Contents

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.

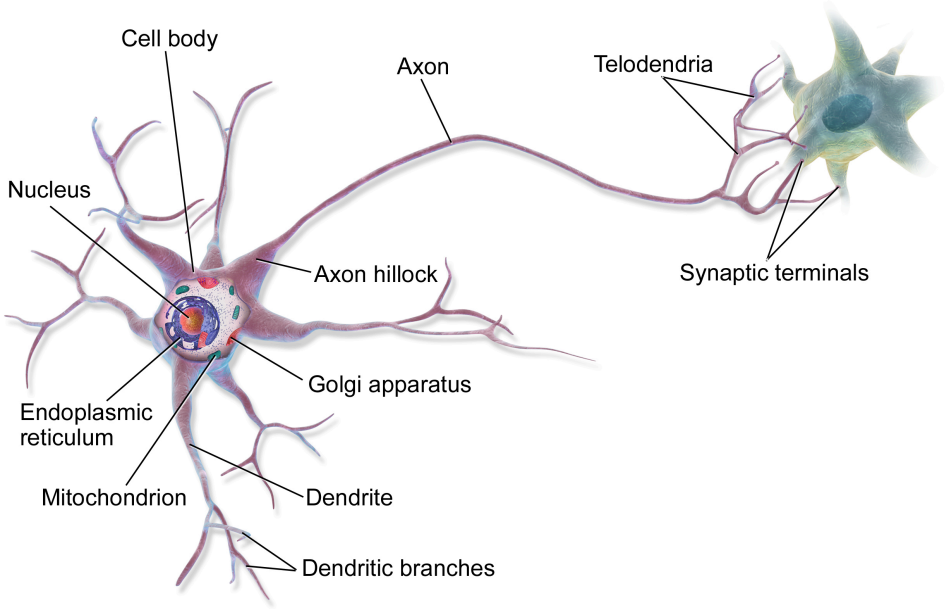


Figure 1: Schematic image of a neuron. By BruceBlaus, CC BY 3.0, via Wikimedia Commons. original image

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.

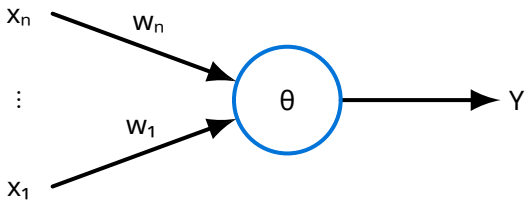
1.2. Threshold Logic Units

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.

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

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.



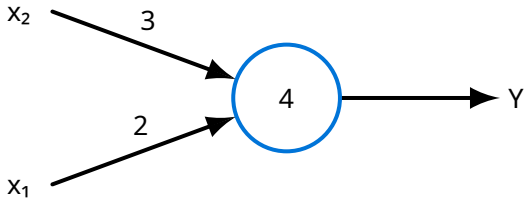
$$y = \begin{cases} 1 \text{ if } \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

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:

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

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.

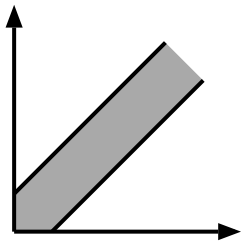
A set of points in the plane is called **convex** if connecting each point of the set with straight lines does not require to go outside of the set. The **convex hull** of a set of points is its the smallest superset that is convex. If two sets of points are both convex and disjoint, they are linearly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

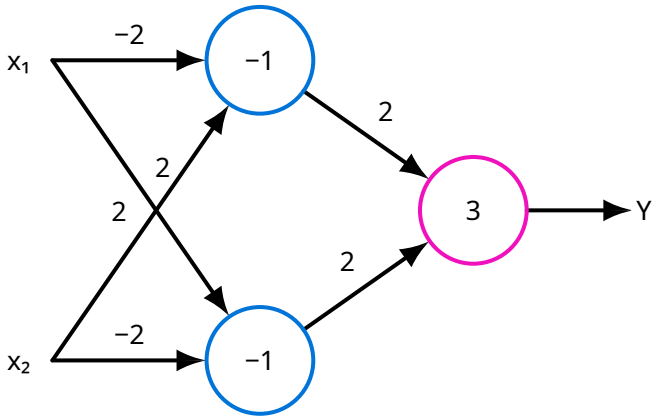


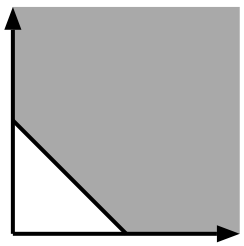
□

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 $(A \to B) \wedge (B \to A)$. 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:

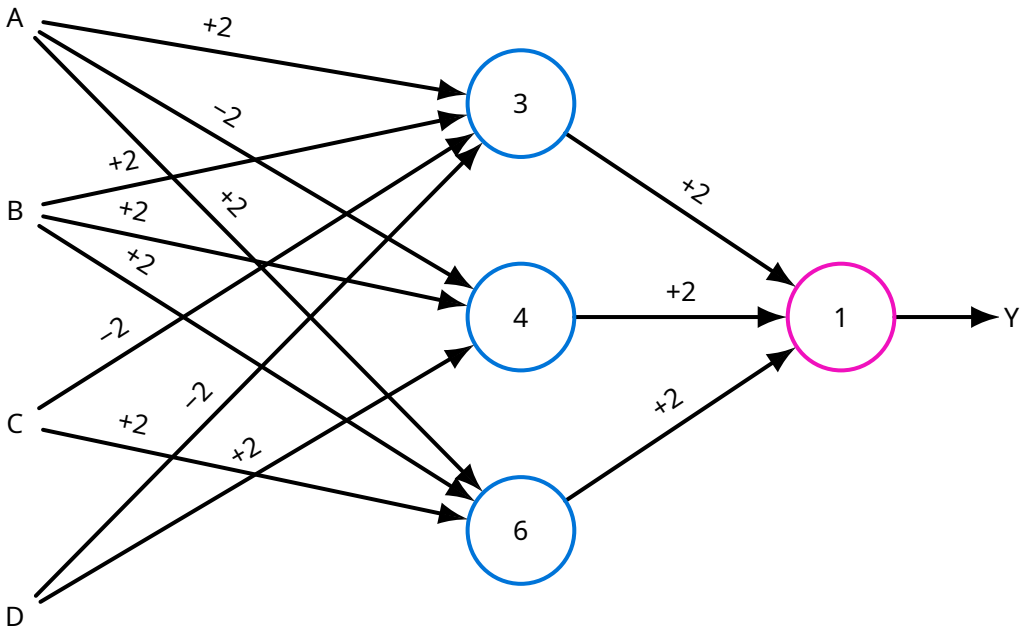
$$w_{j,i} = \begin{cases} +2 \text{ if } l_{j,i} \text{ is a positive literal} \\ -2 \text{ if } l_{j,i} \text{ is a negative literal} \end{cases} \qquad \theta_j = n - 1 + \frac{1}{2} \sum_{i=1}^{n} w_{j,i}$$

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($W = (w_1, ..., w_n)$, $L = ((X_1, y_1), ..., (X_m, y_m))$, θ, η):

```
let e ← ∞                                          // Error
while (e ≠ 0)                                      // Continue until error vanishes
    e ← 0
    for l_i in L do
        let X, y ← l_{i,1}, l_{i,2}                // Unpack
        let ŷ ← 0                                  // Evaluate scalar product
        if (∑_{j=1}^{|X|} X_j · W_j ≥ θ)
            ŷ ← 1
        if (ŷ ≠ y)                                 // Test for output mismatch
            e ← e + |y − ŷ|                        // Update error
            θ ← θ − η · (y − ŷ)                    // Update threshold
            for w_j in W do
                w_j ← w_j + η · (y − ŷ) · X_j      // Update weights
```

TLU-TRAIN-BATCH($W = (w_1, ..., w_n)$, $L = ((X_1, y_1), ..., (X_m, y_m))$, θ, η):

```
let e ← ∞                                          // Error
while (e ≠ 0)                                      // Continue until error vanishes
    e ← 0
    let θ* ← 0                                     // Partial threshold
    let W* ← (0, ..., 0)                           // Partial weights
    for l_i in L do
        let X, y ← l_{i,1}, l_{i,2}                // Unpack
        let ŷ ← 0                                  // Evaluate scalar product
        if (∑_{j=1}^{|X|} X_j · W_j ≥ θ)
            ŷ ← 1
        if (ŷ ≠ y)                                 // Test for output mismatch
            e ← e + |y − ŷ|                        // Update error
            θ* ← θ* − η · (y − ŷ)                  // Partially update threshold
            for w_j in W do
                w*_j ← w*_j + η · (y − ŷ) · X_j    // Partially update weights
        θ ← θ + θ*                                 // Update threshold
        W ← W + W*                                 // Update weights
```

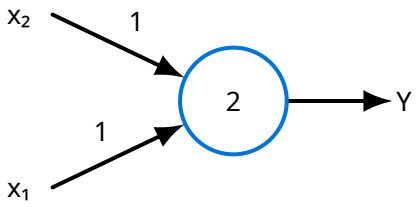
Exercise 1.3.1: Construct a TLU that computes the logical AND between two bits.

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:

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

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.

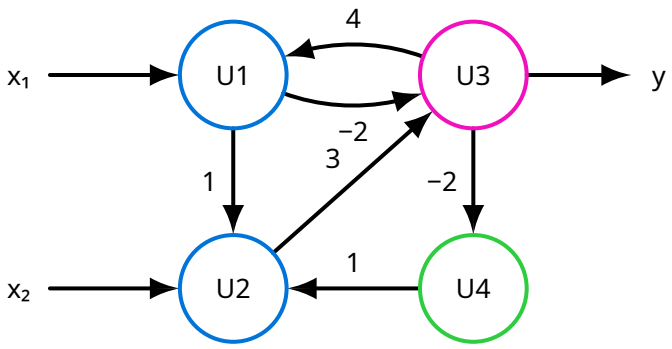
$$U = U_{\text{in}} \cap U_{\text{out}} \cap U_{\text{hidden}} \qquad U_{\text{in}} \neq \emptyset, U_{\text{out}} \neq \emptyset, U_{\text{hidden}} \cap (U_{\text{in}} \cap U_{\text{out}}) = \emptyset$$

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 $u \in U_{\text{in}}$ has a fourth quantity, the **external input** ext_u.

Each neuron $u \in U$ also possesses three functions:

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

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

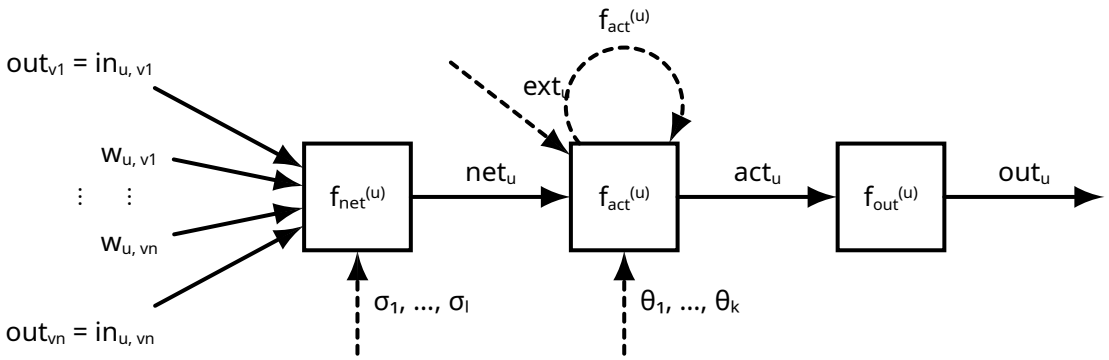


Figure 3: Structure of a neuron

The network input function process the inputs $\text{in}_{u,v_1}, ..., \text{in}_{u,v_n}$ of the neuron u, which are themselves the output $\text{out}_{v_1}, ..., \text{out}_{v_n}$ of other neurons, and the weights $w_{u,v_1}, ..., w_{u,v_n}$, 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:

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

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

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

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:

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

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:

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

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

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

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:

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

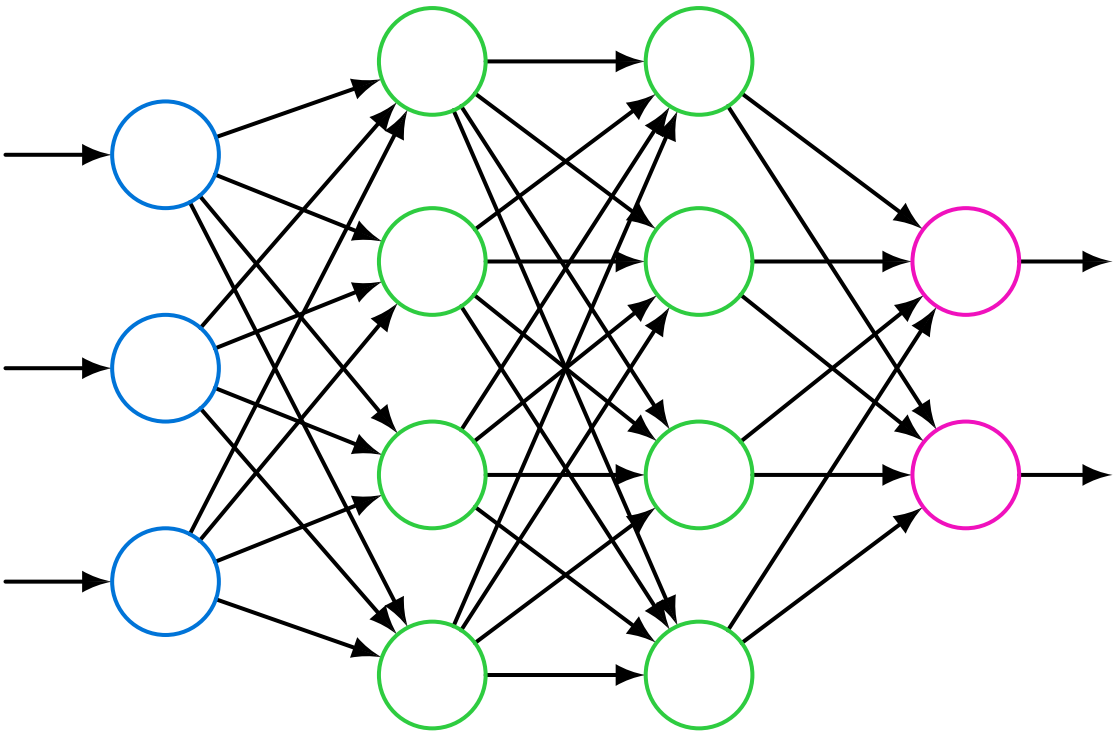


Figure 4: Structure of a generic multilayer perceptron

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:

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

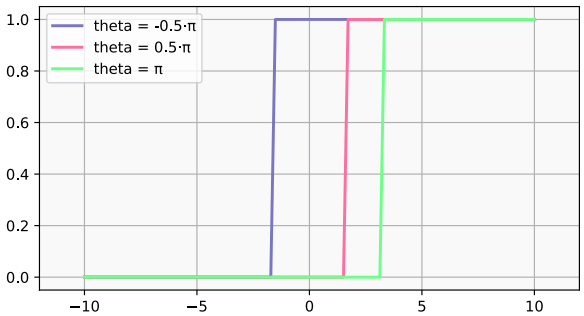
The activation function of hidden neurons is any **sigmoid function**, meaning a monotonic non-decreasing function of the form:

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

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:

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



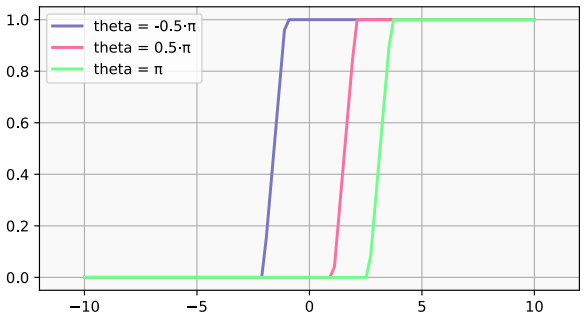
This is because, as it was done for the TLUs, it is possible to move the threshold into the weighted sum and obtain an equivalent function that output 0 if the weighted sum is negative (less than 0) and outputs 1 if positive (greater than 0), and this check can be done by simply looking at the most significant 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:

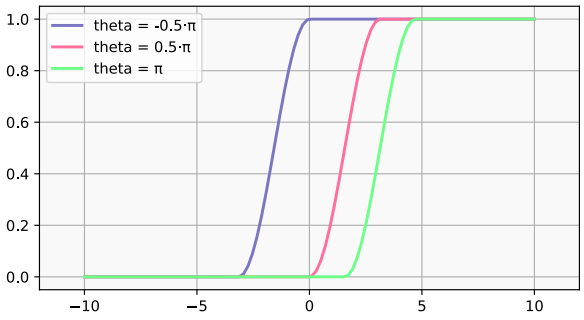
$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1 & \text{if net} > \theta + \frac{1}{2} \\ 0 & \text{if net} < \theta - \frac{1}{2} \\ (\text{net} - \theta) + \frac{1}{2} & \text{otherwise} \end{cases}$$



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:

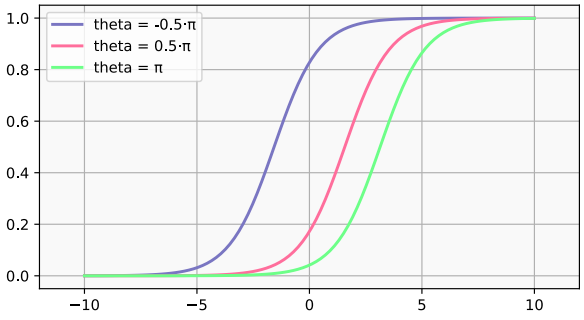
$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1 & \text{if net} > \theta + \frac{\pi}{2} \\ 0 & \text{if net} < \theta - \frac{\pi}{2} \\ \frac{\sin(\text{net} - \theta) + 1}{2} & \text{otherwise} \end{cases}$$



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:

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

[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.

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

$$= \frac{0\cdot\left(1+e^{-(\text{net}\ -\theta)}\right) - \frac{d}{d\ \text{net}}(1) + \frac{d}{d\ \text{net}}\left(e^{-(\text{net}\ -\theta)}\right)}{\left(1+e^{-(\text{net}\ -\theta)}\right)^2} = \frac{\left(e^{-(\text{net}\ -\theta)}\right)\frac{d}{d\ \text{net}}(\text{net}-\theta)}{\left(1+e^{-(\text{net}\ -\theta)}\right)^2} =$$

$$= \frac{e^{-(\text{net}\ -\theta)}}{\left(1+e^{-(\text{net}\ -\theta)}\right)^2} = \frac{1-1+e^{-(\text{net}\ -\theta)}}{\left(1+e^{-(\text{net}\ -\theta)}\right)^2} = \frac{\cancel{1+e^{-(\text{net}\ -\theta)}}}{\left(1+e^{-(\text{net}\ -\theta)}\right)^{\cancel{2}}} - \frac{1}{\left(1+e^{-(\text{net}\ -\theta)}\right)^2} =$$

$$= \frac{1}{1+e^{-(\text{net}\ -\theta)}} - \left(\frac{1}{1+e^{-(\text{net}\ -\theta)}}\right)^2 = f_{\text{act}}(\text{net}, \theta) - \left(f_{\text{act}}(\text{net}, \theta)\right)^2 =$$

$$= f_{\text{act}}(\text{net}, \theta)(1 - f_{\text{act}}(\text{net}, \theta))$$

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

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

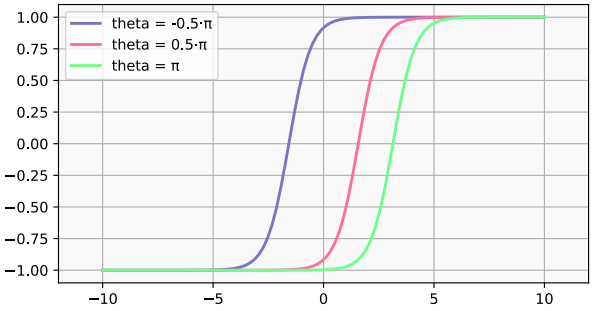
$$e^{-(\text{net}\ -\theta)} f_{\text{act}}(\text{net}, \theta) + f_{\text{act}}(\text{net}, \theta) = 1 \Rightarrow e^{-(\text{net}\ -\theta)} f_{\text{act}}(\text{net}, \theta) = 1 - f_{\text{act}}(\text{net}, \theta) \Rightarrow$$

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

$$\theta - \text{net} = \ln(1 - f_{\text{act}}(\text{net}, \theta)) - \ln(f_{\text{act}}(\text{net}, \theta)) \Rightarrow \text{net} = \theta - \ln\left(\frac{1 - f_{\text{act}}(\text{net}, \theta)}{f_{\text{act}}(\text{net}, \theta)}\right)$$

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:

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



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

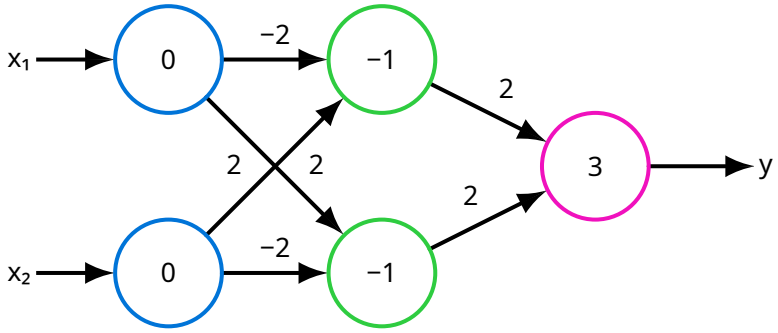
A clear advantage of having a weighted summation as the network input function of a multilayer perceptron is that it translates naturally to matrix multiplication. Let $U_1 = (v_1, ..., v_m)$ and $U_2 = (u_1, ..., u_n)$ be two subsequent layers (U_2 is right after U_1). It is possible to write the network 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 $x_1, x_2, \ldots, x_n$ 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 $x_1 \leq x_2 \leq \ldots \leq x_i \leq \overline{x}$. The neurons of the first layer that will fire are the ones having as threshold $x_1, x_2, \ldots, x_i$.

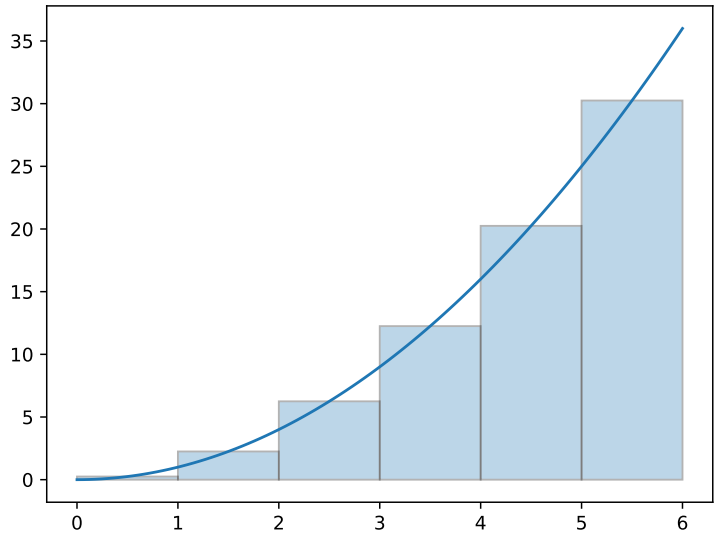
Each pair of adjacent cutoff points induces $n - 1$ intervals $[x_1, x_2], [x_2, x_3], \ldots, [x_{n-1}, x_n]$. Each of these intervals will (more or less accurately) give an approximation for all the values of the true function in said range (the most natural choice of this approximation is the middle 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 $x_1, x_2, \ldots, x_i$. 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.

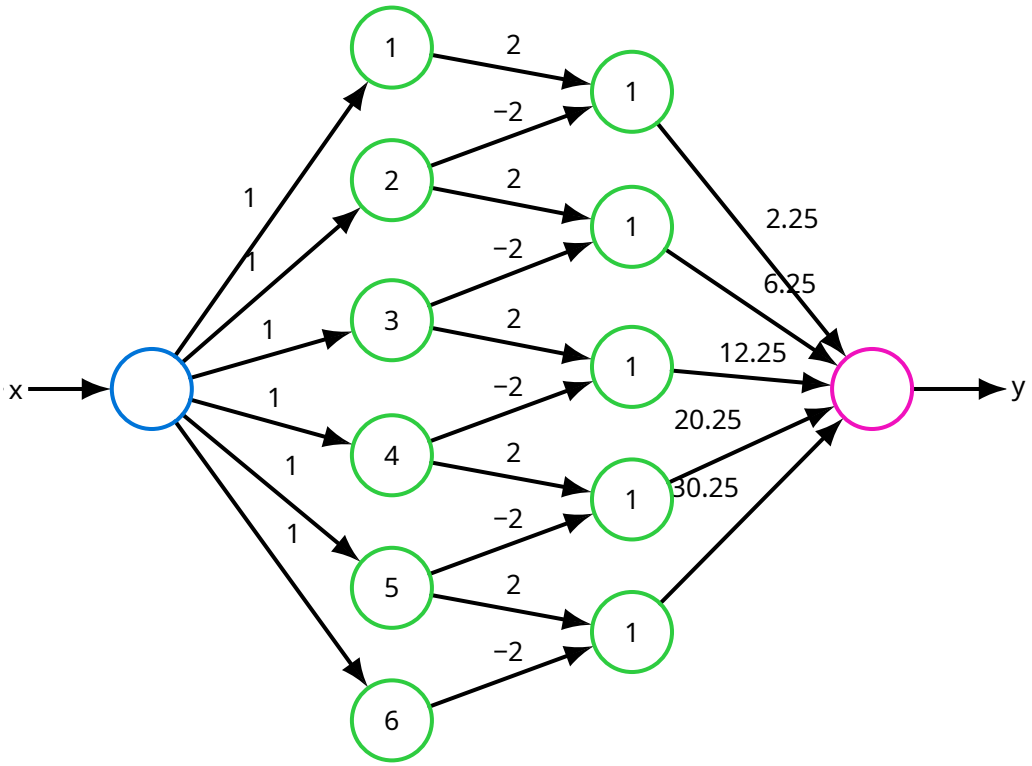
From the output of the network it is possible to know which is the best approximation for a given input, since each of the incoming weights of the input neuron is set to the chosen approximations of the function evaluated at the given input, and only one neuron of the second hidden layer will fire.

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:

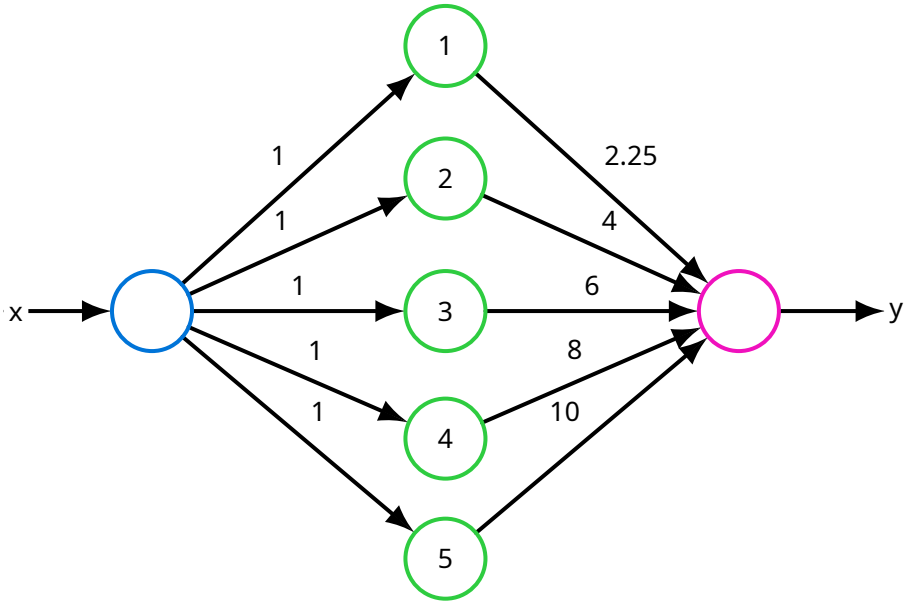
Theorem 1.5.2: Any continuous Riemann-integrable function can be approximated with arbitrary accuracy by a multilayer perceptron of three layers.

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

[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.

(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:

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

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

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

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

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

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

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

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

Setting all partial derivatives equal to 0:

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

Rearranging, one obtains n equations in n unkowns:

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

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:

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

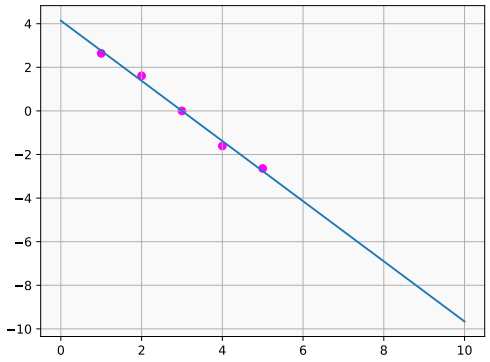
$$\sum_{i=1}^{n} x_i^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 55 \qquad \sum_{i=1}^{n} x_i \tilde{y}_i = 1 \cdot 2.64 + 2 \cdot 1.61 + 3 \cdot 0.00 - 4 \cdot 1.61 - 5 \cdot 2.64 \approx -13.78$$

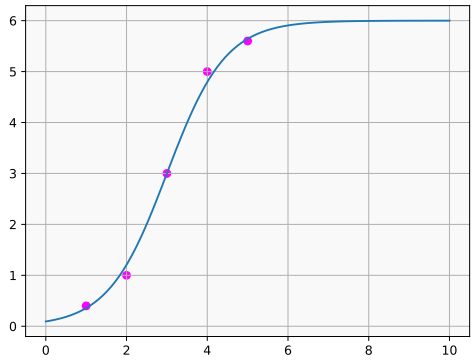
Leads to the following system of equations:

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

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:

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

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 $\text{pred}(u) = \{p_1, \ldots, p_n\} \in U_{k-1}$. The corresponding vector of weights, threshold embedded, is $\boldsymbol{w}_u = \left(-\theta_u, w_{u,p_1}, \ldots, w_{u,p_n}\right)$. The gradient of the total error function with respect to these weights is:

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

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:

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

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:

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

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

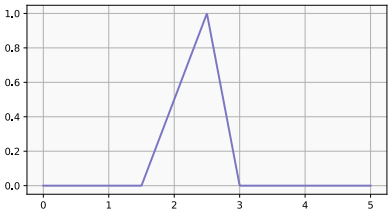
that its degree of membership function is also a convex functions; indeed, such functions are concave functions. Examples of functions with these characteristics are:

- **Triangular functions**

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

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

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

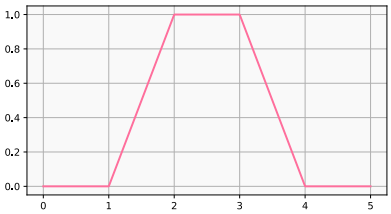


- **Trapezoidal functions**

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

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

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

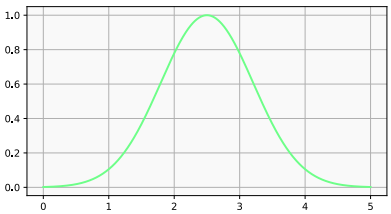


- **Bell-shaped functions**

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

  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:

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

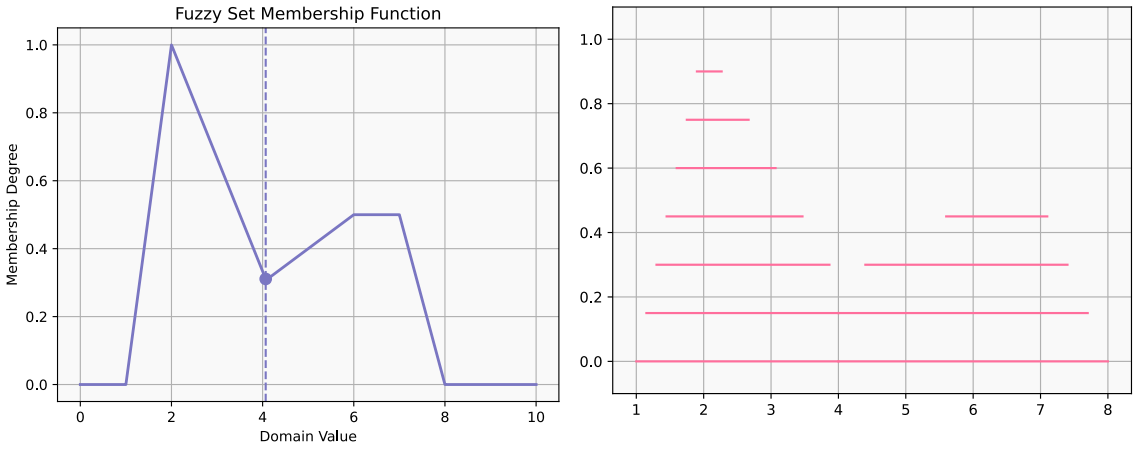
Of course, it would be pointless to store all the α-cuts of a fuzzy set, since it would be no different than storing the entire degree of membership function. However, discretizing the set of all α-cuts into a discrete subset $\left\{[\mu]_{\alpha_1}, [\mu]_{\alpha_2}, ..., [\mu]_{\alpha_k}\right\}$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

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

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).

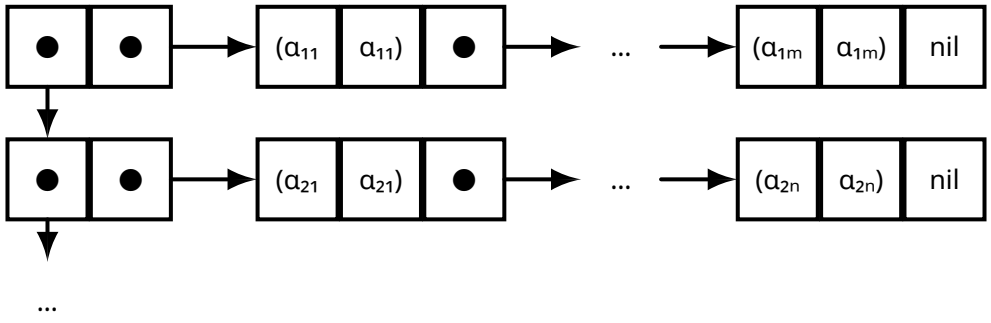


Figure 5: Linked list representation of α-cuts

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.

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

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

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

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 $[0,1]$:

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

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.

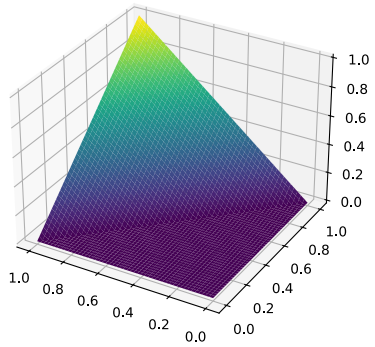
A function $t : [0,1]^2 \mapsto [0,1]$ is said to be a t-**norm**, or **triangular norm**, if it possesses the following properties:

- **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:

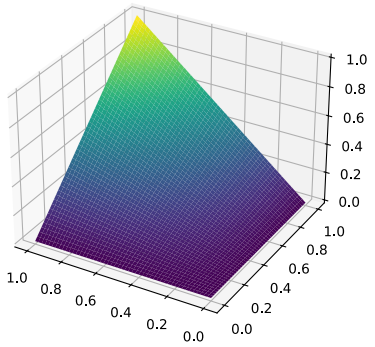
Łukasiewicz t-norm:

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



Algebraic product:

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

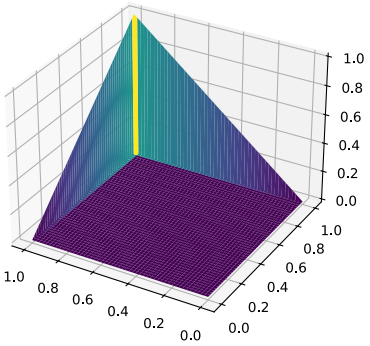


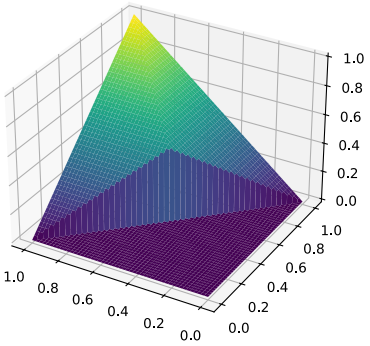
Drastic product:

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

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:

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

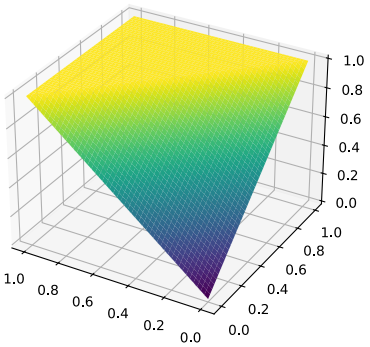
Applying the duality relation to the Łukasiewicz t-norm, the algebraic product and the drastic product one obtains the following conorms:

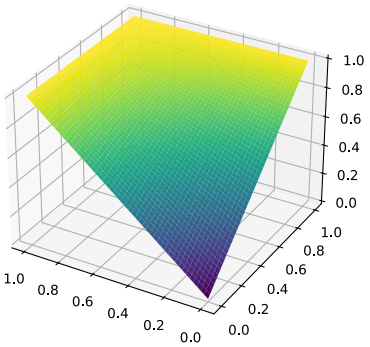
Łukasiewicz t-conorm:

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

Algebraic sum:

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

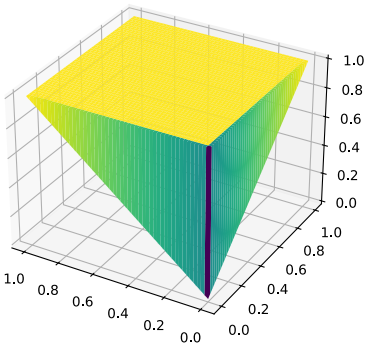


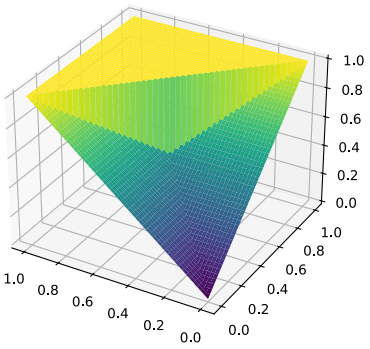


Drastic sum:

Nilpotent maximum:

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





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:

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

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:

$$f[\mu](y) = \sup\{[\![x \in \mu \wedge f(x) = y]\!] \mid x \in X\} = \sup\{t([\![x \in \mu]\!], [\![f(x) = y]\!]) \mid x \in X\} = \\ = \sup\{t(\mu(x), [\![f(x) = y]\!]) \mid x \in X\}$$

Where $x \in \mu \wedge f(x) = y$ plays the role of the proposition $P(x)$ and t is an approprietely-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:

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

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.

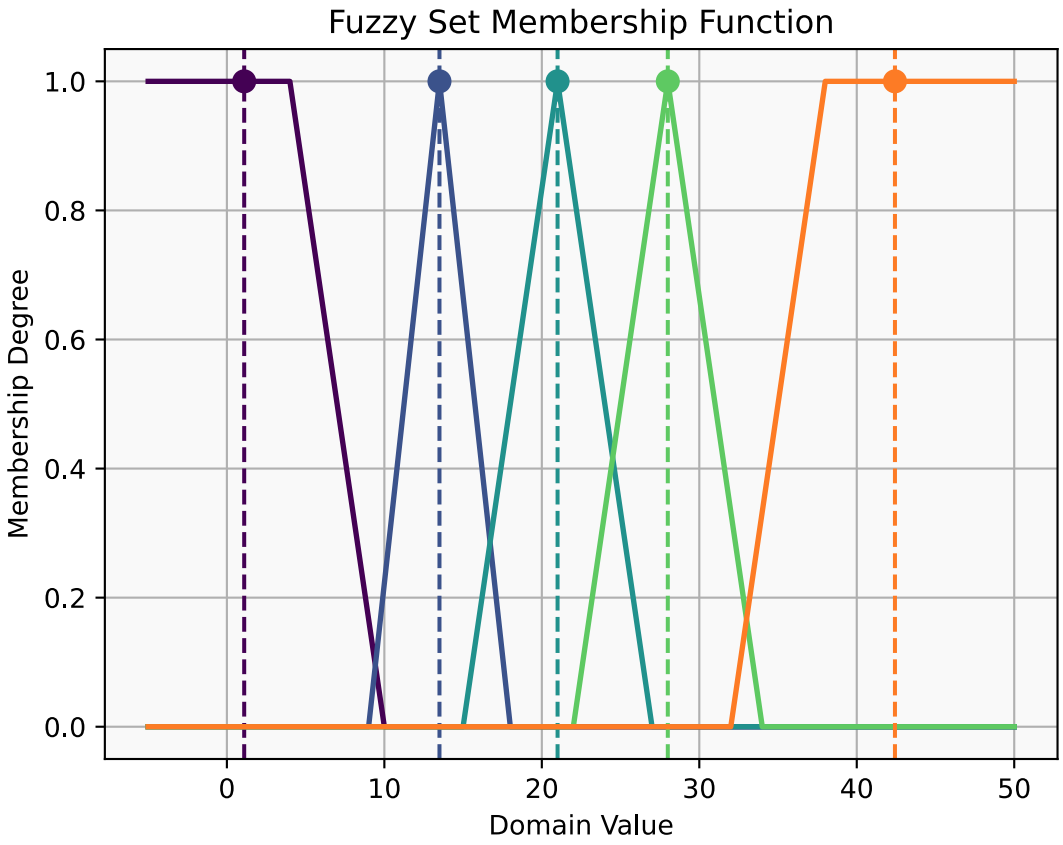
The concept of fuzzy number plays fundamental role in formulating **quantitative fuzzy variables**: those are (mathematical) variables whose possible states are fuzzy numbers. In particular, fuzzy variables that represent linguistic concepts (*small*, *tall*, *hot*, ecc...) are also referred to as **linguistic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

Metaheuristics are a computational framework of techniques and practices used to solve optimization problems. Instead of finding an exact, analytical solution in a single step, which may be impossible or computationally expensive, metaheuristics allows one to find a "sufficiently good" approximated solution through several iterations, where at each step a candidate solution is refined. Since metaheuristics does not prescribe a fixed set of operations, to make use of it it is crucial to properly map the problem at hand to the abstract structures provided by the metaheuristic.

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 major underpinning of biological 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 time. 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.

Formally speaking, an **optimization problem** is a pair (Ω, f). Ω is a set called **search space** that contains all the potential solutions (the "candidates") whereas $f : \Omega \mapsto \mathbb{R}$ is a function called **evaluation function**, that assigns a (real) number to each potential solution $\omega \in \Omega$, representing "how good" said solution is. The "best" solutions, also called **exact solutions**, are those that return the highest value for the evaluation function.

An element $\omega^* \in \Omega$ is an exact of the optimization problem (Ω, f) if and only if it is an **optimum**, either a minimum ($\forall \omega \in \Omega, f(\omega^*) \leq f(\omega)$) or a maximum ($\forall \omega \in \Omega, f(\omega^*) \geq f(\omega)$). Exact solutions can be more than one: in that case, one of them can be chosen arbitrarely. Also note that the search space is, in general, not the entire set of real numbers, but a subset of reals that satisfy some conditions, or **constraints**.

Exercise 3.1.1: 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:

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

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 (done automatically in Python):

```
from sympy import solve
from sympy.abc import x, y, z, L, S
eq1 = y * z + 2 * L * (y + z)
eq2 = x * z + 2 * L * (x + z)
eq3 = y * x + 2 * L * (y + x)
eq4 = 2 * x * y + 2 * x * z + 2 * y * z - S
solve([eq1, eq2, eq3, eq4], [x, y, z, L])
```

Gives the exact solution $x = y = z = \sqrt{S/6}$. □

Optimization problems are ubiquitous in fields where the goal is to maximize the efficiency/performance/return of a process, such as routing problems (*Travelling Salesman Problem*), packing problems (*Knapsack problem*), or scheduling problems (such as air traffic or job scheduling). The approaches to solve them fall into four broad categories:

- **Analytical Solution**: finding the optimum of the evaluation function by computing it directly, such as zeroing the gradient, employs Lagrange multipliers, a KKT system, ecc... If a problem can be solved this way, it is advisable to do so, since the obtained solution is guaranteed to be actually optimal (and not an approximation). However, many problems cannot be solved analitically, for example because it's not possible to zero the gradient (if the degree of the equation is too high) or because the problem is NP-hard, and therefore too computational expensive;
- **Complete/Exhaustive Exploration**: finding the optimum of the evaluation function by trying every possible solution in the search space. Even though the approach is technically correct, since out of all the possible solutions there has to be one or more better than the others, if the search space is too big the approach quickly becomes inefficient. Also, if the search space is not discrete, the approach cannot be applied at all;
- **(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 sufficiently 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 exploit the structure of the search space and how the evaluation function assesses similar elements to control the search. The idea is to notice if it's possible to search for solutions in the search space not at random, but "steering" the search into promising directions and pruning directions that aren't worthwile. Of course, for this to be possible, the evaluation of similar elements of the search space must be similar.

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.

3.2. Evolutionary algorithms

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: a termination criteria):

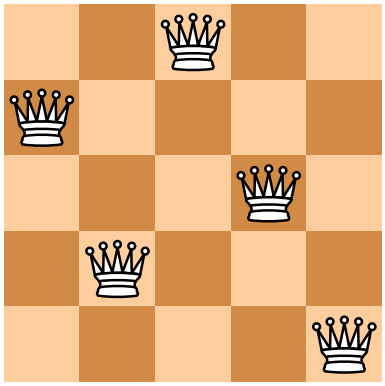
```
1  t ← 0
2  INITIALIZE (population(t))                          // Create the initial population
3  EVALUATE (population(t))                            // Compute fitness
4  while (ε = False):
5      t ← t + 1
6      population(t) ← SELECT-FROM (population(t − 1))  // Select individuals based on 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.

An instructive example of a genetic algorithm is the one that solves the **n-queens problem**. The problem asks to find a way to place on a $n \times n$ chessboard n queens in such a way that they cannot capture each other. That is, in such a way that no queen shares a row, a column of a diagonal with another queen. If the number of queens is very small, the problem can be solved by random search.

Exercise 3.2.1: What would be a possible solution for the 5-queens problem?

Solution:



The search space of the problem, even for the relatively simple case $n = 5$, has cardinality:

$$\frac{5^2!}{(5^2-5)!\ 5!} = \frac{25!}{20!\ 5!} = \frac{25 \times 24 \times 23 \times 22 \times 21}{5 \times 4 \times 3 \times 2 \times 1} = 53130$$

However, it is possible to figure out a possible solution with little effort just by trial and error, such as the one presented on the left.

□

One possible reasonable approach to solving the problem is employing a backtracking algorithm, doing an exhaustive exploration on the search space with a depth-first search. The algorithm exploits the fact that each row of the chessboard must contain exactly one queen. The idea is to place queens row by row, checking in every possible square if this placement obstructs an already placed queen; if this is the case, or if the recursion to the next row returns with the result that no solution can be found because obstructions could not be avoided, the queen is removed again and the algorithm continues with the next square. If no obstructions are found, the

algorithm proceeds recursively to the next row, and terminates when a queen has been (properly) placed on each row. The pseudocode is as follows:

```
N-QUEENS-BACKTRACK(n: integer, k: integer, board: matrix of booleans):
1  if (k ≥ n)
2      return True
3  else
4      for i = 0 to n − 1 do
5          board[i, k] ← True
6          col ← True
7          for j = 1 to k − 1 do
8              if (board[i, j] = False)                    // Another queen in this column
9                  col ← False
10                 break
11         row ← True
12         for j = 1 to min(k, i) do
13             if (board[i − j, k − j] = False)
14                 row ← False
15                 break
16         diag ← True
17         for j = 1 to min(k, n − i − 1) do
18             if (board[i + j, k − j] = False)
19                 diag ← False
20                 break
21         rec ← N-QUEENS-BACKTRACK (n, k + 1, board))    // Recursion succeeds
22         if (row and col and diag and rec)
23             return True
24         else
25             board[i, k] ← False
26     return False
```

The algorithm can be easily extended to yield all possible solutions of the problem, not just one solution. However, the approach is very inefficient; if the interest is in finding just one possible solution disregarding the others, a better approach is a completely analytical solution. Given a chessboard size n, the algorithm is as follows:

- If $n = 2$ or $n = 3$, the n-queens problem has no solution;
- If n is odd, a queen is placed onto the square $(n-1, n-1)$ n is decremented by 1 and the algorithm is repeated;
- If $n \bmod 6 \neq 2$, then the queens in the rows $y = 0, ..., \frac{n}{2} - 1$ are placed in the columns $x = 2y + 1$ and the queens in the rows $y = \frac{n}{2}, ..., n - 1$ are placed in the columns $x = 2y - n$;
- If $n \bmod 6 = 2$, then the queens in the rows $y = 0, ..., \frac{n}{2} - 1$ in the columns $x = (2y + \frac{n}{2}) \bmod n$ and the queens in the rows $y = \frac{n}{2}, ..., n - 1$ in the columns $x = (2y - \frac{n}{2} + 2) \bmod n$.

In order to solve the n-queens problem with an evolutionary algorithm, the first step is writing an encoding of the solution candidates. To do so, it is possible to exploit (again) the fact that each queen must be placed on a single row. A candidate solution can be described by a chromosome having n genes, where each gene refers to

one row of the chessboard (numbered 0 to $n-1$). Each gene has n possible alleles (numbered, again, 0 to $n-1$), each representing the position of the queen on the row.

With this encoding there is a clear distinction between the genotype of the problem (the array of numbers) and its phenotype (the actual placement of the queens on the chessboard). Also, assigning a chromosome to each row has the advantage of excluding candidate solutions with more than one queen per rank (that aren't valid anyway). As a consequence, the search space becomes much smaller, and thus can be explored more quickly and more effectively.

The initial population has no particular requirement (outside having one queen per row), therefore it can be constructed just as random sequences of numbers. Each element of the chromosome is therefore assigned a random number from 0 to $n-1$, and a certain number k of chromosomes is constructed this way.

The fitness function can be constructed by summing all the obstructions for each queen, dividing the result by 2 and changing its sign. Clearly, a solution will have a value of the fitness function equal to 0, whereas any other solution will yield a negative value. The division by two is introduced to remove dupicates, since the obstruction is always symmetric (if queen A obstructs queen B, then queen B also obstructs queen A), whereas the negative sign is introduce in order to have a maximization problem, instead of a minimization.

The termination criteria for this algorithm is, clearly, having found a solution whose value of the fitness function is 0. However, to be more cautios, it is advisable to also introduce an additional criteria that stops the algorithm when a sufficient number of generations has been reached, returning whatever solution has been found (that will be, most likely, suboptimal).

A possible and widely employed selection method that can be used in this context would be the **tournament selection**. The idea is to sample a subset of individuals from the population and then choose the individual that has the highest value of the fitness function (the one that "wins" the "tournament"); if there were to be a tie between two individuals, one of them is chosen at random. The "winning" individual is added to the next generation of the population, whereas the "losing" individual are shuffled back in the current population. This process is repeated until the next generation of the population is complete, which generally just means that it has reached the same size as the current population.

In order to alter the individuals of the population, it is necessary to introduce a genetic operator both for recombination and for variation. The first "shuffles" the genetic traits of the individuals, creating new individuals possessing a combination of traits that neither of the two possess, while the second introduces new traits into the population, traits that no individual might possess.

A choice for the recombination operator is the **one-point crossing-over**, that mimics the way crossing-over functions during gamete duplication. The operator, out of a sample of the entire population, chooses two individuals and then first splits their chromosomes in a random point into two parts, then exchanges the matching parts of the two chromosomes (first part of the first with first part of the second, second part of the first with the second part of the second). This way, two new individuals are created, whose traits may be better than the traits of their "parents".

A choice for the variation operator is the **standard mutation**, where to each gene of each individual is assigned a probability of transitioning into a new random value. Introducing variation is necessary, since recombination alone would just entail reshuffling the existing traits over and over, without the possibility of new alleles. Note that both variation and recombination might result in an individual actually having lower fitness than before, but this is to be expected, since most real-life genetic mutations are indeed detrimental.

The last step is to fix the values of several parameters, in particular: the size of the population to evolve, the maximum number of generations to compute before terminating, the sample size of the tournaments, the fraction of individuals that are subject to crossing over and the probability that a gene is subject to a mutation.

3.3. Related local search algorithms

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.

3.3.1. Gradient ascent and gradient descent

Gradient ascent/descent is an optimization technique focusing on optimizing real-valued functions, meaning locally optimal points of functions when an analytical solution (that is, computing the gradient and setting it equal to 0) is either impractical or impossible to compute. The idea is to evaluate the gradient of the function in a random point, moving to a point in its neighborhood where the gradient is bigger (ascent) or smaller (descent) and repeating the process. This way, over many iterations, there's a guarantee to reach, if not a optimum, at least getting very close.

To determine where to move after having computed the gradient in the current point, 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.

Even though the choice of the direction to take is clear, the choice of "how much" to move in said direction is trickier. Moving in steps that are too small can result in a very slow procedure, whereas moving in steps that are too big can result in a process that oscillates back and forth in the neighborhood of an optimum without actually reaching it. A possible solution would be to move in steps whose size depend on 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.

Since it's not guaranteed that gradient descent will actually find an exact solution, it is necessary to introduce some termination criteria. For example, the process can be terminated when a maximum number of iterations is reached or when the difference between the current and previous solution is smaller than a given difference (that is, there is negligible improvement).

The gradient descent procedure can be stated as follows:

GRADIENT-ASCENT/DESCENT($f : \mathbb{R}^n \to \mathbb{R}$: function, ν: integer, ε: termination criteria):

1 $x = (x_1, ..., x_n) \leftarrow$ RANDOM (n) // Randomly initialize a point
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 $\quad x \leftarrow x \pm \eta\nabla$ // Update candidate solution
5 **while** (**not**(ε))
6 **return** x // Return best solution found

Note that the choice of a starting point for the procedure noteably influences its outcome since, clearly, gradient ascent/descent will take longer when choosing a point that is far from an optimum. Also, choosing a different starting point can result in finding a local or a global optimum, (unless the function is either concave or convex). The only countermeasure is to execute the process multiple times with different starting points and choosing the best result obtained.

3.3.2. Hill climbing

For gradient ascent/descent to be applicable, it is necessary that the function to be optimized is differentiable everywhere, otherwise the gradient cannot be computed.

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 yield 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. The process is repeated until a termination criterion is reached.

This approach, which can be thought of as a "naive" gradient descent, is called **Hill climbing**. The algorithm is as follows:

HILL-CLIMBING($f : \mathbb{R}^n \to \mathbb{R}$: function, ε: termination criteria):

```
x = (x_1, ..., x_n) ← RANDOM (n)                          // Randomly initialize a point
do
    x' = (x'_1, ..., x'_n) ← RANDOM-NEIGHBORHOOD (n, x)   // Random point in the neighborhood
    if (f(x') > f(x))
        x ← x'                                            // Update if there is improvement
while (not(ε))
return x                                                  // Return best solution found
```

Even though this approach can be applied to more classes of functions, it inherits the same issues of gradient ascent/descent, mainly the tendency of getting stuck in local optima.

3.3.3. Simulated annealing

The issue of getting stuck in local optima can be overcome by allowing the process to accept solutions that are suboptimal on the short term, but with the intent of moving towards an even better solution. This is exploited in a local search algorithm called **Simulated annealing**.

The idea is to start from a random point in the domain of the function and try points in its neighborhood for improvements, like hill climbing, but introducing a probability of accepting a worse solution as the current solution candidate. Even though this does not guarantee to actually land in the vicinity of a better solution, there is still a possibility of this to happen.

More specifically, if the point in the vicinity of the current solution yields a better value for the function to optimize, this point is chosen as the new candidate solution. On the other hand, if the point in the vicinity of the current solution yields a worse value, this point is chosen as the new candidate solution with a certain probability.

This probability depends both on the difference of the value of the function to optimize yielded by the current candidate and the new candidate and on a parameter, called **temperature**, that decreases iteration by iteration. If the difference between the current and the new candidate is small, the algorithm will be more inclined to tolerate a worse solution. On the other hand, the tolerance of the algorithm to accept a worse solution will be higher in the earlier iterations, when the temperature is high, and becomes more and more "conservative" as the iterations progress.

The algorithm for simulated annealing is as follows:

SIMULATED-ANNEALING($f : \mathbb{R}^n \to \mathbb{R}$: function, T: float, ε: termination criteria):

```
x = (x_1, ..., x_n) ← RANDOM (n)                    // Randomly initialize a point
Δ_max ← 0                                           // Variability across solutions
do
    x' = (x'_1, ..., x'_n) ← RANDOM-NEIGHBORHOOD (n, x)   // Random point in the neighborhood
    Δ ← f(x') − f(x)                                // Improvement size
    if (|Δ| > Δ_max)
        Δ_max ← |Δ|
    p ← e^{Δ/Δ_max T}                               // Tolerance to worse solutions
    if (Δ > 0 or p ≥ RANDOM-VALUE (0, 1))           // New solution is better or just tolerated
        x ← x'
    T ← T − 0.1                                     // Decrease temperature
while (not(ε))
return x                                            // Return best solution found
```

Note how, when a small value of the temperature is chosen, the algorithm is pretty much identical to hill climbing, since the probability of the algorithm to accept a worse solution is almost zero.

3.3.4. Threshold accepting

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

THRESHOLD-ACCEPTING($f : \mathbb{R}^n \to \mathbb{R}$: function, θ: float, ε: termination criteria):

```
x = (x_1, ..., x_n) ← RANDOM (n)                    // Randomly initialize a point
do
    x' = (x'_1, ..., x'_n) ← RANDOM-NEIGHBORHOOD (n, x)   // Random point in the neighborhood
    if (f(x) − f(x') < θ)                           // New solution is better or just tolerated
        x ← x'
    θ ← θ − 0.1                                     // Decrease threshold
while (not(ε))
return x                                            // Return best solution found
```

3.3.5. Great Deluge Algorithm

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

GREAT-DELUGE($f : \mathbb{R}^n \to \mathbb{R}$: function, θ_0: float, η: float, ε: termination criteria):

```
x = (x_1, ..., x_n) ← RANDOM (n)                          // Randomly initialize a point
t ← 0                                                      // Initialize iteration counter
do
    x' = (x'_1, ..., x'_n) ← RANDOM-NEIGHBORHOOD (n, x)    // Random point in the neighborhood
    if (f(x') ≥ θ_0 + t · η)                               // New solution is better or just tolerated
        x ← x'
    t ← t + 1                                              // Increase iteration counter
while (not(ε))
return x                                                   // Return best solution found
```

3.3.6. Record-to-Record Travel

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

RECORD-TO-RECORD-TRAVEL($f : \mathbb{R}^n \to \mathbb{R}$: function, θ: float, ε: termination criteria):

```
x = (x_1, ..., x_n) ← RANDOM (n)                          // Randomly initialize a point
x_best ← x                                                 // Initialize best solution
do
    x' = (x'_1, ..., x'_n) ← RANDOM-NEIGHBORHOOD (n, x)    // Random point in the neighborhood
    if (f(x') ≥ f(x_best) − θ)                             // New solution is better or just tolerated
        x ← x'
    if (f(x') > f(x_best))                                 // New solution better than the best one
        x_best ← x'                                        // Increase iteration counter
while (not(ε))
return x_best                                              // Return best solution found
```

3.3.7. Case study: the Traveling Salesman Problem

A **Eulerian cycle** of a connected weighted graph is a path that starts and ends in the same node of the graph and that reaches all of its nodes exactly once. The Eulerian cycles having the smallest cumulative weight are called **Hamiltonian cycles**.

One of the most famous problems in mathematical optimization, the **Travelling Salesman Problem** (**TSP**), requires to find, for a given weighted connected graph, an Hamiltonian cycle. The name of the problem comes from the analogy of a traveller that has interest in reaching all cities of their trip (the nodes of the graph) moving from city to city along the roads (the edges on the graph) whose time needed to be moved along is cumulatively the smallest (an Hamiltonian cycle).

More formally, let $G = (V, E, W)$ be a 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. The Travelling Salesman Problem is the optimization problem (Ω, f), where Ω is the set of all possible Eulerian cycles, that is, the set of all possible permutations of indices of the vertices that, two by two, have an edge that connects them:

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

And where the function f is the sum of all the weights of an Eulerian cycle in Ω:

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

The mod n is just to ensure that the last vertex "loops back" and connects to the first.

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.

Under these assumptions, an approximate solution of the TSP can be obtained by applying local search algorithms, such as simulated annealing. To employ such algorithm, it is sufficient to define how to construct an "neighboring" solution (since the input is graphs, the notion of neighborhood is spurious) and how to define the probability of accepting a worse solution.

Given a permutation π, a neighboring solution can be constructed 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.

As always, a new solution is to be accepted as new candidate either if it's better than the current candidate (yields a greater value of the objective function) or, if worse, with probability $e^{\Delta/\Delta_{\max}T}$. The range of qualities $\Delta_{\max}$ can be estimated as:

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

With t being the number of the current iteration and $Q_{\max,t}$ and $Q_{\min,t}$ being the highest qualities found so far among candidate solutions. The temperature can be decreased just as $T = 1/t$.