EE-559 – Deep learning

3.1. The perceptron

François Fleuret

https://fleuret.org/ee559/

Sun Nov 11 21:46:43 UTC 2018

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

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

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

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

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

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

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 values and the weights can be different.

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 values and the weights can be different.

This model was originally motivated by biology, with w_i being the *synaptic weights*, and x_i and f firing rates.

It is a (very) crude biological model.

(Rosenblatt, 1957)

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

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

To make things simpler we take responses ± 1. Let

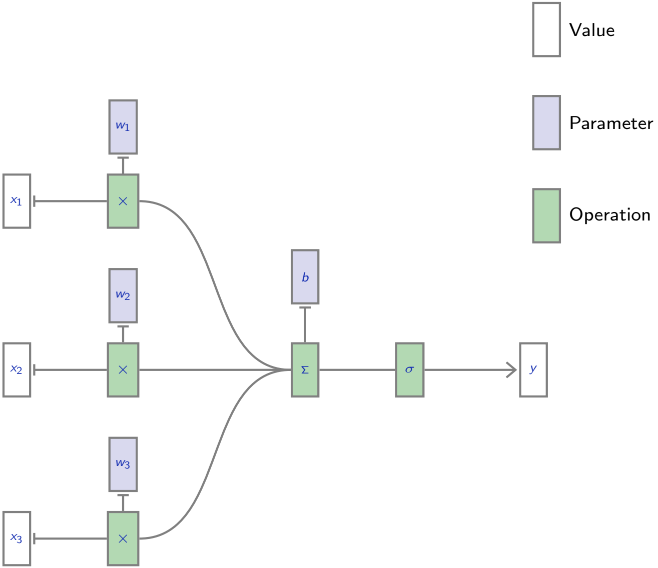
$$\sigma(x) = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

The perceptron classification rule boils down to

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

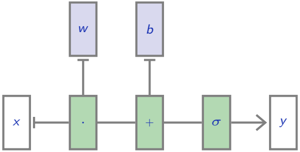
For neural networks, the function σ that follows a linear operator is called the **activation function**.

We can represent this "neuron" as follows:



We can also use tensor operations, as in

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



Given a training set

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

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

Given a training set

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

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 $w^{k+1} = w^k + y_{n_k} x_{n_k}$.

Given a training set

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

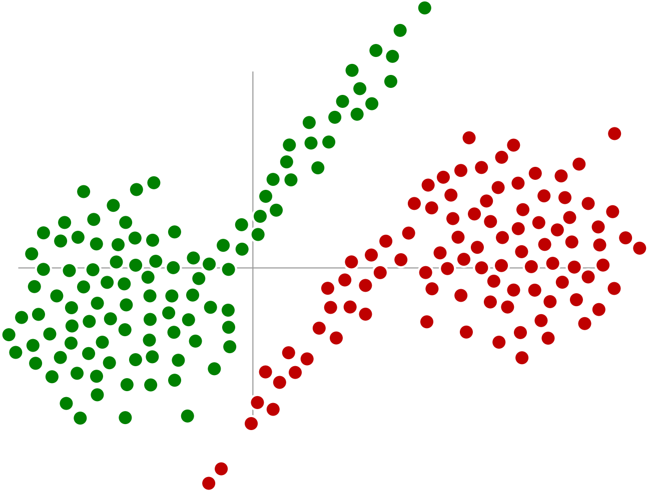
a very simple scheme to train such a linear operator for classification is the **perceptron algorithm:**

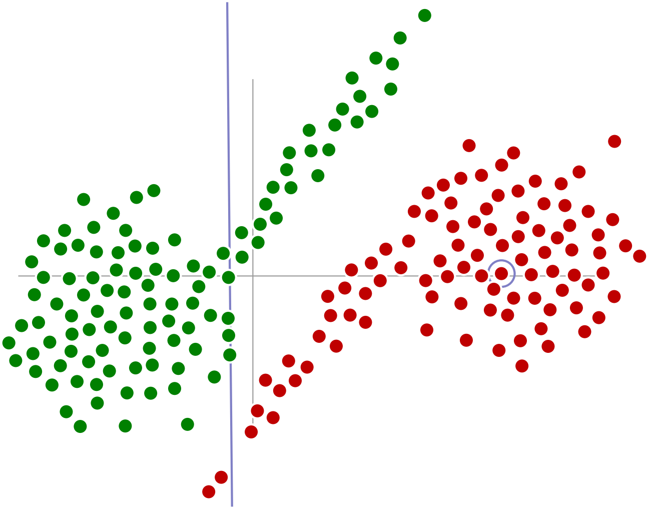
1. Start with $w^0 = 0$,
2. while $\exists n_k$ s.t. $y_{n_k}\left(w^k \cdot x_{n_k}\right) \leq 0$, update $w^{k+1} = w^k + y_{n_k} x_{n_k}$.

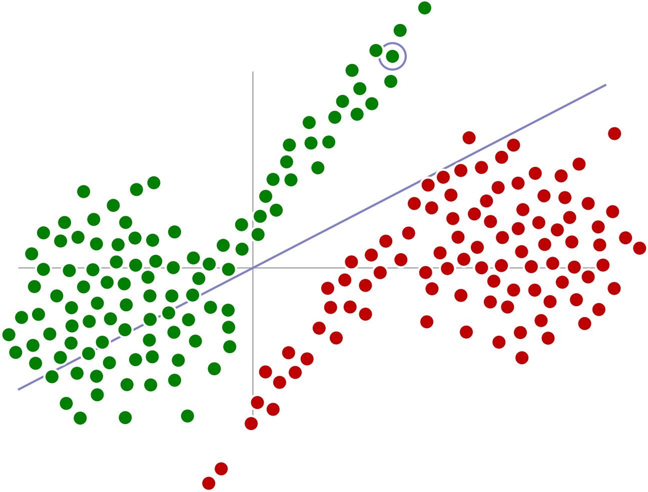
The bias b can be introduced as one of the ws by adding a constant component to x equal to 1.

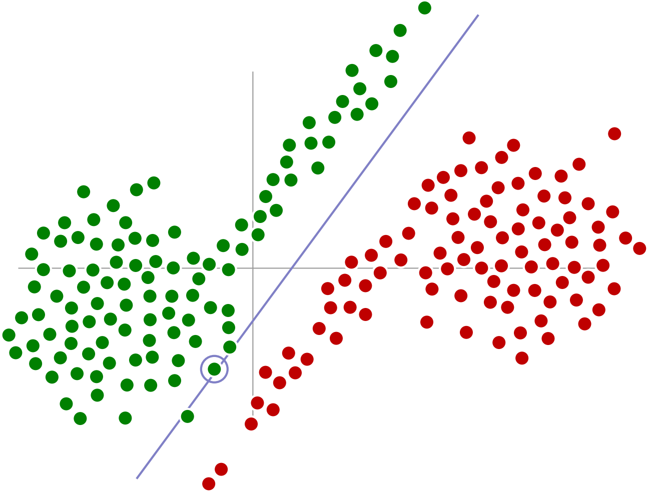
(Rosenblatt, 1957)

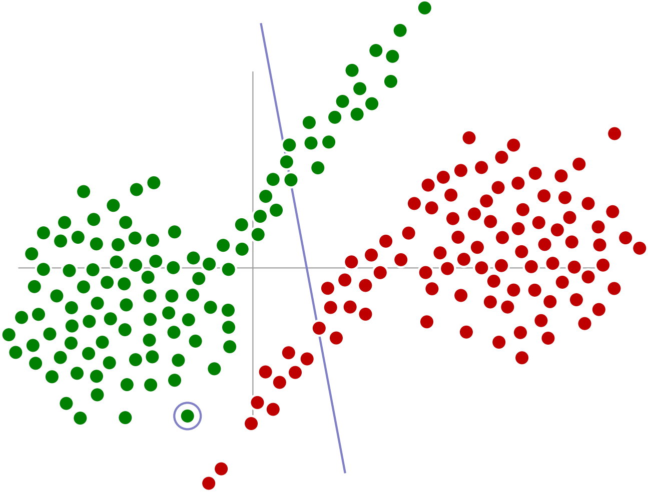
```
def train_perceptron(x, y, nb_epochs_max):
    w = torch.zeros(x.size(1))

    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
```

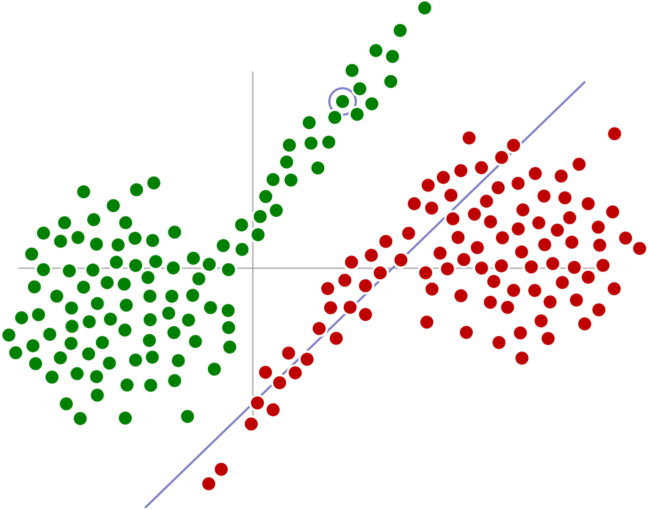


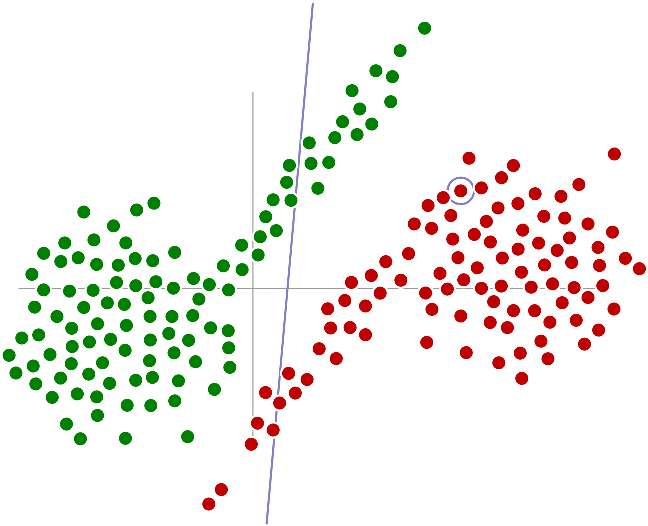


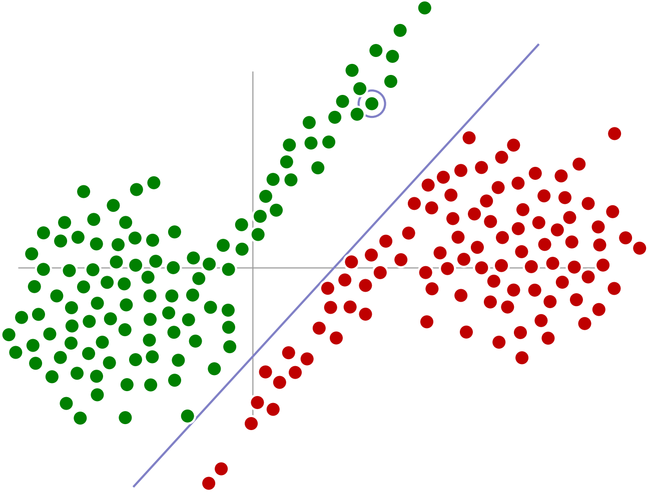




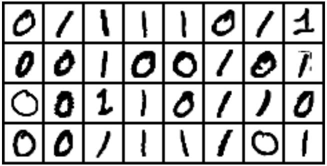




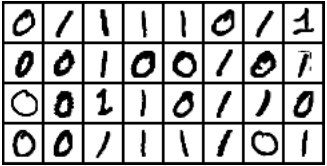




This crude algorithm works often surprisingly well. With MNIST's "0"s as negative class, and "1"s as positive one.

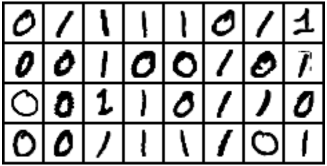


This crude algorithm works often surprisingly well. With MNIST's "0"s as negative class, and "1"s as positive one.

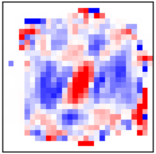


```
epoch 0 nb_changes 64 train_error 0.23% test_error 0.19%
epoch 1 nb_changes 24 train_error 0.07% test_error 0.00%
epoch 2 nb_changes 10 train_error 0.06% test_error 0.05%
epoch 3 nb_changes 6 train_error 0.03% test_error 0.14%
epoch 4 nb_changes 5 train_error 0.03% test_error 0.09%
epoch 5 nb_changes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

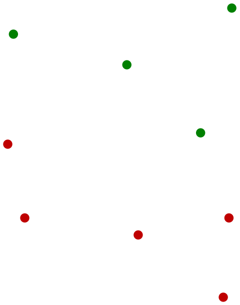
This crude algorithm works often surprisingly well. With MNIST's "0"s as negative class, and "1"s as positive one.



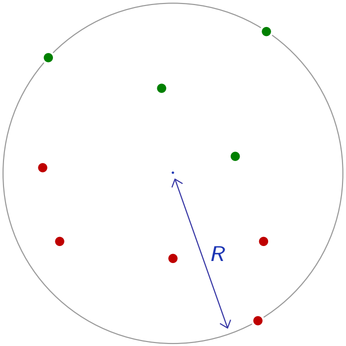
```
epoch 0 nb_changes 64 train_error 0.23% test_error 0.19%
epoch 1 nb_changes 24 train_error 0.07% test_error 0.00%
epoch 2 nb_changes 10 train_error 0.06% test_error 0.05%
epoch 3 nb_changes 6 train_error 0.03% test_error 0.14%
epoch 4 nb_changes 5 train_error 0.03% test_error 0.09%
epoch 5 nb_changes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



We can get a convergence result under two assumptions:



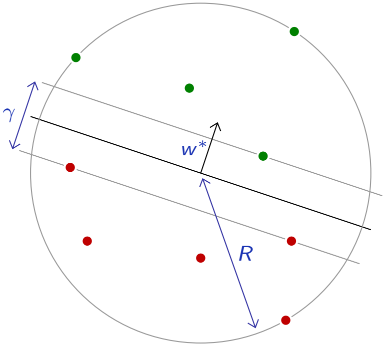
We can get a convergence result under two assumptions:



1. The x_n are in a sphere of radius R:

$$\exists R > 0,\ \forall n,\ \|x_n\| \leq R.$$

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 $\gamma > 0$.
$$\exists w^*, \ \|w^*\| = 1, \ \exists \gamma > 0, \ \forall n, \ y_n \left(x_n \cdot w^*\right) \geq \gamma/2.$$

To prove the convergence, let us make the assumption that there still is a misclassified sample at iteration k, and w^{k+1} is the weight vector updated with it. We have

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

To prove the convergence, let us make the assumption that there still is a misclassified sample at iteration k, and w^{k+1} is the weight vector updated with it. 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

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

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 $\left\lfloor 4R^2/\gamma^2 \right\rfloor$ iterations.

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 $\left\lfloor 4R^2/\gamma^2 \right\rfloor$ 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.

The perceptron stops as soon as it finds a separating boundary.

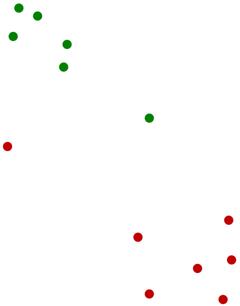
Other algorithms maximize the distance of samples to the decision boundary, which improves robustness to noise.

Support Vector Machines (SVM) achieve this by minimizing

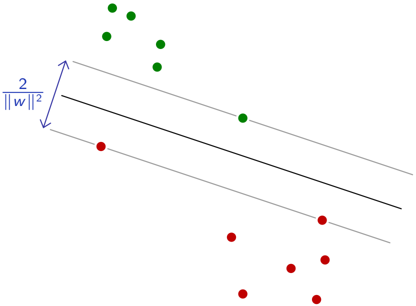
$$\mathscr{L}(w, b) = \lambda \|w\|^2 + \frac{1}{N} \sum_n \max(0, 1 - y_n(w \cdot x_n + b)),$$

which is convex and has a global optimum.

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

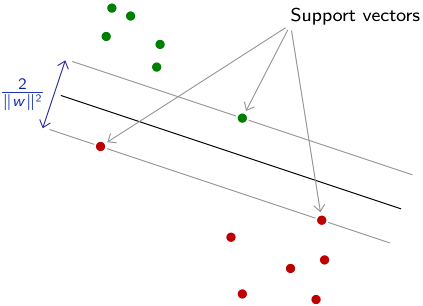


$$\mathscr{L}(w, b) = \lambda \|w\|^2 + \frac{1}{N} \sum_n \max(0, 1 - y_n(w \cdot x_n + b))$$



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 $w \cdot x + b = \pm 1$.

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



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".

The term

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

is the so called "hinge loss"

The end

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.