EE-311—Apprentissage et intelligence artificielle

8. Perceptron multi-couches

François Fleuret (ML : modifs 2022–24)

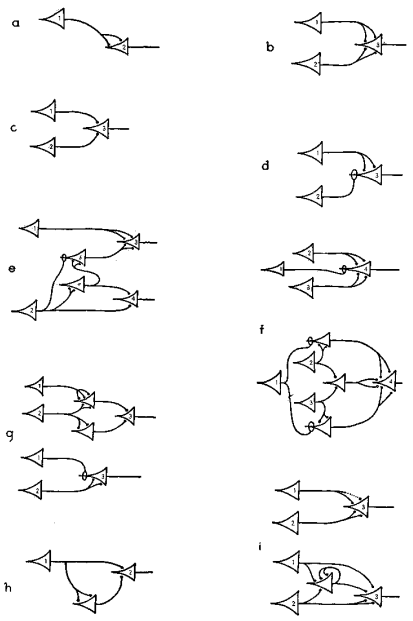
`https://moodle.epfl.ch/course/view.php?id=16090`

Vendredi 11 avril 2025 (compiled 10 avril 2025)

Réseaux de neurones : Contenu

- Origine, inspirations, jalons historiques
- Le perceptron : ancêtres, définition du perceptron, fonction d'activation, représentations graphiques et tensorielles, algorithme du perceptron, convergence, comparaison avec les SVM
- Limitations des modèles linéaires
- Shallow learning (pré-traitement
- Modèles profonds : le perceptron multi-couche
- Formalisme, fonctions d'activation et approximation universelle
- Rappel du descente de gradient
- Entraînement du perceptron multi-couche :
  - propagation vers l'avant
  - rétropropagation du gradient
- Coût computational et interprétation des opérations

Origines de l'idée de neurone artificiel



Réseau de "Threshold Logic Unit"

(McCulloch and Pitts, 1943)

Historique : quelques étapes importantes

1949 – Donald Hebb propose une règle (qui porte son nom) suggérant la formation de connections entre des unités (neurones) qui s'activent en même temps (ou séquentiellement) : "cells that fire together, wire together"

1951 – Marvin Minsky crée le premier réseau de neurones artificiels (règle de Hebb, 40 neurones).

1958 – Frank Rosenblatt créer un perceptron pour classifier des images 20×20.

1959 – David H. Hubel et Torsten Wiesel exposent les processus du cortex visuel des chats (similarités avec opérations en traitement du signal).

1982 – Paul Werbos propose la rétro-propagation du gradient.

Le perceptron

Threshold Logic Unit

Le premier modèle mathématique d'un neurone est la "Threshold Logic Unit,"(McCulloch and Pitts, 1943) qui a des entrées et sorties booléennes (0 ou 1) :

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

Elle est en particulier capable de calculer les trois opérations de l'algèbre booléenne :

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

Donc, **on peut construire n'importe quelle fonction booléenne, donc arithmétique, avec de telles unités élémentaires.**

(McCulloch and Pitts, 1943)

Perceptron

Le perceptron, défini par :

$$f(\vec{x}) = \begin{cases} 1 & \text{si } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{sinon} \end{cases}$$

est très similaire à la Threshold Logic Unit, mais ses entrées x_i sont des valeurs réelles et chacune a un poids spécifique w_i. On appelle biais le paramètre b.

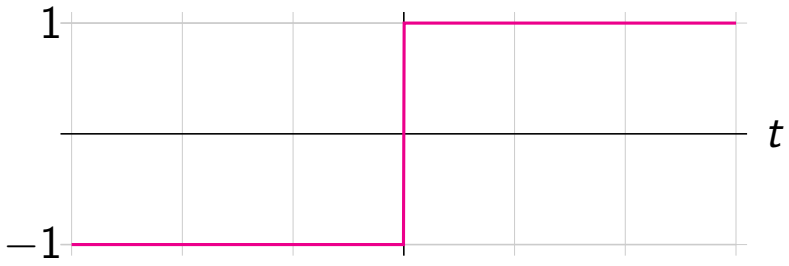
Ce modèle a été motivé initialement par la biologie, avec w_i jouant le rôle des *poids synaptiques*, et les x_i et f des fréquences de potentiels d'action. C'est un modèle très grossier.

(Rosenblatt, 1957)

Fonction signe et fonction d'activation

Pour simplifier les choses, nous considérons des sorties ± 1. Soit

$$\sigma(t) = \begin{cases} 1 & \text{si } t \geq 0 \\ -1 & \text{sinon.} \end{cases}$$



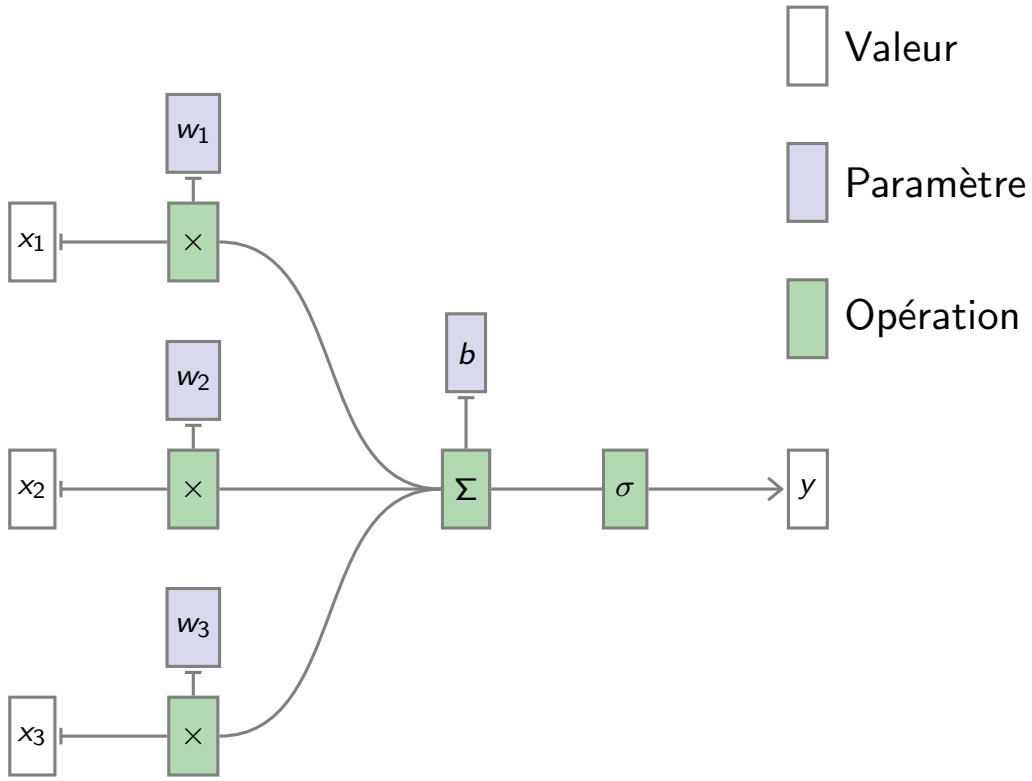
La règle de classification du perceptron peut alors être formalisée par

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

Dans le cas des réseaux de neurones, la fonction σ qui suit un opérateur linéaire est classiquement appelée la **fonction d'activation**.

Représentation graphique

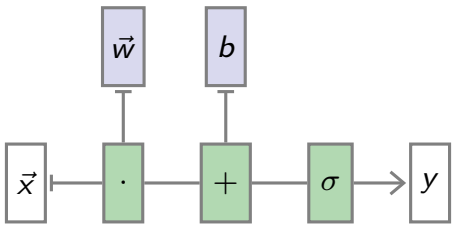
Nous pouvons représenter un “neurone” de la manière suivante :



Représentation tensorielle

Nous pouvons aussi utiliser des opérateurs tensoriels

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



Algorithme du perceptron pour ajuster les poids w_i de $\vec{w}$

Étant donné un ensemble d'apprentissage

$$\mathscr{D} = \left\{\vec{x}^i, y^i\right\}_{i=1,\ldots,n}, \text{ avec } \vec{x}^i \in \mathbb{R}^p \text{ et } y^i \in \{-1, 1\}$$

une technique très simple pour entraîner un tel modèle linéaire est l'**algorithme du perceptron** :

1. Initialiser $k \leftarrow 0$, $\vec{w}^{(k)} \leftarrow \vec{0}$, $i \leftarrow 0$, $\overline{k} \leftarrow k$

   k : index de mise à jour des poids
   $\overline{k}$: état de l'index k avant chaque nouveau passage en revue des données

2. Si $y^i \left(\vec{w}^{(k)} \cdot \vec{x}^i\right) \leq 0$

   les poids $\vec{w}^{(k)}$ produisent une mauvaise classification de la $i^{\text{ème}}$ observation $\vec{x}^i$

   $\vec{w}^{(k+1)} \leftarrow \vec{w}^{(k)} + y^i \vec{x}^i$

   ainsi, $y^i \left(\vec{w}^{(k+1)} \cdot \vec{x}^i\right)$ sera moins négatif

   $k \leftarrow k + 1$
3. Si $i < n - 1$, incrémenter $i \leftarrow i + 1$ puis répéter 2.
4. Si $k = \overline{k}$: *stop* ;

   k n'a plus changé (donc les poids $\vec{w}^{(k)}$ non plus)

   sinon : $i \leftarrow 0$, $\overline{k} \leftarrow k$, répéter 2.

Le biais b est ici absent mais peut être introduit comme une des composantes de $\vec{w}$ en rajoutant une composante constante égale à 1 à tous les $\vec{x}^i$.

(Rosenblatt, 1957)

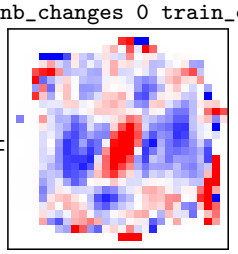
Algorithme du perceptron (NumPy)

```
def train_perceptron(X, y, nb_epochs_max):
    n = X.shape[0] # number of examples
    p = X.shape[1] # number of features
    w = np.zeros((p,))

    for e in range(nb_epochs_max):
        nb_changes = 0
        for i in range(n):
            if X[i].dot(w) * y[i] <= 0:
                # prediction is wrong for ith observation vector X[i]
                w = w + y[i] * X[i]
                nb_changes = nb_changes + 1
        if nb_changes == 0: break;

    return w
```

Classification de chiffres manuscrits avec algo. perceptron

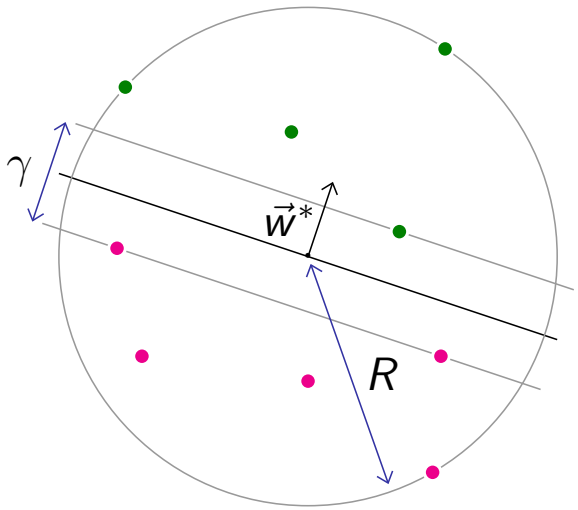
Cet algorithme très simple fonctionne remarquablement bien. Par exemple, avec les images de "1" de MNIST comme classe positive et de "0" comme classe négative (images de $p = 784 = 28 \times 28$ pixels) :

```
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%
```

$\vec{w} =$  bleu : -, rouge : $+$

Convergence de l'algorithme du perceptron

Il est possible d'obtenir une garantie de convergence sous deux hypothèses :



1. Les $\vec{x}^i$ sont dans une sphère de rayon R :
$$\exists R > 0, \ \forall i, \ \|\vec{x}^i\| \leq R.$$
2. Les deux populations peuvent être séparée par une marge $\gamma > 0$:
$$\exists \vec{w}^*, \ \|\vec{w}^*\| = 1, \ \exists \gamma > 0, \text{t.q.} \forall i, \ y^i \left(\vec{x}^i \cdot \vec{w}^*\right) \geq \gamma/2.$$

Preuve de convergence de l'algorithme du perceptron (1/3)

Pour prouver la convergence, faisons l'hypothèse qu'il y a encore un exemple mal classifié après k itérations, et que $\vec{w}^{(k+1)}$ est le vecteur de poids, selon la mise à jour de $\vec{w}^{(k)}$ à l'étape 2.

On a :
$$\begin{aligned}
\vec{w}^{(k+1)} \cdot \vec{w}^* &\overset{\text{Algo perc. étape 2}}{=} \left(\vec{w}^{(k)} + y^i \vec{x}^i\right) \cdot \vec{w}^* \\
&= \vec{w}^{(k)} \cdot \vec{w}^* + y^i \left(\vec{x}^i \cdot \vec{w}^*\right) \\
&\overset{\text{Hyp. 2}}{\geq} \vec{w}^{(k)} \cdot \vec{w}^* + \gamma/2 \\
&\overset{\text{récursion sur } k}{\geq} \vec{w}^{(k-1)} \cdot \vec{w}^* + \gamma/2 + \gamma/2 \\
&\overset{\text{récursion sur } k \text{ et } \vec{w}^{(0)} = \vec{0}}{\geq} (k+1)\,\gamma/2.
\end{aligned}$$

Puisque $\|w^{(k)}\|\|\vec{w}^*\| \geq \vec{w}^{(k)} \cdot \vec{w}^*$, (inégalité de Cauchy-Schwarz) nous obtenons :
$$\begin{aligned}
\|\vec{w}^{(k)}\|^2 &\geq \left(\vec{w}^{(k)} \cdot \vec{w}^*\right)^2 / \|\vec{w}^*\|^2 \\
&\geq k^2\gamma^2/4 \quad (\dagger)
\end{aligned}$$

Preuve de convergence de l'algorithme du perceptron (2/3)

. . . D'autre part :

$$\begin{aligned}
\|\vec{w}^{(k+1)}\|^2 &\overset{\text{déf prod. scal.}}{=} \vec{w}^{(k+1)} \cdot \vec{w}^{(k+1)} \\
&\overset{\text{Algo perc. étape 2}}{=} \left(\vec{w}^{(k)} + y^i \vec{x}^i\right) \cdot \left(\vec{w}^{(k)} + y^i \vec{x}^i\right) \\
&\overset{\text{expansion polynôme}}{=} \underbrace{\vec{w}^{(k)} \cdot \vec{w}^{(k)}}_{\|\vec{w}^{(k)}\|^2} + 2\ \underbrace{y^i\, \vec{w}^{(k)} \cdot \vec{x}^i}_{\leq 0 \text{ car } \vec{x}^i \text{ mal classif.}} + \underbrace{\|\vec{x}^i\|^2}_{\leq R^2 \text{ car hyp. 1}} \\
&\leq \|\vec{w}^{(k)}\|^2 + R^2 \\
&\overset{\text{récursion sur } k}{\leq} \|\vec{w}^{(k-1)}\|^2 + R^2 + R^2 \\
&\overset{\text{récursion sur } k \text{ et } \|\vec{w}^{(0)}\| = 0}{\leq} (k+1)\,R^2. \quad (\ddagger)
\end{aligned}$$

Preuve de convergence de l'algorithme du perceptron (3/3)

En combinant ces deux résultats, (†) et (‡), nous obtenons :

$$k^2\gamma^2/4 \leq \|\vec{w}^k\|^2 \leq \ k\,R^2$$

d'où

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

il ne peut donc pas y avoir d'exemple mal classifié après $\lfloor 4R^2/\gamma^2 \rfloor$ itérations.

Ce résultat est cohérent :

- La borne ne change pas si toute la population est re-scalée,
- plus la marge est importante, plus l'algorithme converge rapidement.

Différence entre perceptron et SVM

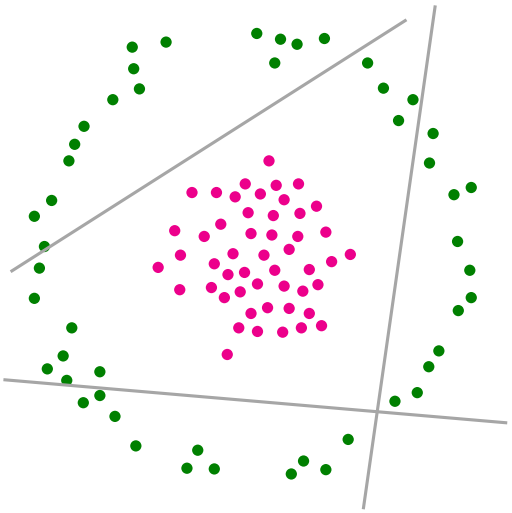
L'algorithme du perceptron s'arrête dès qu'il trouve une frontière de séparation.

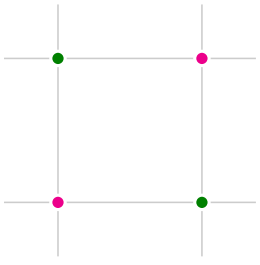
Il se comporte donc différemment d'autres algorithmes, comme par exemple les machines à vecteur de support, qui eux maximisent la distance entre les exemples et la séparation, et sont en conséquence plus robustes au bruit.

Limitation des modèles linéaires

Limitation des modèles linéaires

La principale faiblesse des modèles linéaires est leur manque de capacité à approximer des données arbitraires. En classification par exemple, ils ne peuvent pas traiter des problèmes où les populations ne sont **pas linéairement séparables** :



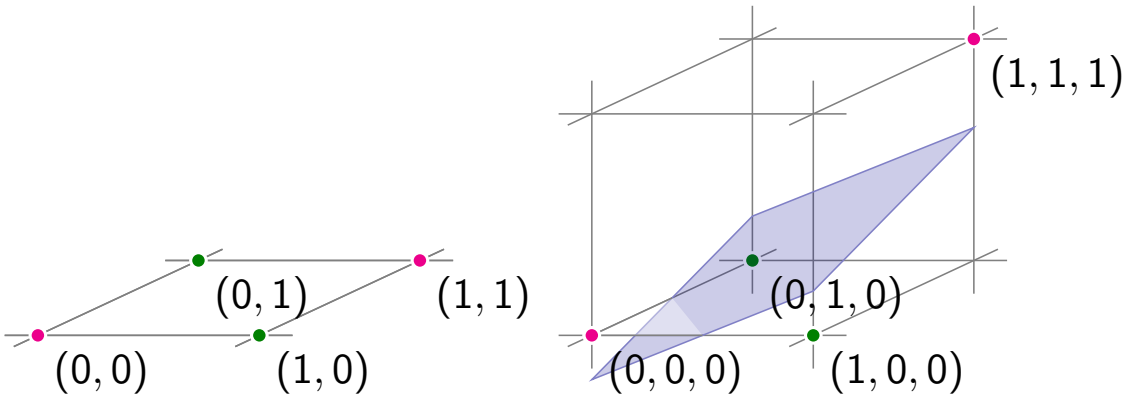


"xor"

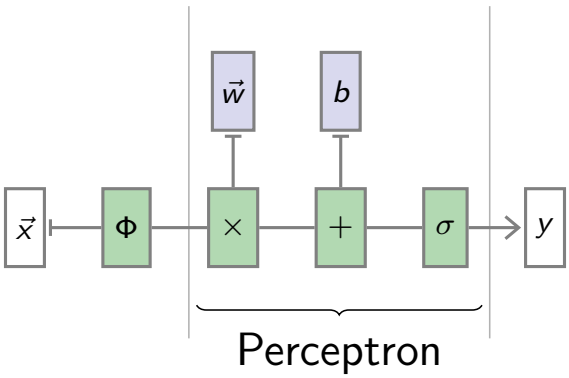
Solution possible pour surmonter limitation des modèles linéaires

L'exemple du xor peut être résolu en pré-traitant les données pour rendre les populations linéairement séparables.

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



Pré-traitement pour rendre séparable les entrées du perceptron



Analogie : régression polynomiale calculée par un pré-traitement suivie d'une régression linéaire

Pour prédire les valeurs d'une fonction $f : [0, 1] \to \mathbb{R}$, $x \mapsto f(x)$ à partir des paires $\left\{x^i, y^i = f\left(x^i\right)\right\}_{i=1,\ldots,n}$ on peut utiliser la régression polynomiale plutôt que linéaire : avec un degré D du polynôme suffisant, on peut approximer n'importe quelle fonction continue réelle sur un compact (théorème de Stone-Weierstrass).

Fixons la transformation :

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

et les paramètres de la régression

$$\alpha = (\alpha_0, \ldots, \alpha_D).$$

Alors, la régression polynomiale sur la variable x peut se faire par une régression *linéaire* sur les $D + 1$ variables de $\Phi(x)$:

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

Application à un problème de classification d'images

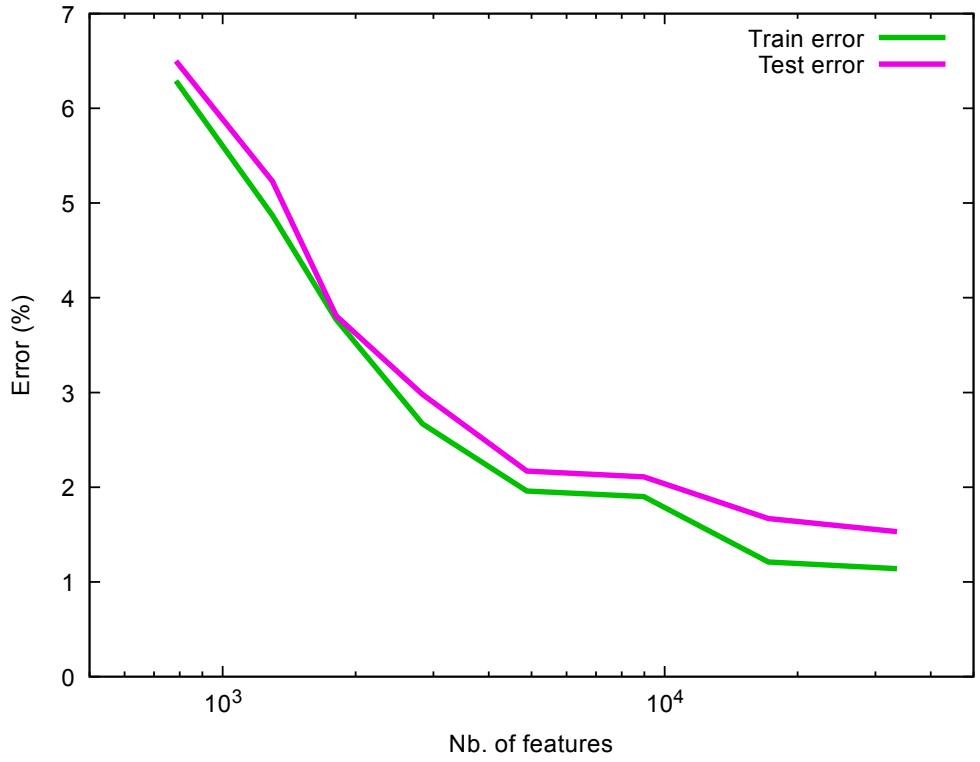
Nous pouvons appliquer la même idée à un problème un peu plus réaliste : séparer les "8" manuscrits de MNIST des autres chiffres avec un perceptron.

Nous rajoutons aux $784 = 28 \times 28$ caractéristiques originales (correspondant aux 28×28 pixels) des produits de paires de pixels, prises au hasard :

$$\Phi : \mathbb{R}^{28 \times 28} \to \mathbb{R}^{28^2 + K}$$

$$x \mapsto (\underbrace{x[1,1], x[1,2], \ldots, x[28,28]}_{\text{Pixels}}, \underbrace{x[i_1, j_1]x[i'_1, j'_1], \ldots, x[i_K, j_K]x[i'_K, j'_K]}_{K \text{ produits}})$$

Erreur en fonction du nombre de charactéristiques (pixels + pixels ajoutés)



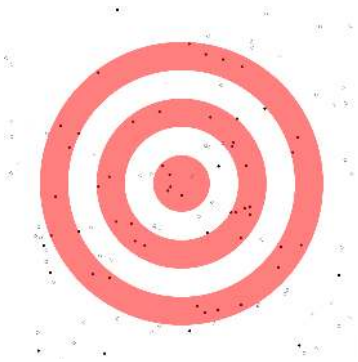
Avantages de l'augmentation avec des caractéristiques supplémentaires

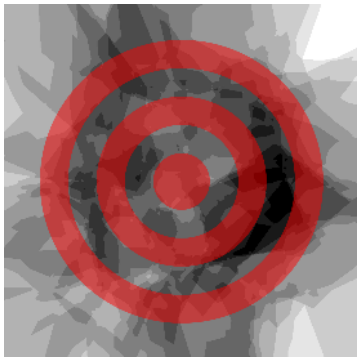
Au delà d'augmenter la capacité pour mieux approximer les données d'apprentissage, la conception de caractéristique est aussi une technique pour rendre le processus d'apprentissage plus robuste au bruit.

En particulier un bon pre-processing doit rendre le signal invariant à des perturbations si l'on sait que celles-ci ne doivent pas changer la prédiction.

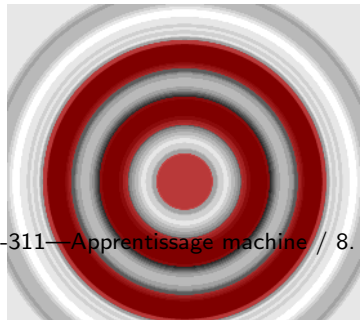
Nous pouvons illustrer l'utilisation de caractéristiques invariantes avec le plus proche voisin sur une tâche avec une symétrie radiale.

Points d'apprentissage Votes (K=11) Prédiction (K=11)





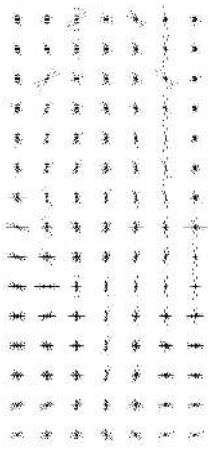
Using 2d coordinates



Application en vision par ordinateur

Un exemple classique de caractéristiques conçues pour fournir une invariance complexe sont les "Histogram of Oriented Gradient" (HOG).

Schématiquement : L'image est partitionnée en blocs de 8×8 pixels, et dans chacun est calculé un histogramme de l'orientation des caractéristiques saillantes (bords des objets) en 9 directions.



Dalal and Triggs (2005) ont combiné ces caractéristiques avec des machines à vecteurs de support pour faire de la détection de personnes.

Cette stratégie combinant des caractéristiques conçues à la mains et un prédicteur paramétrique comme la régression logistique ou une machine à vecteurs de support est souvent qualifiée en anglais de **“shallow learning”**.

Le signal traverse un seul traitement modulé par des paramètres optimisés sur les données.

Modèles profonds

Généralisation : considérons une sortie multi-dimensionnelle plutôt que 1D

Un classifieur linéaire à sortie uni-dimensionnelle de la forme

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

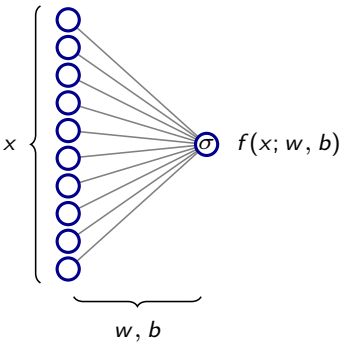
avec $\vec{w} \in \mathbb{R}^D$, $b \in \mathbb{R}$, et $\sigma : \mathbb{R} \to \mathbb{R}$, peut être généralisé à une sortie multi-dimensionnelle (note : on omet désormais la notation $\vec{\cdot}$) en spécifiant :

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

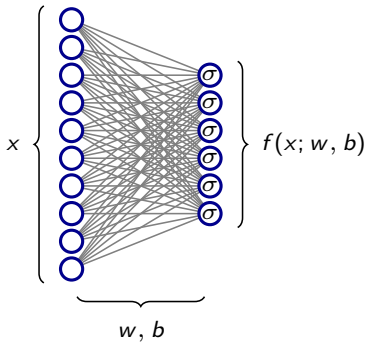
avec $w \in \mathbb{R}^{C \times D}$, $b \in \mathbb{R}^C$, et σ appliquée composante par composante.

Du perceptron simple au modèle multi-couches

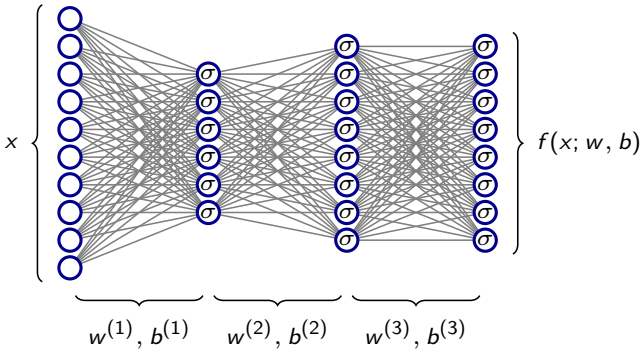
Un modèle à sortie multi-dimensionnelle et multi-couche peut être construit à partir d'une combinaison d'unités linéaires élémentaires, et peut être étendu.



Une seule unité



Une seule couche de plusieurs unités



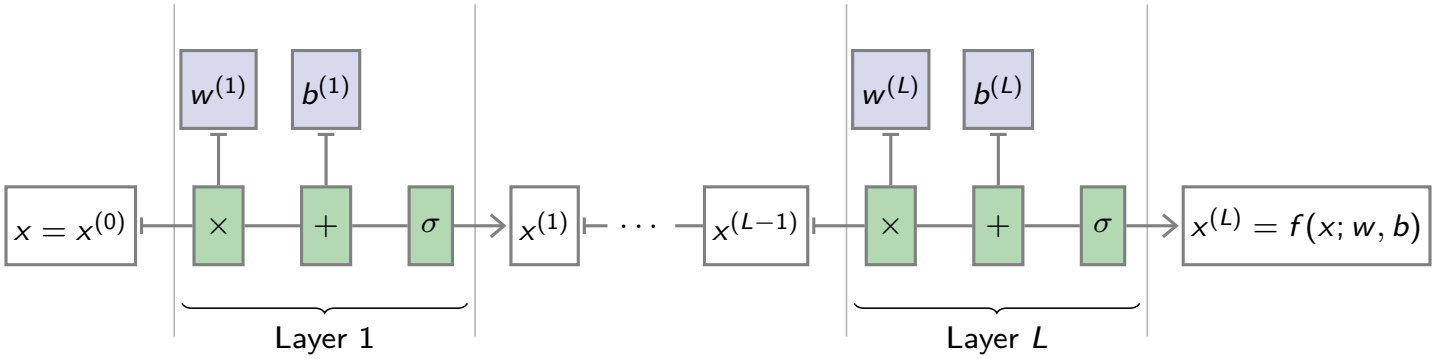
Plusieurs couches de plusieurs unités

Formalisation

Avec $x^{(0)} = x$, ce modèle peut être formalisé comme

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

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



C'est un **perceptron multi-couche.**

Importance de l'activation non-linéaire

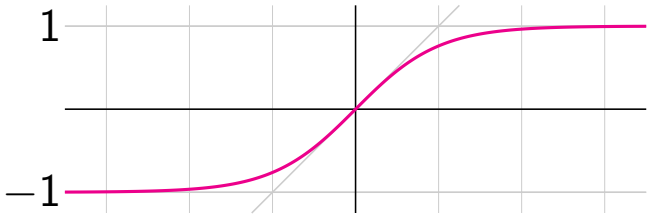
Il est important de remarquer que si σ est une transformation affine, le modèle complet est une composition de transformations affines, et donc lui même une transformation affine.

⚠ **La fonction d'activation σ doit donc être non-linéaire**, sinon le modèle complet se réduira à un simple modèle affine avec une paramétrisation inhabituelle.

Fonctions d'activation classiques

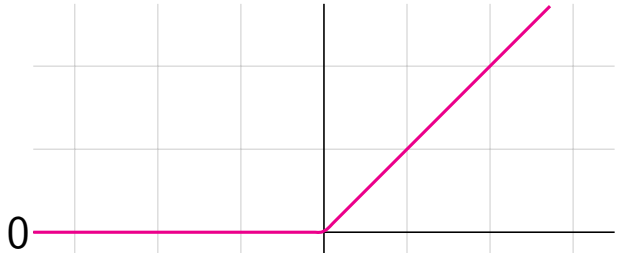
Les deux fonctions d'activation classiques sont la tangente hyperbolique

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



et la *Rectified Linear Unit* (ReLU)

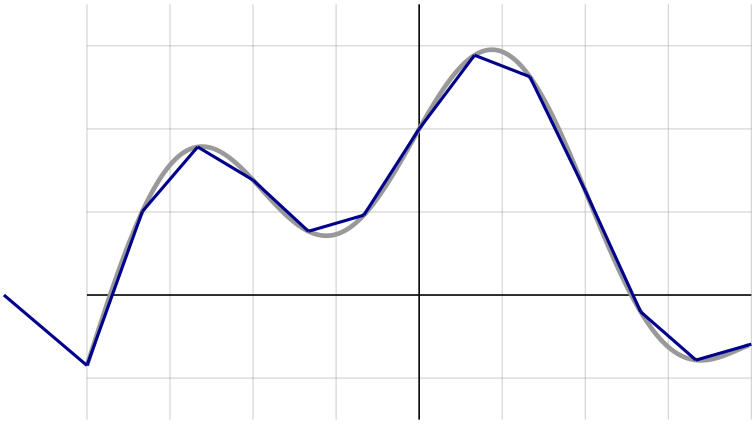
$$x \mapsto \max(0, x)$$



Approximation universelle

N'importe quelle $\psi \in \mathscr{C}([a,b],\mathbb{R})$ peut être approximée avec une combinaison linéaire de ReLU scalées / translatées.

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



C'est également vrai pour d'autres fonctions d'activation sous des hypothèses raisonnables.

Pour étendre ce résultat au cas multi-dimensionnel, $\psi \in \mathscr{C}([0,1]^D,\mathbb{R})$, on peut d'abord approximer sin avec une combinaison de ReLU, et utiliser la densité des séries de Fourier pour obtenir le résultat final :

$$\forall \epsilon > 0, \exists K, w \in \mathbb{R}^{K \times D}, b \in \mathbb{R}^K, \omega \in \mathbb{R}^K, \text{ tel que}$$
$$\max_{x \in [0,1]^D} |\psi(x) - \omega \cdot \sigma(w\,x + b)| \leq \epsilon$$

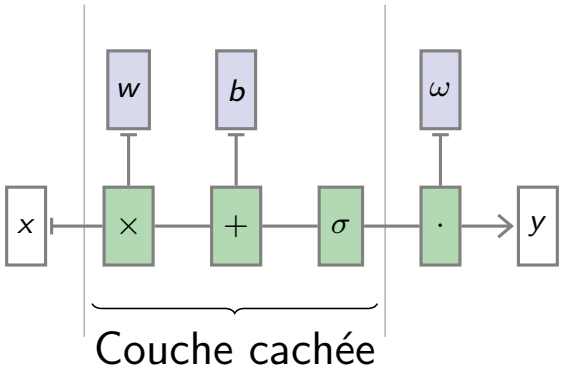
Nous pouvons donc approximer n'importe quelle fonction continue

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

avec un perceptron à une couche cachée

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

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



C'est le **théorème d'approximation universelle.**

Il est important de noter que ce théorème ne dit rien sur la dimension de la représentation dans la couche cachée.

Donc bien qu'il soit important, en particulier pour montrer que ces modèles ne souffrent pas de la limitation fondamentale des modèles linéaire, il ne donne pas d'indice sur le coût computationnel ou le sur-apprentissage.

Descente de gradient

Rappel : la fonction de coût

Nous avons vu qu'entraîner un modèle consiste à trouver des valeurs pour ses paramètres qui minimisent une fonction de coût comme, par exemple, l'erreur quadratique moyenne :

$$\mathscr{L}(w, b) = \frac{1}{n} \sum_{i=1}^{n} \left(f(x^i; w, b) - y^i \right)^2 .$$

D'autres fonctions sont plus adaptées à des problèmes de classification, certaines formes de régression ou d'estimation de densité.

Nous avons aussi vu que la fonction de coût peut être minimisée avec des techniques exactes, par exemple dans le cas quadratique, ou avec des procédures *ad hoc* comme dans le cas de l'erreur empirique pour le perceptron.

Rappel : fonction coût pour régression logistique

Il n'y a généralement pas de méthode *ad hoc*. Nous avons vu qu'à la régression logistique par exemple

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

correspond la fonction de coût

$$\mathscr{L}(w, b) = -\sum_{i=1}^{n} \log \sigma(y^i(w \cdot x^i + b))$$

qui ne peut pas être minimisée analytiquement.

Comme nous l'avons vu, la méthode générale est la descente de gradient.

Rappel : définition du gradient

Étant donnée une fonction

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

son gradient est

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

Descente de gradient

Pour minimiser

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

la descente de gradient utilise une approximation locale linéaire pour progresser itérativement vers un minimum.
Pour $w_0 \in \mathbb{R}^D$, considérons une approximation de $\mathscr{L}$ dans le voisinage de w_0

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

avec un terme quadratique qui ne dépend pas de $\mathscr{L}$.
Calculons le gradient p.r. à w de cette approximation $\tilde{\mathscr{L}}_{w_0}(w)$:

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

qui implique (en trouvant la solution de $\nabla\tilde{\mathscr{L}}_{w_0}(w) = 0$) :

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

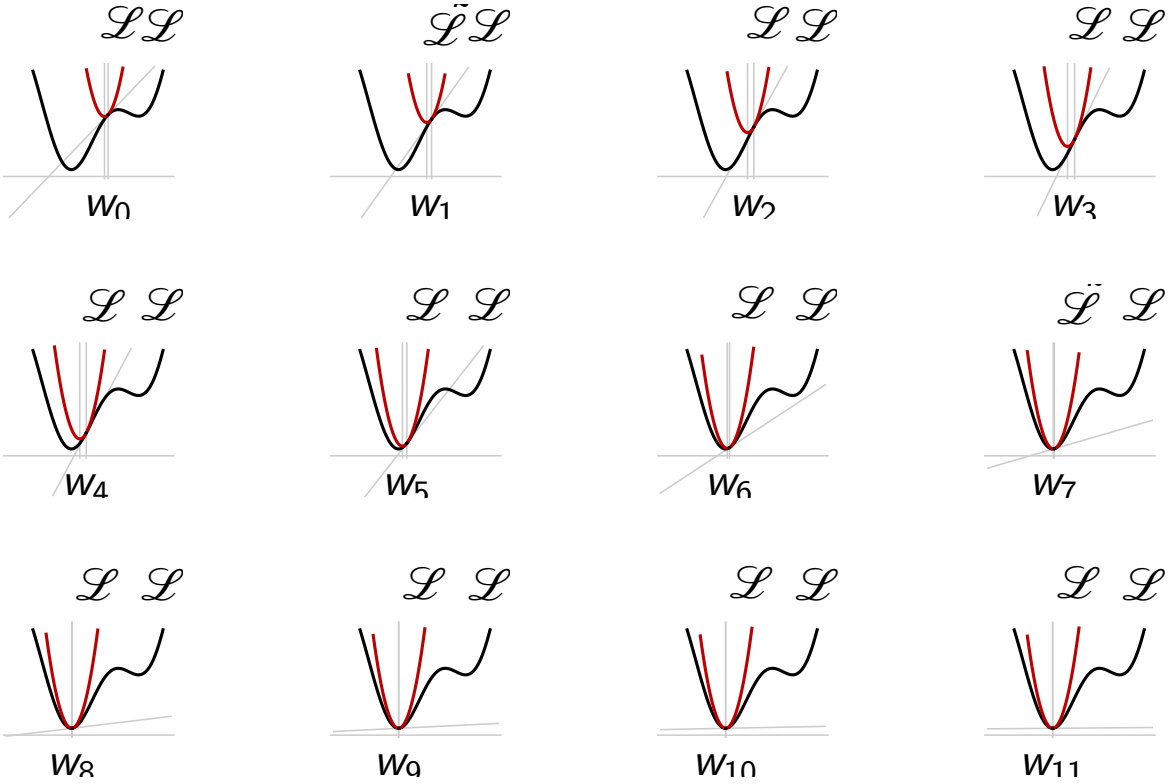
L'algorithme qui en découle met w à jour itérativement de la manière suivante :

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

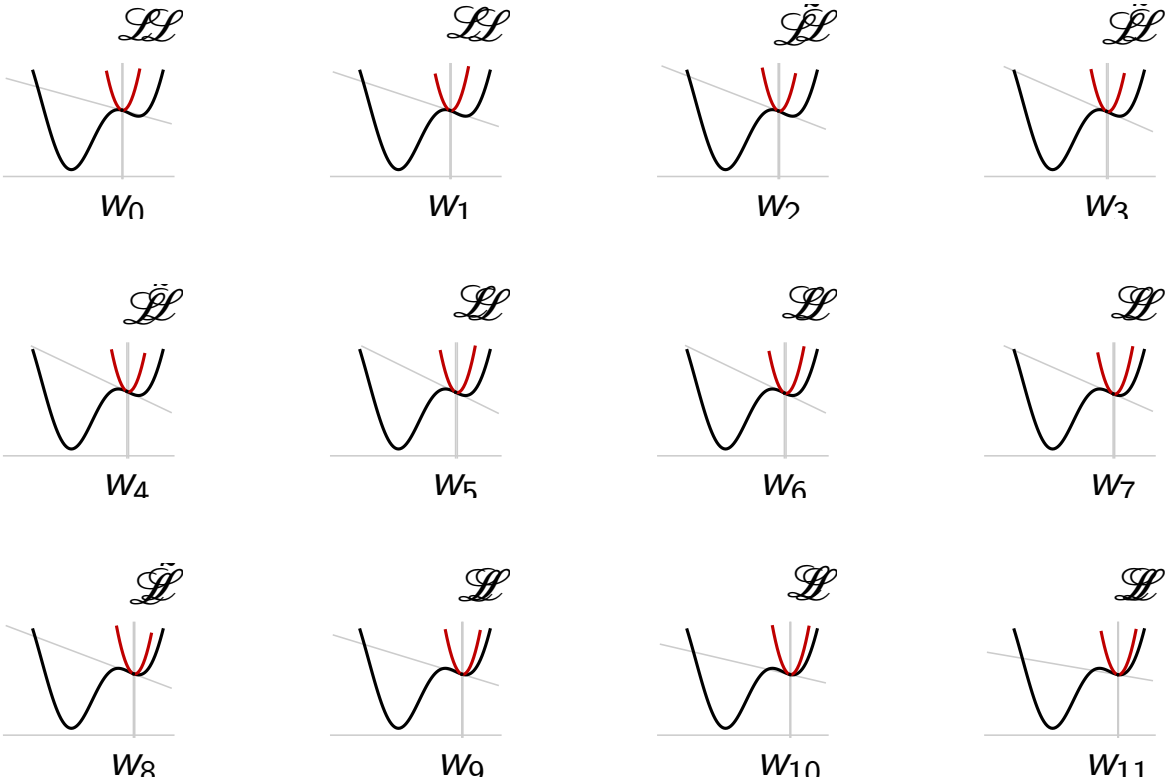
qui peut être interpretée comme "suivre la direction de la pente la plus forte".

Cette procédure conduit [la plupart du temps] à un **minimum local**, et les choix de w_0 et η sont importants.

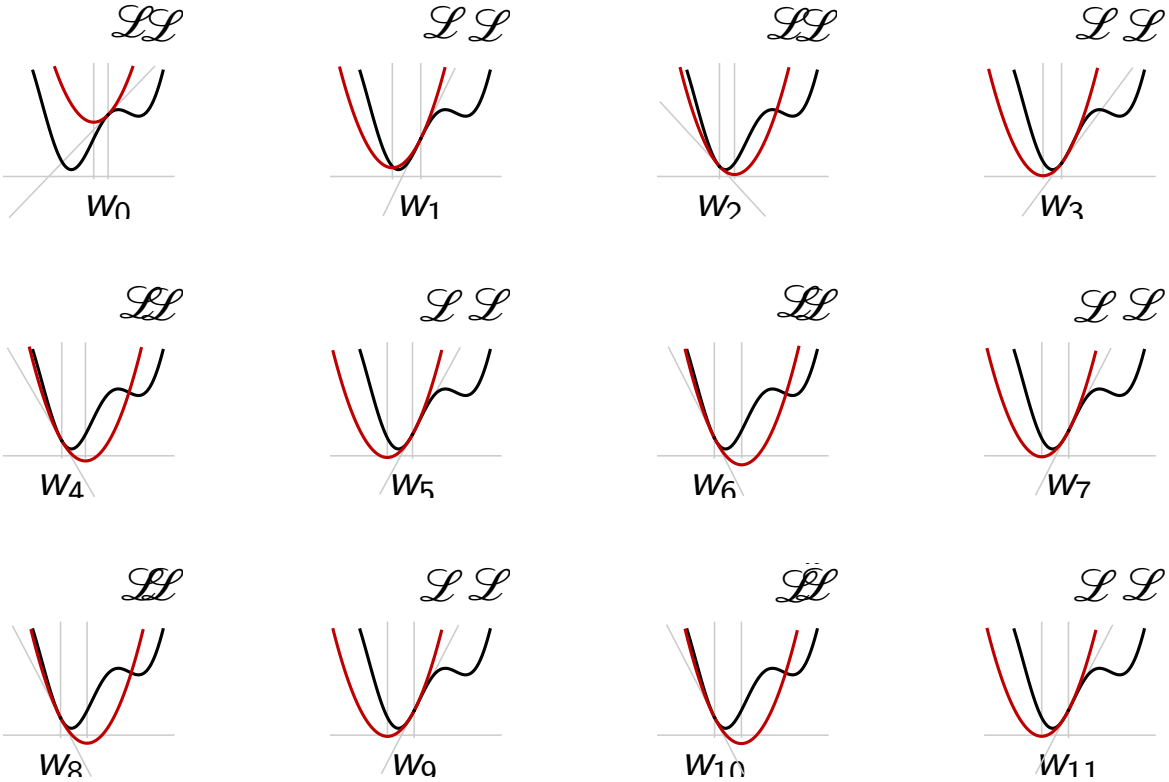
$\eta = 0.125$

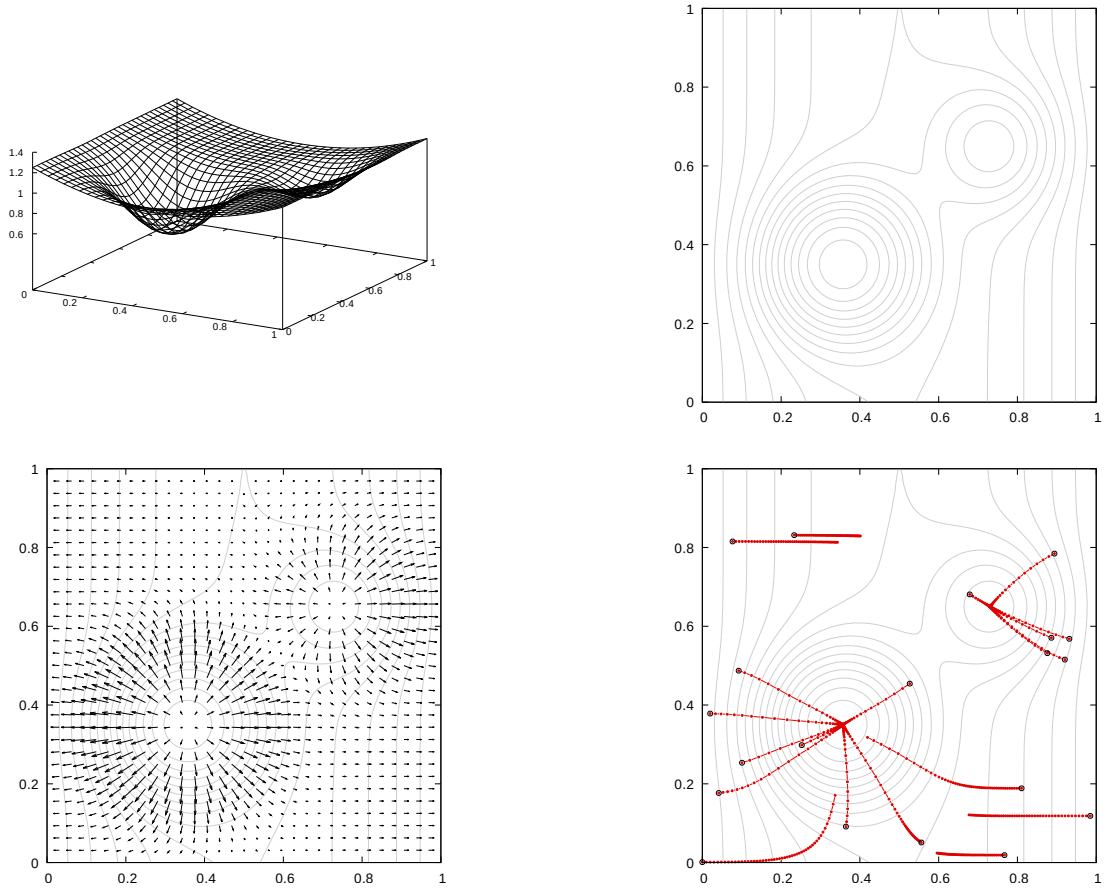


$\eta = 0.125$



$\eta = 0.5$





Nous avons vu que le minimum de la fonction de coût de le régression logistique

$$\mathscr{L}(w, b) = -\sum_{i=1}^{n} \log \sigma(y^i(w \cdot x^i + b))$$

```
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()
```

n'a pas de forme analytique.

Formulaire de dérivées, fonctions logarithme et sigmoïde

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

$$\frac{\mathrm{d}}{\mathrm{d}x} \log(x) = \frac{1}{x}, \quad x > 0$$

$$\frac{\mathrm{d}}{\mathrm{d}x} \sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$\frac{\mathrm{d}}{\mathrm{d}x} \log(\sigma(x)) = (1 - \sigma(x)) = \sigma(-x)$$

Nous pouvons dériver

$$\frac{\partial \mathscr{L}}{\partial b} = -\sum_{i=1}^{n} \underbrace{y^i \, \sigma(-y^i(w \cdot x^i + b))}_{u_i},$$

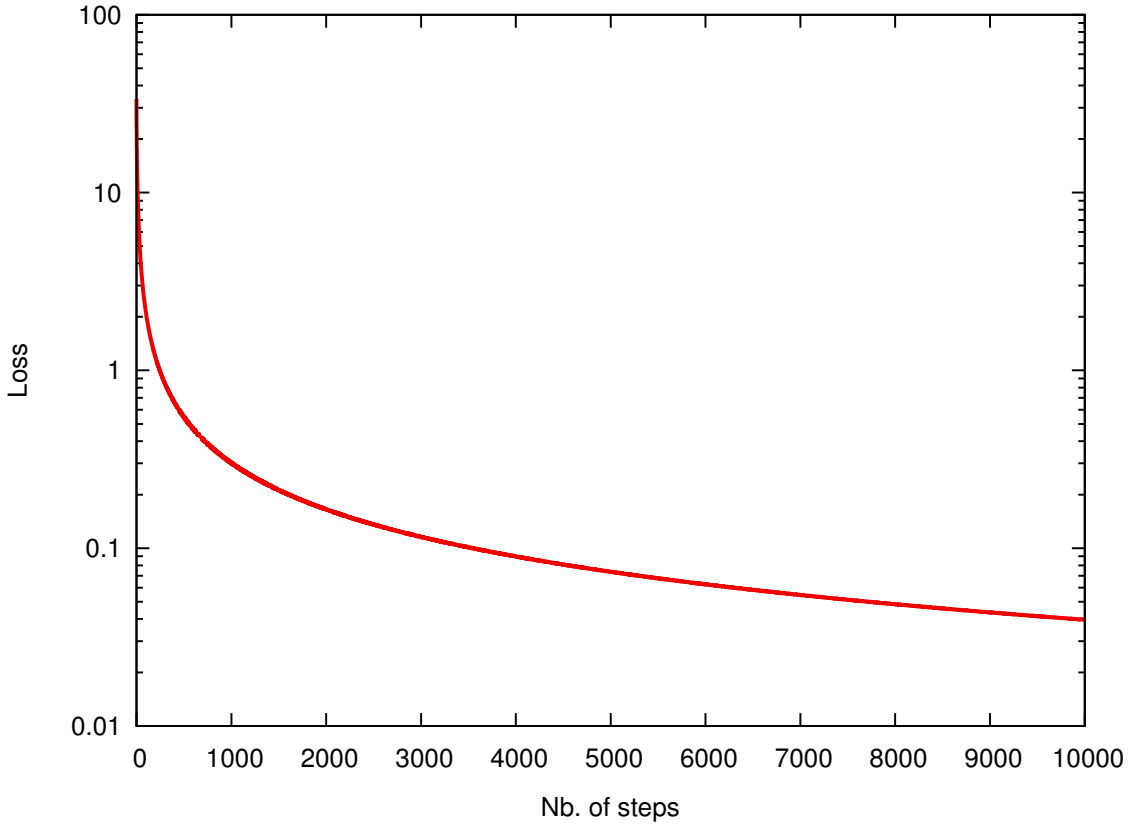
$$\forall d, \ \frac{\partial \mathscr{L}}{\partial w_d} = -\sum_{i=1}^{n} \underbrace{x_d^i \, y^i \, \sigma(-y^i(w \cdot x^i + b))}_{v_d^i},$$

qui peut être implémenté avec

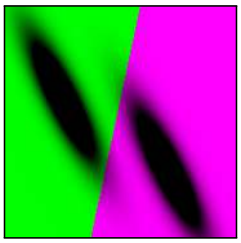
```
def gradient(x, y, w, b):
    u = y * sigmoid(- y * (x.dot(w) + b))
    v = x * u.reshape(-1, 1)
    return - v.sum(0), - u.sum(0)
```

et utilisé dans une descente de gradient de la forme

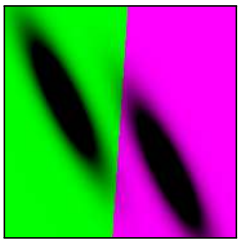
```
w, b = numpy.random.normal(0, 1, (x.shape[1],)), 0
eta = 1e-1

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
```

Avec 100 points d'apprentissage et $\eta = 10^{-1}$.

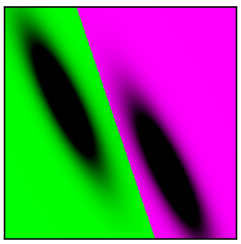


$n = 0$



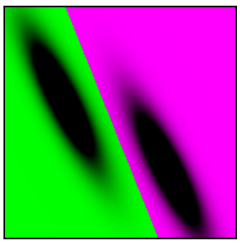
$n = 10$

$n = 10^2$



$n = 10^3$

$n = 10^4$



Solution par Linear Disciminant Analysis (optimal pour ce problème)

Entraînement du perceptron multi-couche

Entraînement d'un perceptron multi-couche à l'aide de la descente de gradient

Nous voulons entraîner un perceptron multi-couche (=ajuster tous ses paramètres) en minimisant un coût calculé sur un ensemble d'apprentissage :

$$\mathscr{L}(w,b) = \sum_{i=1}^{n} \boldsymbol{\ell}(f(x^i; w, b), y^i).$$

Pour utiliser la descente de gradient, nous devons calculer le gradient de la fonction de coût sur chaque exemple par rapport aux paramètres.

C'est à dire, avec $\boldsymbol{\ell}_i = \boldsymbol{\ell}(f(x^i; w, b), y^i)$,

$$\frac{\partial \boldsymbol{\ell}_i}{\partial w_{j,k}^{(\ell)}} \quad \text{et} \quad \frac{\partial \boldsymbol{\ell}_i}{\partial b_j^{(\ell)}}.$$

Propagation vers l'avant : calcul des sorties aux noeuds

Nous considérons un seul exemple d'apprentissage x, et nous introduisons $s^{(1)}, \ldots, s^{(L)}$ comme les sommations avant les fonctions d'activation.

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

Nous posons $x^{(0)} = x$,

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

et nous définissons la sortie du réseau comme $f(x; w, b) = x^{(L)}$.

Ceci est la **propagation vers l'avant.**

Rétro-propagation du gradient : méthode pour calculer les composantes du gradient

L'algorithme de rétro-propagation du gradient n'est qu'une application directe de la dérivation des fonctions composées :

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

L'approximation linéaire d'une composition de fonction est le produit de leurs approximations linéaires.

Ceci se généralise à des compositions plus complexe en grandes dimensions

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

où $J_f(x)$ est le Jacobien de f en x, c'est à dire la matrice de l'approximation linéaire de f dans le voisinage de x.

Rappel d'analyse : règle de composition de fonctions pour une variable

Soit $x = g(t)$ et $y = h(t)$ des fonctions différentiables de t et $z = f(x, y)$ une fonction différentiable de x et y.

Alors $z = f\left(x(t), y(t)\right)$ est une fonction différentiable de t et

$$\frac{dz}{dt} = \frac{\partial z}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial z}{\partial y} \cdot \frac{dy}{dt}$$

Rappel d'analyse : règle de composition des dérivées pour deux variables indépendantes

Soient $x = g(u, v)$ et $y = h(u, v)$ des fonctions différentiables de u et v and $z = f(x, y)$ une fonction différentiable de x et y.

Alors, $f\left(g(u, v), h(u, v)\right)$ est une fonction différentiable de u et v et

$$\frac{dz}{du} = \frac{\partial z}{\partial x} \cdot \frac{dx}{du} + \frac{\partial z}{\partial y} \cdot \frac{dy}{du}$$

ainsi que

$$\frac{dz}{dv} = \frac{\partial z}{\partial x} \cdot \frac{dx}{dv} + \frac{\partial z}{\partial y} \cdot \frac{dy}{dv}$$

Rétro-propagation du gradient : dérivées du coût ℓ par rapport à $s_j^{(\ell)}$ et $x_k^{(\ell-1)}$

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

Comme $s_j^{(\ell)}$ n'influence ℓ que via $x_j^{(\ell)}$ avec

$$x_j^{(\ell)} = \sigma\left(s_j^{(\ell)}\right),$$

nous avons

$$\frac{\partial \ell}{\partial s_j^{(\ell)}} = \frac{\partial \ell}{\partial x_j^{(\ell)}} \frac{\partial x_j^{(\ell)}}{\partial s_j^{(\ell)}} = \frac{\partial \ell}{\partial x_j^{(\ell)}} \sigma'\left(s_j^{(\ell)}\right).$$

Et comme $x_k^{(\ell-1)}$ n'influence ℓ que via les $s_j^{(\ell)}$ avec

$$s_j^{(\ell)} = \sum_k w_{j,k}^{(\ell)} x_k^{(\ell-1)} + b_j^{(\ell)},$$

nous avons

$$\frac{\partial \ell}{\partial x_k^{(\ell-1)}} = \sum_j \frac{\partial \ell}{\partial s_j^{(\ell)}} \frac{\partial s_j^{(\ell)}}{\partial x_k^{(\ell-1)}} = \sum_j \frac{\partial \ell}{\partial s_j^{(\ell)}} w_{j,k}^{(\ell)}.$$

Rétro-propagation du gradient : dérivées du coût ℓ par rapport aux paramètres $w_{j,k}^{(\ell)}$ et $b_\ell^{(\ell)}$

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

Comme $w_{j,k}^{(\ell)}$ et $b_j^{(\ell)}$ n'influencent ℓ que via $s_j^{(\ell)}$ avec

$$s_j^{(\ell)} = \sum_k w_{j,k}^{(\ell)} x_k^{(\ell-1)} + b_j^{(\ell)},$$

nous avons

$$\frac{\partial \ell}{\partial w_{j,k}^{(\ell)}} = \frac{\partial \ell}{\partial s_j^{(\ell)}} \frac{\partial s_j^{(\ell)}}{\partial w_{j,k}^{(\ell)}} = \frac{\partial \ell}{\partial s_j^{(\ell)}} x_k^{(\ell-1)},$$
$$\frac{\partial \ell}{\partial b_j^{(\ell)}} = \frac{\partial \ell}{\partial s_j^{(\ell)}}.$$

Rétro-propagation du gradient : résumé

Pour résumer : nous pouvons calculer les $\frac{\partial \ell}{\partial x_j^{(L)}}$ à partir de la définition de ℓ, et récursivement **propager vers l'arrière** les dérivées du coût par rapport aux *activations* avec :

$$\frac{\partial \ell}{\partial s_j^{(\ell)}} = \frac{\partial \ell}{\partial x_j^{(\ell)}} \sigma'\left(s_j^{(\ell)}\right)$$
$$\frac{\partial \ell}{\partial x_k^{(\ell-1)}} = \sum_j \frac{\partial \ell}{\partial s_j^{(\ell)}} w_{j,k}^{(\ell)}.$$

Ensuite, nous pouvons calculer les dérivées par rapport aux paramètres du modèle avec :

$$\frac{\partial \ell}{\partial w_{j,k}^{(\ell)}} = \frac{\partial \ell}{\partial s_j^{(\ell)}} x_k^{(\ell-1)}$$
$$\frac{\partial \ell}{\partial b_j^{(\ell)}} = \frac{\partial \ell}{\partial s_j^{(\ell)}}.$$

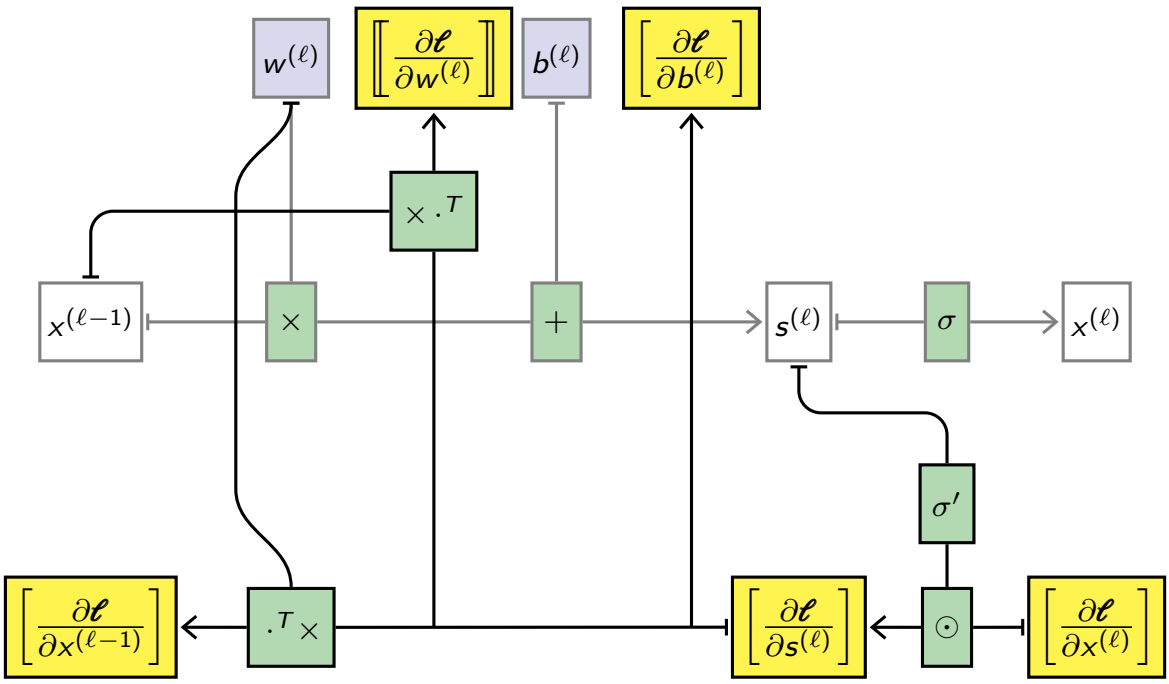
Écriture tensorielle

Pour écrire tout cela de manière tensorielle, si $\psi : \mathbb{R}^N \to \mathbb{R}^M$, nous utiliserons une notation standard du Jacobien

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

et si $\psi : \mathbb{R}^{N \times M} \to \mathbb{R}$, nous utiliserons

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



Propagation vers l'avant

Calculer les activations.

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

Rétro-propagation du gradient

Calculer les dérivées du coût par rapport aux activations :

$$\begin{cases} \left[\frac{\partial \boldsymbol{\ell}}{\partial x^{(L)}}\right] \text{ d'après la définition de } \boldsymbol{\ell} \\ \text{si } \ell < L, \left[\frac{\partial \boldsymbol{\ell}}{\partial x^{(\ell)}}\right] = \left(w^{(\ell+1)}\right)^T \left[\frac{\partial \boldsymbol{\ell}}{\partial s^{(\ell+1)}}\right] \end{cases} \qquad \left[\frac{\partial \boldsymbol{\ell}}{\partial s^{(\ell)}}\right] = \left[\frac{\partial \boldsymbol{\ell}}{\partial x^{(\ell)}}\right] \odot \sigma'\left(s^{(\ell)}\right)$$

Calculer les dérivées du coût par rapport aux paramètres :

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

Itération de descente de gradient

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

Remarques sur la rétro-propagation du gradient

Bien que la rétro-propagation du gradient semble être complexe, elle consiste finalement à calculer des dérivées de fonctions composées.

Comme la propagation vers l'avant, elle peut être exprimée de manière tensorielle. Les calculs lourds et paramétrés sont limités à des opérations linéaires, et les non linéarités sont calculées par composante individuelle.

Coût computationnel

En ce qui concerne le coût computationnel, comme les opérations coûteuses sont, pour de la propagation vers l'avant :

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

et pour la rétro-propagation du gradient :

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

et

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

une approximation grossière est que la rétro-propagation du gradient est deux fois plus coûteuse que la propagation vers l'avant.

Interprétation des opérations dans un réseau

Plus que pour d'autres modèles, il est très difficile d'interpréter les opérations qui sont finalement exécutées par chaque unité d'un perceptron multi-couche une fois qu'il est entraîné.

Exemple : classification binaire, données à 2 variables (1/2)

Considérons un réseau avec une seule couche cachée qui fait une classification à deux classes dans $\mathbb{R}^2$, c'est à dire

$$\mathbb{R}^2 \to \mathbb{R}^2.$$

avec

$$f(x; w, b) = \sigma\left(w^{(2)}\sigma\left(w^{(1)}x + b^{(1)}\right) + b^{(2)}\right).$$

Si ce modèle a D unités cachées, alors

$$w^{(1)} \in \mathbb{R}^{D\times 2}, b^{(1)} \in \mathbb{R}^{D}, w^{(2)} \in \mathbb{R}^{2\times D}, b^{(2)} \in \mathbb{R}^{2}.$$

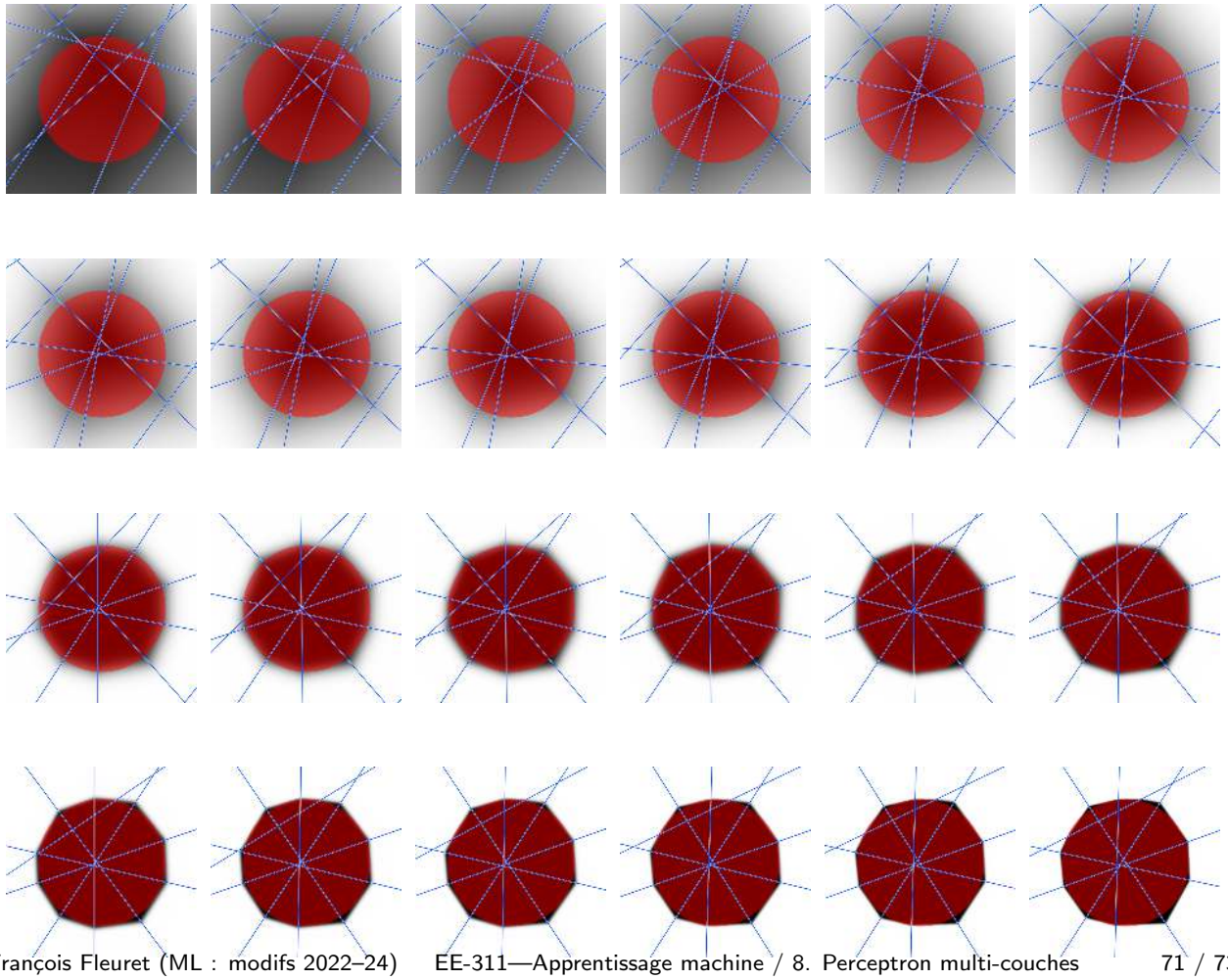
Exemple : classification binaire, données à 2 variables (2/2)

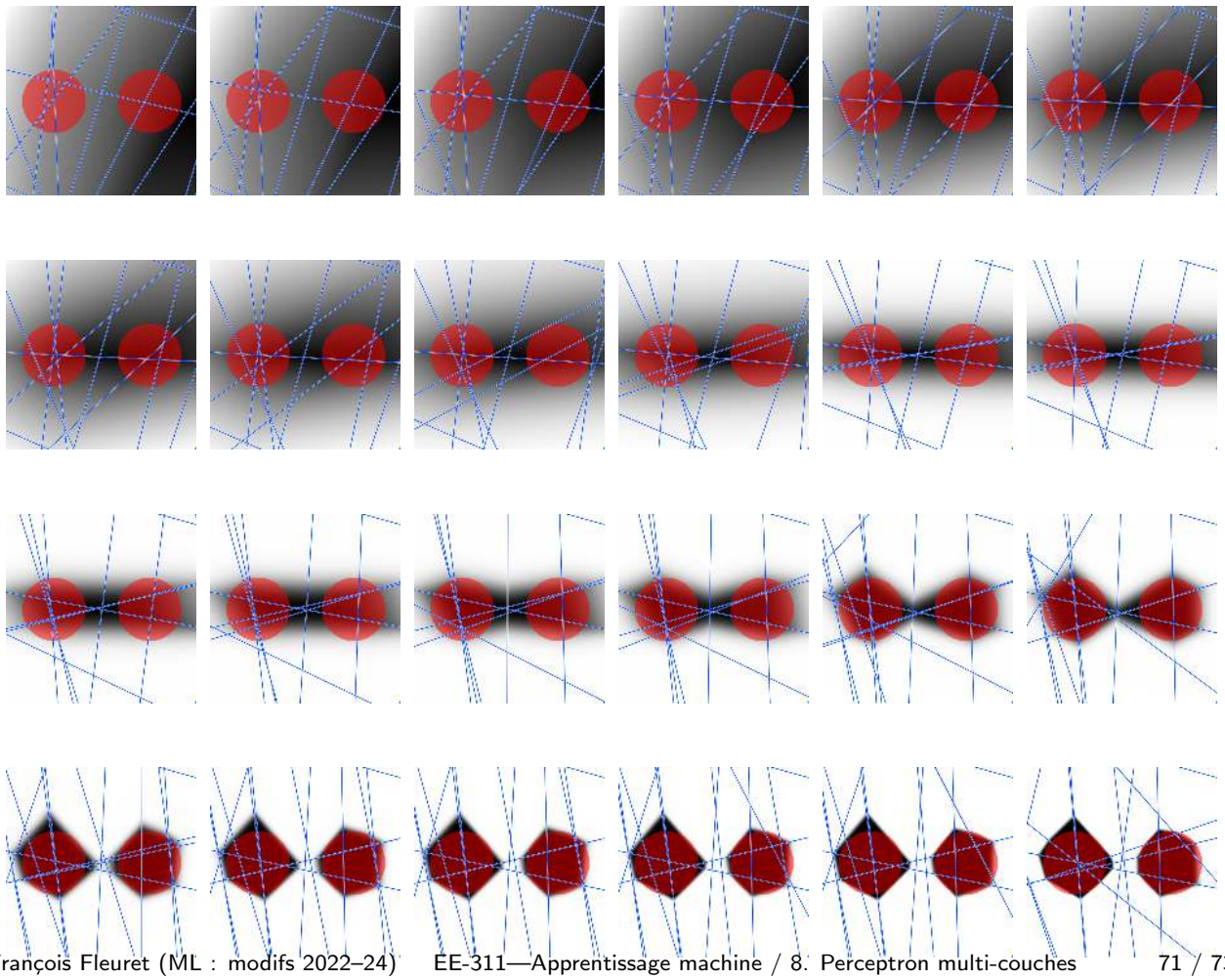
Les activations de la couche cachée pré-non-linéarité sont donc

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

et à chacune des $d = 1, \ldots, D$ composantes de s peut être associé un hyperplan de $\mathbb{R}^2$, donc une droite :

$$H^d = \left\{x \in \mathbb{R}^2 : w_d^{(1)} \cdot x + b_d^{(1)} = 0\right\}.$$





Perceptron et réseaux de neurones

- origine des méthodes avec neurones artificiels remontent aux années 1940
- perceptron : somme pondérée et biais avec un seuillage
- unités simples, mais peuvent être interconnectées
- pour modéliser un système complexe, les paramètre doivent être ajustés
- notation graphique ou tensorielle
- algorithme du perceptron permet d'ajuster les poids (e.g. pour la classification)
- garantie de convergence pour données séparables
- limitation des modèles linéaires : surmonté par l'ajout de variables (apprentissage "shallow")
- apprentissage profond :
  1. entrées-sorties multidimensionnelles
  2. multiple couches
  3. fonctions d'activation non-linéaires
- ajustement des paramètres par descente de gradient
- calcul des activations par propagation en avant
- calcul du gradient par rétro-propagation
- réseaux efficace et mais difficilement interprétable

Guide de lecture pour ce cours

Chloé-Agathe Azencott "Introduction au Machine Learning",
Dunod, 2019, ISBN 978-210-080153-4
Chapitre 7 : Réseaux de neurones artificiels

References

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

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.