Apprentissage Automatique

Réseaux de neurones

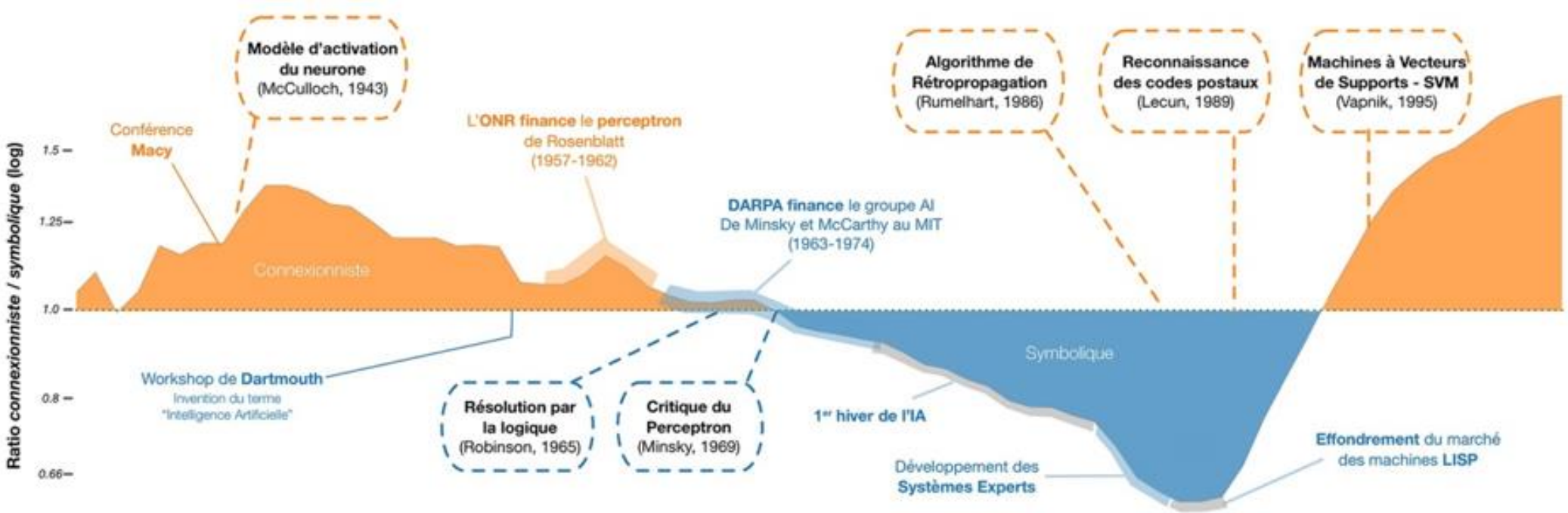
Stéphane Herbin

stephane.herbin@onera.fr

Aujourd'hui

- Réseaux de neurones (ANN = « Artificial Neural Network »)
  - Historique
  - La structure fonctionnelle: du neurone au réseau
  - Le multi couche
  - ANN et approximation universelle
  - ANN et décision: le multi-classe
  - Apprentissage et optimisation: fonction de coût et gradient stochastique
  - Back propagation et dérivation automatique
- TD
  - Familiarisation avec les concepts: applet Tensorflow
  - Programmation sous pytorch
  - Mise en œuvre sur problème de classification simple

Historique

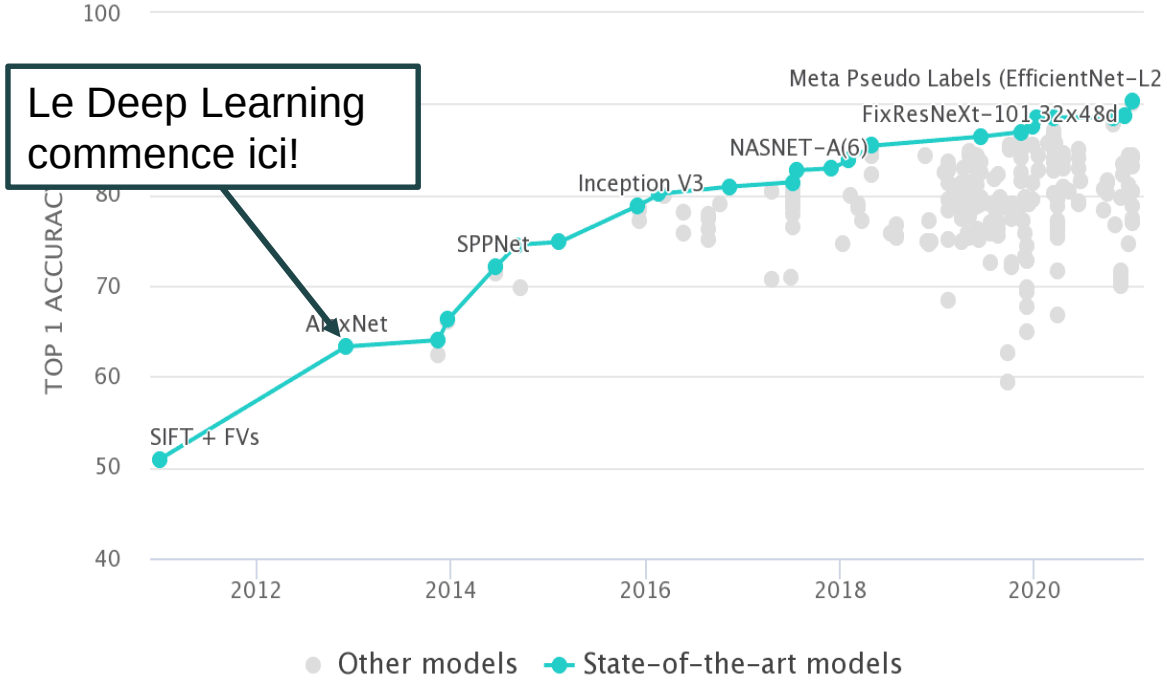


IA symbolique et RN: deux paradigmes de modélisation de l'intelligence

ONERA
THE FRENCH AEROSPACE LAB

La « révolution » du Deep Learning

Image Classification on ImageNet (top 1)



Données

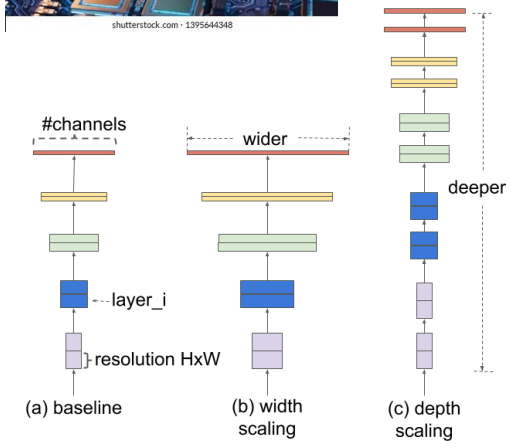
14M images, 1000 classes

Logiciels

Spark, orange, WEKA, mxnet, scikit learn, TensorFlow, Keras, CNTK, PYTORCH, R

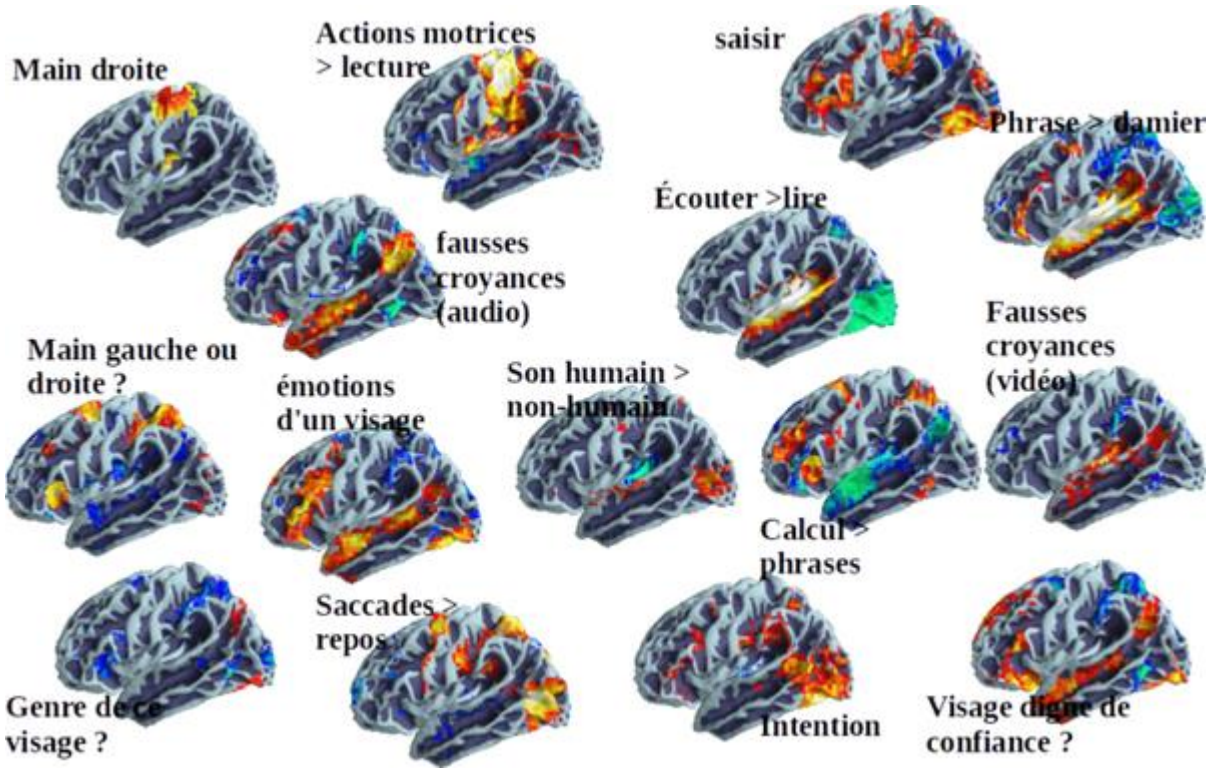
Hardware

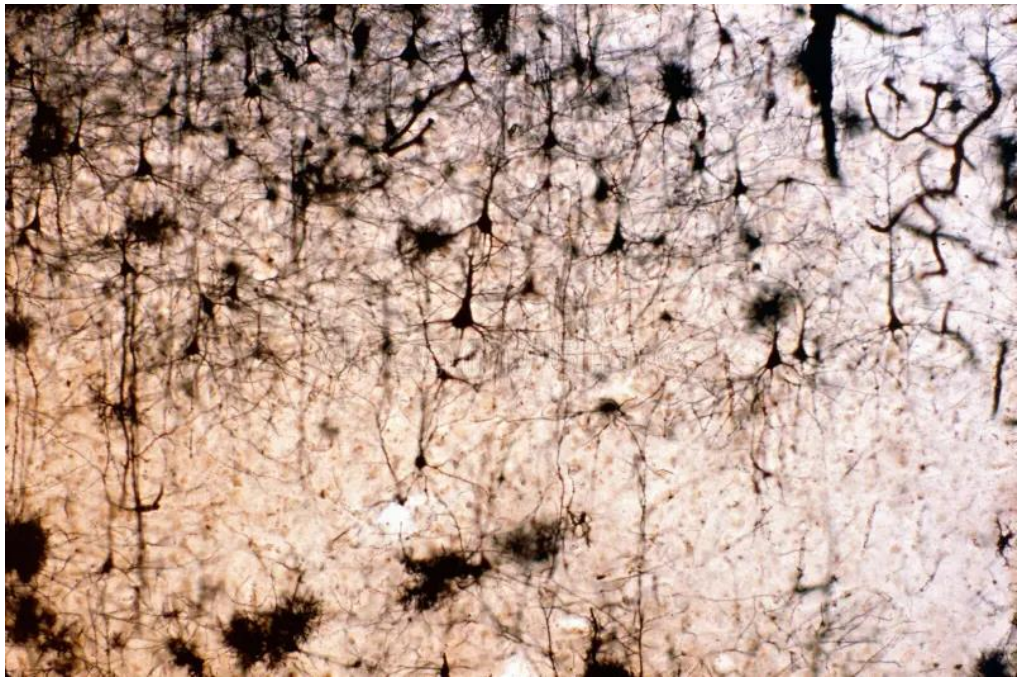
Algorithmes



ONERA THE FRENCH AEROSPACE LAB

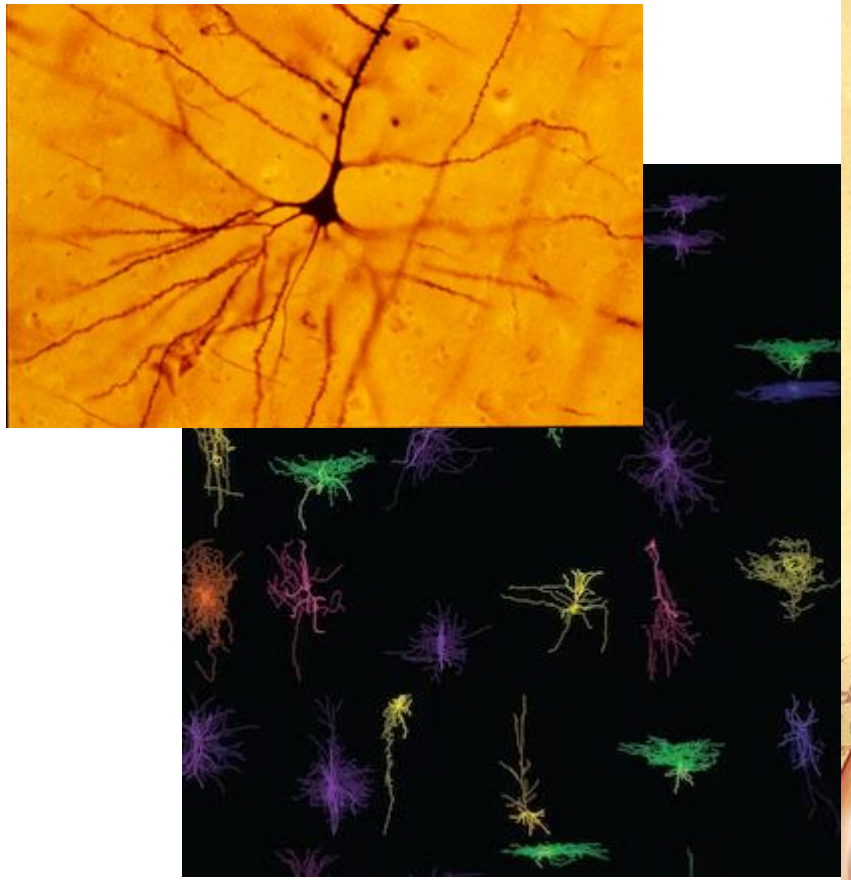
Inspiration biologique

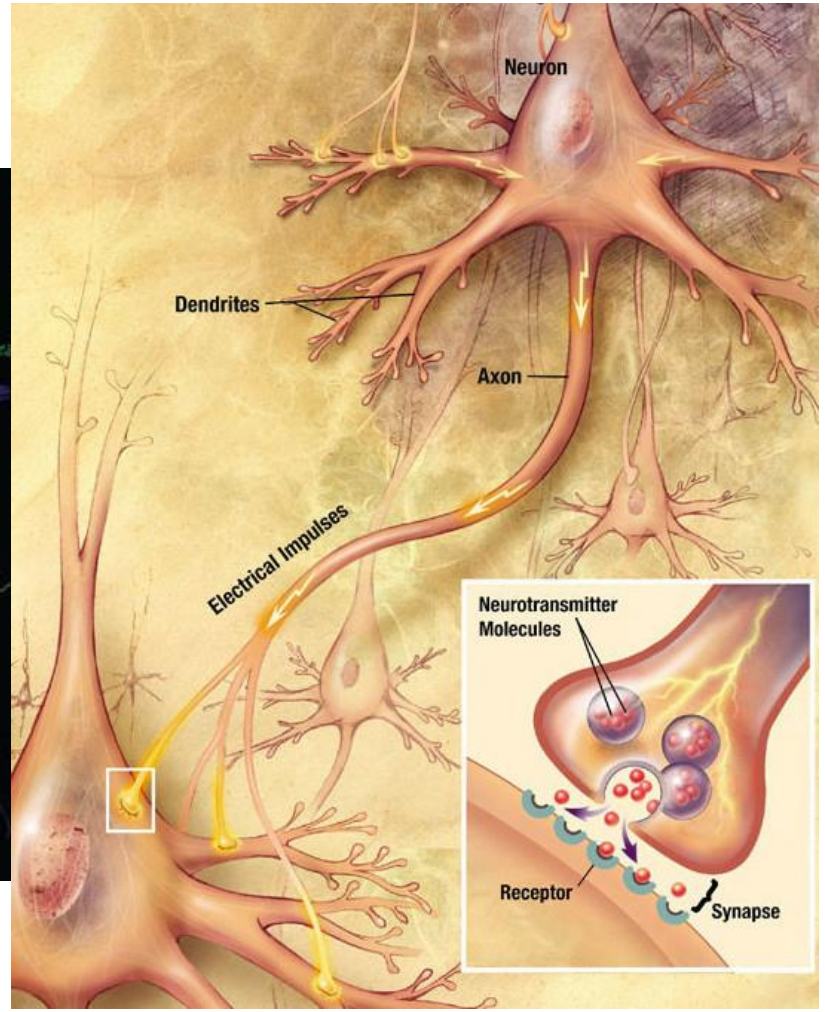


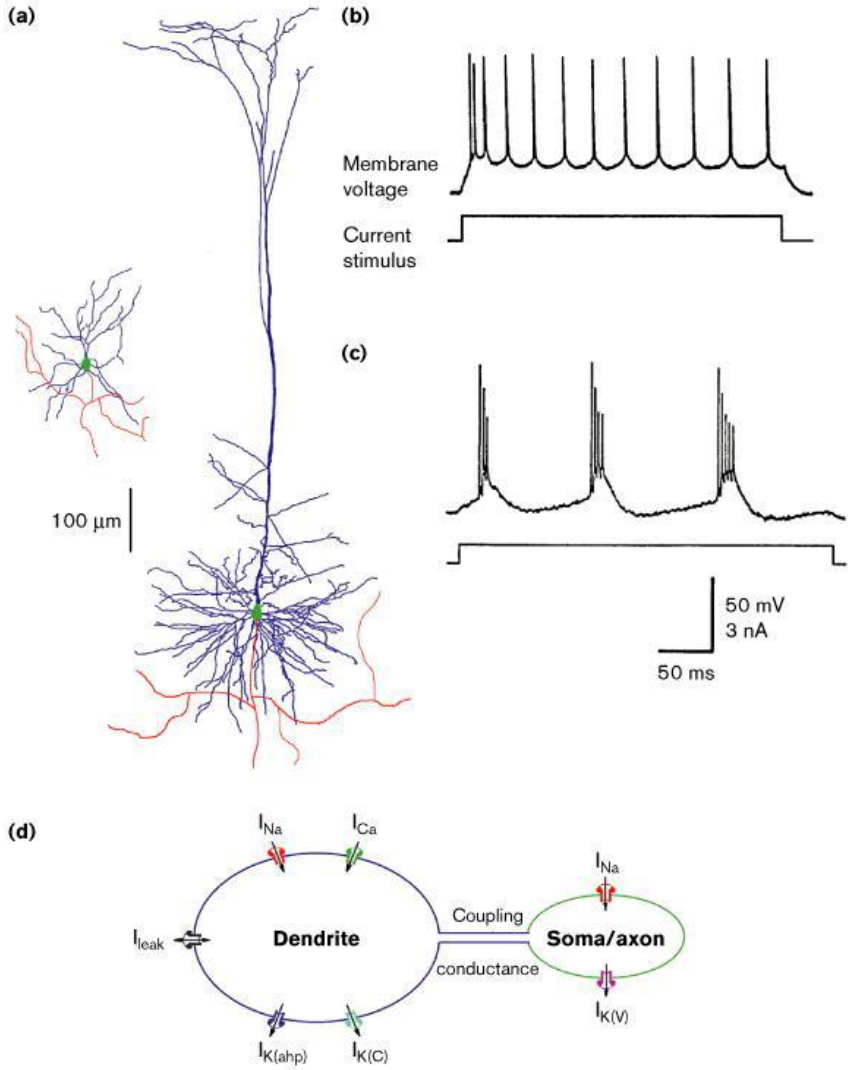


- Cerveau = organe de la pensée
- Il est constitué de neurones organisés en réseaux
- Il doit y avoir de l'intelligence là dedans!

Le neurone biologique





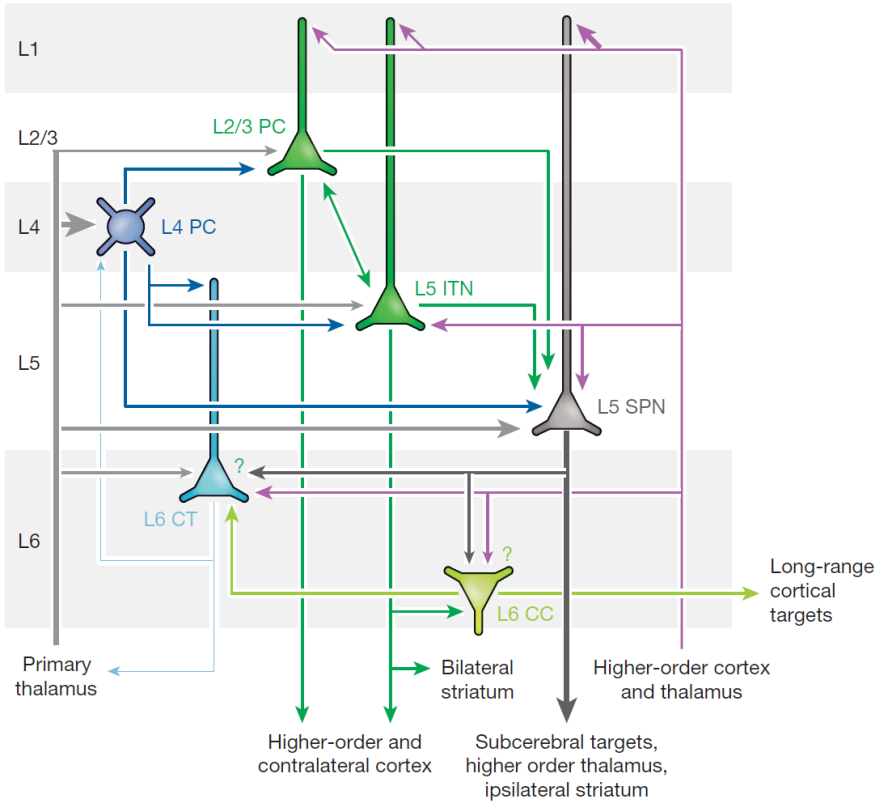


Processus de calcul locaux & connexions synaptiques

Codage « électrochimique » de l'information

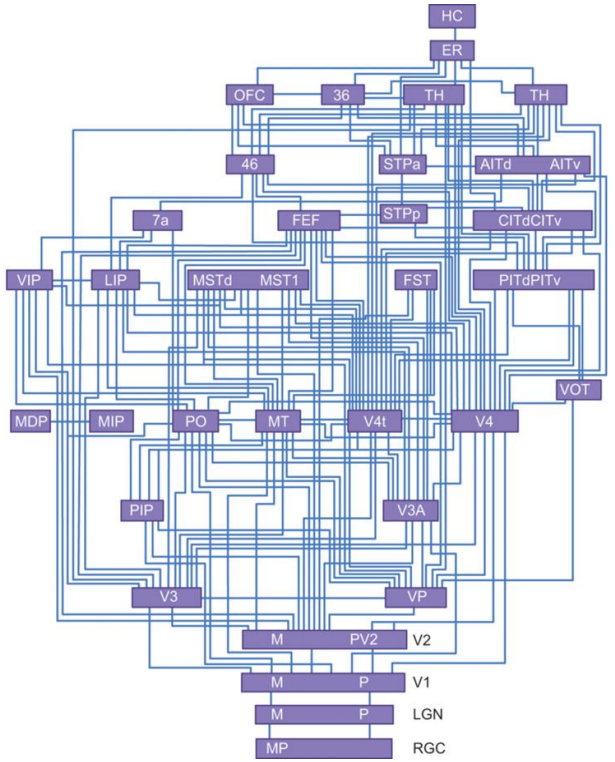
Des réseaux naturels … bouclés

[Harris et Mrsic-Flogel, 2013]



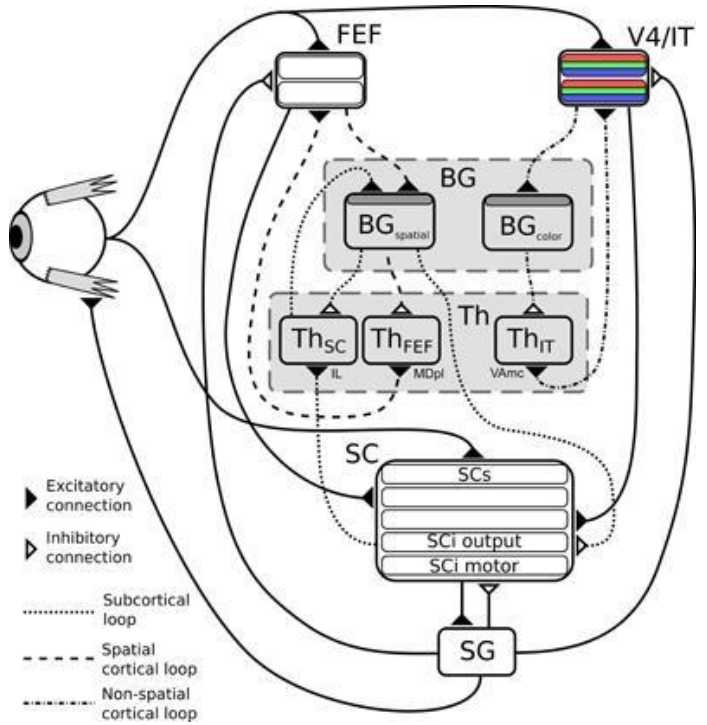
Couches neuronales

[Felleman & Van Essen,1991]



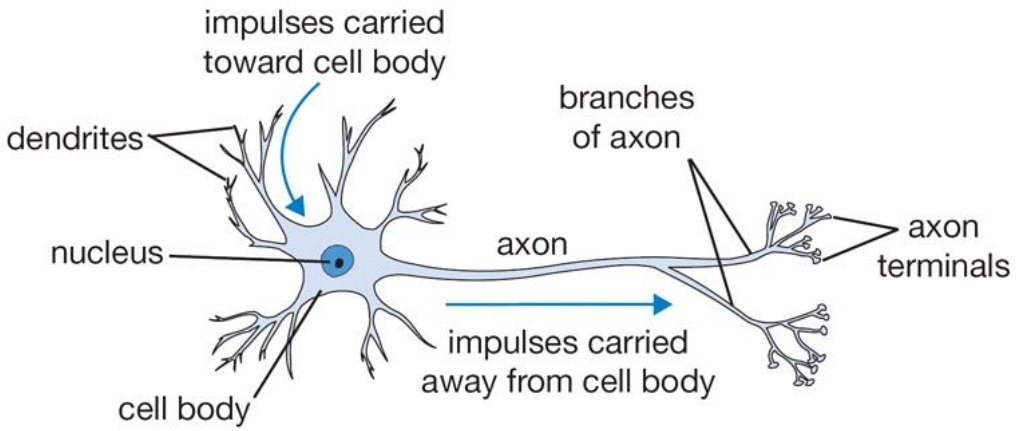
Zones corticales

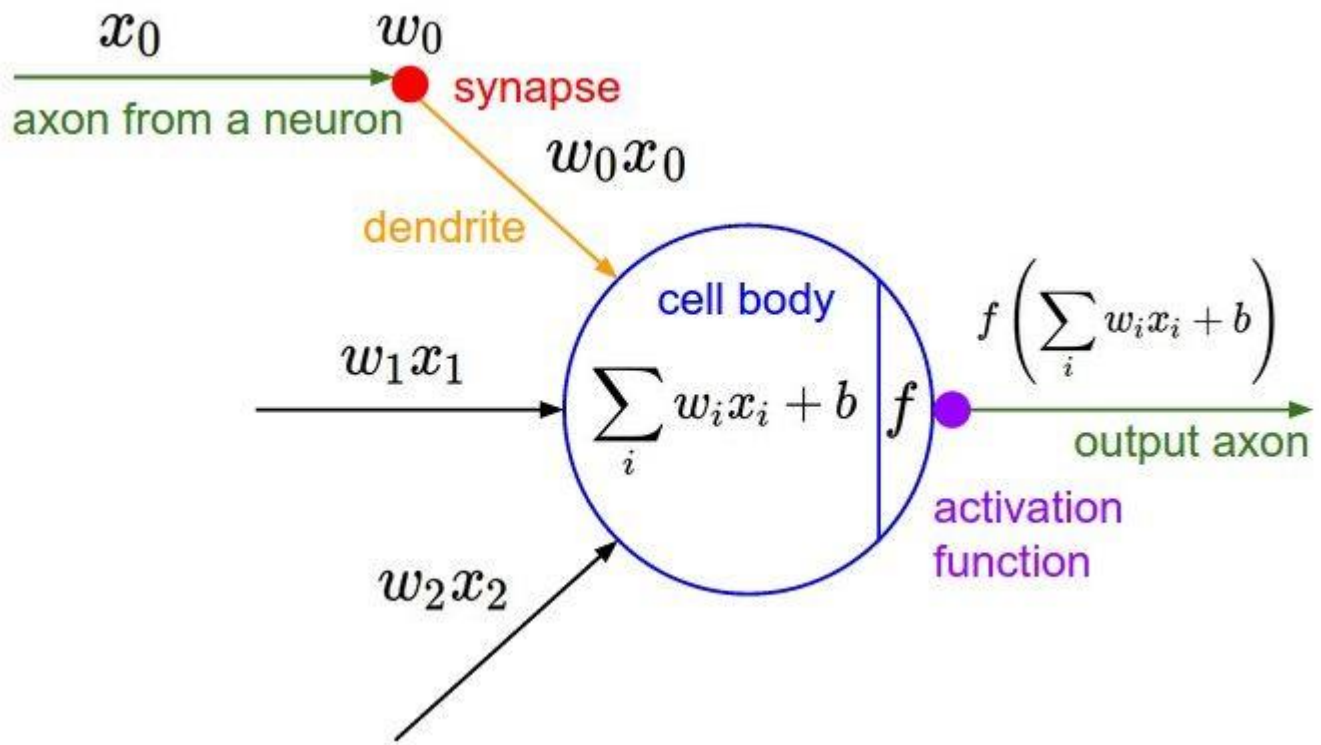
[N'Guyen, Thurat & Girard, 2014]



Structures neuronales

Le neurone artificiel





- Un modèle (très) simplifié (pas de prise en compte de le dynamique) [McCulloch & Pitts,1943]

Interprétation

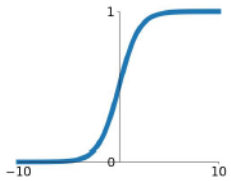
$$y = f\left(\sum_i w_i . x_i + b\right)$$

Un neurone = classifieur linéaire + activation non linéaire f.

Plusieurs fonctions d'activation possibles → rôles différents (proba, signe, sélection…)

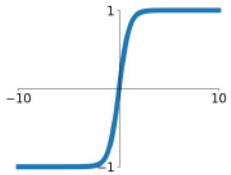
Sigmoid

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



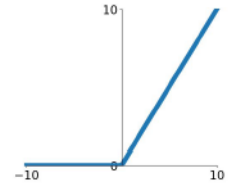
tanh

$\tanh(x)$



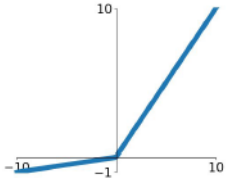
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

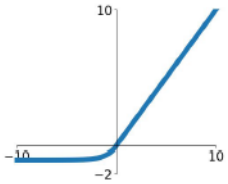


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Réseau = Plusieurs neurones

$$y_j = f\left(\sum_i w_{ji}.x_i + b_j\right)$$

Sous forme vectorielle:

$$\boldsymbol{X} \in \mathbb{R}^{\boldsymbol{d}} \mapsto \boldsymbol{Y} \in \mathbb{R}^{\boldsymbol{p}}$$

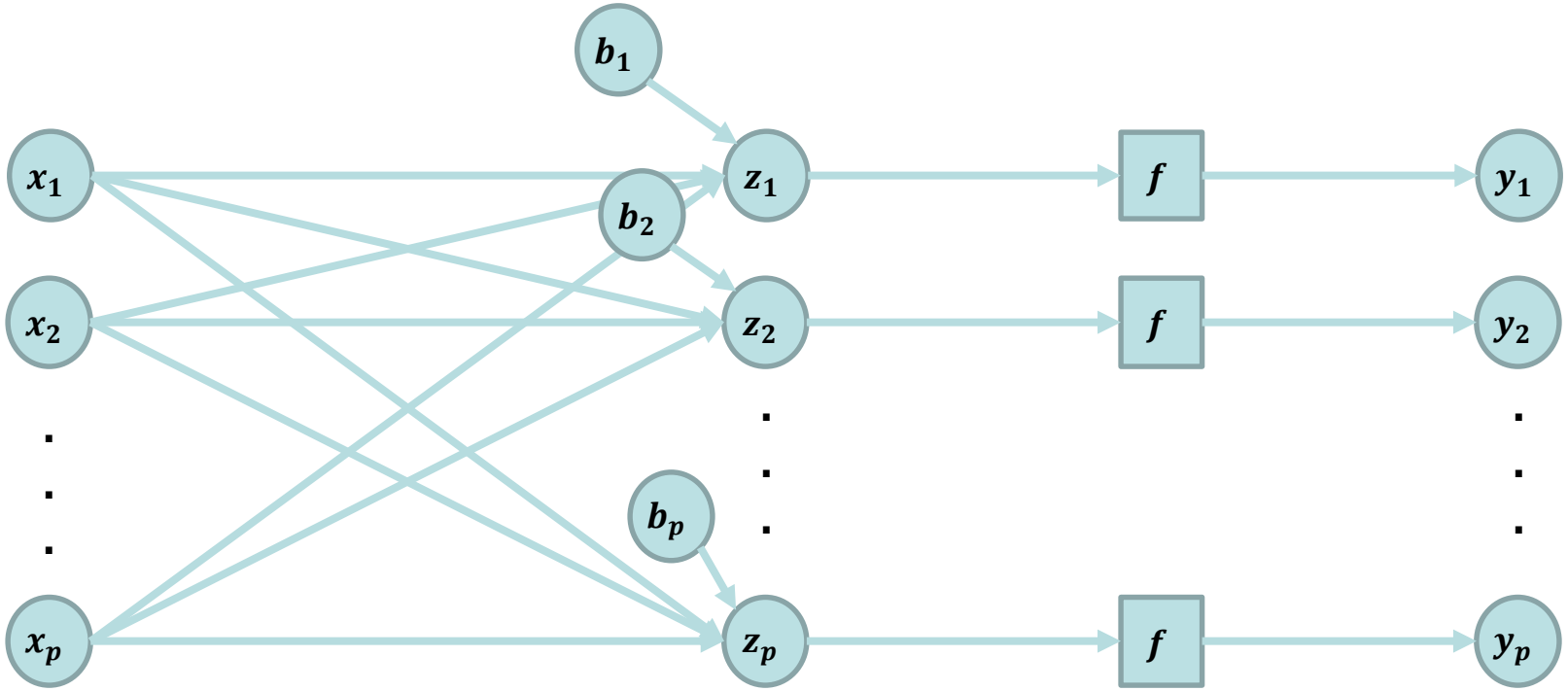
$$\boldsymbol{Y} = f(\boldsymbol{W}.\boldsymbol{X} + \boldsymbol{b})$$

$$\boldsymbol{W} \in \mathbb{R}^{\boldsymbol{p}\times\boldsymbol{d}}$$

$\boldsymbol{Y}$ peut représenter un signal ou une image → filtrage

Réseau « fully connected »

$$\boldsymbol{Y} = f(\boldsymbol{W}.\boldsymbol{X} + \boldsymbol{b})$$



Le perceptron

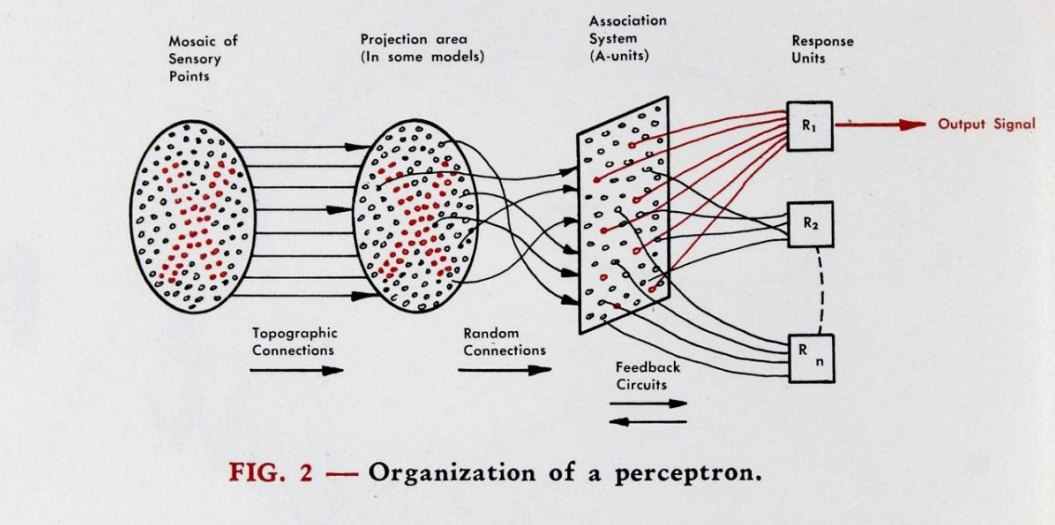
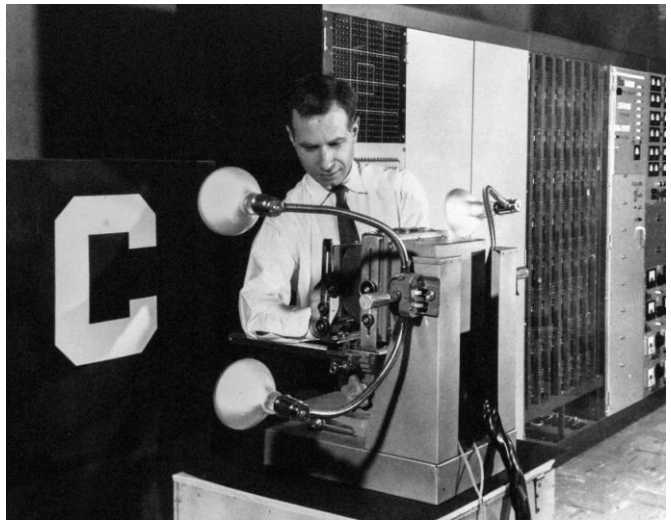


FIG. 2 — Organization of a perceptron.



Perceptron [Rosenblatt, 1958]

Algorithm Perceptron learning algorithm

Input:

A set of training examples $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$
Learning rate $0 < \alpha < 1$
Number of epochs $epochs$

1: Initialize the weight vector $\mathbf{w}$ with random values
2: Initialize the bias $b \leftarrow 0$
3: **for** $i \leftarrow 1$ to $epochs$ **do**
4: $err \leftarrow 0$ ▷ The number of misclassifications
5: **for each** training example $(\mathbf{x}_i, y_i) \in D$ **do**
6: $z_i \leftarrow \mathbf{w}^T \mathbf{x}_i + b$
7: $o_i \leftarrow \begin{cases} 1 & z_i \geq 0 \\ 0 & \text{otherwise} \end{cases}$
8: **if** $y_i \neq o_i$ **then**
9: $\mathbf{w} \leftarrow \mathbf{w} + \alpha(y_i - o_i)\mathbf{x}_i$
10: $b \leftarrow b + \alpha(y_i - o_i)$
11: $err \leftarrow err + 1$
12: **if** $err = 0$ **then break**

Algorithme du perceptron: tous les ingrédients d'apprentissage

```
class Perceptron:
    def __init__(self, learning_rate=0.1, epochs=100):
        self.learning_rate = learning_rate
        self.epochs = epochs

    def predict(self, inputs):
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        return self._activation(weighted_sum)

    def _activation(self, z):
        return 1 if z >= 0 else 0

    def fit(self, X, y):
        self.weights = np.random.rand(X.shape[1])
        self.bias = np.random.rand(1)

        has_error = True
        while has_error:
            has_error = False
            for inputs, target in zip(X, y):
                prediction = self.predict(inputs)
                error = target - prediction
                self.weights += self.learning_rate * error * inputs
                self.bias += self.learning_rate * error
                if error : has_error = True
```

Initialisation aléatoire

Itérations sur exemples

Critère d'erreur

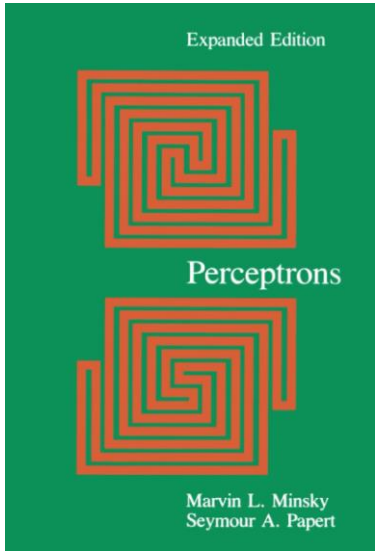
Mise-à-jour incrémentale des poids

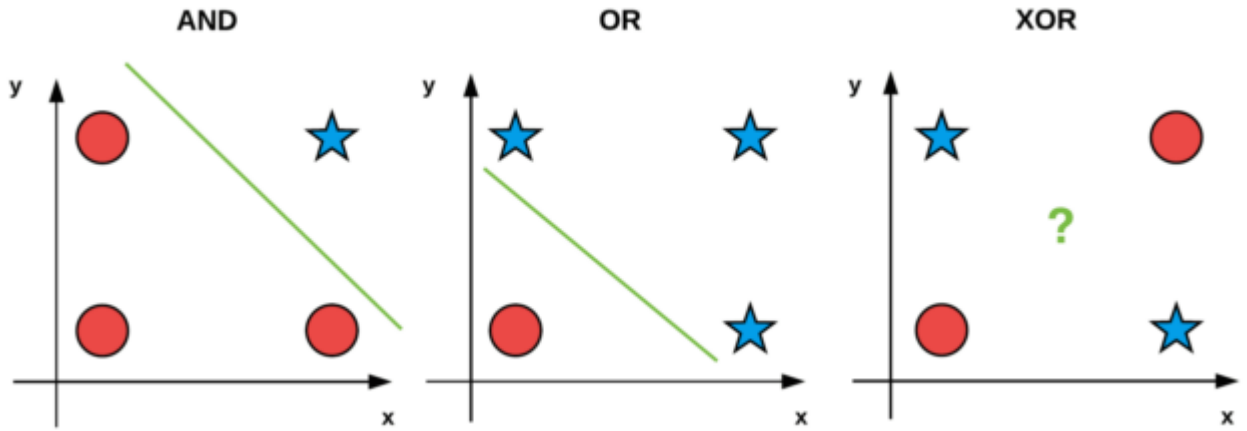
Critère d'arrêt

Les performances du perceptron

- On peut montrer que cet algorithme converge ssi les données sont linéairement séparables.
- Oscillations si non séparabilité ➔ grande limitation !

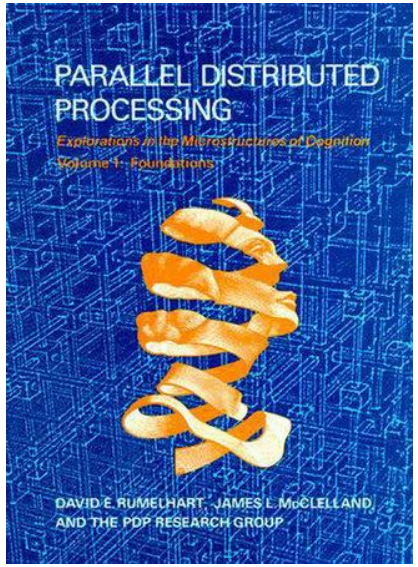
[Minsky & Papert, 1969]



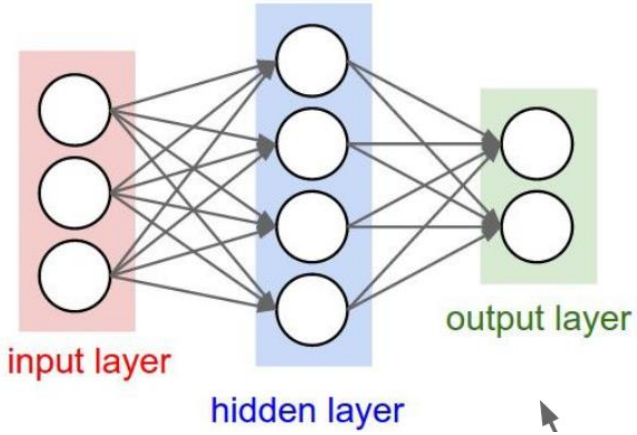


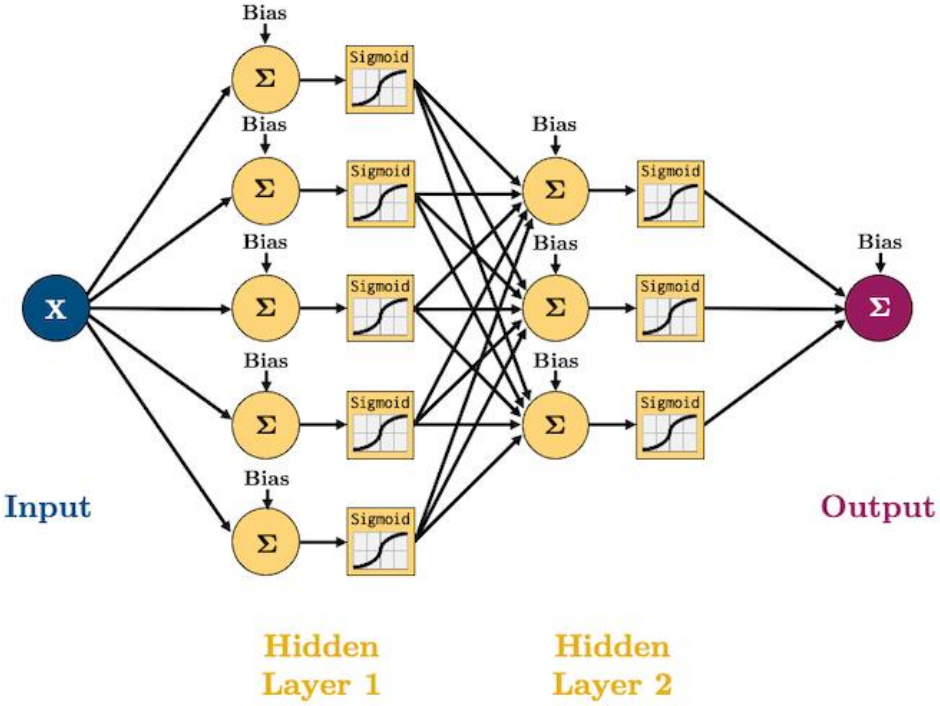
Les réseaux de neurones artificiels

- Composition de « couches » de neurones formels
  - ➔ Augmente l'expressivité des fonctions
- Optimisation par descente de gradient stochastique sur fonction de coût
  - ➔ Principe général d'apprentissage
- ANN = Artificial Neural Network
- MLP = Multi-Layer Perceptron



[Rumelhart & Mc Cleland, 1986]

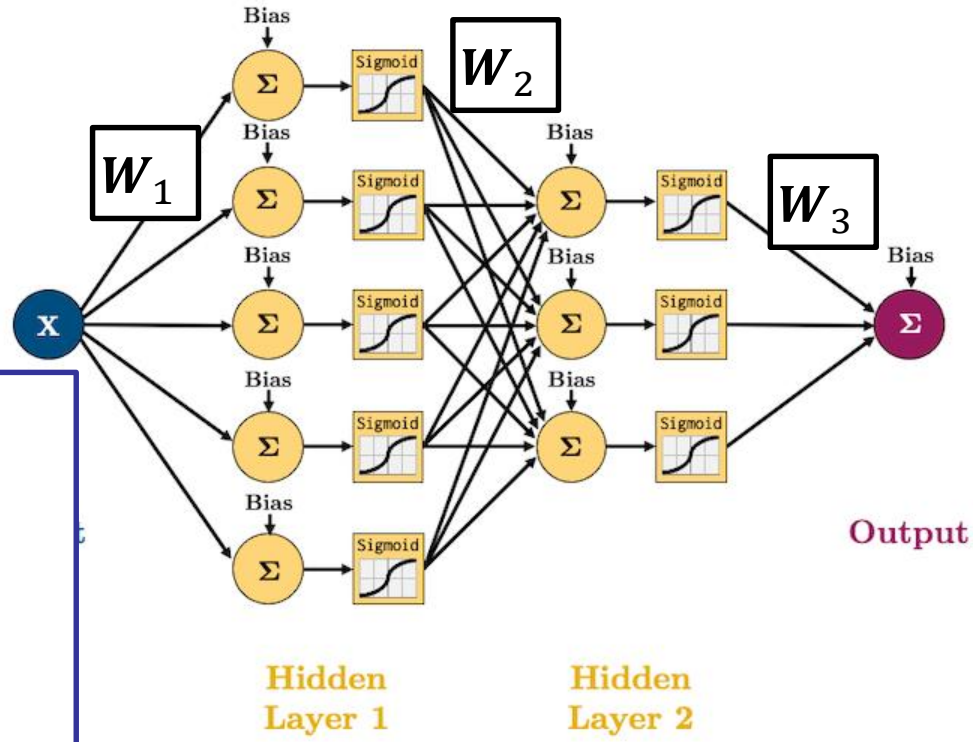




Réseaux multi-couches

Composition de fonctions vectorielles:

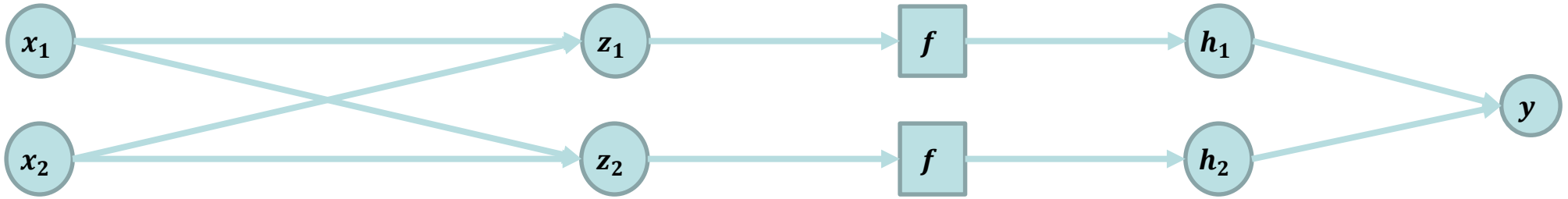
$$y = \boldsymbol{W}_3.f(\boldsymbol{W}_2.f(\boldsymbol{W_1}.x + \boldsymbol{b_1}) + \boldsymbol{b}_2) + b_3$$

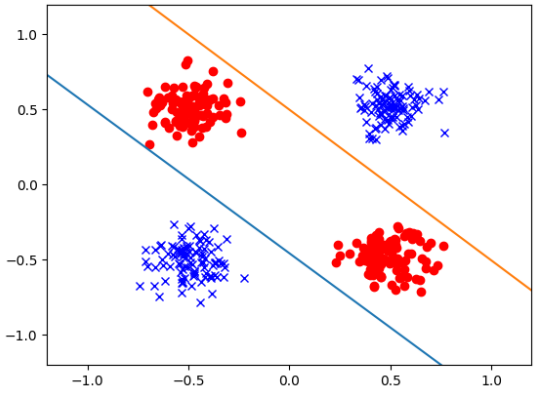


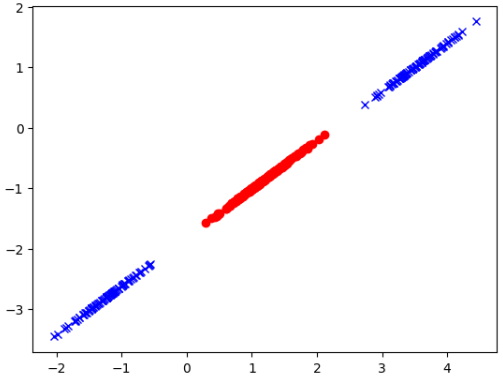

```
f = lambda x: 1.0/(1.0 + np.exp(-x))  # Activation sigmoide

x = np.random.randn(1,1)              # Entrée de dimension 1

W1 = np.random.randn(5,1)             # Première couche de dimension 1x5
b1 = np.random.randn(5,1)             # Biais de la première couche

W2 = np.random.randn(3,5)             # Deuxième couche de dimension 5x3
b2 = np.random.randn(3,1)             # Biais de la deuxième couche

W3 = np.random.randn(1,3)             # Troisième couche de dimension 3x1
b3 = np.random.randn(1,1) )           # Biais de la troisième couche

h1 = f(W1 @ x + b1)                   # Calcul de la première couche cachée
h2 = f(W2 @ h1 + b2)                  # Calcul de la deuxième couche cachée
y = W3 @ h2 + b3                      # Calcul de la sortie
```

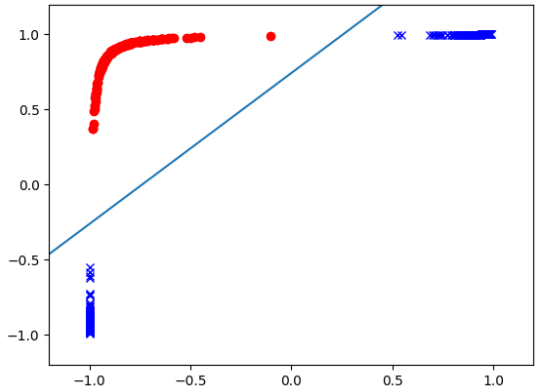
Expressivité des réseaux

- Pourquoi des fonctions d'activation non linéaires?
- Permet de transformer les espaces de représentation → meilleure séparation









Exemple des données binaires et RELU

- Base d'apprentissage $\{\boldsymbol{x}_i, y_i\}_N$ où $x_i \in \{-1,1\}^d$ et $y_i \in \{-1,1\}$.
- On peut écrire de manière exacte une fonction qui passe par tous les points de la base d'apprentissage

$$F(x, W) = \sum_i y_i . \mathrm{ReLU}(1 - \|\boldsymbol{x} - \boldsymbol{x}_i\|_1)$$

où $\mathrm{ReLU}(x) = \max(x, 0)$ est la fonction d'activation de type « Rectified Linear Unit » et $\|.\|_1$ est la norme L_1, $\|\boldsymbol{x}\|_1 = \sum_{k=1}^{d} |x_k|$.

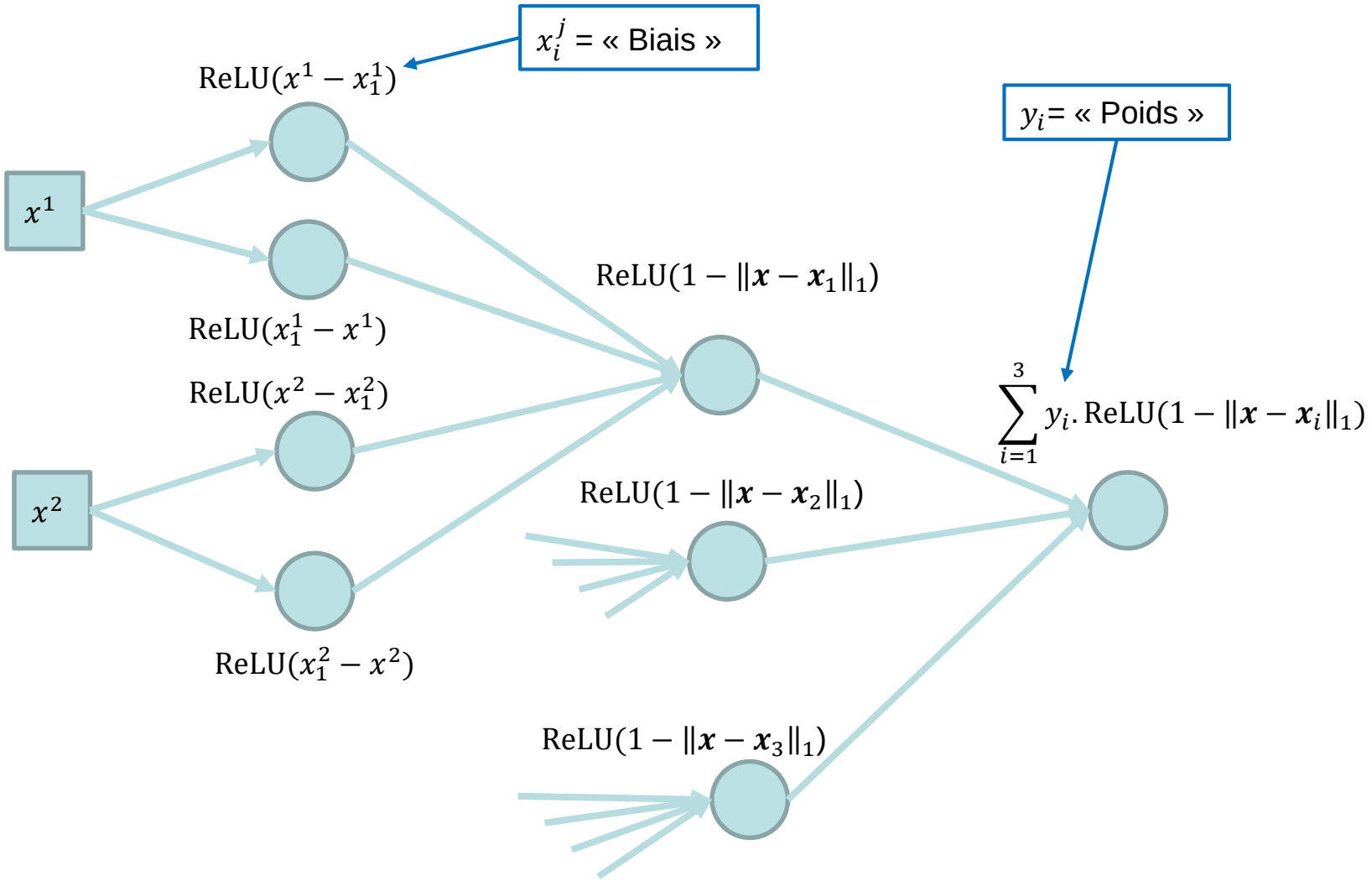
- « Astuce »: On peut écrire la valeur absolue à partir de la fonction ReLU:

$$|x| = \mathrm{ReLU}(\mathrm{x}) + \mathrm{ReLU}(-\mathrm{x})$$

= 3 neurones (2 couches: 2 neurones sur la première, 1 sur la deuxième)

- En combinant les neurones activés par un ReLU, on peut construire la fonction F comme un réseau à deux couches cachées: 2.N.d sur la première, N sur la deuxième, et 1 sur la dernière.

Exemple: réseau ReLU (d=2, N=3)



Approximation de fonction

- Un NN est un « régresseur » qui approxime une fonction multivariée: $\mathbb{R}^d \rightarrow \mathbb{R}^p$

- Classifieur RELU peut passer par tous les points
  ➔ mais sur-apprentissage + grand nombre de neurones

- Réseau = Approximateur universel (une couche cachée + activation non polynomiale suffit)
  ➔ conditions de régularité de la fonction permet de contrôler la complexité
  ➔ compromis taille/nombre de couches cachées

- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*(4), 303-314.
- Leshno, M., Lin, V. Ya., Pinkus, A., & Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, *6*(6), 861-867
- Barron, A. R. (1994). Approximation and estimation bounds for artificial neural networks. Machine Learning, 14(1), 115-133.
- Guliyev, N. J., & Ismailov, V. E. (2018). On the approximation by single hidden layer feedforward neural networks with fixed weights. *Neural Networks*, *98*, 296-304.
- Kidger, P., & Lyons, T. (2020). Universal Approximation with Deep Narrow Networks. *Proceedings of Thirty Third Conference on Learning Theory*, 2306-2327.
- Barron, A. R., & Klusowski, J. M. (2018). Approximation and Estimation for High-Dimensional Deep Learning Networks (arXiv:1809.03090).
- Nakada, R., & Imaizumi, M. (2020). Adaptive Approximation and Generalization of Deep Neural Network with Intrinsic Dimensionality. Journal of Machine Learning Research, 21(174), 1-38.
- E, W., Ma, C., & Wu, L. (2022). The Barron Space and the Flow-Induced Function Spaces for Neural Network Models. Constructive Approximation, 55(1), 369-406.
- Berner, J., Grohs, P., Kutyniok, G., & Petersen, P. (2022). The Modern Mathematics of Deep Learning (p. 1-111).

Comment concevoir les réseaux de neurones?

- Problème à résoudre
  - Classification, régression, détection, prédiction de graphe…
- Données
  - Nature, dimensions, quantité

➔ Architecture
  - Couches, connexions, poids…

- Apprentissage
  - Fonction de coût (« loss »)
  - Optimisation

Formulation optimale (bis !)

- On retrouve le principe usuel: optimisation d'un critère statistique fonction des données pour apprendre le prédicteur $F(\boldsymbol{x}; \boldsymbol{W})$:

$$\boldsymbol{W} = \arg \min_{\boldsymbol{W}'} C(\boldsymbol{\mathcal{L}}, \boldsymbol{W}')$$

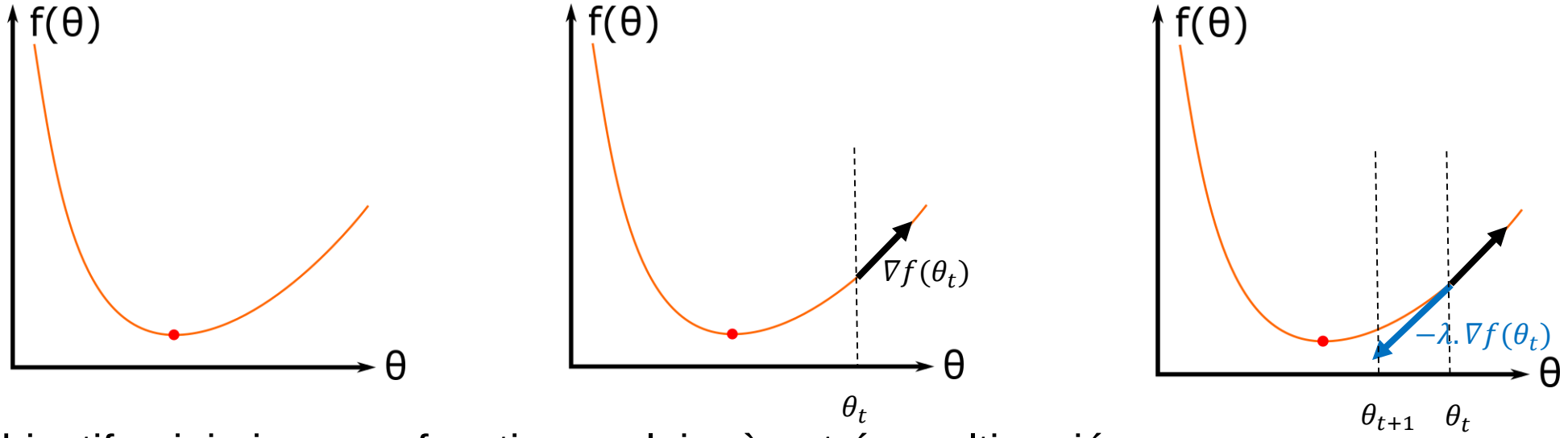
où $\boldsymbol{\mathcal{L}} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$ est la base des données d'apprentissage.

- Le critère est appelé coût (« loss » en anglais) et mesure une statistique sur les écarts de prédiction l:

$$C(\boldsymbol{\mathcal{L}}, \boldsymbol{W}) = \sum_{i=1}^{N} \ell(y_i, F(\boldsymbol{x}_i; \boldsymbol{W}))$$

- Deux questions (« classiques »):
  - Comment construire une « bonne » fonction de coût l?
  - Comment trouver les bons paramètres $\boldsymbol{W}$? → **Descente de gradient**

Descente de gradient



Objectif: minimiser une fonction scalaire à entrée multi-variée

Hypothèse: Fonction continue et différentiable presque partout

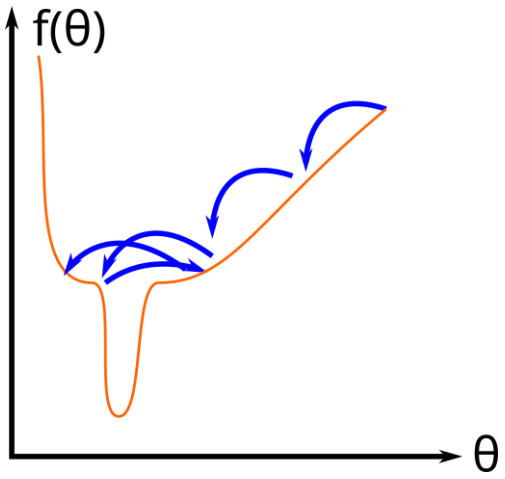
Principe: se déplacer itérativement dans le sens de la pente d'une certaine distance (on parle de « pas » du gradient)

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda.\nabla f_{\boldsymbol{\theta}}(\boldsymbol{\theta}_t)$$

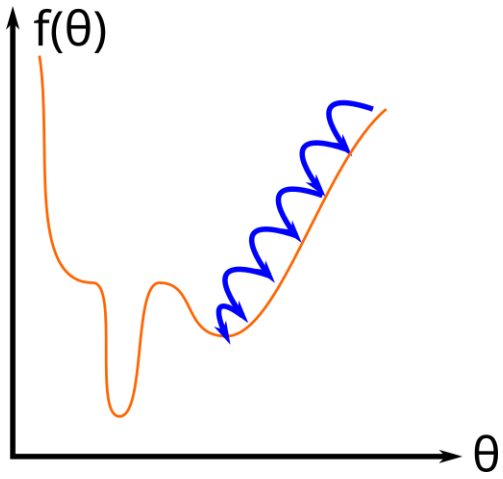
Plus la pente est raide, plus grand sera le déplacement

On s'arrête selon un certain critère, par ex. lorsque les incréments en θ sont inférieurs à un seuil.

Des phénomènes à éviter



Minima très creusés (vallée étroite % pas)



Minima locaux

Le réglage du pas du gradient (λ) est déterminant

➔ Il existe des réglages optimaux (si on connaît bien le problème)

➔ Il existe des stratégies de gestion du pas (« scheduler »)

➔ Il existe des réglages adaptatifs (ADAM…)

Démo

Descente de gradient

```
vector = start

    for _ in range(n_iter):
        diff = -learn_rate * gradient(vector)

        if np.all(np.abs(diff) <= tolerance):
            break

        vector += diff

    return vector
```

Gradient stochastique

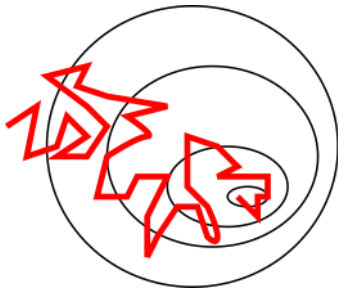
- On veut minimiser $C(\boldsymbol{\mathcal{L}}, \boldsymbol{W}) = \sum_{i=1}^{N} \ell(y_i, F(\boldsymbol{x}_i; \boldsymbol{W}))$ par descente de gradient
  - ➔ N peut être très grand
  - ➔ Chaque pas de gradient peut être très coûteux

- Une réponse: **échantillonner** aléatoirement un ensemble d'exemples (« **batch** ») à chaque étape pour calculer le gradient et mettre à jour l'estimation des paramètres

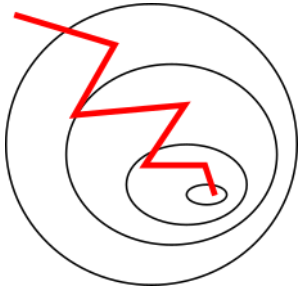
  = Descente de Gradient Stochastique (SGD)

$$\nabla_W C(\boldsymbol{\mathcal{L}}, \boldsymbol{W}) \approx \sum_{i \in batch} \nabla_W l(y_i, F(\boldsymbol{x}_i; \boldsymbol{W}))$$

|batch|=1



|batch|>>1



Descente de gradient stochastique

```
for _ in range(n_iter):
        # Shuffle x and y
        rng.shuffle(x,y)

        # Perform minibatch moves
        for istart in range(0, n_obs, batch_size):
            istop = istart + batch_size
            x_batch, y_batch = x[istart:istop, :], y[istart:istop]

            # Compute the gradient and increment
            grad = np.array(gradient(x_batch, y_batch, vector))
            diff = -learn_rate * grad

            # Check if the increment is small
            if np.all(np.abs(diff) <= tolerance):
                break

            # Update the values of the variables
            vector += diff

    return vector
```

Comment calculer « facilement » le gradient?

- Un principe élémentaire: Rétro-propagation
  - → Calcul exact du gradient par rapport aux paramètres du NN
  - → Parallélisable
  - → Mais peut être gourmand en ressources (mémoire)

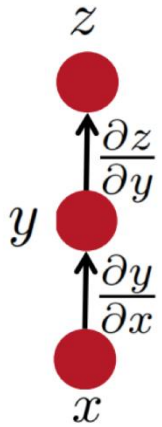
- Généralisable à tout type d'architecture fonctionnelle (pour des fonctions continues et dérivables pp)
  - → Dérivation automatique : dans les environnements (Pytorch, Tensorflow…)
  - → Autograd (https://github.com/HIPS/autograd)
  - → JAX (https://github.com/google/jax)

Un ingrédient « mathématique » de base

- Dérivée des fonctions composées (« chain rule »)
- Variable latente scalaire

$$y = g(x) \in \mathbb{R}, z = f(y) \in \mathbb{R}$$

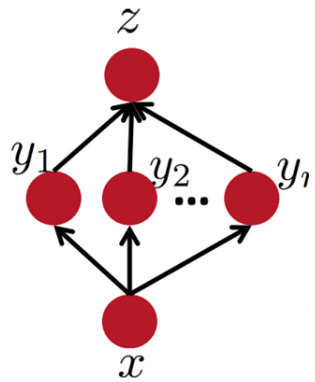
$$\frac{dz}{dx} = \frac{d}{dx}[f \circ g(x)] = f'\big(g(x)\big).g'(x) = \frac{dz}{dy}.\frac{dy}{dx}$$



- Variable latente vectorielle

$$\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{y} = g(\boldsymbol{x}) \in \mathbb{R}^k, z = f(\boldsymbol{y}) \in \mathbb{R}$$

$$\frac{dz}{dx_j} = \sum_i \frac{\partial z}{\partial y_i}.\frac{\partial y_i}{\partial x_j}$$



Ou avec le jacobien $\left(\frac{\partial y}{\partial x}\right) \equiv \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_k}{\partial x_1} & \cdots & \frac{\partial y_k}{\partial x_n} \end{bmatrix}$, $\nabla_x(z) = \left(\frac{\partial y}{\partial x}\right)^T . \nabla_y(z)$

Utiliser l'architecture en couches

- Réseau à m couches entièrement connecté:

$$\boldsymbol{x}_1 = f(\boldsymbol{W}_1.\boldsymbol{x}_0 + \boldsymbol{b}_1) = F_1(x_0, \boldsymbol{W}_1, \boldsymbol{b}_1)$$
$$\boldsymbol{x}_2 = f(\boldsymbol{W}_2.\boldsymbol{x}_1 + \boldsymbol{b}_2) = F_2(x_1, \boldsymbol{W}_2, \boldsymbol{b}_2)$$
$$\vdots$$
$$\boldsymbol{y} = \boldsymbol{x}_m = f(\boldsymbol{W}_m.\boldsymbol{x}_{m-1} + \boldsymbol{b}_m) = F_m(x_{m-1}, \boldsymbol{W}_m, \boldsymbol{b}_m)$$

- De fonction d'activation f, de poids $\boldsymbol{W}_j$ et de biais $\boldsymbol{b}_j$.
- La dernière couche fournit la prédiction $\boldsymbol{y}$ à comparer avec la sortie souhaitée $\boldsymbol{y}^*$
- On cherche à minimiser $\ell(\boldsymbol{y}, \boldsymbol{y}^*)$ par une descente de gradient sur les paramètres du réseau.
- On va appliquer une méthode de **rétro-propagation** pour calculer le gradient par rapport aux paramètres $\boldsymbol{W}_j$ et $\boldsymbol{b}_j$.

- Principe:
  - Les gradients de la couche $j-1$ dépendent des gradients de la couche j.
  - On a besoin de des gradients par rapport aux poids $\boldsymbol{W}_j$ et biais b_j, et par rapport aux sorties intermédiaires x_j.

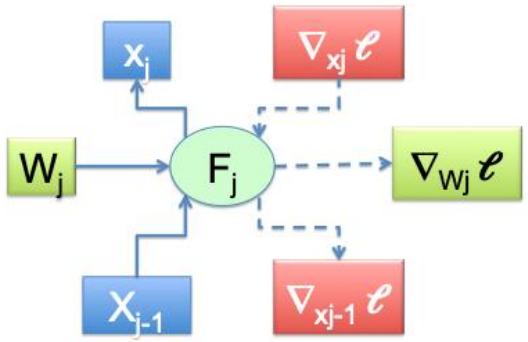
Dérivation des fonctions composées pour NN

- Pour calculer le gradient par rapport aux paramètres $\nabla_{\boldsymbol{W}_j}$ on a besoin des gradients par rapport aux sorties des neurones $\nabla_{\boldsymbol{x}_j}$.

$$\nabla_{\mathbf{W}_j}\ell = \left(\frac{\partial x_j}{\partial \mathbf{W}_j}\right)^T . \nabla_{x_j}\ell = \left(\frac{\partial F_j(., \mathbf{W}_j)}{\partial \mathbf{W}_j}\right)^T . \nabla_{x_j}\ell$$

- Mais pour avoir les gradients $\nabla_{\boldsymbol{x}_j}$ on a besoin des gradients de la couche suivante $\nabla_{\boldsymbol{x}_{j+1}}$ que l'on **rétro-propage**.

$$\nabla_{x_{j-1}}\ell = \left(\frac{\partial x_j}{\partial x_{j-1}}\right)^T . \nabla_{x_j}\ell = \left(\frac{\partial F_j(x_{j-1}, .)}{\partial x_{j-1}}\right)^T . \nabla_{x_j}\ell$$



- Le point de départ est le calcul du gradient de ℓ par rapport à la couche de sortie $\boldsymbol{y}$
- Il faut maintenant calculer les jacobiens $\left(\frac{\partial F_j}{\partial x_{j-1}}\right)$ et $\left(\frac{\partial F_j}{\partial W_j}\right)$

Des jacobiens simples

- Les fonctions F_j sont des composées d'une fonction linéaire et d'une activation non linéaire

$$F_j(\boldsymbol{x}_{j-1}, \boldsymbol{W}_j, \boldsymbol{b}_j) = f(\boldsymbol{W}_j.\boldsymbol{x}_{j-1} + \boldsymbol{b}_j) \in \mathbb{R}^p$$

- On a alors

$$\frac{\partial F_j}{\partial \boldsymbol{x}_{j-1}} = f'(\boldsymbol{W}_j.\boldsymbol{x}_{j-1} + \boldsymbol{b}_j).\boldsymbol{W}_j^T$$

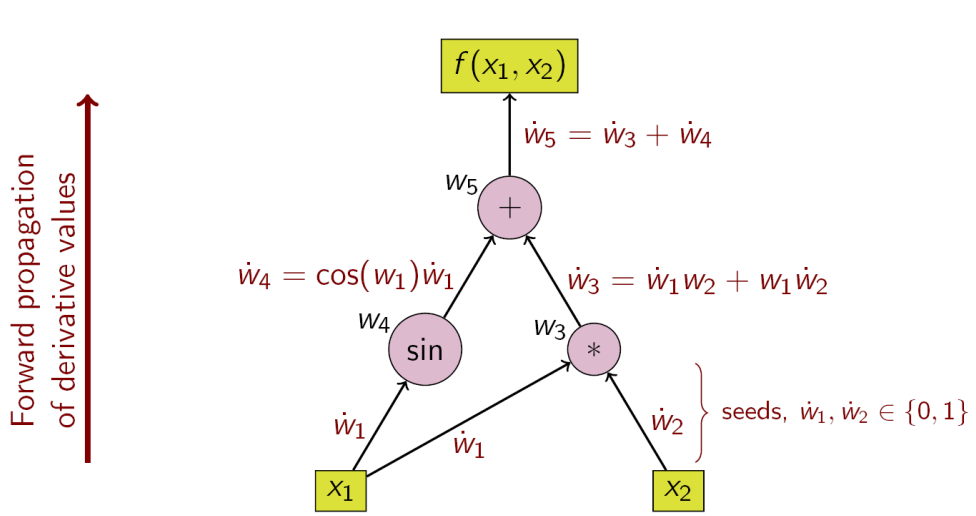
$$\frac{\partial F_j}{\partial \boldsymbol{W}_j} = \boldsymbol{x}_{j-1}^T.f'(\boldsymbol{W}_j.\boldsymbol{x}_{j-1} + \boldsymbol{b}_j)$$

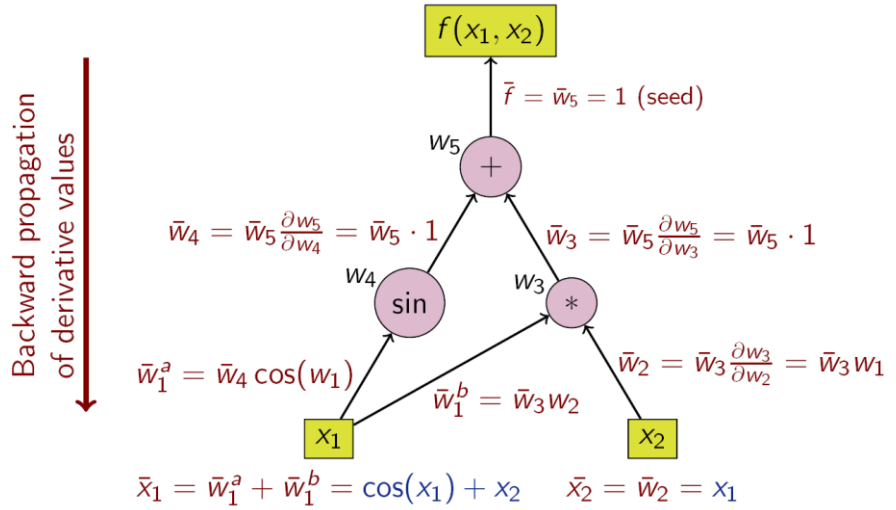
$$\frac{\partial F_j}{\partial \boldsymbol{b}_j} = f'(\boldsymbol{W}_j.\boldsymbol{x}_{j-1} + \boldsymbol{b}_j)$$

Avec $\boldsymbol{W}_j \in \mathbb{R}^{p\times d}$ (d neurones en entrée x p neurones en sortie), $\boldsymbol{x}_{j-1} \in \mathbb{R}^d$ et $\boldsymbol{b}_j \in \mathbb{R}^p$.

Remarques sur dérivation

- Principe général que l'on peut formuler par un graphe fonctionnel
- On a utilisé la formule de dérivation des fonctions composées en mode « rétrograde »
- On peut l'utiliser aussi en mode directe (propagation vers l'avant): c'est efficace si la sortie est de dimension supérieure à l'entrée
- En règle générale, utiliser une bibliothèque logicielle de dérivation automatique pour réaliser les calculs (ne les faites pas à la main!): elles sont présentes dans les environnements de programmation





$$y = \sin(x1) + x1 * x2$$

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

Les fonctions de coût

- Elles mesurent l'écart entre sortie souhaitée $\boldsymbol{y}^*$ et sortie réelle $\boldsymbol{y}$: $\ell(y, y^*)$

- Elles doivent être dérivables pour calculer un gradient

- Elles dépendent du problème = ce que code la dernière couche du réseau:
  - Régression: une valeur réelle $\boldsymbol{y} \in \mathbb{R}^p$
  - Détection (présence/absence): une probabilité $P[y = 1|\boldsymbol{x}]$
  - Classification: un vecteur de probabilités a posteriori $P[y = k|\boldsymbol{x}]$
  - Prédiction multi-label: un vecteur de probabilité pour chaque label $P[y_j = 1|\boldsymbol{x}]$
  - Graphe: une matrice de probabilités de présence d'un arc
  - Détection (localisation): Intersection Over Union de boites englobantes
  - …

Fonctions de coût classiques

- **MSE (Mean Squared Error)** pour régression $\boldsymbol{y}^* \in \mathbb{R}^p$

$$\ell_{MSE}(\boldsymbol{y}, \boldsymbol{y}^*) = \frac{1}{p}\sum_{i=1}^{p}(y_i - y_i^*)^2$$

- **BCE (Binary Cross Entropy)** pour classification binaire $y^* \in \{0,1\}$

$$\ell_{BCE}(y, y^*) = -y^* \log y - (1 - y^*)\log(1 - y)$$

Rem: la sortie y code une probabilité ➔elle doit être dans $[0,1]$. On utilise en général une sigmoïde comme fonction d'activation de la dernière couche pour normaliser la sortie.

Fonctions de coût classiques

- **Cross Entropy** pour multi-classe $y^* \in \{0,1\}^p$ où $y_i^* = \delta_{i=c}$ si la classe cible est c.

$$\ell_{CE}(\boldsymbol{y}, \boldsymbol{y}^*) = -\sum_{i=1}^{p} y_i^* . \log(y_i)$$

Rem: la sortie $\boldsymbol{y}$ code une probabilité a posteriori ➔elle doit être normalisée à 1 ($\sum_i y_i = 1$). On utilise en général une fonction **`softmax:`** $\mathbb{R}^p \rightarrow [0,1]^p$ pour la dernière couche.

$$y_i = \text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

$$\ell_{CE}(\boldsymbol{y}, \boldsymbol{y}^*) = -x_c + \log \sum_j \exp(x_j)$$

On a alors une dérivée qui se calcule facilement:

$$\frac{\partial \ell_{CE}}{\partial x_i} = -\delta_{i=c} + \text{softmax(x)}_\text{i}$$

L’apprentissage version ANN

L’algorithme d’apprentissage standard boucle sur les étapes suivantes:

1. Choix des exemples: « batch »
2. Calcul des activations du réseau: étape « Forward »
3. Calcul de la fonction de coût
4. Calcul des gradients de la fonction de coût par rétro-propagation: étape « Backward »
5. Mise à jour des poids: descente de gradient
6. Mise à jour du pas du gradient (« scheduler »)

Les batchs sont tirés aléatoirement, en général sans remise.

Une « epoch » est un parcours complet du jeu de données.

On analyse le bon/mauvais comportement de l’apprentissage en visualisant les courbes d’évolution du coût et des performances sur un ensemble de validation

Code élémentaire (full numpy)

```python
# Activation sigmoide
f = lambda x: 1.0/(1.0 + np.exp(-x))
# Dérivée activation sigmoide
df = lambda x: np.exp(-x)/(1.0 + np.exp(-x))**2

# réseau à 1 couche cachée et sortie scalaire
class Classifier():
  def __init__(self, nhidden = 3):

    self.w1 = 2*np.random.random((2, nhidden)) - 1
    self.w2 = 2*np.random.random((nhidden, 1)) - 1

    self.b1 = 2*np.random.random((nhidden, 1)) - 1
    self.b2 = 2*np.random.random((1, 1)) - 1

  # Prédiction
  def forward(self,X):
    self.x = X
    self.z1 = self.x @ self.w1 + self.b1.T
    self.h1 = f(self.z1)
    self.z2 = self.h1 @ self.w2 + self.b2.T
    y = f(self.z2)
    return y
```

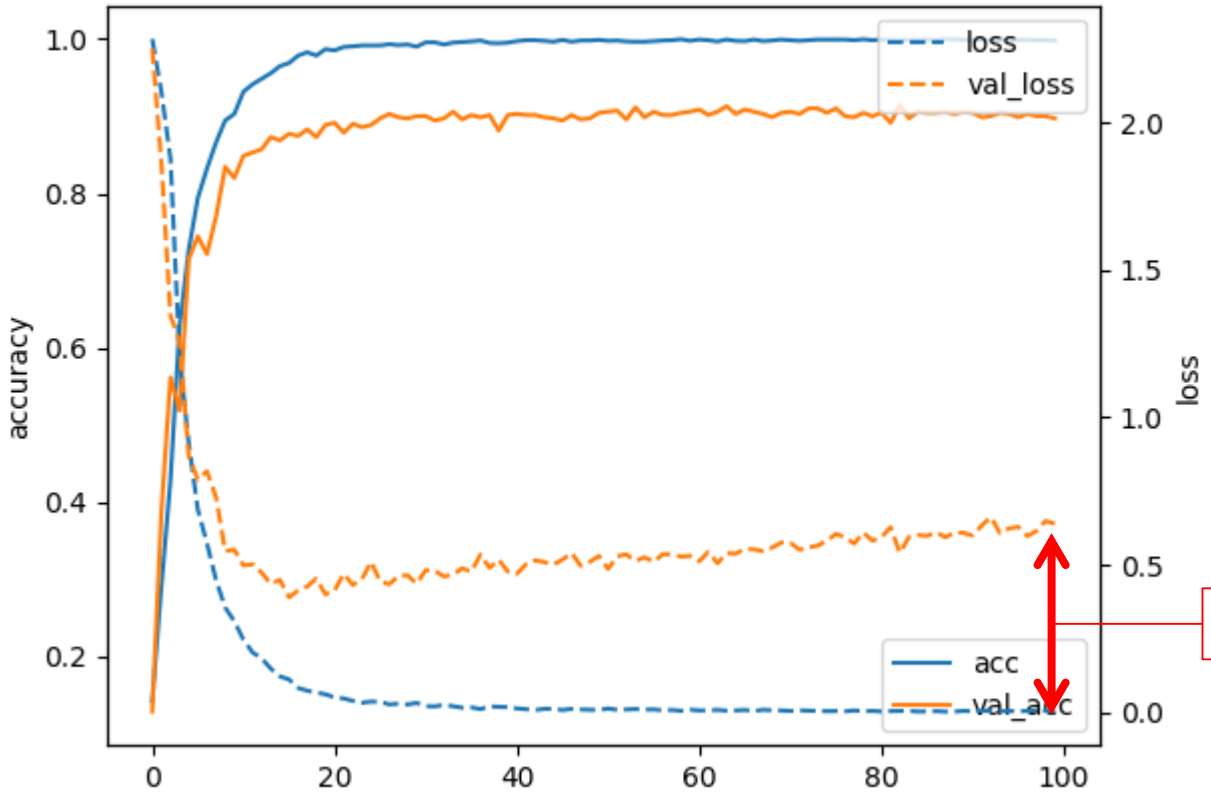
```python
  # Calcul du gradient par rétropropagation
  def backward(self, y_pred, y_true, lr = 1e-3):

    # Calcul du gradient de la fonction de coût (BCE)
    grad_y_pred = (1-y_true)/(1-y_pred) - y_true/y_pred

    # Rétropropagation couche 2
    dz2 = df(self.z2) * grad_y_pred
    grad_w2 = self.h1.T @ dz2
    grad_b2 = dz2.T @ np.ones((len(y_pred),1))

    # Rétropropagation couche 1
    grad_h1 = self.w2.T * dz2
    dz1 = df(self.z1) * grad_h1
    grad_w1 = self.x.T @ dz1
    grad_b1 = dz1.T @ np.ones((len(self.h1),1))

    # Mise à jour des paramètres (descente de gradient)
    self.w1 -= lr * grad_w1
    self.w2 -= lr * grad_w2
    self.b1 -= lr * grad_b1
    self.b2 -= lr * grad_b2
```

Code élémentaire (full numpy)

```
## Définition du classifieur
model = Classifier(nhidden = 2)

## Apprentissage

# Boucle sur les batch
for i in range(1000):

  # Constitution aléatoire du batch (ici avec remise)
  X_batch, X_valid, y_batch, y_valid = train_test_split(X_train, y_train, test_size=0.80)

  # Phase de calcul direct (forward)
  y_pred = model.forward(X_batch)

  # Rétropropagation du gradient avec pas du gradient lr
  model.backward(y_pred, y_batch, lr = 1e-4)

## Evaluation
y_pred_test = model.forward(X_test)
accuracy = ((y_pred_test > 0.5).astype(int) == y_test).sum()/len(y_test)
```

On verra comment implémenter le même modèle à partir de Pytorch

Loss et accuracy



Deux critères

- La fonction de perte (« loss »)
  - Elle sert à optimiser
  - On cherche à la minimiser
- La précision (« accuracy »)
  - Elle mesure la capacité de généralisation du prédicteur
  - On cherche à la maximiser
- Les deux sont reliés
  - La perte est souvent une « convexification » de l'adéquation

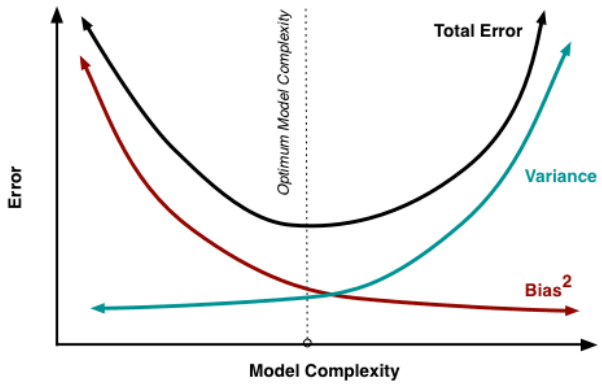
Erreur de généralisation

Remarques sur l’apprentissage

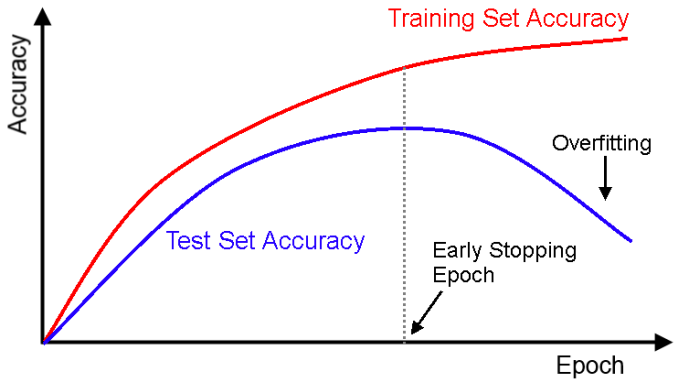
- **Aucune garantie de convergence**: les réseaux neuronaux forment des fonctions non convexes avec de multiples minima locaux.
- De nombreuses **époques** (dizaines de milliers) peuvent être nécessaires.
- Critères de fin : Nombre d'époques ; seuil d'erreur sur l'ensemble d'apprentissage ; pas de diminution de l'erreur ; augmentation de l'erreur sur un ensemble de validation.
- Pour éviter les **minima locaux** : faire plusieurs essais avec différents poids initiaux aléatoires + techniques de majorité ou de vote (approches ensemblistes).
- Deep Learning (prochain cours): réseaux de grande taille et jeu de données massifs pouvant nécessiter de nombreuses heures de calcul (des mois pour certains réseaux!).

Courbes d'apprentissage

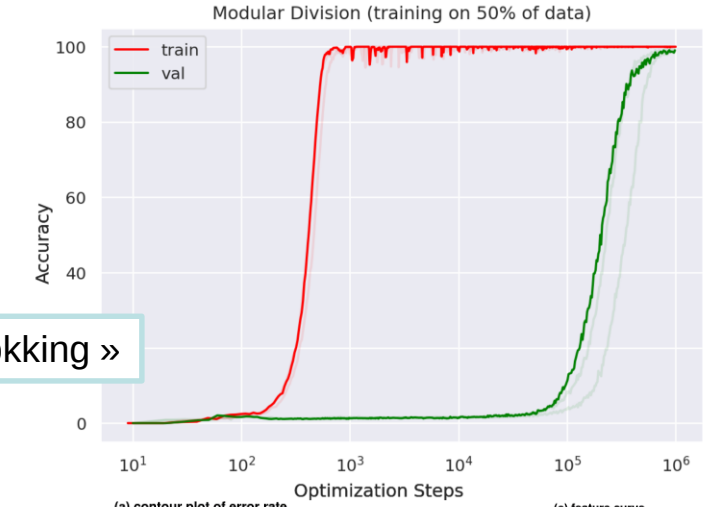
- Généralisation dépend:
  - Nombre de paramètres du modèle (complexité)
  - Nombre de données d'apprentissage
  - Nombre d'epoch
  - Pas du gradient
  - …
- Certaines formes permettent de détecter un mauvais comportement
- Mais Le comportement des réseaux est parfois contre-intuitif



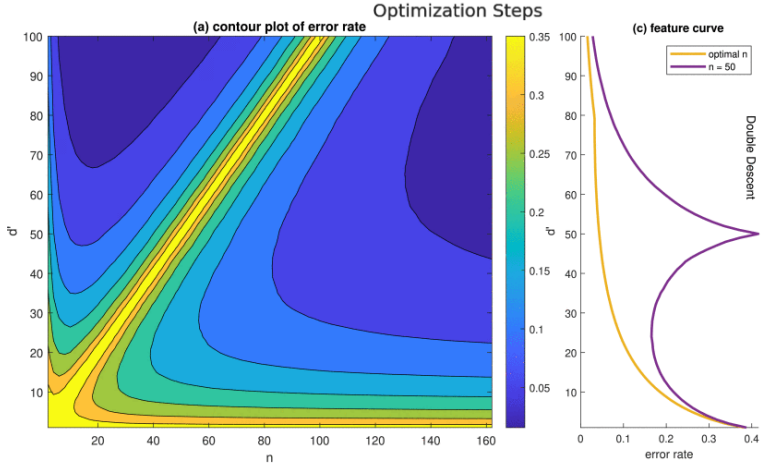
ML Standard



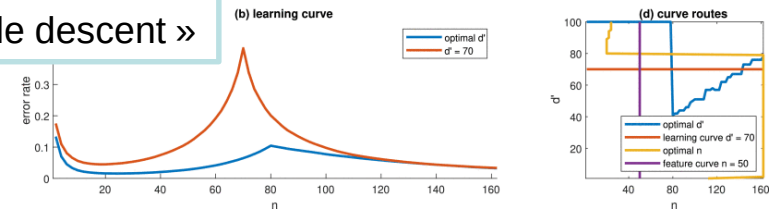
Descente de gradient



« Grokking »



« Double descent »



ONERA
THE FRENCH AEROSPACE LAB

Réseaux de neurones et ML statistique

- Les réseaux de neurones ne résolvent pas tous les problèmes: on a toujours à régler la question de la généralisation!
- Mais on a un nouveau moyen d'action: l'optimisation.

- La régularisation est toujours possible (par ex. en ajoutant une pénalisation L_2 sur les poids)
- Des techniques spécifiques:
  - « Drop-out »
  - Pas adaptatif du gradient stochastique
  - « Batch-norm »

➔Prochain cours

- Couplage complexe entre optimisation et généralisation: peu de résultats théoriques.

Résumé

- Les réseaux de neurones artificiels (ANN) sont des approximateurs paramétriques universels
- On formule le problème d'apprentissage de manière classique: minimisation d'une fonction de coût dépendant des données
- L'architecture des ANN (= espace fonctionnel) est déterminante
- Un algorithme généraliste: la descente de gradient stochastique
- Le gradient peut être calculé de manière automatique par rétro-propagation
- Formulation vectorielle ➔ calcul rapide sur GPU
- Il y a plusieurs hyper-paramètres à régler (batch, pas du gradient, architecture, fonction de coût…)
- Des environnements de programmation adaptés aux ANN