Artificial Intelligence

Massimo Perego

Contents

1 Neural Networks

1.1 Biological Background

Neural Networks need a biological introduction since they're **inspired from the human brain**, neurons in the network are simplified versions of the human neurons.

The human brain is studied in multiple different fields various reasons, in Computer Science the objective of studying the brain is **mimicking certain cognitive capabilities** of human beings, with the **objective** of **solving** learning/adaptation, prediction and optimization **problems**.

But why? Neural networks in Artificial intelligence allow for **highly parallel information processing**.

Neurons have a **cell body**, called **soma**, from which extend several **short ramified branches** called **dendrites** and a **long extension** called **axon**; at its end the axon is heavily ramified.

The axons are fixed paths along which neurons communicate with each other; the axon of a neuron leads to the dendrites of another. The place in which an axon and a dendrite almost touch each other is called **synapse**.

A terminal button of the axon releases **neurotransmitters** that act on the membrane of the receiving dendrite, changing its polarization.
Synapses that reduce the potential difference are called **excitatory**, those that increase it are called **inhibitory**.

If there is enough net excitatory input the **axon is depolarized** (the potential difference is significantly reduced), the cell is "activated", the change in electrical potential is propagated to another cell, where the process is repeated; when this action potential reaches the terminal buttons, neurotransmitters are released.

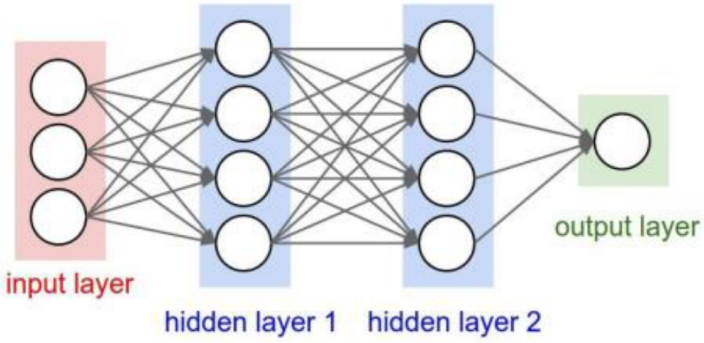
Changes in electrical potential are accumulated at the cell body of a neuron and, if they reach a certain threshold, are propagated along the axon.

Synapses are what enables parallel information to travel, a single neuron can activate multiple synapses and activate many more neurons, permitting complex reasoning.

Advantages of **Neural Networks**:

- **High processing speed** due to massive parallelism given by large number of neurons (in CS implemented by CUDA architectures)
- **Fault Tolerance**: remain functional even if some parts of a network get damaged; if a neuron goes crazy the rest will still function, even if a little bit worse
- **"Graceful Degradation"**: gradual degradation of performance if an increasing number of neurons fails (consequence of the fault tolerance)
- **Well suited for inductive learning** (learning from examples, generalizing from instances, usually the point of a neural network is to have some examples and generalize from there)

It appears to be reasonable to try **mimicking or to recreate these advantages by constructing artificial neural networks**.



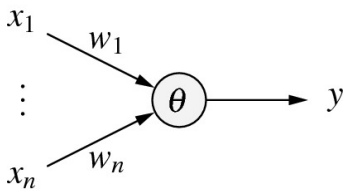
This is the scheme of a standard multi-layer feed-forward neural network (the standard before convolutional neural networks).

1.2 Threshold Logic Unit

The first **abstract model** for an artificial neuron of the brain. A **threshold logic unit** (TLU) is a processing unit (neuron) with several inputs. It can solve a very simple set of problems. **McCulloch-Pitts neuron**

There are several inputs reaching to the **core** (neuron), whose output is delivered to subsequent neurons which are connected with it.

A TLU is a processing unit in which the **output** is **governed** by a **threshold** θ, if it has a **sufficient excitation** from the inputs, then the TLU becomes **active** (value 1) and generates the output y.



There are n inputs $x_1, .., n_n$ and the TLU generates one output y. Each input is **weighted** (w_1, ... , w_n), some can be more relevant than others (we're considering data from the external world after all, not everything must be considered equally).

TLU conditions:

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

The result is more dependent on the more important information. We can control how much of an influence each input has, determined by its weight w.

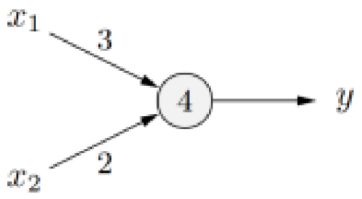
When the TLU is solicited enough if the **excitation** (weighted sum of the inputs) is **over the threshold** θ the output delivered to the terminal synapse and then to another neuron will be 1, 0 otherwise.

1.2.1 Conjunction Example

The **result** is equal to **1 only when the two outputs are equal to 1**. There isn't a standard way to select the threshold θ, it has to be chosen based on the function that has to be implemented.

In this case the θ chosen has to be greater than any individual weight.

$$x_1 \wedge x_2$$



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

(if I sum them, are they enough?).

In this example $w_1 \cdot x_1 + w_2 \cdot x_2 = 3 \cdot x_1 + 2 \cdot x_2$ has to be ≥ 4 to have $y = 1$.

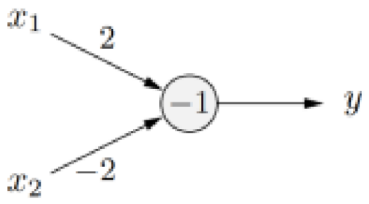
1.2.2 Implication Example

I can choose the value of the threshold in order to have the proper function, in this case applies on the same ways. How can I choose the interconnection weights?

The problem is that there are no general rules, I choose the weights according to the intrinsic relevance of each input variable.

How can I do that when we have a high number of inputs? We'll see that there is a procedure for doing that.

$$x_2 \to x_1$$

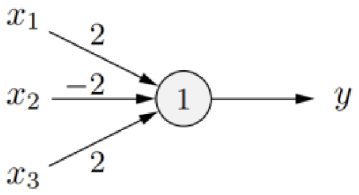


x_1	x_2	$2x_1 - 2x_2$	y
0	0	0	1
1	0	2	1
0	1	-2	0
1	1	0	1

This can also be expressed as $\neg x_2 \vee x_1$, so the output is true unless x_2 is true and x_1 is false.
The output "goes on" if the second one (x_1 in the example) is true without the first one (x_2 in the example) being true.
The weights must be tuned accordingly.

1.2.3 Multiple inputs example

In this case we can see that we have three possible inputs, we can discriminate the inputs in excitatory input and inhibitory input. The first tries to contribute to the final computation of the neuron in such a way that the results will be greater than the threshold, the other neuron does the opposite.



x_1	x_2	x_3	$\sum_i w_i x_i$	y
0	0	0	0	0
1	0	0	2	1
0	1	0	-2	0
1	1	0	0	0
0	0	1	2	1
1	0	1	4	1
0	1	1	0	0
1	1	1	2	1

The result is the weighted sum of multiple inputs, some of which "help" the output towards 1 while others "drag it" towards 0.

1.3 Geometric interpretation

The geometric interpretation is significantly **helpful** to derive a method to **configure the threshold and the weights starting from the data**.

We will consider a single and simple TLU, we will try to understand how we can **interpret the behavior of the TLU** in a geometrical way.

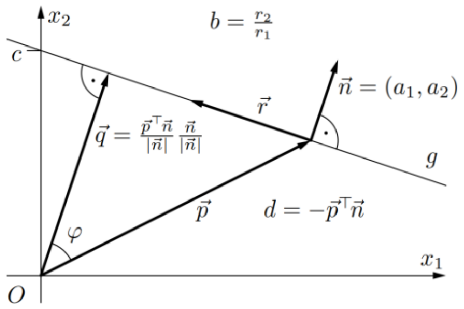
It's possible to represent a straight line on a plane in any of the following forms:

- Explicit form: $g \equiv x_2 = bx_1 + c$
- Implicit form: $g \equiv a_1 x_1 + a_2 x_2 + d = 0$
- Point-Direction Form: $g \equiv \vec{x} = \vec{p} + k\vec{r}$
- Normal Form: $g \equiv (\vec{x} - \vec{p})^\top \vec{n} = 0$

Any of these representations works to represent a straight line in the plane; we're considering just two variables x_1 and x_2.

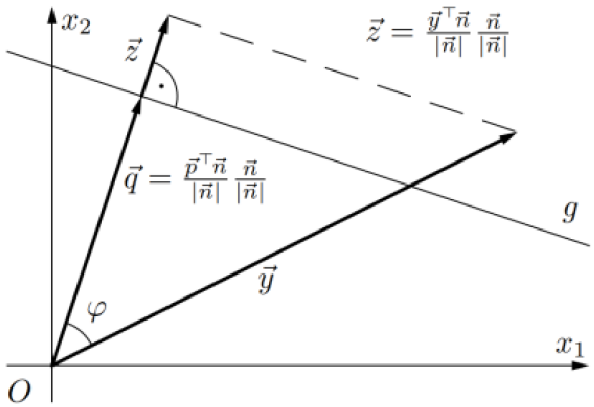
The parameters are:

- b: Gradient of the line
- c: section of the x_2 axis (intercept)
- $\vec{p}$: Vector of a point of the line (base vector)
- $\vec{r}$: Direction vector of the line
- n: Normal vector of the line



In the case of the explicit/implicit form b is the inclination (or gradient) of the line in respect to the horizontal axis, and c is the interception with the vertical axis.

The normal $\vec{n}$ is the vector orthogonal to the straight line. The vector $\vec{p}$ identifies a point on the line. The distance from the line to the origin O is given by $|\vec{q}|$.



We can **determine on which side of the line a point lands**, if we take the vector representing a point $\vec{y}$, and we compute the projection in the direction of the normal, the projection of this is the vector $\vec{z}$.

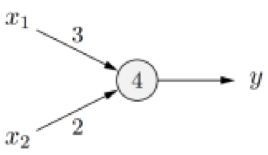
To understand on which side the points are, I just need to see if the vector $\vec{z}$ (which is the projection of our point) is shorter or longer than the point I observe on the straight line, which is pointed by $\vec{q}$. Basically, is the projection along the normal $\vec{n}$ shorter or longer (module is higher or lower) than our distance $\vec{q}$?

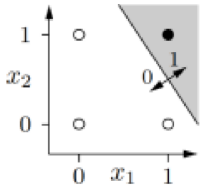
This means that all points (expressed by a vector) which have a module higher than the projected point onto the straight line ($\vec{q}$), are part of the plane above the straight line (they will satisfy the solution), vice versa, they will be below the plane if the module is shorter than $\vec{q}$ (they don't satisfy the condition).

Basically the **straight line** which **defines the behavior of our TLU**, splits the **plane in two parts** (since we're considering only x_1 and x_2).

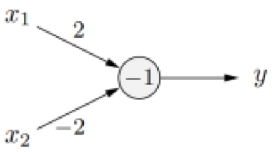
Solution for the conjunction and implication

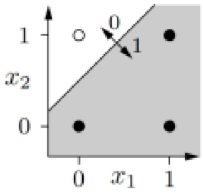
TLU for $x_1 \wedge x_2$





TLU for $x_2 \rightarrow x_1$



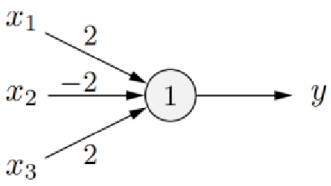


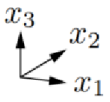
Going back to our earlier examples, for the conjunction this means that there are 3 points in which the output will be 0 and 1 where the result is 1. The line can be represented as

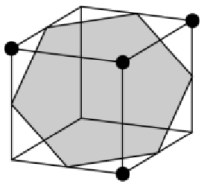
$$3x_1 + 2x_2 - 4 = 0 \Leftrightarrow x_2 = 2 - \frac{3}{2}x_1$$

For the implication the output is 1 when the values are under the line, i.e. in 3 of the cases, under the line $2x_1 - 2x_2 + 1 = 0$.

For three variables the idea has to be generalized to a three-dimensional space, the line becomes a plane which divides the output values.







Using our information the solution can be divided by this hexagon-looking plane (it's just "incomplete"), represented by the formula

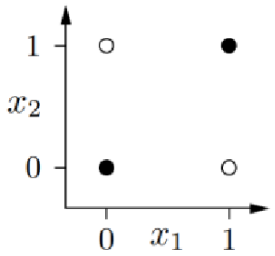
$$2x_1 - 2x_2 + 2x_3 - 1 = 0$$

How are the **threshold and interconnection weights chosen**? We have to look at the geometrical distribution of the points for each possible output and have a line/plane that clearly separates each group.

If I can **clearly divide the groups I can use TLUs** to solve the problem.

For the bi-implication problem $x_1 \leftrightarrow x_2$

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



There is no dividing line, the output groups cannot be separated with a straight line. As a consequence we can't have a TLU solving this problem.

1.3.1 Linear Separability

Two sets of points in a **Euclidean space** are called **linearly separable**, if there exists at least one point, line, plane or hyperplane (depending on the dimension of the Euclidean space), such that **all points** of **one set** lie on **one side** and all points of the **other set** lie on the **other side** of this point, line, plane or hyperplane (or on it).

The point sets can be separated by a **linear decision function**.

This works in n dimensions regardless; any TLU can have n different inputs and each one will be represented by a dimension.

If I have a mono-dimensional space (a line) there will be a point dividing the 2 sets, if we look at a plane a line will divide the sets, for a three-dimensional space a plane divides the sets, ecc.

1.3.2 Convex hull

A set of points in a Euclidean space is called **convex** if it is non-empty and connected (that is, if it is a region) and for every pair of points in it every point on the straight-line segment connecting the points of the pair is also in the set.

In a Euclidean space a **convex set** is a set in which, for **each pair of points**, the **segment that connects them** is **entirely contained in the set**.

The **convex hull** of a set of points X in a Euclidean space is the smallest convex set of points that contains X. Alternatively, the convex hull of a set of points X is the intersection of all convex sets that contain X.

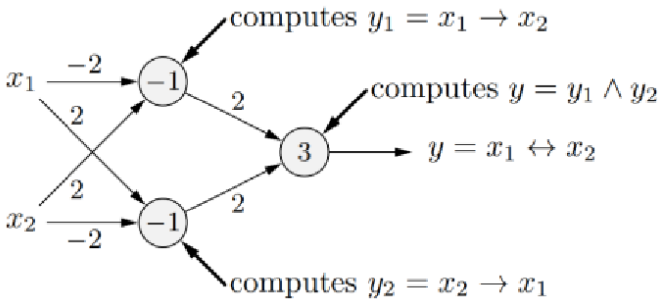
A **convex hull** is the smallest convex set of points that contains X.

1.4 Solution of the bi-implication problem

Two sets of points in Euclidean Space are **linearly separable if and only if their convex hulls are disjoint** (have no point in common).

In the bi-implication problem, the convex hulls are the diagonal line segments. They share their intersection point and this means that they are not disjoint, therefore the double implication is not linearly separable.

We can try solving the complex problems a single neuron can't solve with more neurons

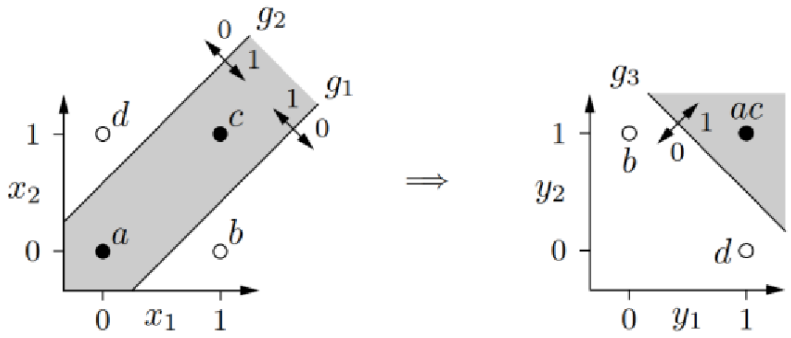


We are creating a network of TLUs and splitting the problem into subproblems.

The problem of implication can be solved in a single TLU and bi-implication is the intersection of implication from both ways so

$$(x_1 \to x_2) \wedge (x_2 \to x_1) \implies x_1 \leftrightarrow x_2$$

Let's see what happens geometrically



The first graph represents the intersection between the first two TLUs, one outputs 1 for all values over the line g_1 (which represents $x_2 \to x_1$), while the other outputs 1 for all values under the line g_2 (which represents $x_1 \to x_2$).

We can then combine the outputs of the first TLUs y_1 and y_2 into the third (intersection), represented in the second graph.
This transforms the "stripe" from before into a plane, **identifying the linear separability**.

So we can see that with points a and c the final output is 1, while for b and d the solicitation is not enough.

1.5 Arbitrary boolean functions

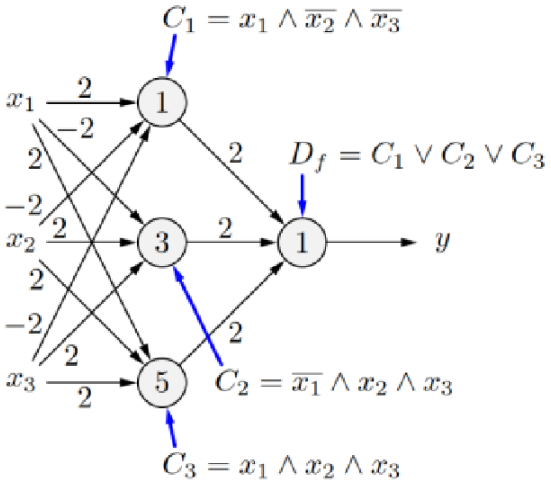
Following the previous reasoning, we can **work with any arbitrary boolean function** by having a **network of TLUs**.

For example, we have these values of y:

x_1	x_2	x_3	y	C_j
0	0	0	0	
1	0	0	1	$x_1 \wedge \overline{x_2} \wedge \overline{x_3}$
0	1	0	0	
1	1	0	0	
0	0	1	0	
1	0	1	0	
0	1	1	1	$\overline{x_1} \wedge x_2 \wedge x_3$
1	1	1	1	$x_1 \wedge x_2 \wedge x_3$

We can now use a network to **compute each of the component for which the output has to be 1 and then put together** all possible values with a conjunction (OR on all values).

The resulting network is



1.6 Training TLUs

TLUs can solve linearly separated problems, but how can we get the right values for the parameters? We want to automatically understand these values.
The geometric interpretation can provide a way to construct TLUs but it has some problems

- Not feasible for more than 3 inputs (we, sadly, live in a three-dimensional space)
- Not an automatic method, there's some human visualization needed to choose the values of the parameters and this process can't be mimicked directly by a machine

What we want to do is to have an automatic way which adjusts the weights and threshold of the network to reach the desired solution (if the two sets are linearly separable).

To have an **automatic training** we can start with **random values** for weights and threshold and determine the **error of the output** (how wrong the output is, based on the correct solution, which we know). We must define the **error** as a **function**, based on **weights and threshold**

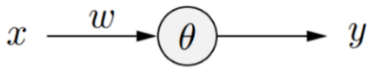
$$e = e(w_1, \ldots, w_n, \theta)$$

We start with random values but we **adapt them** to make the **error smaller** and repeat this step until the error vanishes.

The basic concept is: get random values and change them step by step until they are correct, i.e. the error function is zero.

1.6.1 Negation example

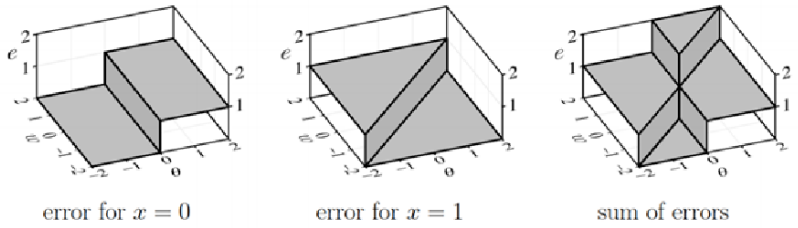
Considering a single input threshold logic unit for the negation $\neg x$



x	y
0	1
1	0

We have one input, one weight, one threshold.

Output error as a function of weight and threshold



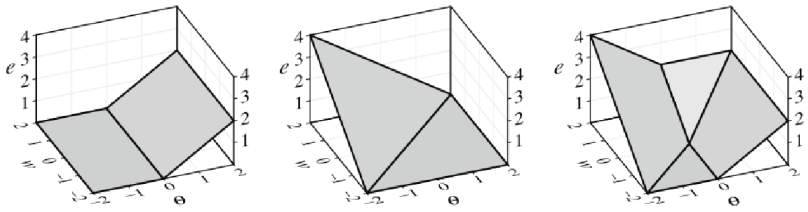
This represents the error for all possible weights and threshold for each value of x.

By intersecting the graphs we can see the region for which the error function is zero for both values of x, and thus the wanted parameters. This is still a graphical way and not automated, it can't be directly mimicked by a machine.

To automate this we would need to read the shape of the error function in each point, since we need to know in which direction to go in order to obtain the correct result. With the error function as defined this is impossible since it consists of **plateaus** (all areas are flat).
If we're in a flat spot how can we determine which way to go if we want to go downwards? (we can't, it's not differentiable).

We need to **modify the error function** to make it differentiable: if the computed output is wrong, take into account **how far from the weighted sum is from the threshold**. The error function tells us how wrong the result is, not only if it's right or wrong.

With the modified error function we obtain

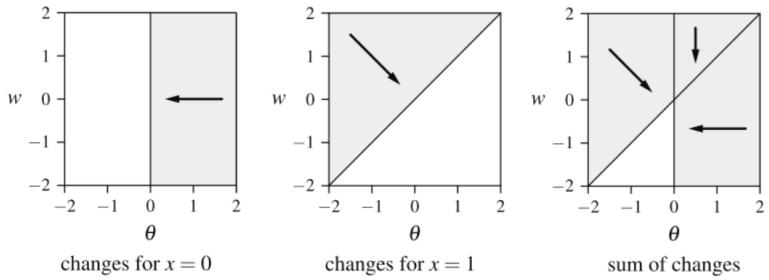


error for $x = 0$ | error for $x = 1$ | sum of errors

Now if a TLU produces a wrong output we can adapt the weight and threshold in such a way to reduce the error, we want to **descend in the error landscape**. We simply **move in the direction in which the error function has the strongest downward slope**.

The parameter changes start with a random point in the error function and the values need to be iteratively adapted **according to the direction corresponding to the current point**. This means that if the output is 1 instead of 0 the threshold is too small and/or the weights are too large, hence they must be tuned accordingly (reducing the weights makes sense only if the corresponding input is 1). The contrary holds for the case in which the output is 0 when it should be 1.

This is a flattened version of the above image, where the dark parts represent values of error over zero; moving in the direction of the arrow decreases the error, until vanishing.



changes for $x = 0$ | changes for $x = 1$ | sum of changes

1.6.2 Types of learning

We can have two different types of training a neural network, differentiated by **how many learning patterns are applied at a time**; they are:

- **Online learning:** we consider one learning pattern at a time, and change the weight accordingly. In the earlier example we may consider the input $x = 0$, adapt the weights accordingly, do the same for $x = 1$ and repeat until the error vanishes. With every example available a training step is carried out.

- **Batch learning:** we collect a sequence of learning patterns during a learning epoch and then compute the cumulative parameter corrections to be applied for the entire set of learning patterns. After all the examples have been explored the aggregated changes are applied. This is also repeated until the error vanishes.

Basically the difference is that in the online learning we give one example at a time (with the relative adaptations), while batch learning applies the changes only after a number of examples.

Batch learning can avoid wasting resources with parallel processing of the examples, if all the samples in the dataset are already known.

If we want something that adapts while new data is acquired online learning might be better.

1.6.3 Delta Rule (Widrow-Hoff)

It's a **training rule**, how do we determine the adaptation of the weights to minimize errors?

Given

- $\vec{x} = (x_1, \ldots, x_n)^T$ as an **input vector** of a TLU
- o as the **desired output** of the vector
- y as the **actual output** of the TLU

We have an output y and a desired value o and the **difference between them tells us how to adapt the values**.

If $y \neq o$, then the threshold θ and the weight vector $\vec{w} = (w_1, \ldots, w_n)^T$ are **adapted as follows** to **reduce the error**:

$$\theta^{(new)} = \theta^{(old)} + \Delta\theta, \text{ with } \Delta\theta = -\eta(o - y)$$
$$w_i^{(new)} = w_i^{(old)} + \Delta w_i, \text{ with } \Delta w_i = \eta(o - y)x_i$$
$$\forall i \in \{1, \ldots, n\}$$

Where η is the **learning rate**, i.e. how much do we value new inputs, it determines the "speed" and size of the updates, 1 at the most, usually small in the range of 2^{-4} to avoid excessive oscillation.

It's basically old value + delta multiplied by a factor η (and the input for the weights).

Example for online training with starting $\theta = 1.5$, $w = 2$ and $\eta = 1$

epoch	x	o	$\vec{x}\vec{w}$	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	−1.5	0	1	−1	0	0.5	2
	1	0	1.5	1	−1	1	−1	1.5	1
2	0	1	−1.5	0	1	−1	0	0.5	1
	1	0	0.5	1	−1	1	−1	1.5	0
3	0	1	−1.5	0	1	−1	0	0.5	0
	1	0	0.5	0	0	0	0	0.5	0
4	0	1	−0.5	0	1	−1	0	−0.5	0
	1	0	0.5	1	−1	1	−1	0.5	−1
5	0	1	−0.5	0	1	−1	0	−0.5	−1
	1	0	−0.5	0	0	0	0	−0.5	−1
6	0	1	0.5	1	0	0	0	−0.5	−1
	1	0	−0.5	0	0	0	0	−0.5	−1

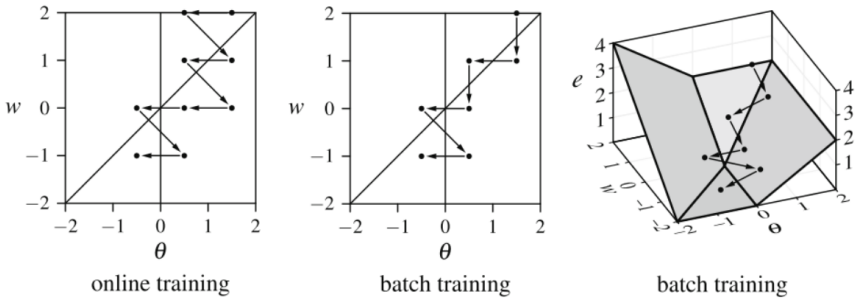
Example for batch training with the same starting values

epoch	x	o	$\vec{x}\vec{w}$	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	−1.5	0	1	−1	0		
	1	0	0.5	1	−1	1	−1	1.5	1
2	0	1	−1.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	0.5	1
3	0	1	−0.5	0	1	−1	0		
	1	0	0.5	1	−1	1	−1	0.5	0
4	0	1	−0.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	−0.5	0
5	0	1	0.5	1	0	0	0		
	1	0	0.5	1	−1	1	−1	0.5	−1
6	0	1	−0.5	0	1	−1	0		
	1	0	−1.5	0	0	0	0	−0.5	−1
7	0	1	0.5	1	0	0	0		
	1	0	−0.5	0	0	0	0	−0.5	−1

The change is delayed, so it takes one epoch more.

We can unify the adaptation rule by turning θ into a weight: fix $\theta = 0$ and add an imaginary input fixed at $x_0 = 1$, this input can be weighted with the negated threshold.

The different progression of online and batch training and a geometric interpretation



We can see that we're descending from a high value of the error function (the starting random value) to a smaller one each time.

1.6.4 Convergence Theorem

Convergence: at some point the error will become 0.

Let $L = \{(\vec{x}_1, o_1), \ldots, (\vec{x}_m, o_m)\}$ be a **set of training patterns**, each **consisting** of an **input vector** $x_i \in \mathbb{R}^n$ (there can be n inputs) and a **desired output** $o_i \in \{0, 1\}$ (out can go one of the two values).

Furthermore, let $L_0 = \{(\vec{x}, o) \in L | o = 0\}$ and $L_1 = \{(\vec{x}, o) \in L | o = 1\}$ (respectively, all the training patterns for which the output should be 0 and all patterns for which it should be 1, you need both to describe all cases).

If L_0 and L_1 are **linearly separable** (all zeros on one side of a line, all the ones on the other), that is, if $\vec{w} \in \mathbb{R}^n$ and $\theta \in \mathbb{R}$ exist such that

$$\forall\, (\vec{x}, 0) \in L_0 : \vec{w}^T \vec{x} < \theta \quad \text{and}$$

$$\forall\, (\vec{x}, 1) \in L_0 : \vec{w}^T \vec{x} \geq \theta$$

then **online as well as batch training terminate**. If the outputs are linearly separable we have convergence.

The algorithm **terminates only when the error vanishes**. The resulting weights and threshold solve the problem.

For **non-linearly separated** problems, the **algorithm doesn't terminate**. It oscillates and repeats computation for the same non-solving $\vec{w}$ and θ.

1.6.5 Encoding Problems

The **parameter correction depends on the encoding of Boolean values**. We use $false = 0$ and $true = 1$ and this may result in less opportunities for correcting the parameters by means of the Delta rule. We cannot adapt a 0, and it's a numerical problem, we can't change the weight corresponding to an input of $false$ since the formula for the weight change contains the input as a factor (anything $\cdot 0 = 0$).

To speed up learning and having more frequent correction opportunities we can adapt a different encoding scheme: **ADALINE** (**ADA**ptive **LIN**ear **E**lement).

ADALINE relies on the **encoding** $true = +1$ **and** $false = -1$.

Basically, in the case of $false$ we don't do anything and we want to do something; we do that by changing the $false$ weight to -1.

1.7 Artificial Neural Network

An Artificial Neural Network (ANN) has a simple **general definition**: it's 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**.

The set of **vertices** (neurons) is **partitioned into**:

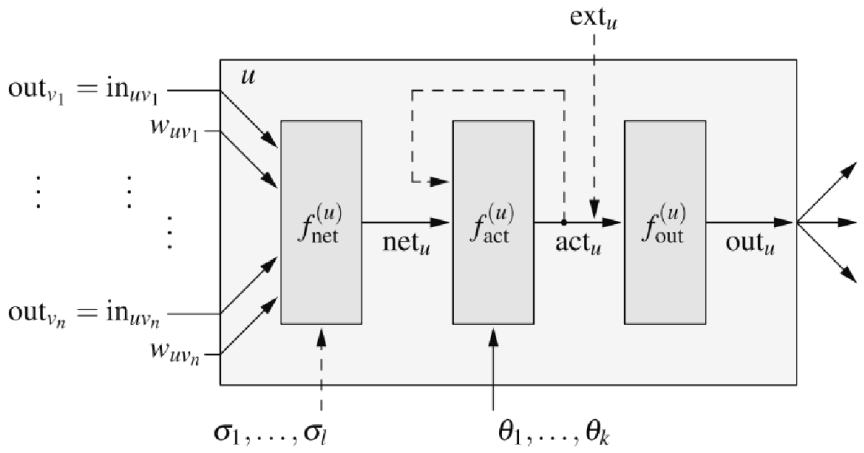
- Set U_{in} of **input** neurons (receive input from the environment).
- Set U_{out} of **output** neurons (emit output to the environment).
- Set U_{hidden} of **hidden** neurons (have no contact with the environment).

The sets U_{in} and U_{out} don't need to be disjoint; a neuron can be both input and output.

1.7.1 General structure of a neuron

Each **connection** (arc from a neuron to the other) $(v, u) \in C$ **possesses a weight** w_{uv}.

Here we can see an example of structure of a neuron:



We can see that there are inputs coming in the network, which represent the stimulus that the neuron receives from the external world, here called in_{uv_n}, each of them with the associated weight w_{uv_n}.

Each input signal can be the output of a previous neuron, the output out_{v_n} for neuron v_n becomes in_{uv_n}.

The **inputs** are **processed in three stages**:

- **Input function** $f_{net}^{(u)}$: from inputs to net_u, takes all the **inputs** and **combines them** according to their value and relative relevance (weight), to **generate the global solicitation** of the neuron.

- **Activation function** $f_{act}^{(u)}$: **analyzes** the global **input** and **generates the activation status** of the neuron. It tells us if the neuron is sufficiently excited. Usually non-linear, the non-linearity gives the neuron its processing capability.

- **Output function** $f_{net}^{(u)}$: takes the excitation status of the neuron and **elaborates the final status** to deliver to subsequent neurons.

Each **neuron** $u \in U$ possesses three (real-valued) **state variables**:

- the **neuron input** net_u: value going in the neuron, usually the sum of all weighted inputs
- the **activation** act_u: the result of the neuron applying a transformation on the input
- the **output** out_u: the final output, after processing the result of the activation

There's also a fourth state variable:

- the **external input** ext_u: it lets us add another variable after the activation phase, if needed

1.8 Types of Artificial Neural Network

There are **two types of ANN**:

- **Feed-forward network (FNN):** its graph doesn't contain any cycle.
- **Recurrent network (RNN):** the graph contains cycles (backwards connections).

The **operation of an ANN** can be divided in:

1. **Input phase:** external **inputs** are **acquired** by input neurons. Inputs are fed into the network
2. **Work phase:** external inputs are switched off while **new outputs are computed** by each neuron. It's the phase in which the output of the network is computed.

If a neuron does not receive any network input, because it doesn't have any predecessor, it simply maintains its activation (and thus its output).

During the working phase, if the input values are steady the computation doesn't change (as in the FNN).
That is not always the case, for example in a RNN the cycles could modify the input of another neuron which has already finished computation, possibly changing its output status.

Re-computation of a neuron output occurs if any of its input change. Temporal order of re-computation depends on the type of ANN.
This phase continues until the eternal **output are steady**, or a **maximum number of** re-computation **iterations** is reached.

All neurons of a network may **recompute** their outputs at the **same time** (**synchronous update**) drawing on the old outputs of their predecessor, or we can also **define an order** in which the neurons **compute their new output** one after the other (**asynchronous update**).

1.8.1 Feed Forward Neural Network

1. **Computation** proceeds **from input** neurons progressively **toward output neurons** by following the topological order of the neuron (left to right basically) in the network.

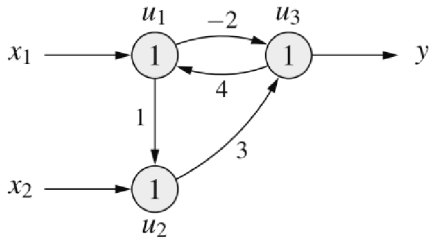
2. The **external inputs are frozen**.

3. **Input neurons compute their outputs** which are maintained steady and **forwarded** to the connected neurons.

4. **Neurons connected** to preceding neurons with steady outputs **generate their respective outputs** and **propagate them** forward to the subsequent neurons, **until the external outputs are generated**.

1.8.2 Recurrent Neural Network

Example of a simple recurrent neural network



	u_1	u_2	u_3	
input phase	**1**	**0**	**0**	
work phase	1	0	**0**	$\text{net}_{u_3} = -2$
	0	0	0	$\text{net}_{u_1} = 0$
	0	**0**	0	$\text{net}_{u_2} = 0$
	0	0	**0**	$\text{net}_{u_3} = 0$
	0	0	0	$\text{net}_{u_1} = 0$

The order in which the neurons are updated is u_3, u_1, u_2. On the right the dataset assumes this ordering with inputs $x_1 = 1$ and $x_2 = 0$. The updates of the neurons are represented in bold.

The phases are:

- **Input phase:** initial activation/outputs. In the input phase the values of u_1 and u_2 are determined by the external inputs, while u_3 is initialized to the (arbitrarily chosen) value 0.
- **Work phase:** activation/outputs of the next neuron to update are computed from the previous outputs and the weights/threshold. Note that external outputs are no longer available as they have been switched off.
- **Stable state:** a unique output is reached.

If we change the order of the updates to u_3, u_2, u_1

	u_1	u_2	u_3	
input phase	**1**	**0**	**0**	
work phase	1	0	**0**	$\text{net}_{u_3} = -2 < 1$
	1	**1**	0	$\text{net}_{u_2} = 1 \geq 1$
	0	1	0	$\text{net}_{u_1} = 0 < 1$
	0	1	**1**	$\text{net}_{u_3} = 3 \geq 1$
	0	**0**	1	$\text{net}_{u_2} = 0 < 1$
	1	0	1	$\text{net}_{u_1} = 4 \geq 1$
	1	0	**0**	$\text{net}_{u_3} = -2 < 1$

The behavior changes completely. As the steps proceed it becomes clear that the seventh step is identical to the first and the computation will repeat indefinitely. The output of the NN depends on the step after which the work phase is terminated. **No stable state**, oscillation of output.

1.9 Configuration of a Neural Network

The structure of the network can be divided in two categories, based on the learning procedure (how the data is given):

- **Fixed learning task** (supervised learning)
- **Free learning task** (unsupervised learning)

Training a neural network consists in adapting the connection weights and possibly some other parameter (like thresholds) such that a certain criterion is optimized.

1.9.1 Fixed learning task

A **fixed learning task** L_{fixed} for a neural network with

- n **input neurons** $U_{in} = \{u_1, \ldots, u_n\}$
- m **output neurons** $U_{out} = \{v_1, \ldots, v_m\}$

is a **set of training patterns** $l = \left(\vec{i}^{(l)}, \vec{o}^{(l)}\right)$ each consisting of

- an **input vector** $\vec{i}^{(l)} = \left(ext_{u_1}^{(l)}, \ldots, ext_{u_n}^{(l)}\right)$
- an **output vector** $\vec{o}^{(l)} = \left(o_{v_1}^{(l)}, \ldots, o_{v_m}^{(l)}\right)$

This is also called **supervised learning**.
It basically consists in giving the NN the inputs and their relative outputs from which to learn. We give the output vector which the network should be able to generate at the end of the configuration.

A fixed learning task is solved when for all training patterns $l \in L_{fixed}$ the neural network computes, from the external inputs contained in the input vector $\vec{i}^{(l)}$ of a training pattern l, the outputs contained in the corresponding output vector $\vec{o}^{(l)}$. Basically it's solved if it gives the right answer for all training inputs we give it.

Error of a fixed learning task: how well a neural network solves a given fixed learning task. Essentially it's the **difference between desired and actual outputs**

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

$$\text{where} \quad e_v^{(l)} = \left(o_v^{(l)} - out_v^{(l)}\right)^2$$

To get a number that reflects the final quality of the network we consider the difference between the actual and desired output. The square is there to avoid negative signs; it also weighs more severely large deviations from the desired output (mean square error).
Basically, the error is the sum of all output discrepancies across all instances. In practice, the optimum can rarely be achieved and thus one may have to accept a partial or approximate solution.

1.9.2 Free learning task

A **free learning task** L_{free} for a neural network with:

- n **input neurons** $U_{in} = \{u_1, \ldots, u_n\}$

is a **set of training patterns** $l = \left(\vec{i}^{(l)}\right)$ each consisting of

- an **input vector** $\vec{i}^{(l)} = \left(ext_{u_1}^{(l)}, \ldots, ext_{u_n}^{(l)}\right)$

There is no output vector. It's also called **unsupervised learning**.
The desired behavior is not defined a priori, it's up to the learning algorithm to produce an output vector.

There is **no desired output** for the training patters. Outputs can be chosen freely by the training method.

Solution: **similar inputs** should lead to **similar outputs**, the learning should lead to the **clustering** of similar input vectors so that the same output is produced for all vectors in the same cluster.

An important aspect for this type of training is **how the similarity** between patterns **is measured**, for example, with the help of a **distance function**.

Preprocessing: to give all neurons the same importance we need to preprocess the data. For each component of the input vector we need to compress the representation in the same range, we need to **normalize the input vector** (e.g. z-score normalization)

$$\mu_k = \frac{1}{|L|} \sum_{l \in L} ext_{u_k}^{(l)}$$

we also need to be sure that the standard deviation σ_k is adjusted as well

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

This is the standard normalization of the deviation σ_k, there's another possibility which is called **unbiased standard deviation** (often preferred by

statistician since is not polarized on the two direction).

The normalization can be necessary to avoid certain numerical problems which can result from an unequal scaling of the different input variables.

All the stimuli are recomputed with respect to the average value and they are normalized in dimension dividing them by the standard deviation. The input vector is normalized to expected value (arithmetic mean) equal to 0 and the standard deviation equal to 1

$$ext_{u_k}^{(l)(new)} = \frac{ext_{u_k}^{(l)(old)} - \mu_k}{\sigma_k}$$

Subtract the mean and divide by 1, all features get kinda in the range from -1 to 1.

Data representation: we need to represent different type of data:

- Numeric data:
  - Real numbers
  - Integer numbers

- Non-numeric (symbolic) data:
  - Symbols, 1-in-N encoding

Simply numbering the different values of attributes can lead to undesired effects if said numbers don't reflect a natural order of the values, so a better option is **1-in-N encoding** in which each nominal attribute is assigned as many (input or output) neurons as it has values: each neuron corresponds to one attribute value.

With the input of a training pattern, the neuron that corresponds to the value of the nominal attribute is set to 1, while all others are set to 0. Only 1 in n neurons (where n is the number of attributes values) is set to 1, the others to 0 (explains the name).
Basically a string of n bits in which only the bit that represent our symbol is raised.

1.10 Multi-layer Perceptrons MLP

An r-**layered perceptron** is a feed-forward neural network with a **strictly layered structure**.

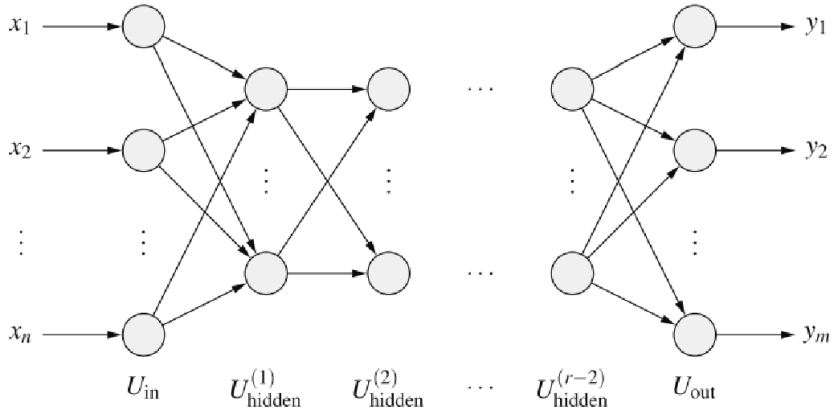
Each layer receives inputs only from the previous one (or the external world) and delivers its output to the subsequent layer; jumps are not allowed. Connections exist only between neurons of consecutive layers. Usually a neuron is fully connected to the neurons of its adjacent layers.

It's a common structure and can have significant processing capabilities, since you can increase them by simply adding more neurons/layers.

There are different kinds of layers:

- **input** layers
- **hidden** layers
- **output** layers

General structure for an r-layered perceptron:



The **network input function** of each hidden neuron and of each output neuron is the **weighted sum of its inputs**

$$f_{net}^{(u)}\left(\vec{w}_u, \vec{in}_u\right) = \vec{w}_u^T \vec{in}_u = \sum_{v \in pred(u)} w_{uv} out_v$$

Basically just each input times their weight, for each neuron.

The **activation function** of each **hidden neuron** is a so-called **sigmoid function**, a monotonically non-decreasing function; non-linear (to add complexity), usually with bounded output (e.g. from -1 to 1), continuous and differentiable

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

(if the bounds are $[-1, 1]$).

Depending on what we want to achieve, the **activation function** of each **output neuron** can also be a **sigmoid or** a **linear** function; basically the output neurons can take all the inputs and throw them outside (linearly), they don't elaborate further.

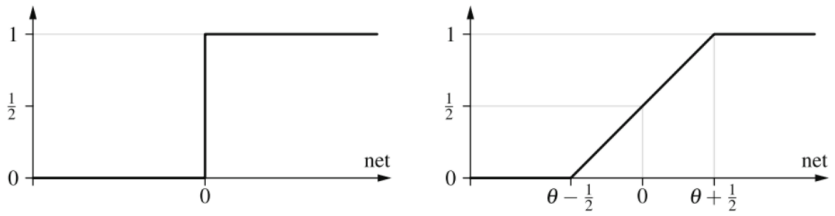
Some examples of sigmoid activation functions:

- Step function (left):

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

- Semi-linear function (right):

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

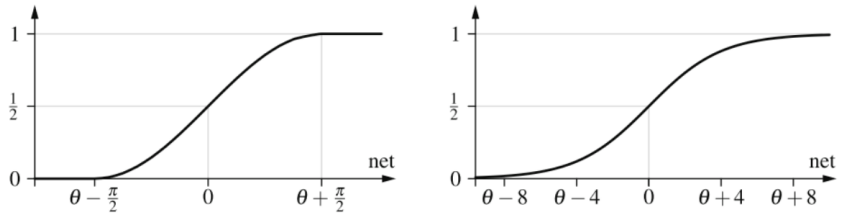


- Sine until saturation (left):

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

- Logistic function (right):

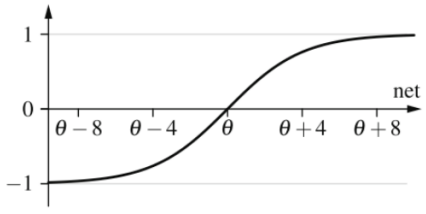
$$f_{act}(net, \theta) = \frac{1}{1 + e^{-(net-\theta)}}$$



These are some **unipolar** sigmoid activation functions.

Sometimes **bipolar sigmoid functions** are used (ranging from -1 to $+1$, two poles, not only one), like the **hyperbolic tangent**, this allows a better response to stimuli, in contrast to something like the step function which can only be 0 or 1

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



Having non-linear functions allows increasing the computational capability. If all neurons had linear activation functions, having multiple hidden would be useless, you could just combine all the weights into one. With non-linear functions the more hidden layers, the more computational capabilities.

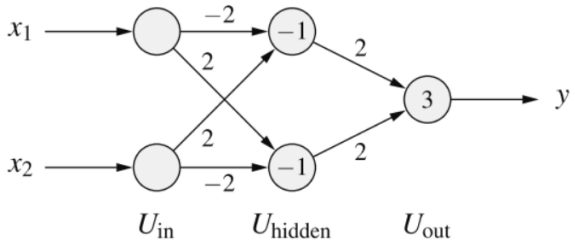
Let $U_1 = \{v_1, \ldots, v_m\}$ and $U_2 = \{u_1, \ldots, u_n\}$ be the **neurons** of two **consecutive layers** of an MLP. The group of **connections between these layers** can be described by using an $n \times m$ **matrix**. The collection of the weights between the layers is:

$$W = \begin{pmatrix} w_{u_1 v_1} & w_{u_1 v_2} & \ldots & w_{u_1 v_m} \\ w_{u_2 v_1} & w_{u_2 v_2} & \ldots & w_{u_2 v_m} \\ \ldots & & & \\ w_{u_n v_1} & w_{u_n v_2} & \ldots & w_{u_n v_m} \end{pmatrix}$$

The computation of the network input can be written as

$$\vec{net}_{U_2} = \boldsymbol{W}\vec{in}_{U_2} = \boldsymbol{W}\vec{out}_{U_1}$$

An **example with the bi-implication** three-layer perceptron:



Weight matrices for input and hidden layer:

$$\boldsymbol{W}_1 = \begin{pmatrix} -2 & 2 \\ 2 & -2 \end{pmatrix} \quad \text{and} \quad \boldsymbol{W}_2 = \begin{pmatrix} 2 & 2 \end{pmatrix}$$

The difference between the NN and the TLU for bi-implication is that in the former we have to divide the structure into input, hidden and output layer, so we need to add a dedicated input layer.

1.10.1 Function approximation

So far we've considered and represented only boolean functions

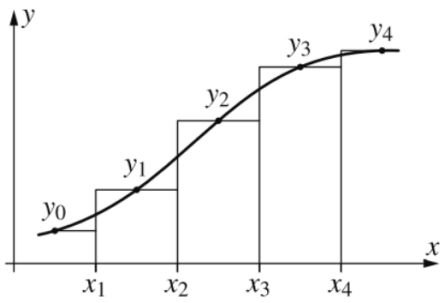
$$f : \{0,1\}^n \to \{0,1\}$$

We now want to be able to use any real number inside an MLP, so we start representing and learning **real-valued functions**

$$f : \mathbb{R}^n \to \mathbb{R}$$

We need to demonstrate that **all Riemann-integrable functions can be approximated by four layer perceptrons** with arbitrary accuracy, provided that the output neuron has the identity, instead of a step function, as its activation function.

Considering this example

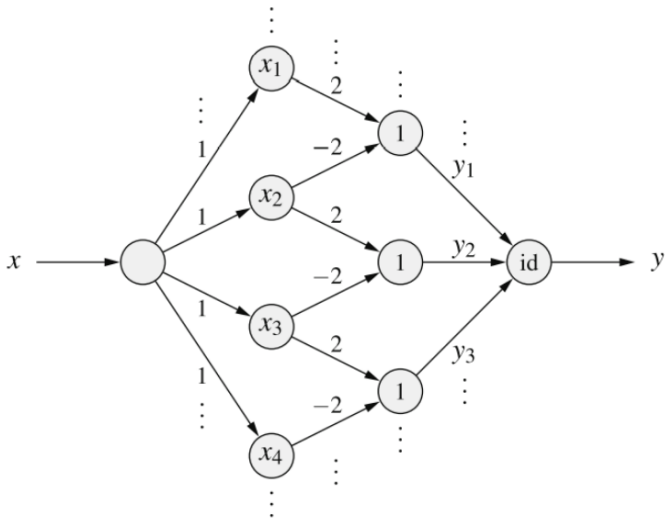


We want to consider points $x_1, \ldots, x_4$, where we compute the value of the function, and we want to find a way to approximate a value in that function for any other x.

We can consider the **midpoint Riemann integration**:

- Approximate a given function by a step function
- Construct a neural network that computes said function
- Error is measured as the area between functions

This is a NN that computes the step function



The **input** is taken by an input neuron, then the **first hidden layer** is composed of a neuron for each of the x_i step borders needed for our approximation. Each of these neuron is able to determine on which side of the step border an input value lies.

In the **second hidden layer** we create a neuron for each step, which receives input from the two neurons in the first hidden layer that refer to the values x_i and x_{i+1} marking the border of the step. The neuron is activated only if the input value is less than x_{i+1} and more than x_i, so there can be only one neuron active in the second layer; determines if the value is in the range of the step.

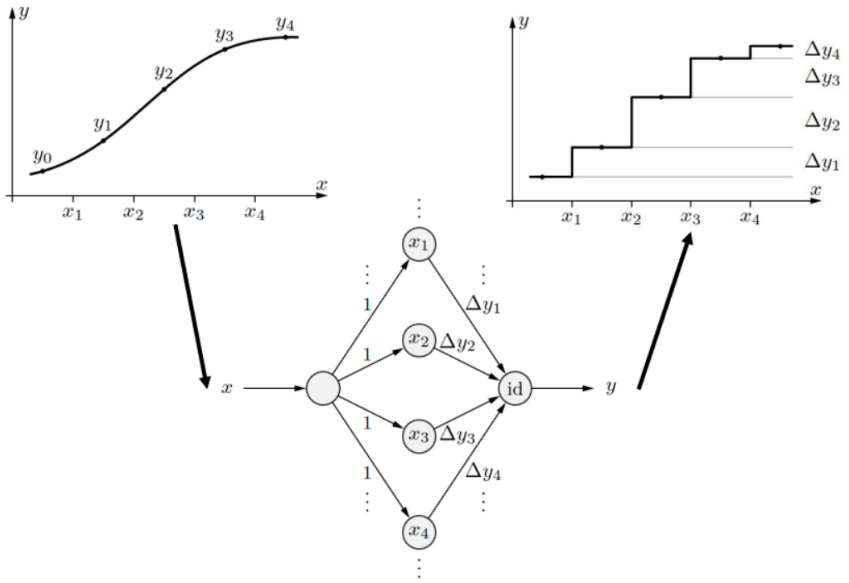
The **connections** from the neurons of the second hidden layer to the output are **weighted** with the **function values of the stair step** represented by the neurons.

The **output layer** receives as input only the height of the stair step in which the input value lies.

The **accuracy can be increased arbitrarily** by **increasing the number of steps**.

1.10.2 Delta approximation approach

We can simplify this method by **considering the Δ variation of the function value** between each step.



This way we need only **three layers**:

- Input
- Hidden layer with a neuron for each of the x_i points considered, each of these neurons determines if the input value is greater than x_i (or not)
- Output

The **connections** between input and hidden layer are **weighted with the relative difference between each step**, so for input x, all neurons with a value lesser than x will be active and "contribute" towards the output value.

1.11 Regression

The training of neural networks is **closely related to regression**, which basically is finding the statistical model that links two variables, extrapolating a straight line that approximates the existing relationship inside a dataset.

All neural networks have the purpose of **minimizing the mean square error**, just like regression tries to do. We can accomplish that by adapting the weights and parameters of the activation function. This leads to the **method of least squares**, also known as **regression**, which is used to find the **best fit polynomials for a given set of data points**.

It will generally not be possible to find a straight line that covers exactly all n points of the data set, so we have to try and find a line that deviates from these measurements as little as possible; the sum of the squared differences is minimized.

1.11.1 Linear regression

If we expect that our quantities x and y exhibit a linear dependence, then we have to identify the parameters a and b that extrapolate the line $y = g(x) = a + bx$ with as little deviation as possible.

Given

- a **dataset** $((x_1, y_1), \ldots, (x_n, y_n))$ of n data tuples
- a **hypothesis about the functional relationship**, e.g.

$$y = g(x) = a + bx$$

The approach is to **minimize the sum of squared errors**

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

The **necessary condition to find the minimum** is having the **partial derivatives** (of the error function in respect to a and b) **equal to 0** (*Fermat's theorem*)

$$\frac{\partial F}{\partial a} = \sum_{i=1}^{n} 2\left(a + bx_i - y_i\right) = 0$$

$$\frac{\partial F}{\partial b} = \sum_{i=1}^{n} 2\left(a + bx_i - y_i\right) x_i = 0$$

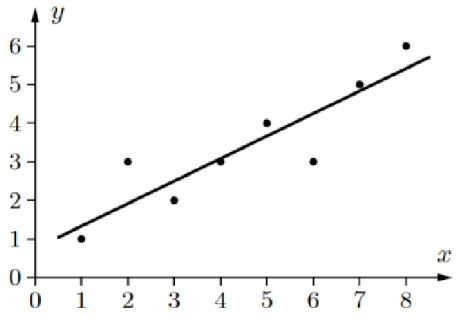
The system **can be solved** with standard methods from linear algebra and the **solution is unique**, unless all values of x coincide (trivial, the data would be useless), in which case there are infinite solutions.

Example: starting from these points

x	1	2	3	4	5	6	7	8
y	1	3	2	3	4	3	5	6

The regression line would become

$$y = \frac{3}{4} + \frac{7}{12}x$$



1.11.2 Polynomial regression

The previous method can be extended to **polynomial of arbitrary order**. Instead of considering a grade equal to one, we consider a polynomial of grade m:

$$y = p(x) = a_0 + a_1 x + \ ... \ + a_m x^m$$

The **minimization** for the sum of squared **errors** can be generalized into

$$F(a_0, a_1, \ ..., a_m) = \sum_{i=1}^{n} \left(p(x_i) - y_i\right)^2 = \sum_{i=1}^{n} (a_0 + a_1 x_1 + \ ... \ + a_m x_i^m - y_i)^2$$

The necessary **conditions for a minimum** are that **all partial derivatives vanish**

$$\frac{\partial F}{\partial a_0} = 0, \ \frac{\partial F}{\partial a_1} = 0, \ \ ... \ , \frac{\partial F}{\partial a_m} = 0$$

It can be solved with standard methods just like before, the **solution is** also **unique**, unless the points lie exactly on a polynomial of lower degree (another trivial case).

1.11.3 Multi-linear regression

We can **generalize** the previous methods to **more than one argument**

$$z = f(x, y) = a + bx + cy$$

We're applying linear regression to multiple variables, the concept is the same as before but multi-linear.

We need to **minimize the sum of squared errors**, the difference between the function f and the sample values z_i

$$F(a, b, c) = \sum_{i=1}^{n} \left(f(x_i, y_i) - z_i\right)^2 = \sum_{i=1}^{n} \left(a + bx_i + cy_i - z_i\right)^2$$

Like before, for a **minimum** we need **all partial derivatives** to **vanish**

$$\begin{aligned}
\frac{\partial F}{\partial a} &= \sum_{i=1}^{n} 2\left(a + bx_i + cy_i - z_i\right) &= 0, \\
\frac{\partial F}{\partial b} &= \sum_{i=1}^{n} 2\left(a + bx_i + cy_i - z_i\right) x_i &= 0, \\
\frac{\partial F}{\partial c} &= \sum_{i=1}^{n} 2\left(a + bx_i + cy_i - z_i\right) y_i &= 0
\end{aligned}$$

As before, **can be solved** and **solution is unique** unless all data points lie on a straight line (there's a plane that can be rotated in any direction and thus infinite solutions).

We can also have a **general multi-linear case** with a **function** defined on m **variables** (there aren't enough dimensions to draw this)

$$y = f(x_1, \ldots, x_m) = a_0 + \sum_{k=1}^{m} a_k x_k$$

Minimize the **sum of squared errors**

$$F(\vec{a}) = (\boldsymbol{X}\vec{a} - \vec{y})^\top (\boldsymbol{X}\vec{a} - \vec{y})$$

Where

$$\boldsymbol{X} = \begin{pmatrix} 1 & x_{11} & \ldots & x_{m1} \\ \ldots & & & \\ 1 & x_{1n} & \ldots & x_{mn} \end{pmatrix}, \quad \vec{y} = \begin{pmatrix} y_1 \\ \ldots \\ y_n \end{pmatrix}, \quad \vec{a} = \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_m \end{pmatrix}$$

We have vectors for the parameters and matrices to define the values.

Necessary conditions for a minimum

$$\vec{\nabla}_{\vec{a}} F(\vec{a}) = \vec{\nabla}_{\vec{a}} (\boldsymbol{X}\vec{a} - \vec{y})^\top (\boldsymbol{X}\vec{a} - \vec{y}) = \vec{0}$$

I need to define the **gradient**, generalization of the partial derivatives on vectors. We need to find where the gradient is 0, which will give us the hyperplane touching the surface in the minimum (kinda like before but not a line, it's a plane in fuck-knows how many dimensions, I guess).

This comes out to a set of regular equations

$$\boldsymbol{X}^\top \boldsymbol{X}\vec{a} = \boldsymbol{X}^\top \vec{y}$$

which has a solution, unless $\boldsymbol{X}^\top \boldsymbol{X}$ is singular

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

This last subsubsection has been discussed very roughly, no need to understand the math exactly (I hope).

1.11.4 Logistic regression

It's reasonable to assume that Neural Network want to use non-linear, and in general non-polynomial functions, so we want to be able to approximate a dataset with **different types of functions**, for example

$$y = ax^b$$

We can find a **transformation to a linear/polynomial case**

$$\ln y = \ln a + b \ln x$$

In the case of an ANN we are interested in the **logistic function**

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

A lot of ANNs use the logistic function as activation, so being able to do linear regression on that would mean being able to effectively determine any parameter of any network.

We can derive from this equation, using a logarithm, a **linear description**; we go from a non-polynomial function to a linear one, to be able to minimize the sum of squared errors. The value a of the function represents the threshold of the output neuron, while the value of b represent the weight of the input.

We can linearize the logistic function by way of the **Logit-transformation**

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

Example:

x	1	2	3	4	5
y	0.4	1.0	3.0	5.0	5.6

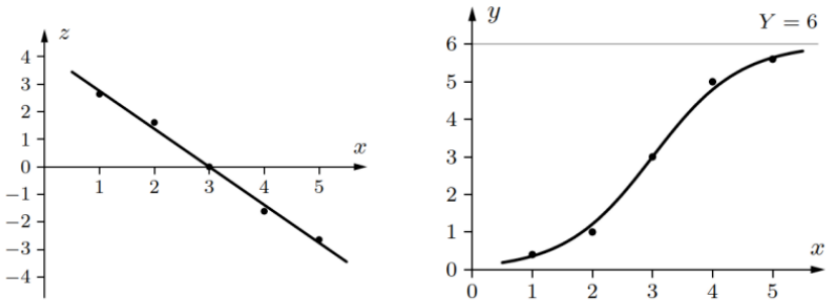
$$z = \ln\left(\frac{Y-y}{y}\right), \quad Y = 6$$

x	1	2	3	4	5
z	2.64	1.61	0.00	-1.61	-2.64

The results of the regression line in the example are:

$$z \approx -1.3775x + 4.133 \quad y \approx \frac{6}{1 + e^{-1.3775x} + 4.133}$$

Plot of logistic functions:



The logistic regression function can be computed by a single neuron.

If we can find a method to determine a logistic regression function, we immediately possess a method to determine the parameters of a two-layered perceptron with a single input.

As before, the initial concepts are important, the exact math and consequences are not (I hope, they're being explained like they're not).

1.11.5 Two-class problems

Going from the regression part to the classification (binary, in this case). The idea is to **model the conditional probability of one class as a logistic function**. I have a function, the two classes are divided by the line (value over the function are class 0, under are class 1).

Let C be a **class of attributes** and $\text{dom}(C) = \{c_1, c_2\}$ and $\vec{X}$ a **random vector** in m dimensions.

Given a set of **data points** $X = \{\vec{x}_1, \ldots, \vec{x}_n\}$ each of which belongs to one of the two classes c_1 and c_2.

We consider the **probability** of belonging to the first class c_1 and the probability to be in the second class c_2

$$P(C = c_1 | \vec{X} = \vec{x}) = p(\vec{x})$$

$$P(C = c_2 | \vec{X} = \vec{x}) = 1 - p(\vec{x})$$

We want to obtain a **simple description of the function** $p(\vec{x})$, we can describe it by a logistic function

$$p(\vec{x}) = \frac{1}{1 + e^{a_0 + \vec{a}\vec{x}}} = \frac{1}{1 + \exp\left(a_0 + \sum_{i=1}^{m} a_i x_i\right)}$$

Then we can apply the **logit transformation** so that we can obtain the formal description of the **probability of being on one or the other class**:

$$\ln\left(\frac{1 - p(\vec{x})}{p(\vec{x})}\right) = a_0 + \vec{a}\vec{x} = a_0 + \sum_{i=1}^{m} a_i x_i$$

How can we find the specific value of the probability (and thus solving the splitting of the example)? We can consider a specific function called **kernel** that describes how strongly a data point influences the probability estimate for neighboring points. This gives us correlation among elements of the same class.

With a kernel we can describe the similarity of a group of examples, the Gaussian function is commonly used. This function increases as the data point considered approaches the middle (the sample is similar to both groups, it's in the middle, value is high) and decreases the further I go

$$K(\vec{x}, \vec{y}) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(\vec{x}-\vec{y})^\top(\vec{x}-\vec{y})}{2\sigma^2}\right)$$

If we want to estimate the **probability density** the expression is

$$\hat{f}(\vec{x}) = \frac{1}{n}\sum_{i=1}^{n} K(\vec{x}, \vec{x}_i)$$

The kernel estimation applied to a two-class problem:

$$\hat{p}(\vec{x}) = \frac{\sum_{i=1}^{n} c(\vec{x}_i)K(\vec{x}, \vec{x}_i)}{\sum_{i=1}^{n} \vec{x}_i}$$

If it is

$$c(\vec{x}) = 1$$

the x_i will belong to class c_1, to the remaining class otherwise.

This page is NOT found in the lesson, it's from external sources.

This classification model allows us to obtain the parameters via **maximizing the likelihood of the data**, we choose the **parameters** that **make the data most likely** by setting up a **likelihood function** which describes how probable the given data is depending on the parameters.

With this approach, we can apply the classification rule to new data:

$$C = c_1 \text{ if } p(x, a) \geq \theta$$

$$C = c_0 \text{ if } p(x, a) < \theta$$

With $\theta = 0.5$.

We now can, in theory, proceed with a linear regression: a necessary condition for a **minimum of the negative log likelihood is that its gradient vanishes**.

The resulting equation system is not linear and difficult to solve analytically, we need to consider a **gradient descent**.

TLDR(?): We transform the non-polynomial function into a linear case, then we can apply the minimization of the sum of squared errors to get a solution. Then the output of the two classes is obtained by putting a threshold the output, over it's one class, under it's the other. We have x and the corresponding y, but there's an extra step, we need to consider a threshold on the result, over it's one class, under it's the other.

(this is based on what he's saying, this man is cray-cray)

1.12 Training MLPs

1.12.1 Gradient descent

Logistic regression would work, but only for two-layer perceptrons, so the general approach is **gradient descent**.

If we can derive from the error function in which direction we have to change the weights and bias the values in order to reduce the error, we obtain a possibility to train the parameters of the network. The average error across all neurons will determine the overall "quality" of our network.
We make small steps into these directions, check the directions again and repeat the process.

We modify the values of the parameters one step at a time and check if the error is reduced (we're converging). We do this since a purely analytical solution would be hard to find. The requirement is to have **differentiable activation and output function**; this means that, in this case, the error function will be differentiable as well and thus we can determine the directions we need to minimize the error function.

The input that minimizes the cost of our error function is the one that gives $\frac{d}{dw}e(w) = 0$, but this is not really feasible for complicated functions, so we can find a **local minimum** (global minimum is also pretty hard).

This concept has to be translated to a scenario with multiple parameters (we have weights, biases and stuff). If we have two variables the error function can be graphed as a surface along the xy plane; to minimize it we need to ask *in which direction the error function decreases more quickly?* And multi-variable calculus has the concept of **gradient** as an answer, defined as the direction of the steepest increase $\vec{\nabla}e(x, y)$ and the length of the vector determines how steep is the increment.

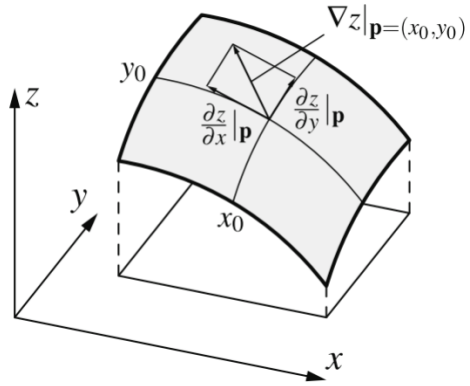
The algorithm will essentially be:

- Compute $\vec{\nabla}e(x, y)$
- Small step in direction $-\vec{\nabla}e(x, y)$
- Repeat until convergence

The negative gradient determines how the weights and biases need to be modified to efficiently decrease the cost. The backpropagation is an algorithm for computing that gradient.

The gradient is a vector expressed in n-dimensions, where n depends on the weights and biases, usually it's a giant number (impossible to visualize). The important thing is the magnitude of each component (so the value) since it will make us able to determine **how sensitive** the error function output will be respect to the weight and bias (how much it moves respect the previous value).

We need to apply this for all the values of the weights of a network. Essentially when we talk about **training** or the process of learning of a NN it's just about **minimizing the error function**. More to that, the negative gradient vector will store the relative importance of the weights for each neuron (how important each neuron is for the total error, this is a value needed for the backpropagation). The gradient of a real-valued function is the set of all partial derivatives. Intuitive interpretation of the gradient of a real-valued function $z = f(x, y)$ at a point (x_0, y_0):



Training becomes very simple: the weights and bias **values are initialized randomly** and then the **gradient** of the error function **is computed** at the point given by these values.

The solution to our learning process is given by **reaching the minimum of the error function**, one small **step** at a time, in the **direction opposite to the gradient**. At the new point we recompute the gradient until a minimum is reached.

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

The **sum of the individual errors over all output neurons** $v \in U_{out}$ **and all training patterns** $l \in L_{fixed}$.

So we need to evaluate the gradient in order to go in the proper direction. In the case of an MLP calculating the gradient means computing the partial derivative of the error function in respect of the threshold and the weights taken as parameters.

Given $\vec{w}_u = (-\theta, w_{u_1}, ..., w_{u_k})$ as the vector of weights of a single layer extended with the threshold, then the **gradient is computed as follows**:

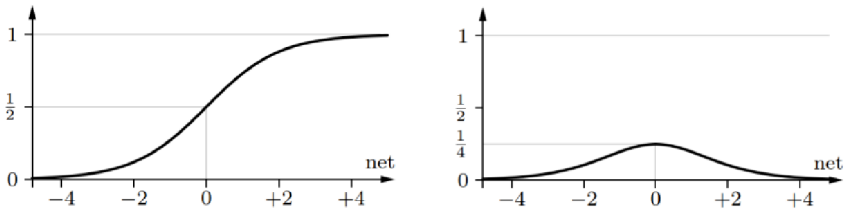
$$\vec{\nabla}_{\vec{w}_u} e = \frac{\partial e}{\partial \vec{w}_u} = \left(-\frac{\partial e}{\partial \theta_u}, \frac{\partial e}{\partial w_{up_1}}, ..., \frac{\partial e}{\partial w_{up_n}} \right)$$

The total error e is given by the sum of individual errors across all neurons and training patterns, so we get:

$$\vec{\nabla}_{\vec{w}_u} e = \frac{\partial e}{\partial \vec{w}_u} = \frac{\partial}{\partial \vec{w}_u} \sum_{l \in L_{fixed}} e^{(l)} = \sum_{l \in L_{fixed}} \frac{\partial e^{(l)}}{\partial \vec{w}_u}$$

Considering the logistic function and its derivative

$$f_{act}(net_u^{(l)}) = \frac{1}{1 + e^{-net_u^{(l)}}} \qquad f'_{net}(net_u^{(l)}) = f_{act}(net_u^{(l)})$$



The **weight changes** (performed on the vector $\vec{w}_u$) are **proportional to the derivative** of function $f'_{act}(net_u^{(l)})$

- Close to 0 weight changes (and thus training speed) are at the highest
- Far from 0 the gradient becomes small and training slow

Gradient descent may be more or less effective depending on the landscape of the error function. We need to find the minimum, if the function is steeper small steps give significant changes, otherwise you'll need more steps.

1.12.2 Error backpropagation

So far, only the output neurons are directly connected to the error, they're the only one that can see "how wrong they are". But we **need to train the whole network**.

For the last layer we can see the error, and thus we can compute the change directly, it's the derivative of the error function with respect to the weight of the neuron.
But what about the layers previous to the last one? We do kinda the same thing, but how much I modified the last layer becomes the error for the second to last one. The partial derivative to that error gives how much we have to modify that layer.
The process can repeat for all layers.

The error values of any (hidden) layer of an MLP can be computed from the error values of its successor. The sum of the errors of the weights of the last layer becomes the error function to minimize for the weights of the layer before that.

The error is computed at the end of the network and then **backpropagated**, we need to modify all the weights in the network. The error signal is transmitted from the output layer backwards.

General structure:

1. Setting the input

$$\forall i \in U_{in} : out_u^{(l)} = ext_u^{(l)}$$

We're just giving the inputs.

2. Forward propagation of the input

$$\forall u \in U_{hidden} \cup U_{out} : out_u^{(l)} = \left(1 + exp\left(-\sum_{p \in pred(u)} w_{up} out_p^{(l)}\right)\right)^{-1}$$

We process the inputs until we get an output, this is the "work phase".

3. Error factor

$$\forall u \in U_{out} : \delta_u^{(l)} = \left(o_i^{(l)} - out_u^{(l)}\right) \lambda_u^{(l)}$$

We compute the error for the last layer (basically just the difference between output obtained and desired).

Weight adaptation

$$\Delta w_{up}^{(l)} = \eta \delta_u^{(l)} out_p^{(l)}$$

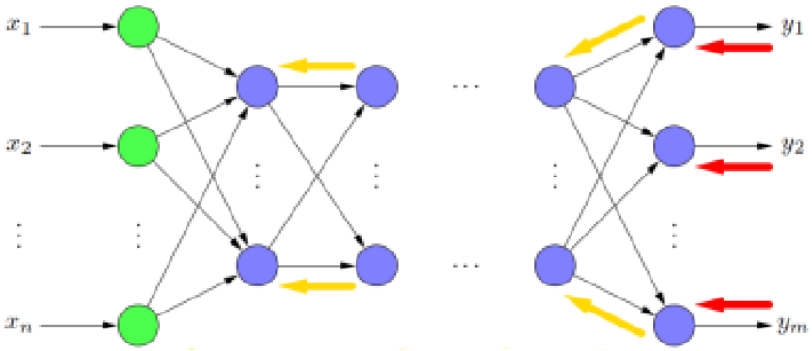
4. Error Backpropagation

$$\forall u \in U_{hidden} : \delta_u^{(l)} = \left(\sum_{s \in succ(u)} \delta_s^{(l)} w_{su} \right) \lambda_u^{(l)}$$

The "new" error factor is computed starting from the error of the subsequent layer.

Derivative of the activation function

$$\forall u \in U_{hidden} \cup U_{out} : \lambda_u^{(l)} = out_u^{(l)} \left(1 - out_u^{(l)}\right)$$



Another explanation (since the last one was shit): We assume that the **activation function a logistic function for each neuron** $u \in U_{hidden} \cup U_{out}$, except for the input neurons.

1. **Apply the input** at the input neurons that is returned without modifications at the subsequent first hidden layer (just give the inputs man).

2. We **compute** for each neuron of the subsequent layers the weighted sum of the inputs, applying at the result our logistic function, **producing the output that will be propagated** throughout the network until the terminal neurons (elaborate the inputs).

3. **Compute the difference between the desired output and the actual output**. Since it's possible to invert the logistic function f_{act}, we can know which was the input (error) that has induced that particular error (δ_u).

4. Now that we have transformed the error of the output variable out_u in the error of the input variable net_u, we can **distribute the error** (with the correction) in a proportional way back to previous neuron, we **backpropagate** the error until the input neurons.

We have to say that given the shape of the logistic function the error can disappear completely, since the gradient will approximate more the **null vector** the more it will be near the zero.

The **weight adaptation** is performed by the following **formula** (this tells how to perform the correction):

$$\forall w_{up} : \delta w_{up} = \eta \delta_u out_p$$

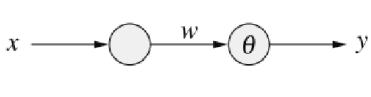
If you initialize the **learning rate** η with too high of a value, instead of descending the curve we risk jumping from a "peak" of the function to another (jumps over the minimum), without ever converging to the minimum. Furthermore, it's NOT certain that the minimum reached in this way is the global minimum of the function.

A **solution** to the problem could consist in **repeating the learning**, initializing the system with a different configuration of weights and threshold,

and then choose at the end which configuration results in the better minimum.

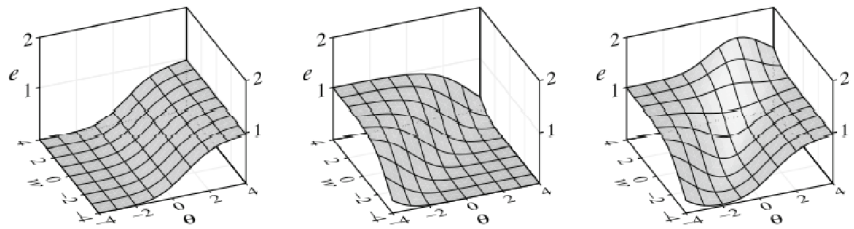
On the contrary, a value too small for η makes the learning too long and takes too many steps. The correct value has to be chosen through trial and error (mostly by vibe I guess).

Negation example: considering the negation $\neg x$, there's a two layer perceptron for said function



x	y
0	1
1	0

By using the logistic activation function we get these squared errors for computing the negation:



error for $x = 0$ | error for $x = 1$ | sum of errors

From a numeric point of view, the error **can't completely vanish due to the properties of the logistic function**, there will always be some residual errors due to the computation of the various parameters. The error keeps getting smaller and smaller but it doesn't vanish completely.

1.12.3 Variants of Gradient Descent

There can be **variants** of the gradient descent:

- *Manhattan training*, considers only the sign of the gradient, fast but might not reach the optimum
- *Flat spot elimination*, the derivative of the action function is lifted by a certain value α so that sufficiently large training steps are taken even in flat spot regions.

- *Momentum term*, at each step a fraction of the preceding weight change is added to a normal gradient descent step.

- *Self-adaptive error backpropagation*, each parameter has a different learning rate, allowing a finer control of the learning process.

- *Resilient error backpropagation*, combination of the Manhattan training with the Self-adaptive error backpropagation.

- *Quick propagation*, instead of using the gradient, approximate the error function to a parabola and jump to the peak.

- *Weight decay*, reduce the weight to prevent getting stuck in the saturation region .

A common combination to avoid overfitting and keeping the absolute value of the weights as little as possible is Momentum term and Weight decay.

1.13 Number of hidden neurons

The **number** of **input neurons** is **dependent on the data** (10 features, 10 neurons), the **number** of **output neurons** depends on the **type of problem** considered (10 classes, 10 neurons).

There's **no fixed rule for the number of neurons in hidden layers**, the process should consist of increasing the number (and thus the capabilities of the network) until there's overfitting (the network is also learning the error in the data).

For a **single hidden layer** the following rule of thumb is popular:

$$\text{Number of hidden neurons} = \frac{\text{number of inputs} + \text{number of outputs}}{2}$$

Underfitting: the number of neurons in the hidden layer is too small, the MLP may not be able to correctly approximate the function (and thus the relation between input and output values) due to a lack of parameters.

Overfitting: the number of neurons in the hidden layer is too big, the MLP learns too much from the examples and the abundance of parameters allows it to fit exactly the training data given, error included, leading to a loss in the ability of generalizing the desired behavior. It's learning the error in the data along the data itself.

1.14 Cross Validation

We need to **evaluate the performance of the network** (is the training good?). The objective is training the model on some data, and it should perform correctly on new data.

To understand the quality of training we split our data in two subsets:

- **Training set:** used for training
- **Validation set:** used for checking the quality of the result

This allows us to see if the NN is overfitting the training data and evaluate the general quality of the learning.

If I increase the neurons and get a lower error on the validation, I'm still underfitting (needs more neurons, more capabilities), while if I increase the capabilities and get a higher error, the NN is overfitting.

The **cross validation** consists in training different MLPs with different numbers of hidden neurons and evaluate them with the two subsets, multiple times with the data randomly split across training and evaluation, averaging the results for each "number of neurons". At the end the number of neurons with the lowest average error is chosen.

1.14.1 N-Fold Cross validation

Another method to evaluate training

- The given **data set is split into** n **parts** or subsets (also **called folds**) of about equal size
- The relative frequency of the output values in the subsets/folds represents as well as possible the relative frequencies of these values in the data set as a whole (**stratification**)
- Out of these n data folds, n **pairs of training and validation data sets are formed** by using one fold as validation and $n-1$ folds for training

This way, **we train the model for each one of the** n **training sets**, avoiding overfitting problems and asymmetric sampling typical of the classic cross validation. In other words the sample is subdivided in group of equal size, one group at time is excluded (this is predicted by using the non excluded groups), in order to evaluate the prediction model used.

Basically, out of n folds, 1 is kept for validation, the other for training and repeat n times, each time with a different validation fold.

The **stopping criterion** is also important, the training continues as long as the error decreases, it stops as soon as the error increases (start of overfitting).

1.14.2 Sensitivity analysis

The great capabilities of NNs also make it difficult to understand exactly what they're doing, the **knowledge learned** is often **encoded in matrices/vectors** of real-valued numbers, often **difficult to understand or extract**. Geometric interpretation are only possible for very simple networks. A NN effectively becomes a *black box*.

We have no real way to know what's inside the network, but we can have a **sensitivity analysis** can be useful to find out **to which inputs the output(s) react(s) most sensitively**; we want to find out what parts of the input influence the output and how. This can give information about which inputs are not needed and may be discarded. Basically we want to answer *"How much the output of each neuron is changing if we change the data set for training (still describing the desired behavior)?"*.

The approach is to **determine the change of output relative to the change of input**:

$$\forall u \in U_{in}: \quad s(u) = \frac{1}{|L_{fixed}|} \sum_{l \in L_{fixed}} \sum_{v \in U_{out}} \frac{\partial out_v^{(l)}}{\partial ext_u^{(l)}}$$

This allows us to have an idea of what the NN is using to make a certain decision, which inputs are more important than others.

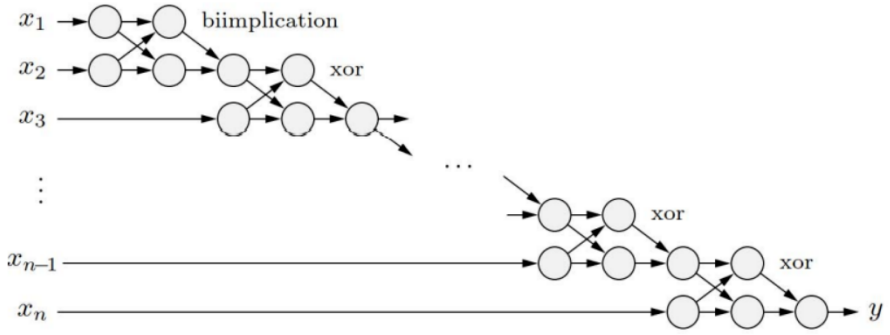
1.15 Deep Learning

"Deep Learning" refers to a NN with **several hidden layers**. With "*depth*" we mean the highest possible number of layers that separate input and output (longest path in the network graph).

An MLP with three layers (one hidden layer) can approximate arbitrarily well any continuous integrable function (**universal approximation theorem**), but depending on the function, a very large quantity of neurons (e.g. exponential in the number of inputs) may be necessary. Allowing for **more hidden layers** may enable to achieve the **same approximation quality with a significantly lower number of neuron**.

Example: considering the n-bit parity function, which outputs 1 only if an even number of inputs is 1, if we want to approximate this function with a single hidden layer we'd need 2^{n-1} neurons, and this number (clearly) **grows exponentially** with the number of inputs.

With multiple hidden layers, **linear growth is possible**:



(this example allows for as many hidden layers as needed).
The disjunctive normal for of this function is composed of 2^{n-1} conjunctions, i.e. the number of combinations with an even number of set bits, this makes the number of hidden neurons grow exponentially.
If we're allowed to use more than one hidden layer, we can construct an MLP that chains together $n-1$ simple networks for computing the biimplication (which basically is a n-bit parity on 2 bits). The number of neurons increases to $n(n+1)-1$.

The concept is that **increasing the number of hidden layers can decrease the complexity required**.

Another motivation for DNN is that **training data sets are limited in size**. A complete training data for any n-ary boolean function has 2^n training examples and in practical problems usually there are much fewer sample cases. Reducing the number of neurons allows us to use data sets limited in size (less stuff to train I guess).

Main problems: while having deeper network can help they may also pose some problems:

- **Overfitting:** we're increasing the number of adaptable parameters and thus increasing capabilities, this can lead to overfitting the sample data
- **Vanishing gradient:** the gradient tends to vanish if many layers are backpropagated through and learning in the early hidden layers can become very slow, it gets smaller and smaller as you get towards the input

There are some possible **solutions to the overfitting** issue:

- **Weight decay:** avoid large values for the weights to prevent an overly precise adaptation
- **Sparsity constraints:** restrict the number of neurons in hidden layers, thus removing some connections, or require that only few of the neurons in the hidden layers should be active (on average)
- **Dropout training:** during training, some units are randomly omitted from the input/hidden layers (for both forward and backward propagation), the network is not fully connected

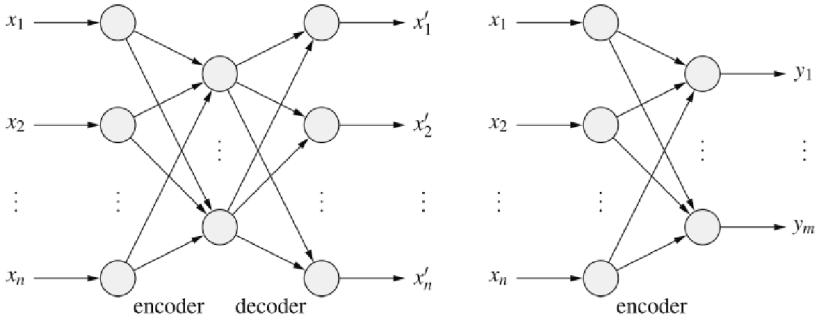
For the vanishing gradient, in principle, it could be counteracted by a large weight, but this leads to overfitting and saturation of the activation functions.

For the **vanishing gradient, solutions**:

- Obviously we can "brute force" it and increase the size of the training set, the number of epoch. It has some clear drawbacks, but as computational power increases this becomes more feasible.

- **Other activation functions** limit the problem of vanishing gradient, functions that have higher values of the gradient for a broader range of values, such as hyperbolic tangent, ReLUs and variations, softplus, ecc.; the point is **having large values of the gradient, close to 1, for larger ranges of inputs**, to avoid the gradient vanishing too fast.

- Another approach is **training the NN layer by layer**, training only the newly added layer in each step. A popular way of doing this is building the network as **stacked autoencoders**. An autoencoder is a 3-layer perceptron that **maps its inputs to approximations of themselves**. The hidden layer forms an encoder into some form of representation and the output layer forms a decoder that (approximately) reconstructs the inputs. It's trained to reconstruct itself, but the hidden layer is smaller so the encoder is forced to learn a meaningful representation of the input features. The hidden layer is expected to construct features.

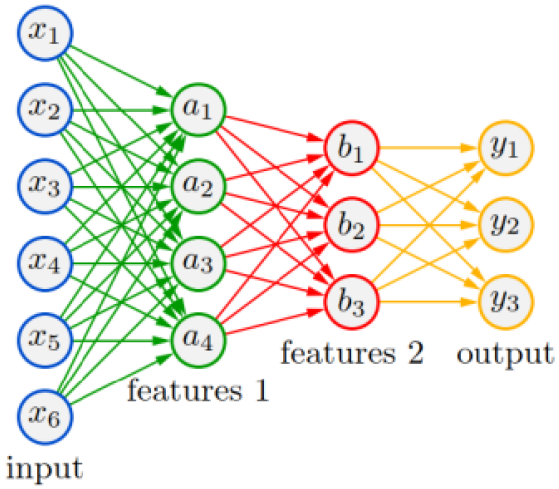


After the first encoder, I remove the decoder layer and add another hidden layer, only this last one added has to be trained, since we kept the already trained encoder part from before. Each encoder can be trained normally with backpropagation.

Problem: if there are **as many (or even more) hidden units as there are inputs** it's likely that the **input will propagate with little to no adjustments** to the output. We need to keep a limited number of neurons. Solutions:

- **Sparse auto-encoders:** provide a (considerably) smaller number of hidden neurons compared to the input layers.
- **Sparse activation scheme:** the number of active neurons in the hidden layer is restricted to a small number. Force sparsity constraints, add in the error function what is basically the number of active neurons so that the training will try to minimize such number.
- **Denoising autoencoders:** add noise to the input (but not to the data used for evaluation) and train the autoencoder to reconstruct the clean version, this way the input can't simply "pass through".

Example: the first layer is trained to reconstruct the input through features, the second layer wants to reconstruct the features created in the first and then these features can be used in a classifier/predictor to have my actual output



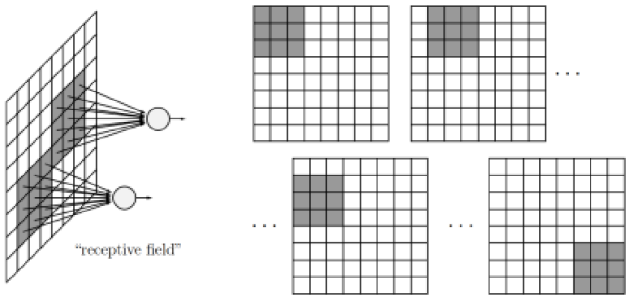
Put everything together and, optionally, fine tune it with backpropagation, but it already should be very close to the optimal value.

1.16 Convolutional Neural Networks

Still deep learning, we still have many layers. The difference with standard MLPs brought by Convolutional Neural Networks CNNs is **reducing the receptive field**, the **number of connections** such that **every neuron is connected to a limited number of input data**; so far everything was fully connected, with CNNs we remove this constraint. Each hidden neuron is connected only to a partial region of the input (previous layer).

This allows to have very deep networks, since the number of connections is reduced. The more hidden layers allow for more abstract representations and thus increasing understanding capabilities of the network without increasing too much the number of connections.

Each neuron of the first hidden layer is **connected to a few input neurons**



In this example the neuron is connected to a 3×3 **kernel**.

The "Convolutional" comes from the fact that the **connections weights are shared**, all neurons in the same layer have the same weights. Since all the connections are kinda the same it's like sliding a window over an image and doing a convolution with that particular kernel. The input field is "moved" step by step over the whole image and doing the equivalent of a convolution.

There are usually multiple neurons with the same receptive field, to compute multiple convolutions for different weight matrices/features.

We're not forced to have always the same kernels across multiple layers, they can be different with different weights.
Based on what the weights are each layer will be trained to "look" for different things in the image.

Neurons in the successor layer apply **maximum pooling** over small regions, after the convolution there's a maximum pooling layer, and it will output only the maximum activation of the neurons for each sampled region (kernel). I maintain the obtained results, but I don't know where, pooling **maintains knowledge** of the features, **not their location** in the image, but we're not interested in the exact location; if I have to make a classification, at the end of multiple pooling layers I will have a single activation value which will give me my answer.
Remove noise, keep knowledge. **Further layers** allow for **high-level features**.

In a CNN multiple stages of convolution, maximum pooling and ReLUs can be present, building a **hierarchy of image features**.

CNNs are **translation invariant**: with the way they're built they don't care where an object is (property of convolution) they just find it, the window is sliding, it will get there eventually. The pooling will remove the location from the image.

1.17 Radial Basis Function Networks

Another topology of NN, Radial Basis Function (RBF) Networks are feed-forward neural networks with a strictly layered structure, with **always three layers**, one hidden layer in which radial basis functions are employed. Kinda the opposite direction of deep learning.

RBF are used as **activation function in the hidden layer**. An RBF is a function that has its peak at zero and decreases in all other directions.

The network **input function of the output neurons** is the **weighted sum of their inputs**. From hidden to output neuron there's a "classical" weighted sum.

The **input function of each hidden neuron** is a **distance** function of the **input** vector and the **weight** vector (distance between input and weight). Each hidden neuron has a weight vector and receives the distance between input and weight vectors; it's not a weighted sum. It can be seen as any type of Euclidean distance function:

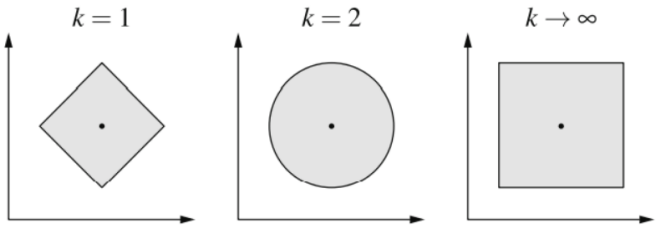
- $\forall u \in U_{hidden} : f_{net}^{(u)}\left(\vec{w}_u, \vec{in}_u\right) = d\left(\vec{w}_u, \vec{in}_u\right)$
- $d(\vec{x}, \vec{y}) = 0 \Leftrightarrow \vec{x} = \vec{y}$
- $d(\vec{x}, \vec{y}) = d(\vec{y}, \vec{x})$, Symmetry
- $d(\vec{x}, \vec{z}) \leq d(\vec{x}, \vec{y}) + d(\vec{y}, \vec{z})$, Triangle inequality

There can be different families of distances, we'll consider the **Minkowski Family**, which are distances defined by the formula

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

With different values of k we have different distances:

- $k = 1$: Manhattan distance
- $k = 2$: Euclidean distance
- $k \to +\infty$: Maximum distance



The **activation function** of each **output neuron** is a **linear function**, weighted sum that propagates to the output.
The **activation function** of each **hidden neuron** is a **radial function**, which decreases monotonically from 0 to ∞

$$f : \mathbb{R}_0^+ \to [0,1] \text{ with } f(0) = 1 \text{ and } \lim_{x \to \infty} f(x) = 0$$

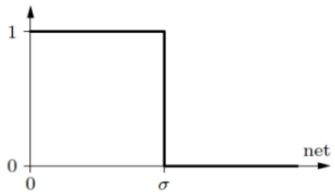
Basically a bell shape, like a Gaussian.
The **size of the catchment region** is **defined by the reference radius** σ, defines the shape and width of the function and thus the area "observed" by the neuron. Some examples of radial functions:

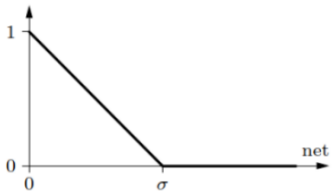
- Rectangle function

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

- Triangle function

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



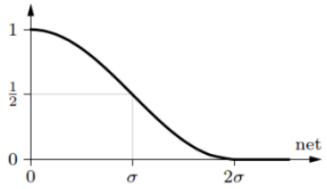


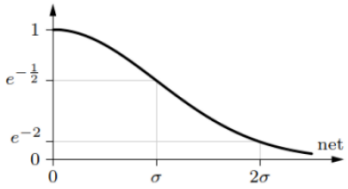
- Cosine until zero

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

- Gaussian function

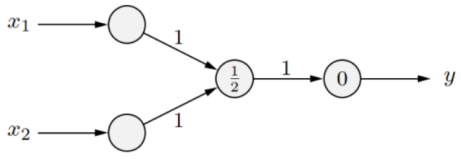
$$f_{act}(net, \sigma) = e^{\frac{net^2}{2\sigma^2}}$$

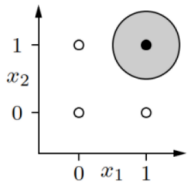




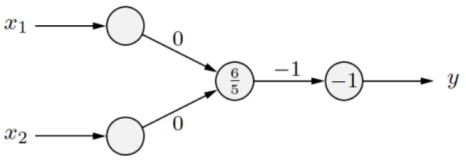
Each neuron can have its own function, maybe just with different σ, the function is activated based on the distance between weight and input; the weight vector acts as center for the function and the activation is proportional to how far the input is to that radial function. Each neuron has a function centered in a different place. This draws a circle (or equivalent) in the feature space, while an MLP draws a line.

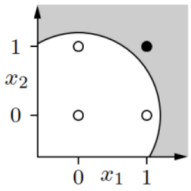
Example: RBF network for the conjunction $x_1 \wedge x_2$. We want to create a circle around the area in which the output is 1:





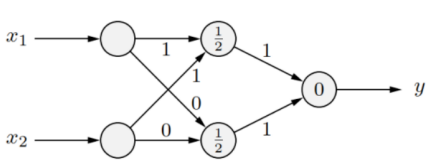
Another approach could consist in considering the RBF in the opposite way, where the data is outside the circle:

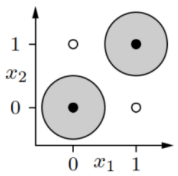




Example: For the bi-implication $x_1 \leftrightarrow x_2$

$$x_1 \leftrightarrow x_2 \equiv (x_1 \wedge x_2) \vee \neg(x_1 \vee x_2)$$

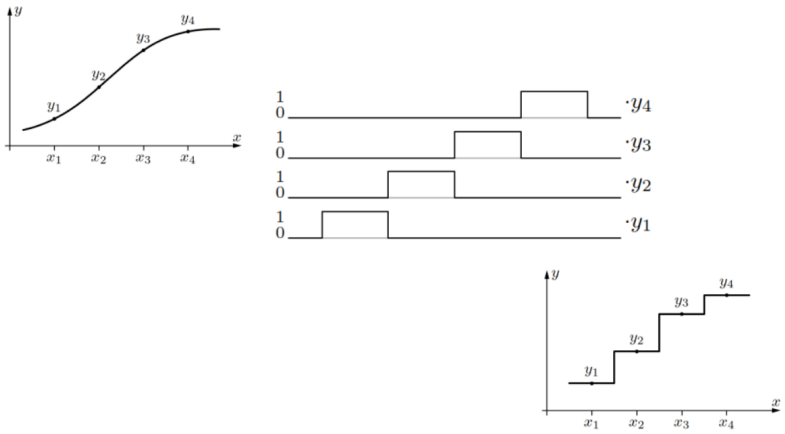




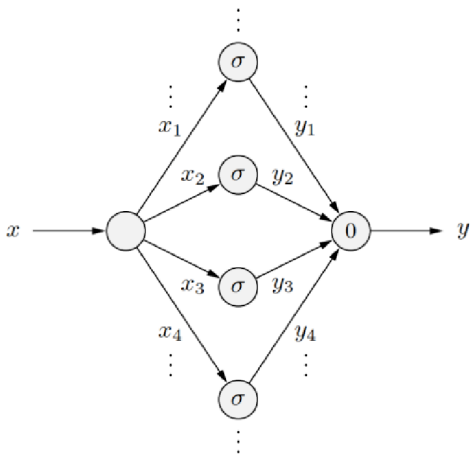
Here we draw 2 circles around the 2 output that give 1, each of the 2 hidden neurons constitutes a circle with different center.

1.17.1 Function approximation

MLPs basically **create a function approximation by a series of steps**. With RBFs we can do the same, but it gives us **more choices for how to approximate**. A **step function** can be modeled by an RBF as a **weighted sum of rectangular functions**, with the same result as MLPs.

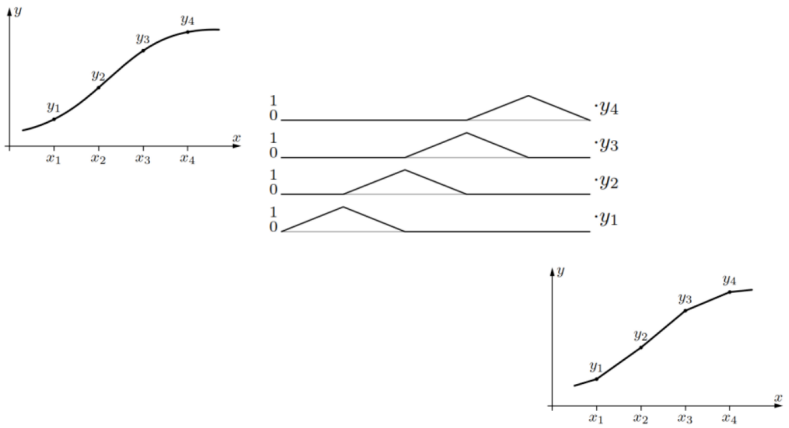


And the NN would be



$$\sigma = \frac{1}{2}\Delta x = \frac{1}{2}(x_{i+1} - x_i)$$

But we're not limited to this, we could use triangular pulses, considerably **improving the approximation** (or same approximation with fewer neurons)



We can **use any kind of function**, not only triangular, for example Gaussian functions which would produce "smooth" transitions between our "steps". The weights between input and hidden determine the locations of the Gaussian functions while the weights of the connections from the hidden neurons to the output neuron determine the height/direction (upward or downward) of the Gaussian functions.

1.17.2 Training Radial Basis Function Networks

We'll consider a case of supervised learning with a **fixed task** $L_{fixed} = \{l_1, ..., l_m\}$ with m training patterns $l = \left(\vec{i}^{(l)}, \vec{o}^{(l)}\right)$.

Considering a **simple radial basis function network** which is a partial view of the general structure of a RBFN, where **each training example is covered by its own radial basis function**. One hidden neuron v_k, $k = 1, ..., m$ for each training pattern

$$\forall k \in \{1, ..., m\} : \vec{w}_{v_k} = \vec{i}^{(l_k)}$$

The weights of the connections to the hidden neurons are simply initialized with the elements of the input vector of this training example.

If the **activation function** is **Gaussian**, the radii σ_k are **chosen heuristically**:

$$\forall k \in \{1, ..., m\} : \sigma_k = \frac{d_{max}}{\sqrt{2m}}$$

d_{max} is the **maximal distance** between the **input vectors** of two training patterns

$$d_{max} = \max_{l_j, l_k \in L_{fixed}} d\left(\vec{l}^{(l_j)}.\vec{l}^{(l_k)}\right)$$

this distance is chosen in a way that doesn't interfere with other patterns. Each radial function should not interfere with other patterns.

Having one hidden neuron for each training pattern means that the function is centered around that specific training pattern, the weights are set to the same values as the pattern, so the function will reach the maximum value for that particular training pattern (the distance will be zero); the function is centered around the specific training pattern.

Initialization: The connections from hidden to output neurons can be initialized by the formula

$$\forall u : \sum_{k=1}^{m} w_{uv_m} out_{v_m}^{(l)} - \theta_u = o_u^{(l)}$$

We can analytically solve the problem to find the values of these weights. We know the desired output, threshold and current output, so we know everything besides the **weights**, which can be **obtained** by **multiplying** the **interconnections weight** by the **matrix $\boldsymbol{A}$** of the **hidden layer output**, considering $\theta = 0$:

$$\boldsymbol{A} \cdot \vec{w}_u = \vec{o}_u$$

Where $\vec{o}_u = \left(o_u^{(l_1)}, \ldots, o_u^{(l_m)}\right)^T$ is the vector of desired outputs, $\theta_u = 0$ and $\boldsymbol{A}$ is the $m \times m$ **matrix** that has for components the **various output of the neurons in the hidden layer**:

$$\boldsymbol{A} = \begin{Bmatrix} out_{v_1}^{(l_1)} & out_{v_2}^{(l_1)} & \ldots & out_{v_m}^{(l_1)} \\ out_{v_1}^{(l_2)} & out_{v_2}^{(l_2)} & \ldots & out_{v_m}^{(l_2)} \\ \vdots & \vdots & \vdots & \vdots \\ out_{v_1}^{(l_m)} & out_{v_2}^{(l_m)} & \ldots & out_{v_m}^{(l_m)} \end{Bmatrix}$$

Each matrix row contains the outputs of the different neurons for one training example, each column contains the outputs of one hidden neuron for all training examples.

Can be **solved by inverting $\boldsymbol{A}$**

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

This method guarantees a **perfect approximation**. This means that is not necessary to train a simple radial basis function network. In general, if we don't want to have a neuron for each pattern, we'll have to select k subsets from the dataset and find, for each subset, a representative that will be associated to a neuron in the hidden layer.

General Radial Basis Functions: Not the simple case, we can't have one neuron for each training sample. RBFN usually (always) possess **fewer neurons** in the hidden layer **than** there are **training examples**, otherwise it would not be feasible due to complexity and sometimes not even useful (what if the examples are clustered together?).

We need to **select a subset** of k **training patterns** as centers

- Connection **weights** for the **hidden neurons**: **training patterns** (the chosen ones)
- Connection **weights** for **output neurons**:

$$\boldsymbol{A} = \begin{Bmatrix} 1 & out_{v_1}^{(l_1)} & out_{v_2}^{(l_1)} & \dots & out_{v_m}^{(l_1)} \\ 1 & out_{v_1}^{(l_2)} & out_{v_2}^{(l_2)} & \dots & out_{v_m}^{(l_2)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & out_{v_1}^{(l_m)} & out_{v_2}^{(l_m)} & \dots & out_{v_m}^{(l_m)} \end{Bmatrix}$$

  Using the pseudo-inverse matrix since it's over-determined (it's not square, m equations for each output neuron from the training examples but only $k+1$ weights from the k neurons and the 1 bias value, with $k < m$; more equations than unknowns, more training patterns than weights).

How do we **choose the centers**? How we select the centers can influence the training

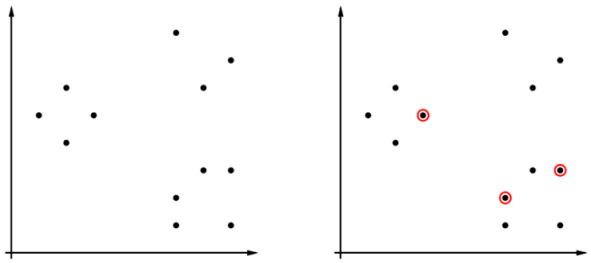
- **Random subsets:** select randomly, fast and only radius and output weights need to be determined, but the performance depends on the choice of data points (on which we have no control over)
- **All data points:** like the simple RBFN, only radius and output weights need to be determined and output values can be achieved exactly, but computing the weights can become unfeasible
- **Clustering:** find the centers of the data with a dedicated algorithm, C-means clustering

C-means clustering: Used to find RBFN centers

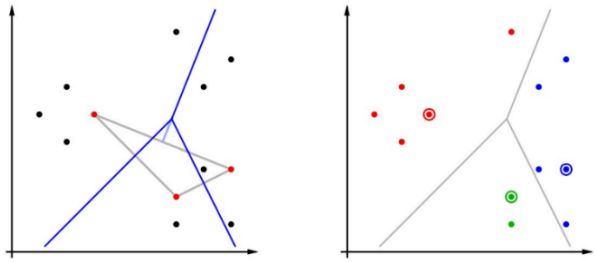
1. Considering a **number of** c **clusters** to be found (input)
2. **Initialize** the cluster centers **randomly**
3. **Assign** each **data point** (the examples) to the **closest cluster center** (or *centroid*)
4. **Adjust** the **position** of the **center** considering the values of the assigned data points (mean vectors of the assigned data points, this is called cluster center update)
5. **Repeat** the **step 3 and 4** until clusters do not move anymore (convergence).

The idea is, get c clusters, start with an approximate center and move it one step at a time, it will reach the actual center of the cluster, eventually.

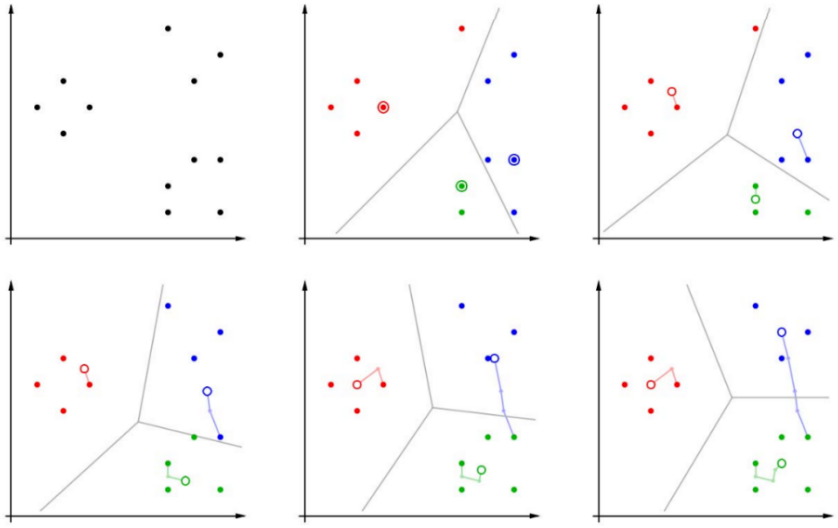
Example: with $c = 3$ we choose random centers



Then we can use the **Delaunay Triangulation** of the centers (make triangles out of the centroids, left), which leads to the **Voronoi Diagram**, by taking the line orthogonal to each one of the three edges (right). This will result in a tessellation of the domain of the function, basically each tassel of the domain is a cluster.



After that we can search for the new center of each cluster, where the sum of the distances is minimal, after finding it we can apply again the same partition (Delunay-Voronoi) as before and repeat the process with the new data points added by the new partition.



We can add backpropagation, we can **train the network**; if we have fewer hidden neurons there are training examples the quality of the RBFN can usually be improved by training. Update rules are analogous to MLPs.

We can update the weights from hidden to output neurons, the center coordinates (weights from input to hidden neurons) and the radii of the radial basis functions. We can do this by **following the backpropagation rules**, using the derivation of the output with respect to the variation of the weight.

A **3 phases RBF** consists of:

- Phase 1: find output connection weights with inverse
- Phase 2: find RBF centers (e.g. clustering)
- Phase 3: error backpropagation

An RBFN with good initialization, clustering and backpropagation can obtain good performance.

1.18 Learning Vector Quantization

What if we don't know the output? So far we've only considered fixed learning tasks (input/output pairs, supervised learning), which allowed us to describe training as error minimization. Now we'll be considering **free learning tasks** (unsupervised learning): the data consists **only** of **input values**/vectors. The **objective** is to **produce similar outputs for similar inputs** (clustering).

The concept behind a lot of unsupervised approaches is **finding clusters** in a given set of data points. The algorithm will work towards finding the given number of clusters and represents them with their center (or a reference vector, same thing).

We can still use the Delunay and Voronoi approach to find clusters. But we want to use **Learning Vector Quantization**, which is a way to **find a suitable quantization** (many-to-few mapping, often to a finite set) of the **input space** (e.g. tessellation of a Euclidean space), a way to **divide in cluster** the input space.

Training **adapts** the **coordinates of reference** or **codebook vectors** (centers), each of which **defines a region in the input space**.

Learning vector quantization **processes the data points one by one** and **adapts only one reference vector per data point**. The procedure is known as **competitive learning**: the training patterns (data points) are traversed one by one. For each training pattern, a "competition" is carried out, which is **won by the output neuron that yields the highest activation for this training pattern** (if distance and activation functions are the same for all output neurons, we may say equivalently: whose reference vector is closest to the training vector). Only this "winner neuron" is adapted.

It's still a way to find the centers of the clusters in a way that makes inferring the class of the object possible. There is a learning process called competitive learning, we find the reference vector (center) closest to our input and its position will be updated before going to the next training pattern.

1.18.1 Learning Vector Quantization Networks

A **Learning Vector Quantization Network** (LVQ/LVQN) is a **feed-forward 2-layered neural network**. It can be seen as RBF without the output layer, the hidden layer is used as output, the hidden layer outputs the corresponding activation for each of the neurons and that's the final output. For this reason "hidden" and "output" neurons will be used interchangeably.

The **network input function** for each **output neuron** is a **distance function** of the **input vector and the weight vector**. The definition of the distance is the same. Each neuron in the hidden layer has its own radial function centered in a different place (it might be the same function, but each one has its own weight vector centering it in a different place) whose input is a distance from input to weight vector (since it acts as center of the function).

The **output function** of **each output neuron** is not a simple function of the activation of the neuron, it **considers the activation of all output neurons**

$$f_{out}^{(u)}(act_u) = \left\{ \begin{array}{cc} 1 & \text{if } act_u = \max_{v \in U_{out}} act_v \\ 0 & \text{otherwise} \end{array} \right\}$$

It's **winner-takes-all**.

Biggest activation will lead to a value of 1, 0 otherwise. We have one neuron winning over all the other, basically we have a connection between all neurons in the output layer such that each of them is able to see what the other neurons are doing (the activation), allowing the confrontation. All outputs will be 0, except one, which will be 1. If more than one unit has the maximal activation, one is selected at random, but if it happens we probably have chosen more cluster than needed.

In the LVQN we can use the same radial functions that are use for any RBFN.

Learning rules: We want to learn the position of the reference vector (center of the radial function) of the cluster. For each training pattern we find the closest reference vector (winner neuron) and adapt only that one. Each reference vector is assigned to a class.

To **adjust the weights** we can use (assuming supervised learning):

- **Attraction rule**, when data point and reference vector have the same class (output is of the correct class)

$$\vec{r}^{(new)} = \vec{r}^{(old)} + \eta \left(\vec{x} - \vec{r}^{(old)} \right)$$

  We move the reference vector closer to the input point.

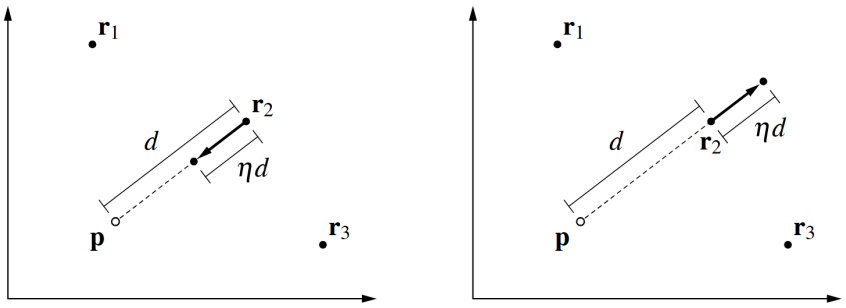
- **Repulsion rule**, when data point and reference vector have different class (output is of the wrong class)

$$\vec{r}^{(new)} = \vec{r}^{(old)} - \eta \left(\vec{x} - \vec{r}^{(old)} \right)$$

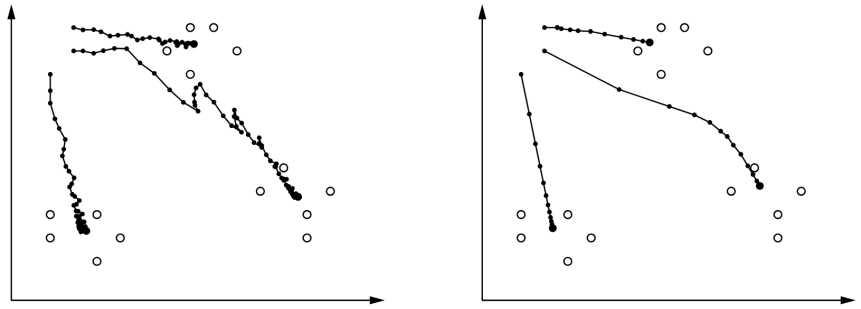
  Complementary to the last one, the output is of the wrong class, the reference vector has to be moved further away.

Where x is the input, r is the reference vector and η is a learning rate ($0 < \eta < 1$).

Representation of the idea:



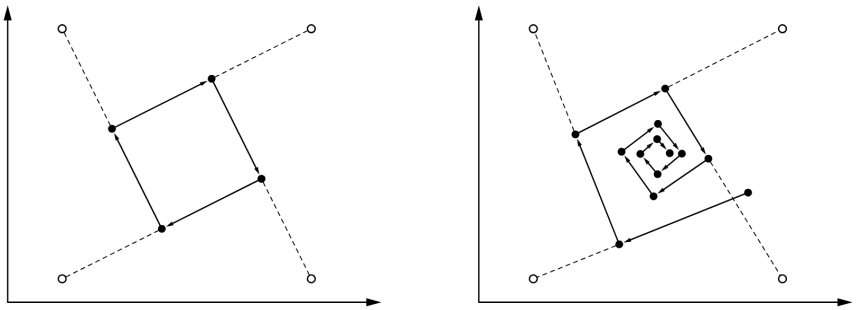
The adaptation of reference vectors through some epochs (not many, there's still some training to do) can be seen here:



Online training on the left, batch training on the right.

We can see the random initialization of the reference vectors and the average movement (especially in batch training) goes towards the center of the clusters, the centers of the Gaussian functions get weighted towards the right values.

Time dependent learning rate: **Fixed learning rate** η can **lead to oscillations**, the centers can move without never converging (left image, $\eta(t) = 0.5$). A solution for this can be **time dependent learning rate**, the parameter η gets smaller with time (right image, $\eta(t) = 0.6 \cdot 0.85^t$, starting at $t = 0$).



This way instead of oscillating forever the value of the reference vector converges to the center.

Update rule for classified data: We can update not only the reference vector that is closest to the data point, but **update the two closest reference vectors**, provided that they are of two different classes. This enables to apply **attraction and repulsion rule** at the **same time**.

It's a way to use the most significant training patterns to update the network more. If the two closest centers to a pattern are from different classes it means that the pattern is on the boundary of said classes, and thus meaningful. We move the correct class closer and the wrong one further away.

Window Rule: Standard learning vector quantization may **drive the reference vectors further and further apart**. The drawback of the repulsion rule is that it might push the centers far away. The **window rule** is a way to reduce this problem, the **update** happens **only if the data point $\vec{x}$ is close to the classification boundary**, the point must lie close to the (hyper-)surface that separates the regions in which different classes are predicted. The formula is:

$$\min\left(\frac{d(\vec{x},\vec{r}_j)}{d(\vec{x},\vec{r}_k)}, \frac{d(\vec{x},\vec{r}_k}{d(\vec{x},\vec{r}_j)}\right) > \theta \quad \text{where} \quad \theta = \frac{1-\xi}{1+\xi}$$

Where ξ is a parameter specified by a user, and it describes the "width" of the window around the classification boundary. Note that $\vec{r}_j$ and $\vec{r}_k$ must be of two different classes.

This rule prevents a divergence of the reference vectors because the **adaptation** caused by a data point **stops as soon as the classification border is far enough away**.

Soft-Learning vector quantization: Instead of the winner-takes-all approach, we can have a **soft assignment**, all the neurons will output *something*, not zero, essentially giving a probability of being correct to each classification. The assumption is that the given data was samples from a mixture of normal distributions. The adaptation is done with the help of **gradient descent**, **maximizing** an objective function that describes the **probability** that a **data point is correctly classified**.

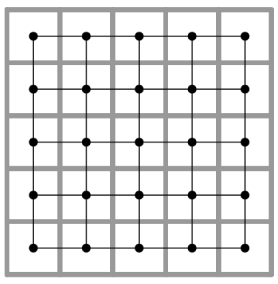
1.19 Self-Organizing Maps

Self organizing maps (SOM), sometimes also called **Kohonen feature map** (by the name of their inventor), are a feed forward neural network with 2 layers, similar to the Vector Quantization Learning networks, **with local connections only among neighboring hidden/output neurons**. We're organizing the centers of neighboring radial functions to be close to each other. An LVQ with centers organized in some way.

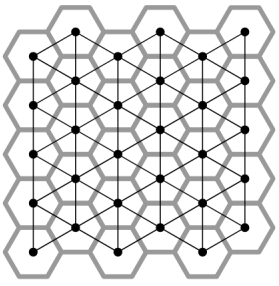
The network **input function** of each input neuron is a **distance function** of **input** and **weight** vector. The **activation function** of each output neuron is a **radial function**.

The output function of each output neuron is the identity, and the output is often discretized according to the winner-takes-all principle.

On the **output neurons** a **neighborhood relationship is defined**. This means that the centers of the radial function are connected together, neighboring neurons will have centers which are close together. This leads to a (usually two-dimensional) **grid of neurons**. For example:



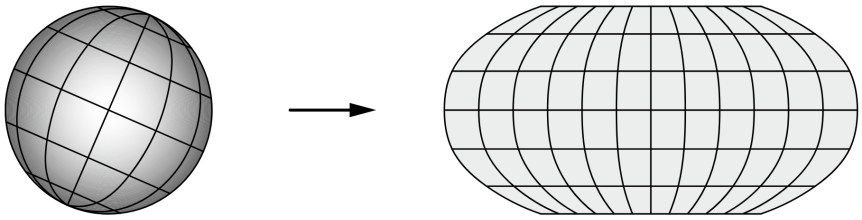
quadratic grid



hexagonal grid

The radial functions are placed in a two-dimensional space. Each neuron is connected to its nearest neighbor and has a region assigned to it. The **neighborhood relationship** of the output neurons is usually **defined** by **arranging these neurons into a** (usually two-dimensional) **grid**. **Reference vectors close to each other** in the input space belong to **neurons** that have a **small distance from each other**.

Topology Preserving Mapping: The SOM perform a **meaningful dimensionality reduction**, e.g. we have data in 3 dimensions, and we want to reduce it to 2 (just like the projection of the globe on a map). But we still want to **preserve the topology** (points near to each other on the globe will be close together on the map), although angles, distances and areas may be distorted. We're not inventing a new map, we're projecting something in a meaningful way, which approximately preserves the position of points.



Images of **points close to each other** in the **original space** should be **close to each other** in the **image space**.

With topology preserving maps we can **map high-dimensional structures** (input space) **onto low-dimensional spaces** (neuron space/grid).

We need to adjust the training to incorporate to maintain the neighboring structure. This works only on **unsupervised learning/free-learning tasks**, the algorithm should be free to adjust the mapping and the classes by itself. In the training we need to respect the neighborhood, so we need to modify the attraction rule to consider the concept of neighborhood:

$$\vec{r}_u^{(new)} = \vec{r}_u^{(old)} + \eta(t) \boldsymbol{f_{nb}} \left(\boldsymbol{d_{neurons}(u, u_*), \varrho(t)} \right) (\vec{x} - \vec{r}_u^{(old)})$$

Where:

- u_* is the **winner neuron** (reference vector closest to the data point)
- the **neighborhood function** f_{nb} is a **radial function**
- **Learning rate** and **neighborhood radius** are **time-dependent**, they both decrease with time

More than one neuron is updated, and it's done in a way which considers the distance between the winner neuron and the one to be adapted. All output neurons in a certain radius will be updated, each with a certain strength (strength and distance defined by the neighborhood function).

The **learning rate** will **decrease with time**, as seen before. The **neighborhood radius** does the same thing, at the start a bigger area around the winner neuron will be updated, as time goes on this area decreases.
This avoids oscillations and ensures convergence.

Training: The weight vector of the neuron in the self-organizing map is **initialized randomly**, the reference vector are placed randomly in the input/data space, using random training examples.

The **training** itself **consists of**:

- **Choosing** a **training sample**/data point
- **Finding** the **winner neuron** with the distance function in the data space (neuron with the closest reference vector)
- Computing the time dependent radius and learning rate and **adapt the corresponding neighbors** of the winner neuron

Example MIA

1.20 Hopfield Networks

The Hopfield Network (HN) is an ANN with a loop in the graph. It's the simplest type of **recurrent network**, a graph with direct cycles. **All** the **neurons** are **both input and output**. There is **no hidden layer** of neurons, each neuron is connected to another (except single neuron loops) and the connection weights are distributed symmetrically. **Each neuron** receives **input from all other neurons** while **not being connected to itself**.

A **Hopfield network** is a **neural network** with a graph $G = (U, C)$ that satisfies the following conditions:

1. $U_{hidden} = \emptyset$, $U_{in} = U_{out} = U$,
2. $C = U \times U - \{(u, u) | u \in U\}$.

The **connection weights** are **symmetric** $\forall u, v \in U,\ u \neq v : w_{uv} = w_{vu}$. The **network input function** of **each neuron** u is the **weighted sum** of the **outputs of all other neurons**, that is

$$\forall u \in U : f_{net}^{(u)}(\vec{w}_u, in_u) = \vec{w}_u in_u = \sum_{v \in U - \{u\}} w_{uv} out_v$$

The **activation function** of each neuron u is a **threshold function**

$$\forall u \in U : f_{act}^{(u)}(net_u, \theta_u) = \begin{cases} 1 & \text{if } net_u \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

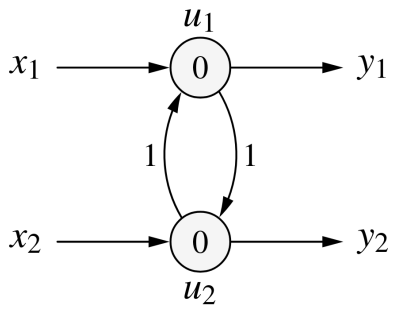
The **output function** of each neuron is the **identity**, that is

$$\forall u \in U : f_{out}^{(u)}(act_u) = act_u$$

Since the connection weights are symmetric the **weight matrix** will obviously be **symmetric**. A Hopfield network with n neurons $u_1, \ldots, u_n$ can be described by the $n \times n$ matrix

$$W = \begin{pmatrix} 0 & w_{u_1 u_2} & \ldots & w_{u_1 u_n} \\ w_{u_1 u_2} & 0 & \ldots & w_{u_2 u_n} \\ \vdots & \vdots & & \vdots \\ w_{u_1 u_n} & w_{u_2 u_n} & \ldots & 0 \end{pmatrix}$$

Simple Examples: A network that oscillates if the activations of the two neurons are updated in parallel, but reaches a stable state if the neurons are updated alternatingly

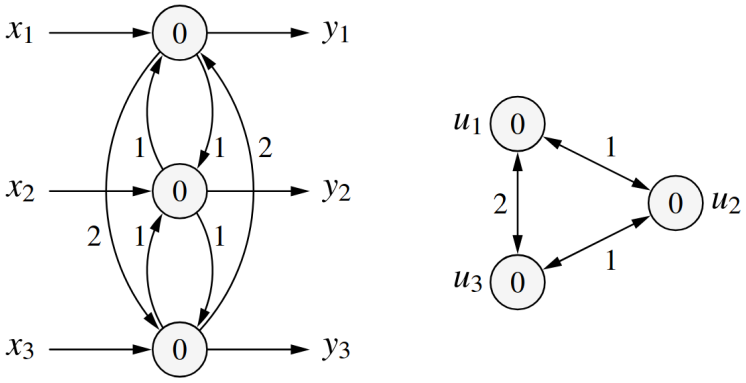


The weight matrix is:

$$W = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

There are no loops in a Hopfield network, that is, no neuron receives its own output as input. All feedback loops run through other neurons: a neuron u receives the outputs of all other neurons as its input and all other neurons receive the output of the neuron u as input.

A three neuron example and its simplified representation (right):



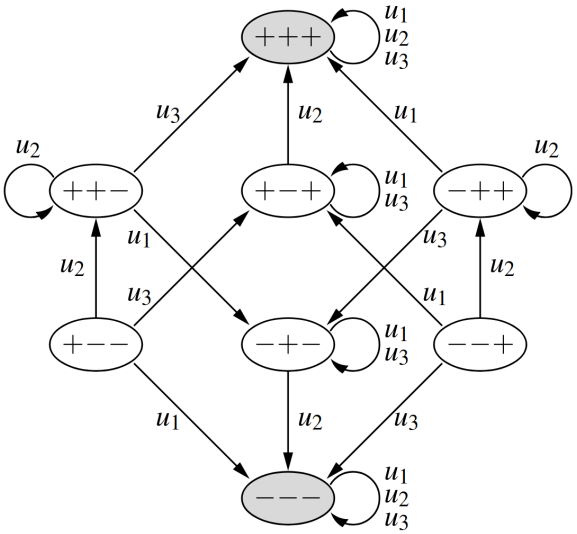
We don't need to explicitly draw input and output arrows since each neuron is both input and output. The weights are symmetric so a single arrow for both sides it's enough.

In a network with cycles, such as a Hopfield network, the **behavior depends** also on the **update order**:

- **Synchronous update:** updating the values at the same time, in parallel. This way the computation can oscillate.

- **Sequential update:** updating the values asynchronously, one after the other. This always leads to convergence, which stable state is reached depends on which neuron is updated first.

For the three neuron example, an asynchronous update leads either to the stable state $(1, 1, 1)$ or to the stable state $(-1, -1, -1)$. This can be seen with the **state graph**. **Each node** in the graph represents a **configuration for each neuron**. The $+/-$ **encodes** the **neuron activation** (active and inactive respectively)



Labels on **arrows** indicate the **neurons**, whose **updates** (activation changes) lead to the corresponding state transitions.
States shown in **gray** are **stable**, cannot be left again. States shown in **white** are **unstable**, may be left again.

1.20.1 Convergence Theorem

The convergence theorem states that, if the **activations of the neurons** of a Hopfield network are **updated sequentially** (asynchronously), then a **stable state is reached** in a **finite number of steps**.
If the **neurons** are **traversed** and updated **cyclically** in an **arbitrary**, but **fixed order**, at **most** $n \cdot 2n$ **steps** (updates of individual neurons) are needed for reach the convergence, where n is the number of neurons of the HN.

We can prove this by defining a function that maps every state of a Hopfield network to a real-valued number, which is reduced with every state transition or at least stays the same. This function is commonly called the **energy function** of the Hopfield network.

So, the system can only evolve **from** one **state with higher energy** to **another with lower energy**. A **stable state** will be a **local minimum** of the energy function.

If in a traversal of all n neurons there are **no activation changes**: a **stable state** has been reached.
If in a traversal of all n neurons **at least one activation changes**: the **previous state cannot be reached again** since updates can't go back to states with higher energy.

1.20.2 Associative Memory

Hopfield networks are very well suited to implement so-called **associative memory**, that is, a kind of memory that is addressed by its contents. If a pattern is presented to an associative memory, the **memory returns** whether this **pattern coincides with one of the stored patterns**. This coincidence need not be exact, it may also return a stored pattern that is as similar as possible to the presented pattern, this way "noisy" patterns can be recognized.

The associative memory works by **exploiting the stable states** of HN. If we determine the **weights and the thresholds** of a HN in such a way that the **patterns to store are exactly the stable states**, the normal update procedure of a Hopfield network finds for any input pattern a similar stored pattern.

Hebbian learning rule:

1. Store one pattern p:
   - find weights so that pattern is a stable state
   - $W = pp^T - E$
   - $w_{uv} = \begin{cases} 0 & u = v \\ 1 & u \neq v,\ act_u^{(p)} = act_v^{(p)} \\ -1 & \text{otherwise} \end{cases}$

2. Storing several patterns
   - Compute W_i for each pattern p_i
   - $W = \sum_i W_i$

1.20.3 Solving Optimization Problems

We can exploit a **minimization of the energy function** to **solve optimization problems**, since a Hopfield network reaches a (local) minimum of said function in every stable state.

We need to:

- Transform **function to optimize** into a **function to minimize**.
- Transform **function** into the **form of an energy function** of a Hopfield network.
- **Read** the **weights and threshold** values **from** the **energy function**.
- **Construct** the corresponding Hopfield network.
- **Initialize Hopfield network randomly** and **update until convergence**.
- Read **solution** from the **stable state reached**.
- **Repeat** several times and **use best solution** found.

The solution found may not be the global optimum since it's just a local minimum.

Also, the asynchronous updates limit the kind of changes that the HN can perform, since a change may violate a constraint and increase total energy but multiple changes together may result in a lower energy state.

The procedure may get stuck in a local optima, similarly to what happens with many other optimization methods (e.g. gradient descent, hill climbing, alternating optimization).

1.20.4 Simulated Annealing

This is a possible approach to solve the problems given by getting stuck in a local minimum of the energy function.

The idea of simulated annealing is to **start** with a **randomly generated candidate solution** of the optimization problem and to **evaluate it**. In every later step, the current **candidate solution is modified** (slightly) and **re-evaluated**. **If** the **new solution is better** than the old, it is **accepted and replaces** the old solution. However, **if it is worse**, it is **accepted** only **with a certain probability** that depends on **how much worse** the new solution is. In addition, this **probability is reduced over time**. Furthermore, the best solution found so far is usually recorded in parallel.

It's an **extension of gradient descent** that tries to **avoid getting stuck**, the transition from higher to lower local minimums should be more probable than the opposite.

There is still *no guarantee that the global optimum will be found.*

Algorithm for a Hopfield network:

- All **neuron activations** are **initialized randomly**.
- The **neurons** of the Hopfield network are **traversed repeatedly** (for example, in some random order).
- For **each neuron**, it is determined whether an **activation change** leads to a **reduction** of the network energy or not.
- An **activation change that reduces** the network energy is **always accepted** (in the normal update process, only such changes occur).
- However, **if** an **activation change increases** the network **energy**, it is **accepted with a certain probability**.

1.21 Boltzmann Machines

An extension to the idea of Hopfield networks, instead of modeling a single pattern, **Boltzmann machines** want to **model a probability distribution**. The network is constructed with and wants to encode a probability distribution of the data over the whole input space, we want to understand how the data is structured and its significance. A standard Boltzmann machine differs from a Hopfield network mainly in how the states of the neurons are updated.

It's possible to define an **energy function** that assigns a numeric **value** (an energy) **to each state** of the network (as seen in HN). With the help of this function a **probability distribution over the states** of the network is defined based on the **Boltzmann distribution**:

$$P(\vec{s}) = \frac{1}{c} e^{-\frac{E(\vec{s})}{kT}}$$

Where:

- $\vec{s}$ describes the (discrete) **state of the system**
- c is a normalization constant
- E is the function that yields the **energy of a state** $\vec{s}$
- T is the thermodynamic temperature of the system
- k is Boltzmann's constant ($k \approx 1.38 \cdot 10^{-23} J/K$)

The state of the machine $\vec{s}$ consists of the **vector** $\vec{act}$ of the **neuron activations**. The energy function of a Boltzmann machine

$$E(\vec{act}) = -\frac{1}{2} \vec{act}^{\top} W \vec{act} + \vec{\theta}^{\top} \vec{act}$$

where W is the matrix of connection weights and $\vec{\theta}$ the vector of threshold values. The energy function depends on both the state (outputs) and configuration of the network.

The **energy change** resulting from the **change of a single neuron** u:

$$\Delta E_u = E_{act_u=1} - E_{act_u=0} = \sum_{v \in U - \{u\}} w_{uv} act_v - \theta_u = net_u - \theta_u$$

The energy difference/change is closely **related** to the **network input**.

The **probability** of a **neuron being active** is a **logistic function** of the (scaled) **energy difference** between its **active** and **inactive state**:

$$P(act_u = 1) = \frac{1}{1 + e^{-\frac{\Delta E_u}{kT}}}$$

Update procedure: The update step is:

- A **neuron** u is **chosen** (randomly)
- The **energy difference** ΔE_u and then the **probability** of the neuron **having activation 1** are **computed**
- The **neuron** is **set to activation 1** with this **probability** (and to 0 with the complement of said probability)

This update is **repeated many times** for randomly chosen neurons. There is no update for weights or threshold, we're talking about an already trained Boltzmann machine.

Simulated annealing is carried out by slowly lowering the temperature T.

This is a **Markov-Chain Monte Carlo (MCMC)** procedure.

After enough steps, the **probability** that the network is in a **specific activation state** depends only on the **energy of that state**, it's *independent of the initial activation state* the process started with. This final situation is also referred to as **thermal equilibrium**.

1.21.1 Training

The **probability distribution** represented by a Boltzmann machine via its energy function can be **adapted to a given sample of data points**.

Boltzmann machine can represent a probability distribution **only if the points in the data set are compatible with a Boltzmann distribution**. In order to mitigate this restriction, the neurons of a Boltzmann machine are divided into **visible** neurons, which **receive the data points** as input, and **hidden** neurons, the **activations** of which are **not fixed by the data points** and thus allow for a more flexible adaptation to the sample data. This is the first deviation from the structure of Hopfield networks.

The objective of the training is to **adapt weight and threshold values** in such a way that the true **distribution** underlying a given **data sample** is **approximated well** by the probability distribution represented by the Boltzmann machine on its visible neurons.

The approach is to **measure the difference between two probability distributions** and use **gradient descent** to **minimize** said difference. Start from random values and, using gradient descent, minimize the difference between the distribution created and the data given.

A commonly used measure of difference is the **Kullback-Leiber information divergence**

$$KL(p_1, p_2) = \sum_{\omega \in \Omega} p_1(\omega) \ln \frac{p_1(\omega)}{p_2(\omega)}$$

Where p_1 refers to the data sample and p_2 refers to the visible neurons of the Boltzmann machine. It's a **measure of divergence** between **two probability distributions**, in our case it becomes like a loss function.

Each training step there are **two phases**:

- **Positive phase**: the visible neurons are fixed to a data point chosen randomly, and the hidden neurons are updated until thermal equilibrium is reached.

- **Negative phase**: all units are updated until thermal equilibrium is reached

Update rule: The update is performed according to the following two equations

$$\Delta w_{uv} = \frac{1}{\eta}\left(p_{uv}^{+} - p_{uv}^{-}\right) \quad \text{and} \quad \Delta\theta_u = -\frac{1}{\eta}\left(p_u^{+} - p_u^{-}\right)$$

Where p_u indicates the probability that neuron u is active, p_{uv} indicates the probability that neurons u and v are both active simultaneously and the $+, -$ as upper indices indicate the phase referred to (positive and negative respectively).

All these probabilities are **estimated** from the **relative frequency** with which the **corresponding situation was observed in the respective phase**.

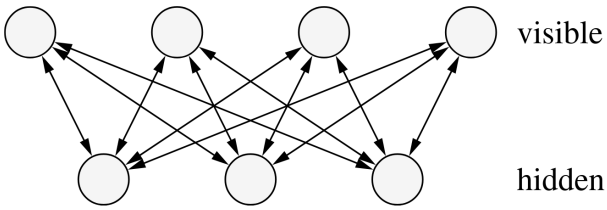
Intuitively: if a **neuron** is **more often active when a training example is presented** (i.e., visible units are fixed) than when the network is allowed to run freely (i.e., visible units are updated), the **probability of the neuron being active is too low**, so the threshold should be reduced.
Similarly, if **two neurons are more often active together** when a **training example is presented** than when the network is allowed to run freely, the **connection weight between them should be increased**, so that they become more likely to be active together.

This training procedure is **impractical unless the networks are very small**. The larger the network the more update steps are needed to obtain sufficiently reliable statistics for the neuron activation used in the update formulas. The complexity grows too quickly with the number of neurons.

1.21.2 Restricted Boltzmann Machines

It's a solution to **limit complexity** with respect to the number of neurons and allow for **efficient training**.

In a Restricted Boltzmann machine (RBM) we have a **bipartite graph** instead of a fully connected one. The connections only go from neurons in the visible layer to neurons in the hidden layer (no connections among the same group).



In an RBM **all input neurons are also output neurons** and vice versa. There are hidden neurons, which are different from input and output neurons. Each input/output neuron receives input from all hidden neurons and vice versa.

Due to the **lack of connections** within visible and hidden units **training** can proceed by **repeating three steps**:

- **First step**: **visible units are fixed** to a randomly chosen data sample $\vec{x}$ and **hidden units are updated** one and in parallel $\vec{y}$, retrieving the **positive gradient** for the weight matrix xy^T.

- **Second step**: **hidden neurons are fixed** to the vector $\vec{y}$, **visible units are updated** one and in parallel to the "reconstruction" $\vec{x}^*$. Then **hidden neurons are updated once more**, $\vec{y}^*$, and the **negative gradient** is obtained $\vec{x}^*\vec{y}^{*T}$.

- **Third step**: **connection weights are updated** with the difference of positive and negative gradient

$$\Delta w_{uv} = \eta \left(\vec{x}_u \vec{y}_u^T - \vec{x}_u^* \vec{y}_u^{*T} \right)$$

The RBM have also been used to build deep networks similar to stacked autoencoders for the MLP.