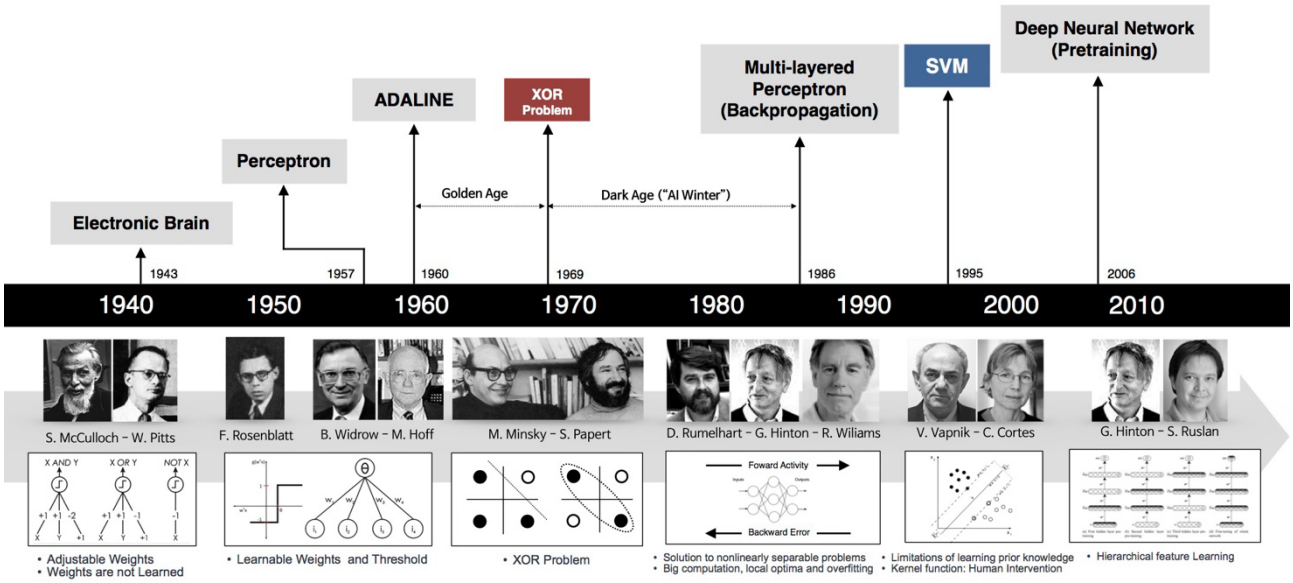
SHORT HISTORY OF (NEURAL-NETWORK BASED) MACHINE LEARNING

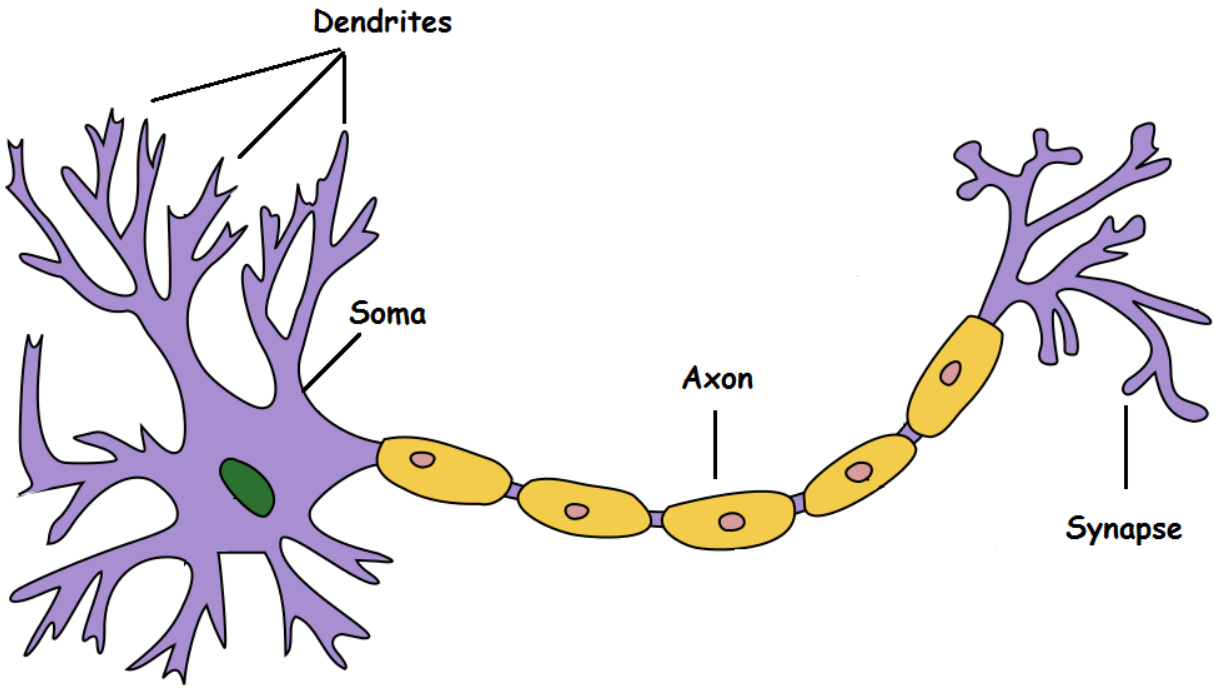


http://beamlab.org/deeplearning/2017/02/23/deep_learning_101_part1.html

A (VERY) SIMPLE (AND INACCURATE) MODEL OF OUR BRAIN

The story of machine learning starts with researchers trying to replicate how our brain works. Hence, the first machine learning models to be proposed where inspired to how the basic computation unit in our brain (the neuron) works.

NEURONS



https://en.wikipedia.org/wiki/Neuron

We can see a neuron as a simple computation machine which takes a series of inputs, processes them, and returns some output. The main components of the neurons are:

1. Dendrites: allow the neuron to take multiple inputs.
2. Soma: is the processing unit of the neuron.
3. Axon: is the "output channel" of the neuron.

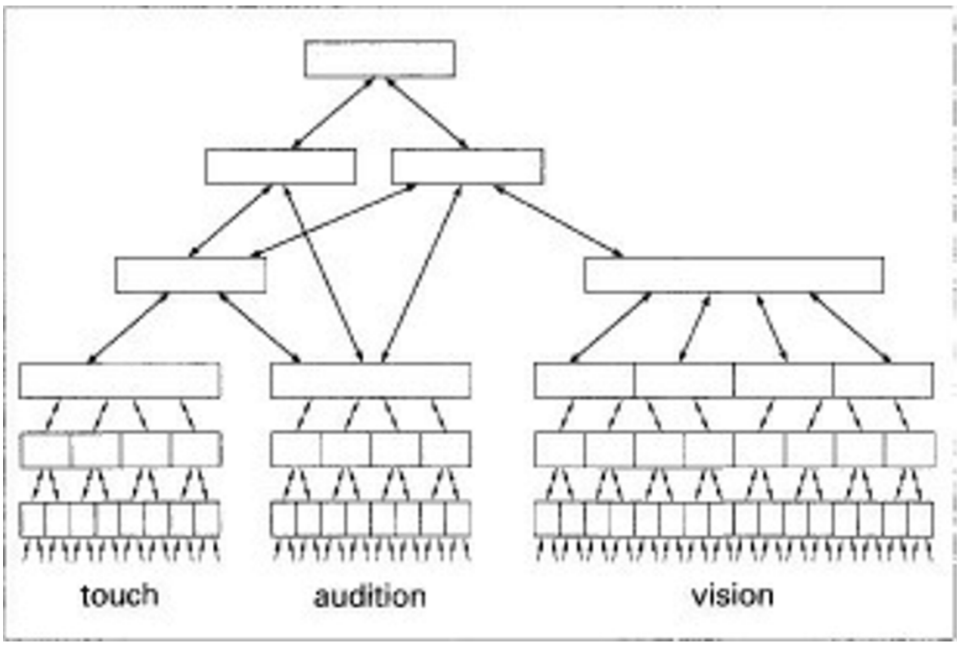
Neurons can communicate with other neurons through electrical signals which are received through the dendrites and transmitted through the axon. These signals tend to have only two levels: "high" and "low". When a neuron emits a high current level, then we say that **the neuron "fires"** or it is active, while the neuron is inactive when it emits a low current level. Hence, we can think of a neuron as a binary computation machine: it takes a set of binary inputs and outputs a binary value (0 or 1).

NEURAL NETWORKS

If neurons are such simple machines, how can our brain work? Many neurons connected with each other form a "network". This network is connected to our sensory organs (e.g., eyes, ears, etc.) to collect input signals. Depending on the observed signals, some neurons will "fire", while others will stay inactive.

Groups of neurons specialize to "fire together" when a given input is shown. For instance, a set of neurons will fire when a horizontal segment is shown in front of the eye, while others will be sensitive to vertical lines.

Neurons tend to be distributed in layers, with neurons within one layer mostly communicating with neurons in the upper and lower layers. Also, connections are mostly going from senses to neurons upper in the hierarchy (though so called "backward connections" do exist.

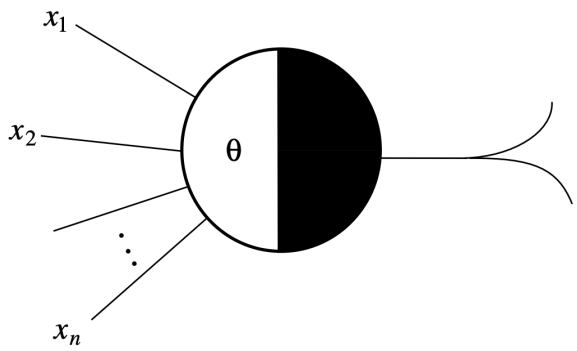


Hawkins, Jeff, and Sandra Blakeslee. *On intelligence*. Macmillan, 2004.

Neurons upper in the hierarchy tend to fire in the presence of more abstract concepts, such as human faces and object categories. This seem to suggest that representations in our head are hierarchical and high-level concepts (e.g., human faces) are represented as a composition of lower-level concepts (e.g., edges or human faces parts such as noses and mouths).

MCCULLOCH–PITTS NEURONS

The first attempt to mathematically model a neuron is due to McCulloch (a neuroscientist) and Walter Pitts (a logician) who proposed the McCulloch–Pitts neuron in 1943. A McCulloch–Pitts can be seen as a unit which takes n inputs and returns one output:

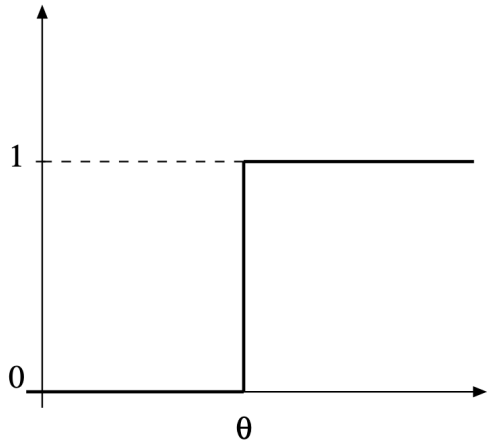


Similarly to the model of a neuron, all inputs and outputs are **binary**. The neuron is also associated with a threshold θ.

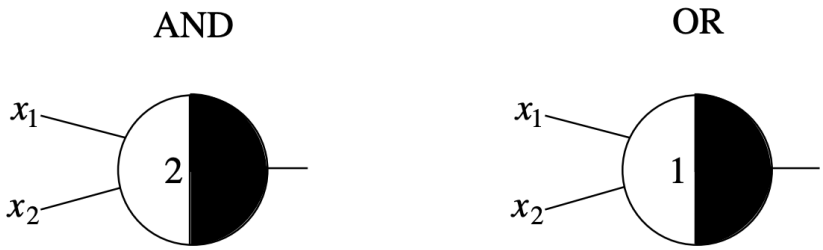
The output can "fan out" many times, connecting to other neurons. The inputs x_i may come from other neurons or be features of an input element. Each input is connected to the neuron via directed edges (all going from the input to the neuron). There are two kinds of edges: **excitatory** and **inhibitory**. The neuron computes a simple computation:

1) Assume the neuron gets $x_1, \dots, x_n$ excitatory inputs and $y_1, \dots, y_m$ inhibitory inputs.
2) If $m \geq 1$ and at least one of the inhibitory inputs is 1, then the neuron is inhibited, and the final computation is 0 (the neuron does not fire).
3) Otherwise, the total excitation is computed summing the values of the excitatory inputs: $x = x_1 + \cdots + x_n$. If $x \geq \theta$ the output is 1 (the unit fires), otherwise the network does not fire.

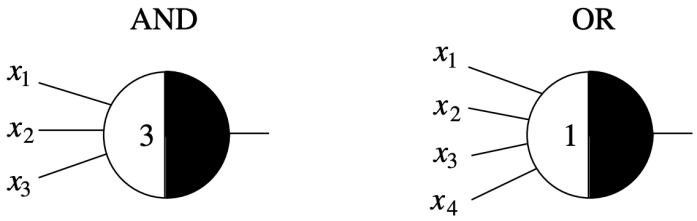
The thresholding operation can be seen as a step function which transforms a non-binary input to a binary output.



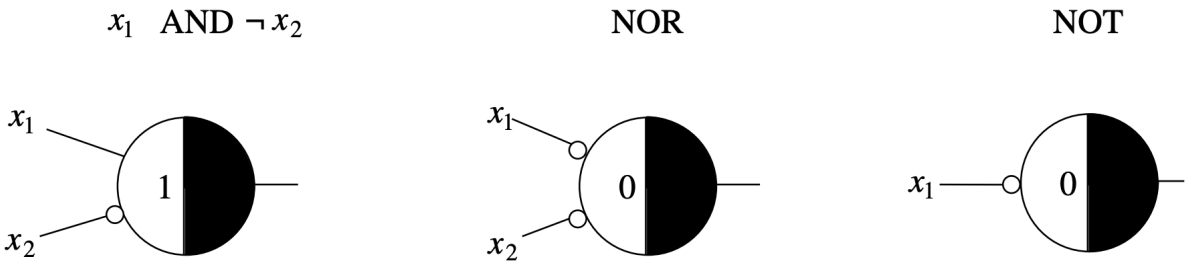
While this model can seem a simple one, an appropriate choice of the threshold θ allows to simply implement AND and OR gates:



This also generalizes to more than one input:



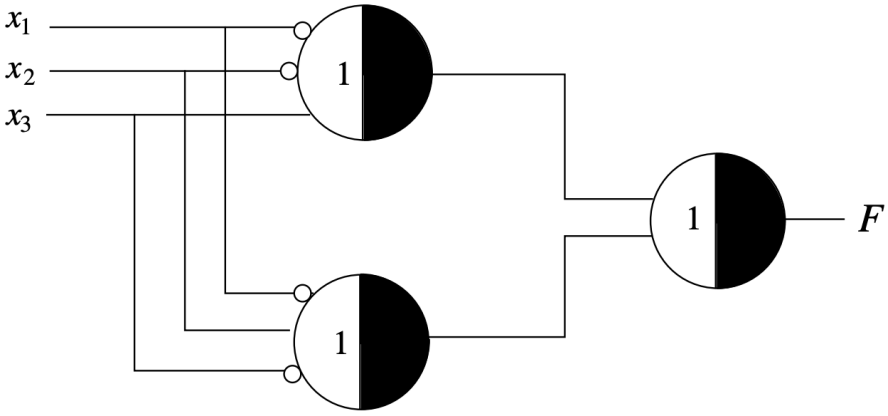
Using inhibitory connections (marked with a dot at the end of the edge) can allow to implement even more complex logical units:



Hence, we have a **parametric model** which can implement different functions with **appropriate choice of the parameters**. The parameters here are:

- The nature of each connection (excitatory or inhibitory).
- The value of the threshold θ.

McCulloch and Pitts also shown that stacking different units could allow to implement more complex functions, thus creating the first kinds of "artificial neural networks".



ROSENBLATT'S PERCEPTRON

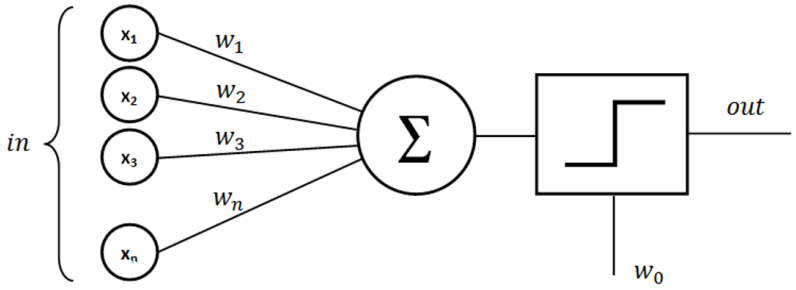
The McCulloch–Pitts neuron has different limitations:

1. It is designed to only handle binary inputs and perform logical operations. Hence, it is not well suited to be used with arbitrary features (e.g., real or integer numbers).
2. Parameters are fixed and should be picked by hand. This means that we can adapt the model to solve different problems, but we should know what the logical solution of the problem is in order to do so. Ideally, we would like to be able to find the appropriate choice for the parameters automatically (with a learning procedure) by looking at a series of examples.
3. Are all input equal? We cannot assign importance to the different input.
4. All functions computed are linearly separable.

In 1958, Rosenblatt proposes the "perceptron", a similar model which overcomes some of the limitations in two ways:

- Assigning weights to the connections. These weights will be multiplied by the input values. Since negative weights are possible, there is no need to distinguish between excitatory and inhibitory connections.
- Providing an effective learning algorithm for the perceptron which can find the appropriate weights of the model (the weights are parameters of the model) if they exist.

The perceptron can be illustrated as follows:



Here, $x_1, x_2, x_3, \dots, x_n$ are the inputs (they can be binary, integer, or real values), $w_1, w_2, w_3, \dots, w_n$ are the weights associated to the inputs, and w_0 is an additional weight which serves as a threshold (denoted as θ in the McCulloch-Pitts model).

The Σ symbol indicates that the model performs a sum of the input values multiplied by the weights, and the bias term. The step function allows to decide if the neuron should fire or not. The computation performed by this model can be written as follows:

$$f(x_1, \dots, x_n) = \begin{cases} 1 & if \sum_{i=1}^{n} x_i w_i > w_0 \\ 0 & otherwise \end{cases}$$

If we introduce an additional "dummy" input $x_0 = 1$, the weighted sum can be simplified as $\sum_{i=0}^{n} x_i w_i$can be rewritten as:

and the model

$$f(x_1, \dots, x_n) = \begin{cases} 1 & if \sum_{i=0}^{n} x_i w_i > 0 \\ 0 & otherwise \end{cases}$$

Note that, while one could think that the sign of w_0 should be changed in the transformation above, this is not necessary because w_0 is a weight which should be learned (we will eventually learn a negative value if necessary).

We can also write the model more compactly as follows:

$$f(\boldsymbol{x}) = [\boldsymbol{w}^T \boldsymbol{x} > 0]$$

Where [.] denotes the Iverson bracket:

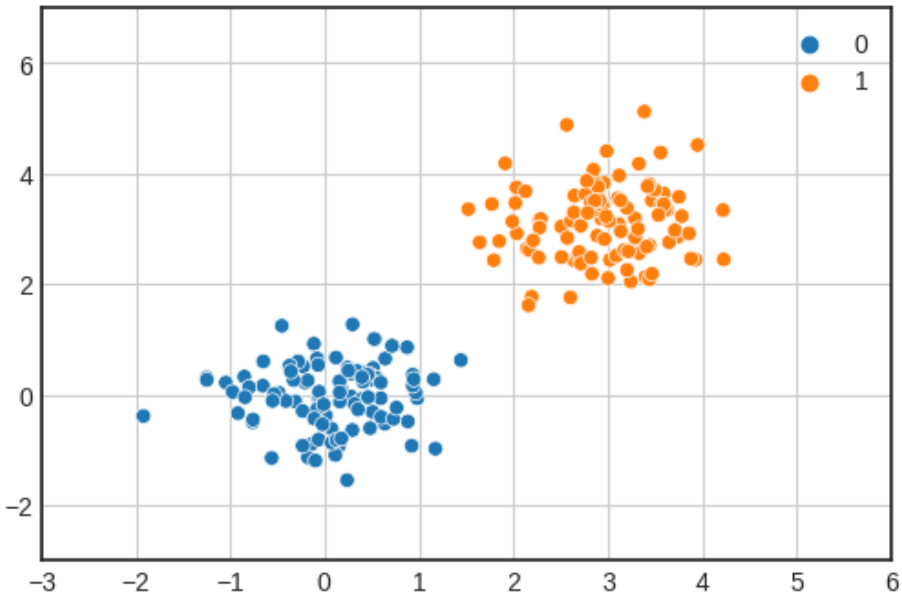
$$[\mathrm{z}] = \begin{cases} 1 \; if \; \mathrm{z} \; is \; true \\ 0 \; otherwise \end{cases}$$

Where $\boldsymbol{w} = (w_0, w_1, \dots, w_n)$ and $\boldsymbol{x} = (x_0, x_1, \dots, x_n)$.

GEOMETRIC INTERPRETATION

Perceptrons take as input feature vectors and output binary values. Hence, they can be seen as **binary classifiers** and this is indeed the first use of perceptrons ever demonstrated.

Consider a dataset of examples with two real features distributed as shown below:



Each example (also called a data point) $\boldsymbol{x} = (x_1, x_2)$ is shown in the plot as a 2D point in the Cartesian space. Points belong to two classes "0" (blue) and "1" (orange).

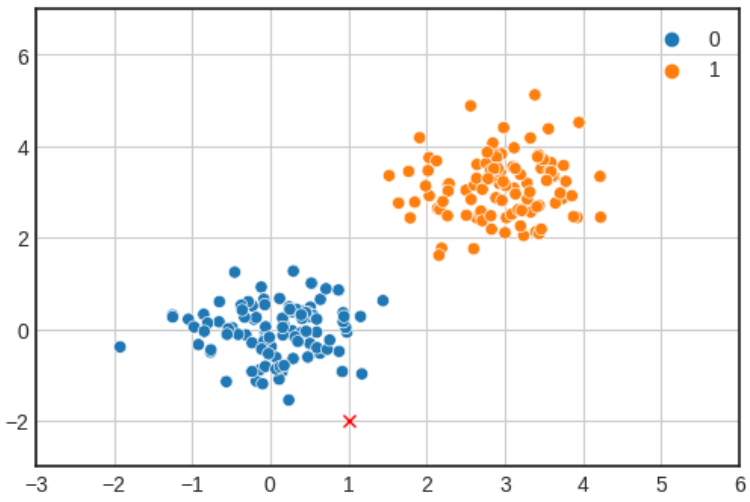
A perceptron can solve this binary classification problem by finding appropriate values for the three parameters (number of dimensions + 1 parameter for the threshold):

$$w_0, w_1, w_2$$

If we train (we will talk more about the learning algorithm later) a perceptron on this data, we may obtain the following solution:

$$\begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} -5.74 \\ 1.86 \\ 1.99 \end{pmatrix}$$

Let's consider a new point of features $(1, -2)$, shown below as a red cross:



For this new input, the perceptron will compute the following value:

$$f\big((1,-2)\big) = \begin{cases} 1 & if\ -5.74 + 1.86 \cdot 1 + 1.99 \cdot -2 > 0 \\ 0 & otherwise \end{cases} = \begin{cases} 1 & if\ -7.86 > 0 \\ 0 & otherwise \end{cases} = 0$$

Which is a reasonable result, considering that the red cross is closer to the blue points than it is to the orange ones. Ideally, we can perform this computation for a random set of points sampled from the space if we want to know how the perceptron works on the whole space that we are observing in the diagram above.

In practice, however, we can observe that the predicted class will be positive for the set of 2D points which satisfy the following inequality:

$$w_1 x_1 + w_2 x_2 + w_0 > 0$$

While the perceptron will classify as negative examples the 2D points which satisfy the following inequality:

$$w_1 x_1 + w_2 x_2 + w_0 < 0$$

We note that the classifier will be maximally uncertain when:

$$w_1 x_1 + w_2 x_2 + w_0 = 0$$

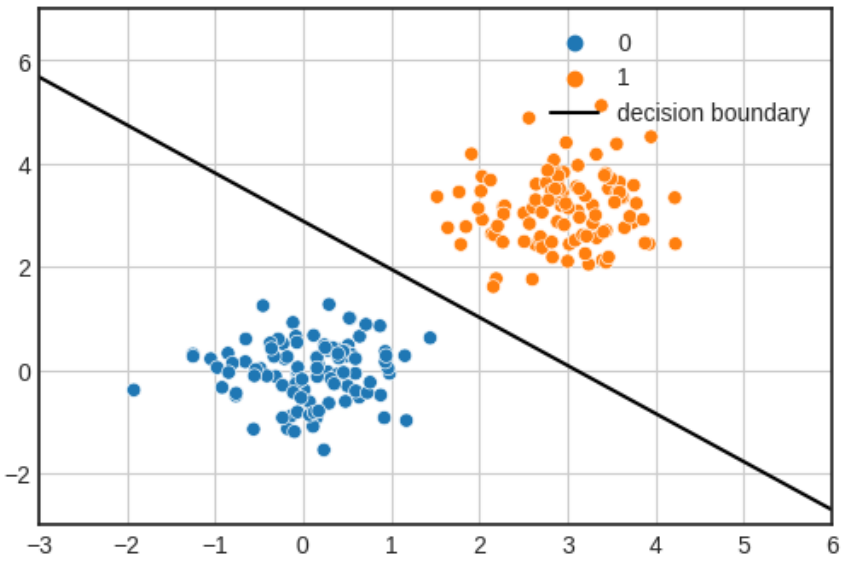
Or equivalently:

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

Replacing $\begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} -5.74 \\ 1.86 \\ 1.99 \end{pmatrix}$, we obtain:

$$x_2 = -0.95 x_1 + 2.88$$

Which is the equation of a line in a 2D space. If we plot this line on the data, we obtain the following:



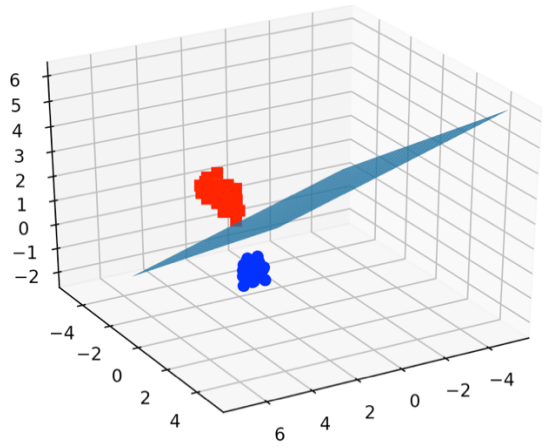
In practice, the perceptron separates the space into two areas:

- All the points above the line satisfy the inequality $w_1 x_1 + w_2 x_2 + w_0 > 0$ and will be classified as positive examples.

- All the points below the line satisfy the inequality $w_1x_1 + w_2x_2 + w_0 < 0$ and will be classified as negative examples.
- For all points which satisfy the equation $w_1x_1 + w_2x_2 + w_0 = 0$, the model is maximally uncertain. In this case, the points may be classified either as positives or negatives (it's a random guess anyway).

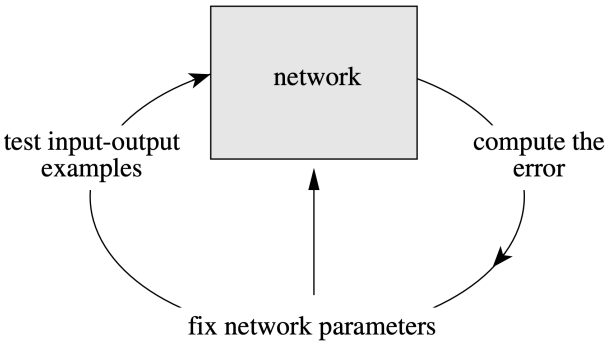
The line that we plot above is called **the decision boundary** as it indicates how the classifier has divided the space into areas.

While we have seen an example in 2D, this reasoning extends to multiple dimensions. In general, the decision boundary is a hyperplane in the n-dimensional space. For instance, in 3D, we could have a decision boundary like this:



LEARNING ALGORITHM

Rosenblatt's main innovation was to provide a training procedure for the perceptron. The proposed approach follows an iterative process in which the parameters of the model are first randomly initialized, then adjusted to reduce the number of mistakes performed on the training data. The process is repeated in an iterative fashion until no errors are made anymore.



More specifically, the algorithm works as follows:

1. Initialize the weights randomly with small values in the $[0,1]$ range and set a random threshold in the range of real numbers $\Re$:
   - $w_i \leftarrow RANDOM([0,1])$
   - $w_0 \leftarrow RANDOM(\Re)$
2. For each example i in the training set, perform these steps of the input $\boldsymbol{x}^{(i)}$:
   - Calculate the predicted label:
     $\hat{y}^{(i)} = f(\boldsymbol{x}^{(i)})$ (where f is the perceptron)
   - Update the weights:

$w_j = w_j + \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$, for all features $i = 0, \dots, n$ (where η is the learning rate)

3. Repeat 2 until no errors are made.

The learning rate regulates how big should the update be. A large value will make the model learn "faster", but its solution could diverge if a too large learning rate is used.

This algorithm is proved to converge to a solution if the input points are linearly separable. If they are not, the algorithm will never converge as it will never find a solution leading to zero errors.

ADALINE VARIATION

The above-described algorithm uses the difference between the predicted and ground truth perceptron output $\left(y^{(i)} - \hat{y}^{(i)}\right)$ to adjust the weight. The ADALINE algorithm uses the same algorithm, but instead uses the difference between the weighted sum of the inputs (pre-threshold) and the desired output. Hence the weight update rule becomes:

$$w_j = w_j + \eta\left(y^{(i)} - \sum_{j=0}^{n} w_j x_j^{(i)}\right)x_j^{(i)}$$

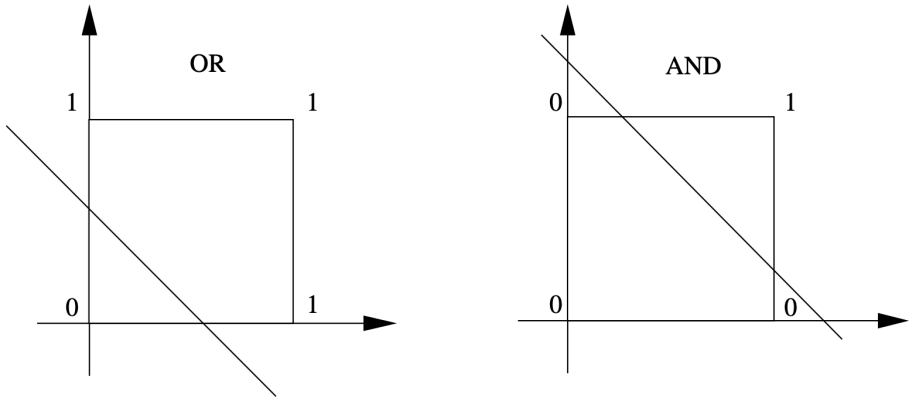
This allows to control the scale of the weight updates depending on how close the predictions are to the threshold.

THE XOR PROBLEM

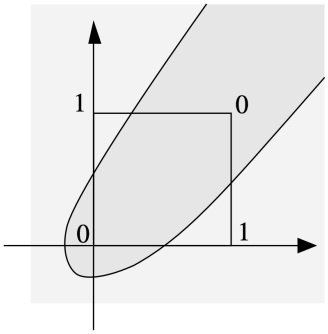
We have seen that an appropriate choice of weights allows a perceptron to find a hyperplane which separates the data into two classes. However, what happens if the data points cannot be separated by a line? To show what happens in this case, we will consider the problem of implementing some Boolean functions. Consider the following table showing the results of AND, OR, AND XOR operations:

x	y	x AND y	x OR y	x XOR y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

We can see the problem of predicting the AND, OR or XOR values form the x and y inputs as three different binary classification problems. It can be easily seen that perceptrons can find solutions (linear decision boundaries) for the AND and OR functions:



However, it is impossible to find a line separating the examples in the case of the XOR:



The figure shows a possible decision boundary which is clearly not linear. This decision boundary cannot be computed by a perceptron. Indeed, it would not be possible to represent that boundary by choosing appropriate weights for the inequality:

$$w_1 x_1 + w_2 x_2 + w_0 > 0$$

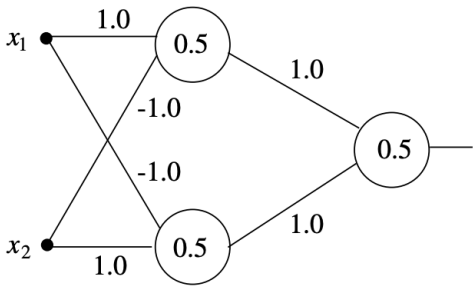
This example suggests that perceptrons cannot solve all kinds of problems. Indeed, a perceptron is a linear classifier and can solve only problems in which the training data are linearly separable. Linear separability can be defined as follows:

Two sets of points A and B in an n-dimensional space are called linearly separable if n+1 real numbers $w_1, \dots, w_{n+1}$ *exist, such that every point* $(x_1, x_2, \dots, x_n) \in A$ *satisfies* $\sum_{i=1}^{n} w_i x_i \geq w_{n+1}$ *and every point* $(y_1, y_2, \dots, y_n) \in$ B *satisfies* $\sum_{i=1}^{n} w_i y_i < w_{n+1}$.

Minsky and Papert (1969) demonstrated that the XOR problem cannot be solved by a perceptron.

XOR NETWORKS

While the XOR problem cannot be solved with a single perceptron, it has been shown that it can be solved by building a two-layered network of perceptron. In particular, it can be seen that the following network can be used to compute the XOR function:



The “0.5” numbers in the units represent the thresholds. Let’s break this down and see why this works. The top neuron in the first layer computes this function:

$$f_1(x_1, x_2) = [x_1 - x_2 > 0.5]$$

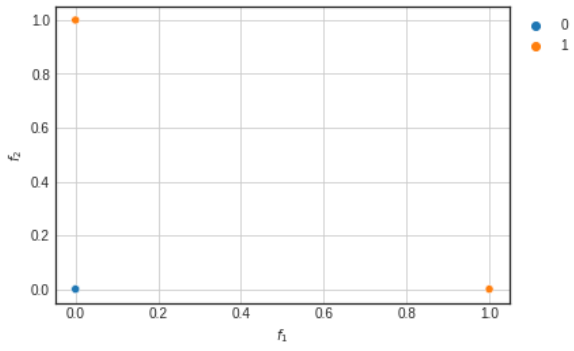
The bottom neuron in the first layer computes this function:

$$f_2(x_1, x_2) = [x_2 - x_1 > 0.5]$$

Let’s compute the results of these functions for all possible inputs:

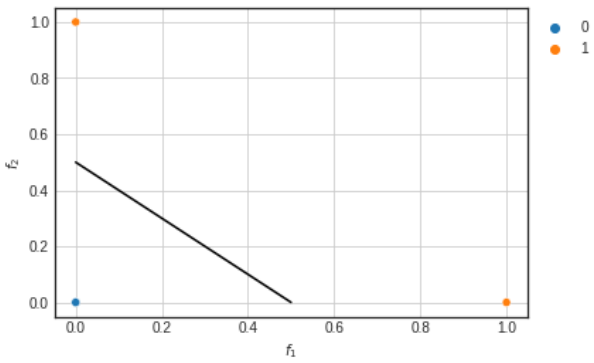
x	y	f1	f2	x XOR y
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

We have “transformed” the (x, y) pair into a (f_1, f_2) pair. Let’s plot this “transformed” dataset:



We note that now the data points are linearly separable! This has been possible by collapsing the inputs $(0,0)$ and $(1,1)$ of the same class “1”, to a single point $(0,0)$. If we use the weights of the final perceptron, we can plot the decision boundary:

$$f_2 = -f_1 + 0.5$$



Hence, the first layer had the role to transform the features in a way to make the problem easier and solvable by the subsequent perceptron.

Even if XOR networks can solve the XOR problem, the perceptron learning algorithm described before cannot be directly applied to these more complex networks. This observation made the first “AI winter” (a period of reduced interest and funding in AI) happen. We would have to wait till the formalization of the backpropagation algorithm to end the AI winter.

REFERENCES

From [1]: Section 2.1, Setion 2.2, Parts of Chapter 3, Parts of Chapter 4, Section 3.2.2, Section 3.3.,

[1] Rojas, R. (2013). Neural networks: a systematic introduction. Springer Science & Business Media. https://page.mi.fu-berlin.de/rojas/neural/neuron.pdf