MODULE 1 : INTRODUCTION DEEP LEARNING
PRISE EN MAIN PYTORCH

Agro–IODAA–Semestre 1

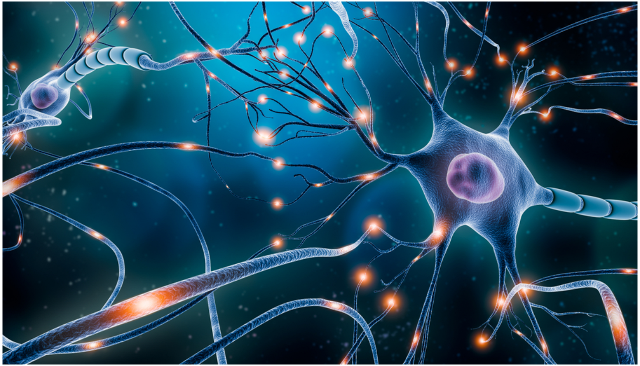
Vincent Guigue (Inspiré de N. Baskiotis & B. Piwowarski)

MIA PARIS-SACLAY EKINOCS

AGROPARISTECH Institut des Sciences et Industries du Vivant et de l'Environnement

Plan

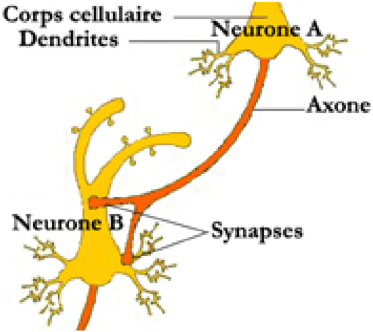
Inspiration biologique

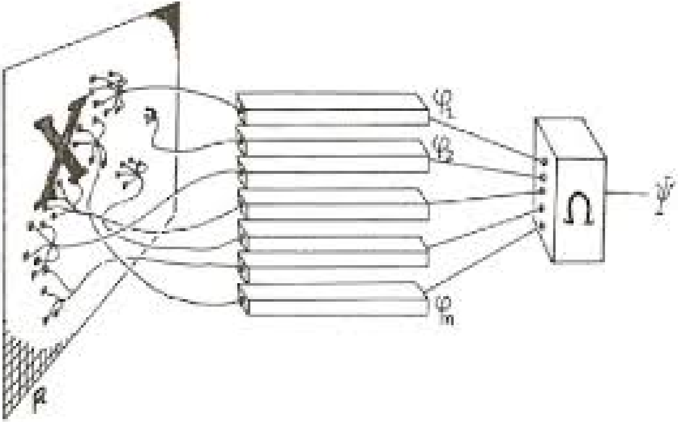


Réseau de neurones

- Opérateur complexe
- Logique d'activation et de fusion des messages
- Nom évocateur et vendeur

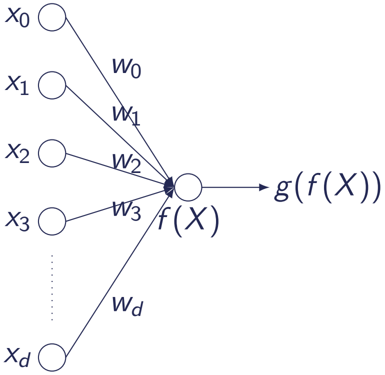
Inspiration biologique





- Feature
- Fusion de message = addition
- Activation = signe (=décision)

Les origines de l'apprentissage profond : le perceptron



Le perceptron

Sur un jeu de données $(\mathbf{x}, y) \in \mathbb{R}^d \times \{-1, 1\}$

- $f_\mathbf{w}(\mathbf{x}) = w_0 + \sum_{i=1}^d x_i w_i = w_0 + < \mathbf{x}, \mathbf{w} >$
- Fonction de décision : $g(x) = sign(x)$

$\rightarrow$ Sortie : $g(f(\mathbf{x})) = sign(< \mathbf{x}, \mathbf{w} >)$

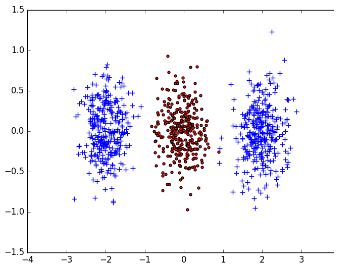
- Problème d'apprentissage : $argmax_\mathbf{w} \mathbb{E}_{x,y}[max(0, -y f_\mathbf{w}(\mathbf{x}))]$

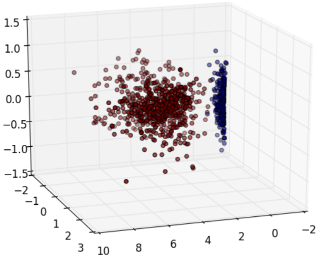
Algorithme du perceptron

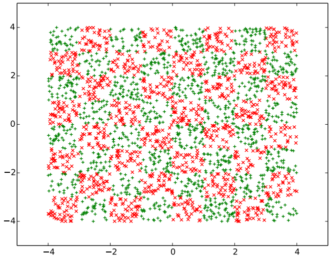
- Tant qu'il n'y a pas convergence :
  - pour tous les exemples (x^i, y^i) :
    - si $(y^i \times < \mathbf{w}.\mathbf{x}^i >) < 0$
      alors $\mathbf{w} = \mathbf{w} + \epsilon y^i \mathbf{x}^i$
- Descente de gradient sur le coût

Limites du perceptron ∼ gradient stochastique

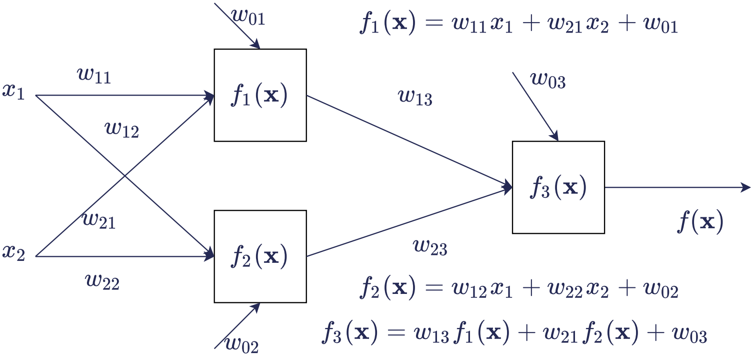
Est-il capable de séparer ces données ?







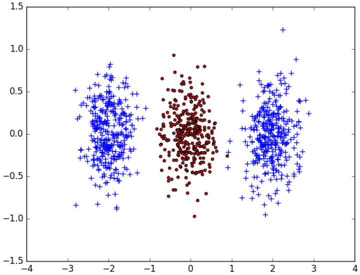
Combinons deux neurones



$$f_1(\mathbf{x}) = w_{11}x_1 + w_{21}x_2 + w_{01}$$

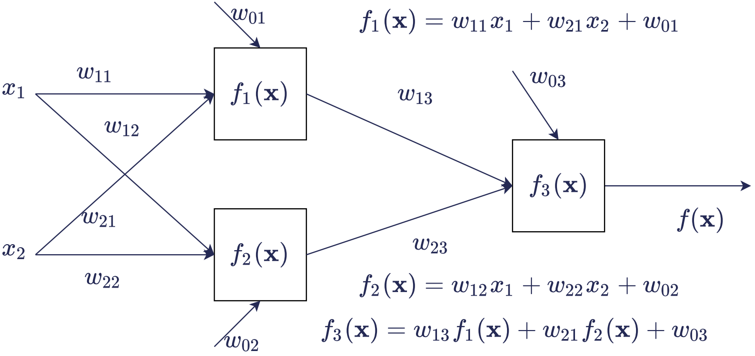
$$f_2(\mathbf{x}) = w_{12}x_1 + w_{22}x_2 + w_{02}$$

$$f_3(\mathbf{x}) = w_{13}f_1(\mathbf{x}) + w_{21}f_2(\mathbf{x}) + w_{03}$$



- Combiner des neurones $\Rightarrow$ suffisant ?

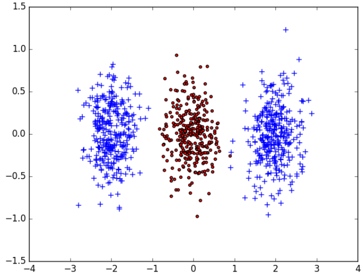
Combinons deux neurones



$$f_1(\mathbf{x}) = w_{11}x_1 + w_{21}x_2 + w_{01}$$

$$f_2(\mathbf{x}) = w_{12}x_1 + w_{22}x_2 + w_{02}$$

$$f_3(\mathbf{x}) = w_{13}f_1(\mathbf{x}) + w_{21}f_2(\mathbf{x}) + w_{03}$$



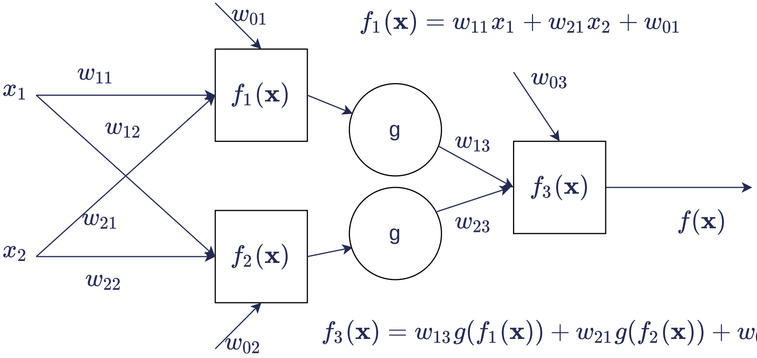
- Combiner des neurones $\Rightarrow$ suffisant ?

$$f(\mathbf{x}) = w_{03} + w_{13}\left(w_{01} + w_{11}x_1 + w_{21}x_2\right) + w_{23}\left(w_{02} + w_{12}x_1 + w_{22}x_2\right)$$

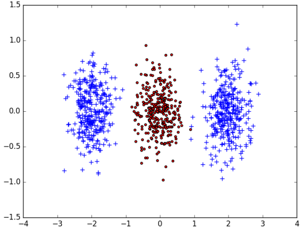
$$= w_{03} + w_{13}w_{01} + w_{23}w_{02} + x_1\left(w_{13}w_{11} + w_{23}w_{12}\right) + x_2\left(w_{11}w_{21} + w_{23}w_{22}\right)$$

Un pas vers les réseaux profonds

$$f_1(\mathbf{x}) = w_{11}x_1 + w_{21}x_2 + w_{01}$$

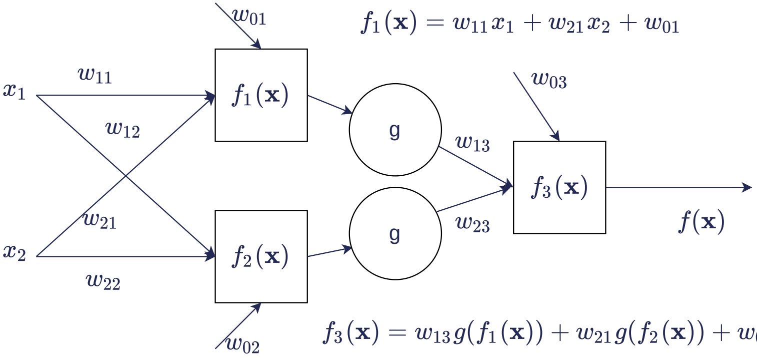


$$f_3(\mathbf{x}) = w_{13}\,g(f_1(\mathbf{x})) + w_{21}\,g(f_2(\mathbf{x})) + w_{03}$$



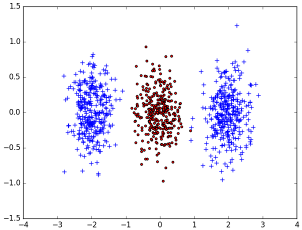
- Quelle non-linéarité ?
  - Fonction *signe* ?
  - ⇒ dérivée problématique ...
  - Fonctions *tanh*, *sigmoïde*, ... + biais

Un pas vers les réseaux profonds



$$f_1(\mathbf{x}) = w_{11}x_1 + w_{21}x_2 + w_{01}$$

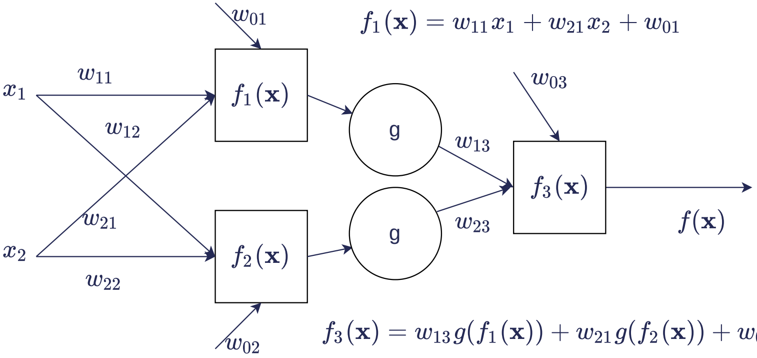
$$f_3(\mathbf{x}) = w_{13}\, g(f_1(\mathbf{x})) + w_{21}\, g(f_2(\mathbf{x})) + w_{03}$$



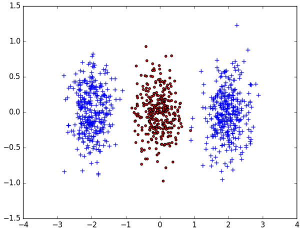
- Quelle non-linéarité ?
  - Fonction *signe* ?
  - $\Rightarrow$ dérivée problématique ...
  - Fonctions *tanh*, *sigmoïde*, ... + biais

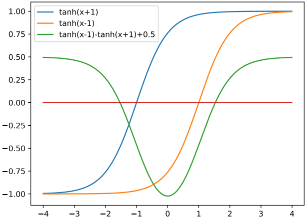
Un pas vers les réseaux profonds

$$f_1(\mathbf{x}) = w_{11}x_1 + w_{21}x_2 + w_{01}$$



$$f_3(\mathbf{x}) = w_{13}g(f_1(\mathbf{x})) + w_{21}g(f_2(\mathbf{x})) + w_{03}$$

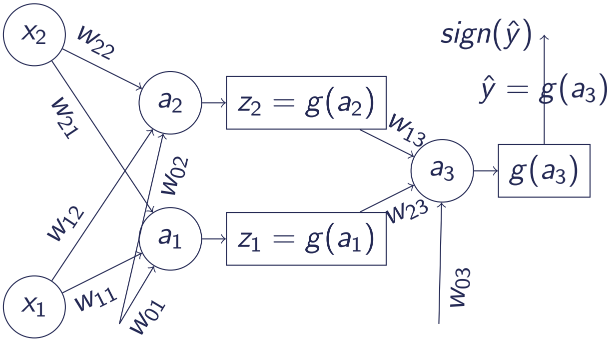




- Quelle non-linéarité ?
  - Fonction *signe* ?

  $\Rightarrow$ dérivée problématique ...
  - Fonctions *tanh*, *sigmoïde*, ... + biais

Pour l'inférence



Inférence

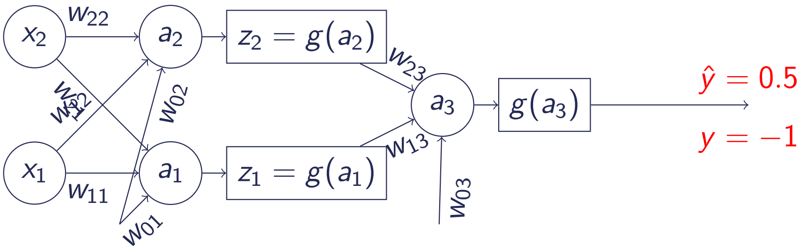
Avec $g(x) = tanh(x)$

- $a_1 = w_{01} + w_{11}x_1 + w_{21}x_2$
- $a_2 = w_{02} + w_{12}x_1 + w_{22}x_2$
- $z_1 = g(a_1)$
- $z_2 = g(a_2)$
- $a_3 = w_{03} + w_{13}z_1 + w_{23}z_2$
- $\hat{y} = g(a_3)$

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

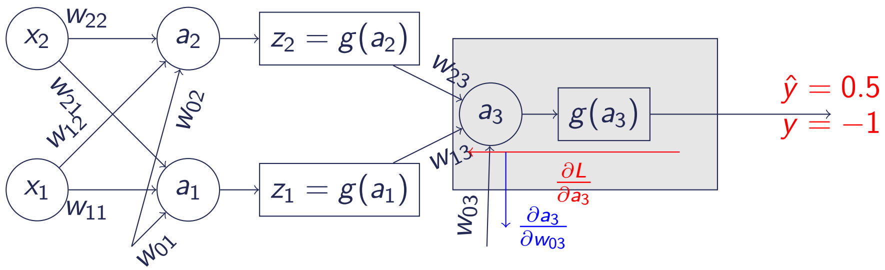
- Choix d'un coût : moindres carrés
  $L(\hat{y}, y) = (\hat{y} - y)^2$ (pourquoi est ce un bon choix ?)
- Mais à quel(s) neurone(s) et comment répartir l'erreur entre les poids ?

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

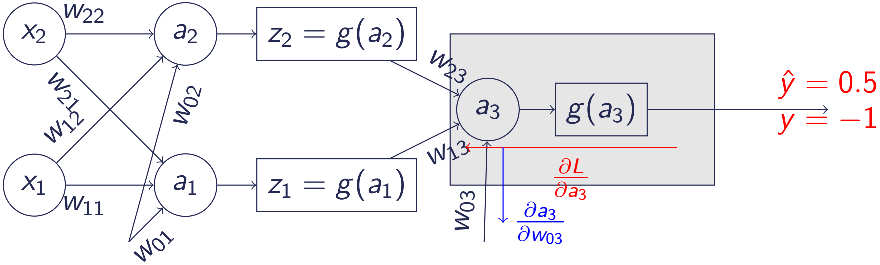
  - corriger un peu tous les poids . . . en estimant la part de chacun dans l'erreur
  - en commençant par la fin et en remontant progressivement

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

Pour l'apprentissage



Pour l'apprentissage



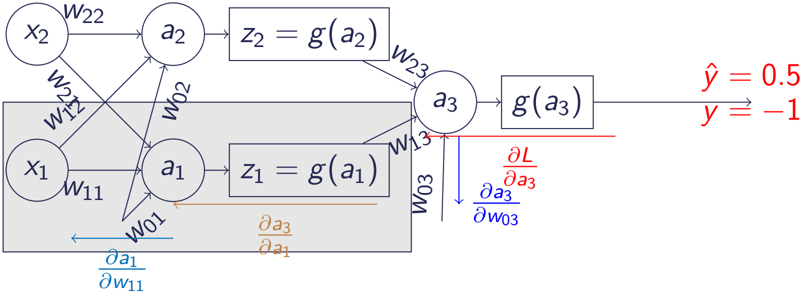
Pour les poids de la dernière couche : gradient $\nabla_{w_{i3}} L(\hat{y}, y)$

On a : $L(\hat{y}, y) = (g(a_3) - y)^2 = \left(g\left(w_{03} + w_{13}z_1 + w_{23}z_2\right) - y\right)^2$

$$\frac{\partial L}{\partial w_{i3}} = \frac{\partial L}{\partial a_3}\frac{\partial a_3}{\partial w_{i3}} \quad \text{avec} \left| \begin{array}{llll} \frac{\partial L}{\partial a_3} & = \frac{\partial L}{\partial g(a_3)}\frac{\partial g(a_3)}{\partial a_3} = \frac{\partial (g(a_3)-y)^2}{\partial a_3} & = 2g'(a_3)(g(a_3) - y) \\ \frac{\partial a_3}{\partial w_{i3}} & = \frac{\partial (w_{03}+w_{13}z_1+w_{23}z_2)}{\partial w_{i3}} & = z_i \end{array} \right.$$

Soit: $\frac{\partial L}{\partial w_{i3}} = 2g'(a_3)(\hat{y} - y)z_i$

Pour l'apprentissage



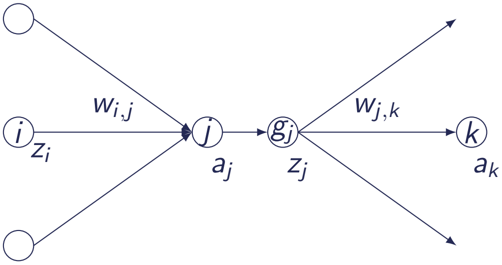
Pour les poids de la première couche: w_{i1} par exemple

$$\frac{\partial L}{\partial w_{i1}} = \frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial w_{i1}} \quad \text{avec} \quad \left| \begin{array}{lll} \frac{\partial L}{\partial a_1} & = \frac{\partial L}{\partial a_3}\frac{\partial a_3}{\partial a_1} & = \frac{\partial L}{\partial a_3} g'(a_1) w_{13} \\ \frac{\partial a_1}{\partial w_{i1}} & = \frac{\partial w_{01}+w_{11}x_1+w_{21}x_2}{\partial w_{i1}} & = x_i \end{array} \right.$$

Soit

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

Cas général dans les couches intermédiaires



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

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

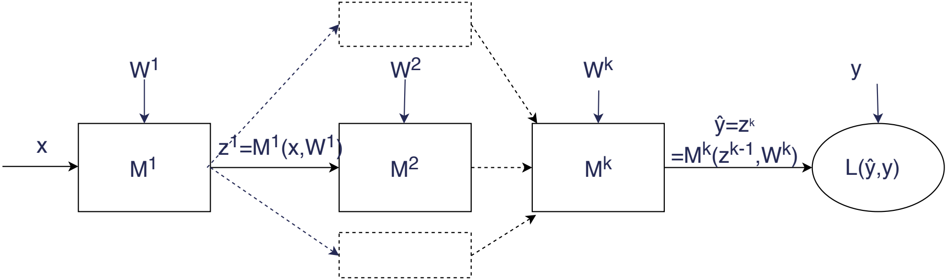
$$\underbrace{\frac{\partial L}{\partial a_j}}_{\text{erreur sur } j} = \sum_k (g'(a_k) w_{jk}) \underbrace{\frac{\partial L}{\partial a_k}}_{\text{erreur à propager}}$$

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

- 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

Un réseau : un assemblage de modules

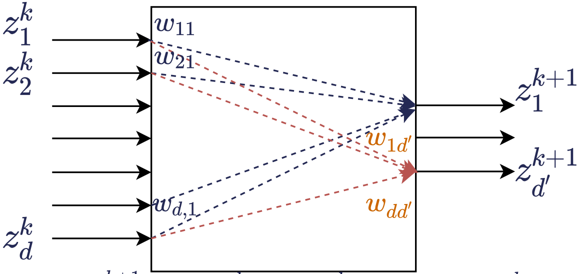


Un module M^k

- 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

$\mathbf{z}^k \in \mathbb{R}^d \qquad \mathbf{W} \in \mathbb{R}^{d \times d'} \qquad \mathbf{z}^{k+1} \in \mathbb{R}^{d'}$

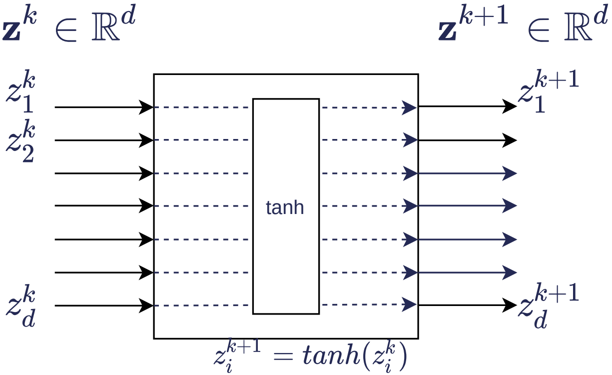


$$z_i^{k+1} = w_{1i}\, z_1^k + w_{2i}\, z_2^k + \ldots + w_{d,i}\, z_d^k$$
$$\mathbf{z}^{k+1} = \mathbf{W}\mathbf{z}^k$$

Couche linéaire [Linear]

Transformation paramétrée de $\mathbb{R}^d$ vers $\mathbb{R}^{d'}$
$M^k(z^{k-1}, W^k) = {W^k}^t z^{k-1}$ avec $W^k \in \mathbb{R}^d \times \mathbb{R}^{d'}$

Type usuel de modules

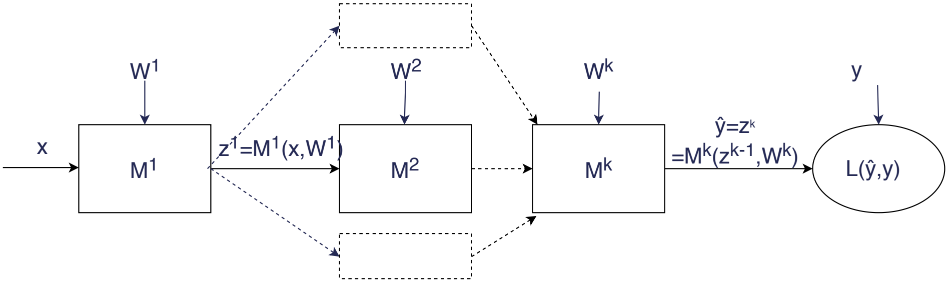


Module d'activation

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

$\Rightarrow$ tanh : $M^k(z^{k-1}, \emptyset) = tanh(z^{k-1}) = (tanh(z_1^{k-1}), tanh(z_2^{k-1}), \ldots, tanh(z_d^{k-1}))$

Type usuel de modules

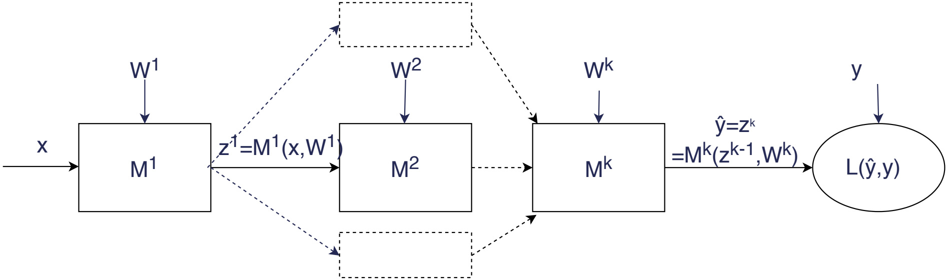


Un coût

Bloc final : deux entrées, la supervision et la sortie du réseau.

et d'autres composantes plus ésotériques

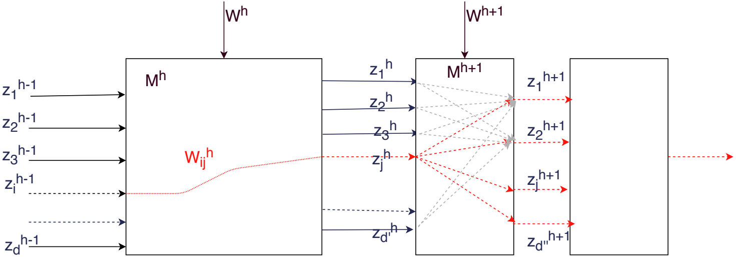
Apprentissage du réseau



Pour apprendre le réseau :

- Pour chaque module : $\nabla_{W^k} L(\hat{y}, y)$
- 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.

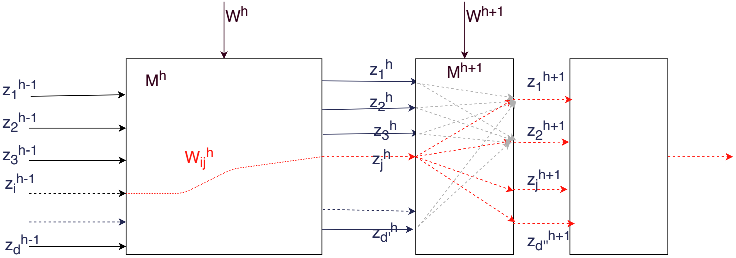
Zoom sur un module



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

- $\frac{\partial L}{\partial w_{ij}^h} = \sum_k \frac{\partial L}{\partial z_k^h}\frac{\partial z_k^h}{\partial w_{ij}^h} = \frac{\partial L}{\partial z_j^h}\frac{\partial z_j^h}{\partial w_{ij}^h} = \frac{\partial L}{\partial z_j^h}\frac{\partial M^h(\mathbf{z}^{h-1},W^h)}{\partial w_{ij}^h}$ (w_{ij}^h n'influe que sur z_j^h)
- $\frac{\partial L}{\partial z_j^h} = \sum_k \frac{\partial L}{\partial z_k^{h+1}}\frac{\partial z_k^{h+1}}{\partial z_j^h} = \sum_k \frac{\partial L}{\partial z_k^{h+1}}\frac{M^{h+1}(\mathbf{z}^h,W^h)}{\partial z_j^j}$
- On introduit $\delta_j^h = \frac{\partial L}{\partial z_j^h} = \sum_k \delta_k^{h+1}\frac{\partial M^{h+1}(\mathbf{z}^h,W^h)}{\partial z_j^j}$: $\frac{\partial L}{\partial w_{ij}^h} = \delta_j^h \frac{\partial M^h(\mathbf{z}^{h-1},W^h)}{\partial w_{ij}^h}$
- Dernière couche, $\delta_j^{last} = \frac{\partial L(\mathbf{z}^{last},y)}{\partial z_j^{last}}$, le gradient du coût wrt prédiction.

Zoom sur un module



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

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

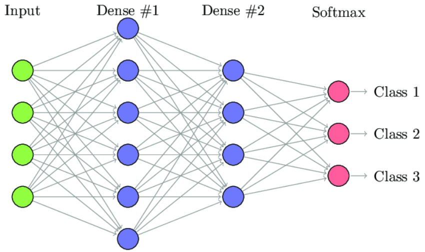
Pour chaque module, on a besoin :

- 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

Plan

Réseau Fully-Connected

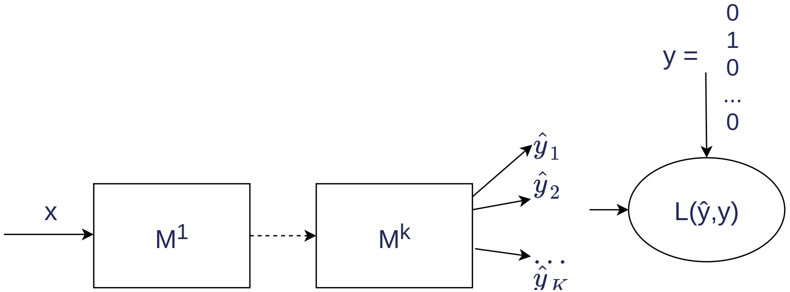
Un réseau *fully-connected* est une succession de couches linéaires et de fonctions d'activation



Propriétés

- Idéal pour les simples tâches de classification (multi-classes également)
- Architecture que l'on retrouve quasiment dans toutes les autres architectures
- Mais non adapté sur des entrées complexes (texte, image)
- Très sujet au sur-apprentissage avec l'augmentation du nombre de couches

Supervision Multi-classe



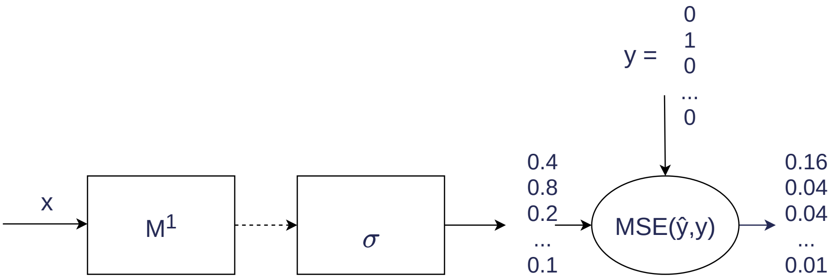
Quand il faut prédire K classes

- K sorties
- Utilisation de vecteurs 1-hot pour la supervision:

$$\mathbf{y} = (0, 0, \dots, 1, \dots, 0)$$

avec $y_i = 0$ pour i différent de la bonne classe,
$y_k = 1$ pour k l'indice de la bonne classe.

Utilisation de la MSE



Fonction de coût problématique

- Sortie du réseau entre 0 et 1 $\Rightarrow$ utilisation d'une sigmoïde
- Mais :
  - Similarité au vecteur de sortie $\Rightarrow$ pas critique $\Rightarrow$ **argmax ++ important**
  - ++ maximisation de la sortie de la bonne classe | – minimisation des autres sorties

Coût Cross-entropique

SoftMax : Sorties $\Rightarrow$ distribution + renforcement du max

$$\text{SoftMax}(z)_i = e^{z_i} / \left(\sum_{j=1}^{K} e^{z_j} \right), \qquad \sum_{i=1}^{K} \text{SoftMax}(z)_i = 1$$

Coût Cross-entropique

- $CE(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{K} y_i \log(\hat{y}_i)$
- Dans le cas où $\mathbf{y}$ est un vecteur one-hot de la classe k :
  $CE(\mathbf{y}, \hat{\mathbf{y}}) = -\log(\hat{y}_k)$
- Combinaison SoftMax et Cross-entropie :

$$CE(\mathbf{y}, SoftMax(z)) = -z_k + \log\left(\sum_{j=1}^{K} e^{z_j} \right)$$

$$\frac{\partial CE(\mathbf{y}, SoftMax(z))_i}{\partial z_i} = Softmax(z)_i - 1_{i=k}$$

Coût Cross-entropique

SoftMax : Sorties $\Rightarrow$ distribution + renforcement du max

$$\text{SoftMax}(z)_i = e^{z_i} / \left(\sum_{j=1}^{K} e^{z_j} \right), \qquad \sum_{i=1}^{K} \text{SoftMax}(z)_i = 1$$

Cross-entropie binaire

Pour le **multi-label** en particulier, cross-entropie sur chaque sortie (considérée comme des Bernoulli indépendantes):

$$BCE(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_{i=1}^{K} y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)$$

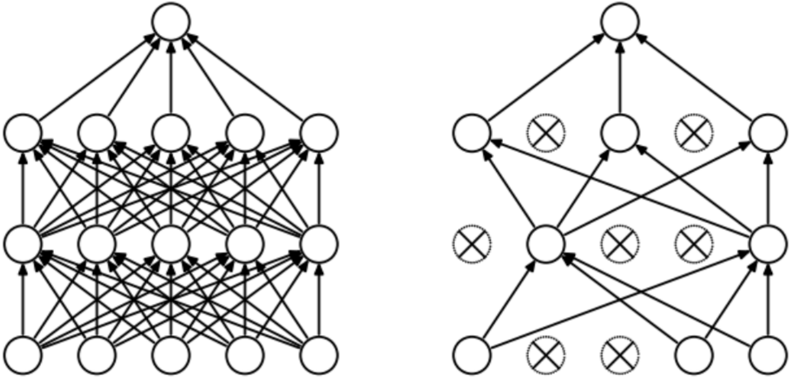
Lutter contre le sur-apprentissage

Différentes techniques qui visent toutes à régulariser le réseau

- Régularisation des couches (l1, l2) : ajout d'un terme de pénalisation en $||W||^p$ sur les poids des couches
- Dropout : retirer pendant une itération quelques neurones au hasard dans le réseau; permet d'augmenter la robustesse du réseau
- Augmented Data : perturbation des données d'entrées pour améliorer la généralisation
- Gradient Clipping : la norme du gradient rétro-propagé est bornée maximalement pour éviter une trop grosse instabilité

Lutter contre le sur-apprentissage

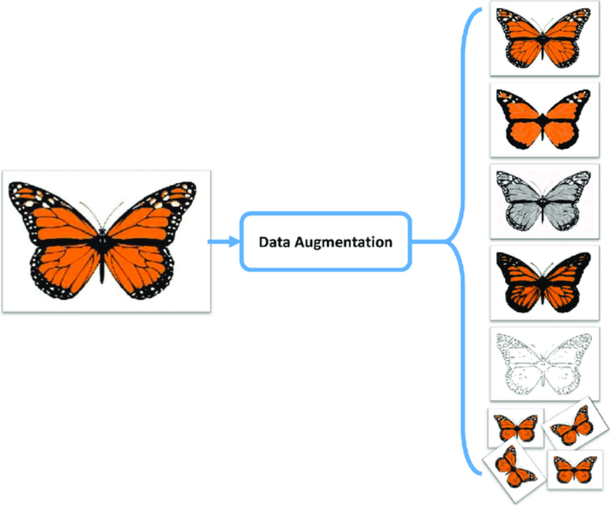
Drop out:



(a) Standard Neural Net (b) After applying dropout.

Lutter contre le sur-apprentissage

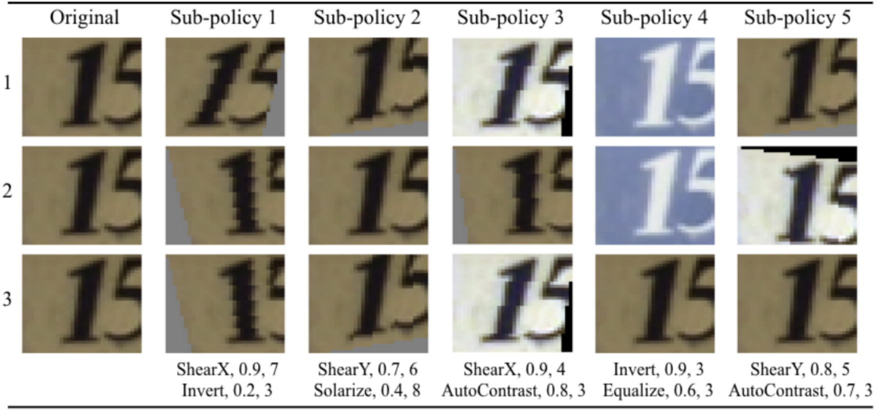
Data augmentation: une idée simple pour régulariser par la masse et les variations



Lutter contre le sur-apprentissage

Data augmentation: comment automatiser le processus?

⇒ outils paramétrables et disponibles dans `torchvision`

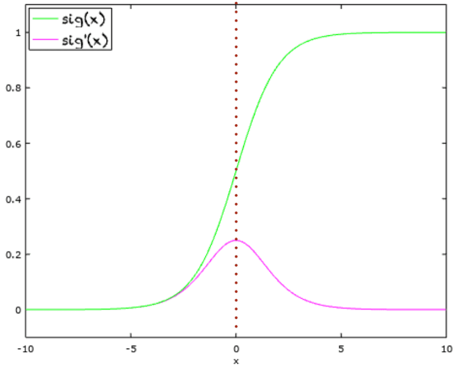


Amélioration du gradient

Gradient vanishing

Le gradient tend à disparaitre:

- Dans les couches éloignées de la supervision
- Dans les sigmoides saturées



Plot of σ(x) and its derivate σ′(x)

Domain: (−∞, +∞)
Range: (0, +1)
σ(0) = 0.5

Other properties

$$\sigma(x) = 1 - \sigma(-x)$$

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

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

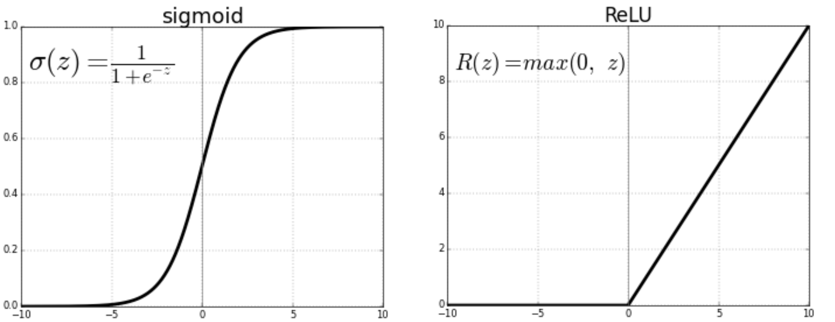
Amélioration du gradient

Gradient vanishing

Le gradient tend à disparaitre:

- Dans les couches éloignées de la supervision
- Dans les sigmoides saturées

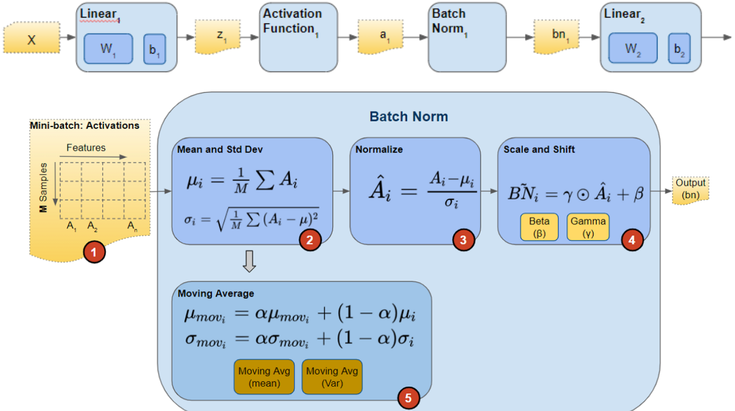
Fonction d'activation spécifique $ReLU(x) = max(0, x)$: permet de garder un gradient fort lorsque le neurone est activé



Amélioration du gradient

Topologie de l'espace de recherche & gradient

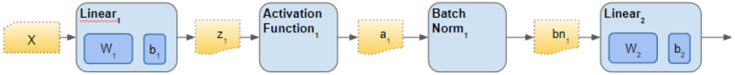
- BatchNorm : pour une couche, centrée/normée chaque sortie (estimation sur chaque mini-batch)
- LayerNorm : à la sortie d'une couche, normalisation de chaque exemple séparément de ses dimensions



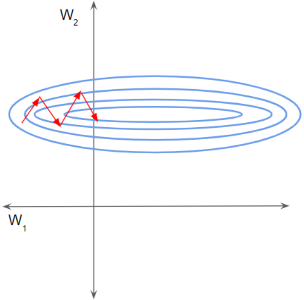
Amélioration du gradient

Topologie de l'espace de recherche & gradient

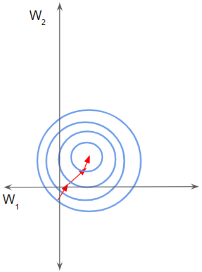
- BatchNorm : pour une couche, centrée/normée chaque sortie (estimation sur chaque mini-batch)
- LayerNorm : à la sortie d'une couche, normalisation de chaque exemple séparément de ses dimensions



Centrer les données = meilleure topologie pour l'apprentissage



$\Rightarrow$



Multiplication des hyper-paramètres

- Architecture = beaucoup d'hyperparamètres
- Besoin de normalisation pour:
  - Mieux comparer les architectures
  - Avoir des a priori sur les bons paramètres

Plan

Ce que nous verrons dans ce module (Travaux Pratiques)

PyTorch, c'est ...

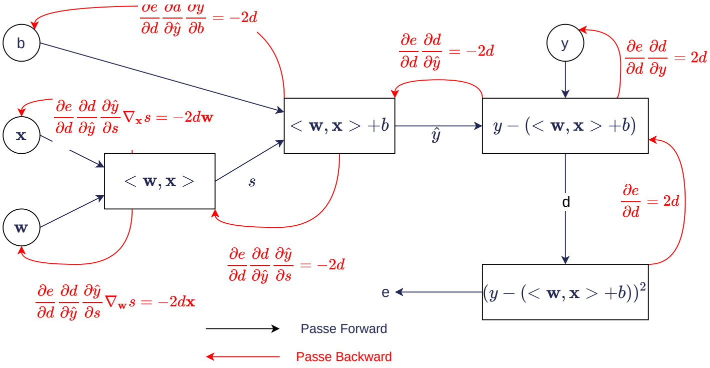
- Framework de dévelop. + apprentissage de réseaux Deep sur CPU et GPU
- Architecture modulaire de contenants + conteneurs ⇒ Architectures flexibles
- Différenciation automatique ⇒ **Autograd**
- Couche d'abstraction pour l'optimisation ⇒ variété de descentes de gradient
- Gestion simplifiée des données 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.

Autograd et Graphe de calcul



Graphe de calcul

- Graphe orienté, décrit l'enchaînement des opérations de calcul
- *Source* = variable d'entrée + 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.