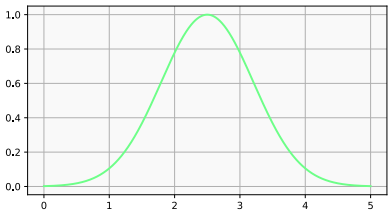


- **Bell-shaped functions**

  $$\Omega_{a,b} : \mathbb{R} \mapsto [0, 1]$$

  with $a, b \in \mathbb{R}$.

  $$\Omega_{a,b}(x) = e^{-\left(\frac{x-a}{b}\right)^2}$$



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. Another representation, called **horizontal representation**, employs so-called α-cuts.

Given a universe set X and a fuzzy set $\mu \in \mathcal{F}(X)$, let α be any number between 0 and 1. The subset $[\mu]_\alpha = \{x \in X \mid \mu(x) \geq \alpha\}$ is referred to as the **α-level set** of μ, or **α-cut** of μ. Similarly, the subset $[\mu]_{\underline{\alpha}} = \{x \in X \mid \mu(x) < \alpha\}$ is referred to as the **strict α-level set** of μ, or **strict α-cut** of μ. That

is, the α-cut of a fuzzy set is the subset that contains all its elements having degree of membership greater or equal than α.

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 can 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 convex sets.

α-cuts uniquely identify fuzzy sets: if all the α-cuts of a fuzzy set μ are known, the degree of membership $\mu(x)$ of an element x can be computed as:

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

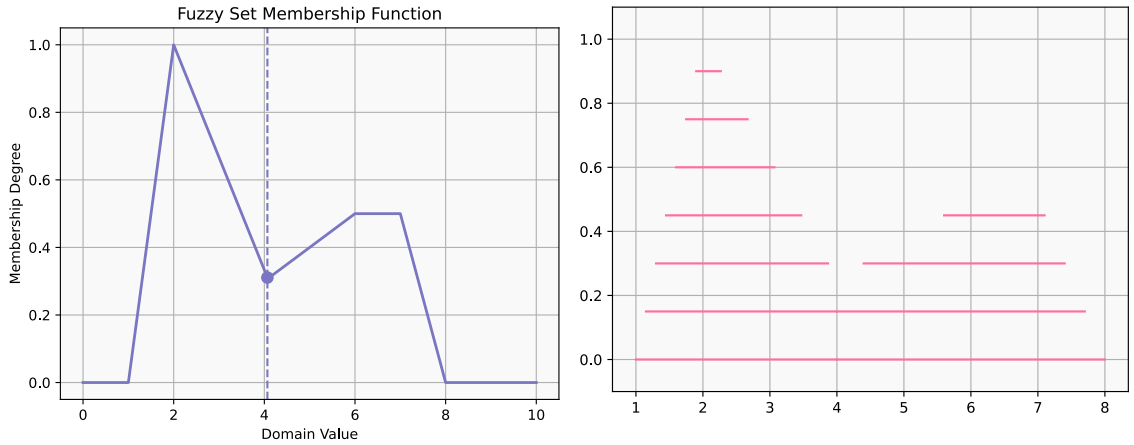
Of course, it would be pointless to store all the α-cuts of a fuzzy set, since it would be no different than storing the entire degree of membership function. However, discretizing the set of all α-cuts into a discrete subset $\left\{[\mu]_{\alpha_1}, [\mu]_{\alpha_2}, ..., [\mu]_{\alpha_k}\right\}$ is sufficient to reconstruct the entire fuzzy set with a surprising degree of accuracy:

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

Exercise 2.1.4: What would be the vertical and horizontal representation of the following degree of membership function?

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

Solution:



□

Formally, given a certain number of α-cuts, the original membership function can be reconstructed by taking the *upper envelope* of said cuts.

Theorem 2.1.1 (Representation theorem): Given $\mu \in \mathcal{F}(X)$ a fuzzy set over a universe set X:

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

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).

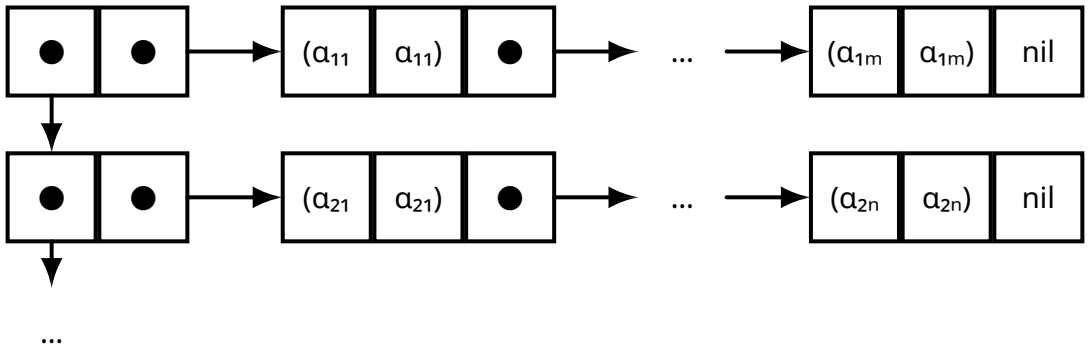


Figure 5: Linked list representation of α-cuts

Lemma 2.1.1: Given $\mu \in \mathcal{F}(X)$ a fuzzy set over a universe set X, $[\mu]_0 = X$.

Lemma 2.1.2: Let $\mu \in \mathcal{F}(X)$ be a fuzzy set over a universe set X, and let $\alpha, \beta \in [0,1]$. If $\alpha < \beta$, then $[\mu]_\alpha \supseteq [\mu]_\beta$.

Lemma 2.1.3: Let $\mu \in \mathcal{F}(X)$ be a fuzzy set over a universe set X. For any $\alpha, \beta \in [0,1]$, $\bigcap_{\alpha:\alpha<\beta} [\mu]_\alpha = [\mu]_\beta$.

The **support** of a fuzzy set $\mu \in \mathcal{F}(X)$ is the (standard) set that contains all of its members having non-zero degree of membership:

$$S(\mu) = [\mu]_{\underline{0}} = \{x \in X \mid \mu(x) > 0\}$$

The **core** of a fuzzy set $\mu \in \mathcal{F}(X)$ is the (standard) set that contains all of its members having membership exactly equal to 1:

$$C(\mu) = [\mu]_1 = \{x \in X \mid \mu(x) = 1\}$$

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

$$h(\mu) = \sup_{x \in X} \{\mu(x)\}$$

2.2. Fuzzy logic

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

Propositions are combined with each other using logical connectives, the most important being:

- `AND`, conjuction, denoted as $\wedge$;
- `OR`, disjounction, denoted as $\vee$;
- `IMPLIES`, implication, denoted as $\rightarrow$;
- `NOT`, negation, denoted as $\neg$.

The first three are binary operators, mapping the set $\{0,1\}^2$ to the set $\{0,1\}$, whereas the last one is unary, mapping $\{0,1\}$ to itself.

$$\wedge, \vee, \rightarrow\colon \{0,1\}^2 \mapsto \{0,1\} \qquad\qquad \neg : \{0,1\} \mapsto \{0,1\}$$

The truth value of propositions combined using logical connectives are evaluated using truth tables:

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

Modelling real-world propositions as having exclusively true or false truth values is often restrictive. It is however possible to extend classical logic to allow propositions to have more than two possible truth values: in such formulations, some statements can be neither true or false.

In particular, **fuzzy logic** is a logic formulation where the possible truth values is any real number in the interval $[0,1]$. That is, the more a truth value is close to 1 the more it is true, the more is close to 0 the more it is false. A truth value of $1/2$ corresponds to complete undeterminacy.

Logical connectives defined with respect to classical logic can be extended to be used in fuzzy logic. However, instead of mapping the set $\{0,1\}$ or the set $\{0,1\}^2$ to the set $\{0,1\}$, these extended operators ought to map the entire interval $[0,1]$ or the interval $[0,1]^2$ to the entire interval $[0,1]$:

$$\wedge, \vee, \rightarrow\colon [0,1]^2 \mapsto [0,1] \qquad\qquad \neg : [0,1] \mapsto [0,1]$$

The most widely employed definition of the logical conjuction and disjunction for fuzzy propositions are, respectively, their minimum and their maximum. In other words, $\alpha \wedge \beta = \min\{\alpha, \beta\}$ and $\alpha \vee \beta = \max\{\alpha, \beta\}$. The negation of a fuzzy proposition is generally given in term of its one-complement: $\neg\alpha = 1 - \alpha$. Implication is either defined as the **Łukasiewicz implication** or the **Gödel implication**, respectively:

$$\alpha \rightarrow \beta = \min\{1 - \alpha + \beta, 1\} \qquad\qquad \alpha \rightarrow \beta = \begin{cases} 1 \text{ if } \alpha \leq \beta \\ 0 \text{ otherwise} \end{cases}$$

Choosing specifically these functions to extend the logical connectives is not arbitrary. Indeed, these functions possess many properties that one expects a logical connective to have. Many more functions belong to the same family, therefore many reasonable choices could be made.

One additional requirement for choice of implementation of the logical connectives is the compatibility between them and their classical logical counterparts. That is, "fuzzy" `AND`, "fuzzy" `OR`, "fuzzy" `IMPLIES` and "fuzzy" `NOT`, when given (exactly) 0 or (exactly) 1 as input, should behave in the exact same way as "classical" `AND`, "classical" `OR`, "classical" `IMPLIES` and "classical" `NOT`, respectively.

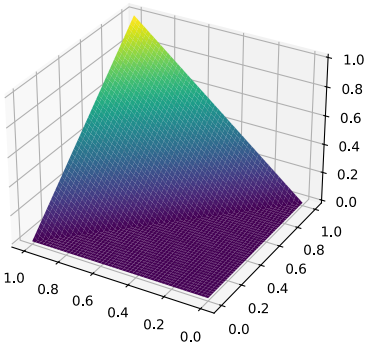
A function $t : [0,1]^2 \mapsto [0,1]$ is said to be a t-**norm**, or **triangular norm**, if it possesses the following properties:

- **Commutativity**: for any α, β, $t(\alpha, \beta) = t(\beta, \alpha)$;
- **Associativity**: for any α, β, γ, $t(t(\alpha, \beta), \gamma) = t(\alpha, t(\beta, \gamma))$;
- **Monotonicity**: for any α, β, γ, if $\beta \leq \gamma$ then $t(\alpha, \beta) \leq t(\alpha, \gamma)$;
- **Boundedness**: for any α, $t(\alpha, 1) = \alpha$.

It is advisable to choose a t-norm as a logical conjunction: indeed, the function $\min\{\alpha, \beta\}$ chosen to define $\alpha \wedge \beta$ is a t-norm. Other examples of t-norms are the:

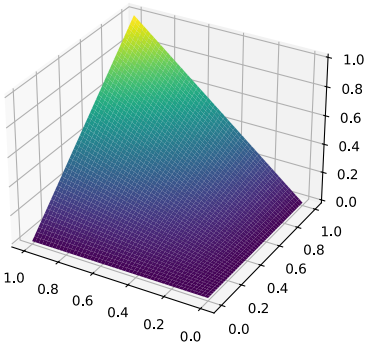
Łukasiewicz t-norm:

$$t(\alpha, \beta) = \max\{\alpha + \beta - 1, 0\}$$



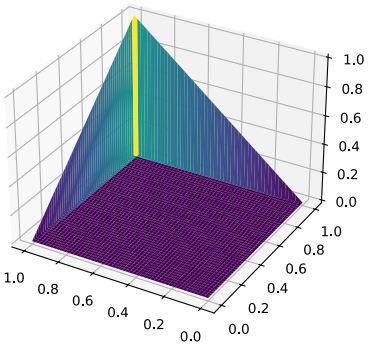
Algebraic product:

$$t(\alpha, \beta) = \alpha \cdot \beta$$



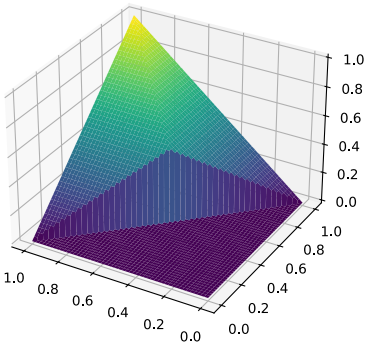
Drastic product:

$$t(\alpha, \beta) = \begin{cases} 0 & \text{if } 1 \notin \{\alpha, \beta\} \\ \min\{\alpha, \beta\} & \text{otherwise} \end{cases}$$



Nilpotent minimum:

$$t(\alpha, \beta) = \begin{cases} \min\{\alpha, \beta\} & \text{if } \alpha + \beta > 1 \\ 0 & \text{otherwise} \end{cases}$$



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 logical conjunction when giving 0 and/or 1 as input.

The family of t-norms is very broad: the only property of $\min\{\alpha, \beta\}$ that stands out among other t -norms, making it an appalling choice for the logical conjunction, is that it is **idempotent**, meaning that $t(\alpha, \alpha) = \alpha$ for all $\alpha \in [0,1]$. Even though idempotency can be a desirable property, it would be a mistake to take it for granted: there are scenarios where idempotency, meaning having min as the logical conjuction, poorly models the reality one intends to model.

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 α, $s(\alpha, 0) = \alpha$. Similarly to how it was done for the t-norm, it is advisable to choose a t-conorm as a logical disjunction, and the function $\max\{\alpha, \beta\}$ 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:

$$s(\alpha, \beta) = 1 - t(1-\alpha, 1-\beta) \qquad t(\alpha, \beta) = 1 - s(1-\alpha, 1-\beta)$$

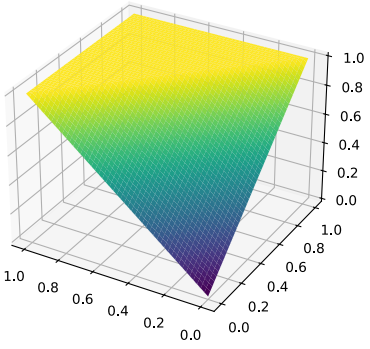
These relations are a generalization of the **De Morgan's Laws** for classical logic:

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

Applying the duality relation to the Łukasiewicz t-norm, the algebraic product and the drastic product one obtains the following conorms:

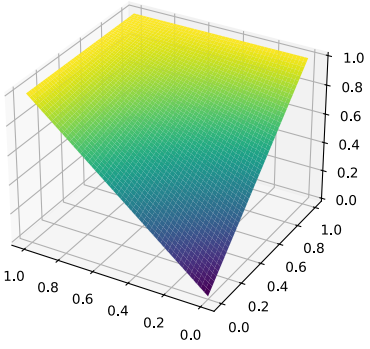
Łukasiewicz t-conorm:

$$s(\alpha, \beta) = \max\{\alpha + \beta, 1\}$$



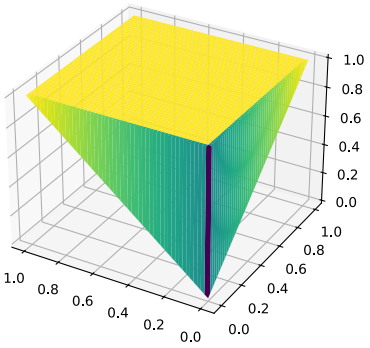
Algebraic sum:

$$s(\alpha, \beta) = \alpha + \beta - \alpha \cdot \beta$$



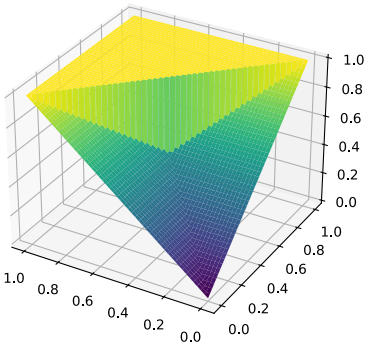
Drastic sum:

$$s(\alpha, \beta) = \begin{cases} 1 & \text{if } 0 \notin \{\alpha, \beta\} \\ \max\{\alpha, \beta\} & \text{otherwise} \end{cases}$$



Nilpotent maximum:

$$t(\alpha, \beta) = \begin{cases} \max\{\alpha, \beta\} & \text{if } \alpha + \beta < 1 \\ 1 & \text{otherwise} \end{cases}$$



Analogously to $\min\{\alpha, \beta\}$, $\max\{\alpha, \beta\}$ is the only t-conorm that is idempotent. Also, $\min\{\alpha, \beta\}$ and $\max\{\alpha, \beta\}$ 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:

$$\vec{t}(\alpha, \beta) = \sup\{\gamma \in [0,1] \mid t(\alpha, \gamma) \leq \beta\}$$

Indeed, the Łukasiewicz implication is obtained by substituting the Łukasiewicz t-norm in the aforementioned formula, whereas the Gödel implication is obtained by using $\min\{\alpha, \beta\}$.

2.3. Extending operators to fuzzy sets

As the name suggests, there exist a relationship between fuzzy sets and fuzzy logic. If the degree of membership of a fuzzy set describes "how much" an element possesses a certain property, the truth value of a fuzzy proposition describes "how truthful" it is to classifiy said element as a member of the set. That is, given an element $x \in X$ and some fuzzy set μ, $\mu(x)$ can be interpreted as the truth value of the fuzzy proposition "x is a member of μ". That is, $\mu(x) = [\![x \in \mu]\!]$.

This 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 instruments of classical set theory, like mappings and quantifiers, to fuzzy sets.

2.3.1. Intersection, union, complement

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. In the case of fuzzy sets, it is reasonable to assume that $(\mu_1 \cap \mu_2)(x)$, 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 degree of membership of x with respect to the two sets taken separately.

As stated, $\mu_1(x)$ and $\mu_2(x)$ should be interpreted as the truth value of the fuzzy propositions "$x \in \mu_1$" and "$x \in \mu_2$", respectively. Following this line of reasoning, $\mu_1(x) \wedge \mu_2(x)$ should be interpreted as the truth value of the fuzzy proposition "$x \in (\mu_1 \cap \mu_2)$". However, $\mu_1(x) \wedge \mu_2(x)$ can be given a more precise formulation, since logical conjunctions are well-modeled by t-norms. Therefore, having chosen a suitable t-norm t:

$$\mu_1(x) \wedge \mu_2(x) = (\mu_1 \cap \mu_2)(x) = [\![x \in (\mu_1 \cap \mu_2)]\!] = t(\mu_1(x), \mu_2(x))$$

And, assuming to choose the max function as the t-norm, one obtains $(\mu_1 \cap \mu_2)(x) = \min_{x \in X}\{\mu_1(x), \mu_2(x)\}$, as expected.

Employing a t-norm for the definition of the intersection between fuzzy sets implies that fuzzy set intersection inherits the four properties of a t-norm. This is important, because those mirrors the properties that classical set intersection possesses:

- Classical set intersection is commutative, so is fuzzy set intersection;
- Classical set intersection is associative, so is fuzzy set intersection;
- Given three classical sets A, B, C, if $A \subseteq B$ then $(A \cap C) \subseteq (B \cap C)$. This is mirrored in the monotonicity property;
- If $M \subseteq X$ is an ordinary subset of X and $\mu \in \mathcal{F}(X)$ is a fuzzy set of X, due to the boundedness property:

$$(\mu \cap I_M)(x) = \begin{cases} \mu(x) & \text{if } x \in M \\ 0 & \text{otherwise} \end{cases}$$

In the same way, it is possible to define the union of two fuzzy sets by picking a suitable t-conorm s:

$$\mu_1(x) \vee \mu_2(x) = (\mu_1 \cup \mu_2)(x) = [\![x \in (\mu_1 \cup \mu_2)]\!] = s(\mu_1(x), \mu_2(x))$$

Where max is the standard choice. Using max and min as definition of the fuzzy union and the fuzzy intersection 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, one has:

$$[\mu_1 \cap \mu_2]_\alpha = [\mu_1]_\alpha \cap [\mu_2]_\alpha \qquad\qquad [\mu_1 \cup \mu_2]_\alpha = [\mu_1]_\alpha \cup [\mu_2]_\alpha$$

To obtain the complement of a fuzzy set, note that $x \in \overline{M} \to \neg(x \in M)$ for any element x and any classical set M. By using $\neg\alpha = 1 - \alpha$ as truth function for the negation, one obtains $\overline{\mu}(x) = 1 - \mu(x)$; this is in accord with the fact that $[\![x \in \overline{\mu}]\!] = [\![\neg(x \in \mu)]\!]$.

Fuzzy set complement, like standard set complement, is **involutory**, meaning that applying it twice is equivalent to not applying it at all: $\overline{\overline{\mu}} = \mu$ for any fuzzy set μ. The standard set intersection of any set with its complement gives the universe set: fuzzy set complement "relaxes" this property as $(\mu \cap \overline{\mu})(x) \leq 0.5$ and $(\mu \cup \overline{\mu})(x) \geq 0.5$ for any fuzzy set μ and any element x.

2.3.2. Universal and existential quantifiers

Extending the universal quantifier $\forall$ and the existential quantifier $\exists$ can be done by building upon the process used to extend conjunction and disjunction, exploiting the relationship between these connectives and the quantifiers.

For a given set $X = \{x_1, ..., x_n\}$ and a predicate $P(x)$, the statement $(\forall x \in X)(P(x))$ is equivalent to $P(x_1) \wedge ... \wedge P(x_n)$. That is, $P(x)$ is true for all members of X if and only if it is true for each member of X individually. This means that $(\forall x \in X)(P(x))$ can be extended in the following way:

$$[\![\forall x \in X : P(x)]\!] = [\![P(x_1) \wedge ... \wedge P(x_n)]\!] = \min\{[\![P(x)]\!] \mid x \in X\}$$

Analogously, the statement $(\exists x \in X)(P(x))$ is equivalent to $P(x_1) \vee ... \vee P(x_n)$, therefore $(\exists x \in X)(P(x))$ can be extended as:

$$[\![\exists x \in X : P(x)]\!] = [\![P(x_1) \vee ... \vee P(x_n)]\!] = \max\{[\![P(x)]\!] \mid x \in X\}$$

If the set X were to be infinite, one would have to substitute the minimum and the maximum with, respectively, the infimum and the supremum:

$$[\![\forall x \in X : P(x)]\!] = \inf\{[\![P(x)]\!] \mid x \in X\} \qquad [\![\exists x \in X : P(x)]\!] = \sup\{[\![P(x)]\!] \mid x \in X\}$$

Choosing min as a t-norm to extend the universal quantifier and max as a t-conorm to extend the existential quantifier is a standard choice. Even though it would be valid, extend the quantifiers using norms that aren't min and max respectively is hardly ever done.

2.3.3. Functions with one argument

Given a classical set $M \subseteq X$ and a function $f : X \mapsto Y$, the image $f[M]$ is the subset of Y containing the images of all the elements of M to whom f is applied. That is:

$$f[M] = \{y \in Y \mid \exists x \in X : x \in M \wedge f(x) = y\} \quad \text{that is} \quad y \in f[M] \iff (\exists x \in X)(x \in M \wedge f(x) = y)$$

Consider a fuzzy set μ and a function f. The previous equation can be rephrased as:

$$[\![y \in f[\mu]]\!] = [\![\exists x \in X : x \in \mu \wedge f(x) = y]\!]$$

Which, with respect to the way the existential quantifier was extended, 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 $P(x)$ and t is an appropriately-chosen t-norm to implement the conjunction between $x \in \mu$ and $f(x) = y$.

Note, however, how the choice of t is completely irrelevant. This is due to the fact that the expression $f(x) = y$ is not fuzzy, since y either is or is not the image of x under f. Therefore, $[\![f(x) = y]\!] \in \{0, 1\}$, which in turn implies that the t-norm t will always be either $t(\mu(x), 0)$ or $t(\mu(x), 1)$. This means that it's possible to apply the boundedness property, giving:

$$t(\mu(x), [\![f(x) = y]\!]) = \begin{cases} \mu(x) & \text{if } [\![f(x) = y]\!] = 1 \\ 0 & \text{otherwise} \end{cases} = \begin{cases} \mu(x) & \text{if } f(x) = y \\ 0 & \text{otherwise} \end{cases}$$

Therefore, $f[\mu](y)$ can be reduced to:

$$f[\mu](y) = \sup\{\mu(x) \mid f(x) = y\}$$

In simpler terms, this just means that the degree of membership of an element $y \in Y$ to the image of the fuzzy set $\mu \in \mathcal{F}(X)$ is the highest degree of membership to X that can be found among the elements of x having y as image through f. This extension of a mapping to fuzzy sets is called **extension principle** (for single-valued functions).

2.3.4. Cartesian product, projection, cylindrical extension

Let M_i with $i = 1, ..., n$ be a family of n classical sets. The cartesian product of said sets is given by the set:

$$M_1 \times M_2 \times ... \times M_n = \{(x_1, x_2, ..., x_n) \mid x_1 \in M_1, x_2 \in M_2, ..., x_n \in M_n\}$$

That is, the set of all possible ordered tuples having as each i-th element an element of the i-th set. Stated otherwise, a tuple $(x_1, ..., x_n)$ is a member of $M_1 \times ... \times M_n$ if and only if each i-th element of the tuple is a member to the i-th member of the product. That is:

$$(x_1, x_2, ..., x_n) \in M_1 \times M_2 \times ... \times M_n \iff x_1 \in M_1 \wedge x_2 \in M_2 \wedge ... \wedge x_n \in M_n$$

Given a family of n fuzzy sets μ_i with $i = 1, ..., n$, this is equivalent to:

$$[\![(x_1, x_2, ..., x_n) \in \mu_1 \times \mu_2 \times ... \times \mu_n]\!] = [\![x_1 \in \mu_1 \wedge x_2 \in \mu_2 \wedge ... \wedge x_n \in \mu_n]\!]$$

Which means that the Cartesian product $\mu_1 \times ... \times \mu_n \in \mathcal{F}(X_1 \times ... \times X_n)$ can be extended as:

$$\begin{aligned}(\mu_1 \times ... \times \mu_n)(x_1, ..., x_n) = [\![x_1 \in \mu_1 \wedge ... \wedge x_n \in \mu_n]\!] = \min\{[\![x_1 \in \mu_1]\!], ..., [\![x_n \in \mu_n]\!]\} = \\ = \min\{\mu_1(x_1), ..., \mu_n(x_n)\}\end{aligned}$$

Consider a Cartesian product $X = X_1 \times ... \times X_n$ with $i \in \{1, ..., n\}$. The mapping:

$$\pi_i : X = X_1 \times ... \times X_n \mapsto X_i,\ \pi_i(x_1, ..., x_n) = x_i$$

That has as input an element of a Cartesian product and returns as output an element of one of the sets that constitutes it is the **projection** of $X_1 \times ... \times X_n$ onto X_i. Applying the extension principle to π_i gives:

$$\pi_i[\mu](x) = \sup\{\mu(x_1, ..., x_{i-1}, x, x_{i+1}, ..., x_n) \mid x_1 \in X_1, ..., x_{i-1} \in X_{i-1}, x_{i+1} \in X_{i+1}, ..., x_n \in X_n\}$$

A special case of a Cartesian product is the **cylindrical extension** of a fuzzy set. Given a fuzzy set $\mu \in \mathcal{F}(X_i)$ and a Cartesian product $X_1 \times ... \times X_n$, the cylindrical extension of μ is the Cartesian product between μ and the characteristic functions of $X_1, ..., X_{i-1}, X_{i+1}, ..., X_n$:

$$\hat{\pi}_i(\mu) = I_{X_1} \times ... \times I_{X_{i-1}} \times \mu \times I_{X_{i+1}} \times ... \times I_{X_n},\ \hat{\pi}_i(\mu)(x_1, ..., x_n) = \mu(x_i)$$

As long as the sets $X_1, ..., X_n$ are nonempty, projecting a cylindrical extension results in the original fuzzy set: $\pi_i[\hat{\pi}_i(\mu)] = \mu$. If the fuzzy sets $\mu_1, ..., \mu_{i-1}, \mu_{i+1}, ..., \mu_n$ are normal, $\pi_i[\mu_1 \times ... \times \mu_n] = \mu_i$ holds.

2.3.5. 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:

$$\begin{aligned} f[\mu_1, ..., \mu_n](y) &= \sup_{(x_1,...,x_n) \in X_1 \times ... \times X_n} \{(\mu_1 \times ... \times \mu_n)(x_1, ..., x_n) \mid f(x_1, ..., x_n) = y\} = \\ &= \sup_{(x_1,...,x_n) \in X_1 \times ... \times X_n} \{\min\{\mu_1(x_1), ..., \mu_n(x_n)\} \mid f(x_1, ..., x_n) = y\} \end{aligned}$$

Which is the most general form of the extension principle.

Exercise 2.3.5.1: How should addition between two fuzzy sets $\mu_1 \in \mathcal{F}(X_1)$, $\mu_2 \in \mathcal{F}(X_2)$ be defined?

Solution: Addition between two fuzzy sets can be thought of as a function $f : \mu_1 \times \mu_2 \mapsto \mu_1 \oplus \mu_2$. In particular, applying the definition:

$$f[\mu_1, \mu_2](y) = \sup_{(x_1,x_2) \in X_1 \times X_2} \{\min\{\mu_1(x_1), \mu_2(x_2)\} \mid x_1 + x_2 = y\}$$

□

2.4. Linguistic variables

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.

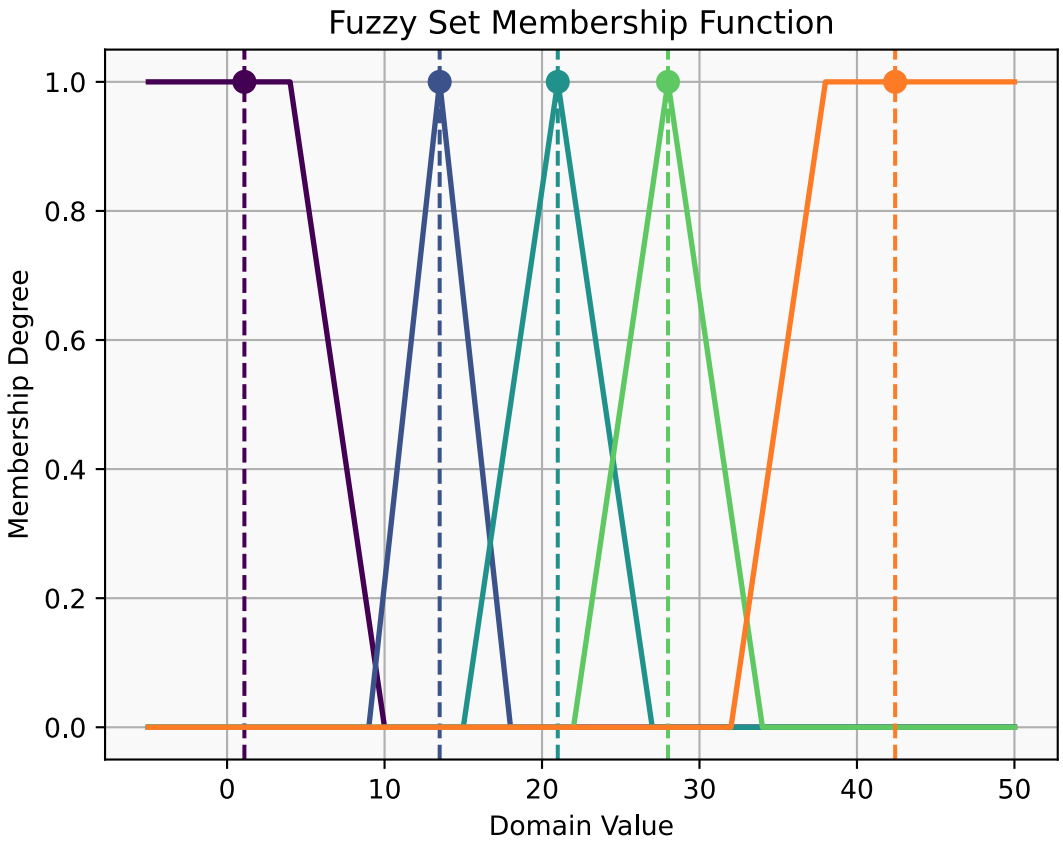
The concept of fuzzy number plays fundamental role in formulating **quantitative fuzzy variables**: those are (mathematical) variables whose possible states are fuzzy numbers. In particular, fuzzy variables that represent linguistic concepts (*small*, *tall*, *hot*, ecc...) are also referred to as **linguistic 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 2.4.1: 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:



□

2.5. Fuzzy reasoning

A (binary) **relation** over the universe sets X and Y is any subset R of the Cartesian product between X and Y. The pairs $(x, y) \in X \times Y$ belonging to the relation R are linked by a semantic connection specified by R.

Relations are a more general form of functions: 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 functions, a relation can be applied to an entire set. If $R \subseteq X \times Y$ is a relation between X and Y and $M \subseteq X$ is a subset of X, the image of M under R is the set:

$$R[M] = \{y \in Y \mid \exists x \in X : (x, y) \in R \wedge x \in M\} \quad \text{that is} \quad y \in R[M] \iff \exists x \in X : (x, y) \in R \wedge x \in M$$

That is, $R[M]$ contains those elements from Y that appear in R paired with an element of M at least once.

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 $\rho(x, y)$.

The extention of the image of a relation to fuzzy sets follows from the definition:

$$\begin{aligned}\rho[\mu](y) = [\![y \in \rho[\mu]]\!] = [\![\exists x \in X : (x,y) \in R \wedge x \in M]\!] = \\ = \sup\{[\![(x,y) \in R \wedge x \in M]\!] \mid x \in X\} = \\ = \sup\{\min\{(x,y) \in R, x \in M\} \mid x \in X\}\end{aligned}$$

The real strength of relations is the fact they can model logical inferences. This allows one to extend logical deductions to fuzzy logic, and being able to reason even in the face of partial truth. Consider a logical deduction based on an implication of the form $x \in A \to y \in B$, with $A \subseteq X$ and $B \subseteq Y$ classical sets. The statement "if x belongs to A then y belongs to B" can be encoded into a relation in the following way:

$$\begin{aligned}R(x,y) = \{(x,y) \in X \times Y \mid x \in A \to y \in B\} = (A \times B) \cup \left(\overline{A} \times \overline{B}\right) \cup \left(\overline{A} \times B\right) = \\ = (A \times B) \cup \left(\overline{A} \cup \overline{A} \times \overline{B} \cup B\right) = (A \times B) \cup \left(\overline{A} \times Y\right)\end{aligned}$$

Since an implication is always true except when the left hand side is true and the right hand side is false. Extending this relation to fuzzy sets using the Gödel implication:

$$\rho(x,y) = [\![(x,y) \in \rho]\!] = [\![x \in \mu \to y \in \nu]\!] = \begin{cases}1 & \text{if } [\![x \in \mu]\!] \leq [\![y \in \nu]\!] \\ [\![y \in \nu]\!] & \text{otherwise}\end{cases} = \begin{cases}1 & \text{if } \mu(x) \leq \nu(y) \\ \nu(y) & \text{otherwise}\end{cases}$$

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:

$$R_2 \circ R_1 = \{(x,z) \in X \times Z \mid \exists y \in Y : (x,y) \in R_1 \wedge (y,z) \in R_2\}$$

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:

$$\begin{aligned}(\rho_2 \circ \rho_1)(x,z) = [\![(x,z) \in (\rho_2 \circ \rho_1)]\!] = [\![\exists y \in Y : (x,y) \in \rho_1 \wedge (y,z) \in \rho_2]\!] = \\ = \sup\{[\![(x,y) \in \rho_1 \wedge (y,z) \in \rho_2]\!] \mid y \in Y\} = \\ = \sup\{\min\{[\![(x,y) \in \rho_1]\!], [\![(y,z) \in \rho_2]\!]\} \mid y \in Y\} = \\ = \sup\{\min\{\rho_1(x,y), \rho_2(y,z)\} \mid y \in Y\}\end{aligned}$$

3. Evolutionary computing

3.1. Optimization problems

3.1.1. 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 3.1.1.1: 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[5] for x, y, z gives the exact solution $x = y = z = \sqrt{S/6}$. □

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

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

- **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.

3.1.2. 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.

3.1.3. 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).

3.2. 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.

3.2.1. 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 $\mathrm{x} = (x_1, ..., x_n) \leftarrow$ random initial solution
2 **do**
3 $\quad \nabla = (\nabla_1, ..., \nabla_n) \leftarrow \left(\frac{\partial}{\partial x_1}(f(x)), ..., \frac{\partial}{\partial x_n}(f(x))\right)$ // Compute gradient
4 $\quad \mathrm{x} \leftarrow x \pm \eta\nabla$ // 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

3.2.2. 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($f : \mathbb{R}^n \to \mathbb{R}, \varepsilon$):
1 $x = (x_1, ..., x_n) \leftarrow$ random initial solution
2 **do**
3 $\quad x' = (x'_1, ..., x'_n) \leftarrow$ random point near x
4 $\quad$ **if** ($f(x') > f(x)$)
5 $\quad\quad x \leftarrow x'$ // Update if there is improvement
6 **while** (**not**(ε))
7 **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).

3.2.3. 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):
1  x = (x_1, ..., x_n) ← random initial solution
2  Δ_max ← 0                                  // Variability across solutions
3  do
4      x' = (x'_1, ..., x'_n) ← random point near x
5      Δ ← f(x') − f(x)                       // Improvement size
6      if (|Δ| > Δ_max)
7          Δ_max ← |Δ|
8      p ← e^{Δ/Δ_max T}                      // Tolerance to worse solutions
9      u ← a value sampled from U ∼ (0, 1)
10     if (Δ > 0 ∨ p ≥ u)                     // New solution is better or tolerated
11         x ← x'
12     T ← T − δ(T)                           // Decrease temperature
13 while (not(ε))
14 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.

3.2.4. 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 $(\textbf{not}(\varepsilon))$
return x // Return best solution found

3.2.5. 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 $(\textbf{not}(\varepsilon))$
return x // Return best solution found

3.2.6. 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
```

3.3. Evolutionary algorithms

A particular framework for solving optimization problems that falls into the guided search category is **Metaheuristics**. An example of metaheuristics is **evolutionary computing**: solving optimization problems constructing algorithms, called **evolutionary algorithms**, that draw inspiration from nature-driven and biological processes, in particular the Theory of Evolution. The Theory explains why the living beings on Earth look as they do and where do they come from.

The major underpinning of the Theory of Evolution 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, but also often happens during reproduction, where the offspring inherits half set of chromosomes from each parent, therefore creating a new unique combination of traits, and 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, however, that such modified traits are not beneficial or harmful in themselves, but only with respect to the environment in which species live. A desirable trait in one environment might turn out to be a burden in a different one.

Biological evolution is an incredibly slow process: each variation is immediately put to the test with respect to an environment and only the beneficial variations are kept and extended. However, the vast majority of random (genetic) modifications are harmful for the individual, either limiting its capabilities or even making it unfit to live, and these get lost in the generations. Only a very slim portion of the changes are actually beneficial; small improvements can accumulate over many generations, leading to surprising complexity and strikingly fitting adaptations (to a specific environment).

Variation (mutation and recombination) and selection are the core principles of biological evolution, but an in-depth analysis reveal many more nuances. A more detailed list of principles of evolution, useful to be taken into account when drawing inspiration for evolutionary algorithms, 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 actually existing life forms are only a tiny fraction of the theoretically possible ones;
- **Variation**: Mutation and genetic recombination continuously create new variants, that may result in a new combination of already existing traits or may introduce a modified, never seen trait altogether;
- **Inheritance**: Genetic variations are passed onto the offspring, whereas physical variations are not;
- **Speciation**: A new species if formed when two or more population subgroups coming from the same species 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;
- **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. That is, variations are not preprogrammed to "push" evolution in one direction;
- **Gradualism**: Variations happen in small steps, thus phylogenetic changes are gradual and relatively slow;
- **Evolution/Transmutation/Inheritance with Modification**: Due to the adaptation to the environment, species are not immutable, evolving instead in the course of time;
- **Discrete Genetic Units**: The genetic information is stored in discrete units, the genes, not in a continuous fashion;
- **Opportunism**: The evolution process builds upon the living beings as they are in the present, does not create variations out of anything, only out of what the species possess;
- **Evolution-strategic Principles**: Not only organisms are optimized for their environment, but also the *mechanisms* of evolution itself, such as reproduction rates, mortality rates, life spans, evolutionary speed, etc...;
- **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 all others;
- **Irreversibility**: The course of evolution is irreversible, that is, a species cannot go "evolve backwards";
- **Unpredictability**: The course of evolution 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 problem of having a species adapt to an environment can be conceived as an optimization problem: "tuning" the characteristics of a species in order to "optimize" them for a specific environment, finding a solution that, even though not the best, is certainly satisfactory. The same approach is what evolutionary computing seeks to apply to the solution of numerical optimization problems.

Evolutionary computing, and metaheuristics in general, falls into the last category. To make good use of evolutionary computing technique, it is important to state how biological terms are translated into computer science.

An **individual**, which is a living organism in biology, corresponds to a candidate solution in computer science. Individuals are the entities to which a fitness is assigned and which are subject to the (natural) selection process.

A **chromosome** is, in biology, a string of DNA enveloped in proteins, that stores the genetic information of an individual, its "blueprint", that encodes its traits. In computer science, its counterpart would be

information stored in bits. Note that most living organisms have several chromosomes, among whose information is (unequally) distributed; in computer science, there is no need to model this aspect, and all genetic information can be combined 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 possible ways in which a gene can exist are called **alleles**; each individual has exactly one allele (that is, one mode of existing) for each gene. The location of a gene in a DNA strand, called **locus** is (pretty much) fixed, meaning that it's (almost) possible to refer to a genetic trait with respect to its position on the strand. In computer science, an allele is simply the value of a computational object, which selects one of several possible properties of a solution candidate that the gene stands for.

In biology, the **genotype** is the genetic configuration of an organism, which alleles are present for each of its genes (or, at least, the ones of interest), whereas the **phenotype** is the physical appearance of an organism, the way the genotype manifests itself. Note that the phenotype is what interacts with the environment, hence it's the phenotype, and not the genotype, that actually determines the fitness of the individual, even though the genotype is still a latent influence, since it determines the phenotype. In computer science, the genotype corresponds to the encoding of a candidate solution, 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 simple set of individuals, usually of the same species; a **generation** is the population at a certain point in time. In biology, no two individuals from the same population can be an exact genetic copy of one another (not even homozygous twins), since the number of possible combinations of genes is too big for this to happen. In computer science, however, the number of genes is limited to a small subset of interest, therefore identical individuals can (co)exist. As a consequence, a population of an evolutionary algorithm is a *multiset* (a set where the elements can appear more than once) of individuals.

A new generation is created by **reproduction**, that is, by the generation of offspring from one or more (usually two) organisms, in which genetic material of the parent individuals may be recombined. The same holds for computer science, only that the child creation process works directly on the chromosomes and that the number of parents may exceed two.

The **fitness** of an individual measures how high its chances of survival and reproduction are due to its adaptation to its environment. The quality of a biological organism with respect to its environment is difficult to assess objectively. Simply defining fitness as "the ability of an individual to survive" would just move the goalpost: a formally more precise and quantifiable definition of fitness would be the number of (average) fertile offspring of an individual. In computer science, the fitness is much easier to quantify, since the optimization problem provides a fitness function with which solution candidates are to be evaluated.

The general idea of an evolutionary algorithm it to employ evolution principles to generate increasingly better solution candidates for the optimization problem at hand. Essentially, this entails evolving a population of solution candidates, selecting the most promising on each generation on the basis of their adaptation to the environment. An evolutionary algorithm requires:

- An encoding for the solution candidates;
- A method to create an initial population;
- A fitness function to evaluate the individuals;
- A selection method on the basis of the fitness function;
- A set of genetic operators to modify chromosomes;
- A termination criterion for the search;
- Values for various parameters.

Since the intent is to evolve a population of solution candidates, it is necessary to find a way of representing them as chromosomes. That is, it is necessary to encode them, as sequences of computational objects. Such an encoding may be so direct that the distinction between genotype and phenotype becomes blurred, or non existent. In other cases there is a clear distinction between the solution candidate and its encoding.

In general, the encoding of the solution candidates is highly dependent on the problem at hand, and there is no one-size fits-all method for doing so. However, it is important to specify that a wrong encoding might result in an unusable evolutionary algorithm, therefore the choice of the encoding must be taken with care.

Once an encoding is chosen, one can create an initial population of solution candidates in the form of chromosomes representing them. Since chromosomes are simple sequences of computational objects, an initial population is commonly created by simply generating random sequences. In some cases, especially if the solution candidates have to satisfy certain constraints, a more refined approach might be needed.

In order to mimic the influence of the environment in biological evolution, one needs a fitness function with which one can evaluate the individuals of the created population. In many cases this fitness function is just the function of optimization problem. However, the fitness function may also contain additional elements that represent constraints that have to be satisfied in order for a solution candidate to be acceptable or that introduce a tendency toward certain additionally desired properties of a solution.

The natural selection process of biological evolution is simulated by a method to select candidate solutions according to their fitness. This method is used to choose the parents of offspring we want to create or to select those individuals that are transferred to the next generation without change. Such a selection method may simply transform the fitness values into a selection probability, such that better individuals have higher chances of getting chosen for the next generation.

The random variation of chromosomes is simulated by so-called genetic operators that modify and recombine chromosomes, for example, mutation, which randomly changes individual genes, and crossing over, which exchanges parts of the chromosomes of parent individuals to produce offspring. Depending on the problem and the chosen encoding, the genetic operators can be very generic or highly problem-specific. The choice of the genetic operators is another element that effort should be spent on, especially in connection with the chosen encoding.

Even though real-world evolution never actually stops, the last needed element for an evolutionary algorithm is a stopping criteria to extract an optimal (final) solution. Such a criterion might be, for example: stop after a given number of iterations, stop after the improvement from one generation to the next is negligible, stop when a user-specified minimum solution quality has been obtained.

To complete the specification of an evolutionary algorithm, one has to choose the values of several parameters, such as: the size of the population to evolve, the fraction of individuals that is chosen from each population to produce offspring, the probability of a mutation occurring in an individual, ecc...

A generic evolutionary algorithm can be written as such:

```
GENERIC-EVOLUTIONARY-ALGORITHM(ε):
1  t ← 0
2  Initialize (population(t))                    // Create the initial population
3  Evaluate (population(t))                      // Compute fitness
4  while (not(ε))
5  |   t ← t + 1
6  |   population(t) ← Select-from
   |   (population"(t − 1))                      // Select individuals based on fitness
7  |   Alter (population(t))                     // Apply genetic operators
8  |   Evaluate (population(t))                  // Evaluate the new population
```

That is, after having created and evaluated an initial population of solution candidates (in the form of chromosomes), a sequence of generations of solution candidates is computed. Each new generation is created by selecting individuals based on their fitness (higher fitness means a higher chance of getting selected). Then genetic operators (like mutation and crossover) are applied to the selected individuals. Next, the modified population (or at least the new individuals in it, which have been created by the genetic operators) is evaluated and the cycle starts over. This process continues until the chosen termination criterion is fulfilled.

In classical mathematical optimization, many techniques and algorithms have been developed that are fairly closely related to evolutionary computing. Such methods are sometimes called **local search methods**, because they explore the search space in small steps, carrying out a local search for better solutions.

Like evolutionary algorithms, these techniques are based on the assumption that similar solution candidates also yield similar values of the function to optimize. The main difference to evolutionary algorithms is that local search methods inspect one solution at a time, instead of an entire population (in some sense, they can be thought of as evolutionary algorithms with population size equal to 1). They are often employed to improve solutions candidates locally or as a final optimization step for the output of an evolutionary algorithm.

3.4. 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 $k = \lceil \log_2\left(\frac{b-a}{\varepsilon}\right) \rceil$, 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 3.4.1: 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 3.4.2: What would be the Gray encodings of the two numbers in Exercise 3.4.1?

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 3.4.3: 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 3.4.4: 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.

3.5. 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**:

$$f_\sigma(s) = f(s) - \mu_f(t) + \beta\sigma_f(t) \qquad \text{with } \mu_f(t) = \frac{\sum_{s\in P(t)} f(s)}{|P|} \text{ and } \sigma_f(t) = \sqrt{\frac{\sum_{s\in P(t)} \left(f(s) - \mu_f(t)\right)^2}{|P| - 1}}$$

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.

3.6. 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      u ← a value sampled from U ~ (0, 1)
3      if (u ≤ p_m)
4          S_i ← 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 ← empty array
2  for i = 1 to |s|
3      X_i = i
4  X ← the first n elements of X
5  X ← random permutation of X
6  foreach x_i in X
7      S_{x_i} ← 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):

for $i = 1$ **to** $|S|$
  $\nu \leftarrow$ a value sampled from $N \sim (0, \sigma)$
  $S_i \leftarrow S_i + \nu$
  $S_i \leftarrow \max\{S_i, l_i\}$
  $S_i \leftarrow \min\{S_i, h_i\}$

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):

$u \leftarrow$ a value sampled from $U \sim (0, 1)$
$\sigma_S \leftarrow \sigma_S \cdot \exp\left(u/\sqrt{|S|}\right)$
for $i = 1$ **to** $|S|$
  $\nu \leftarrow$ a value sampled from $N \sim (0, \sigma_S)$
  $S_i \leftarrow S_i + \nu$
  $S_i \leftarrow \max\{S_i, l_i\}$
  $S_i \leftarrow \min\{S_i, h_i\}$

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):
c ← a random value in {1, 2, ..., |S_1| − 1}
for i = 0 to c − 1
    t ← S_{1,i}
    S_{1,i} ← S_{2,i}
    S_{2,i} ← 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 3.6.1: 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.

3.7. Improving performance through parallelization

3.7.1. 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.

3.7.2. 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.

3.7.3. 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.

3.8. 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...)

3.8.1. Tabu search

Tabu search[6] 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

[6] The name "tabu search" comes from the word "taboo", meaning "forbidden".

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
```

3.8.2. 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.

3.8.3. Differential evolution

Differential evolution tries to exploit the relationships that intercur between solutions.

3.8.4. 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.

3.8.5. Cultural algorithm

3.9. 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.

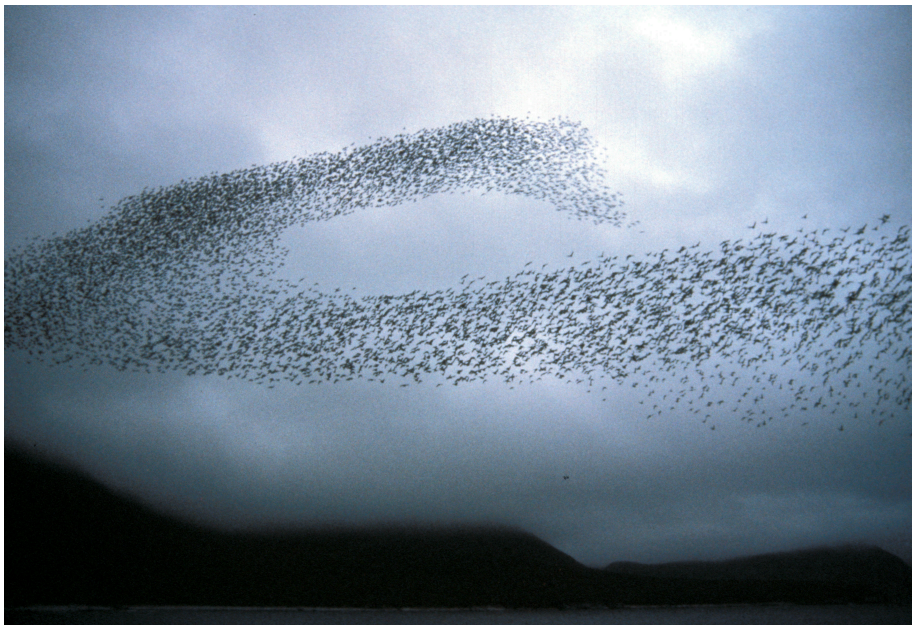


Figure 6: The complex patterns and unison movements of flock of birds is an example of swarm intelligence.

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**.

3.9.1. 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.

3.9.2. 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"[7], 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[8]. 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.

[7] Even though, as any chemical marker, pheromones do evaporate over time, this has little influence in the real world behaviour of ants.

[8] 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.

3.10. 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[9]. 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.

[9]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.

GENERIC-GENETIC-ALGORITHM($f : \mathbb{R}^n \to \mathbb{R}, \mu, p_x, p_m, \varepsilon$):

```
1  pop ← a random sequence of μ bit strings
2  for i = 1 to μ
3  |   pop[i].fitness ← f(pop[i])
4  do
5  |   chosen ← select μ individuals from pop with roulette wheel selection
6  |   newpop ← ∅
7  |   for i = 1 to μ/2
8  |   |   u ← a value sampled from U ~ (0, 1)
9  |   |   if (u ≤ p_x)
10 |   |   |   One-Point-Crossover (chosen[2i - 1], chosen[2i])
11 |   |   Bit-Mutation (chosen[2i - 1], p_m)
12 |   |   Bit-Mutation (chosen[2i], p_m)
13 |   |   newpop ← newpop ∪ {chosen[2i - 1], chosen[2i]}
14 |   pop ← newpop
15 |   for i = 1 to μ
16 |   |   pop[i].fitness ← f(pop[i])
17 while(not(ε))
18 best ← pop[0]
19 for i = 1 to μ
20 |   if (pop[i].fitness > best.fitness)
21 |   |   best ← pop[i]
22 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 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 3.10.1: 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:

$$\mu f^{(t)}_{\text{rel}}(h) = \mu\left(\frac{\sum_{s\in P(t), s\lhd h} f^{(t)}_{\text{rel}}(s)}{N(h,t)}\right) = \mu\left(\frac{\sum_{s\in P(t), s\lhd h} \frac{f_{\text{abs}}(s)}{\sum_{s'\in P(t)} f_{\text{abs}}(s')}}{N(h,t)}\right) = \frac{\mu\Big(\sum_{s\in P(t), s\lhd h} f_{\text{abs}}(s)\Big)}{N(h,t)\Big(\sum_{s'\in P(t)} f_{\text{abs}}(s')\Big)} =$$

$$= \frac{\frac{\sum_{s\in P(t), s\lhd h} f_{\text{abs}}(s)}{N(h,t)}}{\frac{\sum_{s'\in P(t)} f_{\text{abs}}(s')}{\mu}} = \frac{\overline{f_t(h)}}{\overline{f_t}}$$

$\overline{f_t(h)}$, 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:

$$N(h, t + \Delta t_s) = N(h,t)\mu f^{(t)}_{\text{rel}}(h) = N(h,t)\frac{\overline{f_t(h)}}{\overline{f_t}}$$

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 3.10.2: 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 3.10.3: 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 3.10.4: 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.

3.11. 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 3.11.1: What would be the set of terms and functions for the formal language of zeroeth-order logic?

Solution:

$$\mathcal{F} = \{\wedge, \vee, \neg, \Rightarrow\} \qquad\qquad \mathcal{T} = \{a, b, c, ..., 0, 1\}$$

□

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 3.11.2: What would be the well-formed formulas for the formal language of zeroeth-order logic?

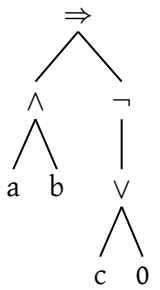
Solution:
- 0 is a well-formed formula;
- 1 is a well-formed formula;
- Any variable ($a, b, c, ...$) is a well-formed formula;
- If X and Y are two well-formed formulas, then $\wedge\,(X, Y)$ is a well-formed formula;
- If X and Y are two well-formed formulas, then $\vee\,(X, Y)$ is a well-formed formula;
- 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 3.11.3: 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 3.11.4: 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, $d_{\max}$):

```
1  P ← ∅                                          // Population starts empty
2  for i = 1 to d_max do
3  |   for j = 1 to 2 · d_max do
4  |   |   P ← P ∪ GP-Initialize-Grow (0, i)
5  |   |   P ← P ∪ GP-Initialize-Full (0, i)
6  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[10]. 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

[10]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 3.11.5: 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.

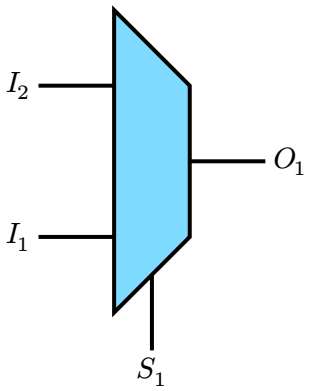
Another operator is **intelligent recombination**, a form of recombination that chooses the crossover points selectively in such a way to prevent, or at least to mitigate, the creation of introns. A third method consists in introducing slight changes in the evaluation function such that what are considered 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.

3.11.1. 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[11].

[11]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.

3.12. 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($f : \mathbb{R}^n \to \mathbb{R}, \mu, \lambda, \mathrm{k}, \theta, \alpha, \varepsilon$):

```
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.

3.13. 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.

3.14. 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**[12], 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 7: 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.

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.

[12]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".

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 8: 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.