Introduction au Deep LEarning
Cours 1
Du perceptron aux réseaux de neurones

Olivier Schwander
olivier.schwander@sorbonne-universite.fr

Master MIND - Sorbonne Université

2025-2026

Slides de N. Baskiotis et V. Guigue

Retour sur le modèle linéaire

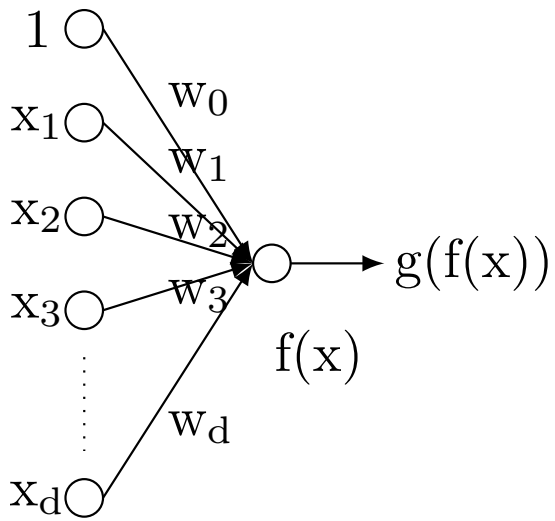
Problématique de l'apprentissage supervisé

- Ensemble d'apprentissage $\{(x^i, y^i)\} \in X \times Y$, ensemble de fonctions $\mathcal{F}$
- un coût $L(\hat{y}, y) : Y \times Y \to \mathbb{R}^+$, trouver $f^* = \text{argmin}_{f \in F} \sum_i L(f(x^i), y^i)$

Perceptron

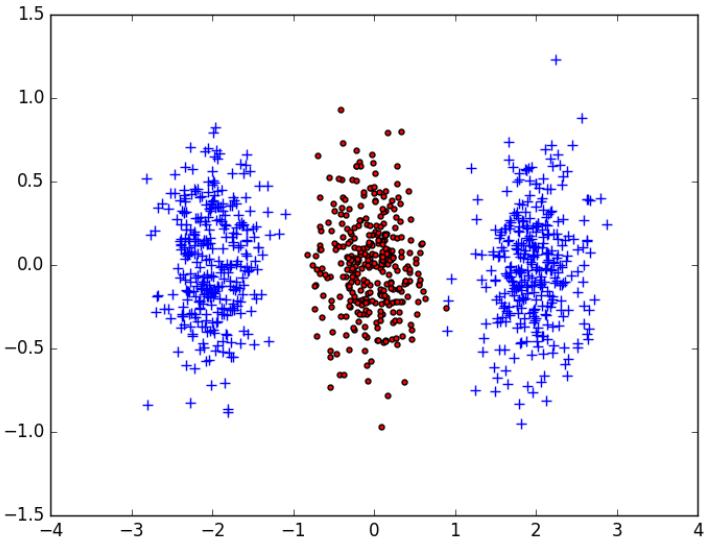
- Hypothèse linéaire : $f_w(x) = w_0 + \sum_{i=1}^{d} w_i x_i$
- Coût perceptron : $L(f_w(x), y) = \max(0, -f_w(x)y)$
- Gradient :

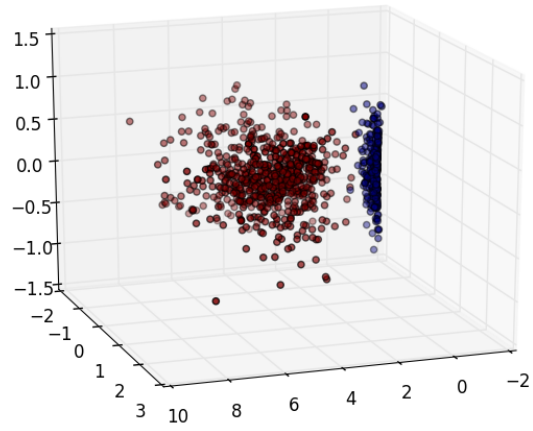
$$\nabla_w L(f_w(x), y) = \begin{cases} 0 & \text{si } (-y < w.x >) < 0 \\ -yx & \text{sinon} \end{cases}$$

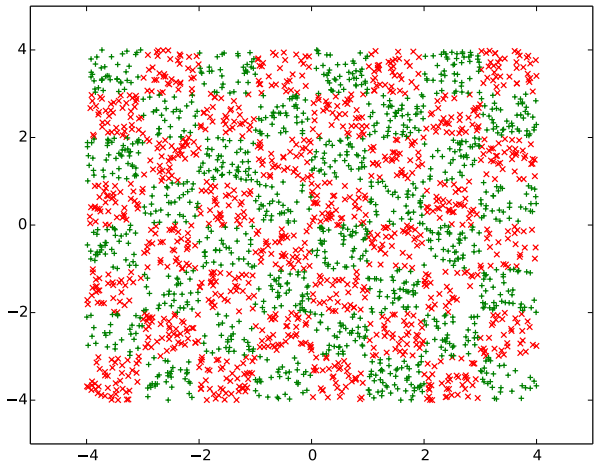


Limites du perceptron

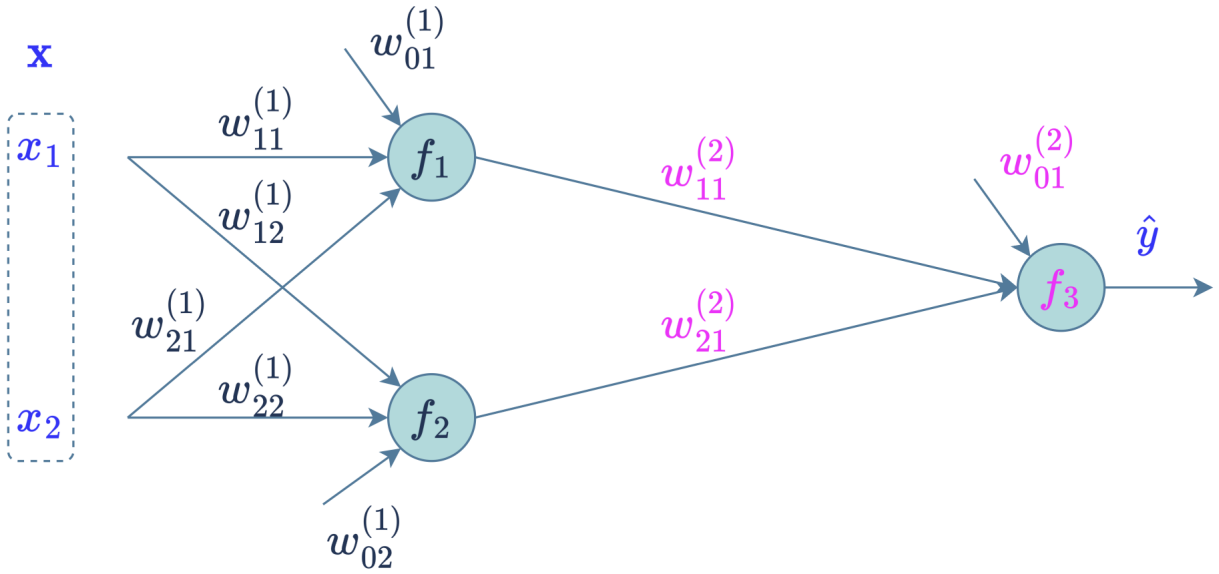
Est-il capable de séparer ces données ?

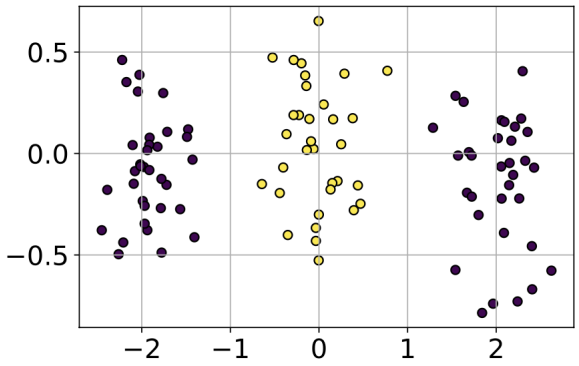






Combinons des neurones



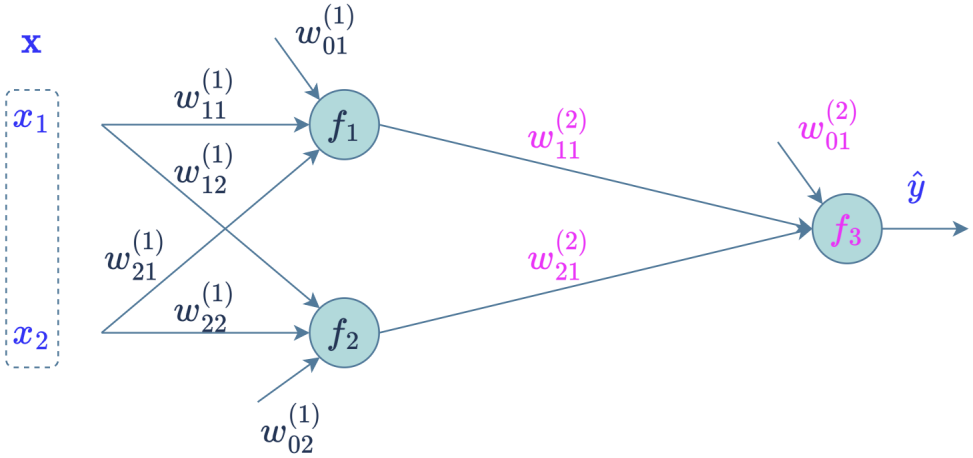


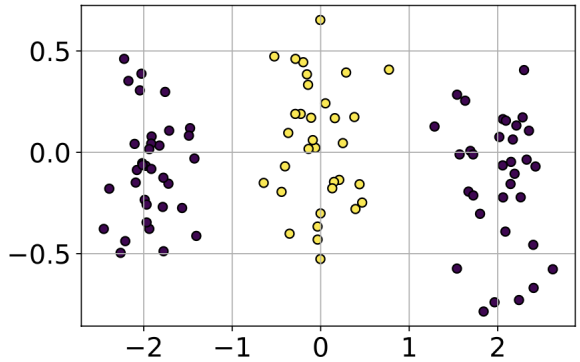
$$f_1(\mathbf{x}) = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{01}^{(1)}, \qquad f_2(\mathbf{x}) = w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{02}^{(1)}$$

$$f_3(\mathbf{x}) = w_{11}^{(2)} f_1(\mathbf{x}) + w_{21}^{(2)} f_2(\mathbf{x}) + w_{01}^{(2)}$$

- Combiner des neurones $\Rightarrow$ suffisant ?

Combinons des neurones





$$f_1(\mathbf{x}) = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{01}^{(1)}, \qquad f_2(\mathbf{x}) = w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{02}^{(1)}$$

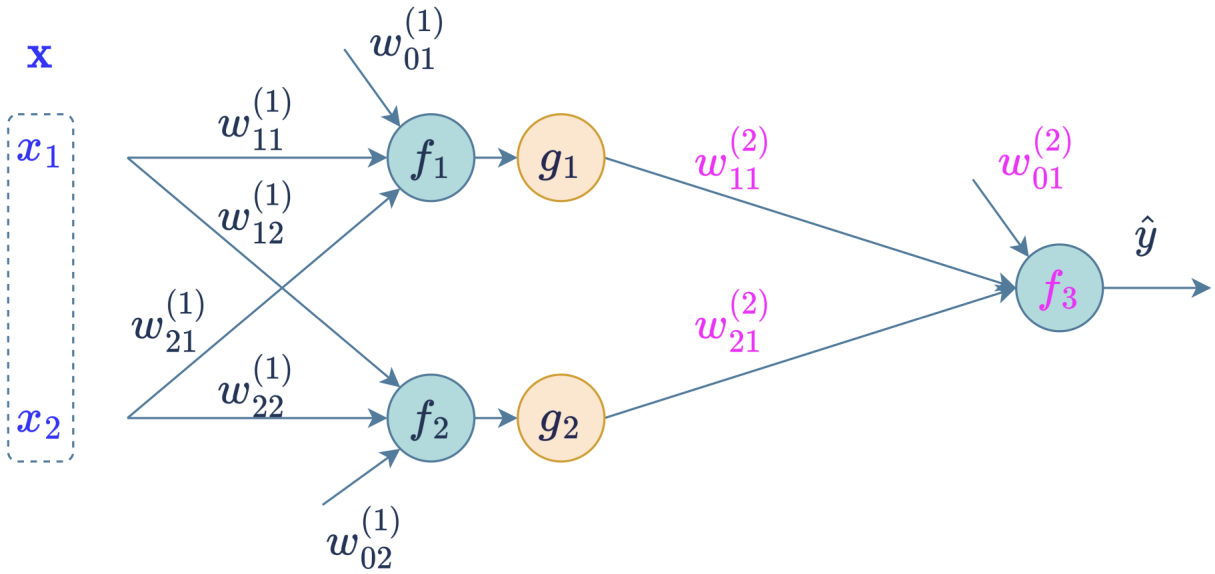
$$f_3(\mathbf{x}) = w_{11}^{(2)} f_1(\mathbf{x}) + w_{21}^{(2)} f_2(\mathbf{x}) + w_{01}^{(2)}$$

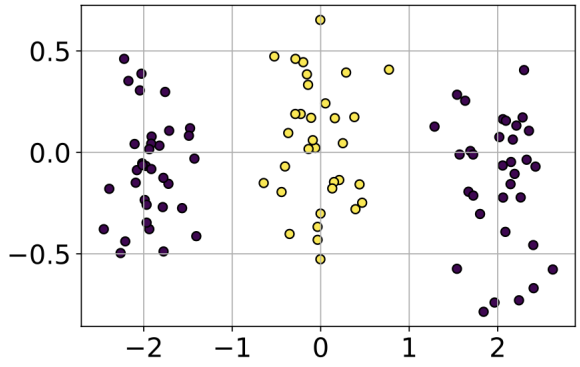
$$f_3(\mathbf{x}) = w_{11}^{(2)}(w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{01}^{(1)}) + w_{21}^{(2)}(w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{02}^{(1)}) + w_{01}^{(2)}$$

$$\Leftrightarrow f_3(\mathbf{x}) = x_1(w_{11}^{(2)} w_{11}^{(1)} + w_{21}^{(2)} w_{12}^{(1)}) + x_2(w_{11}^{(2)} w_{21}^{(1)} + w_{21}^{(2)} w_{22}^{(1)}) + w_{01}^{(2)} + w_{11}^{(2)} w_{01}^{(1)} + w_{21}^{(2)} w_{02}^{(1)}$$

- Combiner des neurones $\Rightarrow$ suffisant ?
  Non ! il faut introduire de la non linéarité, sinon équivalent à un perceptron ...

Un pas vers les réseaux profonds

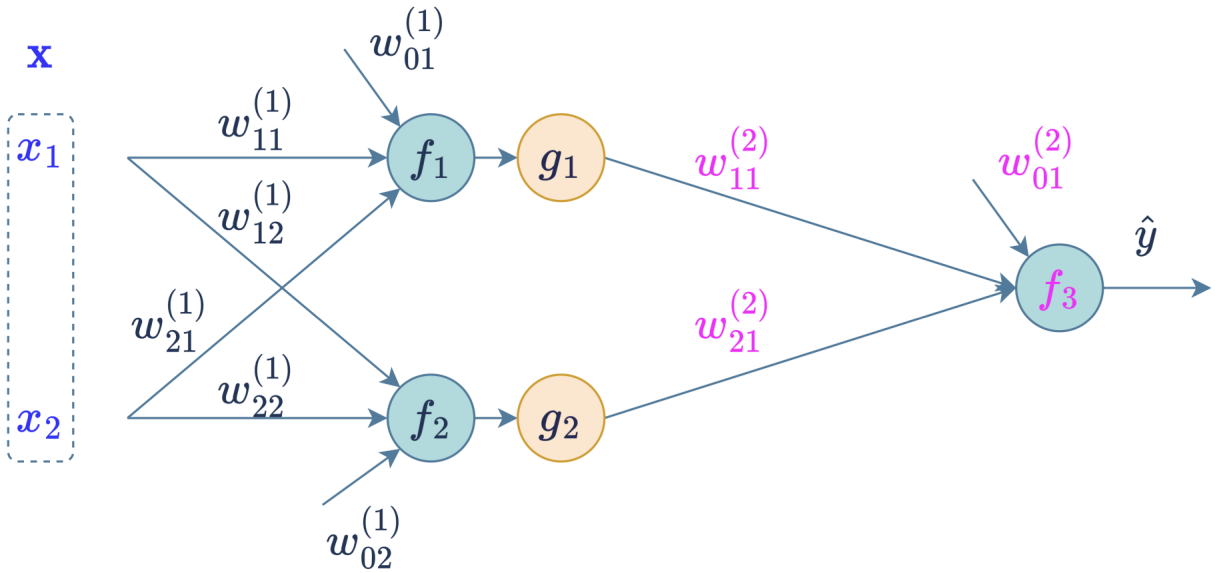


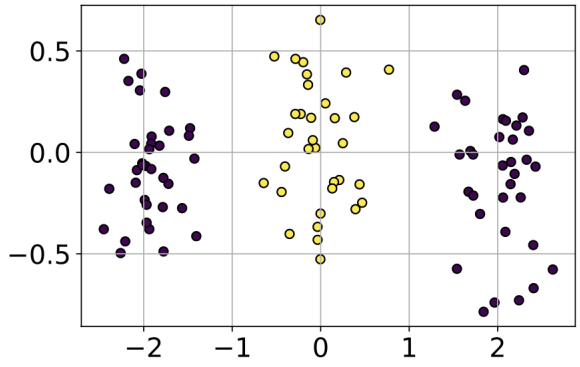


- Quelle non-linéarité ?
  - Fonction signe ?

  $\Rightarrow$ dérivée problématique ...
  - Fonctions tanh, sigmoïde, ...

Un pas vers les réseaux profonds

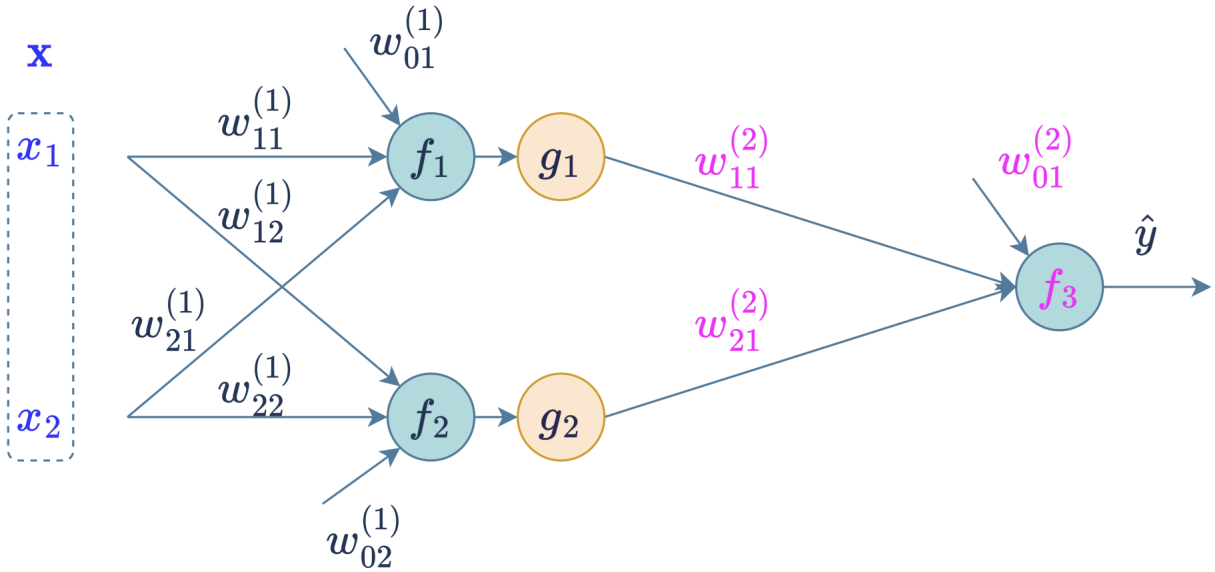


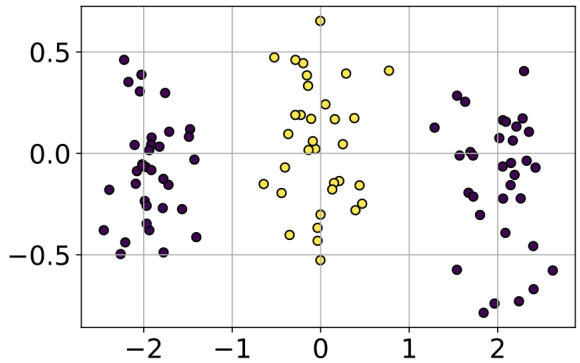


- Quelle non-linéarité ?
  - Fonction signe ?

  $\Rightarrow$ dérivée problématique ...
  - Fonctions tanh, sigmoïde, ...

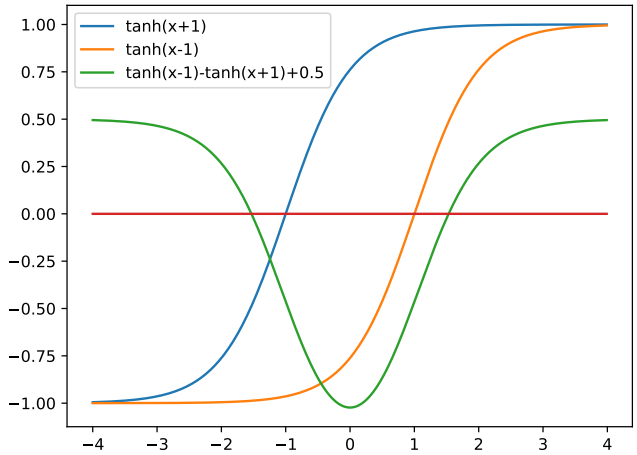
Un pas vers les réseaux profonds





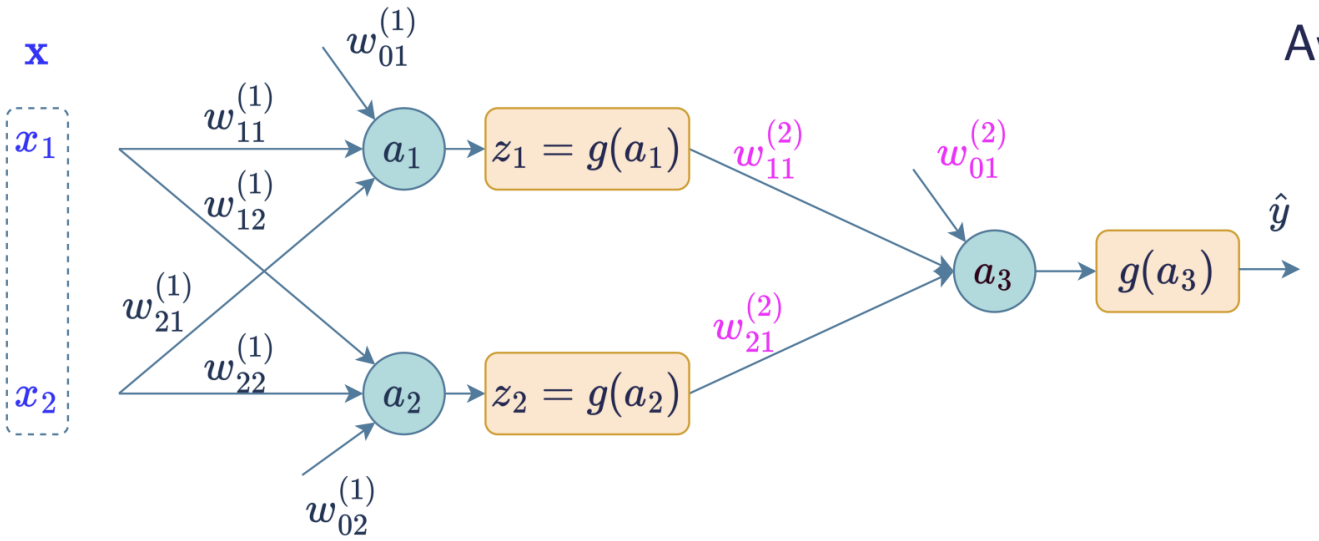
- Quelle non-linéarité ?
  - Fonction signe ?

  $\Rightarrow$ dérivée problématique ...
  - Fonctions tanh, sigmoïde, ...



Plan

Pour l'inférence



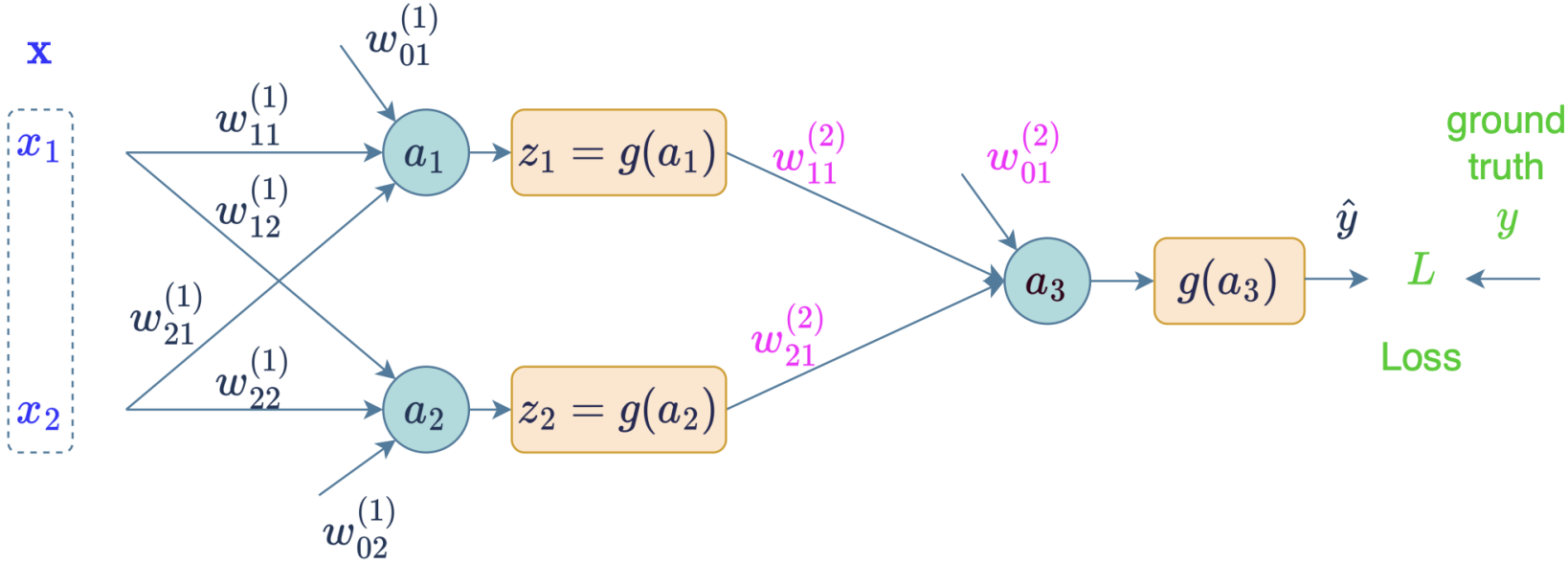
Avec $g(x) = \dfrac{1}{1+\exp(-x)}$

- $a_1 = w_{01} + w_{11}x_1 + w_{21}x_2$
- $a_2 = w_{02} + w_{12}x_1 + w_{22}x_2$
- $z_1 = g(a_1)$
- $z_2 = g(a_2)$
- $a_3 = w_{01} + w_{11}z_1 + w_{21}z_2$
- $\hat{y} = g(a_3)$

Vocabulaire

- Inférence : passe forward
- g fonction d'activation (non linéarité du réseau)
- a_i activation du neurone i
- z_i sortie du neurone i (transformé non linéaire de l'activation).

Pour l'apprentissage



Objectif : apprendre les poids

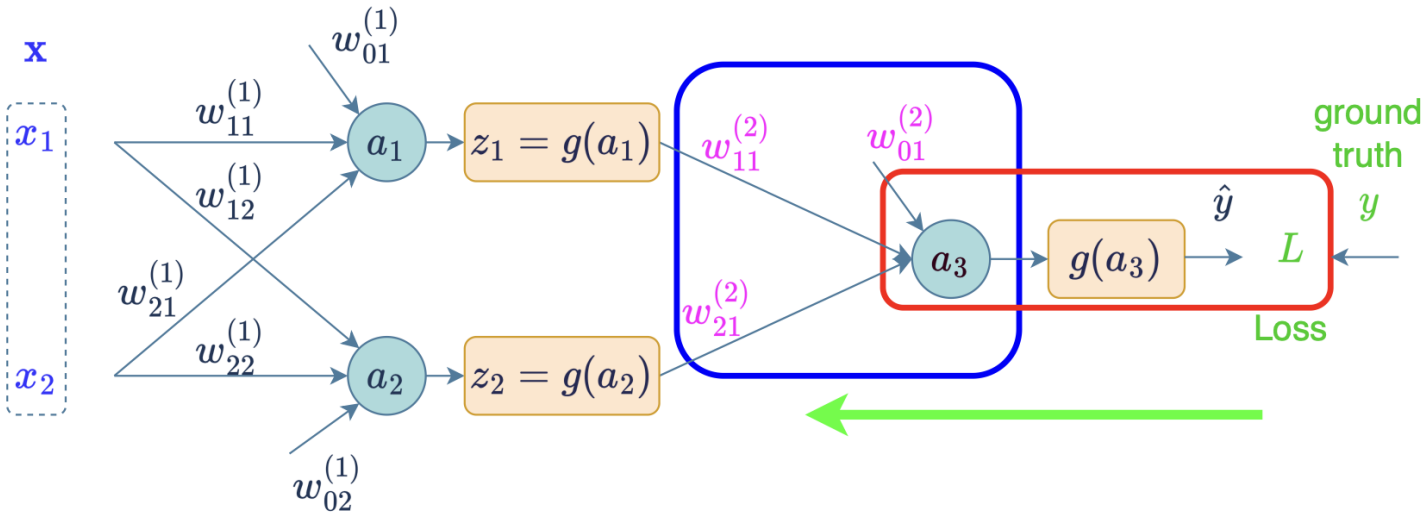
- Choix d'un coût : moindres carrés
  $\mathrm{L}(\hat{\mathrm{y}}, \mathrm{y}) = (\hat{\mathrm{y}} - \mathrm{y})^2$ (pourquoi est ce un bon choix ?)
- Mais à quel(s) neurone(s) et comment répartir l'erreur entre les poids ?

$\Rightarrow$ Rétro-propagation de l'erreur :

- corriger un peu tous les poids ...
- en estimant la part de chacun dans l'erreur
- en commençant par la fin et en figeant au fur et à mesure le réseau

$\Rightarrow$ descente de gradient : on cherche à calculer tous les $\frac{\partial \mathrm{L}(\hat{\mathrm{y}},\mathrm{y})}{\partial \mathrm{w_{ij}}}$

Calcul du gradient : chain rule



Forward:
$\hat{y} = 0.5$
$y = -1$

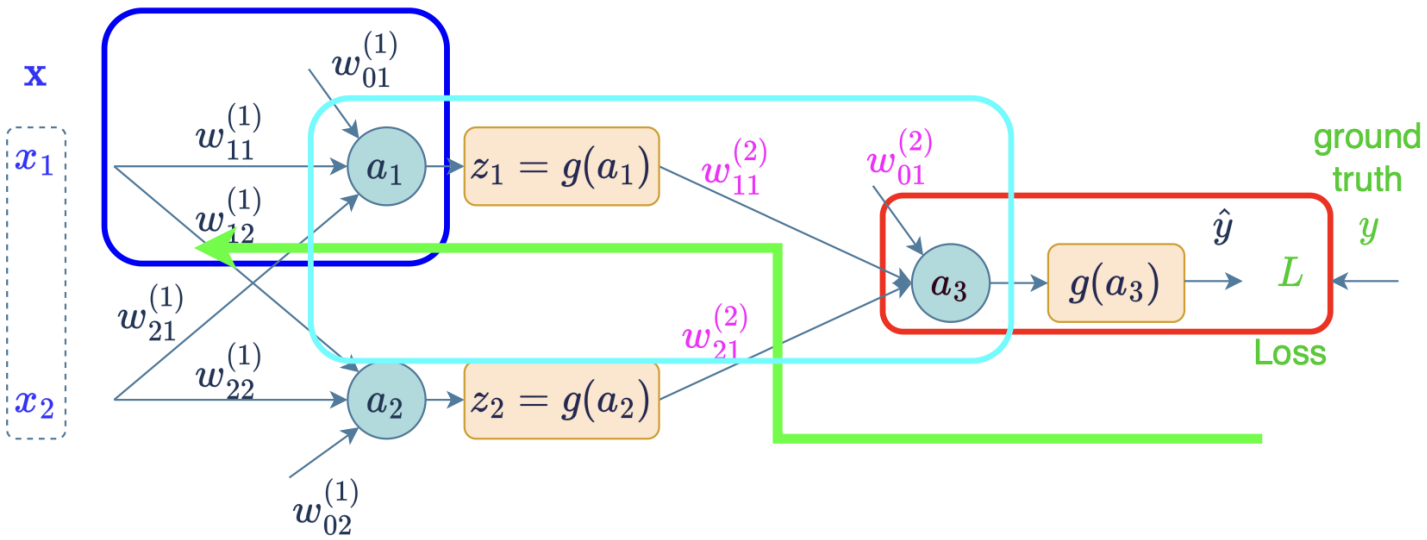
Backward, poids de la **dernière couche** : $\nabla_{w_{ij}^{(2)}} L(\hat{y}, y)$

$$L(\hat{y}, y) = (g(a_3) - y)^2 = \left(g\left(w_{01}^{(2)} + w_{11}^{(2)} z_1 + w_{21}^{(2)} z_2\right) - y\right)^2$$

$$\frac{\partial L}{\partial w_{i1}^{(2)}} = \frac{\partial L}{\partial a_3} \frac{\partial a_3}{\partial w_{i1}^{(2)}} \quad \text{avec} \quad \left| \begin{array}{ll} \frac{\partial L}{\partial a_3} & = \frac{\partial L}{\partial g(a_3)} \frac{\partial g(a_3)}{\partial a_3} = \frac{\partial (g(a_3)-y)^2}{\partial a_3} = 2g'(a_3)(g(a_3) - y) \\ \frac{\partial a_3}{\partial w_{i1}^{(2)}} & = \frac{\partial (w_{01}^{(2)} + w_{11}^{(2)} z_1 + w_{12}^{(2)} z_2)}{\partial w_{i1}^{(2)}} = z_i \end{array} \right.$$

Soit: $\frac{\partial L}{\partial w_{i1}^{(2)}} = 2g'(a_3)(\hat{y} - y)z_i$ $\quad \Longrightarrow$ Mise à jour possible

Calcul du gradient : chain rule



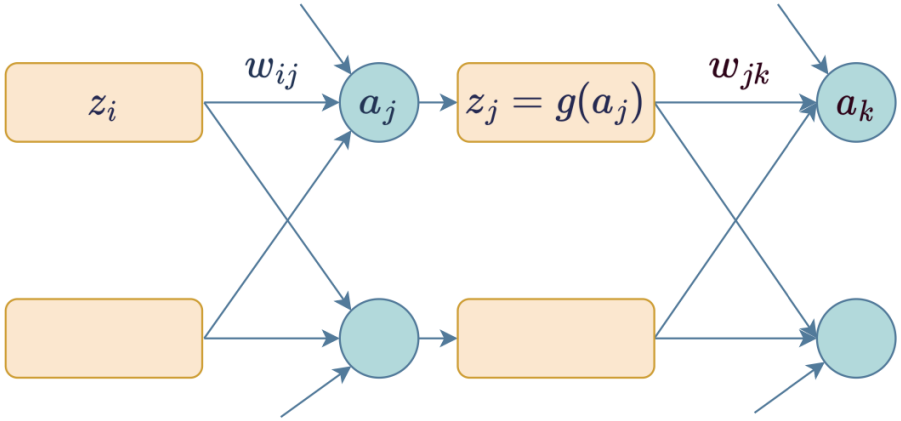
Forward:
$\hat{y} = 0.5$
$y = -1$

Backward, poids de la première couche: $w_{i1}^{(1)}$ (par exemple)

$$\frac{\partial L}{\partial w_{i1}} = \frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial w_{i1}} \quad \text{avec} \quad \left| \begin{array}{lll} \frac{\partial L}{\partial a_1} & = \frac{\partial L}{\partial a_3}\frac{\partial a_3}{\partial a_1} & = \frac{\partial L}{\partial a_3} g'(a_1) w_{11}^{(2)} \\ \frac{\partial a_1}{\partial w_{i1}} & = \frac{\partial w_{01}^{(1)} + w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2}{\partial w_{i1}^{(1)}} & = x_i \end{array} \right.$$

Soit: $\underbrace{\frac{\partial L}{\partial w_{i1}}}_{\text{correction de } w_{i1}} = \frac{\partial L}{\partial a_1} x_i = \underbrace{\frac{\partial L}{\partial a_3}}_{\text{erreur à propager}} \underbrace{g'(a_1) w_{13}}_{\text{poids de la connexion}} x_i$

Calcul du gradient : chain rule



$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial a_j}{\partial w_{ij}} \frac{\partial L}{\partial a_j} = z_i \frac{\partial L}{\partial a_j}$$

$$\frac{\partial L}{\partial a_j} = \sum_k \frac{\partial a_k}{\partial a_j} \frac{\partial L}{\partial a_k}$$

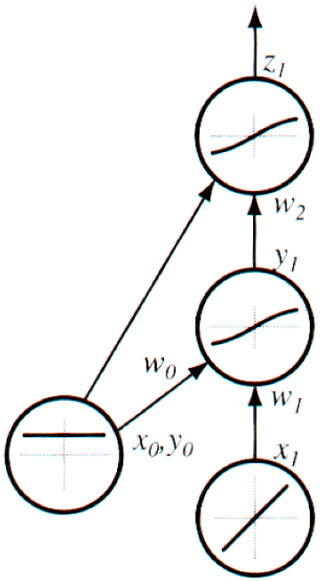
$$\underbrace{\frac{\partial L}{\partial a_j}}_{\text{erreur sur } j} = \sum_k (g'(a_k) w_{jk}) \quad \underbrace{\frac{\partial L}{\partial a_k}}_{\text{erreur à propager}}$$

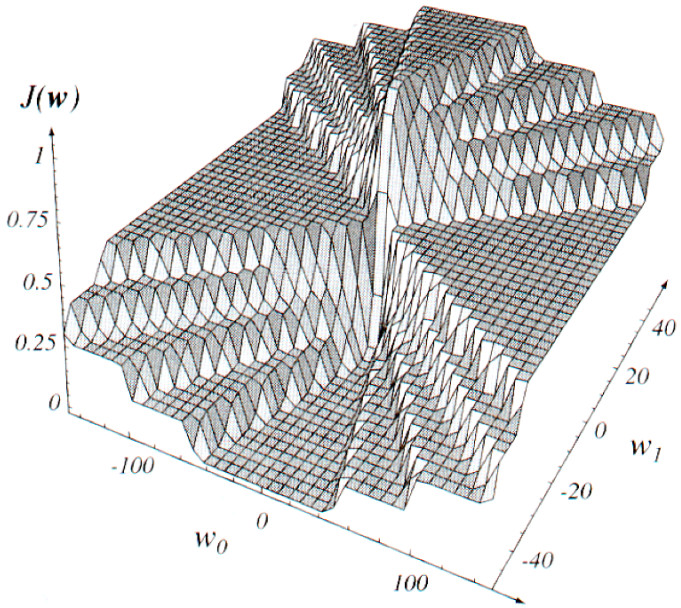
On note: $\delta_j = \frac{\partial L}{\partial a_j}$

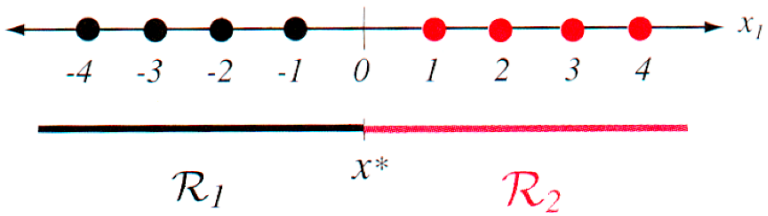
- Lorsque l'erreur *arrive* de plusieurs sources $\Rightarrow$ somme
- Expression de l'erreur de la couche j par rapport à l'erreur de la couche k

Plan

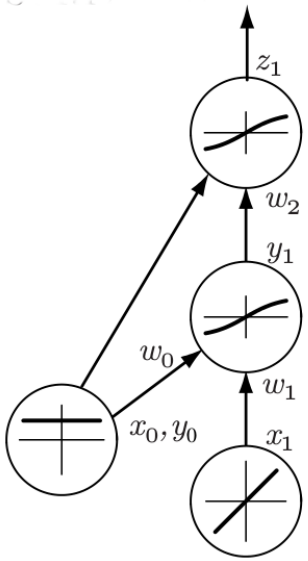
Analyse de la surface d'erreur

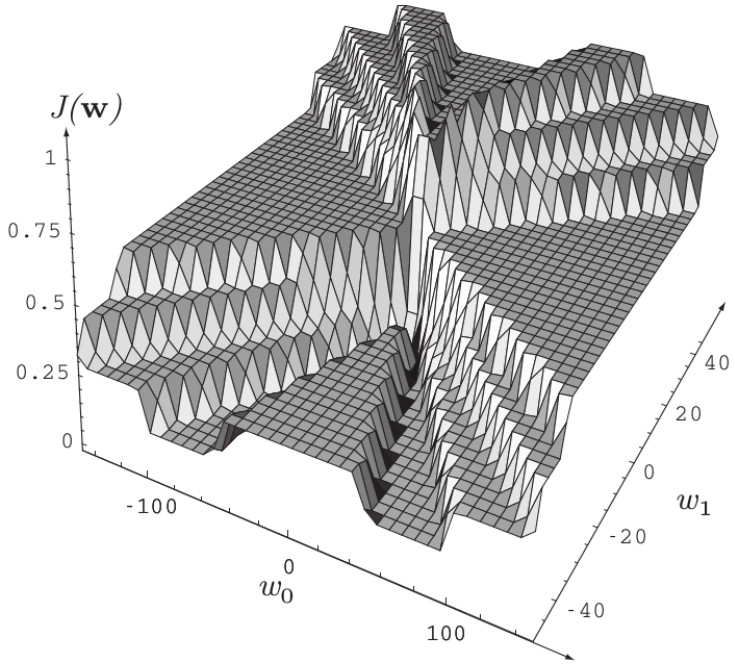


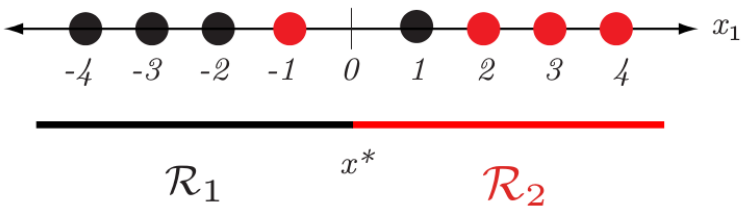




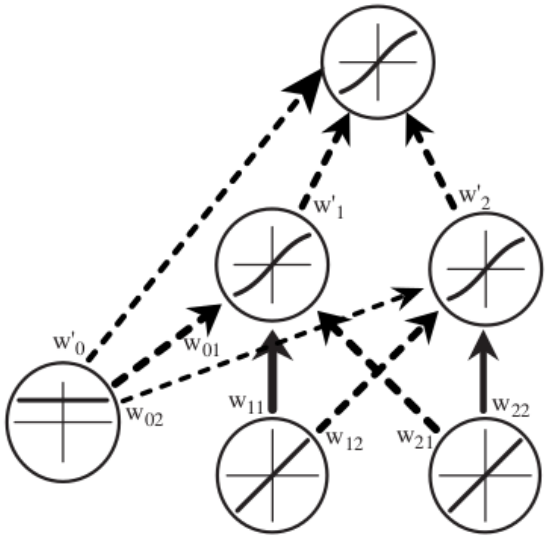
Analyse de la surface d'erreur

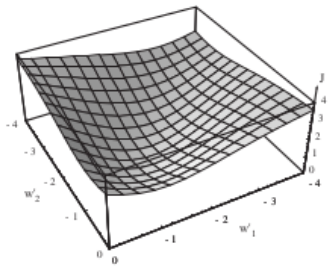


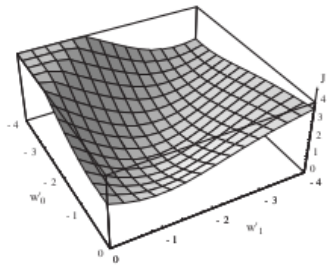


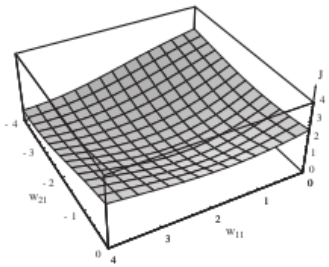


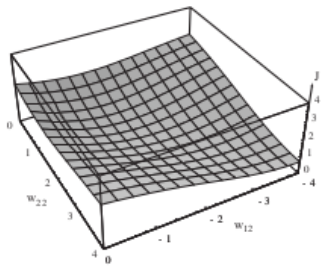
Analyse de la surface d'erreur





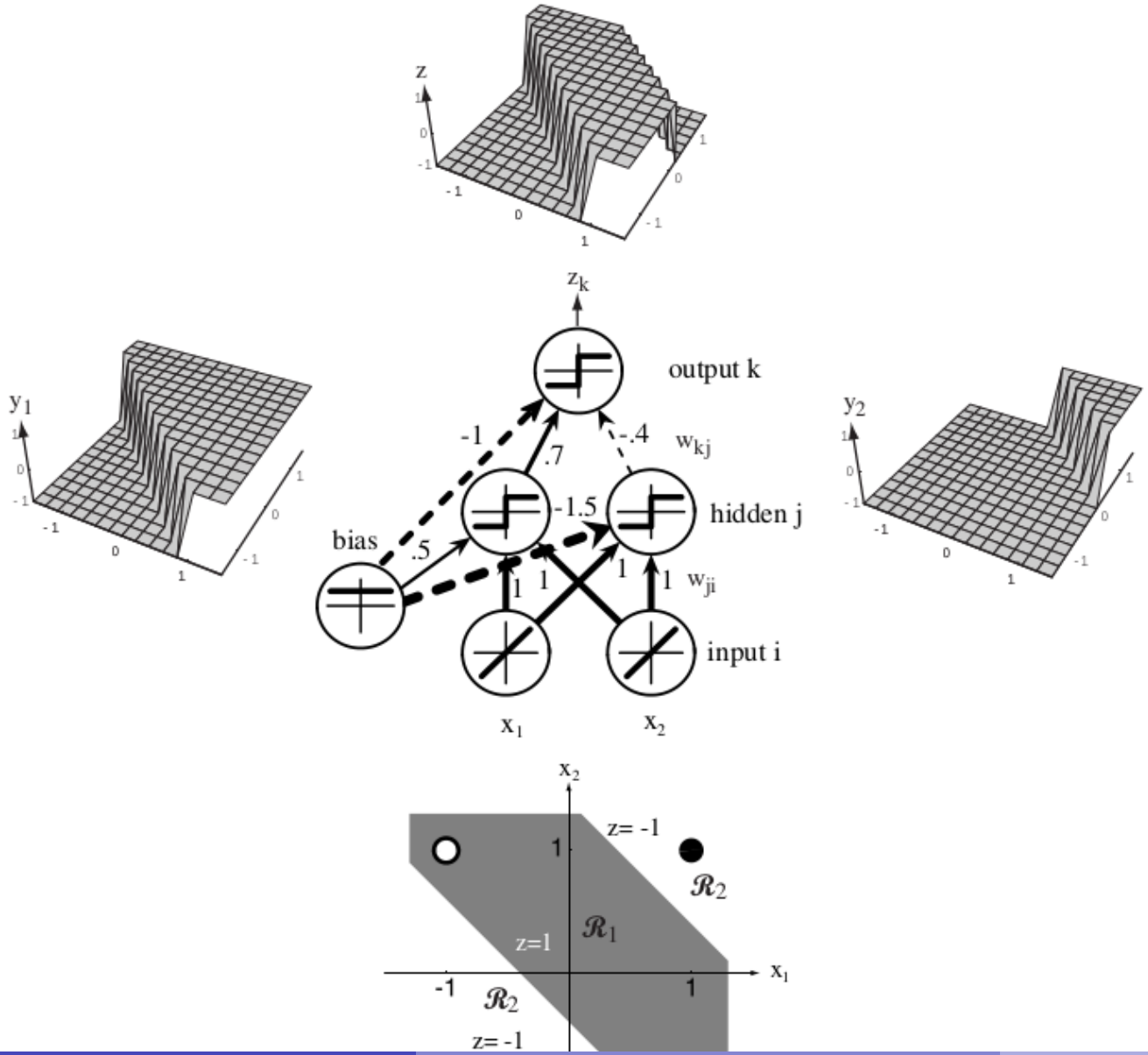






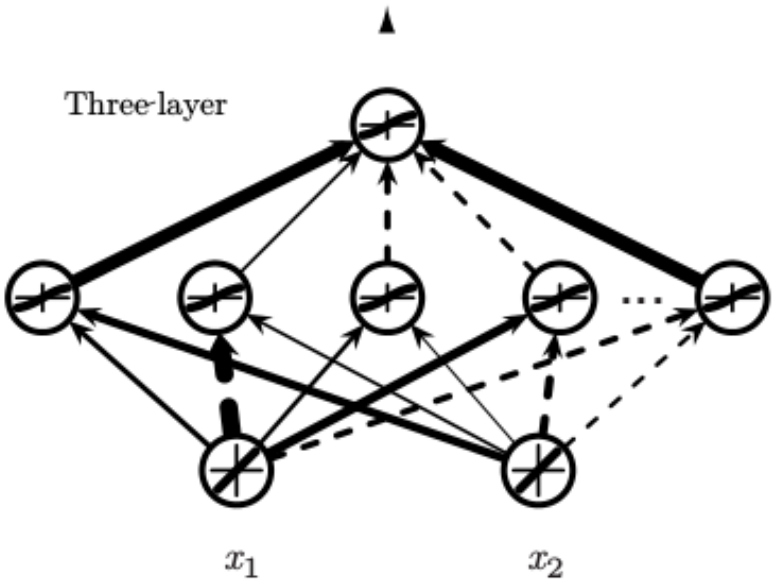
Exemple

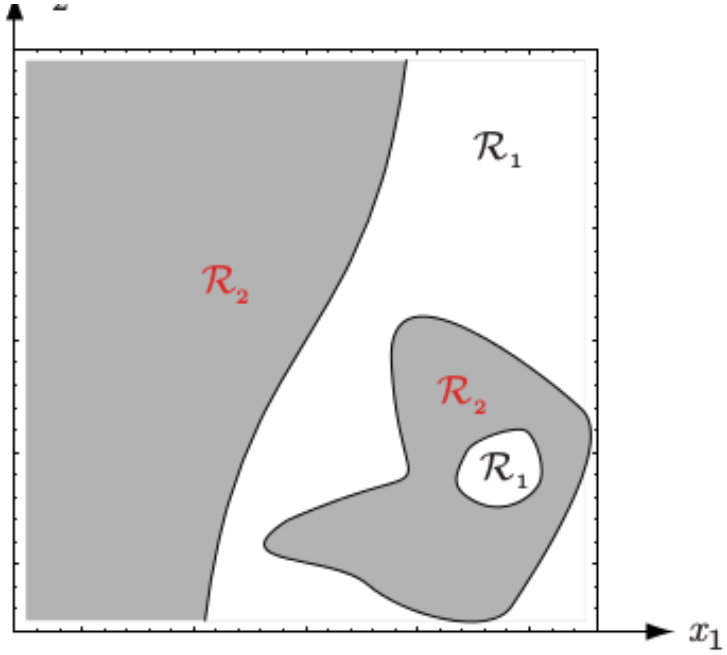
Le XOR selon [Duda et al 00]



Exemple

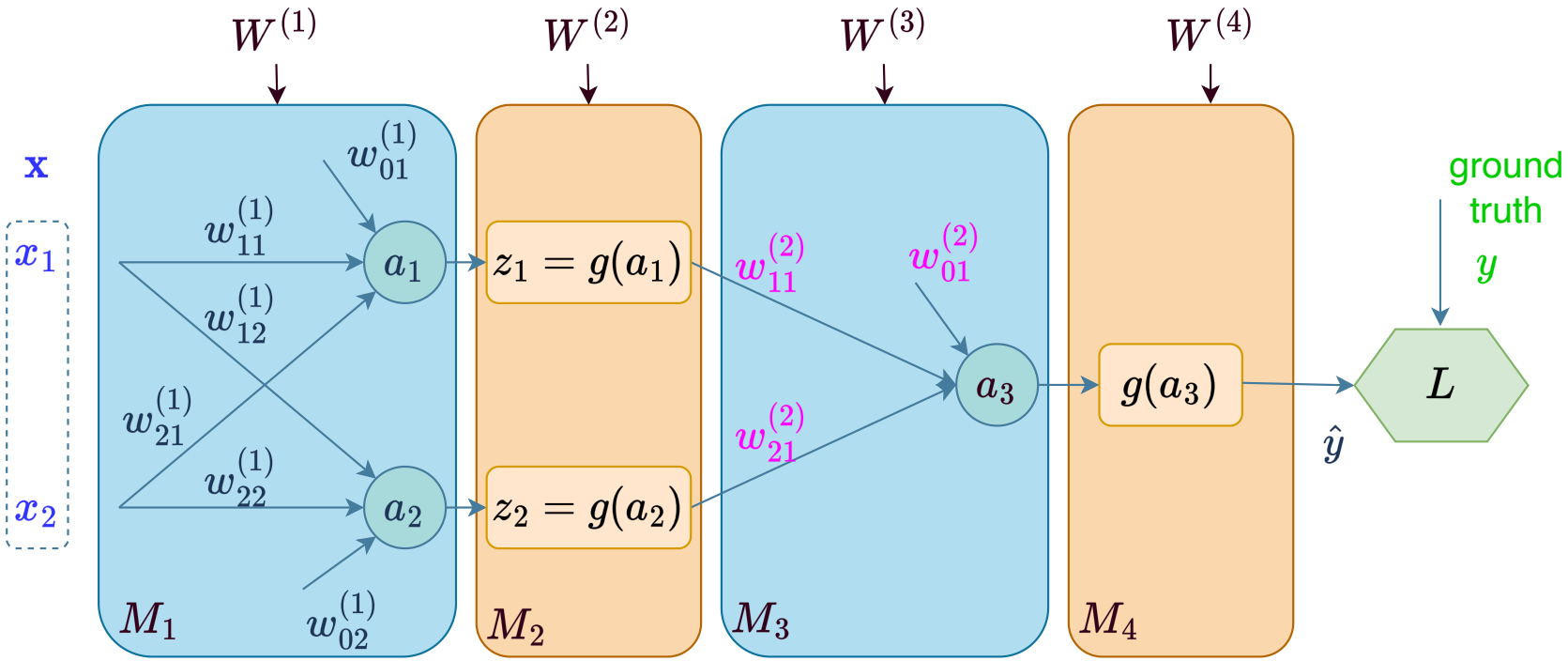
Non convexité des régions apprises [Duda et al 00]





Plan

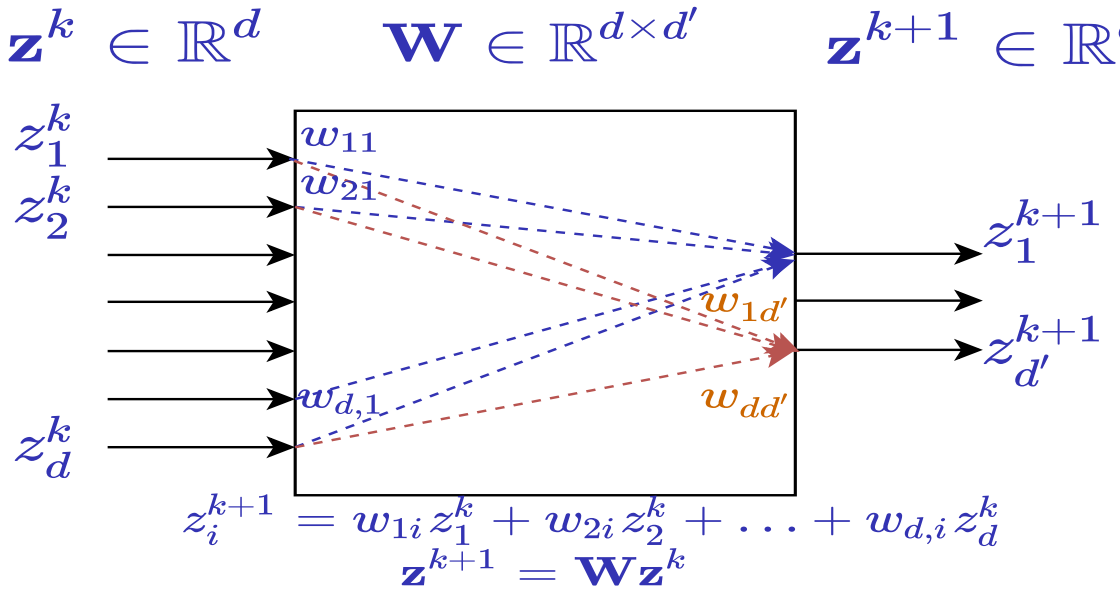
Réseau : assemblage de modules



Un module M^k

- a des entrées : le résultat de la couche précédente z^{k-1}
- a possiblement des paramètres $W^{(k)}$ [vu également comme des entrées]
- produit une sortie z^k

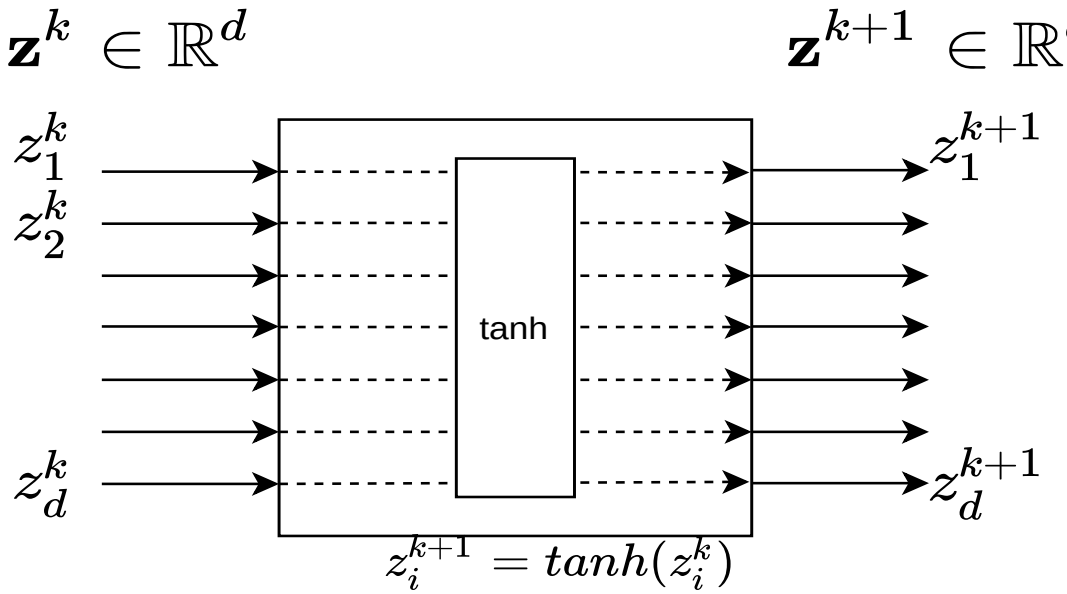
Type usuel de modules : Module linéaire



Transformation linéaire paramétrée de $\mathbb{R}^d$ vers $\mathbb{R}^{d'}$

- $z^k = M^k(z^{k-1}, W^k) = W^k z^{k-1}$ avec $W^k \in \mathbb{R}^d \times \mathbb{R}^{d'}$, $z^{k+1} \in \mathbb{R}^{d'}$
- Chaque sortie $z_i^{k+1} = W_{i,.}^k z^k =< w_i^k, z^k >$ correspond au calcul d'un perceptron
- La matrice W^k est l'empilement des w_i, poids de chaque perceptron.

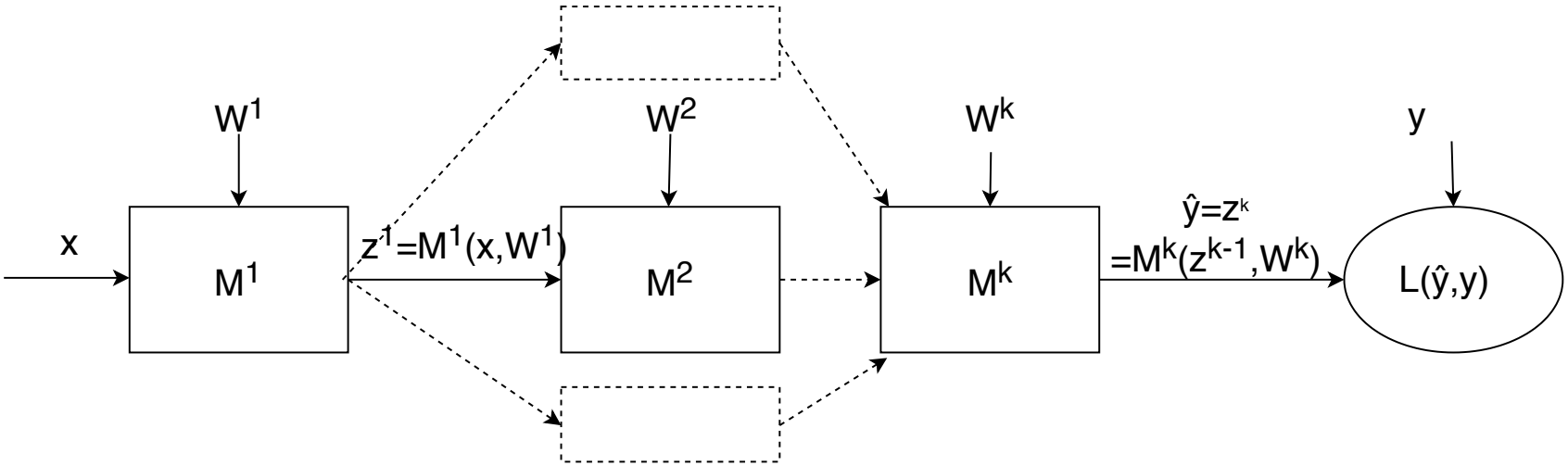
Type usuel de modules : Module d'activation



Fonction d'activation de $\mathbb{R}^d$ vers $\mathbb{R}^d$

- tangente hyperbolique :
  $\mathrm{M^k}(\mathrm{z^{k-1}}, 0) = \tanh(\mathrm{z^{k-1}}) = (\tanh(\mathrm{z_1^{k-1}}), \tanh(\mathrm{z_2^{k-1}}), \ldots, \tanh(\mathrm{z_d^{k-1}}))$
- sigmoïde : $\mathrm{M^k}(\mathrm{z^{k-1}}, 0) = \sigma(\mathrm{z^{k-1}}) = (\sigma(\mathrm{z_1^{k-1}}), \sigma(\mathrm{z_2^{k-1}}), \ldots, \sigma(\mathrm{z_d^{k-1}}))$
- ReLU : $\mathrm{M^k}(\mathrm{z^{k-1}}, 0) = \mathrm{ReLU}(\mathrm{z^{k-1}}) =$
  $(\max(0, \mathrm{z_1^{k-1}}), \max(0, \mathrm{z_2^{k-1}}), \ldots, \max(0, \mathrm{z_d^{k-1}}))$

Type usuel de modules : Module de coût

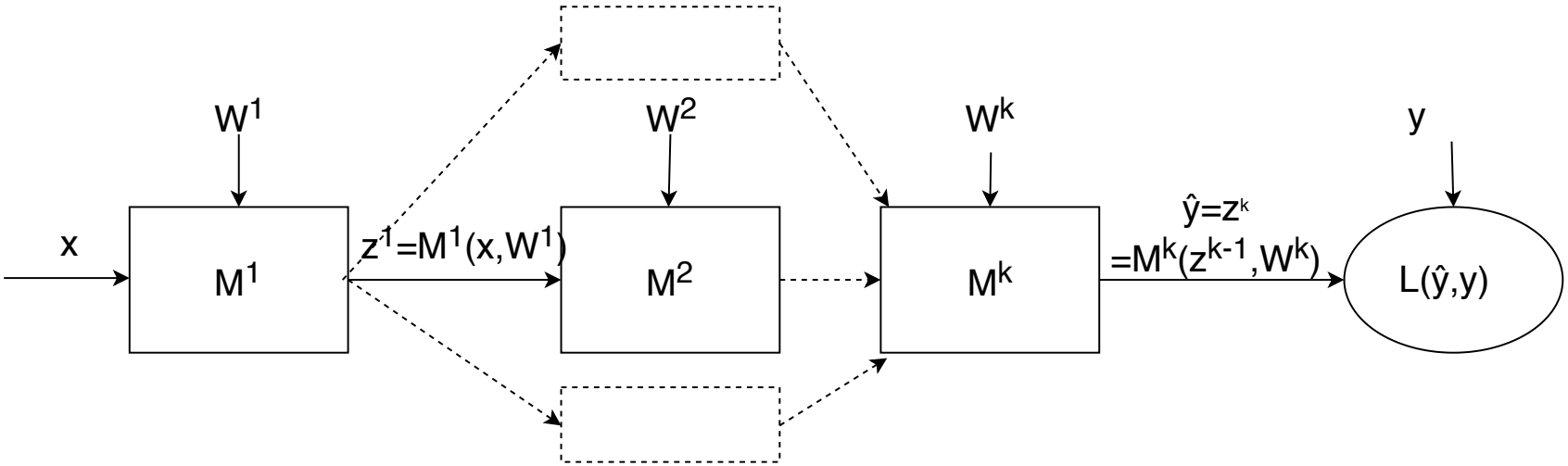


Fonction de coût

Bloc final : deux entrées, la supervision et la sortie du réseau $\hat{y} = z^k$.

- MSE : $L(\hat{y}, y) = \|\hat{y} - y\|^2$
- Negative Log-Likelihood : $L(\hat{y}, y) = -\sum_{i=1}^{d} y_i \log \hat{y}_i$
- KL-divergence : $L(\hat{y}, y) = -\sum_{i=1}^{d} y_i \log \frac{\hat{y}_i}{y_i}$

Un réseau Assemblage de modules

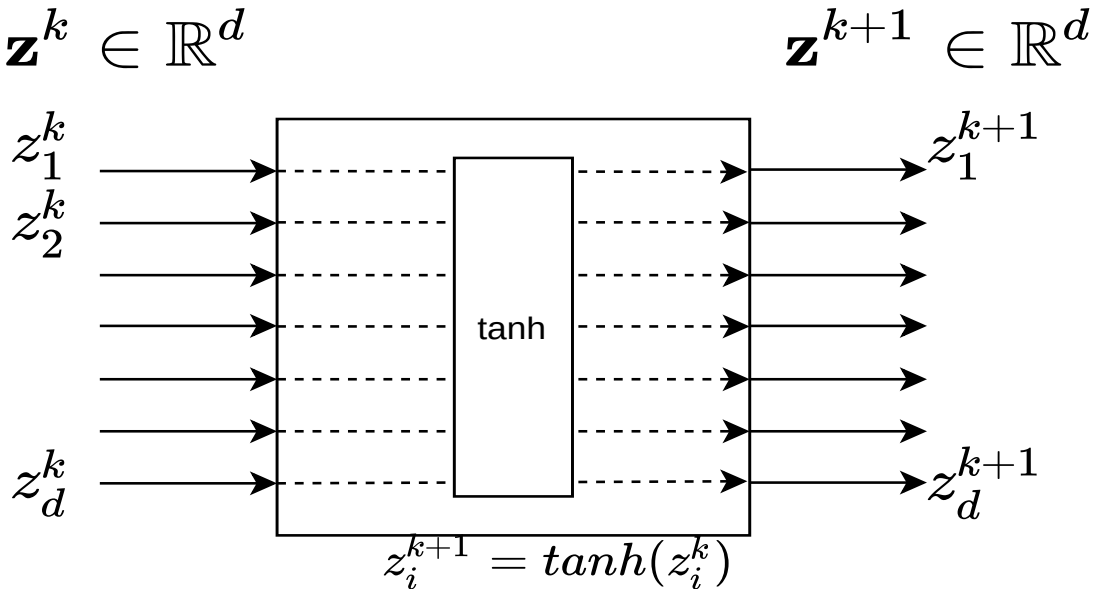


Un module M^k est caractérisé

- par ses entrées : le résultat de la couche précédente z^{k-1} (et potentiellement d'autres variables)
- par possiblement ses paramètres W^k (vu également comme des entrées)
- produit une sortie $z^k = M^k(z^{k-1}, W^k)$

D'un point de vue formel, il n'y pas de différences entre les paramètres du module et les entrées : ce sont tous des arguments de la fonction du module.

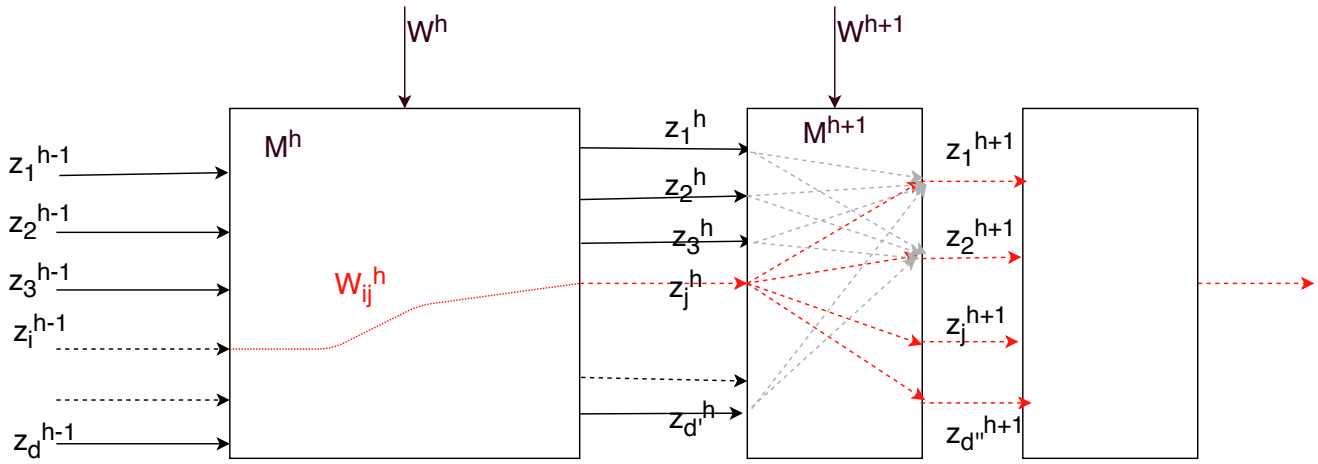
Rétro-propagation du gradient



Pour apprendre le réseau :

- Pour chaque module, il faut calculer $\nabla_{W^k} L(\hat{y}, y)$
- Cas simple : paramètres constants (module d'activation), le gradient est nul (il n'y a rien à apprendre pour ce module)
- Rétro-propagation pour les autres.

Zoom sur un module



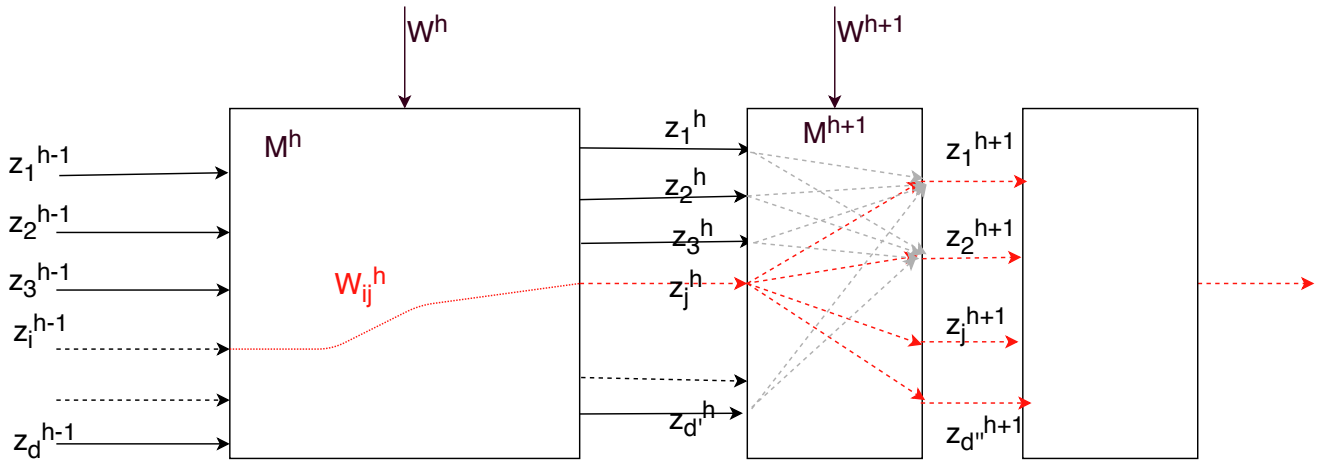
Rétro-propagation pour M^h, $z^h = M(z^{h-1}, W^h)$

- $\frac{\partial L}{\partial w_{ij}^h} = \sum_k \frac{\partial L}{\partial z_k^h}\frac{\partial z_k^h}{\partial w_{ij}^h} = \frac{\partial L}{\partial z_j^h}\frac{\partial z_j^h}{\partial w_{ij}^h} = \frac{\partial L}{\partial z_j^h}\frac{\partial M^h(z^{h-1},W^h)}{\partial w_{ij}^h}$ (w_{ij}^h n'influe que sur z_j^h)
- $\frac{\partial L}{\partial z_j^h} = \sum_k \frac{\partial L}{\partial z_k^{h+1}}\frac{\partial z_k^{h+1}}{\partial z_j^h} = \sum_k \frac{\partial L}{\partial z_k^{h+1}}\frac{M^{h+1}(z^h,W^h)}{\partial z_j^j}$
- On introduit $\delta_j^h = \frac{\partial L}{\partial z_j^h} = \sum_k \delta_k^{h+1}\frac{\partial M^{h+1}(z^h,W^h)}{\partial z_j^j}$: $\frac{\partial L}{\partial w_{ij}^h} = \delta_j^h\frac{\partial M^h(z^{h-1},W^h)}{\partial w_{ij}^h}$

$\Rightarrow$ Les δ_i^h sont calculés en partant de la fin du réseau

- Pour la dernière couche, $\delta_j^{last} = \frac{\partial L(z^{last},y)}{\partial z_j^{last}}$, le gradient du coût par rapport à la prédiction.

Zoom sur un module



Rétro-propagation pour M^h, $z^h = M(z^{h-1}, W^h)$

Avec $\delta_j^h = \frac{\partial L}{\partial z_j^h} = \sum_k \delta_k^{h+1} \frac{\partial M^{h+1}(z^h, W^h)}{\partial z_j^j}$: $\frac{\partial L}{\partial w_{ij}^h} = \delta_j^h \frac{\partial M^h(z^{h-1}, W^h)}{\partial w_{ij}^h}$

Pour chaque module, on a besoin :

- du gradient $\nabla_{W^h} M^h(z^{h-1}, W^h)$ par rapport à ses paramètres : maj des paramètres (nul si pas de paramètres)
- du gradient $\nabla_{z^{h-1}} M^h(z^{h-1}, W^h)$ par rapport à ses entrées : rétro-propagation de l'erreur

Complexité et expressivité

- Efficacité en apprentissage
  - En O (|w|) pour chaque passe d'apprentissage où |w| est le nombre de poids
  - Il faut typiquement plusieurs centaines de passes (voir plus loin)
  - Il faut typiquement recommencer plusieurs dizaines de fois un apprentissage en partant avec différentes initialisations des poids
- Efficacité en reconnaissance
  - Possibilité de temps réel

Expressivité

- Quelle influence du nombre de couches ?
- du nombre de neurones par couche ?

$\Rightarrow$ Une couche cachée suffit pour un apprentissage universel ! Mais ...

Vers les réseaux profonds

Problème : plus le réseau est profond plus il est dur à entraîner

- le gradient s'évapore (vanishing)
- le sur-apprentissage est très favorisé

Quelques solutions

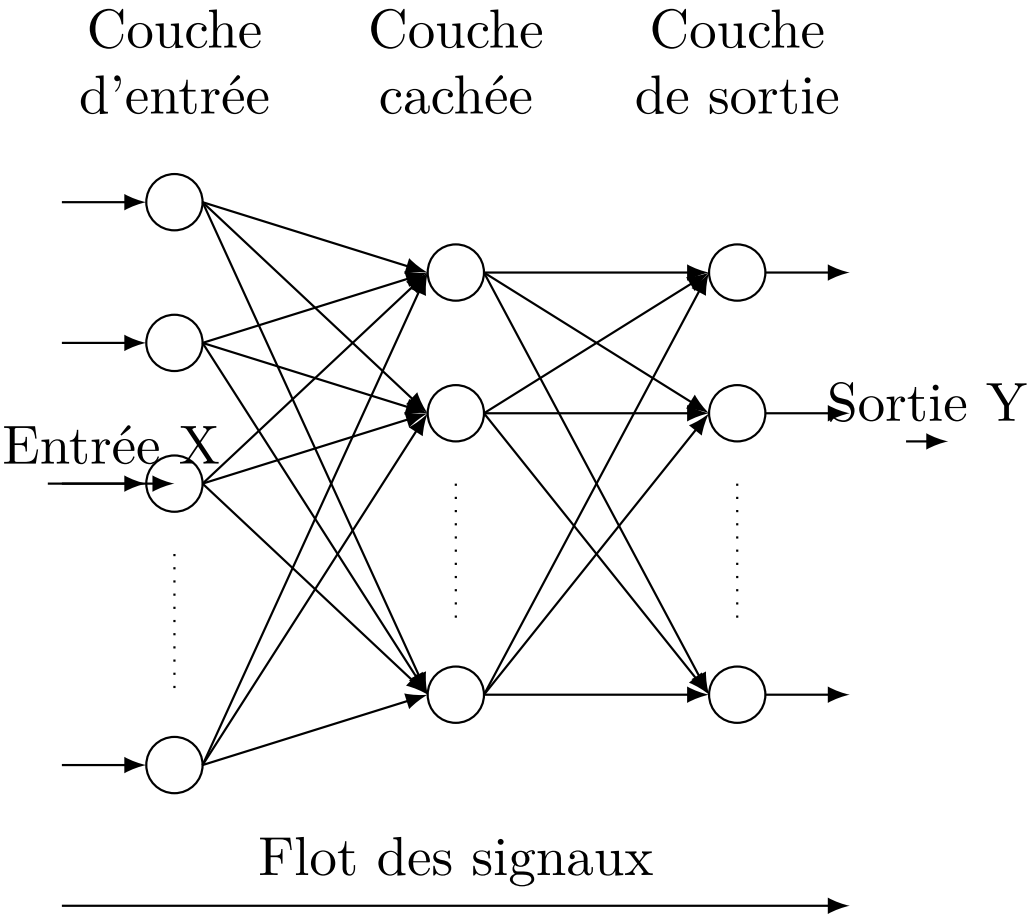
- utiliser des architectures peu propices au sur-apprentissage (convolutives, RBM, ...)
- Early-stopping
- Apprentissage en bruitant les données d'entrées

$\Rightarrow$ pas suffisant

- Première passe d'apprentissage pour "bien initialiser" les couches $\Rightarrow$ (apprentissage couche par couche, auto-encoders)
- Drop-out : permet de limiter le sur-apprentissage (éteindre/supprimer un nombre de neurones aléatoirement pendant l'apprentissage)
- utilisation de fonctions d'activation spécifiques (ReLU etc)
- utilisation d'architecture spécifiques (couches résiduelles etc)

Plan

Topologie typique



Pour chaque neurone k, la sortie z_k

$$z_k = g\left(\sum_{j=0}^{d} w_{j,k} z_j\right) = g\left(a_k\right)$$

où

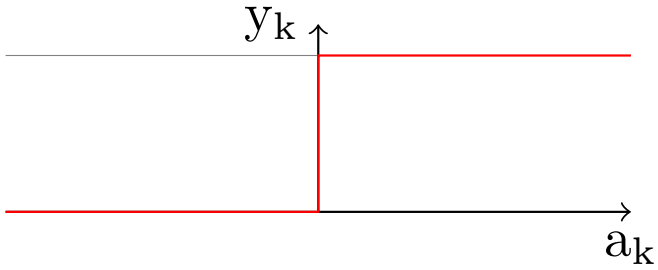
- $w_{j,k}$: poids de la connexion de cellule j à la cellule k
- a_k : activation de la cellule k $a_k = \sum_{j=0}^{d} w_{j,k} z_j$
- g : fonction d'activation

Apprentissage :

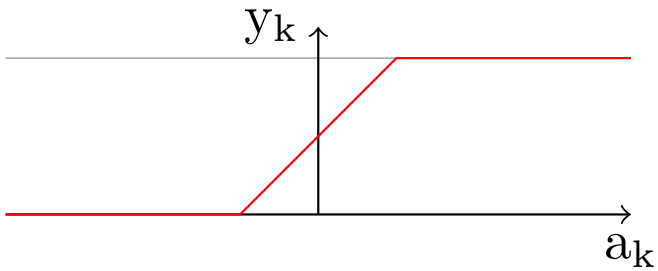
- Minimiser la fonction de coût $L\left(W, \{X, Y\}\right)$ en fonction du paramètre $W = (w_{i,j})$
- Algorithme de rétro-propagation de gradient $\Delta w_{i,j} \propto \frac{\partial L}{\partial w_{i,j}}$

Fonction d'activation

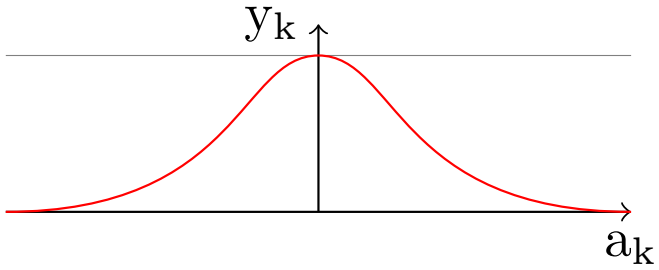
- Fonction à seuil



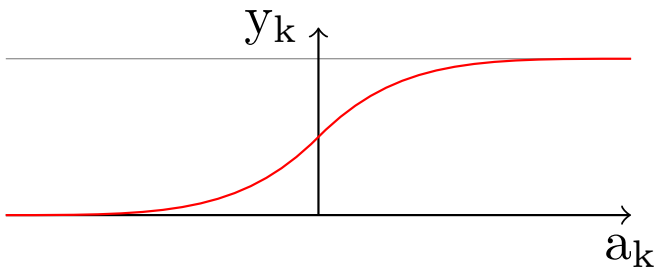
- Fonction à rampe



- Fonction radiale



- Fonction sigmoïde



  - $g(a) = \frac{1}{1+\exp(-a)}$
  - $g'(a) = g(a)(1 - g(a))$

Algorithme

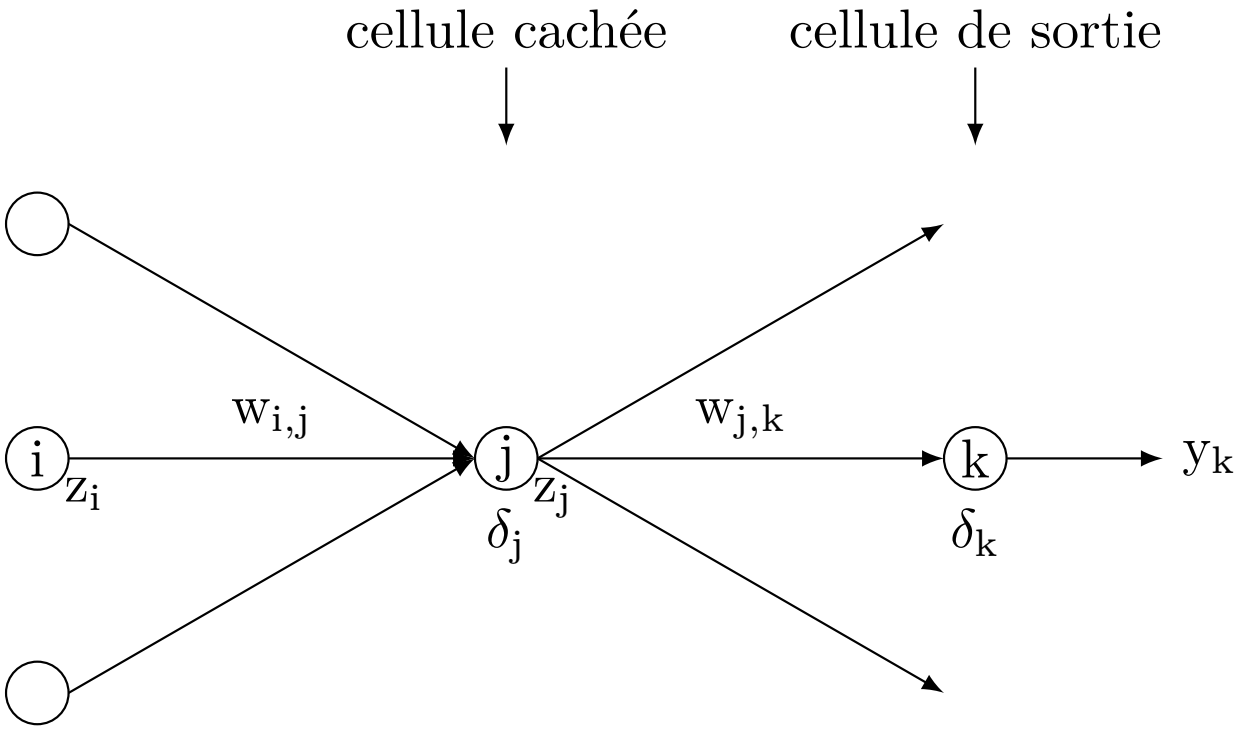
1. Présentation d'un/des exemple(s) parmi l'ensemble d'apprentissage
2. Calcul de l'état du réseau (phase forward)
3. Calcul de l'erreur avec un coût donné : e.g. $= (y - \hat{y})^2$)
4. Calcul des gradients (par l'algorithme de rétro-propagation du gradient)
5. Modification des poids
6. Critère d'arrêt (sur l'erreur, nombre de présentation d'exemples...)
7. Retour en 1

La rétro-propagation de gradient

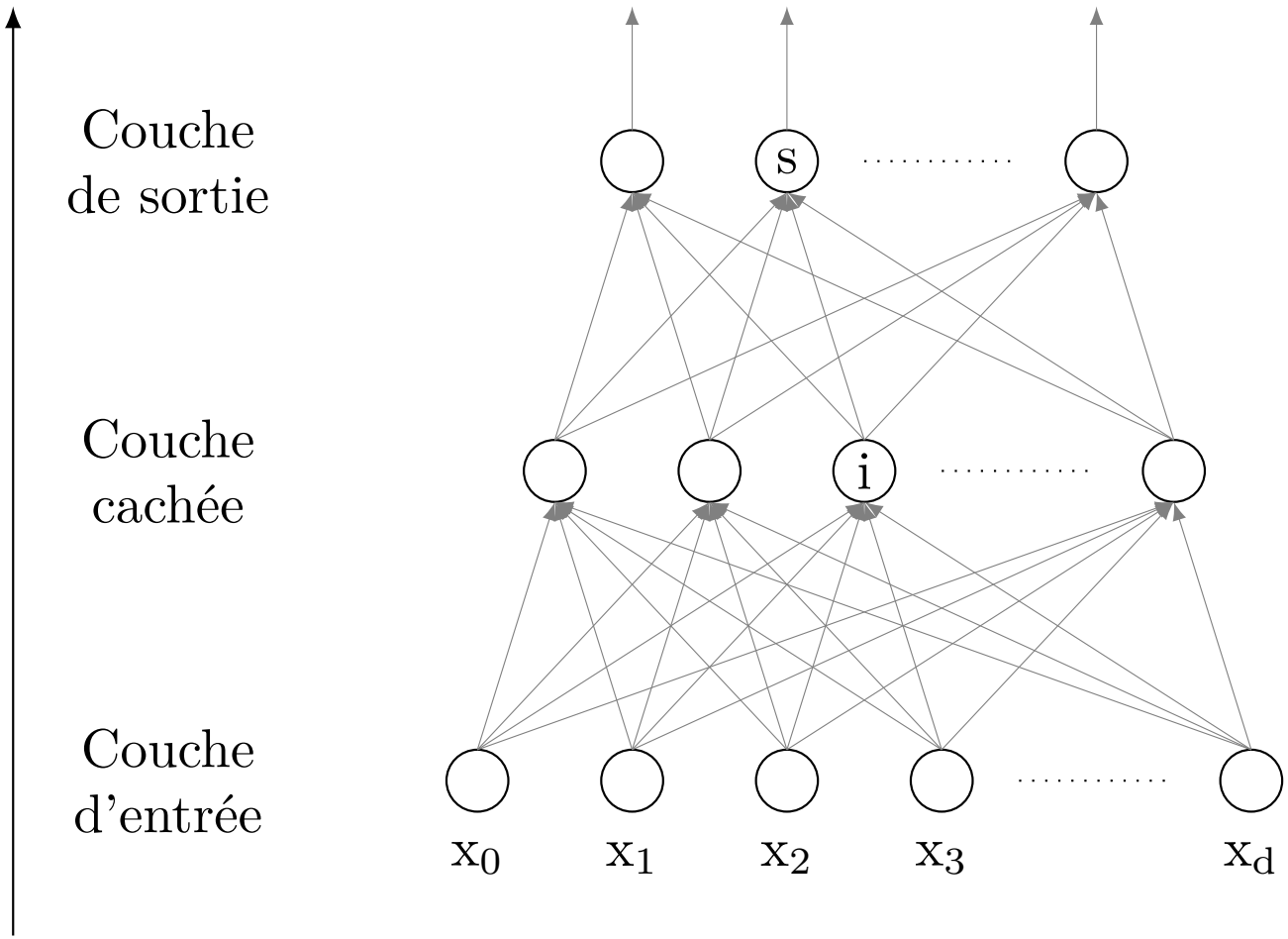
- Le problème :
  - Détermination des responsabilités (credit assignment problem)
  - Quelle connexion est responsable, et de combien, de l'erreur ?
- Principe :
  - Calculer l'erreur sur une connexion en fonction de l'erreur sur la couche suivante
- Deux étapes :
  1. Evaluation des dérivées de l'erreur par rapport aux poids
  2. Utilisation de ces dérivées pour calculer la modification de chaque poids

La rétro-propagation de gradient

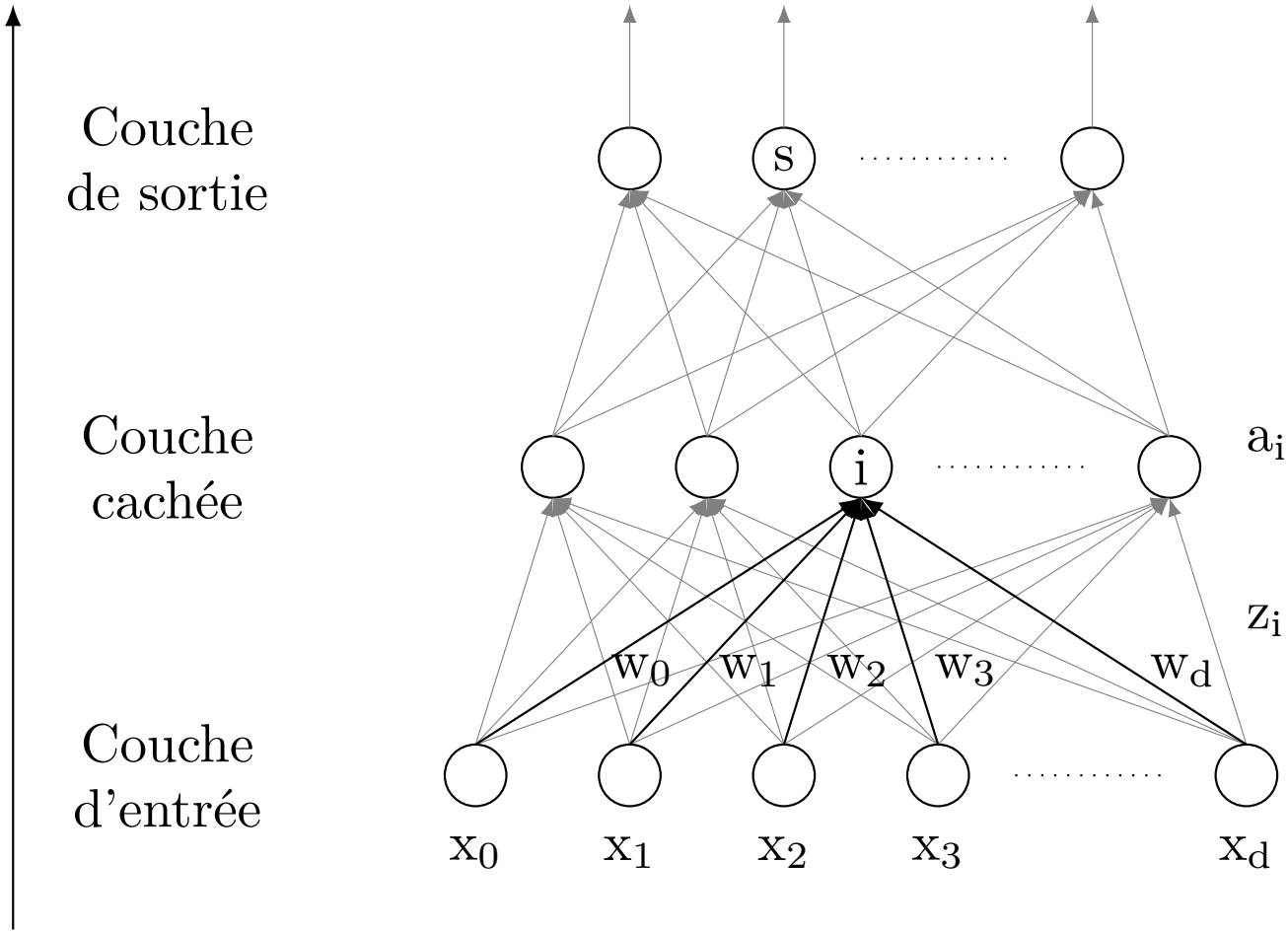
- a_i : activation de la cellule i
- z_i : sortie de la cellule i
- δ_i : erreur attachée à la cellule i



Passe avant (forward) Illustrations J.-N. Vittaut



Passe avant (forward) Illustrations J.-N. Vittaut



$$a_i(x) = w_0 + \sum_{j=1}^{d} w_j x_j$$

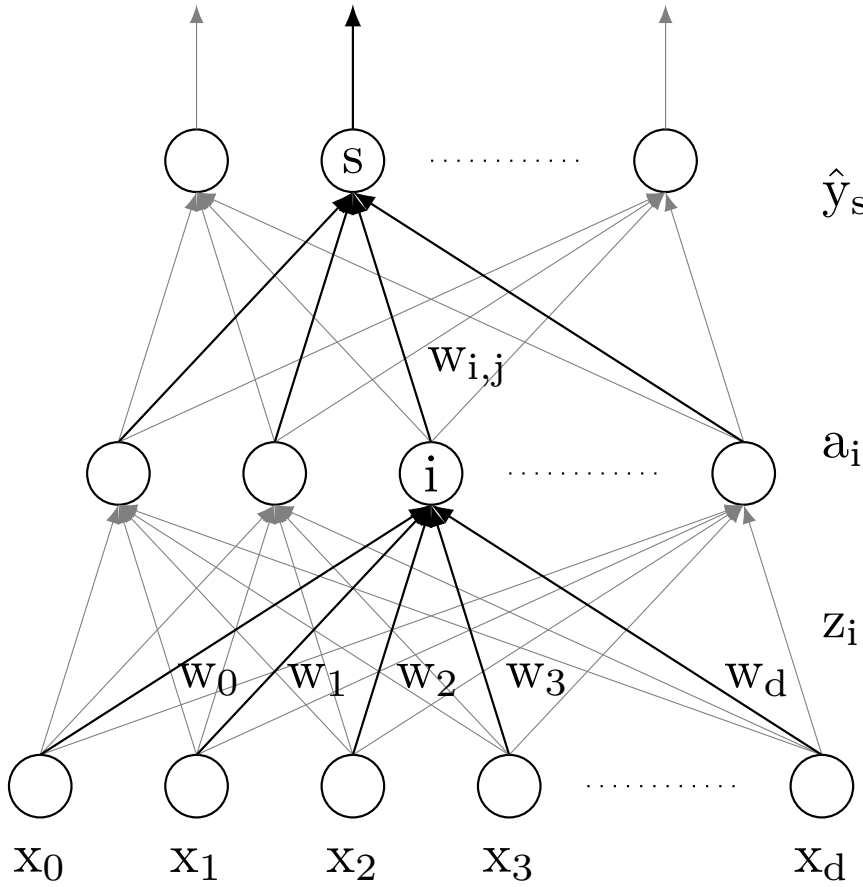
$$z_i(x) = g\left(a_i(x)\right)$$

Passe avant (forward) Illustrations J.-N. Vittaut

Couche de sortie

Couche cachée

Couche d'entrée

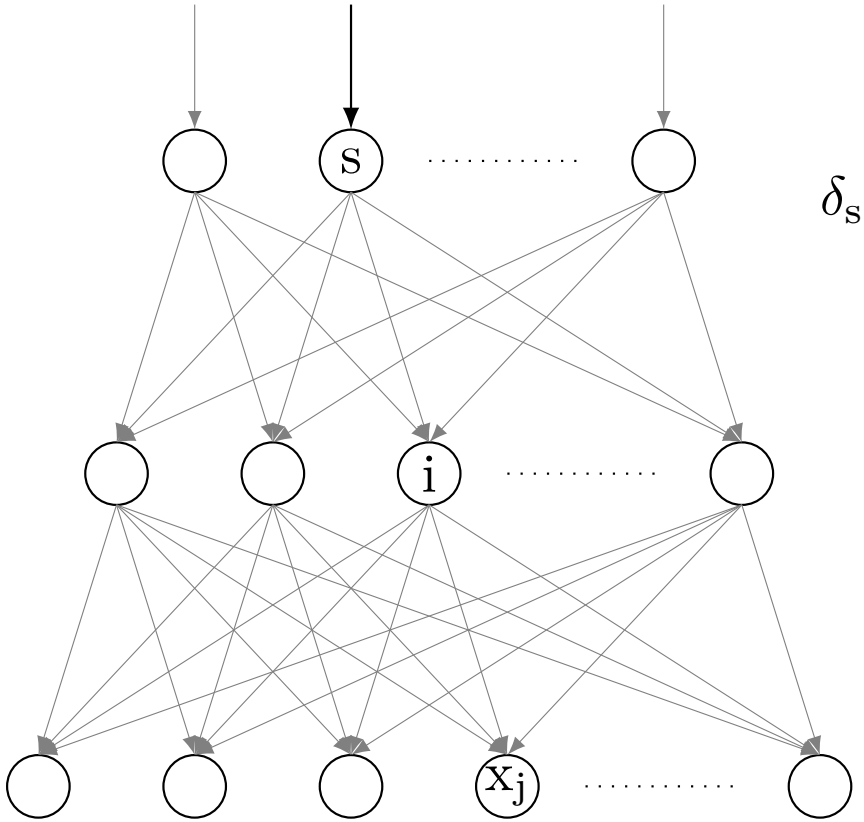


$$\hat{y}_s(x) = \sum_{i=1}^{k} w_{i,s} z_i$$

$$a_i(x) = w_0 + \sum_{j=1}^{d} w_j x_j$$

$$z_i(x) = g\left(a_i(x)\right)$$

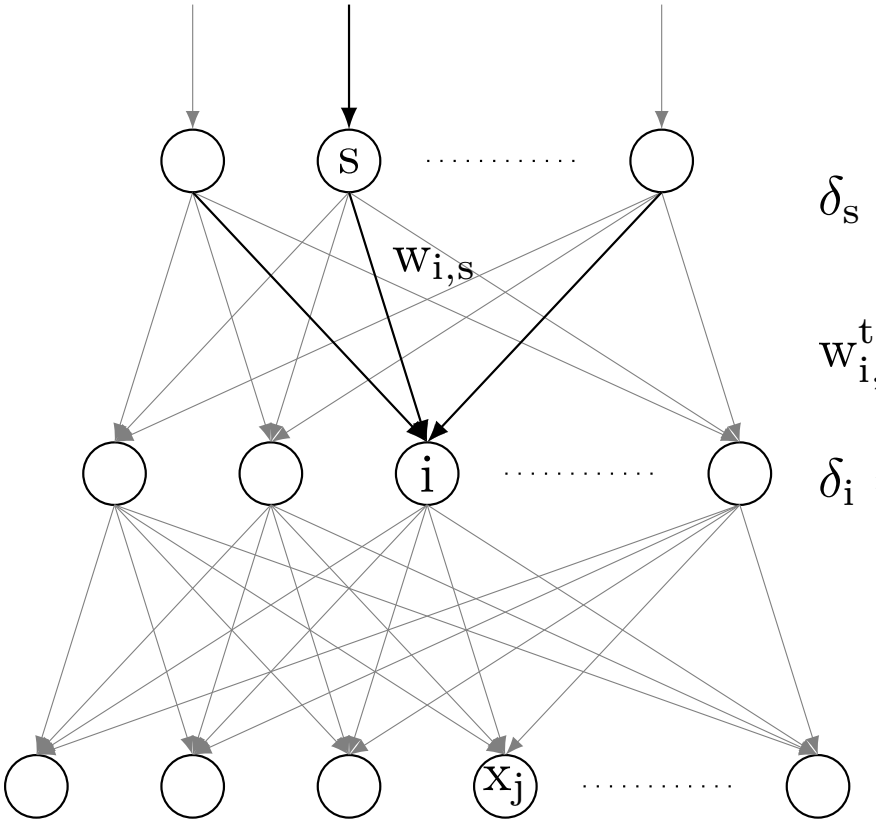
Passe arrière (backward)



$$\delta_s = g'(a_s)(y_s - \hat{y}_s)$$

- δ_s : erreur en sortie
- δ_i : somme des erreurs provenant des cellules suivantes

Passe arrière (backward)



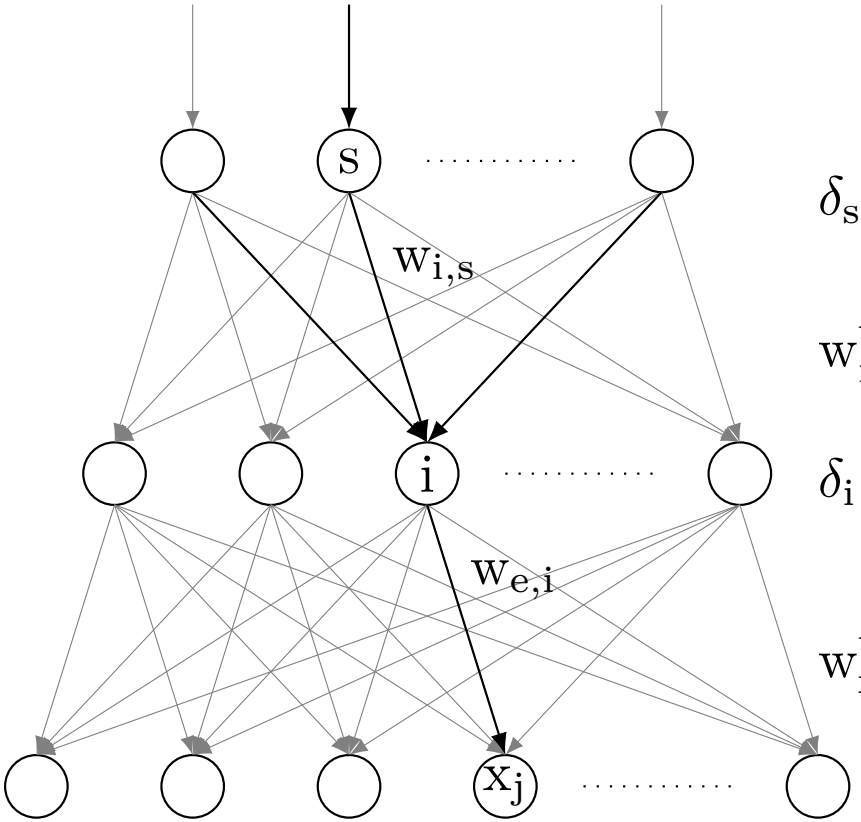
$$\delta_s = g'(a_s)(y_s - \hat{y}_s)$$

$$w_{i,s}^{t+1} = w_{i,s}^{t} - \eta(t)\delta_s a_i$$

$$\delta_i = g'(a_i) \sum_{s \in \text{coucheSuiv.}} w_{i,s}\delta_s$$

- δ_s : erreur en sortie
- δ_i : somme des erreurs provenant des cellules suivantes

Passe arrière (backward)



$$\delta_s = g'(a_s)(y_s - \hat{y}_s)$$

$$w_{i,s}^{t+1} = w_{i,s}^{t} - \eta(t)\delta_s a_i$$

$$\delta_i = g'(a_i) \sum_{s \in \text{coucheSuiv.}} w_{i,s}\delta_s$$

$$w_{x_j,i}^{t+1} = w_{x_j,i}^{t} - \eta(t)\delta_i x_j$$

- δ_s : erreur en sortie
- δ_i : somme des erreurs provenant des cellules suivantes

La rétro-propagation de gradient

- 1. Evaluation de l'erreur L due à chaque connexion : $\frac{\partial L}{\partial w_{i,j}}$
  - calculer l'erreur sur la connexion $w_{i,j}$ en fonction de l'erreur après la cellule j

$$\frac{\partial L}{\partial w_{i,j}} = \frac{\partial L}{\partial a_j} \frac{\partial a_j}{\partial w_{i,j}} = \delta_j z_i$$

  - Pour les cellules de la couche de sortie :

$$\delta_k = \frac{\partial L}{\partial a_k} = g'(a_k)(y_k - \hat{y}_k)$$

  - Pour les cellules d'une couche cachée :

$$\delta_j = \frac{\partial L}{\partial a_j} = \sum_k \frac{\partial L}{\partial a_k} \frac{\partial a_k}{\partial a_j} = \sum_k \delta_k \frac{\partial a_k}{\partial z_j} \frac{\partial z_j}{\partial a_j} = g'(a_j) \cdot \sum_k w_{j,k} \delta_k$$

Plan

PyTorch et Tensorflow

PyTorch, c'est ...

- Un framework pour le développement et l'apprentissage de réseaux Deep sur CPU et GPU
- Une architecture modulaire de contenants et conteneurs pour la construction d'architecture flexible
- Un mécanisme de différenciation automatique : l'Autograd
- Une couche d'abstraction pour l'optimisation qui permet d'utiliser une variété de descentes de gradient
- Une gestion simplifiée des données pour la constitution des mini-batchs

PyTorch vs TensorFlow

- PyTorch plus récent, donc moins intégré dans l'industrie
- Déploiement, rapidité et processus industriel en faveur de TensorFlow
- Flexibilité, prototyping, simplicité en faveur de PyTorch

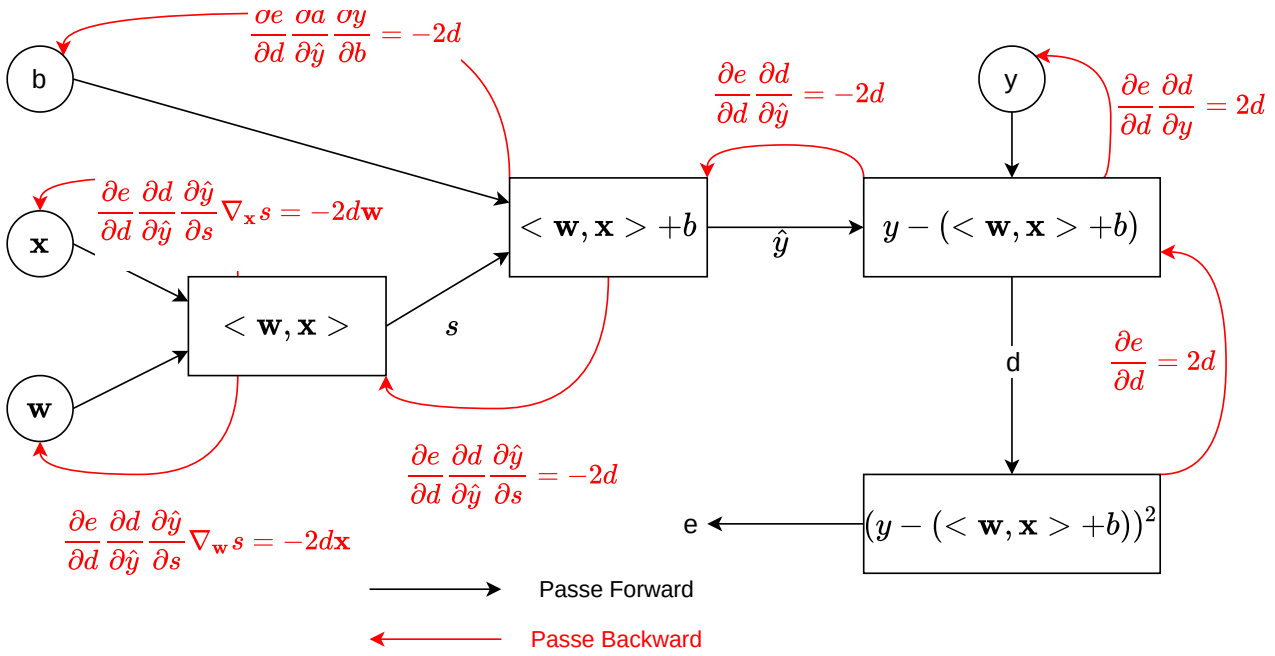
Les deux frameworks ont tendance à se rapprocher en termes de fonctionnalités ces derniers temps.

Objet de base en PyTorch : le Tenseur

```
# Création de tenseurs et caractéristiques
## Créer un tenseur à partir d'une liste
print(torch.tensor([[1.,2.,3.],[2.,3,4.]]))
## Créer un tenseur  tenseur rempli de 1 de taille 2x3x4
print(torch.ones(2,3,4))
## tenseur de zéros de taille 2x3 de type float
print(torch.zeros(2,3,dtype=torch.float))
## tirage uniforme entier entre 10 et 15,
## remarquez l'utilisation du _ dans random pour l'opération inplace
print(torch.zeros(2,3).random_(10,15))
## tirage suivant la loi normale
a=torch.zeros(2,3).normal_(1,0.1)
print(a)
## equivalent à zeros(3,4).normal_
b = torch.randn(3,4)
## Création d'un vecteur
c = torch.randn(3)
## concatenation de tenseurs
print(torch.cat((a,a),1))
## Taille des tenseurs/vecteurs
print(a.size(1),b.shape,c.size())
## Conversion de type
print(a.int(),a.int().type())
```

```
# Opérations élémentaires sur les tenseurs
## produit scalaire (et contrairement à numpy, que produit scalaire)
print(c.dot(c))
## produit matriciel : utilisation de @ ou de la fonction mm
print(a.mm(b), a @ b)
## transposé
print(a.t(),a.T)
## index du maximum selon une dimension
print("argmax : ",a.argmax(dim=1))
## somme selon une dimension/de tous les éléments
print(b.sum(1), b.sum())
## moyenne selon  une dimension/sur tous les éléments
print(b.mean(1), b.mean())
## changer la taille du tenseur (la taille totale doit être inchangée)
print(b.view(2,6))
## somme/produit/puissance termes a termes
print(a+a,a*a,a**2)
## attention ! comme sous numpy, il peut y avoir des pièges !
## Vérifier toujours les dimensions !!
a=torch.zeros(5,1)
b = torch.zeros(5)
## la première opération fait un broadcast et le résultat est tenseur à 2 dimensiosn,
## le résultat de la deuxième opération est bien un vecteur
print(a-b,a.t()-b)
```

Autograd et Graphe de calcul



Graphe de calcul

- Graphe orienté, décrit l'enchaînement des opérations de calcul
- Chaque source est une variable d'entrée, un seul nœud de sortie : le résultat du calcul
- En connaissant les dérivées de chaque opération, le graphe permet de calculer les gradient de la sortie par rapport à chaque variable d'entrée.

Autograd en PyTorch : Régression linéaire

- Nécessite d'avoir le flag requires_grad fixé à True lorsque l'on souhaite calculer le gradient par rapport à ce tenseur :

```
data_x, data_y  = ...
w = torch.randn(1,data_x.size(1),requires_grad=True)
b = torch.randn(1,1,requires_grad=True)
```

- On effectue le calcul

```
yhat = (x @ w.T)+b
loss_mse = ((yhat.view(-1,1)-data_y.view(-1,1))**2).sum()
loss_mse = loss_mse/data_x.size(0)
```

- Puis on exécute l'Autograd

```
loss_mse.backward()
```

- On obtient les gradients de loss_mse par rapport à w et b

```
print(w.grad, b.grad)
```

Les premiers pièges et quelques astuces

- Le graphe de calcul coûte très cher ! (en mémoire, en temps)
  - par défaut, un tenseur est créé sans le flag requires_grad
  - si un backward est effectué ⇒ message d'erreur
  - les gradients intermédiaires ne sont pas stockés ! (juste calculés et oubliés)
- Possibilité de désactiver temporairement l'autograd :

```
with torch.no_grad():
    # graphe de calcul désactivé
    w = ...
    z = ...
# graphe de calcul activé
y = ...
```

- Une fois le backward exécuté, on ne peut plus le refaire (une seule passe) !
- L'opération est cumulative ! La variable grad n'est pas remise à zéro, tout s'accumule (pourquoi ?)
- A retenir : z.backward() ⇒ dérivée partielle de z par rapport à tout ce qui a servi à la construire (et résultats dans la variable grad des variables en question).