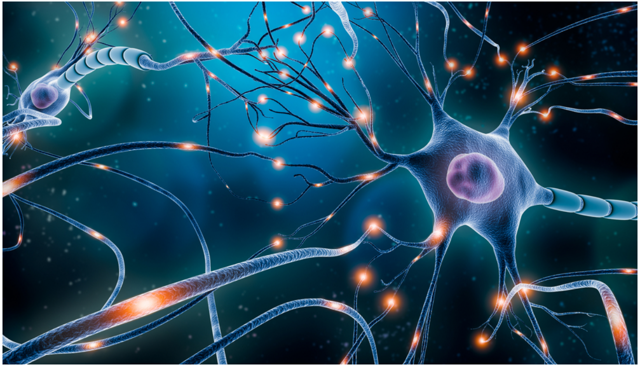
MODULE 1 : INTRODUCTION DEEP LEARNING PRISE EN MAIN PYTORCH

Agro–IODAA–Semestre 1

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

Introduction au Deep Learning

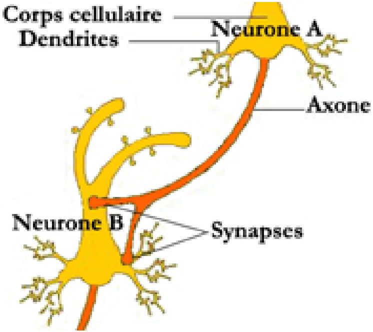
Inspiration biologique [plus ou moins lointaine]

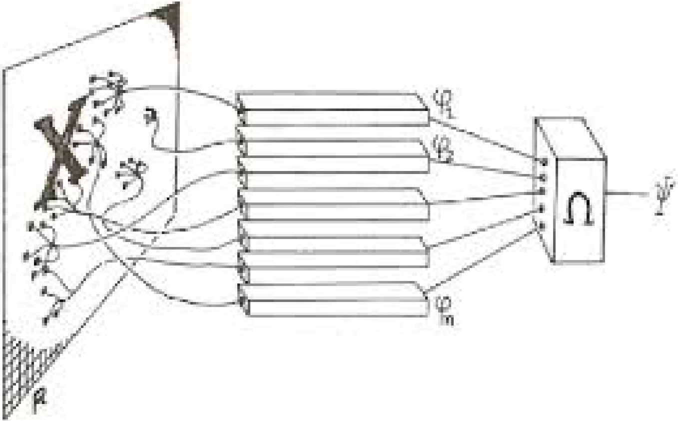


Réseau de neurones

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

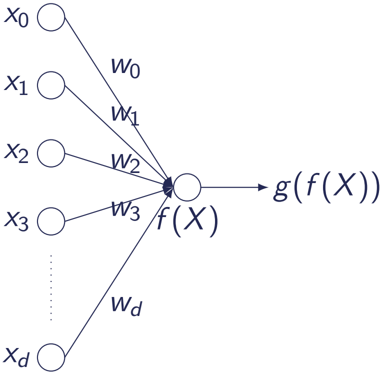
Inspiration biologique [plus ou moins lointaine]





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

Les origines de l'apprentissage profond : le perceptron



Le perceptron

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

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

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

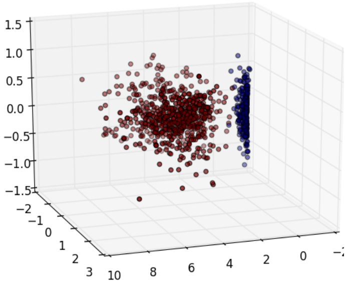
- Problème d'apprentissage : $argmax_{\mathbf{w}} \mathbb{E}_{x,y}[max(0, -y f_{\mathbf{w}}(\mathbf{x}))]$

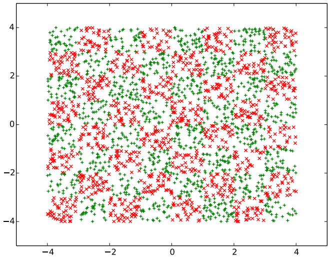
Algorithme du perceptron

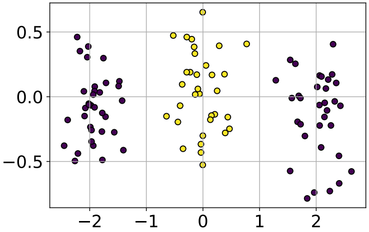
- Tant qu'il n'y a pas convergence :
  - pour tous les exemples (x^i, y^i) :
    - si $(y^i \times < \mathbf{w}.\mathbf{x}^i >) < 0$
      alors $\mathbf{w} = \mathbf{w} + \epsilon y^i \mathbf{x}^i$
- Descente de gradient sur le coût

Limites du perceptron

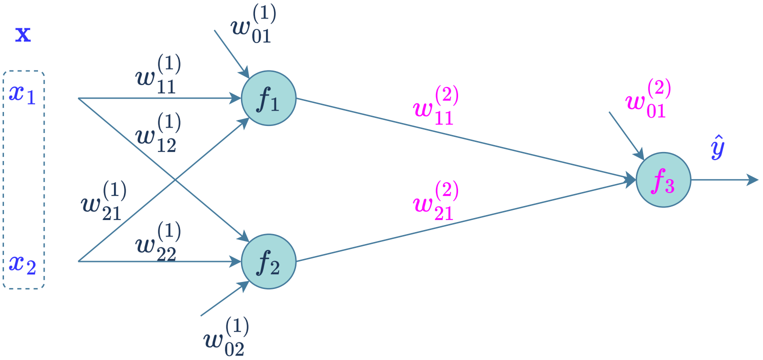
Est-il capable de séparer ces données ?

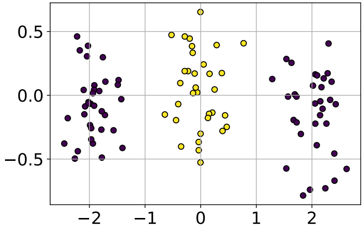






Combinons deux neurones



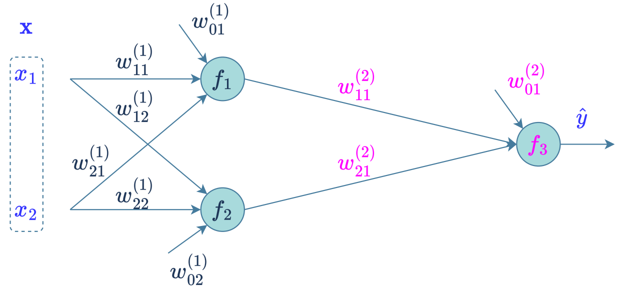


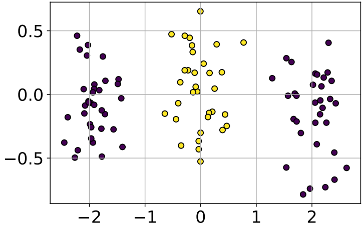
$$f_1(\mathbf{x}) = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{01}^{(1)}, \qquad f_2(\mathbf{x}) = w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{02}^{(1)}$$

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

Combiner des neurones $\Rightarrow$ suffisant ?

Combinons deux neurones





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

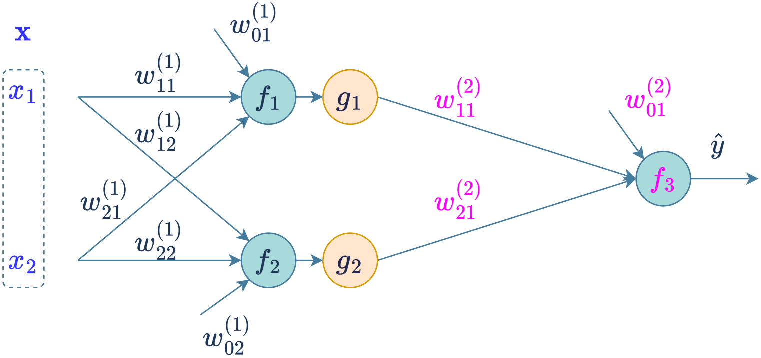
$$f_3(\mathbf{x}) = w_{11}^{(2)}f_1(\mathbf{x}) + w_{21}^{(2)}f_2(\mathbf{x}) + w_{01}^{(2)}$$

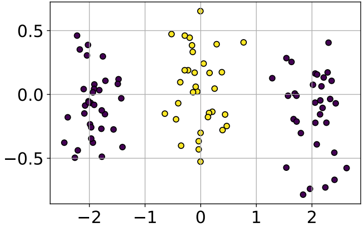
$$f_3(\mathbf{x}) = w_{11}^{(2)}(w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2 + w_{01}^{(1)}) + w_{21}^{(2)}(w_{12}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{02}^{(1)}) + w_{01}^{(2)}$$

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

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

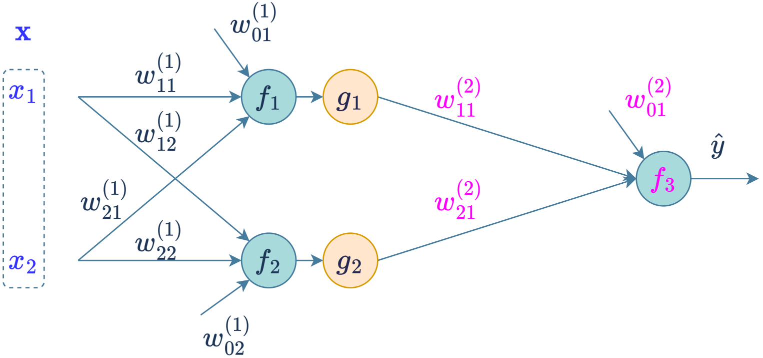
Non-linéarité

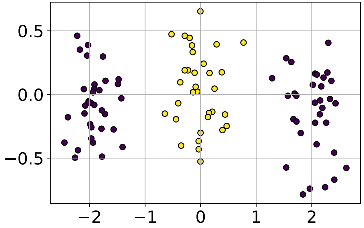




- Quelle non-linéarité ?
  - Fonction *signe* ?
  - ⇒ dérivée problématique . . .
  - Fonctions *tanh*, *sigmoïde*, . . . + biais

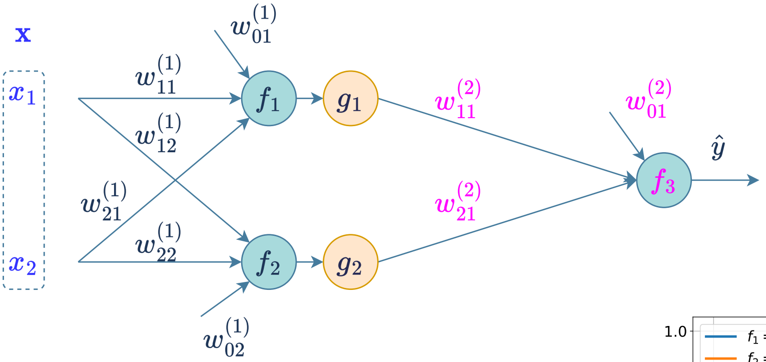
Non-linéarité

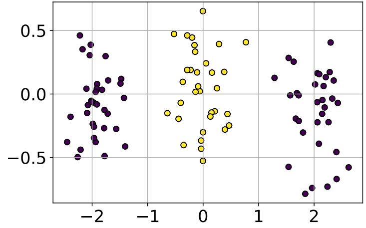


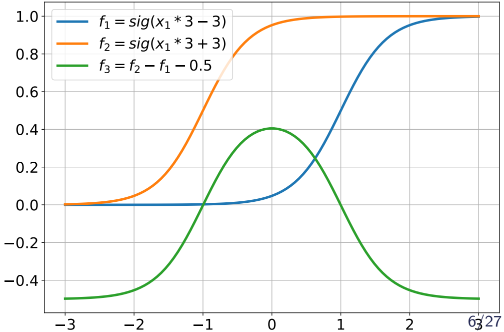


- Quelle non-linéarité ?
  - Fonction *signe* ?
  - $\Rightarrow$ dérivée problématique ...
  - Fonctions *tanh*, *sigmoïde*, ... + biais

Non-linéarité





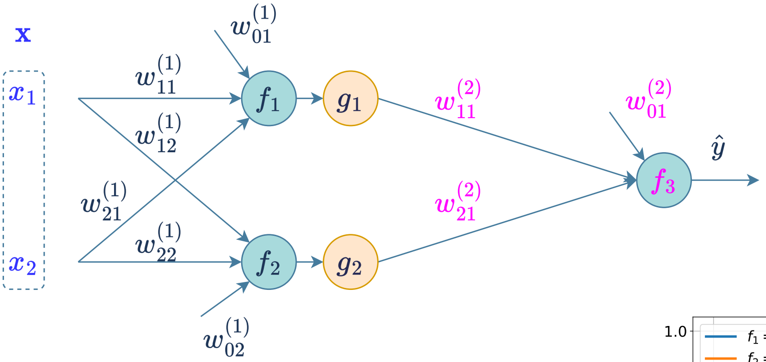


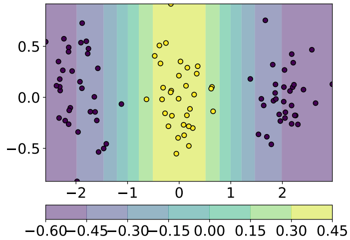
- Quelle non-linéarité ?
  - Fonction *signe* ?

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

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

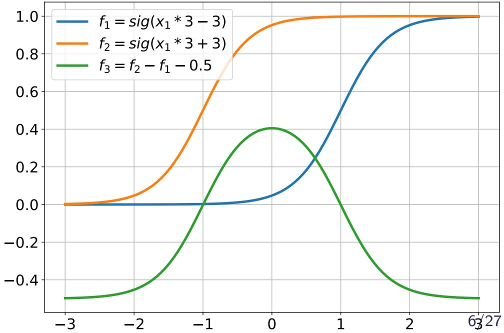
Non-linéarité



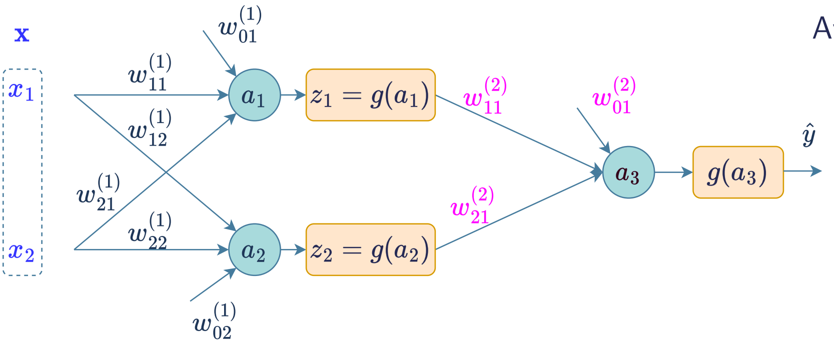


- Quelle non-linéarité ?
  - Fonction *signe* ?
  - $\Rightarrow$ dérivée problématique ...
  - Fonctions *tanh*, *sigmoïde*, ... + biais

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



Vocabulaire de l'inférence



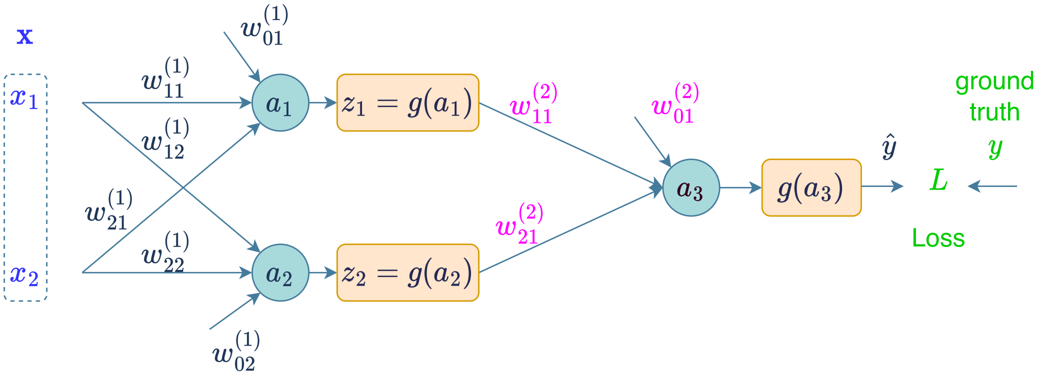
Avec $g(x) = \dfrac{1}{1 + \exp(-x)}$

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

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).

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 ?

$\Rightarrow$ Rétro-propagation de l'erreur

Descente de gradient

Objectif: calculer les gradients partiels par rapport aux paramètres

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

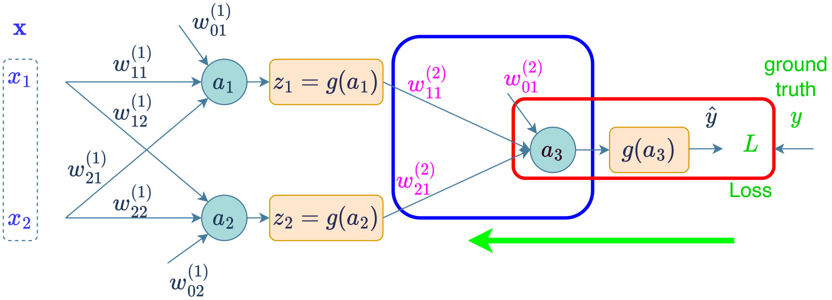
Forward: calcul de $\hat{y}$ [entre autres]

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

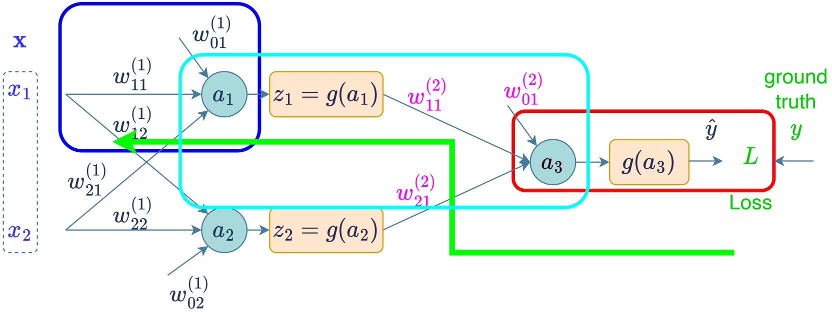
Backward, poids de la **dernière couche** : $\nabla_{w_{ij}^{(2)}} L(\hat{y}, y)$

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

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

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

Calcul du gradient: chain rule [suite]



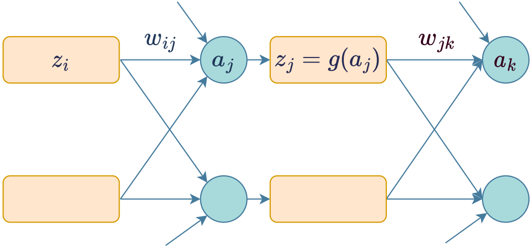
Forward:
$\hat{y} = 0.5$
$y = -1$

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

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

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

Cas général dans les couches intermédiaires



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

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

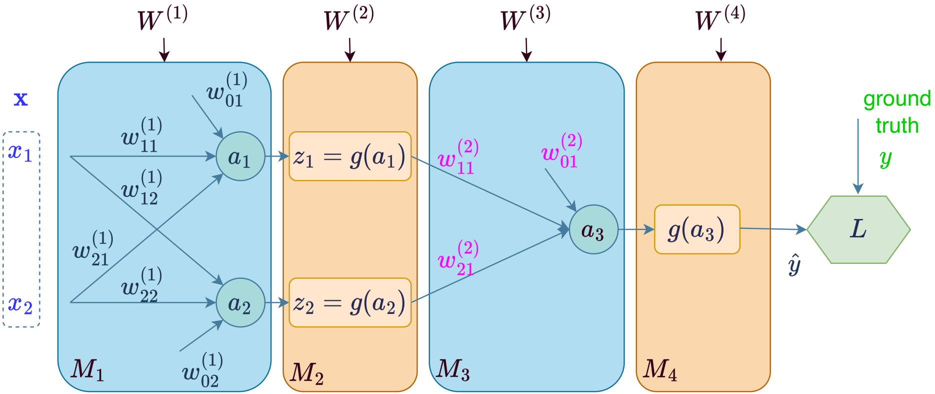
$$\underbrace{\frac{\partial L}{\partial a_j}}_{\text{erreur sur } j} = \sum_k (g'(a_k) w_{jk}) \underbrace{\frac{\partial L}{\partial a_k}}_{\text{erreur à propager}}$$

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

- 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

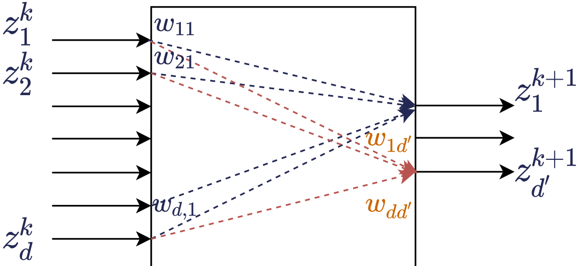


Un module M^k

- 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

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



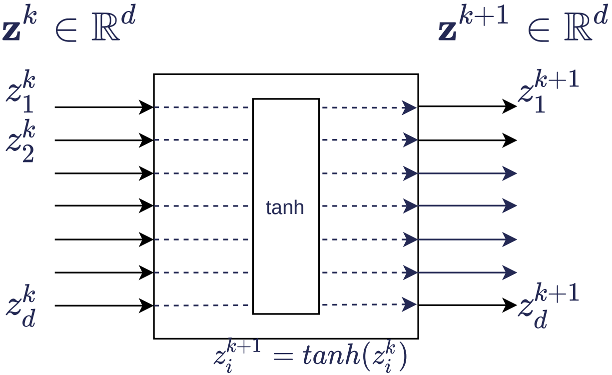
$$z_i^{k+1} = w_{1i}\, z_1^k + w_{2i}\, z_2^k + \ldots + w_{d,i}\, z_d^k$$
$$\mathbf{z}^{k+1} = \mathbf{W}\mathbf{z}^k$$

Couche linéaire [Linear]

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

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

Type usuel de modules

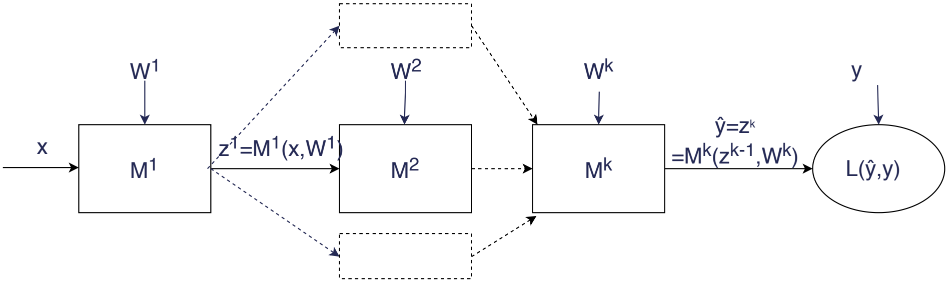


Module d'activation

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

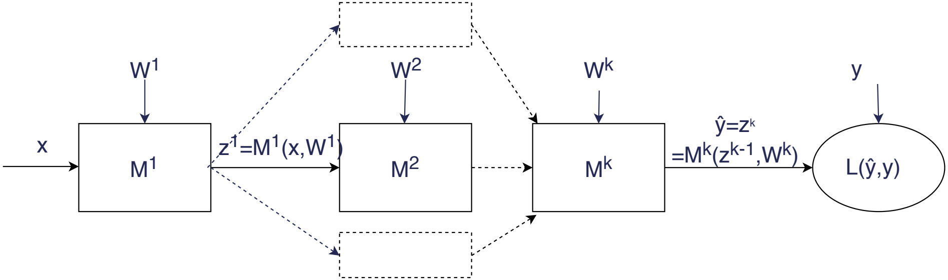


Un coût

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

et d'autres composantes plus ésotériques

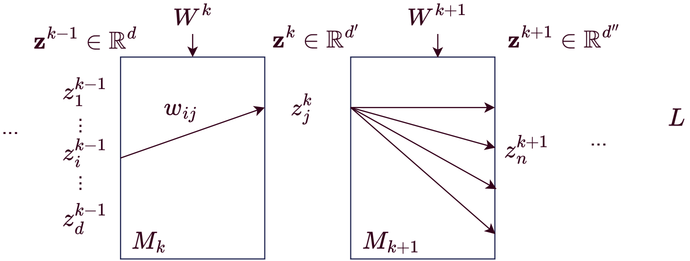
Apprentissage du réseau



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.

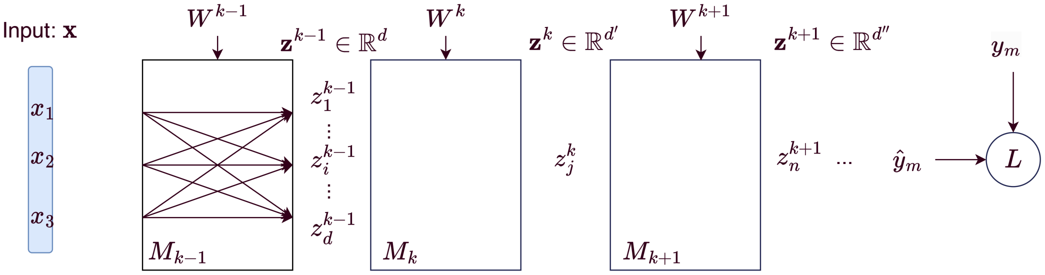
Zoom sur le module k



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

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

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

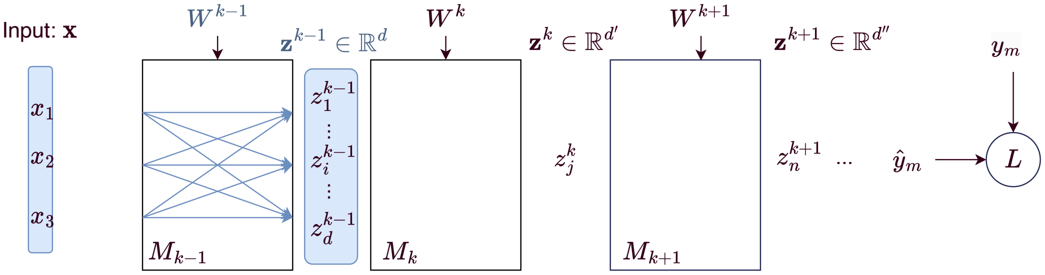


- $z^1 = M_1(\mathbf{x}, W^1)$
- $z^k = M_k(z_{k-1}, W^k)$

\+ Stockage des z^k

- Jusqu'à $\hat{\mathbf{y}}$

Zoom sur le module k: *forward / backward*

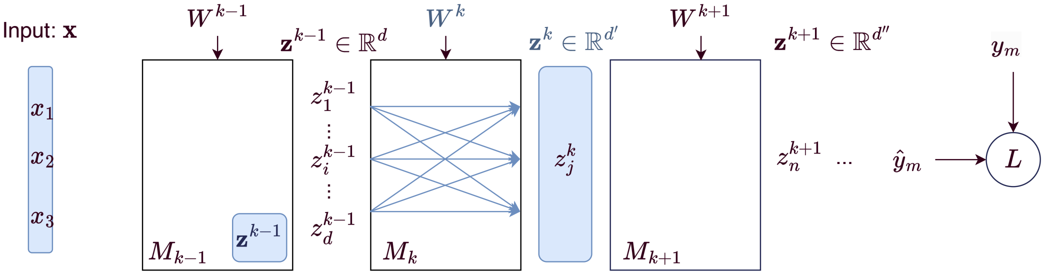


- $z^1 = M_1(\mathbf{x}, W^1)$
- $z^k = M_k(z_{k-1}, W^k)$

+ Stockage des z^k

- Jusqu'à $\hat{\mathbf{y}}$

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

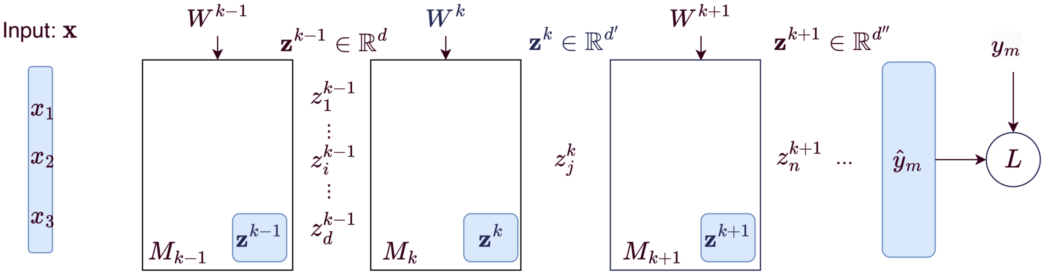


- $z^1 = M_1(\mathbf{x}, W^1)$
- $z^k = M_k(z_{k-1}, W^k)$

\+ Stockage des z^k

- Jusqu'à $\hat{\mathbf{y}}$

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

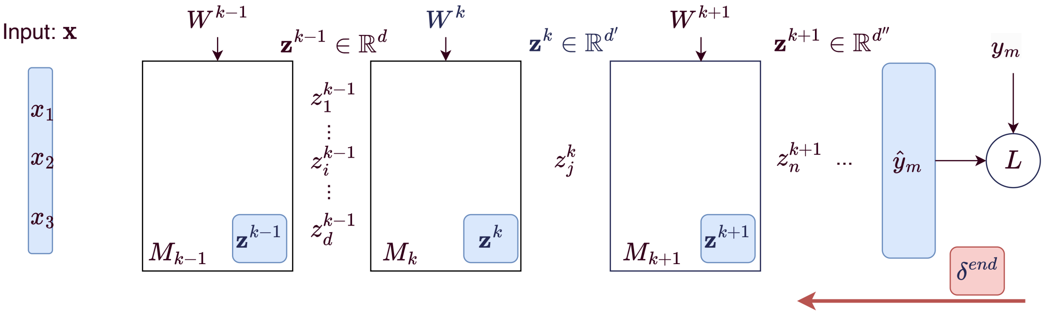


- $z^1 = M_1(\mathbf{x}, W^1)$
- $z^k = M_k(z_{k-1}, W^k)$

\+ Stockage des z^k

- Jusqu'à $\hat{\mathbf{y}}$

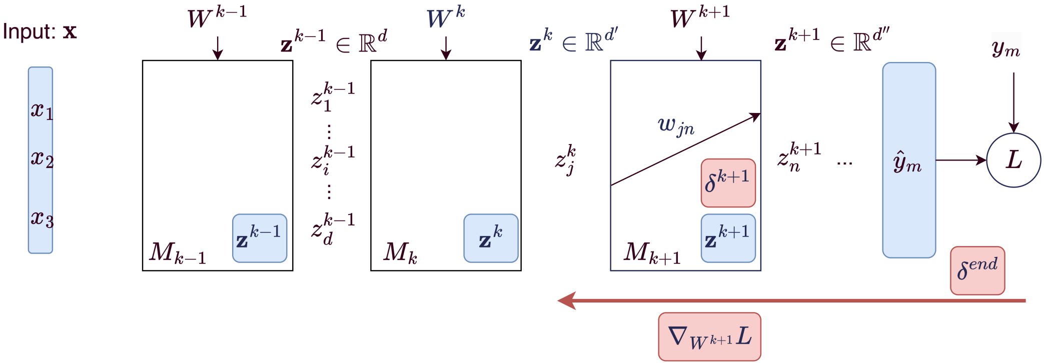
Zoom sur le module k: *forward* / *backward*



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

Dans le cas de la MSE

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

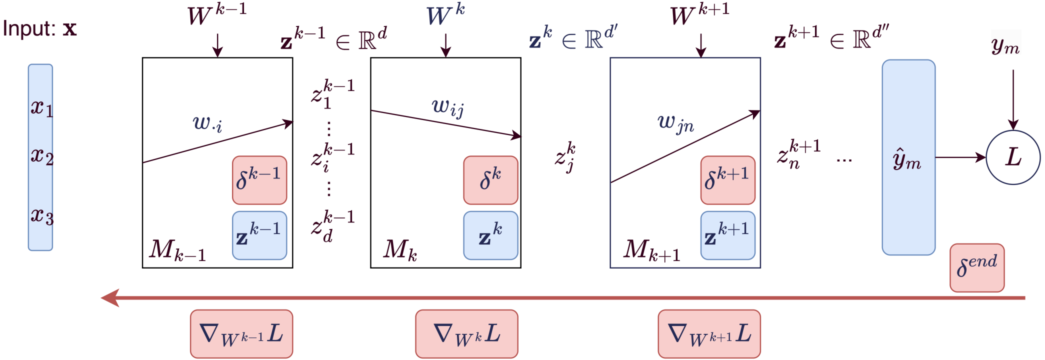


Supposons que $z^{k+1} = \hat{\mathbf{y}}_m = z^{end} \Rightarrow \delta_n^{k+1} = \delta_n^{end} = \frac{2}{N}(z_n^{end} - y_n)$

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*



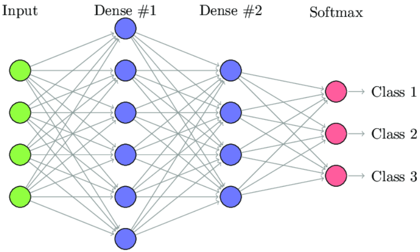
$$\delta_j^k = \frac{\partial L}{\partial z_j^k} = \sum_n \frac{\partial L}{\partial z_n^{k+1}} \frac{\partial z_n^{k+1}}{\partial z_j^k} = \sum_n \delta_n^{end} \frac{M^{k+1}(\mathbf{z}^k, W^{k+1})}{\partial z_j^k} = \sum_n \delta_n^{end} w_{jn}^{k+1}$$

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 = torch.nn.Linear(args)` Inférence: `z = M(x)`

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

```
Xdim = housing_x.size(1)
## Creation d'une couche lineaire de dimension Xdim->1
net = torch.nn.Linear(Xdim, 1)  # recuperation d'un module
## Fonction de cout
mseloss = torch.nn.MSELoss()
## Optimiseur (& recuperation des parametres)
optim = torch.optim.SGD(params=net.parameters(),lr=EPS)

yhat = net(housing_x) # inference des modules = appel direct
loss = mseloss(net(housing_x).view(-1,1),housing_y.view(-1,1))
loss.backward()
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:

```
class DeuxCouches(torch.nn.Module): # extension/heritage
def __init__(self):                 # attributs
    super(DeuxCouches, self).__init__()
    self.un = torch.nn.Linear(Xdim,5)
    self.act = torch.nn.Tanh()
    self.deux = torch.nn.Linear(5,1)
def forward(self,x):                # inference
    return self.deux(self.act(self.un(x)))

netDeuxCouches = DeuxCouches()      # instanciation
netDeuxCouches(housing_x)           # usage en inference
```

Convergence vers une architecture d'apprentissage standard

La standardisation des modules (& architecture logicielle)

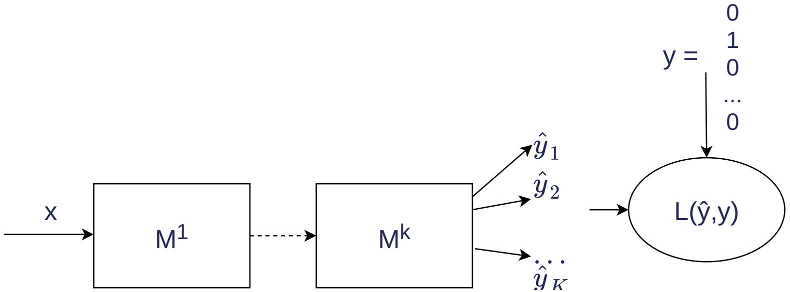
⇒ **Standardisation du processus d'apprentissage**

```python
# donnees:
housing_x, housing_y = ...
# definition de:
net   = ... # reseau de neurones
loss  = ... # cout
optim = ... # optimiseur

for i in range(EPOCHS): # boucle d'apprentissage
    loss = mseloss(net(housing_x).view(-1,1),housing_y.view(-1,1))
    print(f"iteration : {i}, loss : {loss}")
    optim.zero_grad()   # Mise a zero des gradients
    loss.backward()     # Calcul des gradients
    optim.step()        # MAJ des parametres
```

Autres modules importants

Supervision Multi-Classes



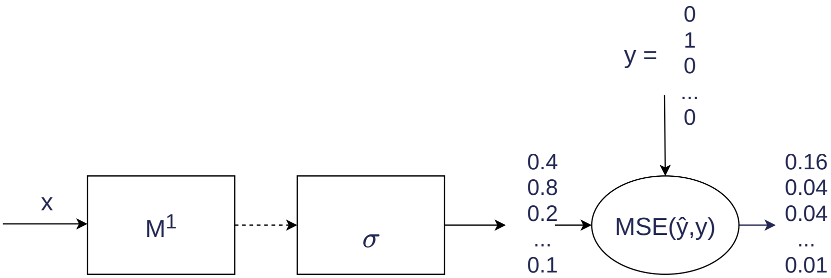
Quand il faut prédire K classes

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

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

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 (=MV multinomiale)

SoftMax : Sorties ⇒ distribution + renforcement du max

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

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 (=MV multinomiale)

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):

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

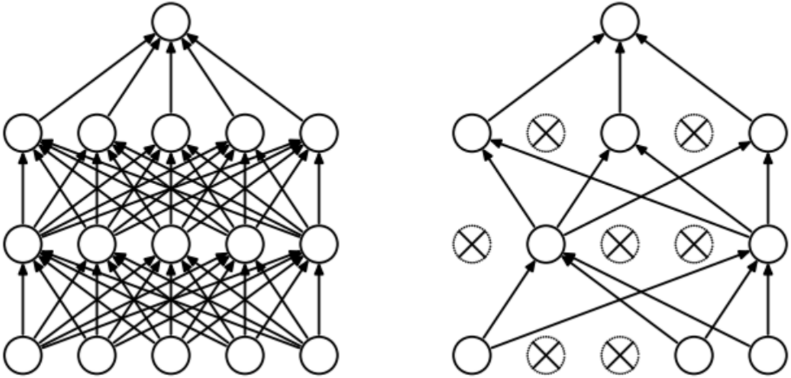
Lutter contre le sur-apprentissage

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

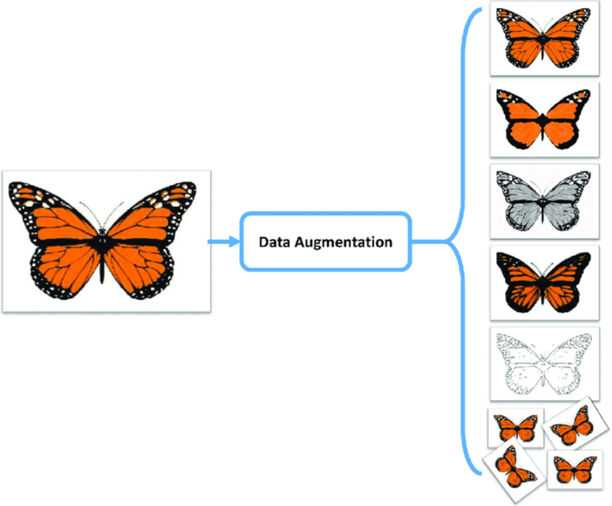
Drop out:



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

Lutter contre le sur-apprentissage

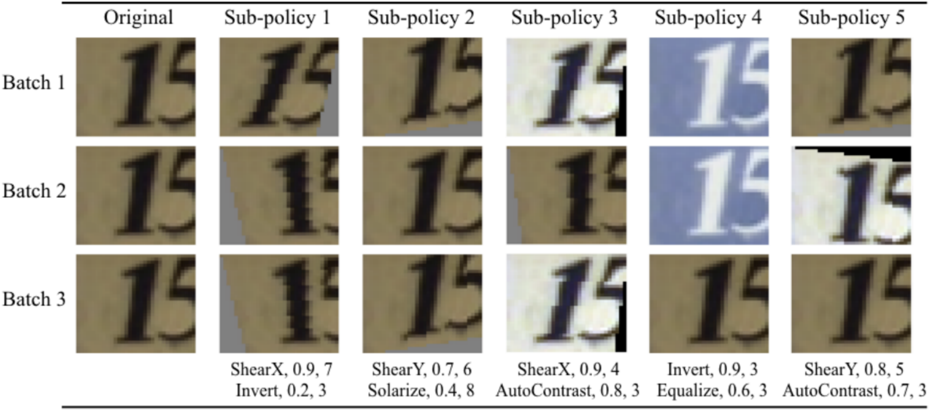
Data augmentation: une idée simple pour régulariser par la masse et les variations



Lutter contre le sur-apprentissage

Data augmentation: comment automatiser le processus?

⇒ outils paramétrables et disponibles dans `torchvision`

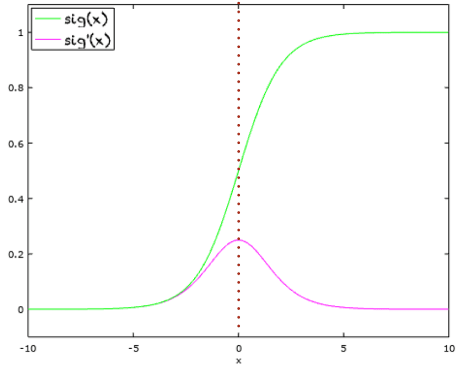


Amélioration du gradient

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

Range: $(0, +1)$

$\sigma(0) = 0.5$

Other properties

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

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

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

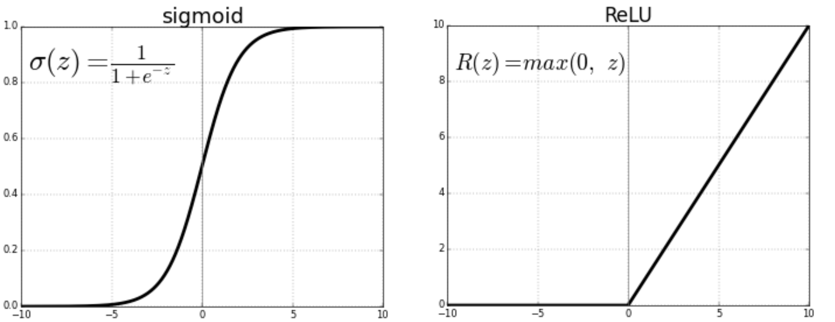
Amélioration du gradient

Gradient vanishing

Le gradient tend à disparaitre:

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

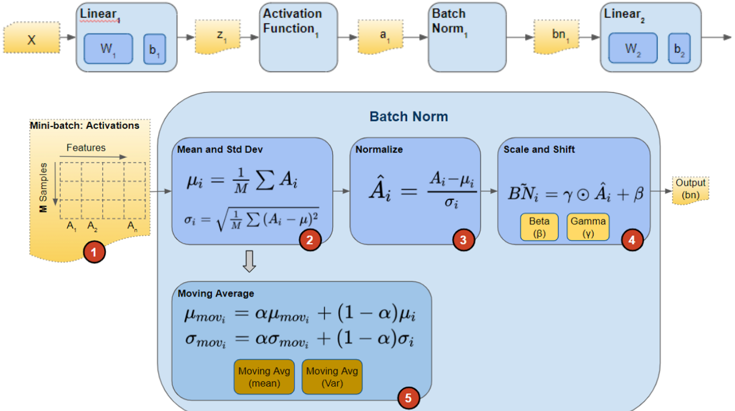
Fonction d'activation spécifique $ReLU(x) = max(0, x)$: permet de garder un gradient fort lorsque le neurone est activé



Amélioration du gradient

Topologie de l'espace de recherche & gradient

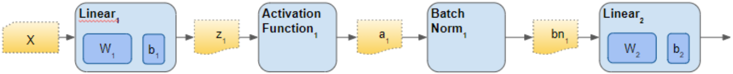
- **BatchNorm** : pour une couche, centrée/normée chaque sortie (estimation sur chaque mini-batch)
- **LayerNorm** : à la sortie d'une couche, normalisation de chaque exemple séparément de ses dimensions



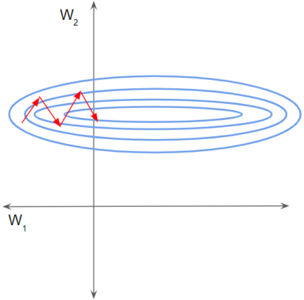
Amélioration du gradient

Topologie de l'espace de recherche & gradient

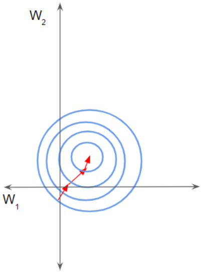
- **BatchNorm** : pour une couche, centrée/normée chaque sortie (estimation sur chaque mini-batch)
- **LayerNorm** : à la sortie d'une couche, normalisation de chaque exemple séparément de ses dimensions



Centrer les données = meilleure topologie pour l'apprentissage



$\Rightarrow$



Conclusion

Multiplication des modules et des hyper-paramètres

- Architecture = beaucoup d'hyperparamètres
- Besoin de normalisation pour:
  - Mieux comparer les architectures
  - Avoir des a priori sur les bons paramètres
- Lexique des modules:
  - comprendre les enjeux, savoir lire les articles

Outils supplémentaires:

- `DataLoader`
- `Check-pointing`
- `TensorBoard`