

MODULE 1 : INTRODUCTION DEEP LEARNING PRISE EN MAIN PYTORCH

Agro-IODAA–Semestre 1



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

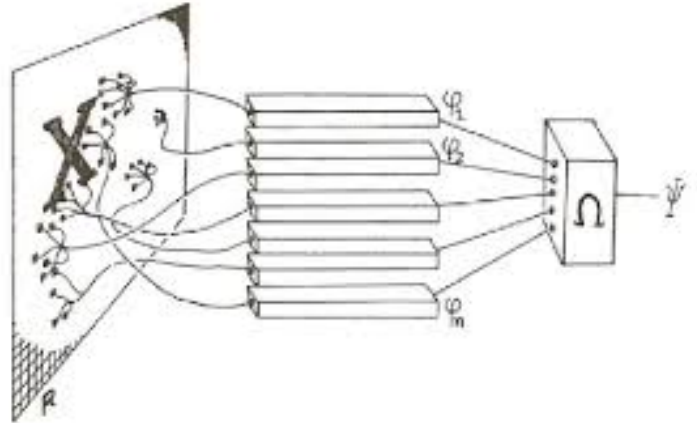
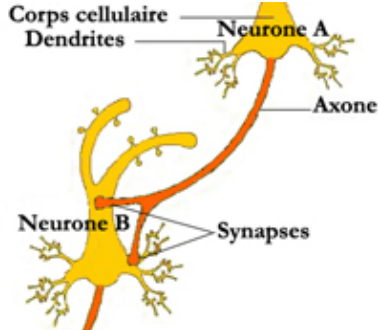


INTRODUCTION AU DEEP LEARNING

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



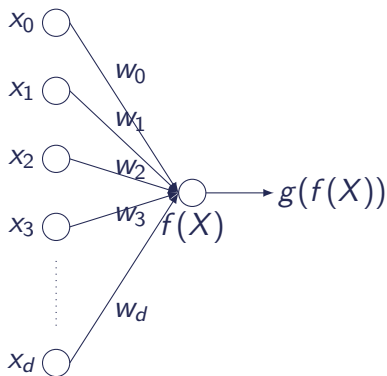
Inspiration biologique [plus ou moins lointaine]



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



Les origines de l'apprentissage profond : le perceptron



Le perceptron

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

- $f_{\mathbf{w}}(\mathbf{x}) = w_0 + \sum_{i=1}^d x_i w_i = w_0 + \langle \mathbf{x}, \mathbf{w} \rangle$

- Fonction de décision : $g(x) = \text{sign}(x)$

→ Sortie : $g(f(\mathbf{x})) = \text{sign}(\langle \mathbf{x}, \mathbf{w} \rangle)$

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

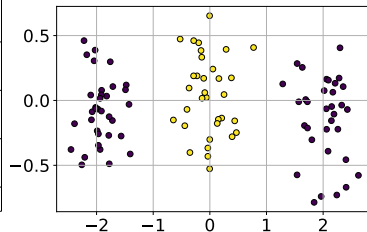
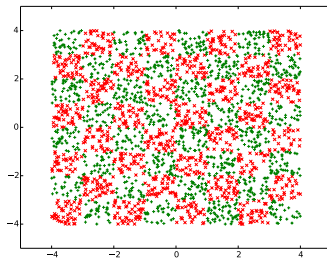
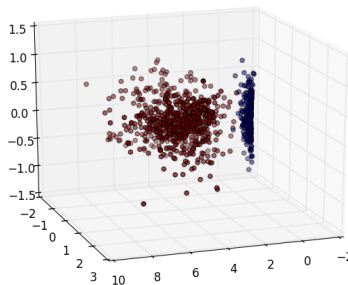
Algorithme du perceptron

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



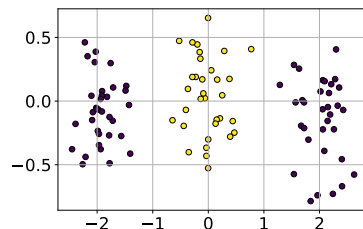
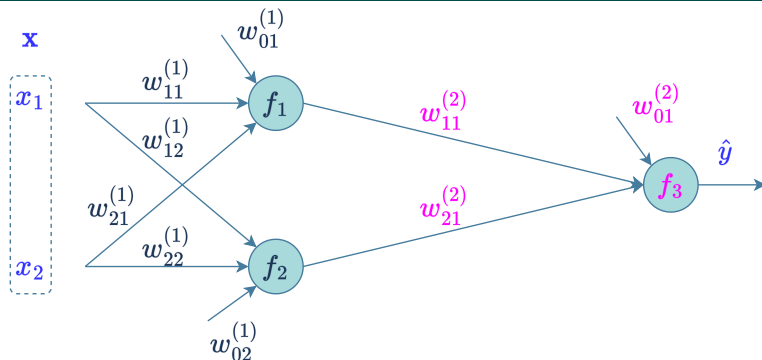
Limites du perceptron

Est-il capable de séparer ces données ?





Combinons deux neurones

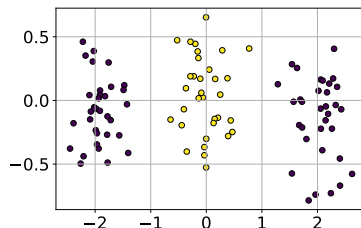
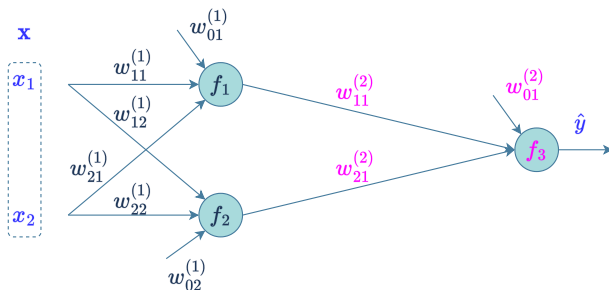


$$f_1(\mathbf{x}) = w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2 + w_{01}^{(1)}, \quad f_2(\mathbf{x}) = w_{12}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{02}^{(1)}$$

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

Combiner des neurones \Rightarrow suffisant ?

Combinons deux neurones



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

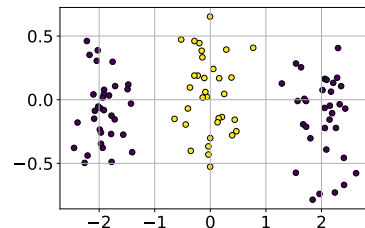
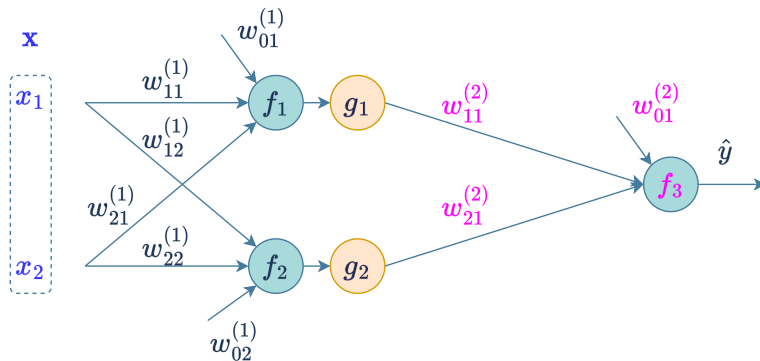
$$f_3(\mathbf{x}) = w_{11}^{(2)}f_1(\mathbf{x}) + w_{21}^{(2)}f_2(\mathbf{x}) + w_{01}^{(2)}$$

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

Non ! il faut introduire de la non linéarité, sinon équivalent à un perceptron ...



Non-linéarité



■ Quelle non-linéarité ?

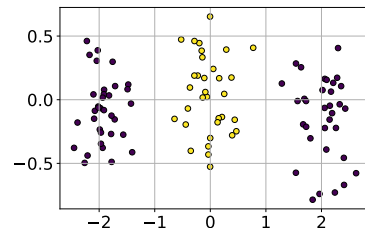
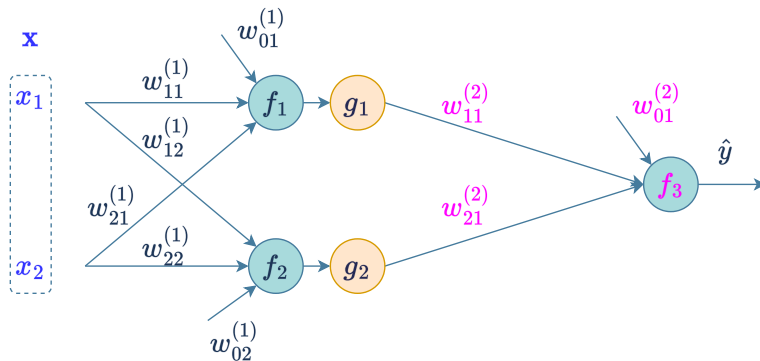
- Fonction *signe* ?

⇒ dérivée problématique ...

- Fonctions *tanh*, *sigmoïde*, ... + biais



Non-linéarité



■ Quelle non-linéarité ?

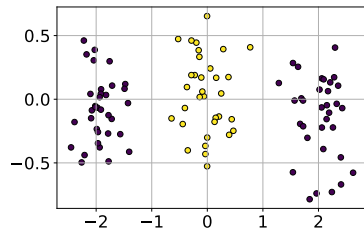
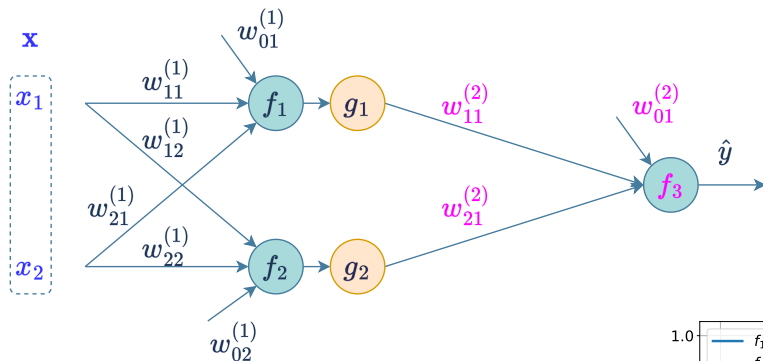
■ Fonction *signe* ?

⇒ dérivée problématique ...

■ Fonctions *tanh*, *sigmoïde*, ... + biais



Non-linéarité



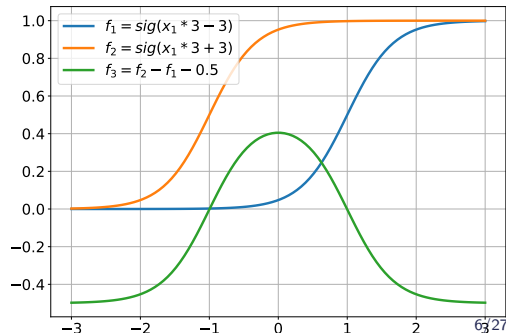
■ Quelle non-linéarité ?

■ Fonction *signe* ?

⇒ dérivée problématique ...

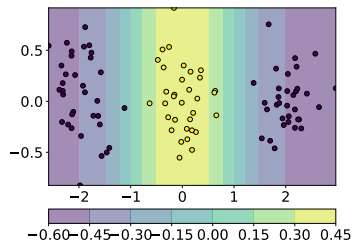
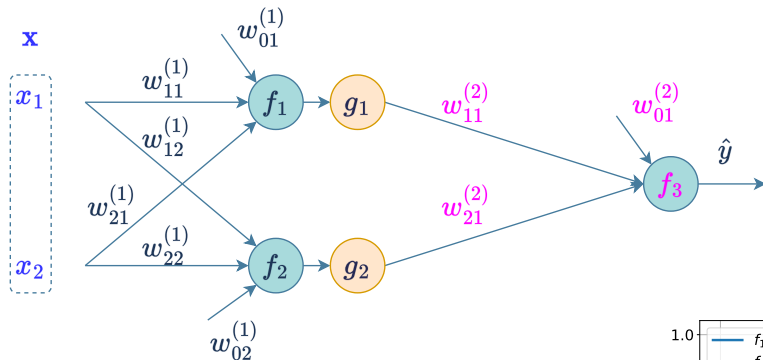
■ Fonctions *tanh*, *sigmoïde*, ... + biais

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





Non-linéarité



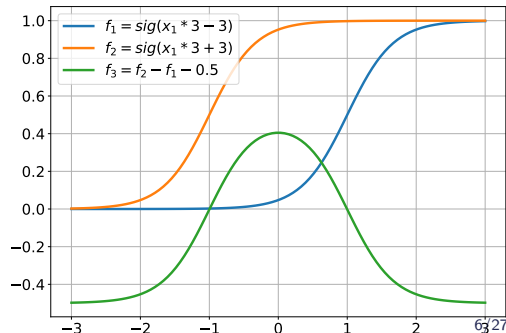
■ Quelle non-linéarité ?

■ Fonction *signe* ?

⇒ dérivée problématique ...

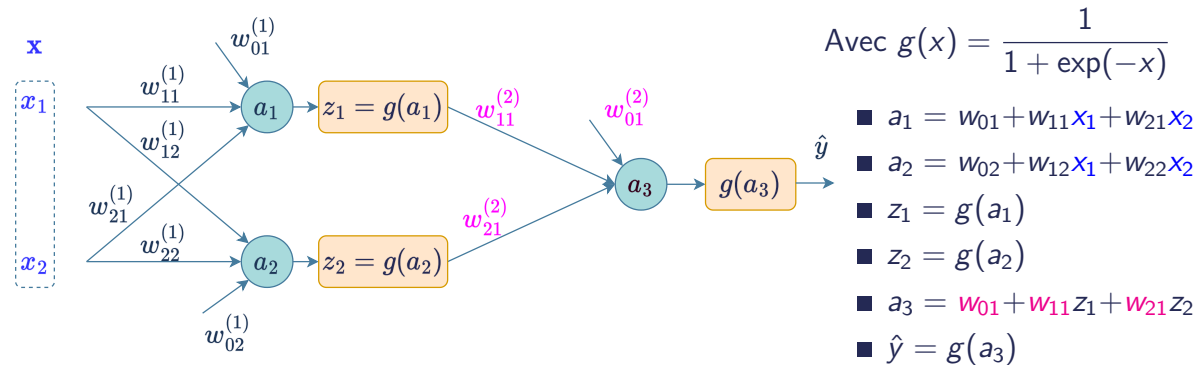
■ Fonctions *tanh*, *sigmoïde*, ... + biais

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





Vocabulaire de l'inférence

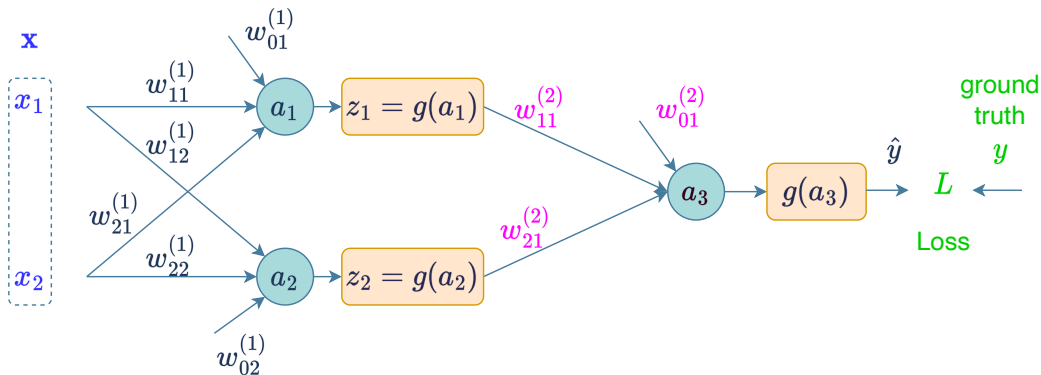


Vocabulaire

- Inférence : *pas 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).



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 comment répartir l'erreur entre les poids ?

⇒ Rétro-propagation de l'erreur



Descente de gradient

Objectif: calculer les gradients partiels par rapport aux paramètres

$$\forall i, j, \quad \frac{\partial L(\hat{y}, y)}{\partial w_{ij}}$$

Forward: calcul de \hat{y}

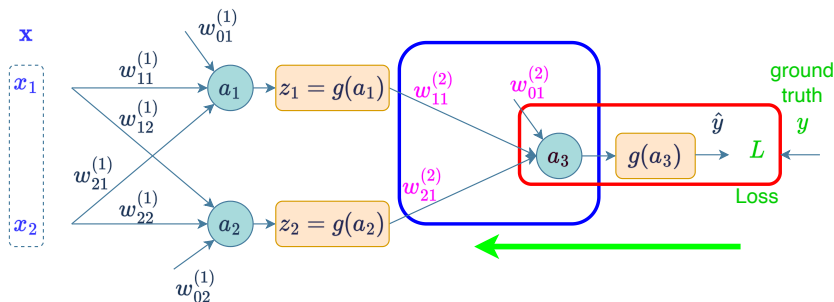
[entre autres]

Backward: calcul des gradients

Optimisation: descente de gradient

$$w_{ij} \leftarrow w_{ij} - \underbrace{\varepsilon}_{\text{Learning rate}} \frac{\partial L(\hat{y}, y)}{\partial w_{ij}}$$

Calcul du gradient: chain rule



Forward:

$$\hat{y} = 0.5$$

$$y = -1$$

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

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

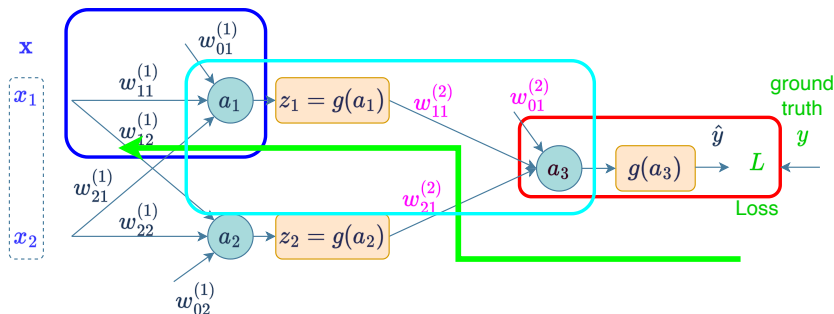
$$\frac{\partial L}{\partial w_{i1}^{(2)}} = \frac{\partial L}{\partial a_3} \frac{\partial a_3}{\partial w_{i1}^{(2)}} \quad \text{avec} \quad \left| \begin{array}{l} \frac{\partial L}{\partial a_3} \\ \frac{\partial a_3}{\partial w_{i1}^{(2)}} \end{array} \right| = \frac{\partial L}{\partial g(a_3)} \frac{\partial g(a_3)}{\partial a_3} = \frac{\partial (g(a_3) - y)^2}{\partial a_3} = 2g'(a_3)(g(a_3) - y)$$

$$= \frac{\partial (w_{01}^{(2)} + w_{11}^{(2)} z_1 + w_{12}^{(2)} z_2)}{\partial w_{i1}^{(2)}} = z_i$$

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

Calcul du gradient: chain rule

[suite]



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

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

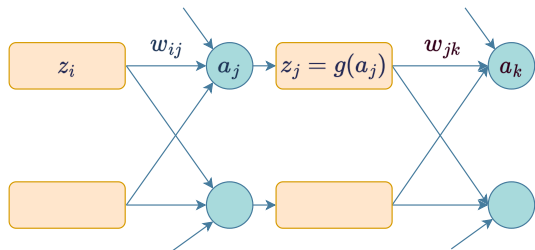
$$\frac{\partial L}{\partial w_{i1}} = \frac{\partial L}{\partial a_1} \frac{\partial a_1}{\partial w_{i1}} \quad \text{avec} \quad \left| \begin{array}{l} \frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial a_3} \frac{\partial a_3}{\partial a_1} = \frac{\partial L}{\partial a_3} g'(a_1) w_{11}^{(2)}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

Soit:

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



Cas général dans les couches intermédiaires



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

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

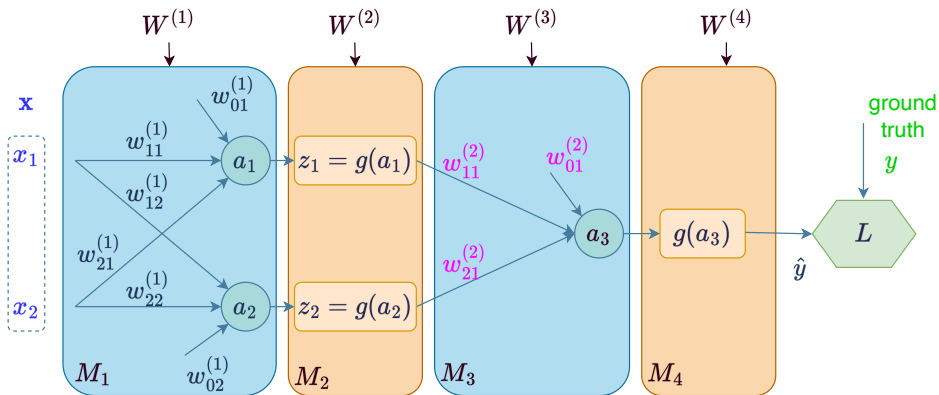
$$\underbrace{\frac{\partial L}{\partial a_j}}_{\text{erreur sur } j} = \sum_k (g'(a_k) w_{jk}) \underbrace{\frac{\partial L}{\partial a_k}}_{\text{erreur à propager}}$$

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

- 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**

ARCHITECTURE MODULAIRE

Réseau : assemblage de modules

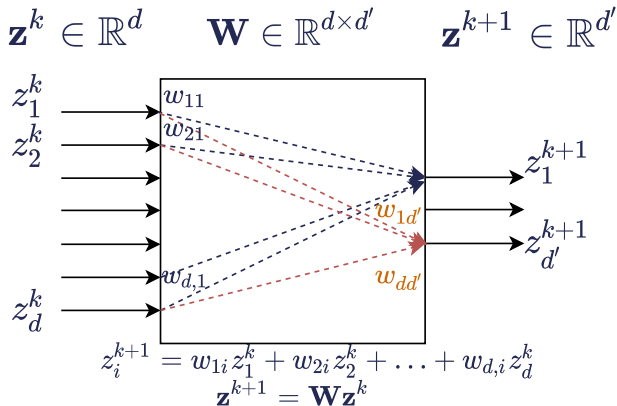


Un module M^k

- 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



Couche linéaire

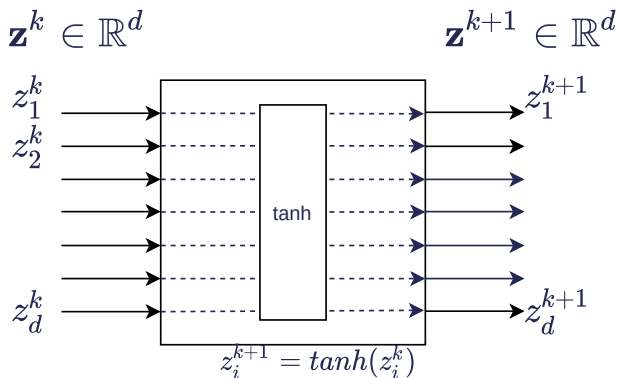
[Linear]

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

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



Type usuel de modules



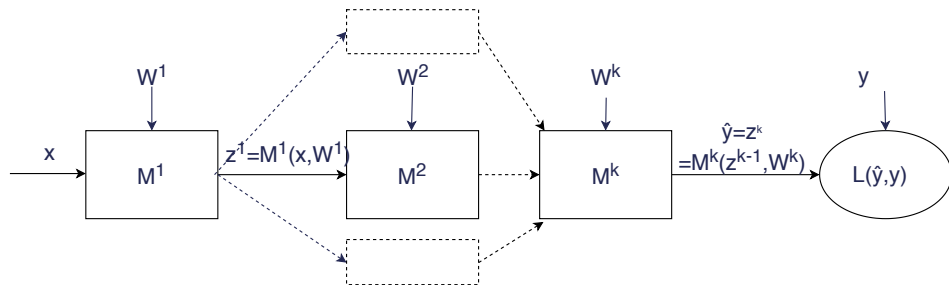
Module d'activation

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

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



Type usuel de modules



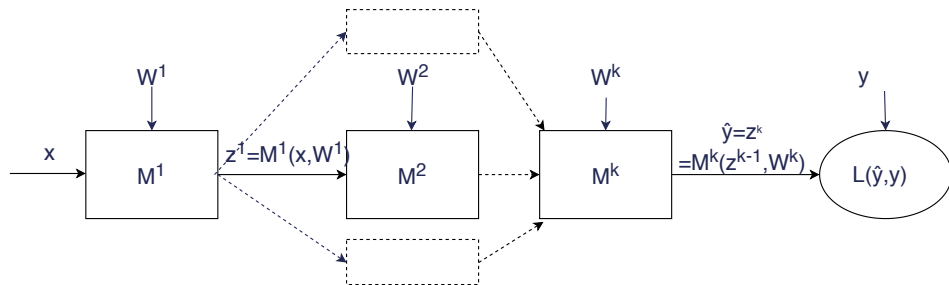
Un coût

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

et d'autres composantes plus ésoétriques



Apprentissage du réseau

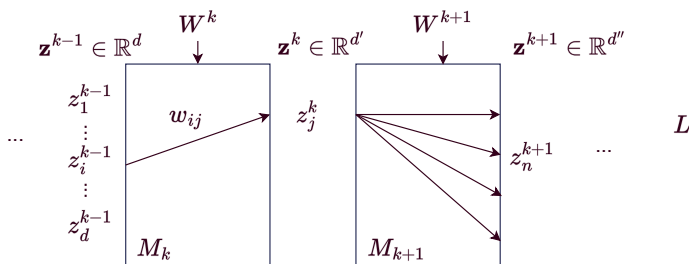


Pour apprendre le réseau :

- 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 le module k

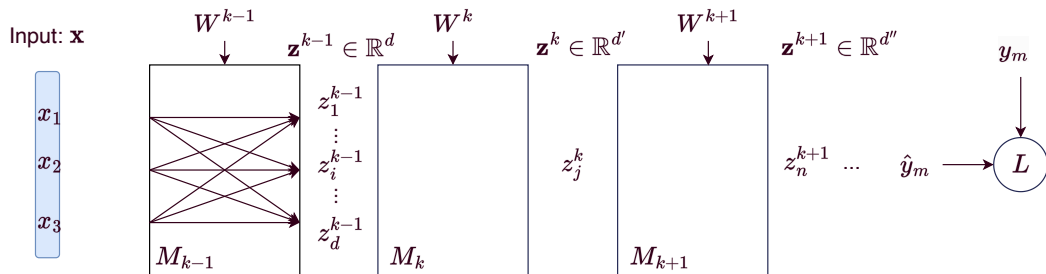


Rétro-propagation pour M^k , $\mathbf{z}^k = M(\mathbf{z}^{k-1}, W^k)$

- $\frac{\partial L}{\partial w_{ij}^k} = \sum_j \frac{\partial L}{\partial z_j^k} \frac{\partial z_j^k}{\partial w_{ij}^k} = \frac{\partial L}{\partial z_j^k} \frac{\partial z_j^k}{\partial w_{ij}^k} = \frac{\partial L}{\partial z_j^k} \frac{\partial M^k(\mathbf{z}^{k-1}, W^k)}{\partial w_{ij}^k}$ [w_{ij}^k n'impacte que z_j^k]
- $\frac{\partial L}{\partial z_j^k} = \sum_n \frac{\partial L}{\partial z_n^{k+1}} \frac{\partial z_n^{k+1}}{\partial z_j^k} = \sum_n \frac{\partial L}{\partial z_n^{k+1}} \frac{M^{k+1}(\mathbf{z}^k, W^{k+1})}{\partial z_j^k}$ [w_{ij}^k impacte tous les z_n^{k+1}]
- On introduit $\delta_j^k = \frac{\partial L}{\partial z_j^k} = \sum_n \delta_n^{k+1} \frac{\partial M^{k+1}(\mathbf{z}^k, W^{k+1})}{\partial z_j^k}$: $\frac{\partial L}{\partial w_{ij}^k} = \delta_j^k \frac{\partial M^k(\mathbf{z}^{k-1}, W^k)}{\partial w_{ij}^k}$
- Dernière couche, $\delta_j^{\text{end}} = \frac{\partial L(\mathbf{z}^{\text{end}}, y)}{\partial z_j^{\text{end}}}$, le gradient du coût wrt prédiction.



Zoom sur le module k : *forward* / *backward*



- $\mathbf{z}^1 = M_1(\mathbf{x}, W^1)$

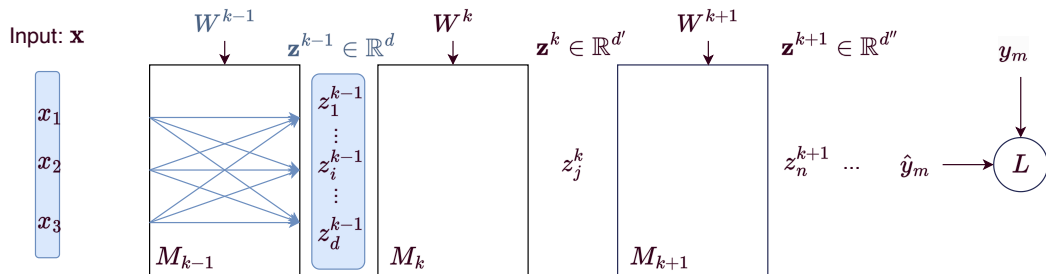
- $\mathbf{z}^k = M_k(\mathbf{z}_{k-1}, W^k)$

+ Stockage des \mathbf{z}^k

- Jusqu'à $\hat{\mathbf{y}}$



Zoom sur le module k : *forward* / *backward*



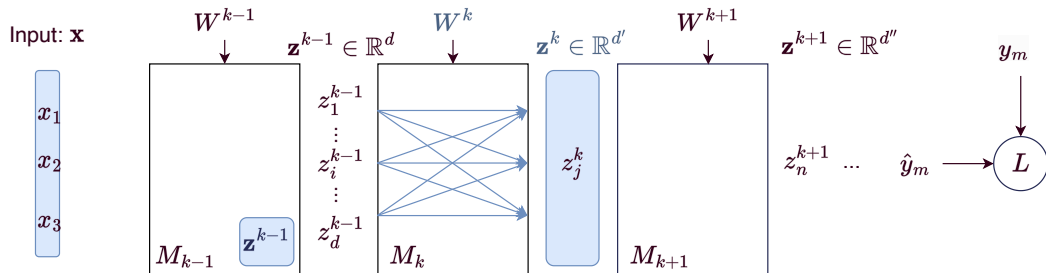
- $\mathbf{z}^1 = M_1(\mathbf{x}, W^1)$

- $\mathbf{z}^k = M_k(\mathbf{z}_{k-1}, W^k)$

+ Stockage des \mathbf{z}^k

- Jusqu'à $\hat{\mathbf{y}}$

Zoom sur le module k : *forward* / *backward*



- $\mathbf{z}^1 = M_1(\mathbf{x}, W^1)$

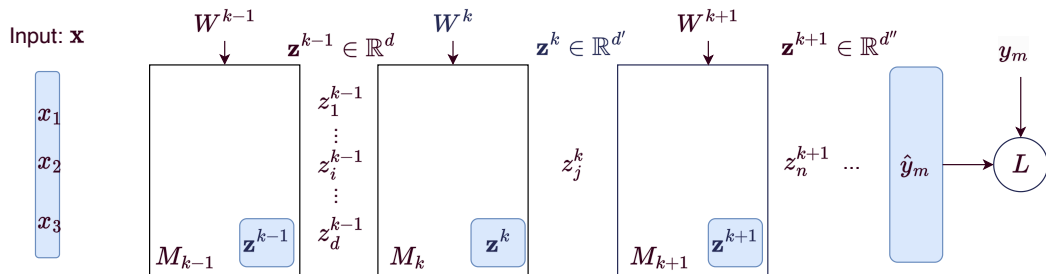
- $\mathbf{z}^k = M_k(\mathbf{z}_{k-1}, W^k)$

+ Stockage des \mathbf{z}^k

- Jusqu'à $\hat{\mathbf{y}}$



Zoom sur le module k : *forward* / *backward*



- $\mathbf{z}^1 = M_1(\mathbf{x}, W^1)$

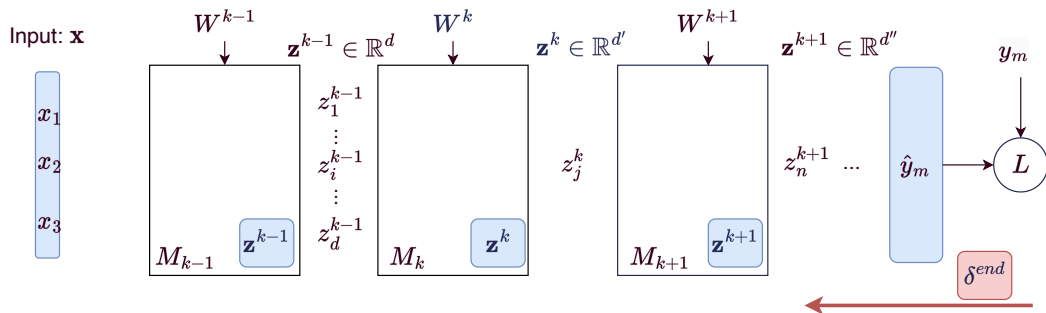
- $\mathbf{z}^k = M_k(\mathbf{z}_{k-1}, W^k)$

+ Stockage des \mathbf{z}^k

- Jusqu'à $\hat{\mathbf{y}}$



Zoom sur le module k : *forward* / *backward*

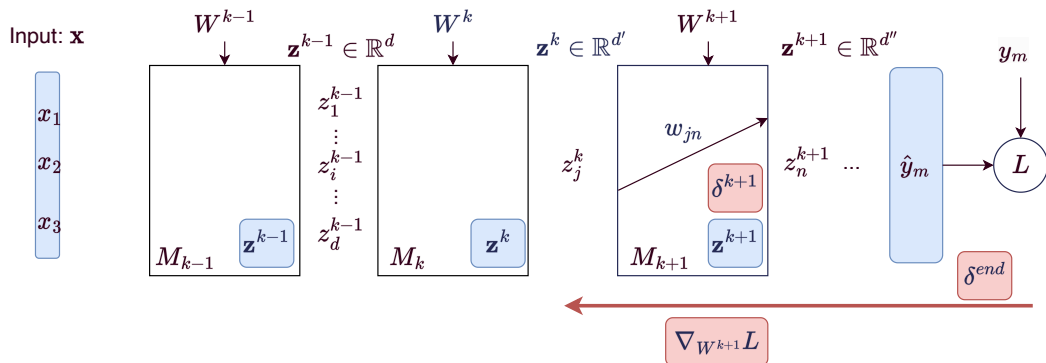


$$\hat{\mathbf{y}}_m \equiv \mathbf{z}^{end}, \quad \delta_j^{end} = \frac{\partial L(\mathbf{z}^{end}, y)}{\partial z_n^{end}} = \frac{2}{N} (z_n^{end} - y_n)$$

Dans le cas de la MSE



Zoom sur le module k : *forward* / *backward*



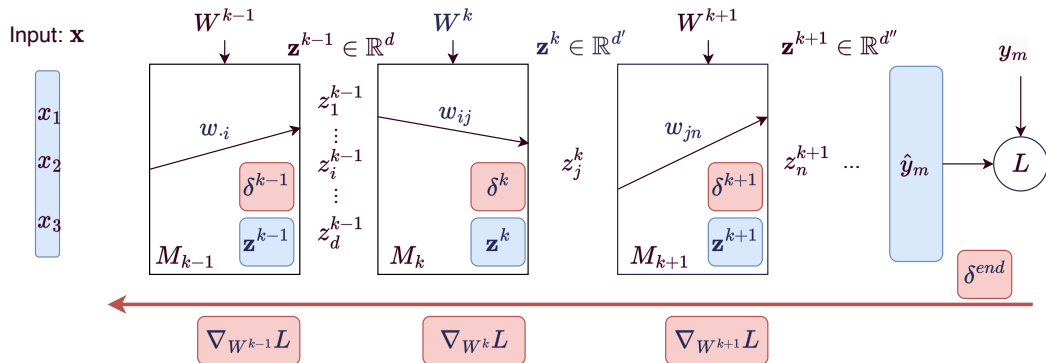
Supposons que $\mathbf{z}^{k+1} = \hat{\mathbf{y}}_m = \mathbf{z}^{end} \Rightarrow \delta_n^{k+1} = \delta_n^{end} = \frac{2}{N}(\mathbf{z}_n^{end} - y_n)$

Gradient: $\frac{\partial L}{\partial w_{jn}^{k+1}} = \delta_n^{end} \frac{\partial z_n^{end}}{\partial w_{jn}^{k+1}} = \delta_n^{end} \frac{\partial M^{k+1}(\mathbf{z}^k, W^{k+1})}{\partial w_{jn}^{k+1}} = \delta_n^{end} z_j^k$

Dans le cas d'un module linéaire



Zoom sur le module k : *forward* / *backward*



$$\delta_j^k = \frac{\partial L}{\partial z_j^k} = \sum_n \frac{\partial L}{\partial z_n^{k+1}} \frac{\partial z_n^{k+1}}{\partial z_j^k} = \sum_n \delta_n^{end} \frac{\partial M^{k+1}(\mathbf{z}^k, W^{k+1})}{\partial z_j^k} = \sum_n \delta_n^{end} w_{jn}^{k+1}$$

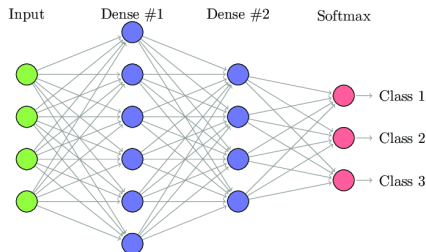
Dans le cas d'un module linéaire

PREMIER RÉSEAU DE NEURONES



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



Architecture logicielle

Récupération de modules existants pour une construction rapide

Création: $M = \text{torch.nn.Linear}(\text{args})$ Inférence: $z = M(x)$

Réseau à une couche linéaire: (=décision linéaire simple):

```
1 Xdim = housing_x.size(1)
2 ## Creation d'une couche lineaire de dimension Xdim->1
3 net = torch.nn.Linear(Xdim, 1) # recuperation d'un module
4 ## Fonction de cout
5 mseloss = torch.nn.MSELoss()
6 ## Optimiseur (& recuperation des parametres)
7 optim = torch.optim.SGD(params=net.parameters(), lr=EPS)
8
9 yhat = net(housing_x) # inference des modules = appel direct
10 loss = mseloss(net(housing_x).view(-1,1), housing_y.view(-1,1))
11 loss.backward()
12 optim.step()
```



Un réseau de neurones est un module

Module

- définition des couches
+ inférence (=forward, définition des entrées/sorties)
- ... Et c'est tout (backward automatique !)

Exemple de développement:

```
1  class DeuxCouches(torch.nn.Module): # extension/heritage
2  def __init__(self): # attributs
3      super(DeuxCouches, self).__init__()
4      self.un = torch.nn.Linear(Xdim,5)
5      self.act = torch.nn.Tanh()
6      self.deux = torch.nn.Linear(5,1)
7  def forward(self, x): # inference
8      return self.deux(self.act(self.un(x)))
9
10 netDeuxCouches = DeuxCouches() # instantiation
11 netDeuxCouches(housing_x) # usage en inference
```



Convergence vers une architecture d'apprentissage standard

La standardisation des modules (& architecture logicielle)

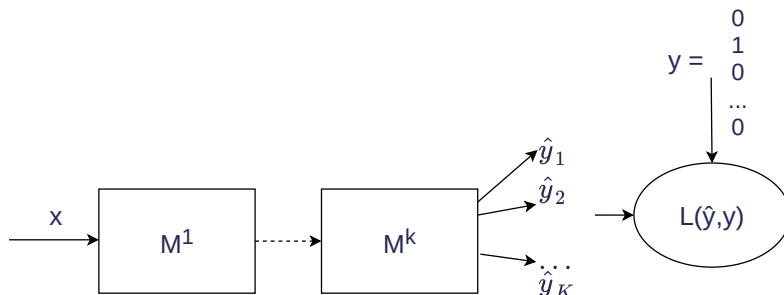
⇒ **Standardisation du processus d'apprentissage**

```
1 # donnees:
2 housing_x, housing_y = ...
3 # definition de:
4 net = ... # reseau de neurones
5 loss = ... # cout
6 optim = ... # optimiseur
7
8 for i in range(EPOCHS): # boucle d'apprentissage
9     loss = mse_loss(net(housing_x).view(-1,1), housing_y.view(-1,1))
10    print(f"iteration : {i}, loss : {loss}")
11    optim.zero_grad() # Mise a zero des gradients
12    loss.backward() # Calcul des gradients
13    optim.step() # MAJ des parametres
```

AUTRES MODULES



Supervision Multi-classe



Quand il faut prédire K classes

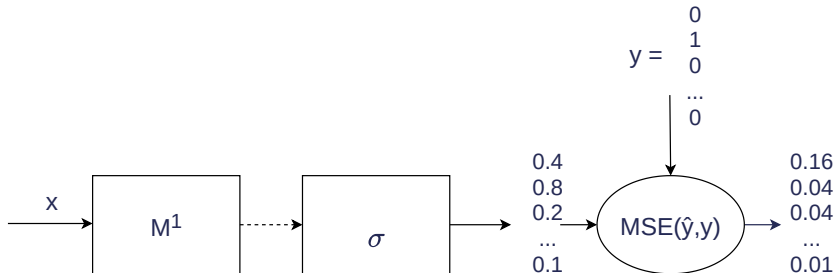
- K sorties
- Utilisation de vecteurs 1-hot pour la supervision:

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

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

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

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 :

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

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



Coût Cross-entropique

SoftMax : Sorties \Rightarrow distribution + renforcement du max

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

Cross-entropie binaire

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

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



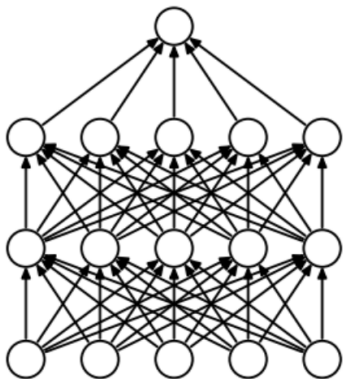
Lutter contre le sur-apprentissage

Différentes techniques qui visent toutes à régulariser le réseau

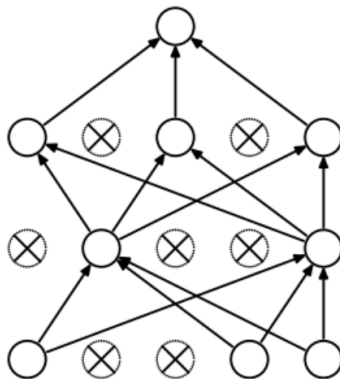
- Régularisation des couches (l_1 , l_2) : ajout d'un terme de pénalisation en $\|W\|^p$ sur les poids des couches
- Dropout : retirer pendant une itération quelques neurones au hasard dans le réseau; permet d'augmenter la robustesse du réseau
- Augmented Data : perturbation des données d'entrées pour 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:



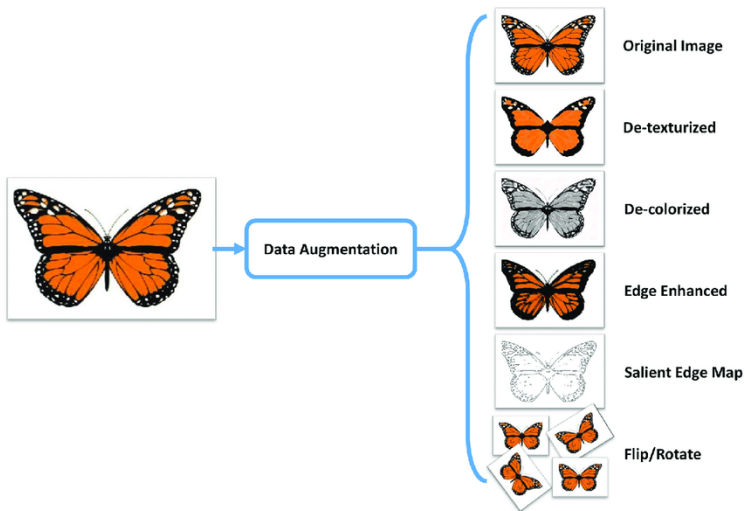
(a) Standard Neural Net



(b) After applying dropout.

Lutter contre le sur-apprentissage

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

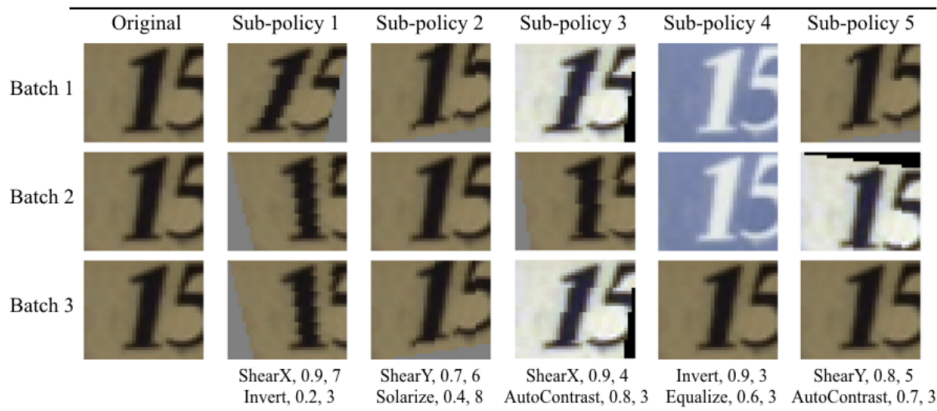




Lutter contre le sur-apprentissage

Data augmentation: comment automatiser le processus?

⇒ outils paramétrables et disponibles dans torchvision



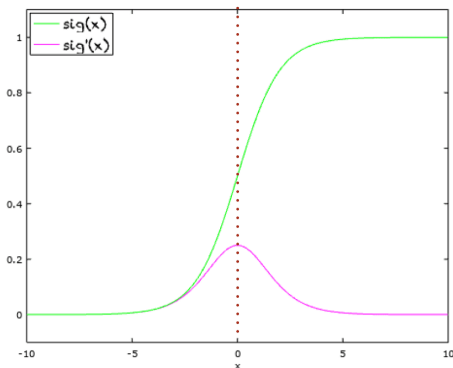


Amélioration du gradient

Gradient vanishing

Le gradient tend à disparaître:

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



Plot of $\sigma(x)$ and its derivate $\sigma'(x)$

Domain: $(-\infty, +\infty)$

Range: $(0, +1)$

$\sigma(0) = 0.5$

Other properties

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

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

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



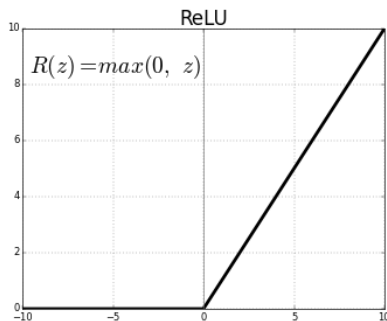
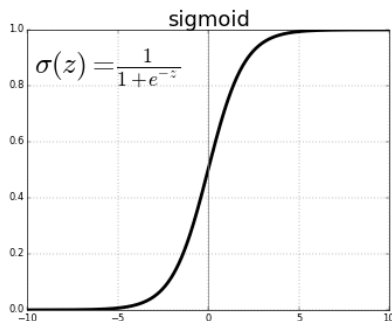
Amélioration du gradient

Gradient vanishing

Le gradient tend à disparaître:

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

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

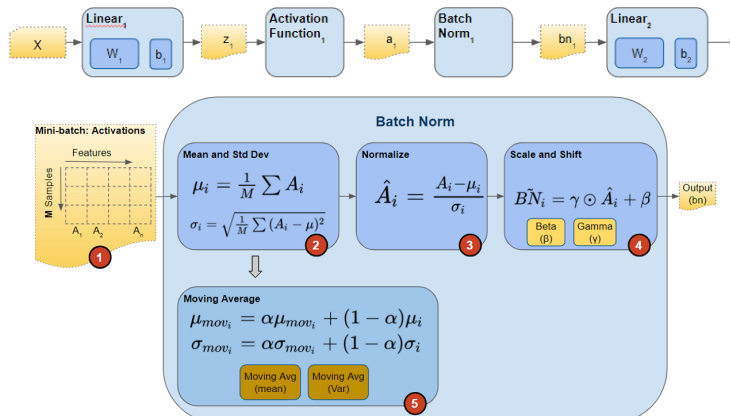




Amélioration du gradient

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

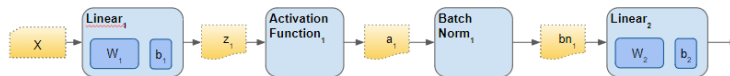




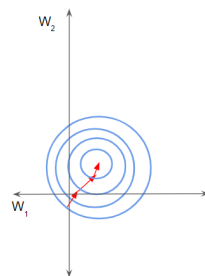
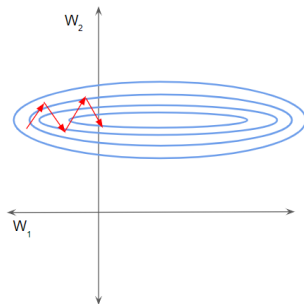
Amélioration du gradient

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





Multiplication des hyper-paramètres

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