Apprentissage statistique : modélisation décisionnelle et apprentissage profond (RCP209)

Introduction à l'apprentissage supervisé

Nicolas Audebert
nicolas.audebert@lecnam.net
http://cedric.cnam.fr/vertigo/Cours/ml2/

Département Informatique
Conservatoire National des Arts & Métiers, Paris, France

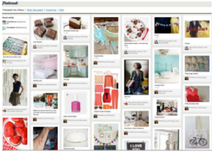
06 avril 2023

Plan du cours

Données massives

- Accès à de vastes quantités de données non-structurées : images, vidéos, sons, textes, signaux, etc.

BBC : 2,4m vidéos



Facebook : 350M images

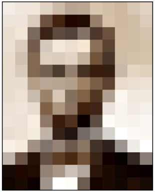
100m caméras de surveillance

- Besoin d'accéder à ces données, de les indexer, de les classer : **reconnaissance**

$\Rightarrow$ Sous quelle représentation ?

Extraire du sens des signaux bas niveau

Problème : combler l'écart entre le signal et la sémantique.



243	239	240	225	206	185	188	218	211	206	216	225
242	239	218	110	67	31	34	152	213	206	208	221
243	242	123	58	94	82	132	77	108	208	208	215
235	217	115	212	243	236	247	139	91	209	208	211
233	208	131	222	219	226	196	114	74	208	213	214
232	217	131	116	77	150	69	56	52	201	228	223
232	232	182	186	184	179	159	123	93	232	235	235
232	236	201	154	216	133	129	81	175	252	241	240
235	238	230	128	172	138	65	63	234	249	241	245
237	236	247	143	59	78	10	94	255	248	247	251
234	237	245	193	55	33	115	144	213	255	253	251
248	245	161	128	149	109	138	65	47	156	239	255
190	107	39	102	94	73	114	58	17	7	51	137
23	32	33	148	168	203	179	43	27	17	12	8
17	26	12	160	255	255	109	22	26	19	35	24

Ce que l'humain perçoit
vs.
ce que la machine perçoit

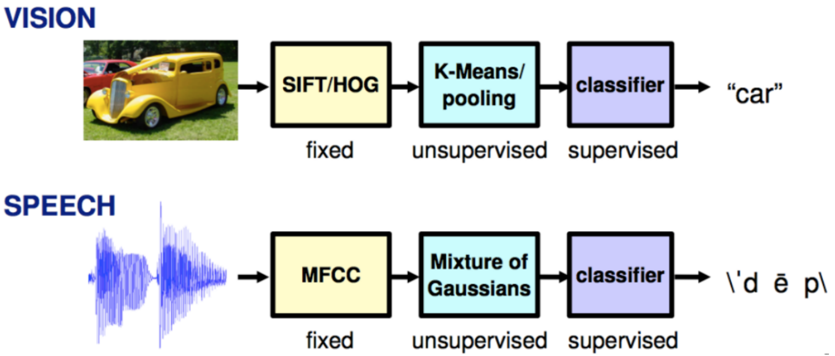
- Variations d'illumination
- Variations de points de vue
- Objets déformables
- Variance et diversité intra-classe
- ...

⇒ **Comment concevoir de "bonnes"représentations des données ?**

Apprentissage statistique classique

Des données à la décison :

1. Choix de caractéristiques descriptives *hand-crafted*
   - descripteurs images (SIFT/HOG), coefficients audio (MFCC), n-grammes/*bag-of-words*, etc.
   - nécessite des connaissances expertes du domaine
   - savoir *a priori*
2. Entraînement d'un modèle de décisionnel
3. Évaluation des performances

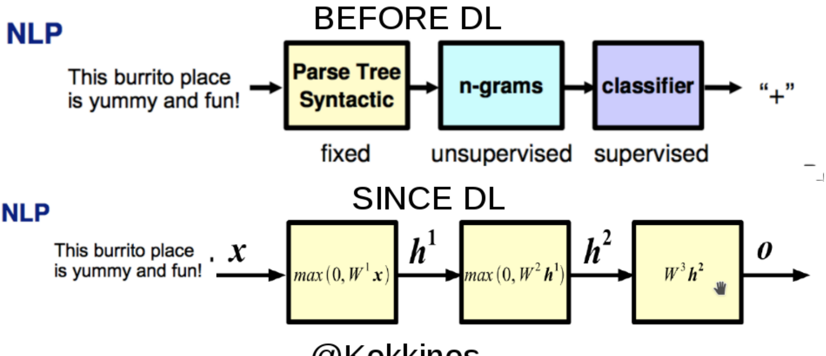


Apprentissage statistique classique

Des données à la décison :

1. Choix de caractéristiques descriptives *hand-crafted*
   - descripteurs images (SIFT/HOG), coefficients audio (MFCC), n-grammes/*bag-of-words*, etc.
   - nécessite des connaissances expertes du domaine
   - savoir *a priori*
2. Entraînement d'un modèle de décisionnel
3. Évaluation des performances

Apprentissage profond ⇒ apprentissage de **représentations**



Historique des réseaux de neurones

- **1943 :** Neurone artificiel McCulloch et Pitts 1943
- **1957 :** Perceptron Rosenblatt 1957
- **1974 :** Algorithme de rétropropagation Werbos 1975
- **1980 :** Neocognitron (premier réseau “profond”) à poids partagés Fukushima 1980
- **1989 :** LeNet-5 (premier réseau convolutif) LeCun et al. 1989
- **2011/2012** : DanNet/AlexNet (premiers modèles à l’état de l’art en reconnaissance d’image) Cireşan, Meier et Schmidhuber 2012 ; Krizhevsky, Sutskever et Hinton 2012

Plan du cours

1. Réseaux de neurones artificiels
2. Optimisation des réseaux profonds
3. Réseaux convolutifs (CNN)
4. Réseaux convolutifs modernes
5. Réseaux récurrents (RNN)
6. Réseaux récurrents modernes
7. Reconnaissance d'image avancée
8. Apprentissage profond avancé

Pour approfondir

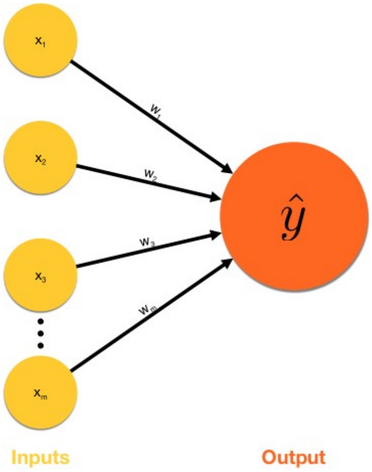
- RCP211 : Intelligence artificielle avancée
  - Apprentissage par renforcement
  - Modèles génératifs profonds (GAN, VAE)
  - Robustesse et incertitude des réseaux profonds
- RCP217 : Intelligence artificielle pour le multimédia
  - Traitement du langage naturel (NLP)
  - Séries temporelles, audio et vidéo
  - Systèmes de recommandations et graphes

Plan du cours

Neurone formel

- Modèle simplifié d'un neurone réel (1943)
  McCulloch et Pitts 1943
  - Entrée : vecteur $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{x} = \{x_1, x_2, \dots, x_m\}$,
  - Poids w_j, pondération de la connexion entre l'entrée j et la sortie,
  - Sortie $\hat{y} \in \mathbb{R}$ (scalaire), $\hat{y} = \sum_{j=1}^{m} w_j x_j$.
- Expression matricielle :

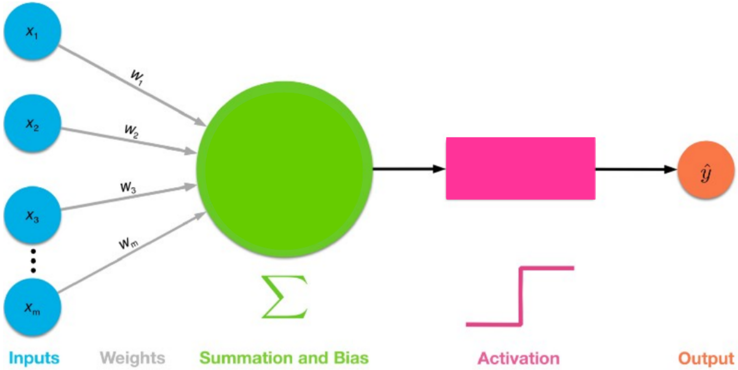
$$\hat{y} = \mathbf{w}^T\mathbf{x} = \begin{bmatrix} w_1 & w_2 & \cdots & w_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$



Neurone artificiel

- Fonction de $\mathbf{x} \in \mathbb{R}^m$ vers $\hat{y} \in \mathbb{R}$:
  1. Fonction affine (linéaire plus translation par un biais b scalaire) : $s = \mathbf{w}^T\mathbf{x} + b$
  2. Fonction d'activation non-linéaire $\phi : \mathbb{R} \to \mathbb{R}$: $\hat{y} = \phi(s)$
- Expression du neurone artificiel "complet"

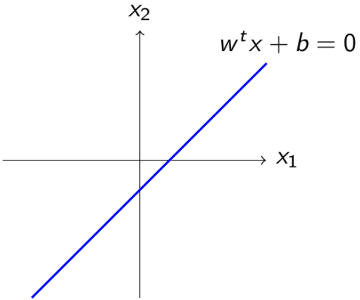
$$\hat{y} = \phi\left(\mathbf{w}^T\mathbf{x} + b\right)$$



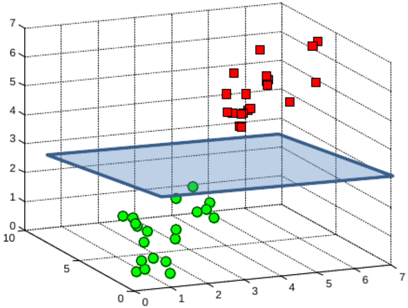
Partie linéaire

- Projection affine : $s = \mathbf{w}^\top \mathbf{x} + b = \sum_{i=1}^{m} w_i x_i + b$
  - $\mathbf{w}$: vecteur à un hyperplan de $\mathbb{R}^m \Rightarrow s = 0$ définit une **frontière linéaire**
  - bias b définit un écart par rapport à la position de l'hyperplan

Hyperplan en dimension 2 : droite



Hyperplan en dimension 3 : plan

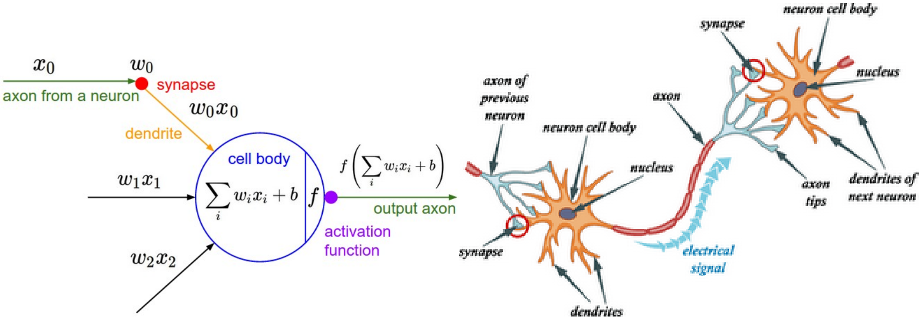


Fonction d'activation non-linéaire

$$\hat{y} = \sigma\left(\mathbf{w}^T\mathbf{x} + b\right)$$

- Fonction identité : $\phi(x) = x$
- Fonction de Heaviside (échelon) : $\phi(x) = \begin{cases} 1 & \text{si} x \geq 0 \\ 0 & \text{sinon} \end{cases}$
- Sigmoïde : $\sigma(x) = \frac{1}{1+e^{-x}}$
- Tangente hyperbolique : $\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- *Rectified Linear Unit* (ReLU) : $\phi(x) = \begin{cases} x & \text{si} x \geq 0 \\ 0 & \text{sinon} \end{cases}$

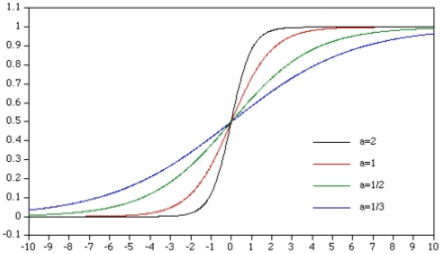
Lien avec les neurones biologiques



- Neurone formel, activation Heaviside H : $\hat{y} = H(\mathbf{w}^\top \mathbf{x} + b)$
  - $\hat{y} = 1$ (neurone actif) $\Leftrightarrow \mathbf{w}^\top \mathbf{x} \geq -b$
  - $\hat{y} = 0$ (neurone inactif) $\Leftrightarrow \mathbf{w}^\top \mathbf{x} < -b$
- **Neurones biologiques** : la sortie est active $\Leftrightarrow$ la somme des entrées pondérées par le poids des connexions est synaptique est supérieure à un seuil.

Activation sigmoïde

- Équation du neurone artificiel : $\hat{y} = \phi(\mathbf{w}^\top \mathbf{x} + b)$
- Sigmoïde : $\phi(z) = \sigma_a(z) = (1 + e^{-az})^{-1}$

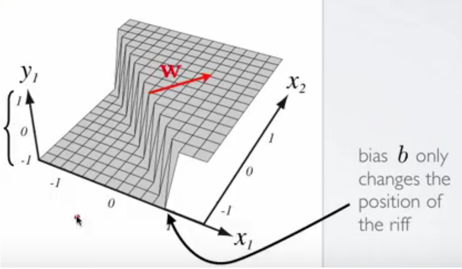


- $a \uparrow$: se rapproche de l'échelon de Heaviside ($a \to \infty$)
- Sigmoïde : deux régimes
  - $x \approx 0 \to$ régime linéaire ($\sigma(x) \approx ax$)
  - $x << 0$ ou $x >> 0 \to$ régime de saturation ($\sigma(x) \approx \pm 1$)

Application à la classification binaire

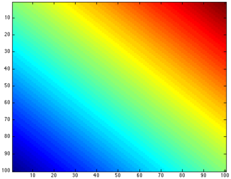
- Chaque entrée $\mathbf{x}$ appartient soit à la classe 0, soit à la classe 1.
- Sortie du neurone artificiel (sigmoïde) : $\hat{y} = \frac{1}{1+e^{-a(\mathbf{w}^\top \mathbf{x}+b)}}$
- Interprétation probabiliste $\Rightarrow \hat{y} \sim P(1|\mathbf{x})$
  - L'entrée $\mathbf{x}$ est classée comme classe 1 si $P(1|\mathbf{x}) > 0.5 \Leftrightarrow \mathbf{w}^\top \mathbf{x} + b > 0$
  - L'entrée $\mathbf{x}$ est classée comme classe 0 si $P(1|\mathbf{x}) < 0.5 \Leftrightarrow \mathbf{w}^\top \mathbf{x} + b < 0$

  $\Rightarrow \mathrm{signe}(\mathbf{w}^\top \mathbf{x} + b)$: la frontière de classification est linéaire dans l'espace d'entrée !



Exemple jouet en 2D

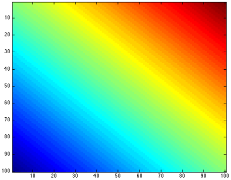
- En 2D ($m = 2$), $\mathbf{x} = \{x_1, x_2\} \in [-5; 5]^2$
- Poids fixés $\mathbf{w} = [1; 1]$ et $b = -2$
- Résultat de la fonction affine : $s = \mathbf{w}^\top \mathbf{x} + b = x_1 + x_2 - 2$



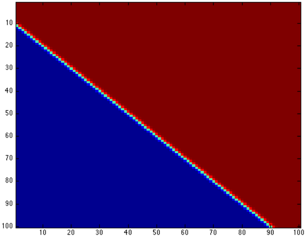
- Activation non-linéaire, sigmoïde $a = 10$: $\hat{y} = \left(1 + e^{-a(\mathbf{w}^\top \mathbf{x} + b)}\right)^{-1}$

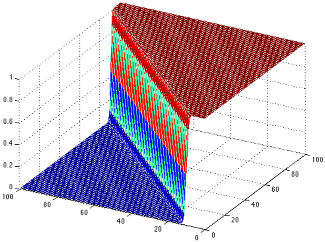
Exemple jouet en 2D

- En 2D ($m = 2$), $\mathbf{x} = \{x_1, x_2\} \in [-5; 5]^2$
- Poids fixés $\mathbf{w} = [1; 1]$ et $b = -2$
- Résultat de la fonction affine : $s = \mathbf{w}^\top \mathbf{x} + b = x_1 + x_2 - 2$



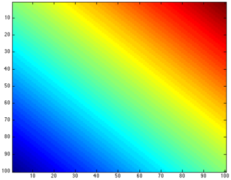
- Activation non-linéaire, sigmoïde $a = 10$: $\hat{y} = \left(1 + e^{-a(\mathbf{w}^\top \mathbf{x} + b)}\right)^{-1}$



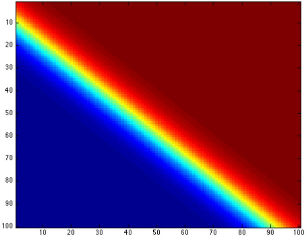


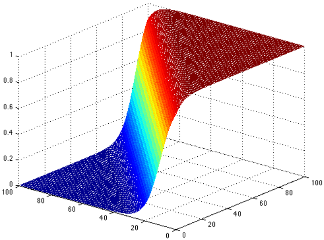
Exemple jouet en 2D

- En 2D ($m = 2$), $\mathbf{x} = \{x_1, x_2\} \in [-5; 5]^2$
- Poids fixés $\mathbf{w} = [1; 1]$ et $b = -2$
- Résultat de la fonction affine : $s = \mathbf{w}^\top \mathbf{x} + b = x_1 + x_2 - 2$

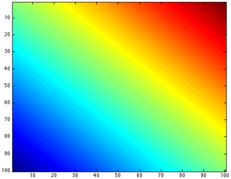


- Activation non-linéaire, sigmoïde $a = 1$: $\hat{y} = \left(1 + e^{-a(\mathbf{w}^\top \mathbf{x} + b)}\right)^{-1}$



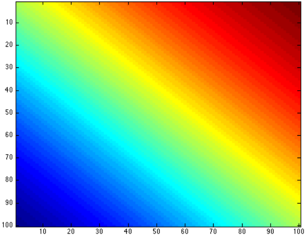


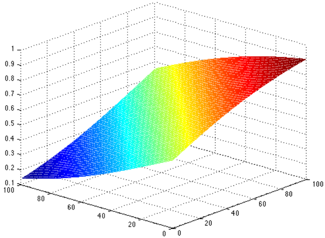
Exemple jouet en 2D



- En 2D ($m = 2$), $\mathbf{x} = \{x_1, x_2\} \in [-5; 5]^2$
- Poids fixés $\mathbf{w} = [1; 1]$ et $b = -2$
- Résultat de la fonction affine : $s = \mathbf{w}^\top \mathbf{x} + b = x_1 + x_2 - 2$

- Activation non-linéaire, sigmoïde $a = 0.1$: $\hat{y} = \left(1 + e^{-a(\mathbf{w}^\top \mathbf{x} + b)}\right)^{-1}$



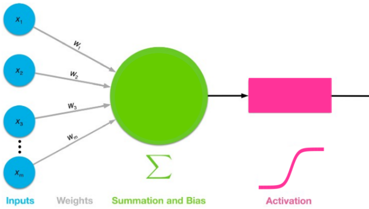


Du neurone formel au réseau de neurones

- Neurone artificiel :
  1. Une seule sortie scalaire $\hat{y}$
  2. Frontière linéaire pour la classification binaire
- Sortie scalaire unique : expressivité limitée pour la plupart des tâches
  - Comment gérer la classification multi-classe ?

  $\Rightarrow$ utiliser plusieurs neurones de sortie plutôt qu'un seul !

  $\Rightarrow$ modèle neuronal dit **Perceptron** (Rosenblatt 1957)

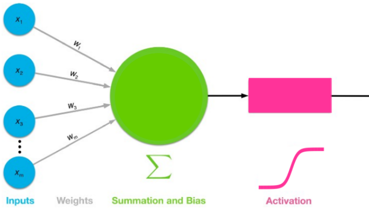


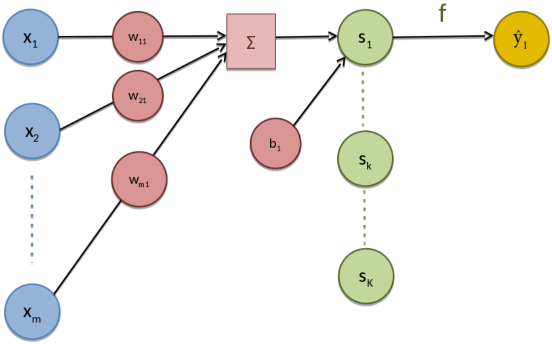
Du neurone formel au réseau de neurones

- Neurone artificiel :
  1. Une seule sortie scalaire $\hat{y}$
  2. Frontière linéaire pour la classification binaire
- Sortie scalaire unique : expressivité limitée pour la plupart des tâches
  - Comment gérer la classification multi-classe ?

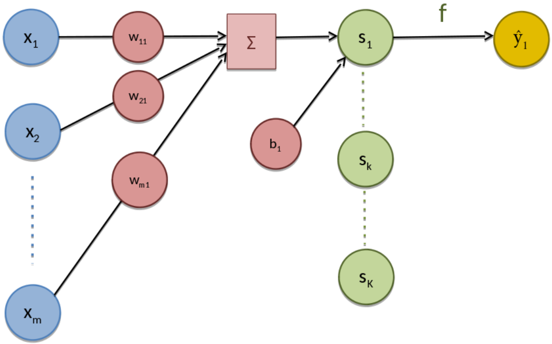
  ⇒ utiliser plusieurs neurones de sortie plutôt qu'un seul !

  ⇒ modèle neuronal dit **Perceptron** (Rosenblatt 1957)



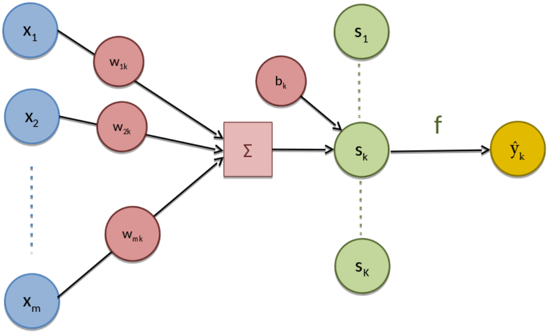


Perceptron et classification multi-classe



- Entrée $\mathbf{x}$ in $\mathbb{R}^m$ ("couche"d'entrée)
- Chaque sortie $\hat{y_1}$ est un neurone artificiel ("couche"de sortie). Par exemple, pour la sortie $\hat{y_1}$:
  - Transformation affine : $s_1 = \mathbf{w_1}^T\mathbf{x} + b_1$
  - Activation non-linéaire σ : $\hat{y_1} = \sigma(s_1)$
- Paramètres de la transformation linéaire la sortie $\hat{y_1}$:
  - poids $\mathbf{w_1} = \{w_{1,1}, w_{2,1}, \ldots, w_{m,1}\} \in \mathbb{R}^m$
  - biais $b_1 \in \mathbb{R}$

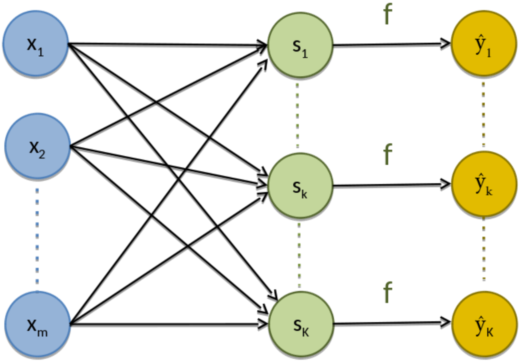
Perceptron et classification multi-classe



- Entrée $\mathbf{x}$ in $\mathbb{R}^m$
- Chaque sortie $\hat{y}_k$ est un neurone artificiel. Pour chaque sortie $\hat{y}_k$:
  - Transformation affine : $s_k = \mathbf{w_k}^T \mathbf{x} + b_k$
  - Activation non-linéaire σ : $\hat{y}_k = \sigma(s_k)$
- Paramètres de la transformation linéaire la sortie $\hat{y}_k$:
  - poids $\mathbf{w_k} = \left\{w_{1,k}, w_{2,k}, \ldots, w_{m,k}\right\} \in \mathbb{R}^m$
  - biais $b_k \in \mathbb{R}$

Perceptron en résumé

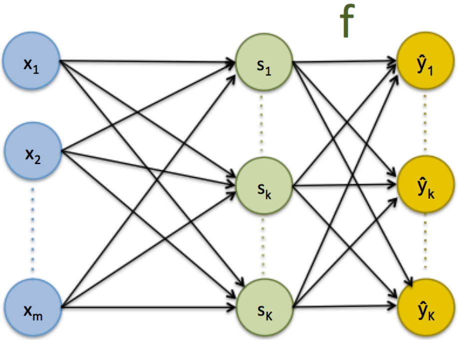
- Entrée $\mathbf{x} \in \mathbb{R}^m$ $(1 \times m)$, sortie $\hat{y}$: concaténation de k neurones
- Transformation affine ($\sim$ multiplication matricielle) : $\mathbf{s} = \mathbf{x}\mathbf{W} + \mathbf{b}$
  - avec $\mathbf{W}$ une matrice de poids de dimensions $m \times k$ – les colonnes sont les vecteurs $\mathbf{w_k}$,
  - et $\mathbf{b}$ le vecteur de biais – dimension $1 \times k$.
- Activation non-linéaire élément par élément (*pointwise*) : $\hat{\mathbf{y}} = \sigma(\mathbf{s})$



Application à la classification multi-classe

- Comment généraliser la sigmoïde au cas à plusieurs classes ?
  - on cherche f telle que $f(s_j) = P(j|\mathbf{x})$ (interprétation probabiliste)
  - contrainte : $\sum_{j=1}^{k} f(s_j) = 1$.
- **Activation softmax** :

$$\hat{y}_i = f(s_i) = \frac{e^{s_i}}{\sum_{j=1}^{k} e^{s_j}}$$



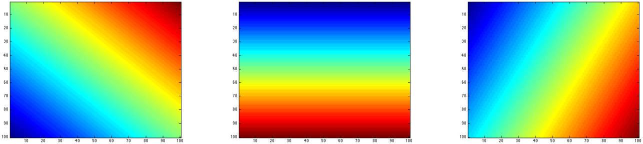
$\Rightarrow$ on appelle ce modèle la **régression logistique**.

Exemple jouet en 2D pour trois classes

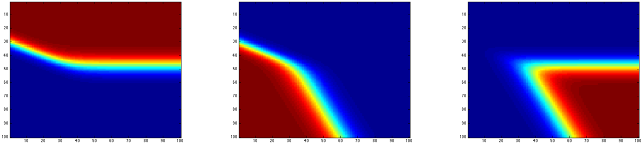
- $\mathbf{x} = \{x_1, x_2\} \in [-5;5] \times [-5;5]$, $\hat{y} \in \{0, 1, 2\}$ (3 classes)

$\mathbf{w_1} = [1;1],\ b_1 = -2$ $\mathbf{w_2} = [0;-1],\ b_2 = 1$ $\mathbf{w_3} = [1;-0.5],\ b_3 = 10$

Transformation affine : $\mathbf{s}_k = \mathbf{w_k}^T\mathbf{x} + b_k$



Softmax : $P(k|\mathbf{x}, \mathbf{W})$

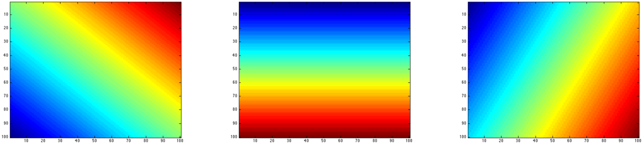


Exemple jouet en 2D pour trois classes

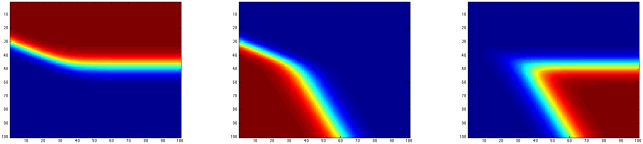
- $\mathbf{x} = \{x_1, x_2\} \in [-5; 5] \times [-5; 5], \qquad \hat{y} \in \{0, 1, 2\}$ (3 classes)

$\mathbf{w_1} = [1; 1],\ b_1 = -2 \qquad \mathbf{w_2} = [0; -1],\ b_2 = 1 \qquad \mathbf{w_3} = [1; -0.5],\ b_3 = 10$

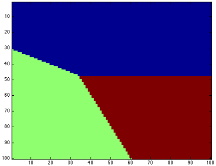
Transformation affine : $\mathbf{s}_k = \mathbf{w_k}^T \mathbf{x} + b_k$



Softmax : $P(k|\mathbf{x}, \mathbf{W})$



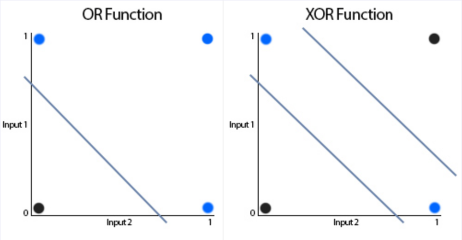
Classe prédite : $k^* = \arg\max_k P(k|\mathbf{x}, \mathbf{W})$



Limites du perceptron

Le problème du XOR

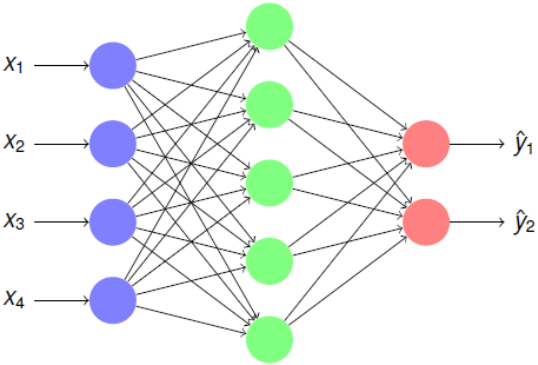
- Régression logistique (LR) : perceptron à une couche d'entrée et une couche de sortie
  - LR ne peut exprimer que des frontières de décisions linéaires
- **XOR :** (NON 1 ET 2) OU (NON 2 ET 1)
  - La frontière optimale pour XOR est non-linéaire.



⇒ constat pessimiste : le perceptron échoue sur un problème en apparence très simple…

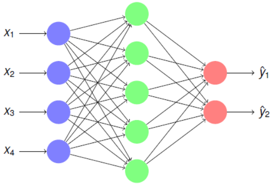
Au-delà des frontières linéaires

- La régression logistique est limitée aux frontières linéaires.LR : limited to linear boundaries
  - $\Rightarrow$ **Solution :** ajouter une couche de neurones !
- Entrée : $\mathbf{x} \in \mathbb{R}^m$, ici par exemple $m = 4$.
- Sortie : $\hat{\mathbf{y}} \in \mathbb{R}^k$ (k classes), par ex. $k = 2$.
- **Couche "cachée"** de taille l : $\mathbf{h} \in \mathbb{R}^l$, par ex. $l = 5$



Perceptron multi-couche

- **Couche cachée $\mathbf{h}$** : projection de l'entrée $\mathbf{x}$ vers un nouvel espace $\mathbb{R}^l$
  - $\mathbf{h}$ est une représentation intermédiaire de $\mathbf{x}$ qui sera ensuite utilisée pour la classification $\hat{\mathbf{y}}$: $\mathbf{h} = \phi(\mathbf{W}_s\mathbf{h} + \mathbf{b}_s)$ par la régression logistique.
- Réseau de neurones avec au moins une couche cachée : PMC, perceptron multi-couche (*multi-layer perceptron*, MLP)



Attention !

La non-linéarité de la fonction d'activation est indispensable pour le perceptron multi-couche :

- $\hat{\mathbf{y}} = \phi(\mathbf{W}_s\mathbf{h} + \mathbf{b}_s)$ et $\mathbf{h} = \phi(\mathbf{W}_e\mathbf{x} + \mathbf{b}_e)$

$\Rightarrow$ $\hat{\mathbf{y}} = \phi(\mathbf{W}_s\phi(\mathbf{W}_e\mathbf{x} + \mathbf{b}_e) + \mathbf{b}_s)$
  - Si $\phi = \mathrm{Id}$, alors $\hat{\mathbf{y}} = \phi(\mathbf{W}_s\mathbf{W}_e\mathbf{x} + \mathbf{W}_s\mathbf{b}_e + \mathbf{b}_s)$
  - équivalent à un perceptron classique avec $\mathbf{W} = \mathbf{W}_s\mathbf{W}_e$ et $\mathbf{b} = \mathbf{W}_s\mathbf{b}_e + \mathbf{b}_s$!

Garanties théoriques

- Quelle est l'expressivité des perceptrons multi-couche ?
  - Peut-on approcher n'importe quelle fonction à l'aide d'un PMC ?

Théorème d'approximation universelle

Soit f une fonction continue sur (des compacts de) $\mathbb{R}^d$. Alors, pour tout $\epsilon > 0$, il existe un perceptron $\hat{f} : x \rightarrow \mathbf{W}_s\phi(\mathbf{W} \cdot x + b)$ multi-couche à une couche cachée de nombre de neurones fini qui est une approximation de f à ϵ près, c'est-à-dire :

$$\max_{x \in \mathbb{R}^d} \| f(x) - \hat{f}(x) \| < \epsilon$$

si ϕ est une fonction d'activation non-polynomiale.

- Démontré pour les fonctions d'activation sigmoïde en 1989 Cybenko 1989
- Étendu à l'ensemble des perceptrons multi-couche en 1991 Hornik 1991
- D'autres résultats existent : profondeur arbitraire, largeur arbitraire, etc.

Limitations

- Une seule couche cachée suffit mais le nombre de neurones n'est pas borné.
- La démonstration prouve l'existence mais ne donne pas de méthode pour déterminer les paramètres $\mathbf{W}_s, \mathbf{W}$ et b.

Garanties théoriques

- Quelle est l'expressivité des perceptrons multi-couche ?
  - Peut-on approcher n'importe quelle fonction à l'aide d'un PMC ?

Théorème d'approximation universelle

Soit f une fonction continue sur (des compacts de) $\mathbb{R}^d$. Alors, pour tout $\epsilon > 0$, il existe un perceptron $\hat{f} : x \to \mathbf{W}_s \phi(\mathbf{W} \cdot x + b)$ multi-couche à une couche cachée de nombre de neurones fini qui est une approximation de f à ϵ près, c'est-à-dire :

$$\max_{x \in \mathbb{R}^d} \| f(x) - \hat{f}(x) \| < \epsilon$$

si ϕ est une fonction d'activation non-polynomiale.

- Démontré pour les fonctions d'activation sigmoïde en 1989 Cybenko 1989
- Étendu à l'ensemble des perceptrons multi-couche en 1991 Hornik 1991
- D'autres résultats existent : profondeur arbitraire, largeur arbitraire, etc.

Limitations

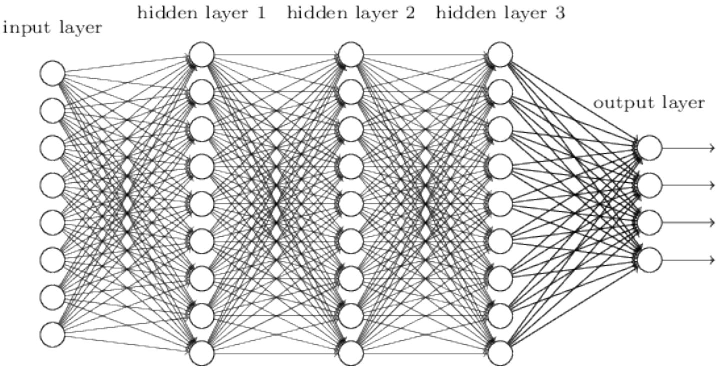
- Une seule couche cachée suffit mais le nombre de neurones n'est pas borné.
- La démonstration prouve l'existence mais ne donne pas de méthode pour déterminer les paramètres $\mathbf{W}_s, \mathbf{W}$ et b.

Réseaux de neurones profonds

- Augmenter le nombre de couche cachées : réseaux de neurones profonds (*deep neural networks*)
- Chaque couche $\mathbf{h}^l$ projette les activations de la couche précédente $\mathbf{h}^{l-1}$ dans un nouvel espace intermédiaire

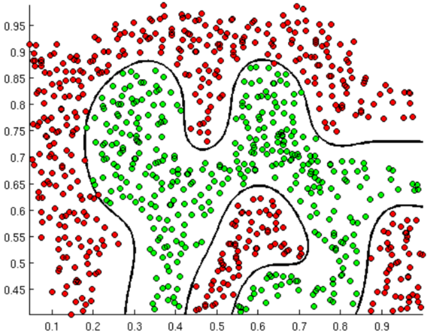
⇒ construction incrémentale d'un espace intermédiaire de représentation utile pour la tâche visée

l'apprentissage de représentation est la base de l'apprentissage profond !



En résumé

- Neurone artificiel : représentation grossière d'un neurone biologique
  - Poids des connexions synaptiques
  - Fonction de transfert non-linéaire
- Perceptron : une matrice de poids projetant la couche d'entrée vers la couche de sortie
  - limité à des frontières de décision linéaires
- Réseaux de neurones profonds : perceptrons multi-couche applicables aux problèmes dont les frontières de décision sont non-linéaires



$\Rightarrow$ comment trouver les paramètres du modèle (poids des connexions) ? $\rightarrow$ **apprentissage supervisé**

Plan du cours

Algorithme du gradient

Soit $f : \mathbb{R}^d \to \mathbb{R}$ une fonction réelle à d variables *différentiable*.

- On note $\nabla f(\mathbf{x})$ le *gradient* de f au point $\mathbf{x} = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$.

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_d} \right)$$

- Si $\nabla f(\mathbf{x}^*) = 0$, alors $\mathbf{x}^*$ est un *extremum local* de f.
    - Si f est convexe ou concave, alors c'est un *extremum global*.
- On cherche à *minimiser* f, c'est-à-dire trouver $\mathbf{x}^*$ tel que $f(\mathbf{x}^*) < f(\mathbf{x})$ pour $\mathbf{x} \in \mathbb{R}^d$.

Algorithme du gradient Cauchy 1847

Soit $\mathbf{x}^{(0)} \in \mathbb{R}^d$ un point de départ et $\epsilon > 0$. On définit la suite $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots$ telle que :

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \alpha_t \nabla f(\mathbf{x}^{(t+1)})$$

Si $\|\nabla f(\mathbf{x}^{(t)})\| \leq \epsilon$, on s'arrête.

- α_t peut être obtenu par divers moyens, comme une recherche linéaire. Dans notre cas, on fixera $\alpha_t = \alpha > 0$ constant.

Optimisation du *multi-layer perceptron* (MLP)

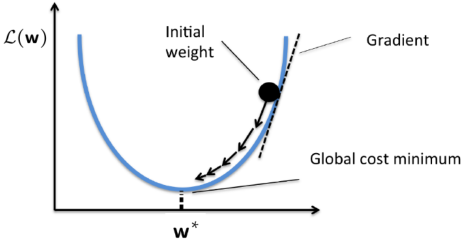
- Entrée $\mathbf{x} \in \mathbb{R}^m$, sortie $\mathbf{y} \in \mathbb{R}^p$
- Paramètres $\mathbf{w}$ (poids et biais du modèle)
- Le MLP est un modèle paramétrique $\mathbf{x} \Rightarrow \mathbf{y}$: $f_\mathbf{w}(\mathbf{x}_i) = \hat{\mathbf{y}}_i$
- Apprentissage **supervisé** : jeu d'entraînement annoté $\mathcal{A} = \{(\mathbf{x}_i, \mathbf{y}_i^*)\}_{i \in \{1,2,\ldots,N\}}$
  - Fonction de coût $\mathcal{L} = \sum_{i=1}^{N} \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i^*)$ entre la prédiction du modèle et l'annotation
- *Hypothèses :* les paramètres $\mathbf{w} \in \mathbb{R}^d$ sont des réels, $\mathcal{L}$ est différentiable.
- Gradient $\nabla_\mathbf{w} = \frac{\partial \mathcal{L}}{\partial \mathbf{w}}$: direction de plus forte pente dans l'espace des paramètres permettant de faire décroître le coût $\mathcal{L}$

Algorithme d'optimisation

Considérons le vecteur des paramètres $\mathbf{w} \in \mathbb{R}^d = (w_1, w_2, \ldots, w_d)$.
Le *gradient* du coût $\mathcal{L}$ par rapport à $\mathbf{w}$ est le vecteur des dérivées partielles du coût par rapport à chaque paramètre w_i :

$$\nabla_{\mathbf{w}}\mathcal{L} = \begin{pmatrix} \frac{\partial \mathcal{L}(\hat{\mathbf{y}},\mathbf{y}^*;\mathbf{w})}{\partial w_1} & \frac{\partial \mathcal{L}(\hat{\mathbf{y}},\mathbf{y}^*;\mathbf{w})}{\partial w_2} & \cdots & \frac{\partial \mathcal{L}(\hat{\mathbf{y}},\mathbf{y}^*;\mathbf{w})}{\partial w_d} \end{pmatrix}$$

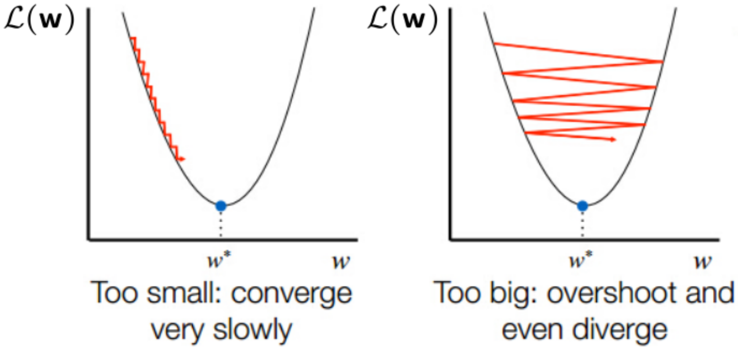
- Algorithme de la descente de gradient :
  1. Initialisation aléatoire des paramètres $\mathbf{w}$
  2. Mise à jour à l'itération t : $\boxed{\mathbf{w}_i^{(t+1)} = \mathbf{w}_i^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}_i}}$
  3. Répéter l'étape 2 jusqu'à convergence, par ex. $\|\nabla_{\mathbf{w}}\mathcal{L}\|^2 \approx 0$ (le coût cesse de décroître).



Pas d'apprentissage

Équation de mise à jour : $\boxed{\mathbf{w}_i^{(t+1)} = \mathbf{w}_i^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}_i}}$ η **pas d'apprentissage** (*learning rate*)

- **Peut-on garantir la convergence de l'algorithme vers un minimum ?** $\Rightarrow$ seulement pour une valeur de η "bien choisie".

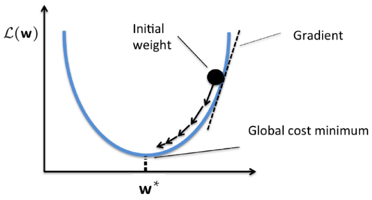


Minimum local ou global

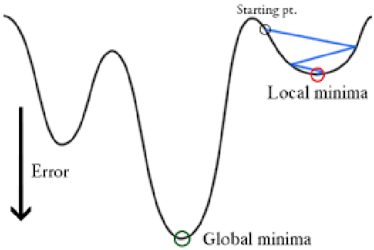
Équation de mise à jour : $\boxed{\mathbf{w}_i^{(t+1)} = \mathbf{w}_i^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}_i}}$

- **Le minimum trouvé est-il le meilleur (minimum global)?**

$\Rightarrow$ garanti seulement si la fonction $\mathcal{L}(\mathrm{w})$ à minimiser est **convexe**



(a) Fonction convexe



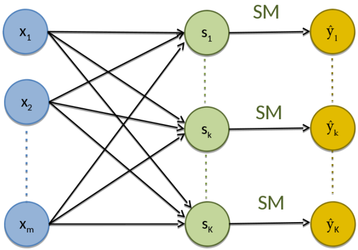
(b) Fonction non-convexe

- dans la quasi-totalité des cas, la fonction de coût utilisée n'est pas convexe par rapport aux paramètres du modèle...

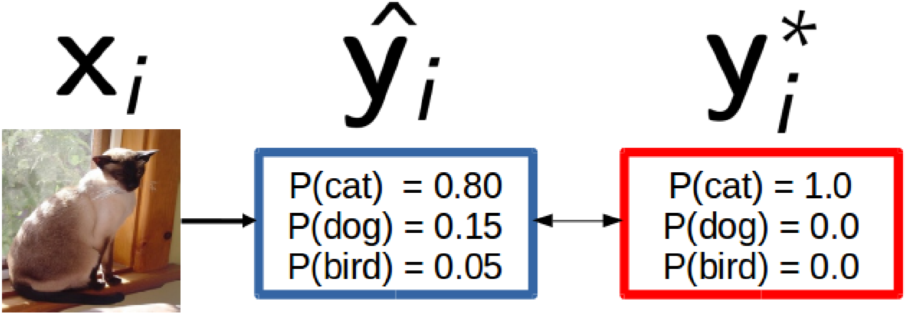
Fonctions de coût pour le perceptron

- Perceptron : $\hat{\mathbf{y}} = \phi\left(\mathbf{x}_i \mathbf{W} + \mathbf{b}\right)$
- Régression :
  - Fonction de coût : L2 (MSE) $\|\hat{\mathbf{y}} - \mathbf{y}\|^2$, L1 (MAE) $|\hat{\mathbf{y}} - \mathbf{y}|$, etc.
- Classification multi-classe à K classes (régression logistique) :
  - $\mathbf{y} \in \{1; 2; \ldots; K\}$
  - Activation softmax : $\phi(\mathbf{s}_k) = P(k|\mathbf{x_i}) = \frac{e^{\mathbf{s}_k}}{\sum_{k'=1}^{K} e^{\mathbf{s}_{k'}}}$
  - $\hat{\mathbf{y}}_i = \arg\max_k P(k|\mathbf{x}_i; \mathbf{W}, \mathbf{b})$
  - Fonction de coût : $\ell_{0/1}(\hat{\mathbf{y}}_i, \mathbf{y}_i^*) = \begin{cases} 1 & \text{si } \hat{\mathbf{y}}_i \neq \mathbf{y}_i^* \\ 0 & \text{sinon} \end{cases}$: **perte 0/1**

$\Rightarrow$ non-différentiable !



Régression logistique : encodage des variables



- Vérité terrain (étiquettes de supervision) $\mathbf{y}_i^* \in \{1, 2, \ldots, K\}$
- Encodage *one hot* pour chaque étiquette :

$$y_{c,i}^* = \begin{cases} 1 & \text{si } c \text{ est la classe de } \mathbf{x}_i \\ 0 & \text{sinon} \end{cases}$$

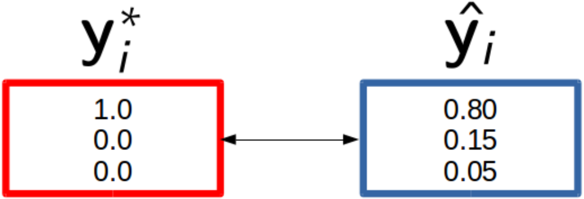
$$\mathbf{y}_i^* = \left(0, 0, \ldots, \underbrace{1}_{c^e\text{composante}}, \ldots, 0\right)$$

Régression logistique : fonction de coût

- Fonction de coût : entropie croisée (*cross-entropy*, CE) : ℓ_{CE}
- ℓ_{CE} : divergence de Kullback-Leiber entre la distribution de probabilité $\mathbf{y}_i^*$ de la vérité terrain et la distribution prédite $\widehat{\mathbf{y}}_i$

$$\ell_{CE}(\mathbf{y}_i^*, \widehat{\mathbf{y}}_i) = \mathrm{KL}(\mathbf{y}_i^*, \widehat{\mathbf{y}}_i) = -\sum_{c=1}^{K} \underbrace{y_{c,i}^*}_{= 0 \text{ sauf pour } c^*} \log(\hat{y}_{c,i}) = -\log(\hat{y}_{c^*,i})$$

- Attention ! La divergence KL (et donc l'entropie croisée) est asymétrique : $KL(\widehat{\mathbf{y}}_i, \mathbf{y}_i^*) \neq KL(\mathbf{y}_i^*, \widehat{\mathbf{y}}_i)$



$$KL(\mathbf{y}_i^*, \hat{\mathbf{y}}_i) = -log(\hat{y}_{c^*,i}) = -log(0.8) \approx 0.22$$

Entraînement de la régression logistique

- $\mathcal{L}_{CE}(\mathbf{W}, \mathbf{b}) = \frac{1}{N} \sum_{i=1}^{N} \ell_{CE}(\hat{\mathbf{y}}_i, \mathbf{y}_i^*) = -\frac{1}{N} \sum_{i=1}^{N} \log(\hat{y}_{c^*,i})$
- ℓ_{CE} est une borne supérieure de la perte $\ell_{0/1}$
  - minimiser $\ell_{CE} \Rightarrow$ minimiser la perte $\ell_{0/1}$
- ℓ_{CE} est différentiable $\Rightarrow$ **descente de gradient !**
  - ℓ_{CE} est convexe mais seulement par rapport à $\mathbf{y}$ (pas par rapport à $\mathbf{w}$)

Calcul du gradient

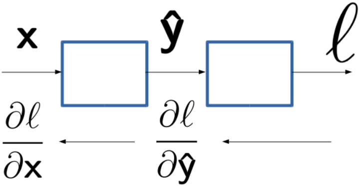
Descente de gradient : $\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} - \eta \frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{W}}$ et $\mathbf{b}^{(t+1)} = \mathbf{b}^{(t)} - \eta \frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{b}}$

- Principal problème : calculer $\frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{W}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \ell_{CE}}{\partial \mathbf{W}}$
- Propriété centrale : *chain rule* (dérivation des fonctions composées)

$$\frac{\partial \mathbf{x}}{\partial z} = \frac{\partial \mathbf{x}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial z}$$

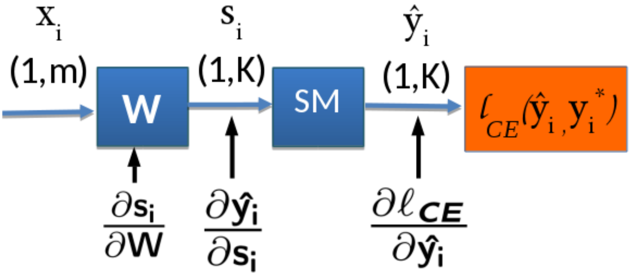
- rétropropagation du gradient de l'erreur par rapport à $\mathbf{y}$ en erreur par rapport à $\mathbf{w}$

Chain rule



$$\boxed{\frac{\partial \ell}{\partial \mathbf{x}} = \frac{\partial \ell}{\partial \widehat{\mathbf{y}}} \frac{\partial \widehat{\mathbf{y}}}{\partial \mathbf{x}}}$$

- Régression logistique : $\boxed{\frac{\partial \ell_{CE}}{\partial \mathbf{W}} = \frac{\partial \ell_{CE}}{\partial \widehat{\mathbf{y}_i}} \frac{\partial \widehat{\mathbf{y}_i}}{\partial \mathbf{s}_i} \frac{\partial \mathbf{s}_i}{\partial \mathbf{W}}}$

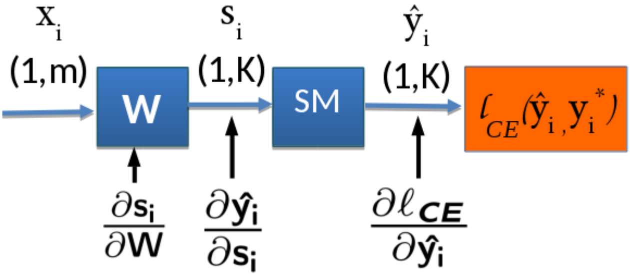


Régression logistique : calcul des gradients (1/2)

$$\frac{\partial \ell_{CE}}{\partial \mathbf{W}} = \frac{\partial \ell_{CE}}{\partial \widehat{\mathbf{y}}_\mathbf{i}} \frac{\partial \widehat{\mathbf{y}}_\mathbf{i}}{\partial \mathbf{s}_\mathbf{i}} \frac{\partial \mathbf{s}_\mathbf{i}}{\partial \mathbf{W}}, \quad \ell_{CE}(\hat{\mathbf{y}}_i, \mathbf{y}_i^*) = -log(\hat{y}_{c^*,i})$$

Mise à jour pour 1 example :

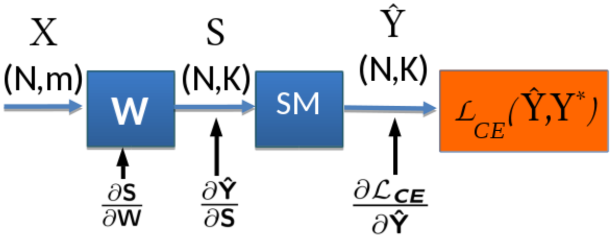
- $\boxed{\frac{\partial \ell_{CE}}{\partial \widehat{\mathbf{y}}_\mathbf{i}} = \frac{-1}{\hat{y}_{c^*,i}} = \frac{-1}{\widehat{\mathbf{y}}_\mathbf{i}} \odot \delta_{\mathbf{c},\mathbf{c}^*}}$ avec $\delta_{c,c^*} = (0, \ldots, \underbrace{1}_{c^*}, \ldots, 0)$
- $\boxed{\frac{\partial \ell_{CE}}{\partial \mathbf{s}_\mathbf{i}} = \widehat{\mathbf{y}}_\mathbf{i} - \mathbf{y}_\mathbf{i}^* = \delta_\mathbf{i}^\mathbf{y}}$
- $\boxed{\frac{\partial \ell_{CE}}{\partial \mathbf{W}} = \mathbf{x_i}^T \delta_\mathbf{i}^\mathbf{y}}$



Régression logistique : calcul des gradients (2/2)

Pour l'ensemble du jeu de données $\mathbf{X}$ ($N \times m$), étiquettes $\mathbf{Y}^*$ et prédictions $\widehat{\mathbf{Y}}$ ($N \times K$) :

- $\mathcal{L}_{CE}(\mathbf{W}, \mathbf{b}) = -\frac{1}{N}\sum_{i=1}^{N} log(\hat{y}_{c^*,i})$, $\frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}_{CE}}{\partial \widehat{\mathbf{Y}}} \frac{\partial \widehat{\mathbf{Y}}}{\partial \mathbf{S}} \frac{\partial \mathbf{S}}{\partial \mathbf{W}}$
- $\boxed{\frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{s}} = \widehat{\mathbf{Y}} - \mathbf{Y}^* = \Delta^y}$
- $\boxed{\frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{W}} = \mathbf{X}^T \Delta^y}$



Entraînement du modèle

- La fonction de coût est calculée sur tout le jeu de données d'entraînement :

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \ell \left(\hat{\mathbf{y}}_i, \mathbf{y}_i^*; \mathbf{w}, \mathbf{b} \right)$$

- Optimisation par descente de gradient :

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}} \left(\mathbf{w}^{(t)} \right)$$

- La complexité du calcul du gradient $\nabla_{\mathbf{w}}^{(t)} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i^*)}{\partial \mathbf{w}} \left(\mathbf{w}^{(t)} \right)$ croît linéairement avec :
  - la dimensionalité de $\mathbf{w}$ (nombre de paramètres du modèle),
  - N, la taille du jeu de données (nombre d'exemples d'apprentissage).

$\Rightarrow$ **coûteux, même pour des modèles de dimensionalité modérée et des jeux de données de taille moyenne !**

Descente de gradient stochastique

- **Solution :** approximation du vrai gradient $\nabla_{\mathbf{w}}^{(t)} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i^*)}{\partial \mathbf{w}} \left(\mathbf{w}^{(t)}\right)$ sur un échantillon d'exemples (un *batch* ou "lot")

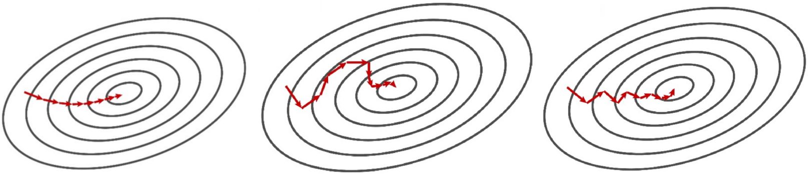
$\Rightarrow$ **Stochastic Gradient Descent (SGD)**

- Version *online* : mise à jour à partir d'un seul exemple

$$\nabla_{\mathbf{w}}^{(t)} \approx \frac{\partial \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i^*)}{\partial \mathbf{w}} \left(\mathbf{w}^{(t)}\right)$$

- Version par mini-batch : mise à jour sur $B << N$ exemples :

$$\nabla_{\mathbf{w}}^{(t)} \approx \frac{1}{B}\sum_{i=1}^{B} \frac{\partial \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i^*)}{\partial \mathbf{w}} \left(\mathbf{w}^{(t)}\right)$$



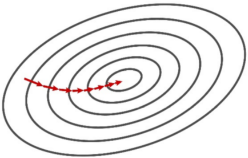
(a) Gradient complet (b) SGD (online) (c) SGD (mini-batch)

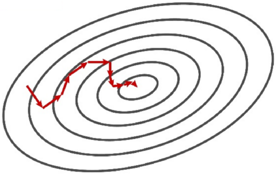
Avantages et inconvénients

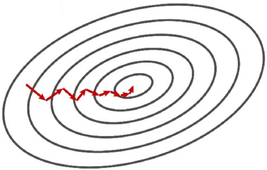
La descente de gradient stochastique produit une approximation du véritable gradient $\nabla_{\mathbf{w}}$.

- estimation bruitée, pouvant envoyer une direction incorrecte (pente forte pour un exemple mais pas pour les autres),
- sensibilité aux *outliers* pour la version en ligne,
+ mises à jour des poids plus nombreuses car itérations moins coûteuses : $\times N$ mises à jour (online), $\times \frac{N}{B}$ mises à jour (mini-batch)
+ à nombre de calculs égal, convergence plus rapide

La SGD est incontournable pour la convergence des modèles profonds sur des jeux de données massifs.


(a) Gradient complet

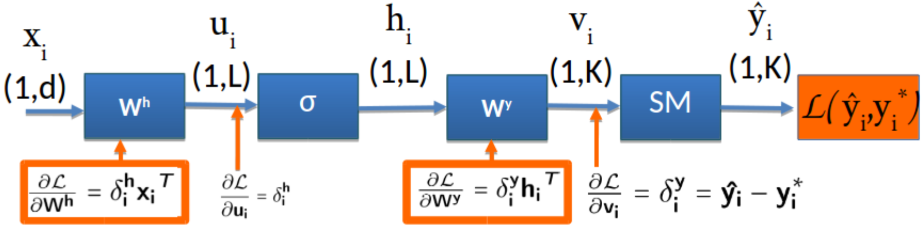

(b) SGD (online)


(c) SGD (mini-batch)

Entraînement du perceptron : rétropropagation

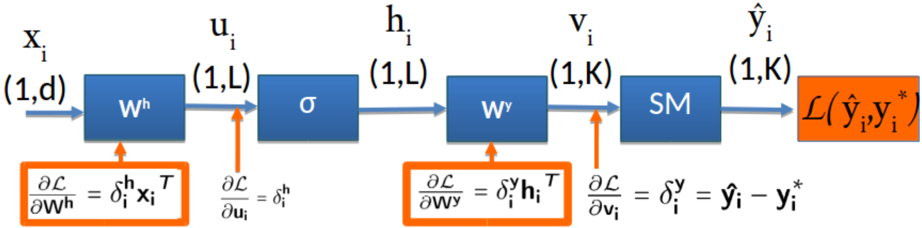
- Passer de la régression logistique au perceptron multi-couche implique l'ajout d'une couche cachée (+ sigmoïde)
- Entraînement : apprendre les paramètres $\mathbf{W}^y$ et $\mathbf{W}^h$ (+ bias) par **rétropropagation**

$\boxed{\frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{W}^y} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \ell_{CE}}{\partial \mathbf{W}^y}}$ et $\boxed{\frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{W}^h} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \ell_{CE}}{\partial \mathbf{W}^h}}$



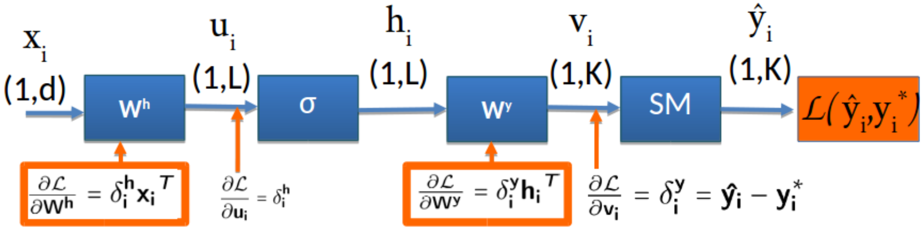
- La dernière couche est équivalente à une régression logistique.
- Couche cachée : $\frac{\partial \ell_{CE}}{\partial \mathbf{W}^h} = \mathbf{x_i}^T \frac{\partial \ell_{CE}}{\partial \mathbf{u_i}} \Rightarrow$ **calcul de** $\frac{\partial \ell_{CE}}{\partial \mathbf{u_i}} = \delta_i^{\mathbf{h}}$

Rétropropagation du gradient



- Coût pour un seul exemple : $\mathcal{L}_{\mathbf{W}}(\widehat{\mathbf{y}}_\mathrm{i}, \mathbf{y}_\mathrm{i}^*) = -\log(\hat{y}_{c^*,i})$
- $\frac{\partial \mathcal{L}}{\partial \mathbf{v_i}} = \delta_\mathrm{i}^\mathrm{y} = \widehat{\mathbf{y}}_\mathrm{i} - \mathbf{y}_\mathrm{i}^*$ et $\frac{\partial \mathcal{L}}{\partial \mathbf{W^y}} = \delta_\mathrm{i}^\mathrm{y} {\mathbf{h}_\mathrm{i}}^T$
- **Pour revenir en arrière** : on calcule $\frac{\partial \mathcal{L}}{\partial \mathbf{u_i}} = \delta_\mathrm{i}^\mathrm{h}$
- Si on connaît $\delta_\mathrm{i}^\mathrm{h} \Rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{W^h}} = \delta_\mathrm{i}^\mathrm{h} {\mathbf{x}_\mathrm{i}}^T \sim$ régression logistique

Rétropropagation dans le perceptron



- Le calcul de $\frac{\partial \ell_{CE}}{\partial \mathbf{u_i}} = \delta_i^{\mathbf{h}} \Rightarrow$ s'obtient par *chain rule* :

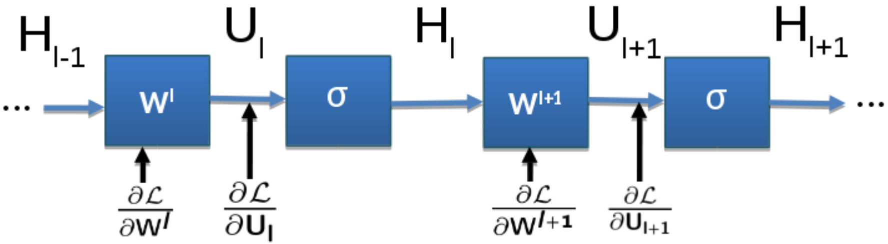
$$\frac{\partial \ell_{CE}}{\partial \mathbf{u_i}} = \frac{\partial \ell_{CE}}{\partial \mathbf{v_i}} \frac{\partial \mathbf{v_i}}{\partial \mathbf{h_i}} \frac{\partial \mathbf{h_i}}{\partial \mathbf{u_i}}$$

- ce qui permet d'obtenir :

$$\frac{\partial \ell_{CE}}{\partial \mathbf{u_i}} = \delta_i^{\mathbf{h}} = {\delta_i^{\mathbf{y}}}^T \mathbf{W^y} \odot \sigma'(\mathbf{h_i}) = {\delta_i^{\mathbf{y}}}^T \mathbf{W^y} \odot (\mathbf{h_i} \odot (1 - \mathbf{h_i}))$$

Rétropropagation dans les réseaux profonds

- Perceptron multi-couche : ajout d'encore plus de couches
- Rétropropagation $\sim$ Perceptron : **en supposant que l'on connaise** $\frac{\partial \mathcal{L}}{\partial \mathbf{U_{l+1}}} = \Delta^{l+1}$
  - $\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{l+1}} = \mathbf{H_l}^T \Delta^{l+1}}$
  - $\boxed{\text{Calcul de } \frac{\partial \mathcal{L}}{\partial \mathbf{U_l}} = \Delta^l}$ $(= {\Delta^{l+1}}^T \mathbf{W^{l+1}} \odot \mathbf{H_l} \odot (1 - \mathbf{H_l})$ pour la sigmoïde)
  - $\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{W}^l} = \mathbf{H_{l-1}}^T \Delta^{h_l}}$



Plan du cours

La rétropropagation en résumé

- **Rétropropagation (*backpropagation*)** : solution pour l'entraînement de bout en bout de tous les paramètres des réseaux profonds
  - Algorithme clé de l'apprentissage profond !

Historique de la rétropropagation

- Dérivation des fonctions composées (Leibniz, 1676)
- Optimisation par descente de gradient (Cauchy 1847)
- Premières applications de la rétropropagation (programmation dynamique, contrôle Kelley 1960 ; Bryson et Ho 1969)
- Formalisation de la rétropropagation pour les réseaux de neurones (Linnainmaa 1970 ; Werbos 1975)
- Application aux réseaux de neurones profonds (Rumelhart, Hinton et Williams 1986 ; LeCun 1988)

Mais y a-t-il des inconvénients à utiliser la rétropropagation ?

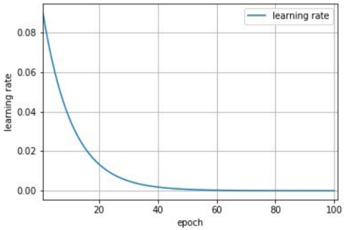
Optimisation : politique du pas d'apprentissage

- Mise à jour par descente de gradient : $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}}^{(t)}$
- Comment choisir le pas d'apprentissage η ?

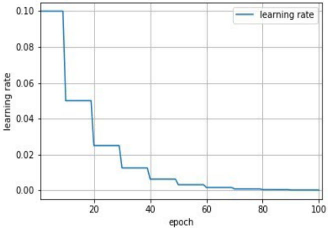
Politiques d'adaptation du pas d'apprentissage

En général, l'heurisiste consiste à faire décroître η durant l'apprentissage (*learning rate "decay"*).

- Décroissance inversement proportionnelle à l'itération : $\eta_t = \frac{\eta_0}{1+r \cdot t}$, r vitesse de décroissance
- Décroissance exponentielle : $\eta_t = \eta_0 \cdot e^{-\lambda t}$
- Décroissance en escalier : $\eta_t = \eta_0 \cdot r^{\frac{t}{t_u}}$ ...



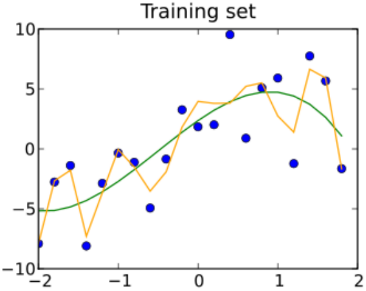
Décroissance exponentielle ($\eta_0 = 0.1$, $\lambda = 0.1$)



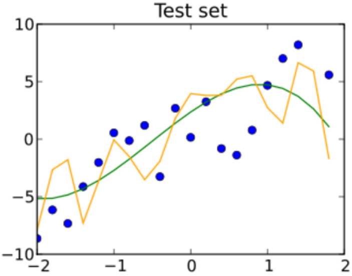
Décroissance *step* ($\eta_0 = 0.1$, $r = 0.5$, $t_u = 10$)

Optimisation, généralisation et sur-apprentissage

- **Apprentissage :** minimisation d'une fonction de coût $\mathcal{L}$ sur un jeu de données $\Longrightarrow$ risque empirique
  - Jeu d'apprentissage : ensemble d'échantillons représentatif de la distribution des données **et** des étiquettes
  - Objectif : apprendre une fonction de prédiction avec une erreur faible **sur la distribution (inconnue) des données réelles** $\Longrightarrow$ risque espéré



$\mathcal{L}_{\text{train}} = 4$, $\mathcal{L}_{\text{train}} = 9$



$\mathcal{L}_{\text{test}} = 15$, $\mathcal{L}_{\text{test}} = 13$

Optimisation $\neq$ apprentissage ! $\Longrightarrow$ **sur-apprentissage $\neq$ généralisation**

Régularisation

Régularisation

Réduire la capacité du modèle pour réduire l'écart entre performances entraînement/test.

- Avec un modèle de suffisamment grande capacité, améliore la généralisation et donc les performances en test.

- Régularisation structurelle : ajout d'une contrainte sur les poids pour les forcer à suivre un a priori

$$\mathcal{L}_r(\mathbf{w}) = \mathcal{L}(\mathbf{y}; \mathbf{y}^*; \mathbf{w}) + \alpha R(\mathbf{w})$$

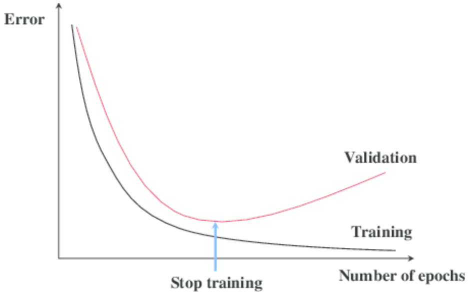
- *Weight decay* : régularisation L_2 pour pénaliser les poids de norme élevée :

$$R(\mathbf{w}) = ||\mathbf{w}||^2$$

  - Utilisation courante pour les réseaux de neurones
  - Justifications théoriques et bornes de généralisation dans le cas des SVM
- Autres possibilités pour $R(\mathbf{w})$: régularisation L_1, Dropout, etc.

Régularisation et hyperparamètres

- Hyperparamètres pour les réseaux de neurones :
  - Hyperparamètres d'optimisation : pas d'apprentissage (et évolution), nombre d'itérations, régularisation, etc.
  - Hyperparamètres d'architecture : nombre de couches, nombre de neurones, choix de la non-linéarité, etc.
- Le réglage des hyperparamètres permet d'ajuster la capacité du modèle et d'améliorer la généralisation :
  - L'utilisation d'un jeu de validation est cruciale !



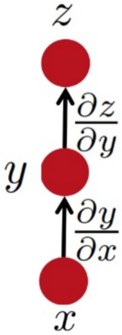
Plan du cours

Optimisation des réseaux profonds

- Optimisation des réseaux profonds : descente de gradient sur la fonction de coût $\mathcal{L}$

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta \nabla \mathcal{L}\left(\mathbf{w}^{t}\right)$$

- Rétropropagation de l'erreur : algorithme pour calculer $\nabla \mathcal{L}\left(\mathbf{w}^{t}\right)$ dans un réseau de neurones
  - Expression analytique du gradient par *chain rule*
  - mais le calcul du gradient peut être difficile à réaliser efficacement



Deux sources de difficultés

1. Problèmes d'**optimisation numérique**
2. **Différenciation automatique** (*automatic differenciation*)

Problèmes d'optimisation numérique

- L'accumulation des approximations (arrondis) pose problème.
- *Underflow* : $x \approx 0$ ou $x = 0$ provoque des comportements différents
  - Division par 0 $\Rightarrow$ indéterminé (*not a number*)
- *Overflow* : grandes valeurs de $x > 0$ ou $x < 0 \Rightarrow$ *not a number*

Exemple : softmax sur $\mathbf{x} = \{x_1, x_2, ..., x_K\}$

$s(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$

- Que se passe-t-il si $x_i = C$ pour tout i?
- En théorie, $s(\mathbf{x})_i = \frac{1}{K}$ quel que soit i, mais en pratique :
  - $C \to -\infty$, $e^C \to 0$, division par 0, *not a number*! (underflow)
  - $C \to +\infty$, $e^C \to +\infty$, *not a number*! (overflow)

Problèmes d'optimisation numérique

- L'accumulation des approximations (arrondis) pose problème.
- *Underflow* : $x \approx 0$ ou $x = 0$ provoque des comportements différents
  - Division par 0 $\Rightarrow$ indéterminé (*not a number*)
- *Overflow* : grandes valeurs de $x > 0$ ou $x < 0$ $\Rightarrow$ *not a number*

Exemple : softmax sur $\mathbf{x} = \{x_1, x_2, ..., x_K\}$

$s(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$

- Que se passe-t-il si $x_i = C$ pour tout i ?
- En théorie, $s(\mathbf{x})_i = \frac{1}{K}$ quel que soit i, mais en pratique :
  - $C \to -\infty$, $e^C \to 0$, division par 0, *not a number* ! (underflow)
  - $C \to +\infty$, $e^C \to +\infty$, *not a number* ! (overflow)

Problèmes d'optimisation numérique

- L'accumulation des approximations (arrondis) pose problème.
- *Underflow* : $x \approx 0$ ou $x = 0$ provoque des comportements différents
  - Division par 0 $\Rightarrow$ indéterminé (*not a number*)
- *Overflow* : grandes valeurs de $x > 0$ ou $x < 0 \Rightarrow$ *not a number*

Exemple : softmax sur $\mathbf{x} = \{x_1, x_2, ..., x_K\}$

$s(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$

- Que se passe-t-il si $x_i = C$ pour tout i ?
- En théorie, $s(\mathbf{x})_i = \frac{1}{K}$ quel que soit i, mais en pratique :
  - $C \to -\infty$, $e^C \to 0$, division par 0, *not a number* ! (underflow)
  - $C \to +\infty$, $e^C \to +\infty$, *not a number* ! (overflow)

Problèmes d'optimisation numérique

- L'accumulation des approximations (arrondis) pose problème.
- *Underflow* : $x \approx 0$ ou $x = 0$ provoque des comportements différents
  - Division par 0 $\Rightarrow$ indéterminé (*not a number*)
- *Overflow* : grandes valeurs de $x > 0$ ou $x < 0 \Rightarrow$ *not a number*

Exemple : softmax sur $\mathbf{x} = \{x_1, x_2, ..., x_K\}$

$s(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$

- Que se passe-t-il si $x_i = C$ pour tout i ?
- En théorie, $s(\mathbf{x})_i = \frac{1}{K}$ quel que soit i, mais en pratique :
  - $C \to -\infty, e^C \to 0$, division par 0, *not a number* ! (underflow)
  - $C \to +\infty, e^C \to +\infty$, *not a number* ! (overflow)

Problèmes d'optimisation numérique

- L'accumulation des approximations (arrondis) pose problème.
- *Underflow* : $x \approx 0$ ou $x = 0$ provoque des comportements différents
  - Division par 0 $\Rightarrow$ indéterminé (*not a number*)
- *Overflow* : grandes valeurs de $x > 0$ ou $x < 0 \Rightarrow$ *not a number*

Exemple : softmax sur $\mathbf{x} = \{x_1, x_2, ..., x_K\}$

$s(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$

- Que se passe-t-il si $x_i = C$ pour tout i ?
- En théorie, $s(\mathbf{x})_i = \frac{1}{K}$ quel que soit i, mais en pratique :
  - $C \to -\infty, e^C \to 0$, division par 0, *not a number* ! (underflow)
  - $C \to +\infty, e^C \to +\infty$, *not a number* ! (overflow)

Problèmes d'optimisation numérique

- L'accumulation des approximations (arrondis) pose problème.
- *Underflow* : $x \approx 0$ ou $x = 0$ provoque des comportements différents
  - Division par 0 $\Rightarrow$ indéterminé (*not a number*)
- *Overflow* : grandes valeurs de $x > 0$ ou $x < 0$ $\Rightarrow$ *not a number*

Exemple : softmax sur $\mathbf{x} = \{x_1, x_2, ..., x_K\}$

$s(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$

- Que se passe-t-il si $x_i = C$ pour tout i ?
- En théorie, $s(\mathbf{x})_i = \frac{1}{K}$ quel que soit i, mais en pratique :
  - $C \to -\infty, e^C \to 0$, division par 0, *not a number* ! (underflow)
  - $C \to +\infty, e^C \to +\infty$, *not a number* ! (overflow)

Une solution

Ajouter une stabilisation numérique au dénominateur :

$$\mathbf{z} = \mathbf{x} - \max_i(x_i) \Rightarrow s(\mathbf{z}) = s(\mathbf{x})$$

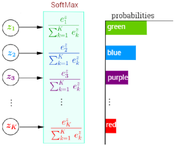
- $\max_i(e^{z_i}) = 1 \Rightarrow$ pas d'overflow
- $\max_i(e^{z_i}) = 1 \Rightarrow$ pas d'underflow

Cas de l'entropie croisée

- $s(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$
- $\mathbf{z} = \mathbf{x} - \max_i(x_i) \Rightarrow s(\mathbf{z}) = s(\mathbf{x})$

Que se passe-t-il pour le calcul de $\log[s(\mathbf{z})]$, par exemple dans l'entropie croisée ?

- $s(\mathbf{z}) = 0$ (underflow) $\Rightarrow \log[s(\mathbf{z})] \rightarrow -\infty$: *not a number* !



Solution : stabiliser le log

$$log[s(\mathbf{x})_i] = x_i - \log\left[\sum_{j=1}^{K} e^{x_j}\right]$$

- À nouveau, on substitue : $\mathbf{z} = \mathbf{x} - \max_i(x_i)$
- $\log[s(\mathbf{z})_i] = \log[s(\mathbf{x})_i]$?

$\Rightarrow$ underflow, overflow ?

Calcul des dérivées partielles

Plusieurs méthodes permettent de calculer les dérivées de façon automatique :

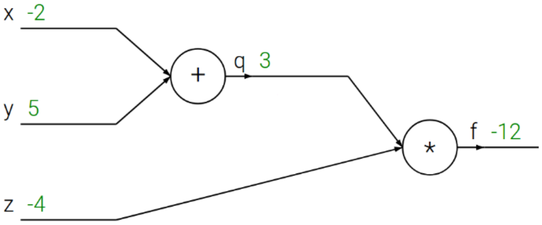
- Approximation numérique : $f'(x) \approx \frac{f(x+h) * f(x)}{h}$ (méthode des éléments finis)
  - $\oplus$ Ne nécessite aucune connaissance sur f
  - $\ominus$ Approximation, stabilité numérique (underflow/overflow)
  - $\ominus$ Requiert d'évaluer plusieurs fois f
- Dérivation symbolique ($\sim$ Mathematica)
  - $\ominus$ Expressions qui deviennent rapidement complexes, termes redondants $\Rightarrow$ faible efficacité

$f(x)$	$f'(x)$	$f'(x)$
$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$

- **Différenciation automatique**
  - Alterne entre dérivation symboliques et étapes de simplification
  - Différenciation symbolique au niveau des opérations élémentaires
  - Simplification en conservant les résultats numériques intermédiaires

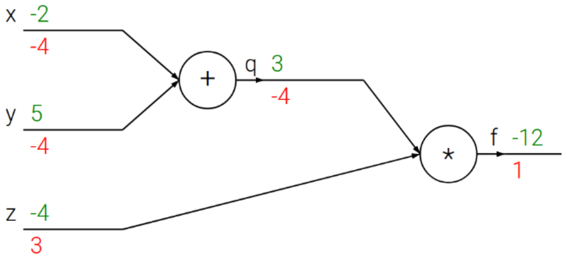
Automatic Differentiation (AD)

- Définition d'un **graphe de calcul** : abstraction pour représenter les séquences d'opérations à inverser pour rétropropager le gradient
- Exemple :[1] : $f(x, y, z) = (x + y) * z$
- Objectif : calculer les valeurs numériques de $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ et $\frac{\partial f}{\partial z}$
- Supposons que $x = -2$, $y = 5$, $z = -4$, propagation avant $\Rightarrow$ $q = 3$, $f = -12$



Rétropropagation et graphe de calcul

- $\frac{\partial f}{\partial f} = 1$, rétropropagation pour déterminer $\Rightarrow \frac{\partial f}{\partial q}, \frac{\partial f}{\partial z}$
- $f = q \star z \Rightarrow \frac{\partial f}{\partial q} = z = -4, \boxed{\frac{\partial f}{\partial z} = q = 3}$, rétropropagation pour $\Rightarrow \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}$
- $q = x + y \Rightarrow \boxed{\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = -4 \star 1 = -4}\ \boxed{\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = -4 \star 1 = -4}$



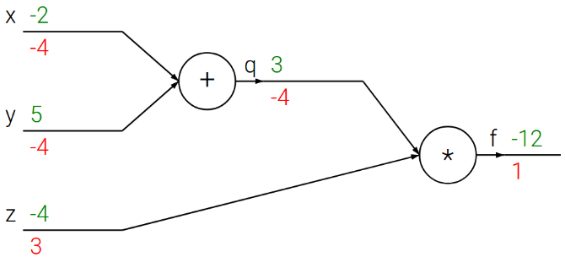
Différenciation automatique : *forward pass, backward pass*

- Différenciation automatique dans le graphe de calcul : passe avant (*forward*) puis différenciation automatique en passe arrière (*backward*)
  - *Backward pass* : calcul récursif des dérivées de haut en bas
  - **Programmation dynamique** : gain d'efficacité important par rapport à la différenciation

Une implémentation différent possible est l'autodifférenciation en passe avant :

1. $\frac{\partial x}{\partial x} = 1,$
2. $\frac{\partial q}{\partial x} = \frac{\partial q}{\partial x} \cdot \frac{\partial x}{\partial x} = 1,$
3. $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x} \cdot \frac{\partial x}{\partial x} = -4$

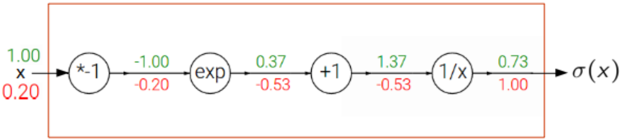
En pratique, pour un graphe à N nœuds, il faut N passes avant pour la *forward* AD, contre 1 passe avant + 1 passe arrière pour la *backward* AD (*backprop*).



Quelles opérations implémenter ?

- La différenciation symbolique est implémentée seulement au niveau des opérations "atomiques" :
  - Les opérateurs arithmétiques binaires : $+, -, \times, /$, etc.
  - Les fonctions "élémentaires" : $\exp, \log, \cos, \sin$, etc.
- Les fonctions dont les dérivées sont connues, avec une expression analytique dont le comportement numérique est stable :
  - Sigmoïde : $\sigma(x) = \frac{1}{1+e^{-x}} \implies \sigma'(x) = \sigma(x)\left[1 - \sigma(x)\right]$

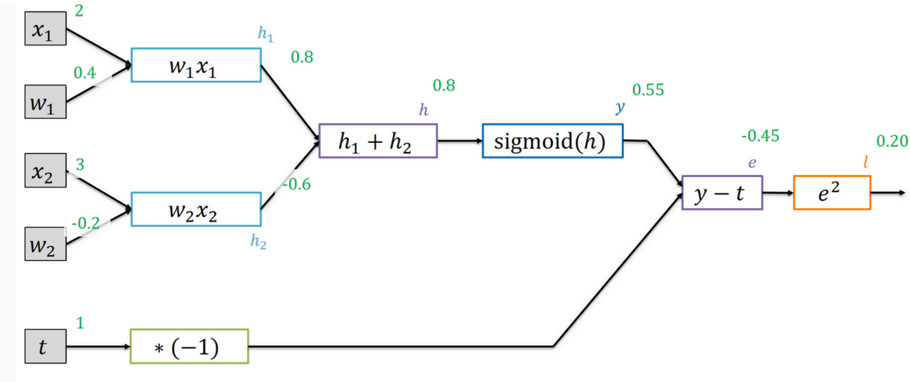
$$x = 1.0 \Rightarrow \sigma(x) = 0.73, \sigma'(x) = 0.2$$



Le graphe de calcul est remplaçable par un seul "bloc"dont l'inverse est $\sigma'(x)$.

Mise en œuvre pour un réseau de neurones

$$\mathbf{h} = f(\mathbf{x}, \mathbf{W})$$



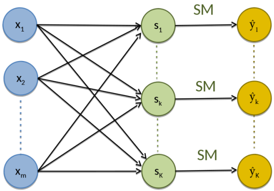
$\Rightarrow$ *Backward* : stockage des valeurs numériques des gradients $\frac{\partial \mathbf{h}}{\partial \mathbf{W}}$, $\frac{\partial \mathbf{h}}{\partial \mathbf{x}}$

Ressources logicielles pour l'apprentissage profond

- Fonctionnalités indispensables :
  - Construction du graphe de calcul et auto-différenciation (*autograd*)
  - Exécution transparente sur CPU ou GPU (pas de programmation bas niveau)
- De nombreuses bibliothèques disponibles : MatConvNet (Matlab), Caffe/Caffe2 (C++/Python), Torch (Lua/Python/C++), Theano (Python), TensorFlow (Python), JAX (Python), MXNet (C++/Java), etc.
- Actuellement, les deux principales bibliothèques logicielles sont :
  1. TensorFlow/Keras (soutenu par Google)
  2. PyTorch (soutenu par Facebook)

Keras : implémentation de la régression logistique

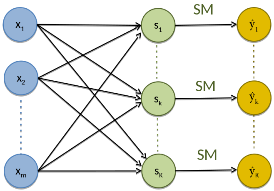
- Entrée : $\mathbf{x}_i \in \mathbb{R}^d$, $d = 784$, étiquette $\mathbf{y}_i$ à $K = 10$ classes.
- Activations : $\mathbf{s}_i = \mathbf{x}_i \mathbf{W} + \mathbf{b}$
- Sortie (softmax) : $\hat{\mathbf{y}}_k = e^{\mathbf{s}_k} / (\sum_{k'=1}^{K} e^{\mathbf{s}_{k'}})$
- Fonction de coût : entropie croisée $= \frac{1}{N} \sum_{i=1}^{N} \ell_{CE}(\hat{\mathbf{y}}_i, \mathbf{y}_i^*)$
- Nombre de paramètres : $784 \times 10 + 10 = 7850$




```
from tensorflow import keras
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
# Pré-traitement et normalisation
X_train, X_test = X_train.reshape(-1, 784), X_test.reshape(-1, 784)
X_train, X_test = X_train / 255., X_test / 255.
# Encodage one_hot des vecteurs de classe
Y_train, Y_test = keras.utils.to_categorical(y_train, 10), keras.utils.to_categorical(y_test, 10)
```

Keras : implémentation de la régression logistique

- Entrée : $\mathbf{x}_i \in \mathbb{R}^d$, $d = 784$, étiquette $\mathbf{y}_i$ à $K = 10$ classes.
- Activations : $\mathbf{s}_i = \mathbf{x}_i \mathbf{W} + \mathbf{b}$
- Sortie (softmax) : $\hat{\mathbf{y}}_k = e^{\mathbf{s}_k} / (\sum_{k'=1}^{K} e^{\mathbf{s}_{k'}})$
- Fonction de coût : entropie croisée $= \frac{1}{N} \sum_{i=1}^{N} \ell_{CE}(\hat{\mathbf{y}}_i, \mathbf{y}_i^*)$
- Nombre de paramètres : $784 \times 10 + 10 = 7850$

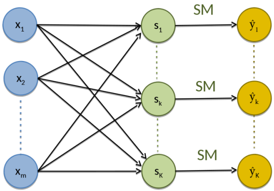



```
from tensorflow import keras
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
# Pré-traitement et normalisation
X_train, X_test = X_train.reshape(-1, 784), X_test.reshape(-1, 784)
X_train, X_test = X_train / 255., X_test / 255.
# Encodage one_hot des vecteurs de classe
Y_train, Y_test = keras.utils.to_categorical(y_train, 10), keras.utils.to_categorical(y_test, 10)
```

```
# Instancie un modèle vide
from tensorflow.keras import Sequential
model = Sequential()
# Ajoute les couches au modèle
from tensorflow.keras.layers import Dense, Activation
model.add(Dense(10, input_dim=784, name='fc1'))
```

Keras : implémentation de la régression logistique

- Entrée : $\mathbf{x}_i \in \mathbb{R}^d$, $d = 784$, étiquette $\mathbf{y}_i$ à $K = 10$ classes.
- Activations : $\mathbf{s}_i = \mathbf{x}_i\mathbf{W} + \mathbf{b}$
- Sortie (softmax) : $\hat{\mathbf{y}}_k = e^{\mathbf{s}_k} / (\sum_{k'=1}^{K} e^{\mathbf{s}_{k'}})$
- Fonction de coût : entropie croisée $= \frac{1}{N}\sum_{i=1}^{N} \ell_{CE}(\hat{\mathbf{y}}_i, \mathbf{y}_i^*)$
- Nombre de paramètres : $784 \times 10 + 10 = 7850$




```python
from tensorflow import keras
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
# Pré-traitement et normalisation
X_train, X_test = X_train.reshape(-1, 784), X_test.reshape(-1, 784)
X_train, X_test = X_train / 255., X_test / 255.
# Encodage one_hot des vecteurs de classe
Y_train, Y_test = keras.utils.to_categorical(y_train, 10), keras.utils.to_categorical(y_test, 10)
```

```python
# Instancie un modèle vide
from tensorflow.keras import Sequential
model = Sequential()
# Ajoute les couches au modèle
from tensorflow.keras.layers import Dense, Activation
model.add(Dense(10, input_dim=784, name='fc1'))
```

Keras : apprentissage/inférence

- Visualisation d'un résumé du modèle :

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
fc1 (Dense)                  (None, 10)                7850
_________________________________________________________________
activation_1 (Activation)    (None, 10)                0
=================================================================
Total params: 7,850.0
Trainable params: 7,850.0
Non-trainable params: 0.0
```

- Entraînement des paramètres sur le jeu de données d'apprentissage :

```
# Apprentissage, méthode .fit() "à la scikit-learn"
model.fit(X_train, y_train,batch_size=128, epochs=20,verbose=1)
```

Keras : évaluation du modèle

- Évaluation des performances sur le jeu de test :

```
scores = model.evaluate(X_test, Y_test, verbose=0)
print("%s TEST: %.2f%%" % (model.metrics_names[0], scores[0]*100))
print("%s TEST: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
Epoch 1/10
60000/60000 [==============================] - 1s - loss: 0.4778 - acc: 0.8692
Epoch 2/10
60000/60000 [==============================] - 1s - loss: 0.3357 - acc: 0.9062
Epoch 3/10
60000/60000 [==============================] - 1s - loss: 0.3132 - acc: 0.9117
Epoch 4/10
60000/60000 [==============================] - 1s - loss: 0.3016 - acc: 0.9150
Epoch 5/10
60000/60000 [==============================] - 1s - loss: 0.2941 - acc: 0.9180
Epoch 6/10
60000/60000 [==============================] - 1s - loss: 0.2883 - acc: 0.9196
Epoch 7/10
60000/60000 [==============================] - 1s - loss: 0.2840 - acc: 0.9205
Epoch 8/10
60000/60000 [==============================] - 1s - loss: 0.2804 - acc: 0.9215
Epoch 9/10
60000/60000 [==============================] - 1s - loss: 0.2775 - acc: 0.9230
Epoch 10/10
60000/60000 [==============================] - 1s - loss: 0.2750 - acc: 0.9234
loss TEST: 27.20%
acc TEST: 92.20%
```

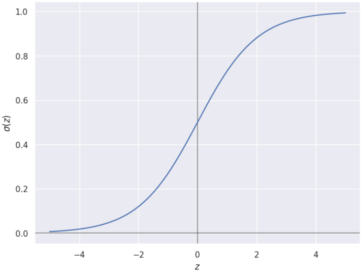
$\Longrightarrow$ mise en pratique dans la séance de travaux pratiques “Deep Learning avec Keras”

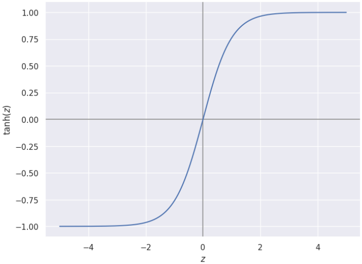
Plan du cours

Non-linéarités modernes

- Activations non-linéaires classiques : sigmoïde, $\tanh$

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}-e^{-z}}$$

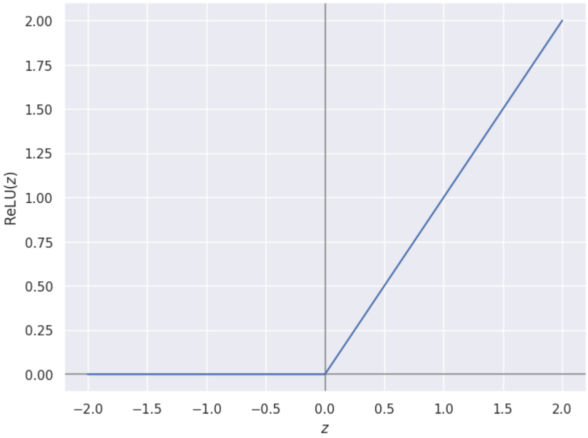




⊖ Régime de saturation $\implies$ dérivée nulle hors de $x \approx 0$
  - $\Rightarrow$ Gradient “évanescent”(*vanishing gradient*) : rétropropagation limitée
  - $\Rightarrow$ Convergence lente car la mise à jour des paramètres est de faible amplitude

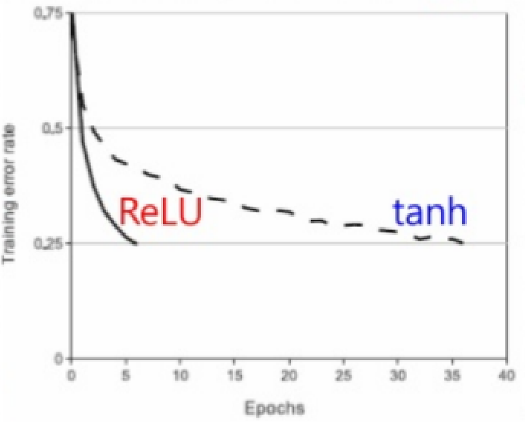
Rectified Linear Unit (ReLU) Nair et Hinton 2010

$$ReLU(z) = \begin{cases} z \text{ si } z \geq 0 \\ 0 \text{ sinon} \end{cases} = \max\{0, z\}$$



Rectified Linear Unit (ReLU)

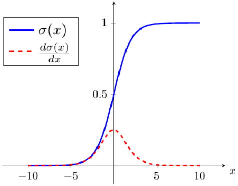
- Réduire l'occurrence des gradients évanescents permet :
  - ⇒ d'entraîner des modèles plus profonds (la décroissance de la norme du gradient est plus lente),
  - ⇒ d'accélérer la convergence du modèle.



Exemple : CNN à 4 couches entraîné sur CIFAR-10. Remplacer $\tanh$ par ReLU permet de converger en $6\times$ moins d'itérations. Extrait de Krizhevsky, Sutskever et Hinton 2012

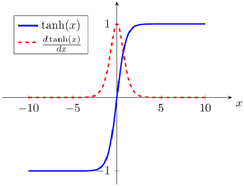
Panel de fonctions d'activation

Sigmoïde



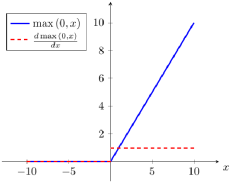
- Saturation
- Calcul coûteux
- Non centrée sur zéro

Tanh



- Saturation
- Calcul coûteux
- Centrée sur zéro

ReLU

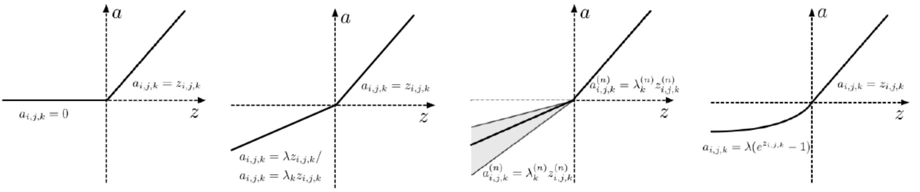


- Pas de saturation
- Efficace à calculer
- Non centrée sur zéro
- Les activations négatives sont ignorées

Fonctions d'activation non-linéaire paramétriques

Deux observations concernant la fonction d'activation ReLU :

1. Pour $z < 0$, l'activation ReLU vaut 0 avec un gradient nul $\implies$ pas de gradient propagé.
2. Peut-on déterminer automatiquement certains paramètres de la fonction d'activation (pente, seuil, etc.)?



(a) ReLU (b) LReLU/PReLU (c) RReLU (d) ELU

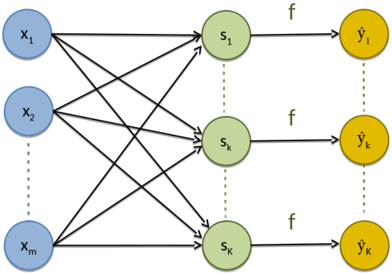
- Leaky ReLU (LReLU) : pente négative λ hyperparamètre à régler
- Parametric ReLU (PReLU) : pente négative λ_k apprise pendant l'entraînement
- Randomized ReLU (RReLU) : pente négative λ_k^n tirée selon une loi uniforme
- Exponential Linear Unit (ELU) : paramètre de lissage λ hyperparamètre à régler

Initialisation des poids

Comment initialiser les paramètres du modèle ?

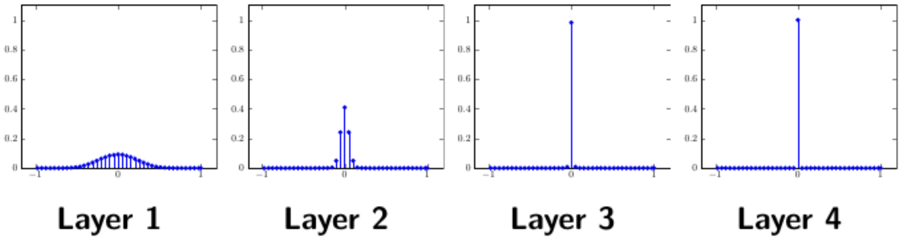
La fonction objectif à minimiser non-convexe par rapport aux paramètres des modèles profonds $\implies$ l'initialisation des poids est importante !

- *Zero-init :* toutes les connexions à zéro $\implies$ tous les neurones ont la même sortie, et donc le même gradient !
- Initialisation aléatoire faible, par exemple $\mathbf{W} \sim \mathcal{U}(-0.1, +0.1)$ ou $\mathbf{W} \sim \mathcal{N}(0, \sigma^i)$
- Initialisation "Glorot" :
  - Pour une entrée $\mathbf{x}$ de dimension m et une sortie s, on a : $\mathrm{Var}[s] = m\,\mathrm{Var}[\mathbf{w}]\,\mathrm{Var}[\mathbf{x}]$
  - Idée : initialiser les poids par $\mathbf{W} \sim \frac{1}{\sqrt{m}} \mathcal{N}(0, \sigma^i)$ Glorot et Bengio 2010



Exemples d'initialisation

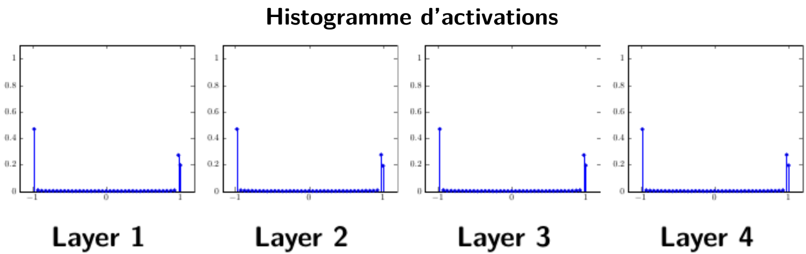
Histogramme d'activations



- Perceptron multi-couche : 10 couches, 500 neurones par couche.
- Activation : $\tanh$
- Initialisation : $w \sim \mathcal{N}(0, \sigma^i)$

$\Longrightarrow$ dans le cas σ^i faible, les activations sont toutes proches de 0, mauvaise initialisation.

Exemples d'initialisation

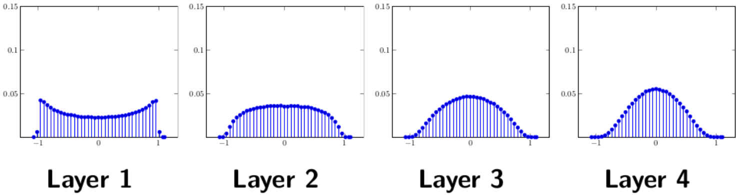


- Perceptron multi-couche : 10 couches, 500 neurones par couche.
- Activation : $\tanh$
- Initialisation : $w \sim \mathcal{N}(0, \sigma^i)$

$\Longrightarrow$ dans le cas σ^i grand, les activations sont toutes saturées (± 1), mauvaise initialisation (gradient évanescent).

Exemples d'initialisation

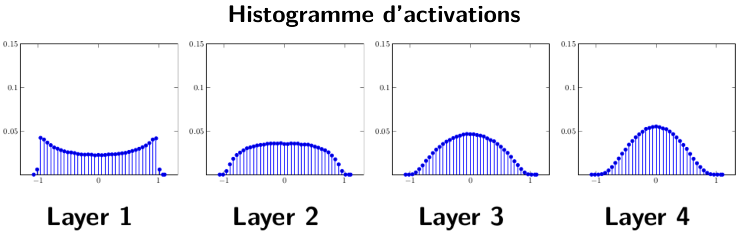
Histogramme d'activations



- Perceptron multi-couche : 10 couches, 500 neurones par couche.
- Activation : $\tanh$
- Initialisation : "Glorot", $\mathbf{W} \sim \frac{1}{\sqrt{500}}\mathcal{N}(0, 1)$

$\Longrightarrow$ variance adaptative et contrôlée pour chaque couche.

Exemples d'initialisation

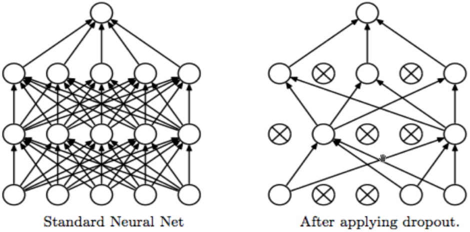


- Perceptron multi-couche : 10 couches, 500 neurones par couche.
- Activation : ReLU
- Dans le cas ReLU : $\mathrm{Var}[s] = 2m\,\mathrm{Var}[\mathbf{w}]\,\mathrm{Var}[\mathbf{x}] \implies \mathbf{W} \sim \frac{1}{\sqrt{2m}}\,\mathcal{N}(0, \sigma^i)$
- Initialisation : "Glorot", $\mathbf{W} \sim \frac{1}{\sqrt{1000}}\mathcal{N}(0,1)$

$\implies$ variance adaptative et contrôlée pour chaque couche.

Régularisation : Dropout Srivastava et al. 2014

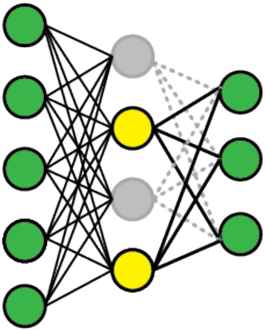
- Supprime aléatoirement un neurone caché avec une probabilité p, par exemple $p = \frac{1}{2}$.
- Méthode de régularisation pour limiter le sur-apprentissage
  - Éviter la co-adaptation
  - Forcer la redondance entre les poids
- Interprétation alternative : combinaison de nombreux réseaux de neurones qui diffèrent légèrement



Credits: Geoffrey E. Hinton, NIPS 2012

Dropout : implémentation

- Optimisation : le dropout est différentiable
  - Les activations des neurones *dropped* sont mises à zéro
  - Les mises à jour de ces neurones sont ignorées
- À l'inférence, plusieurs possibilités :
  - Appliquer le dropout en inférence et moyenner sur les prédictions de sortie
  - Alternative plus rapide : inférence normale mais multiplier par p toutes les activations
    - Équivalent à la moyenne géométrique pour un perceptron
    - Approximation assez bonne pour un perceptron multi-couche



Bibliographie I

- BRYSON, Arthur E. et Yu-Chi HO (1969). *Applied Optimal Control : Optimization, Estimation, and Control*. A Blaisdell Book in the Pure and Applied Sciences. Waltham, Mass : Blaisdell Pub. Co. 481 p.
- CAUCHY, Augustin Louis (juill. 1847). *Comptes rendus hebdomadaires des séances de l'Académie des sciences*. Paris : Gauthier-Villars. URL : http://gallica.bnf.fr/ark:/12148/bpt6k2982c.
- CIREŞAN, Dan C., Ueli MEIER et Jürgen SCHMIDHUBER (2012). "Multi-Column Deep Neural Networks for Image Classification". In : *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Washington, DC, United States, p. 3642-3649. ISBN : 978-1-4673-1226-4. URL : http://dl.acm.org/citation.cfm?id=2354409.2354694.
- CYBENKO, George (1er déc. 1989). "Approximation by Superpositions of a Sigmoidal Function". In : *Mathematics of Control, Signals and Systems* 2.4, p. 303-314. ISSN : 0932-4194, 1435-568X. DOI : 10.1007/BF02551274. URL : https://link.springer.com/article/10.1007/BF02551274.

Bibliographie II

FUKUSHIMA, Kunihiko (1er avr. 1980). “Neocognitron : A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”. In : *Biological Cybernetics* 36.4, p. 193-202. ISSN : 0340-1200, 1432-0770. DOI : 10.1007/BF00344251. URL : https://link.springer.com/article/10.1007/BF00344251.

GLOROT, Xavier et Yoshua BENGIO (31 mars 2010). “Understanding the Difficulty of Training Deep Feedforward Neural Networks”. In : *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, p. 249-256. URL : http://proceedings.mlr.press/v9/glorot10a.html (visité le 13/04/2018).

HORNIK, Kurt (1er jan. 1991). “Approximation Capabilities of Multilayer Feedforward Networks”. In : *Neural Networks* 4.2, p. 251-257. ISSN : 0893-6080. DOI : 10.1016/0893-6080(91)90009-T. URL : http://www.sciencedirect.com/science/article/pii/089360809190009T.

KELLEY, Henry J. (oct. 1960). “Gradient Theory of Optimal Flight Paths”. In : *ARS Journal* 30.10, p. 947-954. DOI : 10.2514/8.5282. URL : https://arc.aiaa.org/doi/10.2514/8.5282 (visité le 07/04/2023).

Bibliographie III

KRIZHEVSKY, Alex, Ilya SUTSKEVER et Geoffrey E. HINTON (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In : *Proceedings of the Neural Information Processing Systems (NIPS)*. NeurIPS, p. 1097-1105. URL : http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

LECUN, Yann (1988). "A Theoretical Framework for Back-Propagation". In : *Proceedings of the 1988 Connectionist Models Summer School, CMU, Pittsburg, PA*. Sous la dir. de D. TOURETZKY, G. HINTON et T. SEJNOWSKI, p. 21-28.

LECUN, Yann et al. (déc. 1989). "Backpropagation Applied to Handwritten Zip Code Recognition". In : *Neural Computation* 1.4, p. 541-551. ISSN : 0899-7667. DOI : 10.1162/neco.1989.1.4.541.

LINNAINMAA, Seppo (1970). "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors". Helsinki University.

MCCULLOCH, Warren S. et Walter H. PITTS (1943). "A Logical Calculus of the Ideas Immanent in Nervous Activity". In : *Bulletin of Mathematical Biophysics* 5, p. 115-133. URL : http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf.

NAIR, Vinod et Geoffrey E. HINTON (2010). "Rectified Linear Units Improve Restricted Boltzmann Machines". In : *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, p. 807-814.

Bibliographie IV

- Rosenblatt, Frank (1957). *The Perceptron : A Probabilistic Model for Information Storage and Organization In The Brain.*
- Rumelhart, D. E., G. E. Hinton et R. J. Williams (1986). "Learning Internal Representations by Error Propagation". In : sous la dir. de David E. Rumelhart, James L. McClelland et given-i=CORPORATE family=PDP Research Group given=CORPORATE. Cambridge, MA, USA : MIT Press, p. 318-362. ISBN : 978-0-262-68053-0. URL : http://dl.acm.org/citation.cfm?id=104279.104293 (visité le 13/04/2018).
- Srivastava, Nitish et al. (2014). "Dropout : A Simple Way to Prevent Neural Networks from Overfitting". In : *Journal of Machine Learning Research* 15, p. 1929-1958. URL : http://jmlr.org/papers/v15/srivastava14a.html (visité le 19/01/2016).
- Werbos, Paul John (1975). "Beyond Regression : New Tools for Prediction and Analysis in the Behavioral Sciences". Harvard University. 906 p. Google Books : z81XmgEACAAJ.