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.

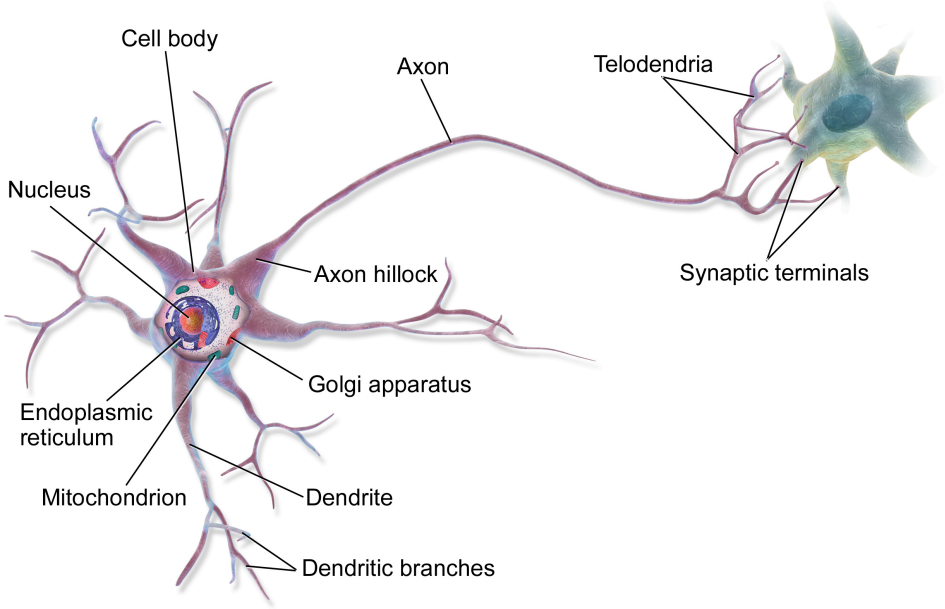


Figure 1: Schematic image of a neuron. By BruceBlaus, CC BY 3.0, via Wikimedia Commons. original image

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.

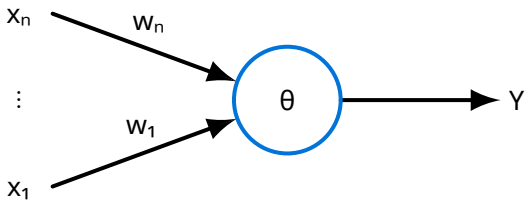
1.2. Threshold Logic Units

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.



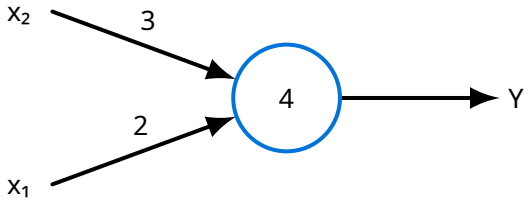
$$y = \begin{cases} 1 \text{ if } \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

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.

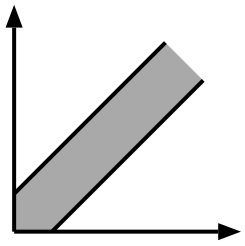
A set of points in the plane is called **convex** if connecting each point of the set with straight lines does not require to go outside of the set. The **convex hull** of a set of points is its the smallest superset that is convex. If two sets of points are both convex and disjoint, they are linearly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

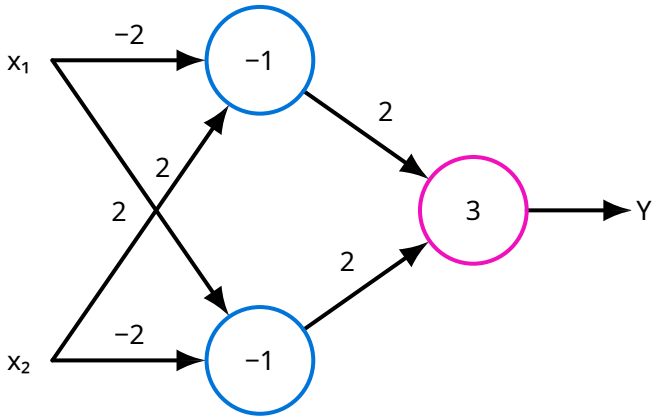


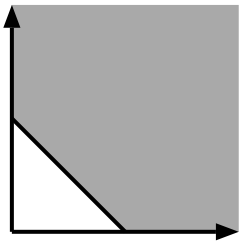
□

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:

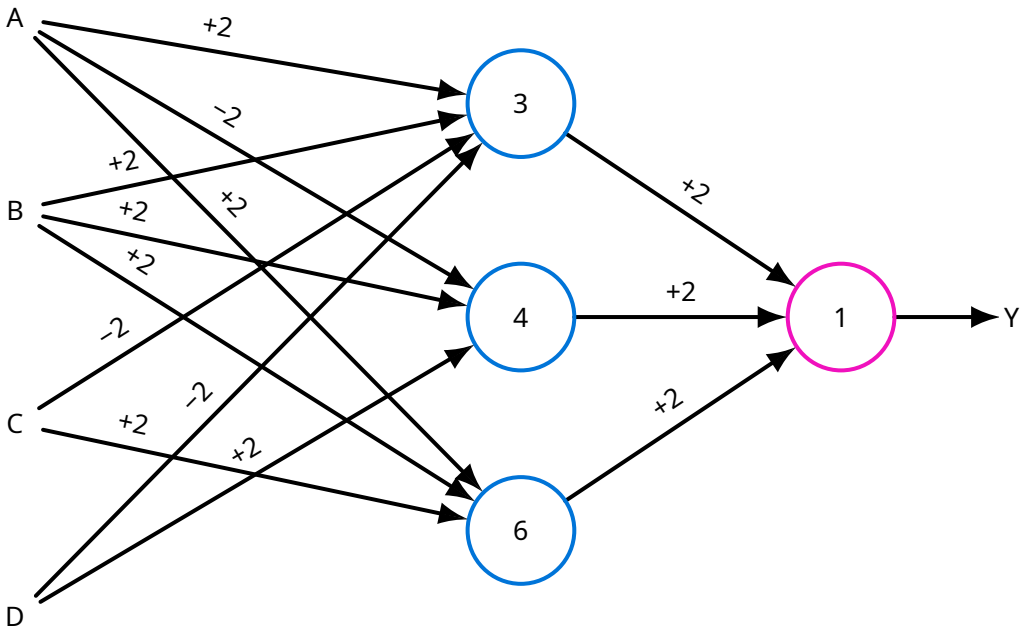
$$w_{j,i} = \begin{cases} +2 \text{ if } l_{j,i} \text{ is a positive literal} \\ -2 \text{ if } l_{j,i} \text{ is a negative literal} \end{cases} \qquad \theta_j = n - 1 + \frac{1}{2} \sum_{i=1}^{n} w_{j,i}$$

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
```

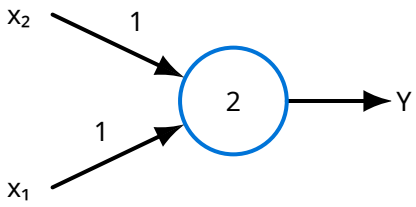
Exercise 1.3.1: Construct a TLU that computes the logical AND between two bits.

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.

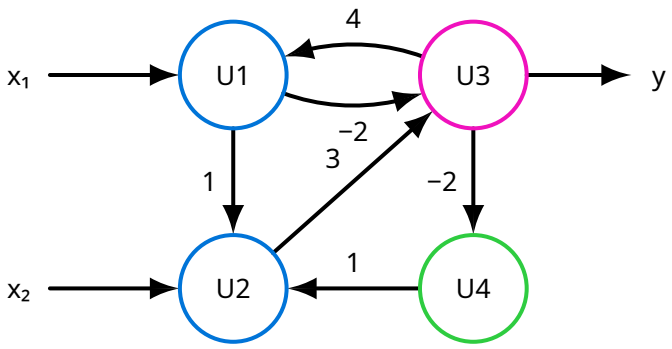
$$U = U_{\text{in}} \cap U_{\text{out}} \cap U_{\text{hidden}} \qquad U_{\text{in}} \neq \emptyset, U_{\text{out}} \neq \emptyset, U_{\text{hidden}} \cap (U_{\text{in}} \cap U_{\text{out}}) = \emptyset$$

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.

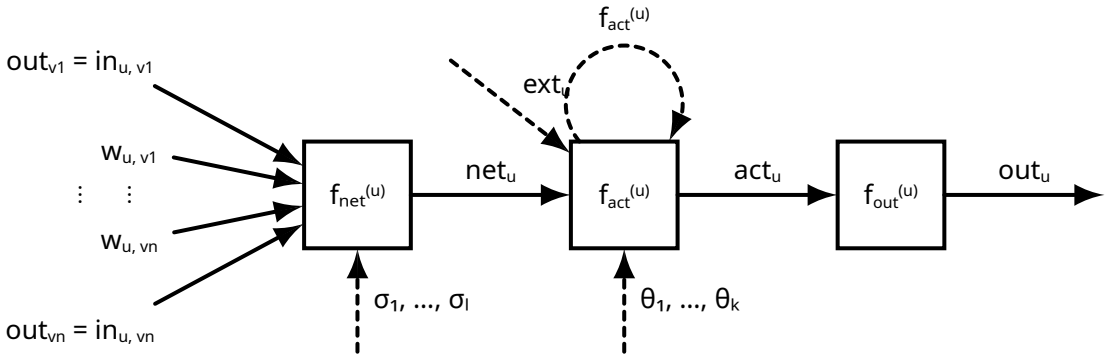


Figure 3: Structure of a neuron

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:

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

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

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

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**):

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

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:

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

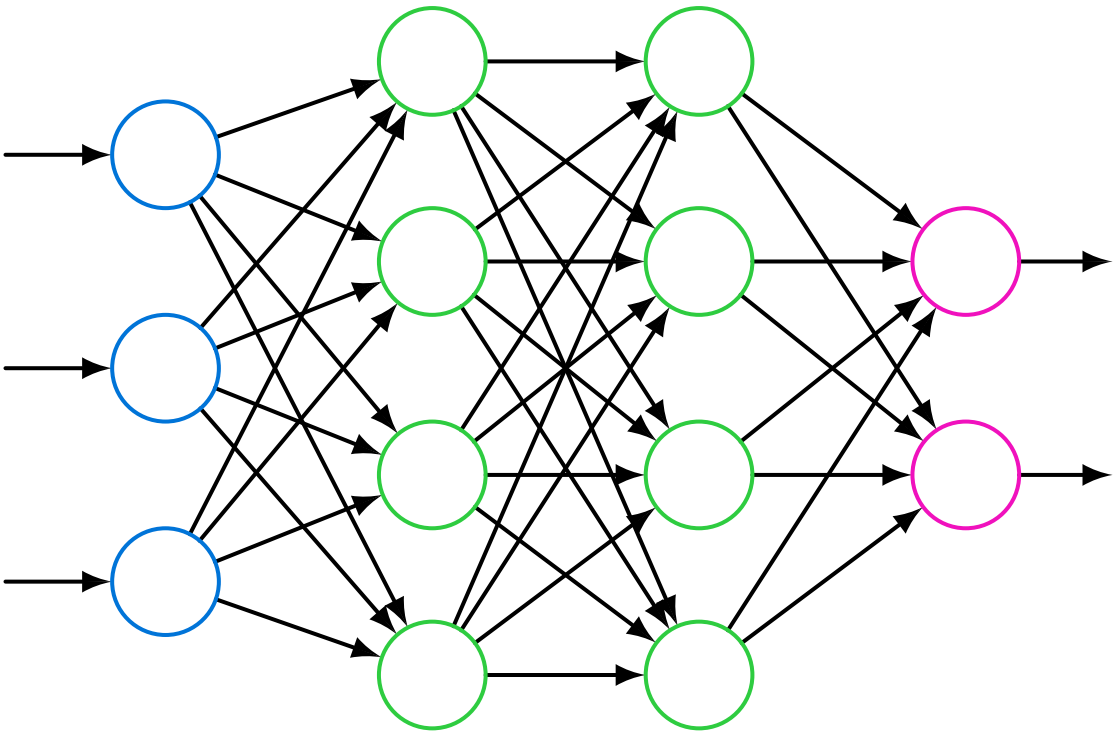


Figure 4: Structure of a generic multilayer perceptron

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:

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

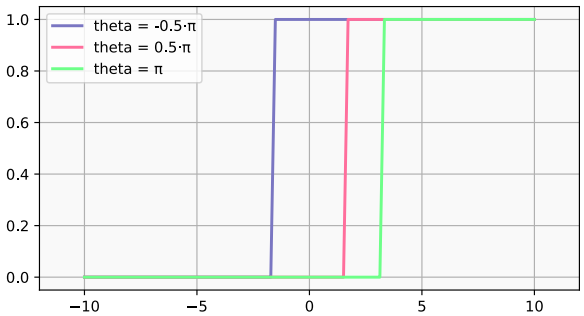
The activation function of hidden neurons is any **sigmoid function**, meaning a monotonic non-decreasing function of the form:

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

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:

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



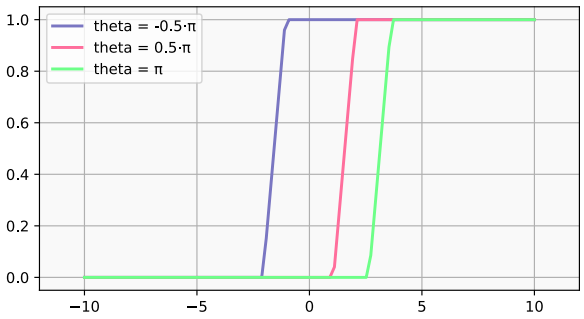
This is because, as it was done for the TLUs, it is possible to move the threshold into the weighted sum and obtain an equivalent function that output 0 if the weighted sum is negative (less than 0) and outputs 1 if positive (greater than 0), and this check can be done by simply looking at the most significant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

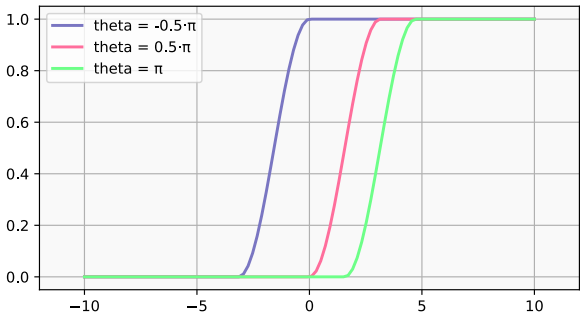
$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1 & \text{if net} > \theta + \frac{1}{2} \\ 0 & \text{if net} < \theta - \frac{1}{2} \\ (\text{net} - \theta) + \frac{1}{2} & \text{otherwise} \end{cases}$$



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:

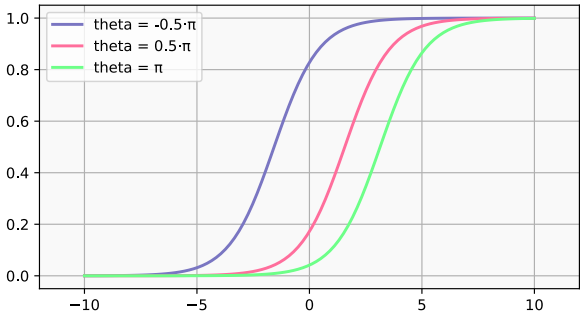
$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1 & \text{if net} > \theta + \frac{\pi}{2} \\ 0 & \text{if net} < \theta - \frac{\pi}{2} \\ \frac{\sin(\text{net} - \theta) + 1}{2} & \text{otherwise} \end{cases}$$



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.

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

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

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

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

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

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

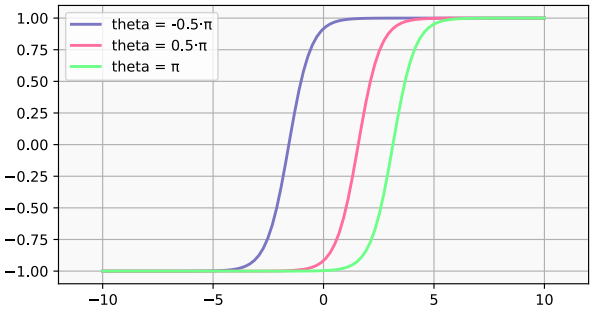
$$e^{-(\text{net}\ -\theta)} f_{\text{act}}(\text{net}, \theta) + f_{\text{act}}(\text{net}, \theta) = 1 \Rightarrow e^{-(\text{net}\ -\theta)} f_{\text{act}}(\text{net}, \theta) = 1 - f_{\text{act}}(\text{net}, \theta) \Rightarrow$$

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

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

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

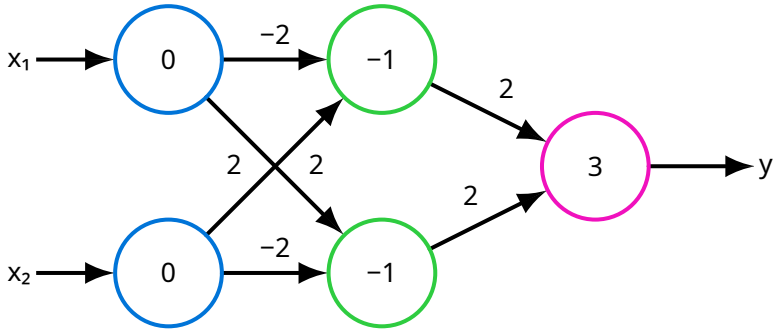
A clear advantage of having a weighted summation as the network input function of a multilayer perceptron is that it translates naturally to matrix multiplication. Let $U_1 = (v_1, ..., v_m)$ and $U_2 = (u_1, ..., u_n)$ be two subsequent layers (U_2 is right after U_1). It is possible to write the network 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 $x_1, x_2, \ldots, x_i$.

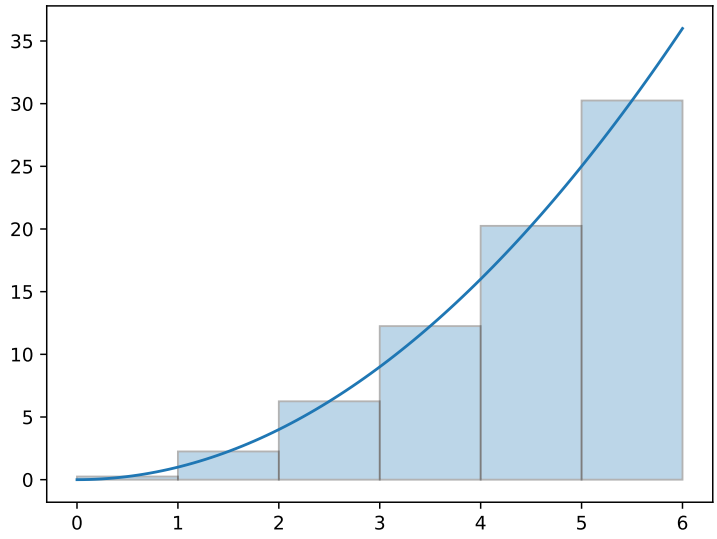
Each pair of adjacent cutoff points induces $n - 1$ intervals $[x_1, x_2], [x_2, x_3], \ldots, [x_{n-1}, x_n]$. Each of these intervals will (more or less accurately) give an approximation for all the values of the true function in said range (the most natural choice of this approximation is the middle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

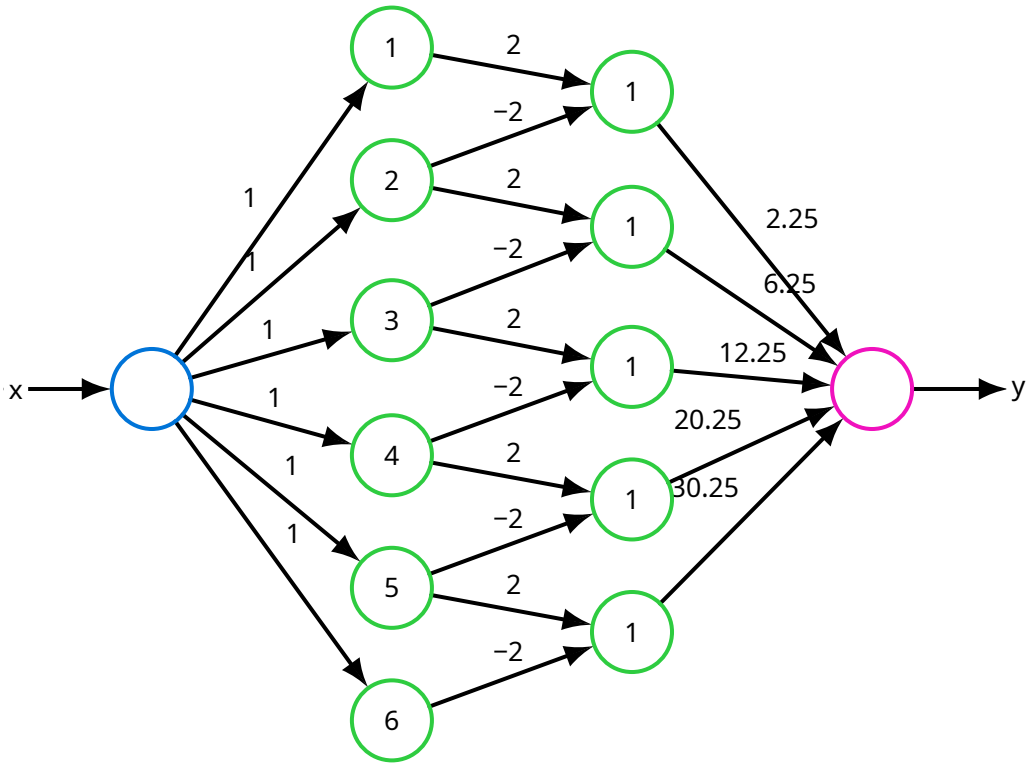
From the output of the network it is possible to know which is the best approximation for a given input, since each of the incoming weights of the input neuron is set to the chosen approximations of the function evaluated at the given input, and only one neuron of the second hidden layer will fire.

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:

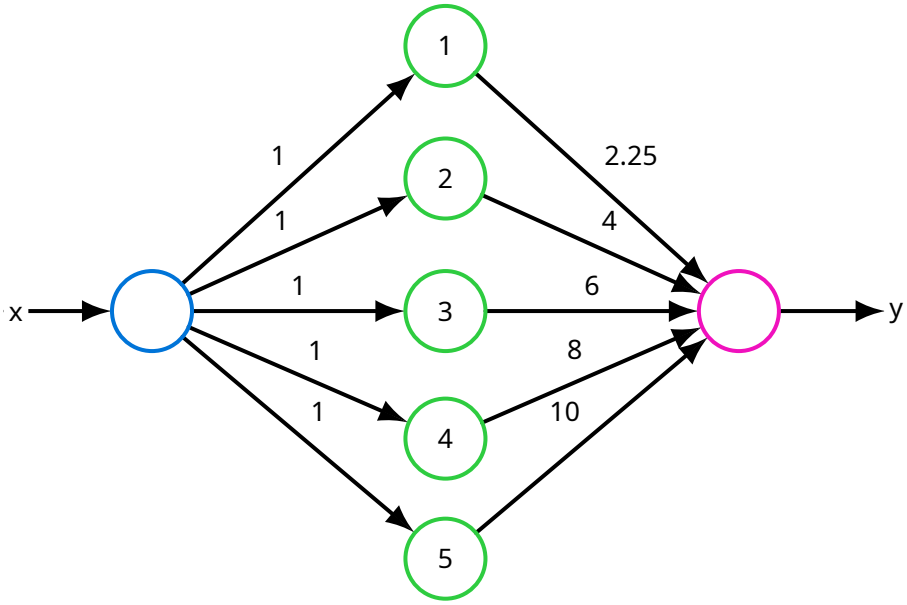
Theorem 1.5.2: Any continuous Riemann-integrable function can be approximated with arbitrary accuracy by a multilayer perceptron of three layers.

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.