Deep learning

3.1. The perceptron

François Fleuret

https://fleuret.org/dlc/

The first mathematical model for a neuron was the Threshold Logic Unit, with Boolean inputs and outputs:

$$f(x) = \mathbf{1}_{\{w \sum_i x_i + b \geq 0\}}.$$

It can in particular implement

$$\begin{aligned} or(u, v) &= \mathbf{1}_{\{u+v-0.5 \geq 0\}} && (w = 1, b = -0.5) \\ and(u, v) &= \mathbf{1}_{\{u+v-1.5 \geq 0\}} && (w = 1, b = -1.5) \\ not(u) &= \mathbf{1}_{\{-u+0.5 \geq 0\}} && (w = -1, b = 0.5) \end{aligned}$$

Hence, **any Boolean function can be build with such units.**

(McCulloch and Pitts, 1943)

The perceptron is very similar

$$f(x) = \begin{cases} 1 & \text{if} \quad \sum_i w_i \, x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

but the inputs are real valued and weights can be different (Rosenblatt, 1957).

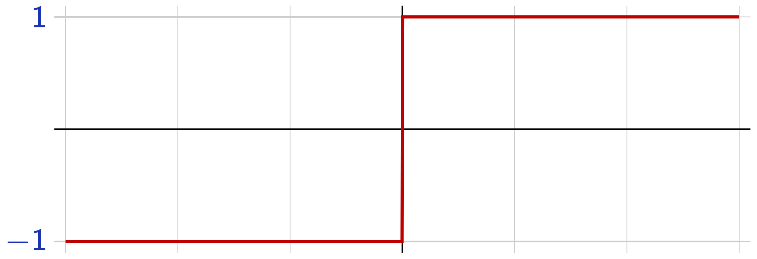
It was originally motivated by biology, with w_i being the *synaptic weights*, and x_i and f firing rates. However, it is a (very) crude biological model.

Notes

The perceptron extends the Threshold Logic Unit by to real-numbered inputs, and apply a different multiplicative weight to each.
This results in an affine expression $\sum_i w_i \, x_i + b = 0$, that defines an hyperplane, and the perceptron splits the input space in two subspaces, and responds 1 on one side of that hyperplane, and 0 on the other side.
Although this model was motivated by biology, it is an extremely crude model and does not reflect the complexity of real neurons which are a very complex machinery with a lot of chemical processing going on.

To make things simpler we take responses ± 1. Let

$$\sigma(x) = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

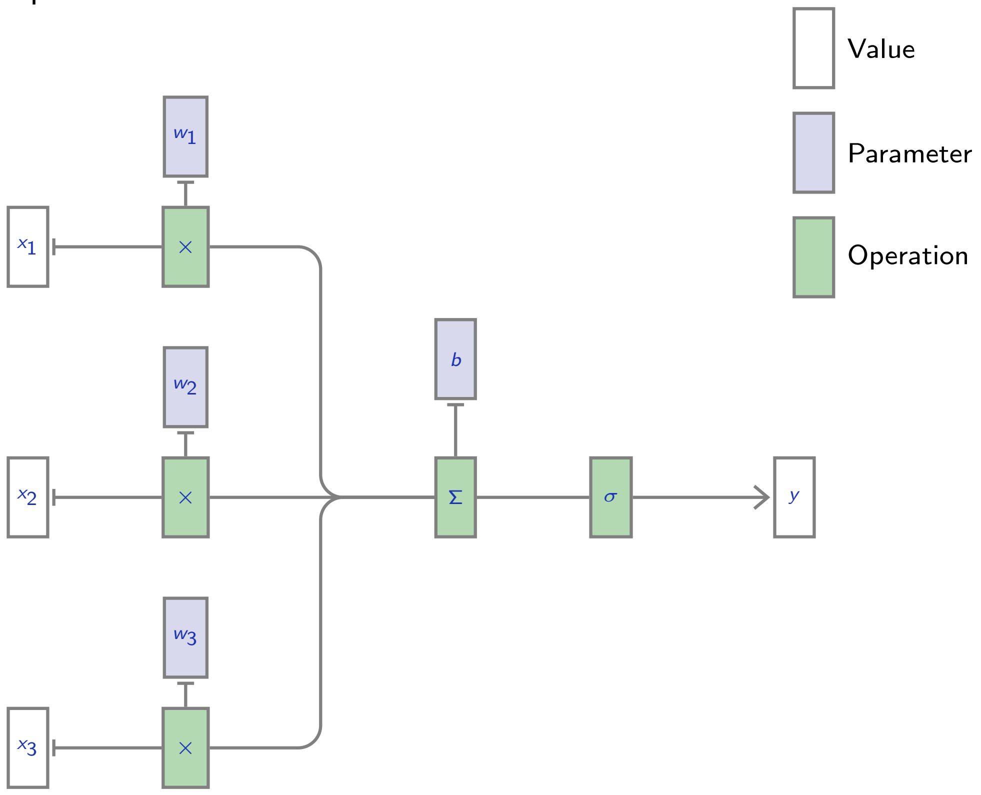


The perceptron classification rule boils down to

$$f(x) = \sigma(w \cdot x + b).$$

For neural networks, the function σ that follows a linear operator is called the **activation function**.

We can represent this "neuron" as follows:



Notes

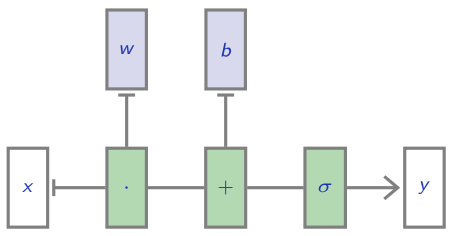
On this graph,

- (x_1, x_2, x_3) is the input to the neuron,
- (w_1, w_2, w_3) are its weights and b its bias,
- each $\times$ block computes a product, Σ a sum, and σ the non-linear activation function, resulting in the value y, which is the neuron's output.

As we will see, the neuron's parameters w_1, w_2, w_3, b are the quantities optimized during training.

We can also use tensor operations, as in

$$f(x) = \sigma(w \cdot x + b).$$



Notes

By analogy with the previous slides, we have:

- $x = (x_1, x_2, x_3)$,
- $w = (w_1, w_2, w_3)$,
- $b \in \mathbb{R}$ (as before)

Given a training set

$$(x_n, y_n) \in \mathbb{R}^D \times \{-1, 1\}, \quad n = 1, \ldots, N,$$

a very simple scheme to train such a linear operator for classification is the **perceptron algorithm:**

1. Start with $w^0 = 0$,
2. while $\exists n_k$ s.t. $y_{n_k} \left(w^k \cdot x_{n_k} \right) \leq 0$, update $w^{k+1} = w^k + y_{n_k} x_{n_k}$.

The bias b can be introduced as one of the ws by adding a constant component to x equal to 1.

Notes

To get an intuition of why the perceptron algorithm works, let's consider a misclassified sample x_n with label 1: we have $y_n(w \cdot x_n) \leq 0$. As long as this sample is misclassified, the weight vector gets updated by adding x_n, which by linearity adds $x_n \cdot x_n = \|x_n\|^2$ to the perceptron's response on x_n each time, which will be positive eventually.
For simplicity, the bias value can be introduced inside the weight vector:

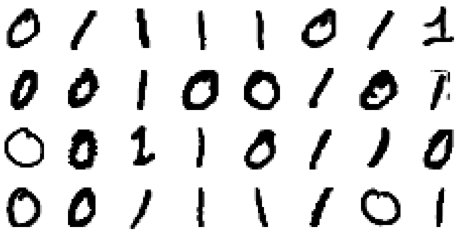
$$\sum_i w_i x_i + b = [w_1, \ldots, w_D, b] \cdot [x_1, \ldots, x_D, 1]$$

```
def train_perceptron(x, y, nb_epochs_max):
    w = torch.zeros(x.size(1))

    for e in range(nb_epochs_max):
        nb_changes = 0
        for i in range(x.size(0)):
            if x[i].dot(w) * y[i] <= 0:
                w = w + y[i] * x[i]
                nb_changes = nb_changes + 1
        if nb_changes == 0: break;

    return w
```

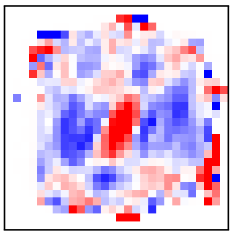
Notes

In the implementation, we have an argument to specify the maximum number of times where we loop through all the samples. It may happen that the algorithm never converges, in particular when there are no solution that separate properly the two populations, so we force the function to terminate before convergence.

This crude algorithm works often surprisingly well. With MNIST's "0"s as negative class, and "1"s as positive one.



```
epoch 0 nb_changes 64 train_error 0.23% test_error 0.19%
epoch 1 nb_changes 24 train_error 0.07% test_error 0.00%
epoch 2 nb_changes 10 train_error 0.06% test_error 0.05%
epoch 3 nb_changes 6 train_error 0.03% test_error 0.14%
epoch 4 nb_changes 5 train_error 0.03% test_error 0.09%
epoch 5 nb_changes 4 train_error 0.02% test_error 0.14%
epoch 6 nb_changes 3 train_error 0.01% test_error 0.14%
epoch 7 nb_changes 2 train_error 0.00% test_error 0.14%
epoch 8 nb_changes 0 train_error 0.00% test_error 0.14%
```



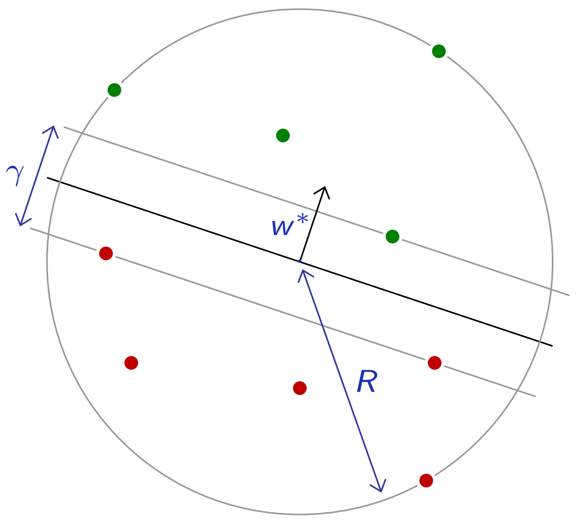
Notes

We can apply the perceptron algorithm to a simple computer vision classification problem, using the classes "0" and "1" from the MNIST dataset.
MNIST is a collection of 28×28 gray scale images. They can be "unfolded" into a 1d vector, by concatenating all the rows into one single vector of dimension 784.
The learned weight vector w can also be interpreted by reshaping it into an image of size 28×28. Since the weight vector has positive and negative values, we represent it with shades of blue for negative values, and shades of red for positive ones. The stronger the color, the larger the absolute value, white for zero.
Since the model here is linear, the 2d-reshaped vector w can be seen as a template that is applied on the input image: the dot product will sum the weights on the black pixel of the input image, and ignore the others. And indeed:

- the template has more positive weights in the center where the images of "1" have black pixels and not those of "0",
- the template has more negative weights on the left and right of the center, where images of "0" have black pixels but not those of "1".

Note that we reach a training error of 0%, meaning here that images of "0" and "1" of MNIST can be separated with an hyperplane.

We can get a convergence result under two assumptions:



1. The x_n are in a sphere of radius R:
$$\exists R > 0,\ \forall n,\ \|x_n\| \leq R.$$
2. The two populations can be separated with a margin γ:
$$\exists w^*,\ \|w^*\| = 1,\ \exists \gamma > 0,\ \forall n,\ y_n \left(x_n \cdot w^*\right) \geq \gamma/2.$$

To prove the convergence, let us make the assumption that there still is a misclassified sample at iteration k.

We have

$$\begin{aligned} w^{k+1} \cdot w^* &= \left(w^k + y_{n_k} x_{n_k}\right) \cdot w^* \\ &= w^k \cdot w^* + y_{n_k} \left(x_{n_k} \cdot w^*\right) \\ &\geq w^k \cdot w^* + \gamma/2 \\ &\geq (k+1)\,\gamma/2. \end{aligned}$$

Since

$$\|w^k\| \|w^*\| \geq w^k \cdot w^*,$$

we get

$$\begin{aligned} \|w^k\|^2 &\geq \left(w^k \cdot w^*\right)^2 / \|w^*\|^2 \\ &\geq k^2 \gamma^2 / 4. \end{aligned}$$

And

$$\begin{aligned}
\|w^{k+1}\|^2 &= w^{k+1} \cdot w^{k+1} \\
&= \left(w^k + y_{n_k} x_{n_k}\right) \cdot \left(w^k + y_{n_k} x_{n_k}\right) \\
&= w^k \cdot w^k + 2 \underbrace{y_{n_k}\, w^k \cdot x_{n_k}}_{\leq 0} + \underbrace{\|x_{n_k}\|^2}_{\leq R^2} \\
&\leq \|w^k\|^2 + R^2 \\
&\leq (k+1)\, R^2.
\end{aligned}$$

Notes

When a sample x_{n_k} is misclassified, by definition, we have $y_{n_k}\, w^k \cdot x_{n_k} \leq 0$.
With assumption 1 from previous slides, the samples are contained in a ball of radius R, so $\|x_{n_k}\|^2 \leq R^2$.

Putting these two results together, we get

$$k^2\gamma^2/4 \leq \|w^k\|^2 \leq \; k\,R^2$$

hence

$$k \leq 4R^2/\gamma^2,$$

hence no misclassified sample can remain after $\lfloor 4R^2/\gamma^2 \rfloor$ iterations.

This result makes sense:

- The bound does not change if the population is scaled, and
- the larger the margin, the more quickly the algorithm classifies all the samples correctly.

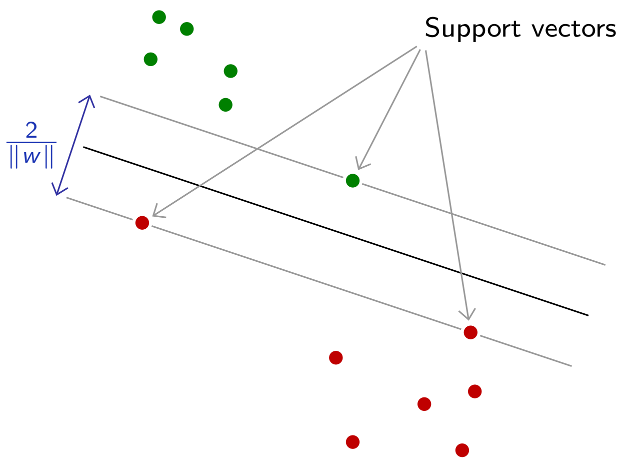
The perceptron stops as soon as it finds a separating boundary. Other algorithms maximize the distance of samples to the decision boundary, which improves robustness to noise.

Support Vector Machines (SVM) achieve this by minimizing

$$\mathscr{L}(w, b) = \lambda \|w\|^2 + \frac{1}{N} \sum_n \max(0, 1 - y_n(w \cdot x_n + b)),$$

which is convex and has a global optimum.

$$\mathscr{L}(w, b) = \lambda \|w\|^2 + \frac{1}{N} \sum_n \max(0, 1 - y_n(w \cdot x_n + b))$$



Minimizing $\max(0, 1 - y_n(w \cdot x_n + b))$ pushes the nth sample beyond the plane $w \cdot x + b = y_n$, and minimizing $\|w\|^2$ increases the distance between the $w \cdot x + b = \pm 1$.

At convergence, only a small number of samples matter, the "support vectors".

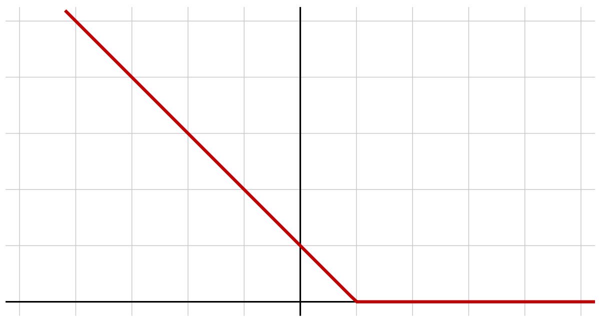
Notes

The boundary is only defined by support vectors, the points which actually matter to characterize the boundary between the two populations.

The term

$$\max(0, 1 - \alpha)$$

is the so called "hinge loss"



References

W. S. McCulloch and W. Pitts. **A logical calculus of the ideas immanent in nervous activity**. The bulletin of mathematical biophysics, 5(4):115–133, 1943.

F. Rosenblatt. **The perceptron–A perceiving and recognizing automaton**. Technical Report 85-460-1, Cornell Aeronautical Laboratory, 1957.