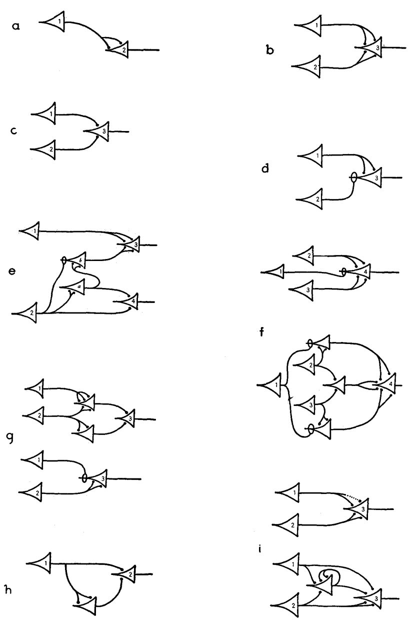
EE-613 – Machine Learning for Engineers

The Multi-Layer Perceptron

François Fleuret
https://www.idiap.ch/~odobez/teaching-epfl-ee613-2019.php
Thu Nov 21, 2019



Networks of “Threshold Logic Unit”

(McCulloch and Pitts, 1943)

1949 – Donald Hebb proposes the Hebbian Learning principle.

1951 – Marvin Minsky creates the first ANN (Hebbian learning, 40 neurons).

1958 – Frank Rosenblatt creates a perceptron to classify 20×20 images.

1959 – David H. Hubel and Torsten Wiesel demonstrate orientation selectivity and columnar organization in the cat's visual cortex.

1982 – Paul Werbos proposes back-propagation for ANNs.

The perceptron

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 values and the weights can be different.

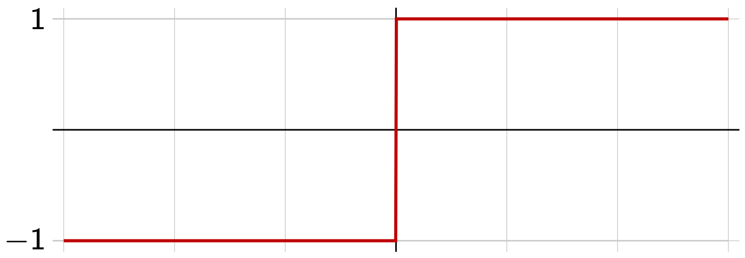
This model was originally motivated by biology, with w_i being the *synaptic weights*, and x_i and f firing rates.

It is a (very) crude biological model.

(Rosenblatt, 1957)

To make things simpler we take responses ± 1. Let

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

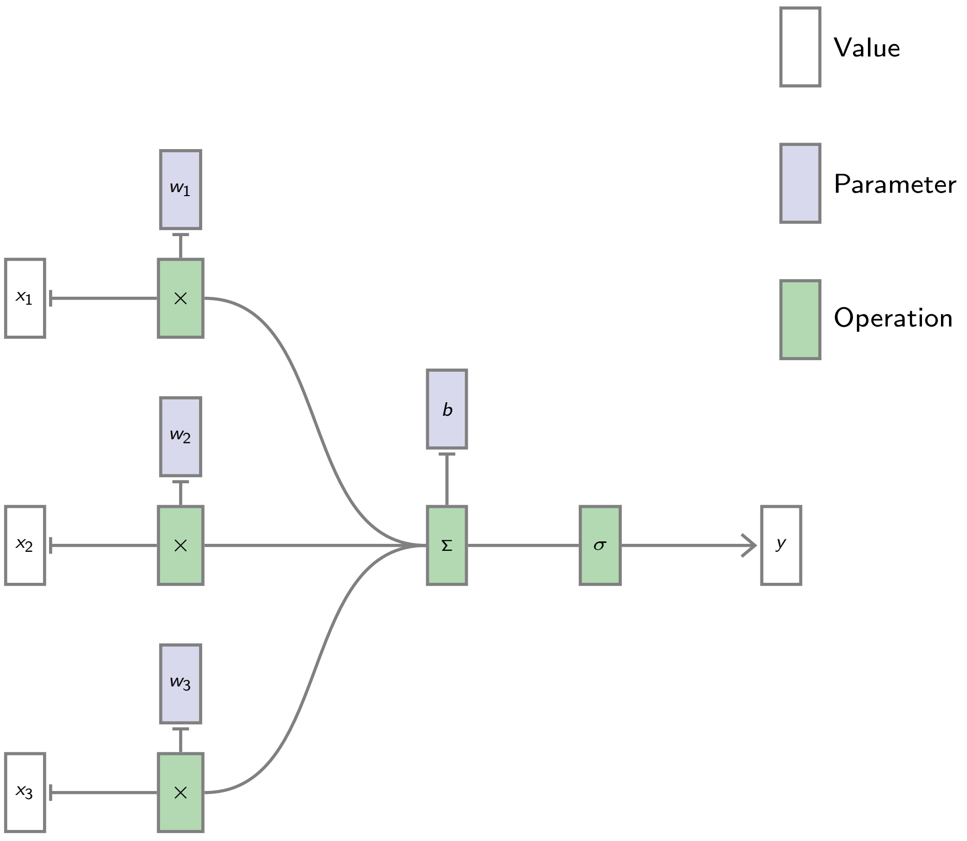


The perceptron classification rule boils down to

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

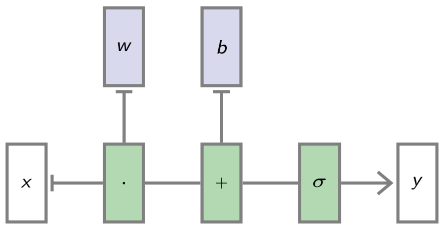
For neural networks, the function σ that follows a linear operator is called the **activation function**.

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:**

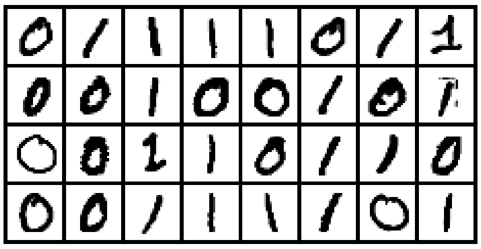
1. Start with $w^0 = 0$,
2. while $\exists n_k$ s.t. $y_{n_k}\left(w^k \cdot x_{n_k}\right) \leq 0$, update $w^{k+1} = w^k + y_{n_k} x_{n_k}$.

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

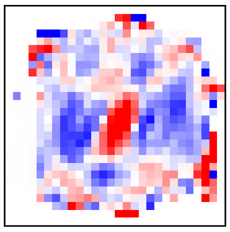
(Rosenblatt, 1957)

```
def train_perceptron(x, y, nb_epochs_max):
    w = np.zeros((x.shape[1],))

    for e in range(nb_epochs_max):
        nb_changes = 0
        for i in range(x.shape[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
```

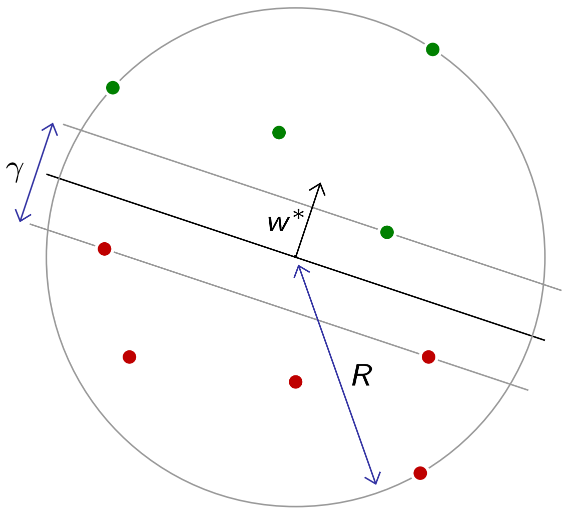
This crude algorithm works often surprisingly well. With MNIST's "0"s as negative class, and "1"s as positive one.



```
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:



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

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

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.

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.

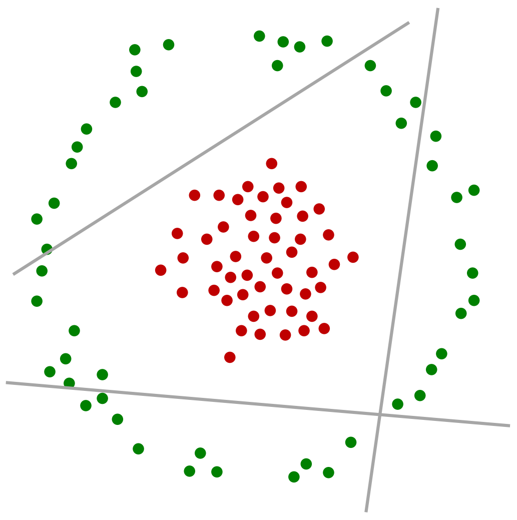
Support Vector Machines (SVM) achieve this by minimizing

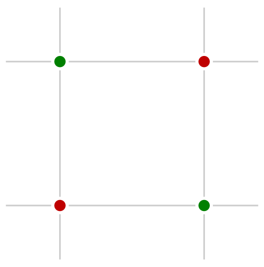
$$\mathscr{L}(w, b) = \lambda \|w\|^2 + \frac{1}{N} \sum_n \max(0, 1 - y_n(w \cdot x_n + b)),$$

which is convex and has a global optimum.

Limitation of linear models

The main weakness of linear predictors is their lack of capacity. For classification, the populations have to be **linearly separable**.

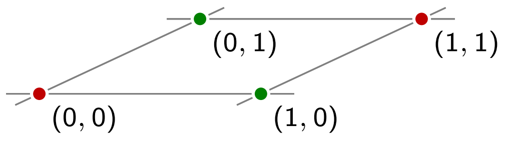


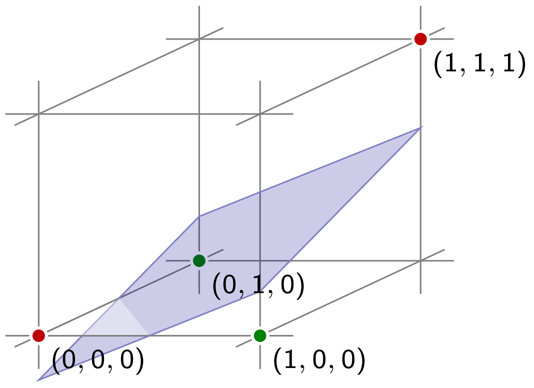


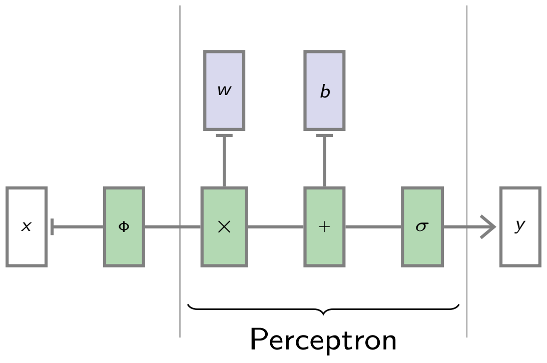
“xor”

The xor example can be solved by pre-processing the data to make the two populations linearly separable.

$$\Phi : (x_u, x_v) \mapsto (x_u, x_v, x_u x_v).$$







This is similar to the polynomial regression. If we have

$$\Phi : x \mapsto (1, x, x^2, \dots, x^D)$$

and

$$\alpha = (\alpha_0, \dots, \alpha_D)$$

then

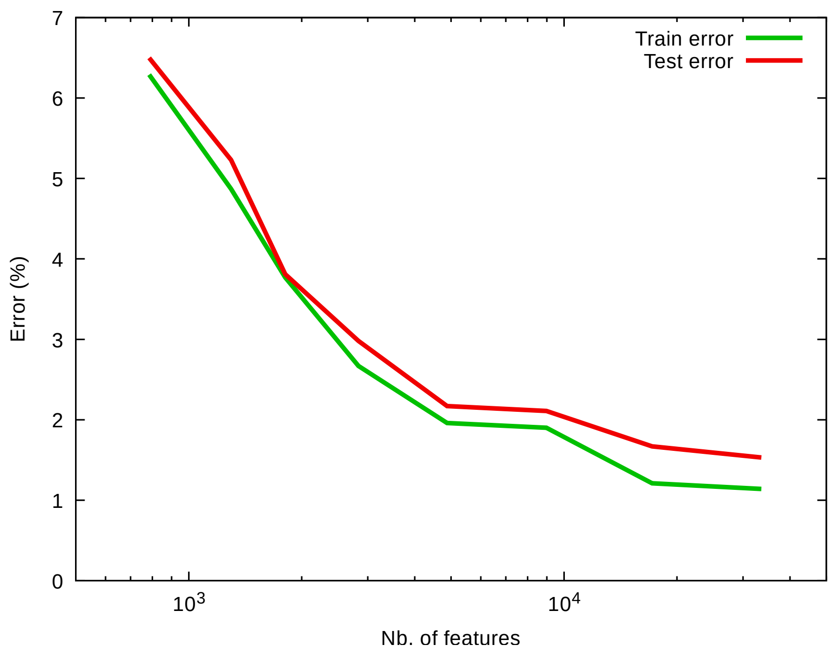
$$\sum_{d=0}^{D} \alpha_d x^d = \alpha \cdot \Phi(x).$$

By increasing D, we can approximate any continuous real function on a compact space (Stone-Weierstrass theorem).

It means that we can make the capacity as high as we want.

We can apply the same to a more realistic binary classification problem: MNIST's “8” vs. the other classes with a perceptron.

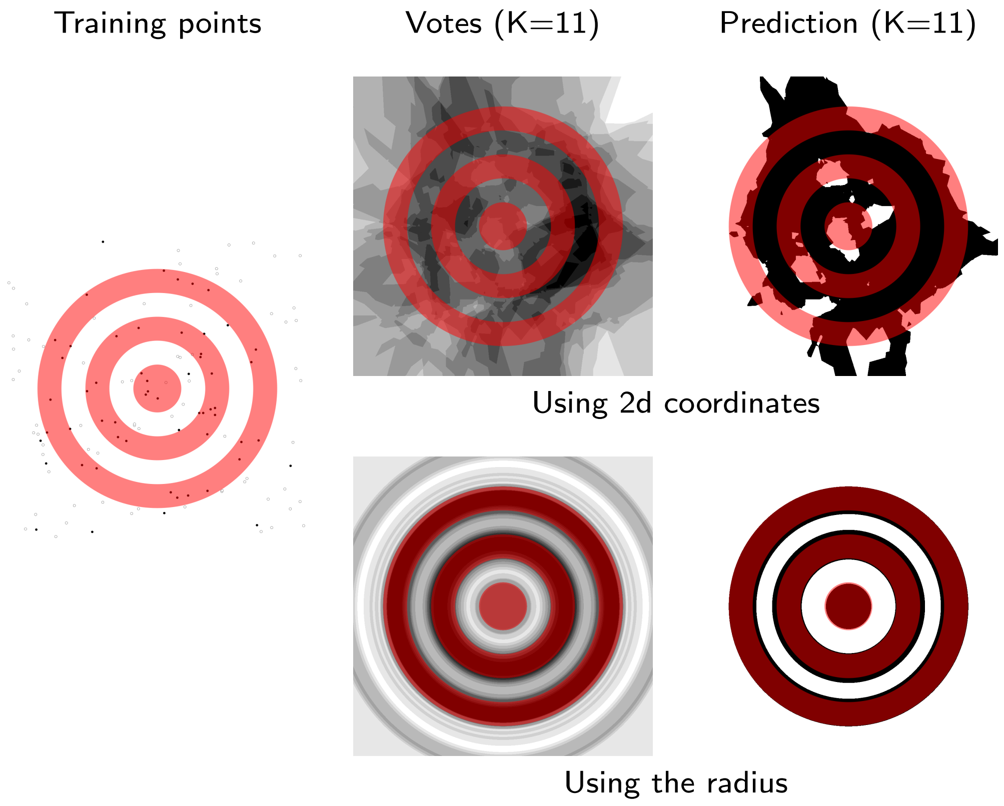
The original 28×28 features are supplemented with the products of pairs of features taken at random.



Beside increasing capacity to fit the data better, “feature design” may also be a way of reducing capacity without hurting the bias, or with improving it.

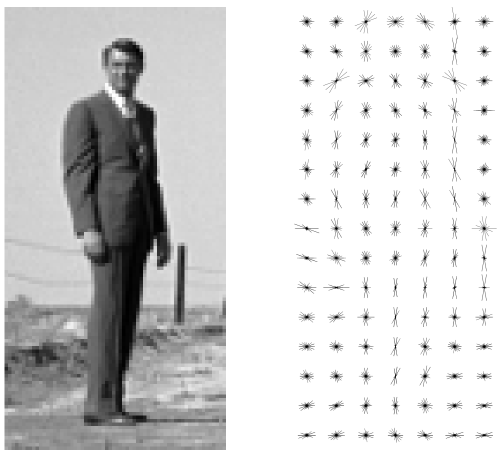
In particular, good features should be invariant to perturbations of the signal known to keep the value to predict unchanged.

We can illustrate the use of features with k-NN on a task with radial symmetry. Using the radius instead of 2d coordinates allows to cope with label noise.



A classical example is the “Histogram of Oriented Gradient” descriptors (HOG), initially designed for person detection.

Roughly: divide the image in 8×8 blocks, compute in each the distribution of edge orientations over 9 bins.



Dalal and Triggs (2005) combined them with a SVM, and Dollár et al. (2009) extended them with other modalities into the “channel features”.

Training a model composed of manually engineered features and a parametric model such as logistic regression is now referred to as **“shallow learning”**.

The signal goes through a single processing trained from data.

Deep models

A linear classifier of the form

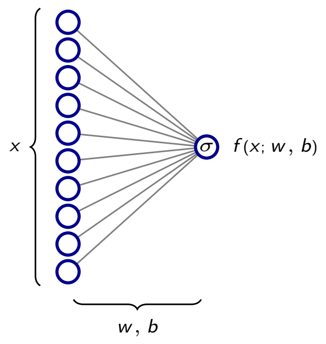
$$\begin{aligned} \mathbb{R}^D &\to \mathbb{R} \\ x &\mapsto \sigma(w \cdot x + b), \end{aligned}$$

with $w \in \mathbb{R}^D$, $b \in \mathbb{R}$, and $\sigma : \mathbb{R} \to \mathbb{R}$, can naturally be extended to a multi-dimension output by applying a similar transformation to every output

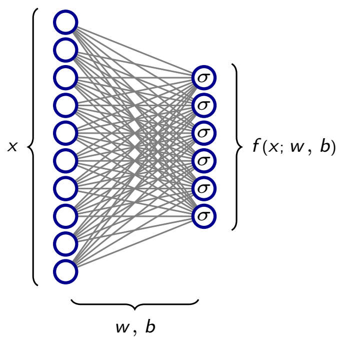
$$\begin{aligned} \mathbb{R}^D &\to \mathbb{R}^C \\ x &\mapsto \sigma(wx + b), \end{aligned}$$

with $w \in \mathbb{R}^{C \times D}$, $b \in \mathbb{R}^C$, and σ is applied component-wise.

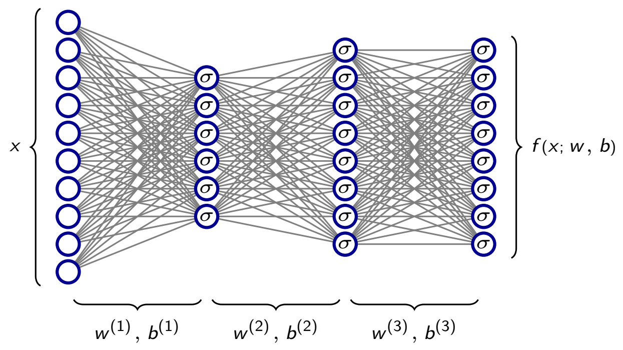
Even though it has no practical value implementation-wise, we can represent such a model as a combination of units. More importantly, we can extend it.



Single unit



One layer of units

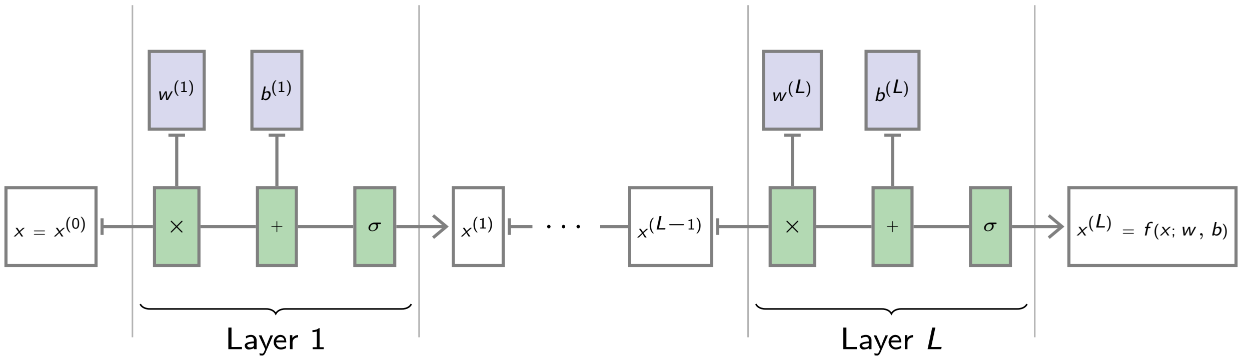


Multiple layers of units

This latter structure can be formally defined, with $x^{(0)} = x$,

$$\forall l = 1, \dots, L,\ x^{(l)} = \sigma\left(w^{(l)} x^{(l-1)} + b^{(l)}\right)$$

and $f(x; w, b) = x^{(L)}$.



Such a model is a **Multi-Layer Perceptron (MLP).**

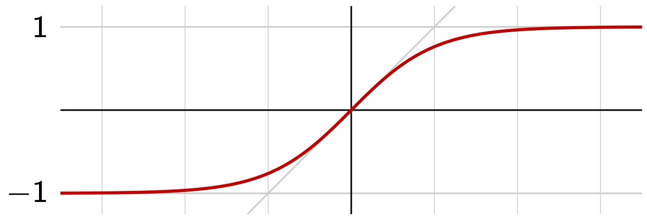
Note that if σ is an affine transformation, the full MLP is a composition of affine mappings, and itself an affine mapping.

Consequently:

⚠ **The activation function σ should be non-linear**, or the resulting MLP is an affine mapping with a peculiar parametrization.

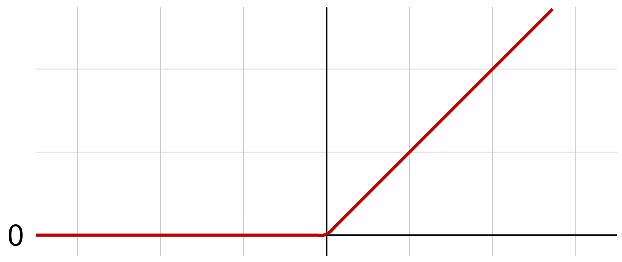
The two classical activation functions are the hyperbolic tangent

$$x \mapsto \frac{2}{1+e^{-2x}} - 1$$



and the rectified linear unit (ReLU)

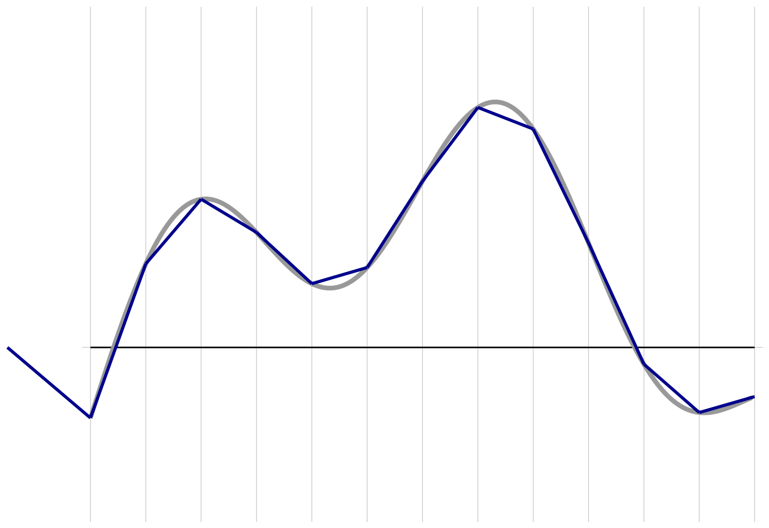
$$x \mapsto \max(0, x)$$



Universal approximation

We can approximate any $\psi \in \mathscr{C}([a,b],\mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



This is true for other activation functions under mild assumptions.

Extending this result to any $\psi \in \mathscr{C}([0,1]^D,\mathbb{R})$ requires a bit of work.

First, we can use the previous result for the sin function

$$\forall A > 0, \epsilon > 0,\ \exists N,\ (\alpha_n, a_n) \in \mathbb{R} \times \mathbb{R}, n = 1, \dots, N,$$
$$\text{s.t.} \max_{x \in [-A,A]} \left| \sin(x) - \sum_{n=1}^{N} \alpha_n \sigma(x - a_n) \right| \leq \epsilon.$$

And the density of Fourier series provides

$$\forall \psi \in \mathscr{C}([0,1]^D,\mathbb{R}), \delta > 0, \exists M, (v_m, \gamma_m, c_m) \in \mathbb{R}^D \times \mathbb{R} \times \mathbb{R}, m = 1, \dots, M,$$
$$\text{s.t.} \max_{x \in [0,1]^D} \left| \psi(x) - \sum_{m=1}^{M} \gamma_m \sin(v_m \cdot x + c_m) \right| \leq \delta.$$

So, $\forall \xi > 0$, with

$$\delta = \frac{\xi}{2}, A = \max_{1 \le m \le M} \max_{x \in [0,1]^D} |v_m \cdot x + c_m|, \text{ and } \epsilon = \frac{\xi}{2 \sum_m |\gamma_m|}$$

we get, $\forall x \in [0,1]^D$,

$$\left| \psi(x) - \sum_{m=1}^{M} \gamma_m \left(\sum_{n=1}^{N} \alpha_n \sigma(v_m \cdot x + c_m - a_n) \right) \right|$$

$$\le \underbrace{\left| \psi(x) - \sum_{m=1}^{M} \gamma_m \sin(v_m \cdot x + c_m) \right|}_{\le \frac{\xi}{2}}$$

$$+ \underbrace{\sum_{m=1}^{M} |\gamma_m| \underbrace{\left| \sin(v_m \cdot x + c_m) - \sum_{n=1}^{N} \alpha_n \sigma(v_m \cdot x + c_m - a_n) \right|}_{\le \frac{\xi}{2 \sum_m |\gamma_m|}}}_{\le \frac{\xi}{2}}$$

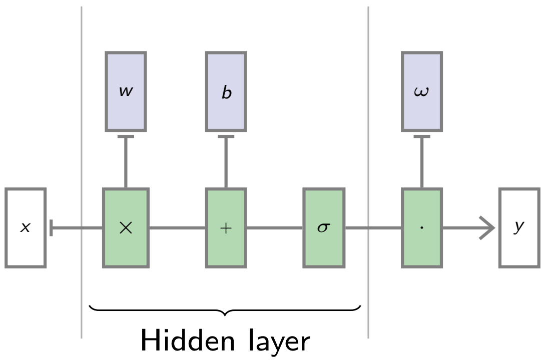
So we can approximate any continuous function

$$\psi : [0,1]^D \to \mathbb{R}$$

with a one hidden layer perceptron

$$x \mapsto \omega \cdot \sigma(w\, x + b),$$

where $b \in \mathbb{R}^K$, $w \in \mathbb{R}^{K \times D}$, and $\omega \in \mathbb{R}^K$.



This is the **universal approximation theorem.**

A better approximation requires a larger hidden layer (larger K), and this theorem says nothing about the relation between the two.

Gradient descent

Training consists of finding the model parameters minimizing an empirical risk or loss, for instance the mean-squared error (MSE)

$$\mathscr{L}(w, b) = \frac{1}{N} \sum_n (f(x_n; w, b) - y_n)^2 .$$

Other losses are more fitting for classification, certain regression problems, or density estimation. We will come back to this.

The loss may be minimized with an analytic solution for the MSE, or with *ad hoc* recipes for the empirical error rate (*e.g.* perceptron).

There is generally no *ad hoc* method. The logistic regression for instance

$$P_w(Y = 1 \mid X = x) = \sigma(w \cdot x + b), \text{ with } \sigma(x) = \frac{1}{1 + e^{-x}}$$

leads to the loss

$$\mathscr{L}(w, b) = -\sum_n \log \sigma(y_n(w \cdot x_n + b))$$

which cannot be minimized analytically.

The general minimization method used in such a case is the **gradient descent**.

Given a functional

$$\begin{aligned} f : \mathbb{R}^D &\to \mathbb{R} \\ x &\mapsto f(x_1, \dots, x_D), \end{aligned}$$

its gradient is the mapping

$$\begin{aligned} \nabla f : \mathbb{R}^D &\to \mathbb{R}^D \\ x &\mapsto \left(\frac{\partial f}{\partial x_1}(x), \dots, \frac{\partial f}{\partial x_D}(x) \right). \end{aligned}$$

To minimize a functional

$$\mathscr{L} : \mathbb{R}^D \to \mathbb{R}$$

the gradient descent uses local linear information to iteratively move toward a (local) minimum.

For $w_0 \in \mathbb{R}^D$, consider an approximation of $\mathscr{L}$ around w_0

$$\tilde{\mathscr{L}}_{w_0}(w) = \mathscr{L}(w_0) + \nabla \mathscr{L}(w_0)^T (w - w_0) + \frac{1}{2\eta} \|w - w_0\|^2.$$

Note that the chosen quadratic term does not depend on $\mathscr{L}$.

We have

$$\nabla \tilde{\mathscr{L}}_{w_0}(w) = \nabla \mathscr{L}(w_0) + \frac{1}{\eta}(w - w_0),$$

which leads to

$$\underset{w}{\operatorname{argmin}} \, \tilde{\mathscr{L}}_{w_0}(w) = w_0 - \eta \nabla \mathscr{L}(w_0).$$

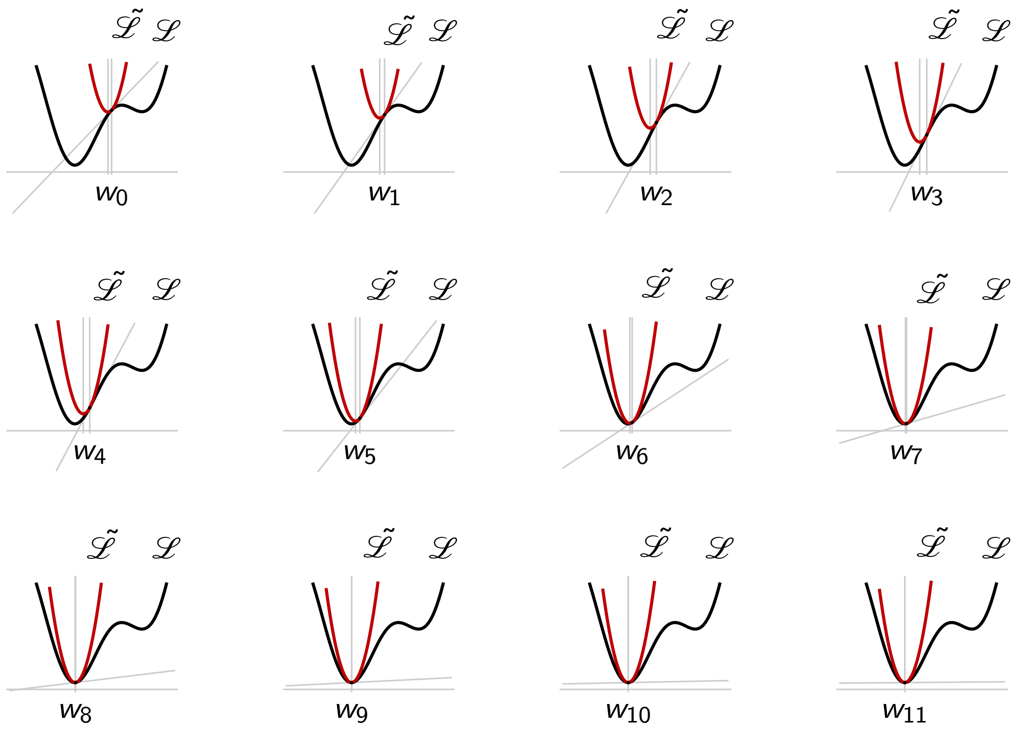
The resulting iterative rule, which goes to the minimum of the approximation at the current location, takes the form:

$$w_{t+1} = w_t - \eta \nabla \mathscr{L}(w_t),$$

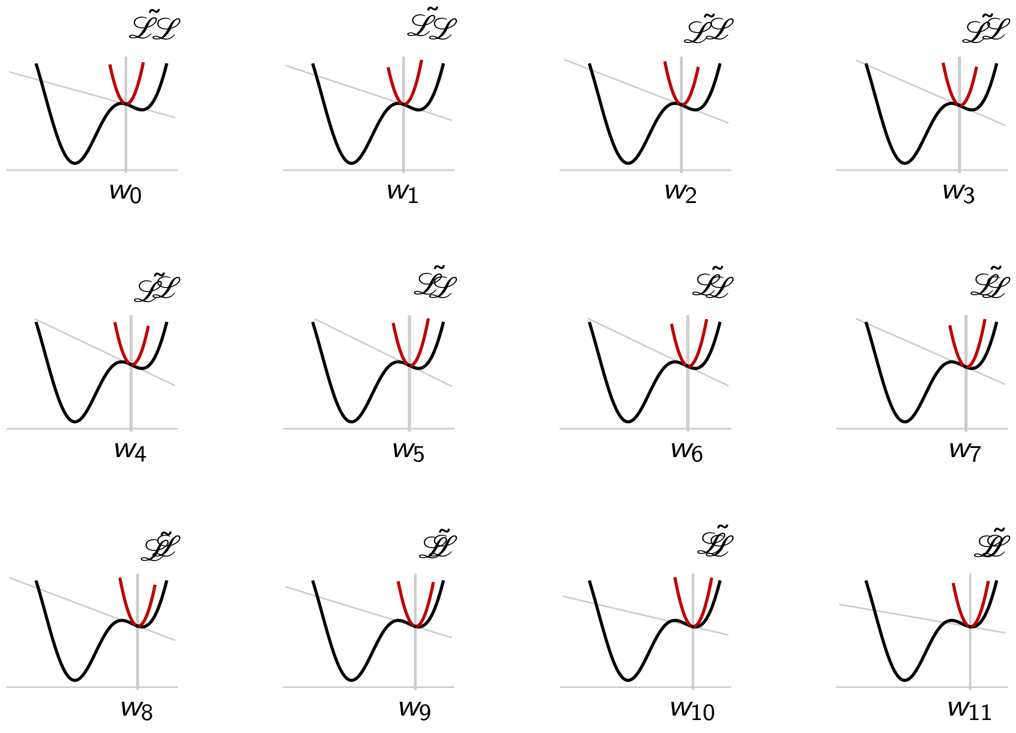
which corresponds intuitively to "following the steepest descent".

This [most of the time] eventually ends up in a **local** minimum, and the choices of w_0 and η are important.

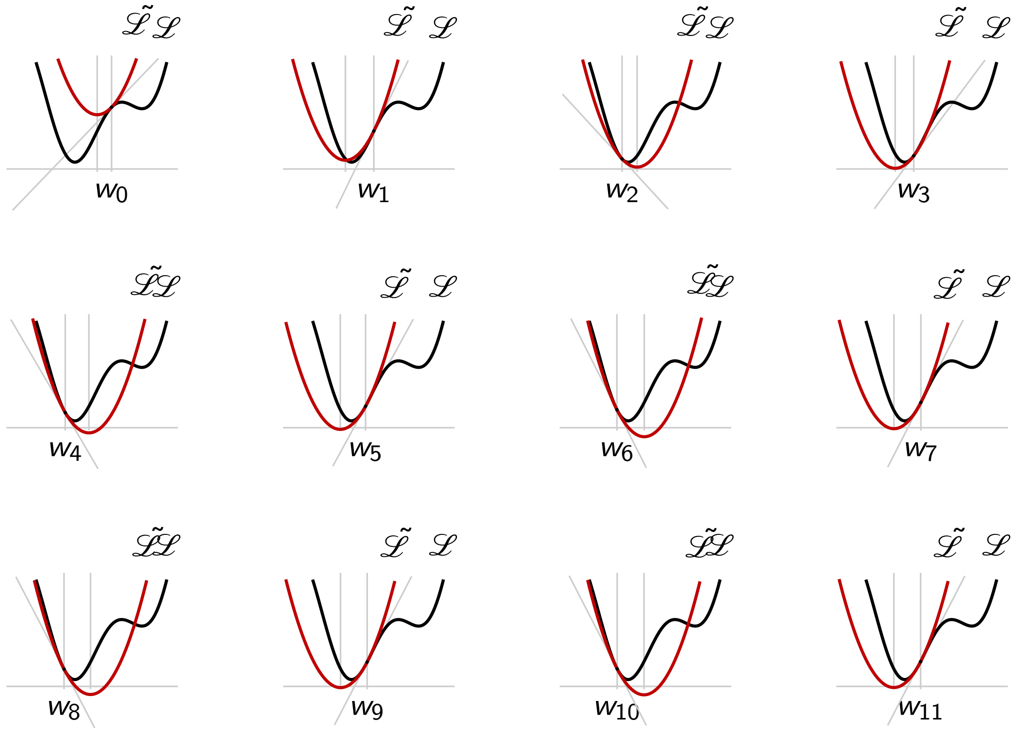
$\eta = 0.125$

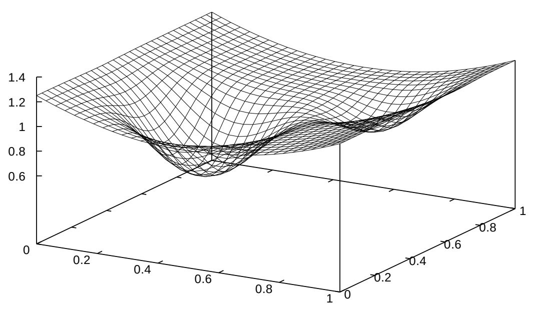


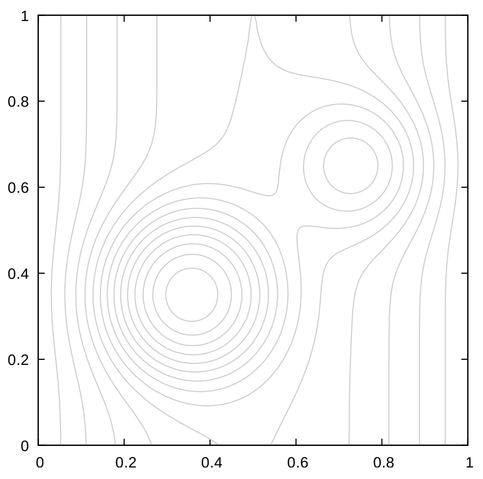
$\eta = 0.125$

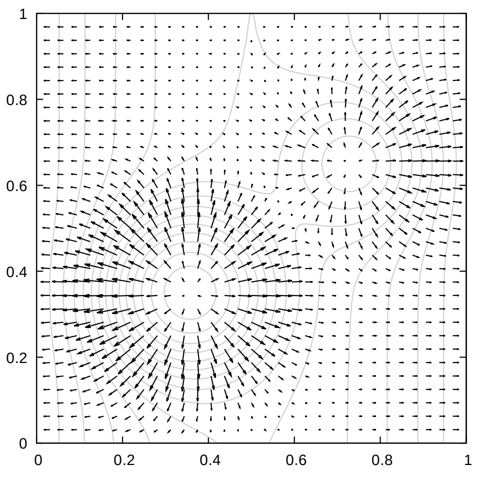


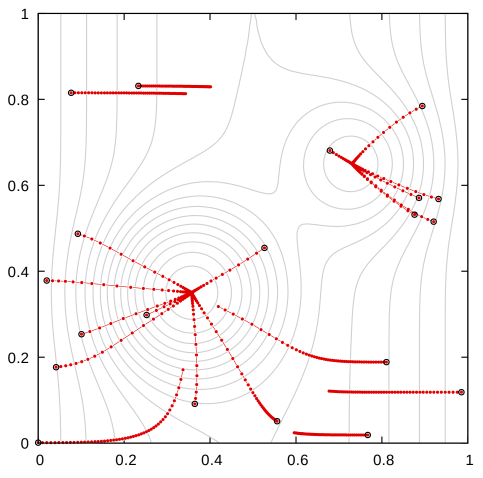
$\eta = 0.5$


$\tilde{\mathscr{L}}$ $\mathscr{L}$
w_0
w_1
w_2
w_3
w_4
w_5
w_6
w_7
w_8
w_9
w_{10}
w_{11}


1.4
1.2
1
0.8
0.6
0
0.2
0.4
0.6
0.8
1


0
0.2
0.4
0.6
0.8
1


0
0.2
0.4
0.6
0.8
1


0
0.2
0.4
0.6
0.8
1

We saw that the minimum of the logistic regression loss

$$\mathscr{L}(w, b) = -\sum_n \log \sigma(y_n(w \cdot x_n + b))$$

does not have an analytic form.

```
def sigmoid(x):
    return 1 / (1 + numpy.exp(-x))

def loss(x, y, w, b):
    return - numpy.log(sigmoid(y * (x.dot(w) + b))).sum()
```

We can derive

$$\frac{\partial \mathscr{L}}{\partial b} = -\sum_n \underbrace{y_n\, \sigma(-y_n(w \cdot x_n + b))}_{u_n},$$

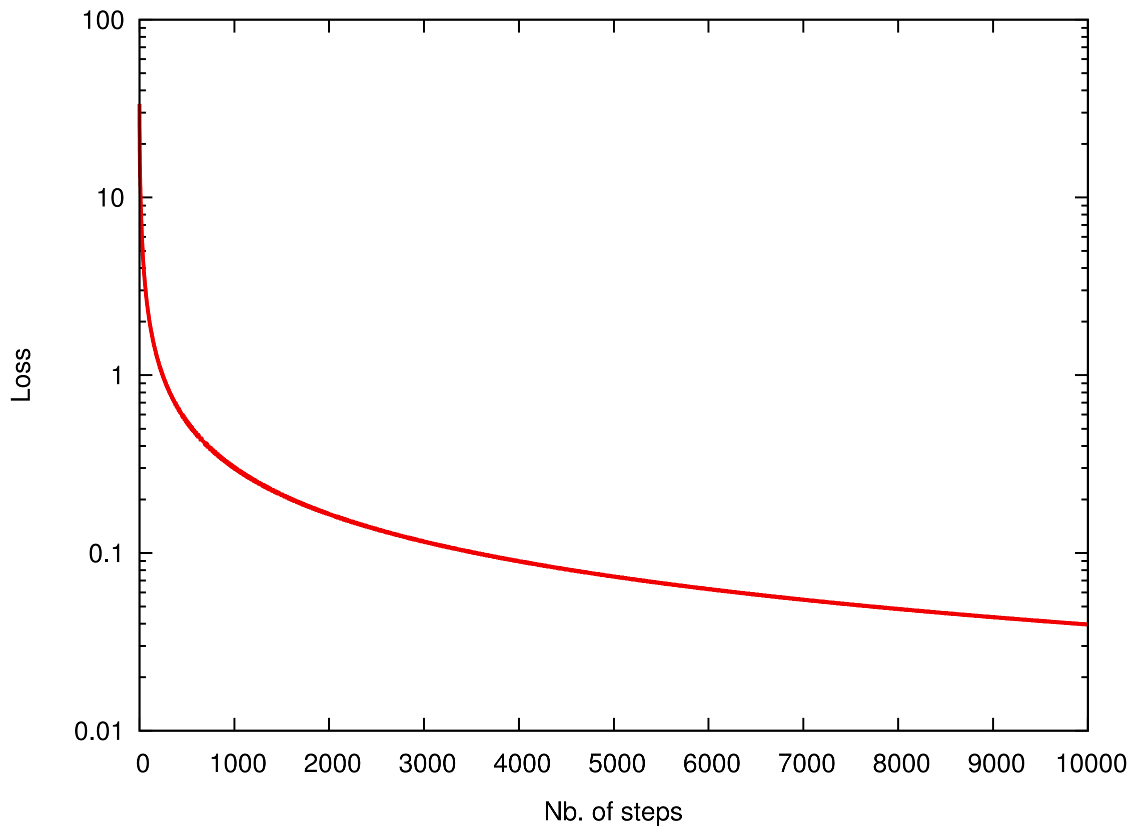
$$\forall d,\ \frac{\partial \mathscr{L}}{\partial w_d} = -\sum_n \underbrace{x_{n,d}\, y_n\, \sigma(-y_n(w \cdot x_n + b))}_{v_{n,d}},$$

which can be implemented as

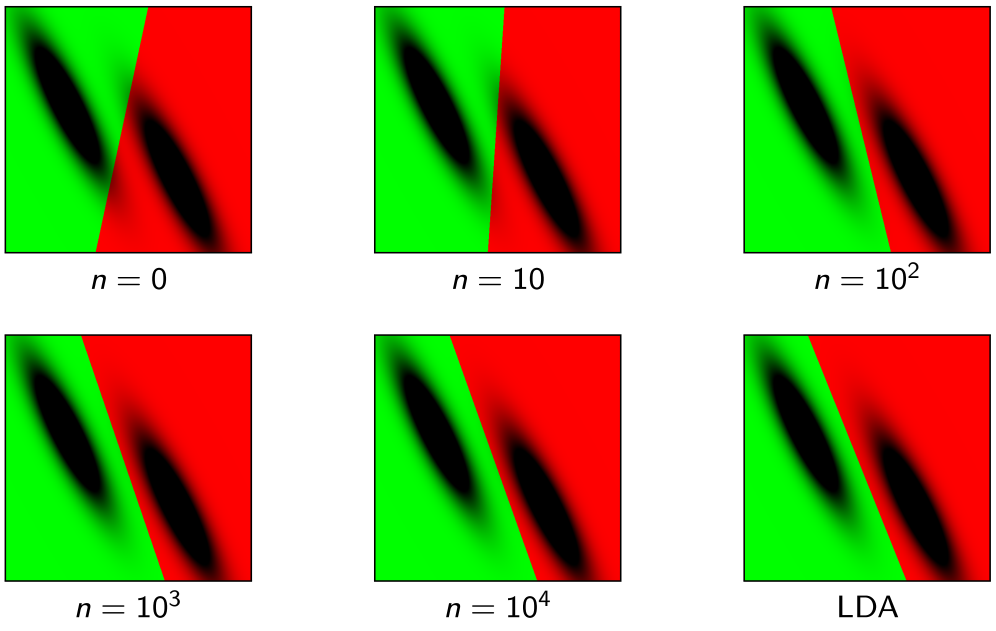
```
def gradient(x, y, w, b):
    u = y * sigmoid(- y * (x.dot(w) + b))
    v = x * u.reshape(-1, 1)
    return - v.sum(0), - u.sum(0)
```

and the gradient descent as

```
w, b = numpy.random.normal(0, 1, (x.shape[1],)), 0
eta = 1e-1

for k in range(nb_iterations):
    print(k, loss(x, y, w, b))
    dw, db = gradient(x, y, w, b)
    w -= eta * dw
    b -= eta * db
```



With 100 training points and $\eta = 10^{-1}$.



We want to train an MLP by minimizing a loss over the training set

$$\mathscr{L}(w, b) = \sum_n \ell(f(x_n; w, b), y_n).$$

To use gradient descent, we need the gradient of the per-sample loss with respect to the parameters.

That is, With $\ell_n = \ell(f(x_n; w, b), y_n)$,

$$\frac{\partial \ell_n}{\partial w_{i,j}^{(l)}} \quad \text{and} \quad \frac{\partial \ell_n}{\partial b_i^{(l)}}.$$

For clarity, we consider a single training sample x, and introduce $s^{(1)}, \dots, s^{(L)}$ as the summations before activation functions.

$$x^{(0)} = x \xrightarrow{w^{(1)}, b^{(1)}} s^{(1)} \xrightarrow{\sigma} x^{(1)} \xrightarrow{w^{(2)}, b^{(2)}} s^{(2)} \xrightarrow{\sigma} \dots \xrightarrow{w^{(L)}, b^{(L)}} s^{(L)} \xrightarrow{\sigma} x^{(L)} = f(x; w, b).$$

Formally we set $x^{(0)} = x$,

$$\forall l = 1, \dots, L, \quad \left\{ \begin{array}{l} s^{(l)} = w^{(l)} x^{(l-1)} + b^{(l)} \\ x^{(l)} = \sigma\left(s^{(l)}\right), \end{array} \right.$$

and we set the output of the network as $f(x; w, b) = x^{(L)}$.

This is the **forward pass**.

The core principle of the back-propagation algorithm is the "chain rule" from differential calculus:

$$(g \circ f)' = (g' \circ f) f'.$$

The linear approximation of a composition of mappings is the product of their individual linear approximations.

This generalizes to longer compositions and higher dimensions

$$J_{f_N \circ f_{N-1} \circ \dots \circ f_1}(x) = J_{f_1}(x)\, J_{f_2}(f_1(x))\, J_{f_3}(f_2(f_1(x))) \dots J_{F_N}(f_{N-1}(\dots(x)))$$

where $J_f(x)$ is the Jacobian of f at x, that is the matrix of the linear approximation of f in the neighborhood of x.

$$x^{(l-1)} \xrightarrow{w^{(l)},\, b^{(l)}} s^{(l)} \xrightarrow{\sigma} x^{(l)}$$

Since $s_i^{(l)}$ influences ℓ only through $x_i^{(l)}$ with

$$x_i^{(l)} = \sigma(s_i^{(l)}),$$

we have

$$\frac{\partial \ell}{\partial s_i^{(l)}} = \frac{\partial \ell}{\partial x_i^{(l)}} \frac{\partial x_i^{(l)}}{\partial s_i^{(l)}} = \frac{\partial \ell}{\partial x_i^{(l)}} \sigma'\left(s_i^{(l)}\right),$$

And since $x_j^{(l-1)}$ influences ℓ only through the $s_i^{(l)}$ with

$$s_i^{(l)} = \sum_j w_{i,j}^{(l)} x_j^{(l-1)} + b_i^{(l)},$$

we have

$$\frac{\partial \ell}{\partial x_j^{(l-1)}} = \sum_j \frac{\partial \ell}{\partial s_i^{(l)}} \frac{\partial s_i^{(l)}}{\partial x_j^{(l-1)}} = \sum_i \frac{\partial \ell}{\partial s_i^{(l)}} w_{i,j}^{(l)}.$$

$$x^{(l-1)} \xrightarrow{w^{(l)},\, b^{(l)}} s^{(l)} \xrightarrow{\sigma} x^{(l)}$$

Since $w_{i,j}^{(l)}$ and $b_i^{(l)}$ influences ℓ only through $s_i^{(l)}$ with

$$s_i^{(l)} = \sum_j w_{i,j}^{(l)} x_j^{(l-1)} + b_i^{(l)},$$

we have

$$\frac{\partial \ell}{\partial w_{i,j}^{(l)}} = \frac{\partial \ell}{\partial s_i^{(l)}} \frac{\partial s_i^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial \ell}{\partial s_i^{(l)}} x_j^{(l-1)},$$

$$\frac{\partial \ell}{\partial b_i^{(l)}} = \frac{\partial \ell}{\partial s_i^{(l)}}.$$

To summarize: we can compute $\frac{\partial \ell}{\partial x_i^{(L)}}$ from the definition of ℓ, and recursively **propagate backward** the derivatives of the loss w.r.t the activations with

$$\frac{\partial \ell}{\partial s_i^{(l)}} = \frac{\partial \ell}{\partial x_i^{(l)}} \sigma'\left(s_i^{(l)}\right)$$

and

$$\frac{\partial \ell}{\partial x_j^{(l-1)}} = \sum_i \frac{\partial \ell}{\partial s_i^{(l)}} w_{i,j}^{(l)}.$$

And then compute the derivatives w.r.t the parameters with

$$\frac{\partial \ell}{\partial w_{i,j}^{(l)}} = \frac{\partial \ell}{\partial s_i^{(l)}} x_j^{(l-1)},$$

and

$$\frac{\partial \ell}{\partial b_i^{(l)}} = \frac{\partial \ell}{\partial s_i^{(l)}}.$$

To write all this in tensorial form, if $\psi : \mathbb{R}^N \to \mathbb{R}^M$, we will use the standard Jacobian notation

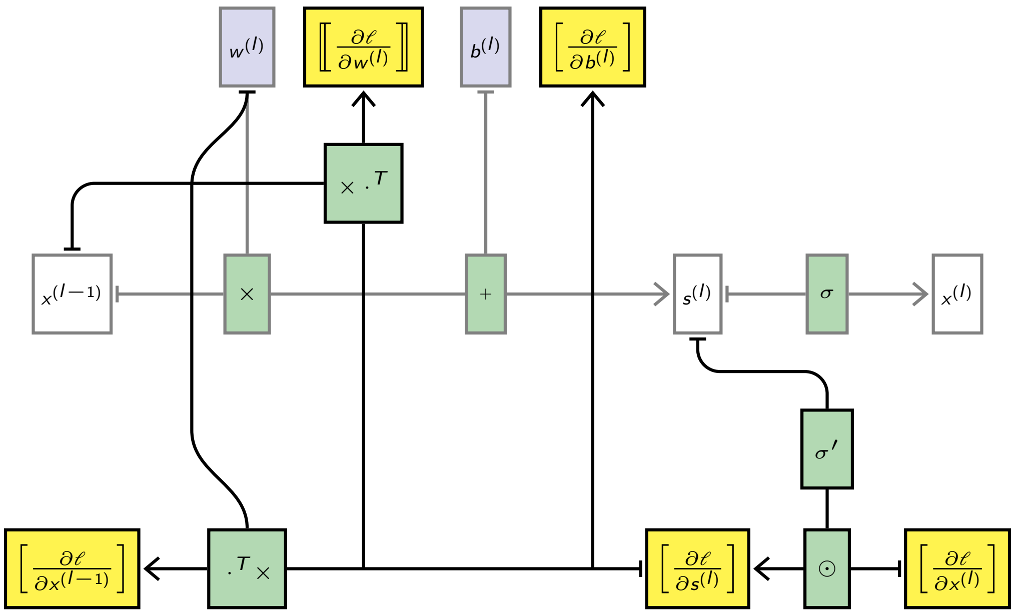
$$\left[\frac{\partial \psi}{\partial x}\right] = \begin{pmatrix} \frac{\partial \psi_1}{\partial x_1} & \dots & \frac{\partial \psi_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial \psi_M}{\partial x_1} & \dots & \frac{\partial \psi_M}{\partial x_N} \end{pmatrix},$$

and if $\psi : \mathbb{R}^{N\times M} \to \mathbb{R}$, we will use the compact notation, also tensorial

$$\left[\!\left[\frac{\partial \psi}{\partial w}\right]\!\right] = \begin{pmatrix} \frac{\partial \psi}{\partial w_{1,1}} & \dots & \frac{\partial \psi}{\partial w_{1,M}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \psi}{\partial w_{N,1}} & \dots & \frac{\partial \psi}{\partial w_{N,M}} \end{pmatrix}.$$

A standard notation (that we do not use here) is

$$\left[\frac{\partial \ell}{\partial x^{(l)}}\right] = \nabla_{x^{(l)}}\ell \quad \left[\frac{\partial \ell}{\partial s^{(l)}}\right] = \nabla_{s^{(l)}}\ell \quad \left[\frac{\partial \ell}{\partial b^{(l)}}\right] = \nabla_{b^{(l)}}\ell \quad \left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right] = \nabla_{w^{(l)}}\ell.$$


$w^{(l)}$
$\left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right]$
$b^{(l)}$
$\left[\frac{\partial \ell}{\partial b^{(l)}}\right]$
$\times\ .^T$
$x^{(l-1)}$
$\times$
$+$
$s^{(l)}$
σ
$x^{(l)}$
σ'
$\left[\frac{\partial \ell}{\partial x^{(l-1)}}\right]$
$.^T \times$
$\left[\frac{\partial \ell}{\partial s^{(l)}}\right]$
$\odot$
$\left[\frac{\partial \ell}{\partial x^{(l)}}\right]$

Forward pass

Compute the activations.

$$x^{(0)} = x, \quad \forall l = 1, \dots, L, \begin{cases} s^{(l)} = w^{(l)} x^{(l-1)} + b^{(l)} \\ x^{(l)} = \sigma\left(s^{(l)}\right) \end{cases}$$

Backward pass

Compute the derivatives of the loss wrt the activations.

$$\begin{cases} \left[\frac{\partial \ell}{\partial x^{(L)}}\right] \text{ from the definition of } \ell \\ \text{if } l < L, \left[\frac{\partial \ell}{\partial x^{(l)}}\right] = \left(w^{(l+1)}\right)^T \left[\frac{\partial \ell}{\partial s^{(l+1)}}\right] \end{cases} \qquad \left[\frac{\partial \ell}{\partial s^{(l)}}\right] = \left[\frac{\partial \ell}{\partial x^{(l)}}\right] \odot \sigma'\left(s^{(l)}\right)$$

Compute the derivatives of the loss wrt the parameters.

$$\left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right] = \left[\frac{\partial \ell}{\partial s^{(l)}}\right] \left(x^{(l-1)}\right)^T \qquad \left[\frac{\partial \ell}{\partial b^{(l)}}\right] = \left[\frac{\partial \ell}{\partial s^{(l)}}\right].$$

Gradient step

Update the parameters.

$$w^{(l)} \leftarrow w^{(l)} - \eta \left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right] \qquad b^{(l)} \leftarrow b^{(l)} - \eta \left[\frac{\partial \ell}{\partial b^{(l)}}\right]$$

In spite of its hairy formalization, the backward pass is a simple algorithm: apply the chain rule again and again.

As for the forward pass, it can be expressed in tensorial form. Heavy computation is concentrated in linear operations, and all the non-linearities go into component-wise operations.

Regarding computation, since the costly operation for the forward pass is

$$s^{(l)} = w^{(l)} x^{(l-1)} + b^{(l)}$$

and for the backward

$$\left[\frac{\partial \ell}{\partial x^{(l)}}\right] = \left(w^{(l+1)}\right)^T \left[\frac{\partial \ell}{\partial s^{(l+1)}}\right]$$

and

$$\left[\!\left[\frac{\partial \ell}{\partial w^{(l)}}\right]\!\right] = \left[\frac{\partial \ell}{\partial s^{(l)}}\right] \left(x^{(l-1)}\right)^T,$$

the rule of thumb is that the backward pass is twice more expensive than the forward one.

Stochastic Gradient descent

To minimize a loss of the form

$$\mathscr{L}(w) = \sum_{n=1}^{N} \underbrace{\ell(f(x_n; w), y_n)}_{\ell_n(w)}$$

the standard gradient-descent algorithm update has the form

$$w_{t+1} = w_t - \eta \nabla \mathscr{L}(w_t).$$

A straight-forward implementation would be

```
for k in range(nb_epochs):
    output = mlp.forward(x)
    mlp.compute_grad(dloss(y, output))
    mlp.gradient_descent_step(eta)
```

However, the memory footprint is proportional to the full set size. This can be mitigated by summing the gradient through "mini-batches":

```
for k in range(nb_epochs):
    mlp.zero_grad()
    for b in range(0, x.shape[0], nb):
        output = mlp.forward(x[b:b+nb])
        mlp.accumulate_grad(dloss(y[b:b+nb], output))
    mlp.gradient_descent_step(eta)
```

While it makes sense in principle to compute the gradient exactly, in practice:

- It takes time to compute (more exactly **all our time!**).
- It is an empirical estimation of an hidden quantity, and any partial sum is also an unbiased estimate, although of greater variance.
- It is computed incrementally

  $$\nabla \mathscr{L}(w_t) = \sum_{n=1}^{N} \nabla \ell_n(w_t),$$

  and when we compute ℓ_n, we have already computed $\ell_1, \dots, \ell_{n-1}$, and we could have a better estimate of w^* than w_t.

To illustrate how partial sums are good estimates, consider an ideal case where the training set is the same set of $M \ll N$ samples replicated K times. Then

$$\begin{aligned}\mathscr{L}(w) &= \sum_{n=1}^{N} \ell(f(x_n; w), y_n) \\ &= \sum_{k=1}^{K} \sum_{m=1}^{M} \ell(f(x_m; w), y_m) \\ &= K \sum_{m=1}^{M} \ell(f(x_m; w), y_m).\end{aligned}$$

So instead of summing over all the samples and moving by η, we can visit only $M = N/K$ samples and move by $K\eta$, which would cut the computation by K.

Although this is an ideal case, there is redundancy in practice that results in similar behaviors.

The **stochastic gradient descent** consists of updating the parameters w_t after every sample

$$w_{t+1} = w_t - \eta \nabla \ell_{n(t)}(w_t).$$

However this does not benefit from the speed-up of batch-processing.

The **mini-batch stochastic gradient descent** is the standard procedure for deep learning. It consists of visiting the samples in "mini-batches", each of a few tens of samples, and updating the parameters each time.

$$w_{t+1} = w_t - \eta \sum_{b=1}^{B} \nabla \ell_{n(t,b)}(w_t).$$

The order $n(t, b)$ to visit the samples can either be sequential, or uniform sampling, usually without replacement.

The stochastic behavior of this procedure helps evade local minima.

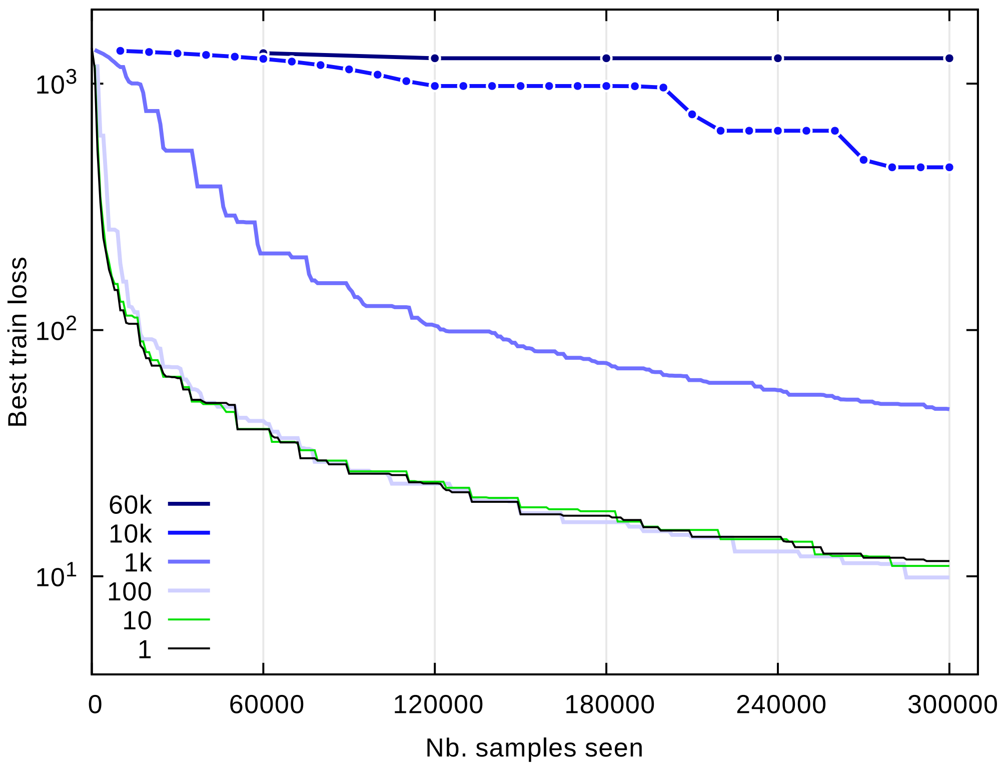
So our exact gradient descent with mini-batches

```
for k in range(nb_epochs):
    mlp.zero_grad()
    for b in range(0, x.shape[0], nb):
        output = mlp.forward(x[b:b+nb])
        mlp.accumulate_grad(dloss(y[b:b+nb], output))
    mlp.gradient_descent_step(eta)
```

can be modified into the mini-batch stochastic gradient descent as follows:

```
for k in range(nb_epochs):
    for b in range(0, x.shape[0], nb):
        output = mlp.forward(x[b:b+nb])
        mlp.compute_grad(dloss(y, output))
        mlp.gradient_descent_step(eta)
```

Mini-batch size and loss reduction (MNIST)



Convolution layers

If they were handled as normal “unstructured” vectors, large-dimension signals such as sound samples or images would require models of intractable size.

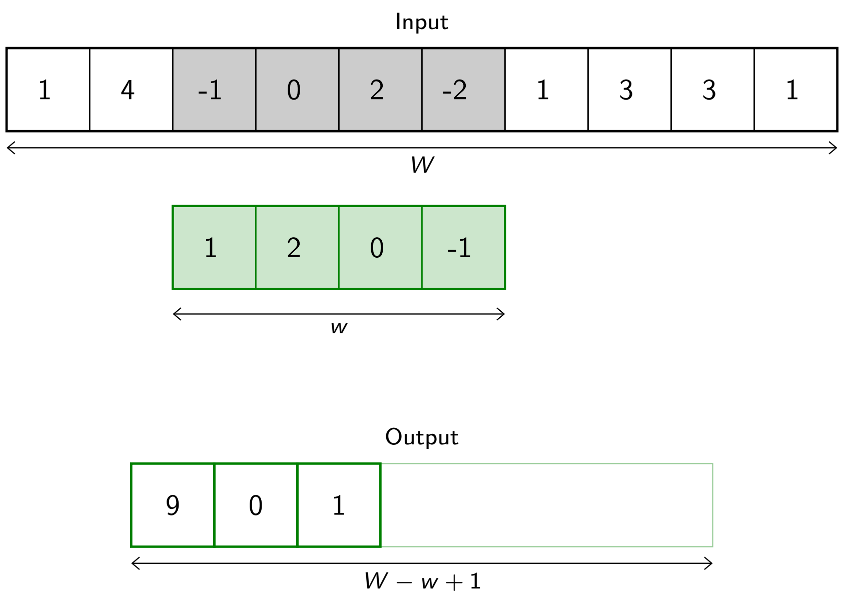
For instance a linear layer taking a 256×256 RGB image as input, and producing an image of same size would require

$$(256 \times 256 \times 3)^2 \simeq 3.87e{+}10$$

parameters, with the corresponding memory footprint ($\simeq$150Gb !), and excess of capacity.

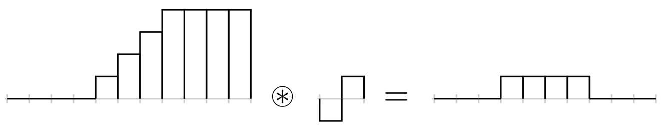
Moreover, this requirement is inconsistent with the intuition that such large signals have some “invariance in translation”. **A representation meaningful at a certain location can / should be used everywhere.**

A convolution layer embodies this idea. It applies the same linear transformation locally, everywhere, and preserves the signal structure.

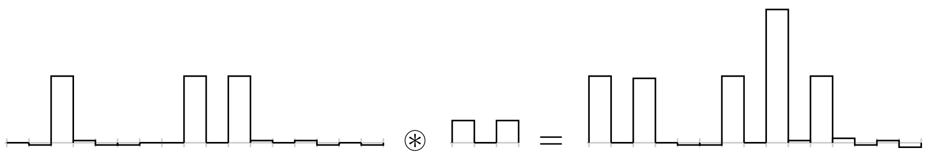


Convolution can implement in particular differential operators, *e.g.*

$$(0,0,0,0,1,2,3,4,4,4,4) \circledast (-1,1) = (0,0,0,1,1,1,1,0,0,0).$$



or crude "template matcher", *e.g.*



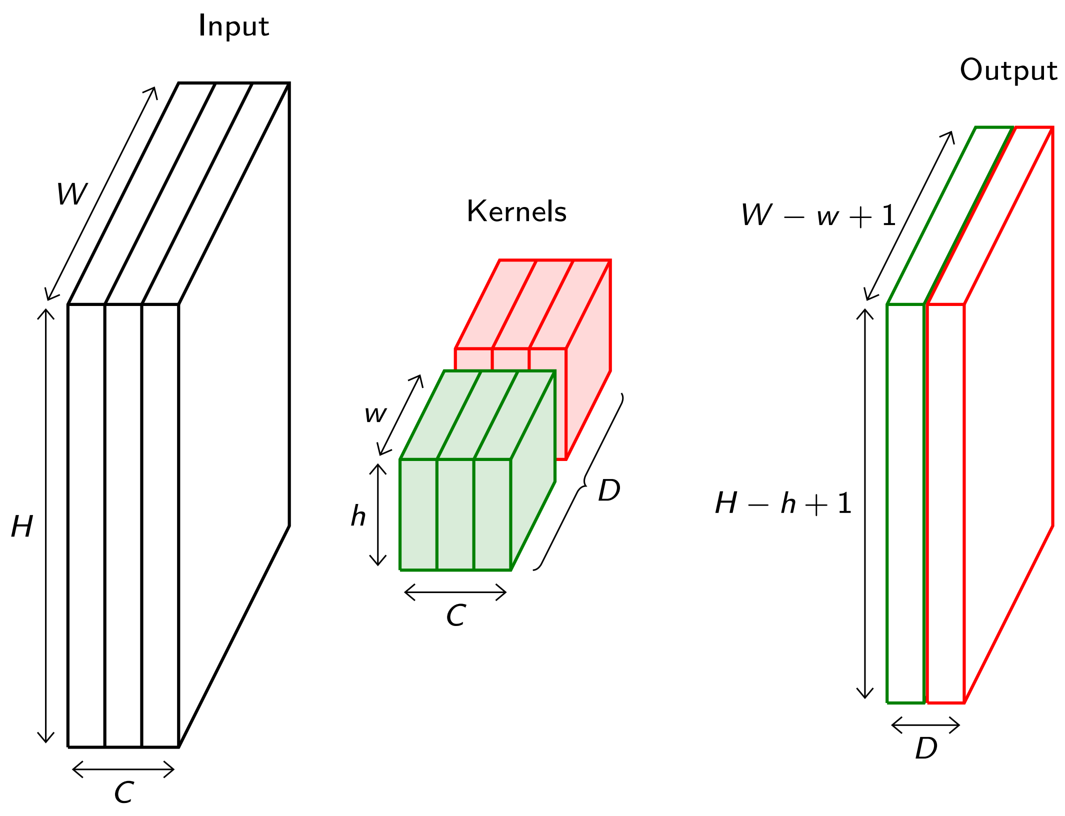
Both of these computation examples are indeed "invariant by translation".

It generalizes naturally to a multi-dimensional input, although specification can become complicated.

Its most usual form for "convolutional networks" processes a 3d tensor as input (*i.e.* a multi-channel 2d signal) to output a 2d tensor. The kernel is not swiped across channels, just across rows and columns.

In this case, if the input tensor is of size $C \times H \times W$, and the kernel is $C \times h \times w$, the output is $(H - h + 1) \times (W - w + 1)$.

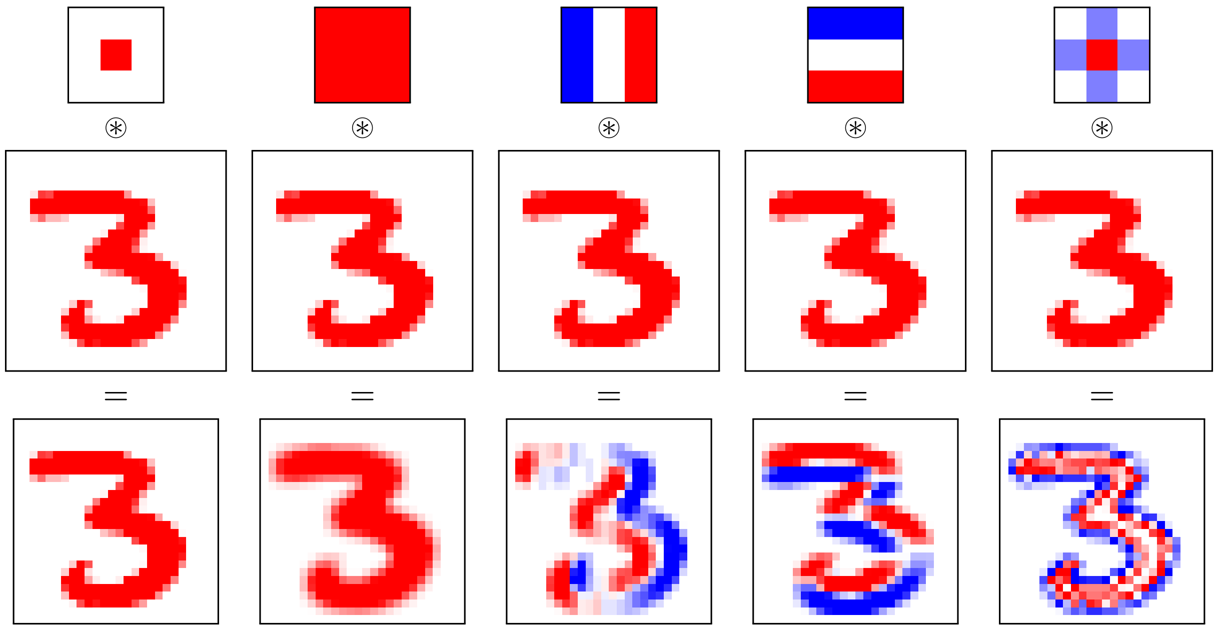
In a standard convolution layer, D such convolutions are combined to generate a $D \times (H - h + 1) \times (W - w + 1)$ output.



Note that a convolution **preserves the signal support structure**.

A 1d signal is converted into a 1d signal, a 2d signal into a 2d, and neighboring parts of the input signal influence neighboring parts of the output signal.

A 3d convolution can be used if the channel index has some metric meaning, such as time for a series of grayscale video frames. Otherwise swiping across channels makes no sense.



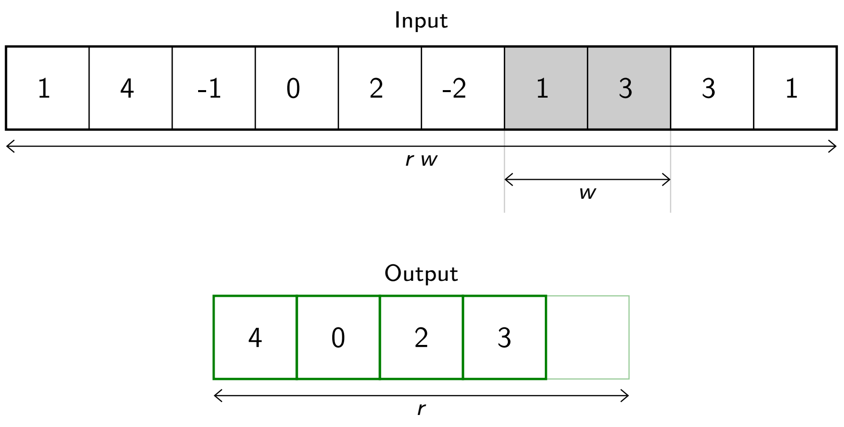
Pooling

The historical approach to compute a low-dimension signal (*e.g.* a few scores) from a high-dimension one (*e.g.* an image) was to use **pooling** operations.

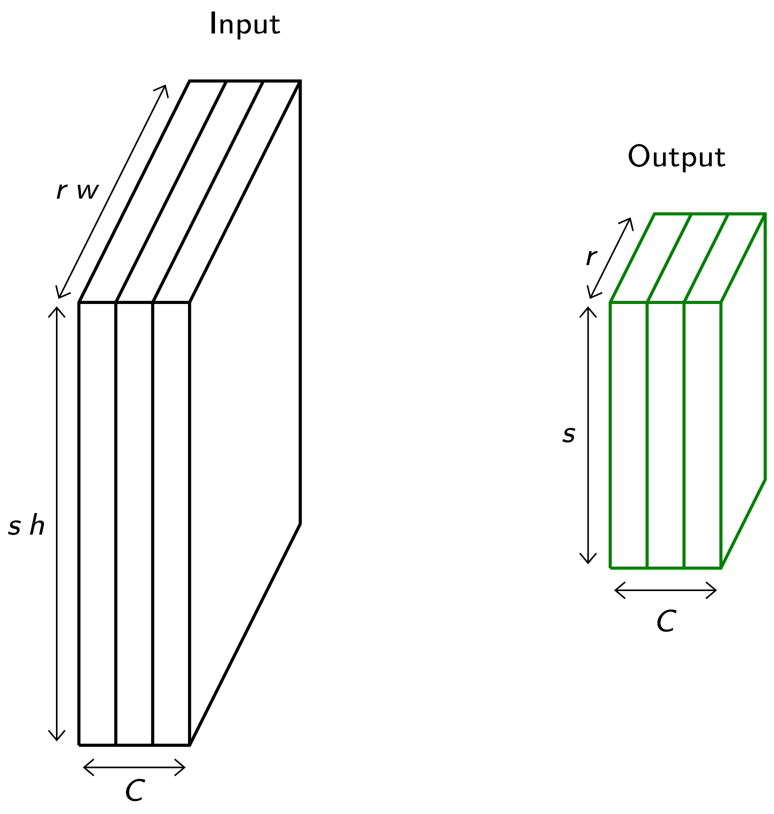
Such an operation aims at grouping several activations into a single "more meaningful" one.

The most standard type of pooling is the **max-pooling**, which computes max values over non-overlapping blocks.

For instance in 1d with a kernel of size 2:



The **average pooling** computes average values per block instead of max values.

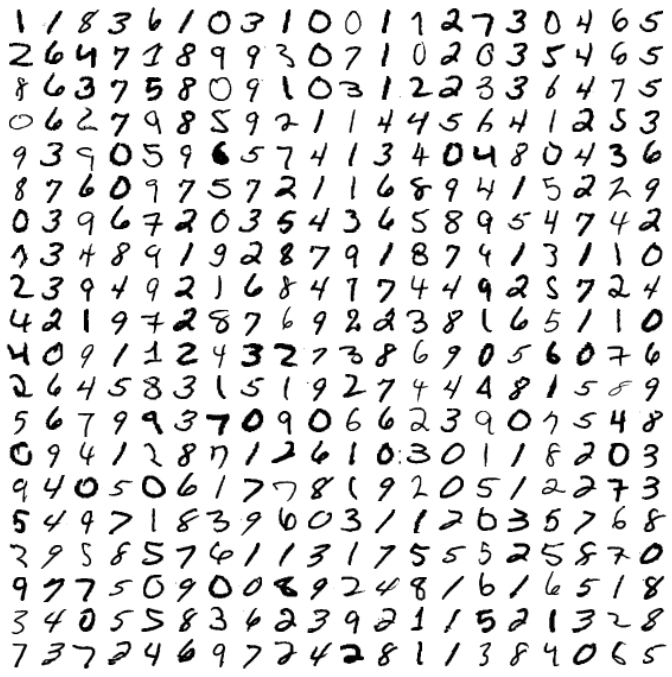


Deep learning frameworks

Back-propagation can be extended to arbitrary graphs of tensors operations, including convolutions, pooling and many more.

This is usually done in a deep-learning specific framework such as PyTorch or TensorFlow, which provide automatic differentiation and GPU computation.

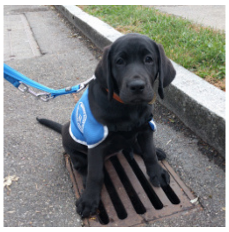
MNIST



(leCun et al., 1998)

```
model = nn.Sequential(
    nn.Conv2d(1, 32, 5),
    nn.ReLU(),
    nn.MaxPool2d(3),
    nn.Conv2d(32, 64, 5),
    nn.ReLU(),
    nn.MaxPool2d(2),
    Shape1D(),
    nn.Linear(256, 256),
    nn.ReLU(),
    nn.Linear(256, 10)
)

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(model.parameters(), lr = 1e-2)

for e in range(nb_epochs):
    for input, target in data_loader_iterator(train_loader):
        output = model(input)
        loss = criterion(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Training time $<$10s (GPU), error $\simeq$1%



```
import PIL, torch, torchvision

img = torchvision.transforms.ToTensor()(PIL.Image.open('blacklab.jpg'))
img = img.view(1, img.size(0), img.size(1), img.size(2))
img = 0.5 + 0.5 * (img - img.mean()) / img.std()

alexnet = torchvision.models.alexnet(pretrained = True)
alexnet.eval()
output = alexnet(Variable(img))
scores, indexes = output.data.view(-1).sort(descending = True)

class_names = eval(open('imagenet1000_clsid_to_human.txt', 'r').read())
for k in range(15):
    print('#{:d} ({:.02f}) {:s}'.format(k, scores[k], class_names[indexes[k]]))
```

```
#1 (12.26) Weimaraner
#2 (10.95) Chesapeake Bay retriever
#3 (10.87) Labrador retriever
#4 (10.10) Staffordshire bullterrier, Staffordshire bull terrier
#5 (9.55) flat-coated retriever
#6 (9.40) Italian greyhound
#7 (9.31) American Staffordshire terrier, Staffordshire terrier
#8 (9.12) Great Dane
#9 (8.94) German short-haired pointer
#10 (8.53) Doberman, Doberman pinscher
#11 (8.35) Rottweiler
#12 (8.25) kelpie
```

Weimaraner

Chesapeake Bay retriever

References

N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Conference on Computer Vision and Pattern Recognition (CVPR), pages 886–893, 2005.

P. Dollár, Z. Tu, P. Perona, and S. Belongie. Integral channel features. In British Machine Vision Conference, pages 91.1–91.11, 2009.

Y. leCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, 1998.

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.