

Apprentissage automatique 1 (EMINF2C1)

Introduction à l'apprentissage profond
M1 IAFA, IMA-RO

Contributeurs :
Stergos Afantinos
Farah Benamara
Sandrine Mouysset
Philippe Muller

Thomas Pellegrini (resp.)

Contacts : prenom.nom@irit.fr

Semestre 8

- Cours, 12 heures
- TD, 8 heures, 2 groupes
- TP, 10h, 4 groupes

Modalités de Contrôle des Connaissances

- CT : 70%, examen écrit de deux heures
- CCTP : 30%, note : un compte-rendu de TP (à définir),
règle "16" : ABI → note 0, ABJ → coef 0

Objectifs généraux

- Acquérir des connaissances théoriques et un savoir-faire pratique en apprentissage automatique profond
- Devenir autonomes face à un problème qui fait appel à des techniques d'apprentissage profond
- Avoir une idée des limites et des risques de ces approches

Bibliographie

- Eli Stevens, Luca Antiga, Thomas Viehmann. Deep learning with PyTorch. Manning, 2020.
<https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf>

Aperçu général du contenu

- Du perceptron au perceptron multi-couche (MLP)
- Entraînement d'un réseau de neurones
- Réseaux convolutifs, Convolutional Neural Networks (CNN)
- Réseaux de neurones récurrents, Recurrent Neural Networks (RNN)
- Modèles séquence à séquence (seq2seq), mécanismes d'attention
- Limites actuelles des modèles profonds : interprétabilité, biais, robustesse, exemples adversaires

Sommaire cours 1

- Introduction à l'apprentissage profond : historique, exemples d'application
- Le perceptron, brique de base historique
- Du perceptron au perceptron multi-couche (MLP)
- les différents types de réseaux de neurones
- Algorithme de rétro-propagation et différentiation automatique
- Règles d'actualisation des poids d'un réseau

Qu'est-ce que l'apprentissage automatique *(Machine Learning)* ?

Création et analyse de méthodes et modèles qui tirent des informations des données et servent à faire des prédictions

Domaine vaste qui implique des outils et des idées de plusieurs champs

- Algorithmique et Informatique
- Probabilités et Statistiques
- Optimisation
- Algèbre linéaire et dérivées partielles

Observations. Exemples ou données utilisés pour l'apprentissage ou pour l'évaluation

Attributs (Features) : x . Représentations numériques des observations

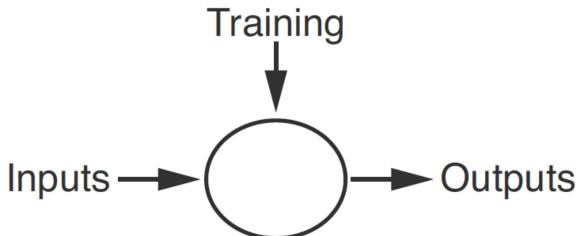
Étiquettes (Labels) : y . Valeurs / catégories associées aux observations

Corpus d'apprentissage / validation / test (Train / Valid / Test, Eval) : Ensembles disjoints d'observations utilisés pour entraîner et évaluer une méthode d'apprentissage

Différents grands paradigmes d'apprentissage

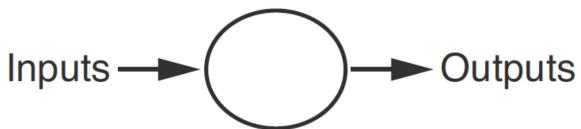
Supervised learning

Learns known patterns
Takes labeled input data
Predicts outcome/future



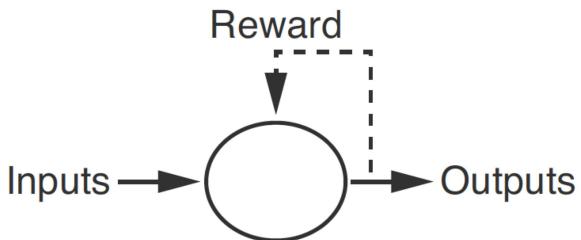
Unsupervised learning

Learns unknown patterns
Takes unlabeled input data
Finds hidden patterns



Reinforcement learning

Generates data
Takes labeled input data
Interacts with environment
Learns series of actions



Exemples d'apprentissage supervisé

Classification. Assigner une catégorie à chaque observation

- Les catégories sont discrètes
- Le label est un indice de classe : $y \in \{0, \dots, K - 1\}$
- Exemple : reconnaissance de chiffres manuscrits
 - x : ?
 - y : ?

Régression. Prédire une valeur réelle

- Les catégories sont continues
- Le label est un nombre réel $y \in \mathbb{R}$
- Exemple : prédire le cours du blé
 - x : ?
 - y : ?

« Le Deep learning est un ensemble d'algorithmes d'apprentissage automatique qui tente d'apprendre à **plusieurs échelles** qui correspondent à **differents niveaux d'abstraction**, à l'aide de modèles à **plusieurs couches**. »

Le deep learning est :

- un ensemble de techniques d'apprentissage automatique probabiliste
- un apprentissage automatique de représentations hiérarchiques de paramètres (features)
- fondé en général sur des réseaux de neurones artificiels dits profonds ou **Deep Neural Networks**, en abrégé **DNN**

NEURAL NETWORK

Topics: multilayer neural network

- Could have L hidden layers:

- layer input activation for $k > 0$ ($\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$)

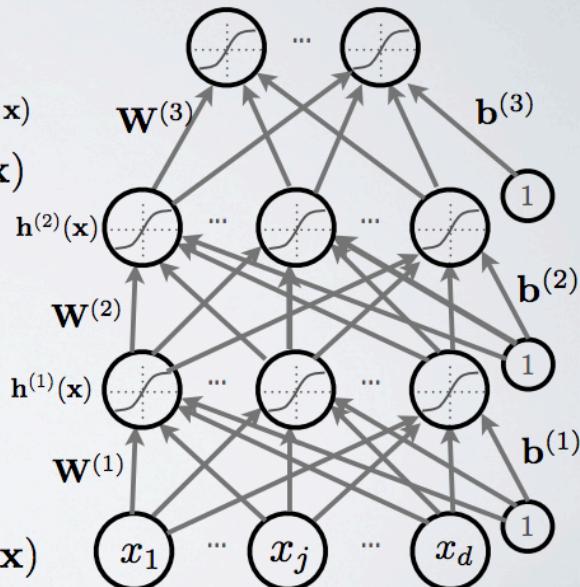
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

- hidden layer activation (k from 1 to L):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

- output layer activation ($k=L+1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$

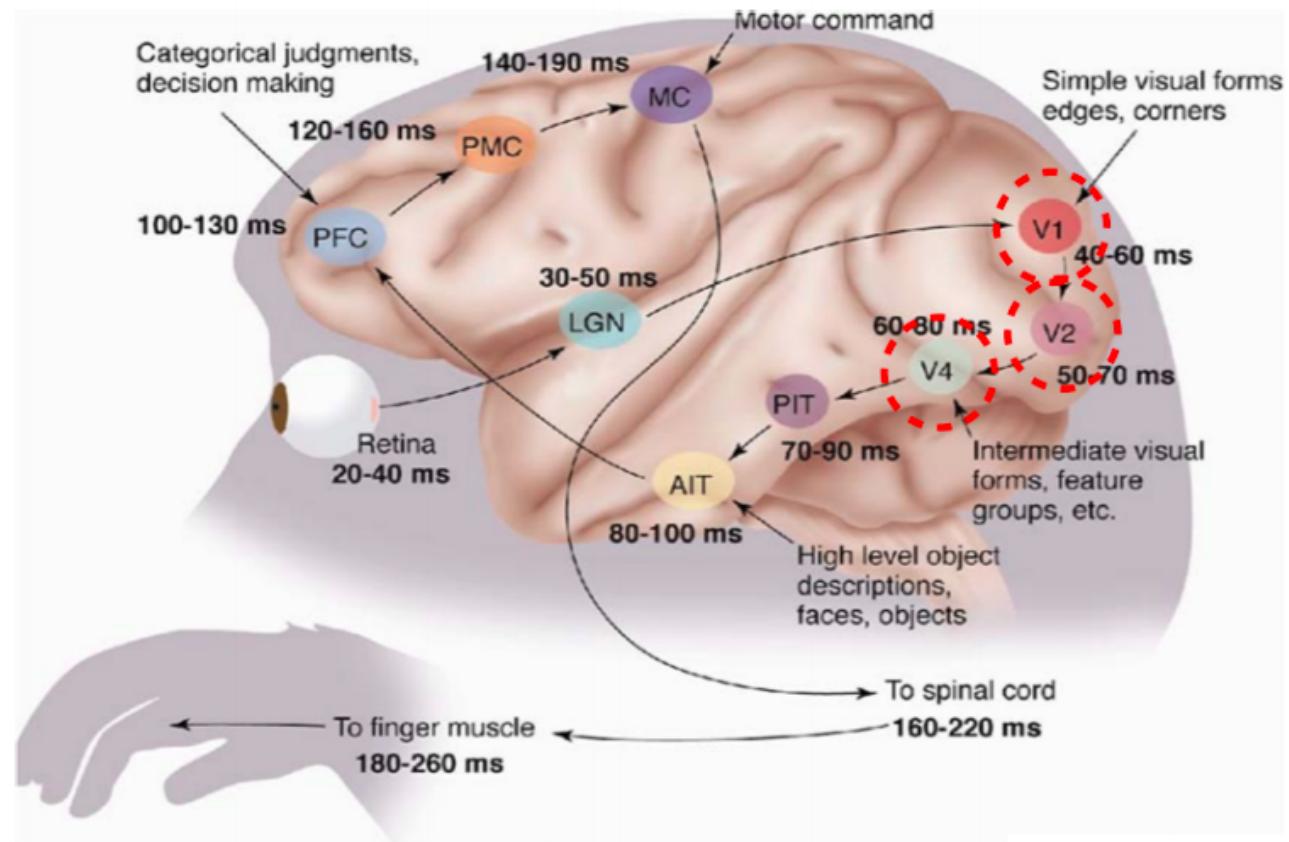


Slide de Hugo Larochelle

<http://info.usherbrooke.ca/hlarochelle/ift725深深.pdf>

- Réseaux de neurones multicouches
- Chaque couche correspond à une représentation dite "distribuée" des données (*distributed representation*)
 - les neurones ne s'excluent pas mutuellement
 - Deux ou plus neurones peuvent être "activés" en même temps,
 - Chaque neurone code un attribut spécifique des données
 - Les neurones ne correspondent pas à un partitionnement de l'entrée

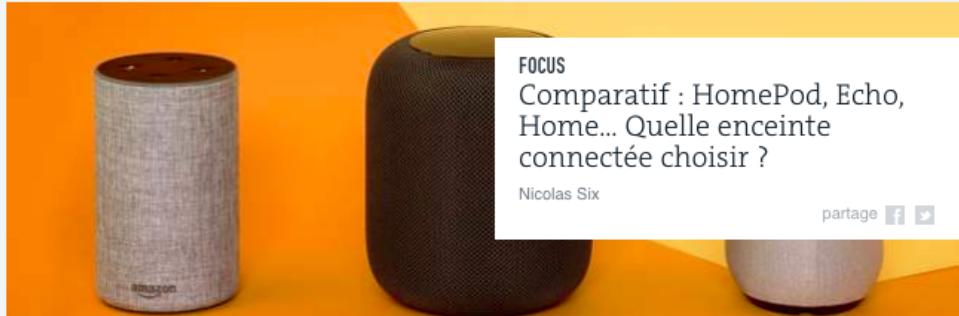
Deep learning : inspiré du cortex visuel



[picture from Simon Thorpe]

Reconnaissance vocale et traduction automatique

SAMEDI 23 JUIN



JEUDI 21 JUIN

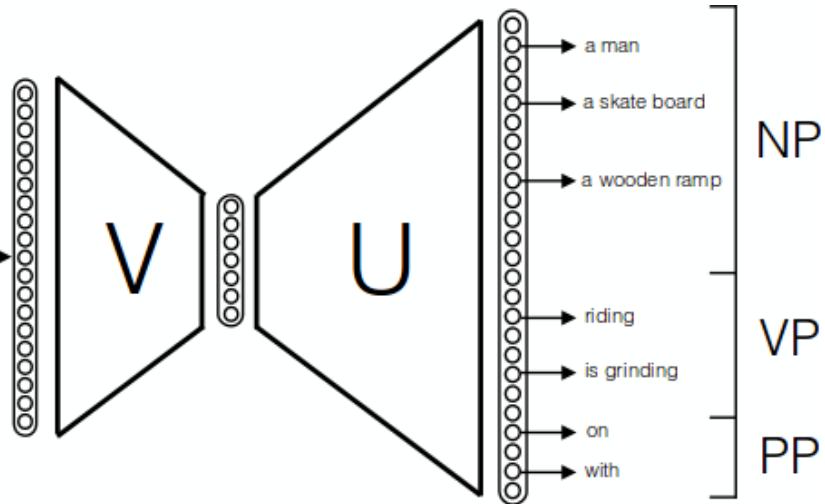
15h00



François Yvon (Professeur au département d'Informatique de l'université Paris-Sud)

partage  

Image captioning



A man in a helmet skateboarding before an audience.
Man riding on edge of an oval ramp with a skate board.
A man riding a skateboard up the side of a wooden ramp.
A man on a skateboard is doing a trick.
A man is grinding a ramp on a skateboard.

Remi Lebret et. al. Phrase-based Image Captioning. ICML 2015

A person riding a motorcycle on a dirt road.



Two dogs play in the grass.



A group of young people playing a game of frisbee.



Two hockey players are fighting over the puck.



A herd of elephants walking across a dry grass field.



A close up of a cat laying on a couch.



Vinyals et al, 2015

18/101

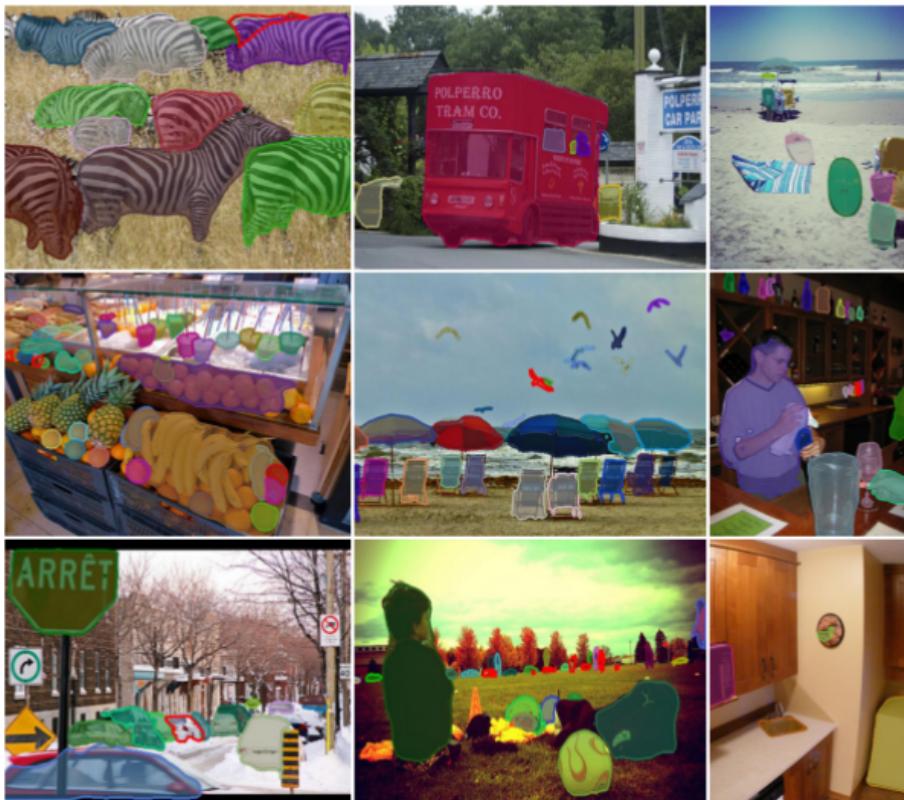
Segmentation sémantique



Self-driving car. Source : NVIDIA

<https://blogs.nvidia.com/blog/2015/02/24/deep-learning-drive/>

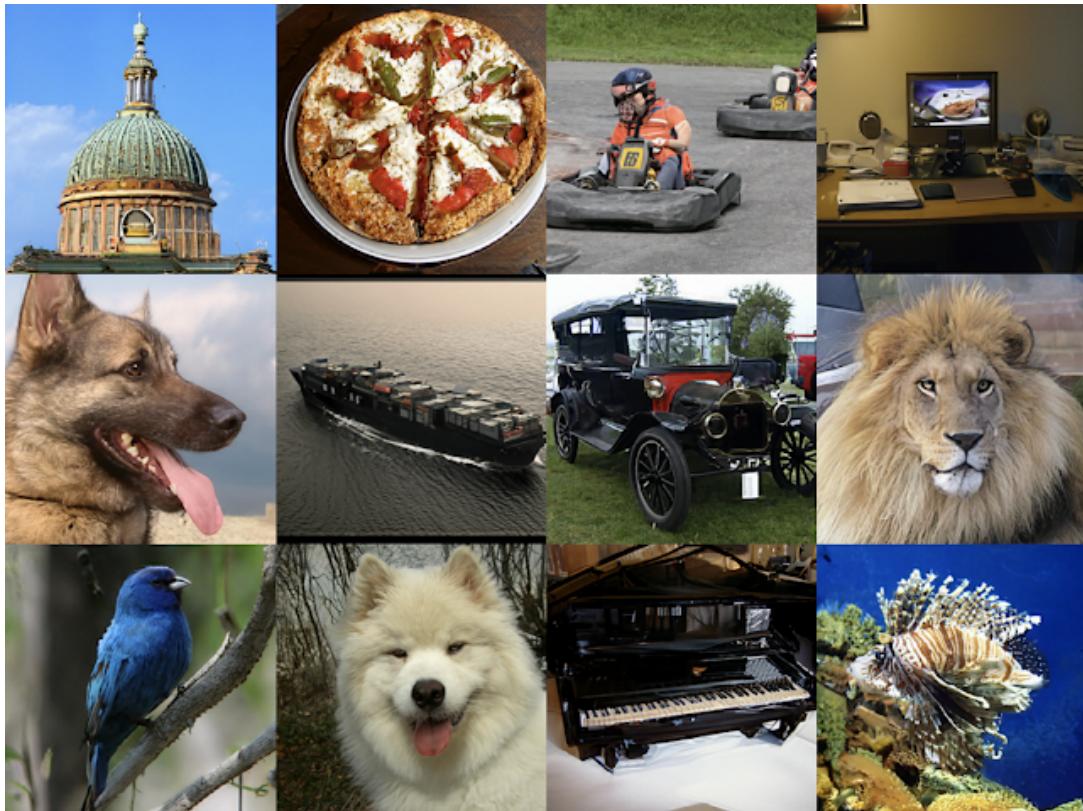
Segmentation sémantique (2)



Pinheiro et al, 2016

20/101

Génération d'images



Class-conditional ImageNet generation with diffusion models

[https://ai.googleblog.com/2021/07/
high-fidelity-image-generation-using.html](https://ai.googleblog.com/2021/07/high-fidelity-image-generation-using.html)

Restons humbles !

14h30



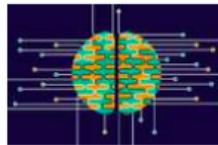
POINT DE VUE

20 Intelligence artificielle : « Les “machines intelligentes” ne sont pas près de remplacer les êtres humains » 7

L'utilisation de robots supprimera certes des emplois, mais en créera d'autres, estime Ulrich Spiesshofer, PDG d'un groupe international d'ingénierie, dans une tribune au « Monde ».

Ulrich Spiesshofer (PDG d'ABB, l'un des leaders mondiaux dans les technologies de l'énergie et de l'automation)

15h33



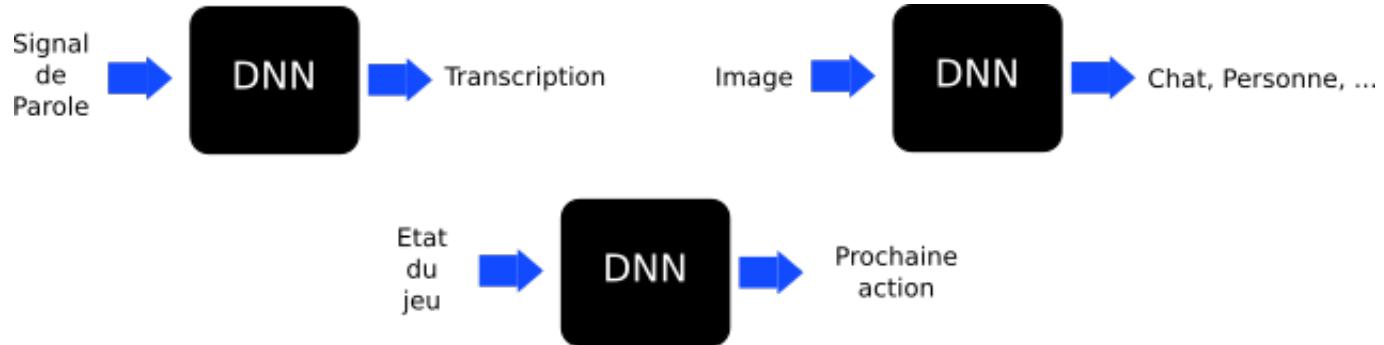
Comment dépasser les limites du deep learning ? POST DE BLOG

Un article de la Technology Review pointe les problématiques actuelles des technologies du deep learning et ouvre quelques portes sur les nouvelles recherches en IA.

partage

<http://internetactu.blog.lemonde.fr/2017/11/19/comment-depasser-les-limites-du-deep-learning/>

Que sont les réseaux de neurones ?



Qu'y a-t-il dans ces boîtes noires ?

Les fondamentaux : le perceptron



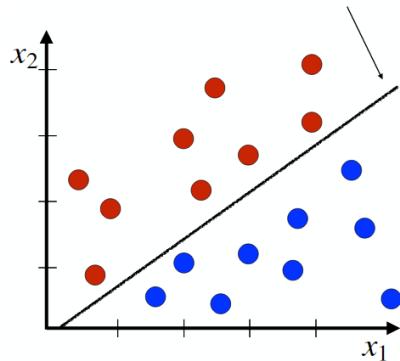
- Frank Rosenblatt, inventeur de la "solution à tout", le "perceptron", 1956
- Algorithme de classification inspiré d'un modèle très simplifié du cerveau humain

Le perceptron

- On suppose que l'on a un problème à deux classes (c_- , c_+) avec m exemples d'apprentissage :

$$\mathcal{D} = \{(\mathbf{x}^j, y^j), j = 1 \dots m\}, \text{ avec } \mathbf{x}^j \in \mathbb{R}^d, y^j \in \{-1, +1\}$$

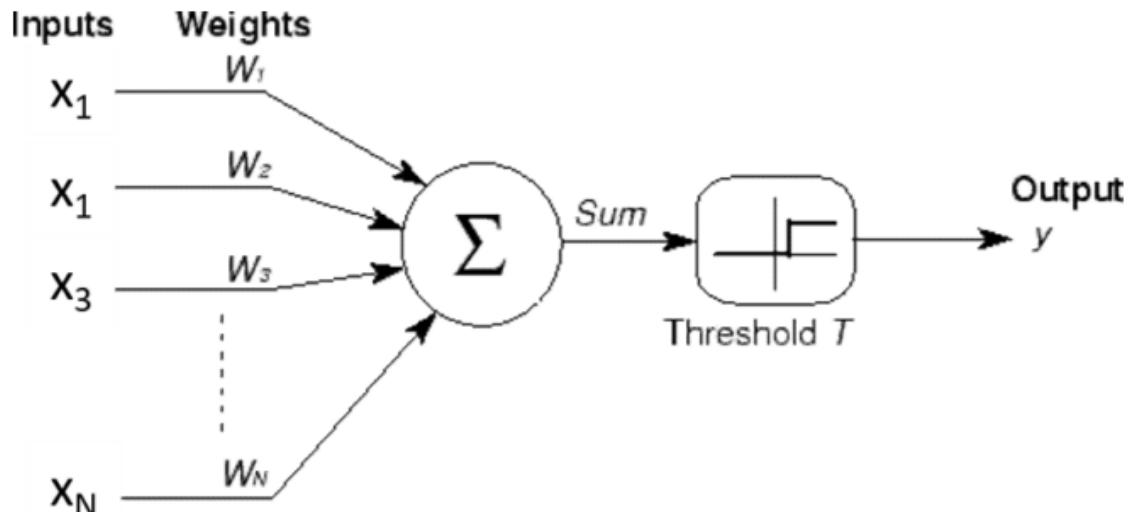
- Hypothèse : les deux classes sont linéairement séparables par un hyperplan, par ex. une droite si on est en 2-d



Prédiction :

$$y(x) = \hat{y} = \begin{cases} +1 & \text{si } w_1x_1 + w_2x_2 > T \\ -1 & \text{sinon} \end{cases}$$

$$\text{Prédiction : } y(x) = \hat{y} = \begin{cases} +1 & \text{si } w_1x_1 + w_2x_2 > T \\ -1 & \text{sinon} \end{cases}$$



- Les features sont combinés linéairement
⇒ perceptron = "classifieur linéaire"
- le "neurone" est activé si la somme dépasse un seuil T
(Threshold)

On pose $w_0 = -T$ et $z = w_0 \times 1 + w_1 x_1 + w_2 x_2$

Les prédictions deviennent : $\hat{y} = \begin{cases} +1 & \text{si } z > 0 \\ -1 & \text{sinon} \end{cases}$

Écriture "vectorielle" :

w =

x =

z =

Comment trouver les valeurs des poids w et du seuil T ?

Algorithme du perceptron :

- ➊ On initialise w à $\vec{0}$
- ➋ On fait des prédictions sur les points
- ➌ On modifie w , **uniquement pour les points mal classés**
- ➍ On recommence à l'item 2 jusqu'à convergence

Comment trouver les valeurs des poids w et du seuil T ?

Algorithme du perceptron :

- ➊ On initialise w à $\vec{0}$
- ➋ On fait des prédictions sur les points
- ➌ On modifie w , **uniquement pour les points mal classés**
- ➍ On recommence à l'item 2 jusqu'à convergence

⇒ Quelle règle pour actualiser w ?

Comment trouver les valeurs des poids w et du seuil T ?

Algorithme du perceptron :

- ➊ On initialise \mathbf{w} à $\vec{0}$
- ➋ On fait des prédictions sur les points
- ➌ On modifie \mathbf{w} , **uniquement pour les points mal classés**
- ➍ On recommence à l'item 2 jusqu'à convergence

⇒ Quelle règle pour actualiser \mathbf{w} ?

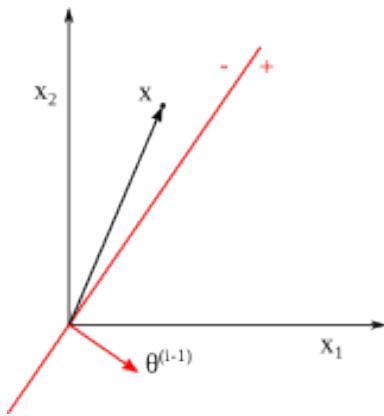
$$\mathbf{w} = \begin{cases} \mathbf{w} + \mathbf{x} & \text{si } y = 1 \text{ et } \hat{y} = -1 \\ \mathbf{w} - \mathbf{x} & \text{si } y = -1 \text{ et } \hat{y} = 1 \end{cases}$$

Soit

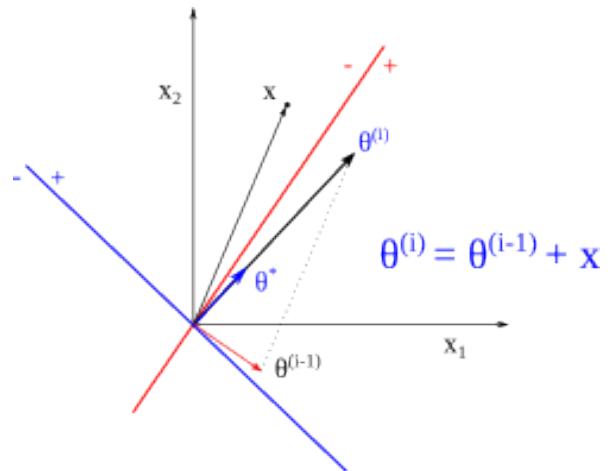
$$\boxed{\mathbf{w} = \mathbf{w} + y\mathbf{x} \text{ si } \mathbf{x} \text{ mal classé}}$$

Illustration 2D d'une itération de la règle du perceptron

- $\mathbf{x} \in c_+$ mal classé à l'itération $i - 1$



- $\mathbf{x} \in c_+$ bien classé à l'itération i



Variantes équivalentes de la règle

- $\mathbf{w} = \mathbf{w} + y\mathbf{x}$ si \mathbf{x} mal classé
- Avec η appelé "pas d'apprentissage" ou *learning rate* :
 $\mathbf{w} = \mathbf{w} + \eta y\mathbf{x}$ si \mathbf{x} mal classé
- On peut enlever le test conditionnel :
 $\mathbf{w} = \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$

Algorithme perceptron online ($\mathcal{D}, y \in \{-1, +1\}$)

- 1: Initialiser $\mathbf{w} \leftarrow \mathbf{0}$
 - 2: TANT QUE pas convergence FAIRE
 - 3: POUR j de 1 à m FAIRE
 - 4: SI $y^j \mathbf{w}^t \mathbf{x}^j \leq 0$ ALORS
 - 5: $\mathbf{w} \leftarrow \mathbf{w} + y^j \mathbf{x}^j$
-

Algorithme perceptron online ($\mathcal{D}, y \in \{-1, +1\}$)

- 1: Initialiser $\mathbf{w} \leftarrow \mathbf{0}$
- 2: TANT QUE pas convergence FAIRE
- 3: POUR j de 1 à m FAIRE
- 4: SI $y^j \mathbf{w}^t \mathbf{x}^j \leq 0$ ALORS
- 5: $\mathbf{w} \leftarrow \mathbf{w} + y^j \mathbf{x}^j$

- Apprentissage online : le modèle est actualisé à chaque exemple vu
- Apprentissage batch : le modèle est actualisé après avoir vu le corpus entier

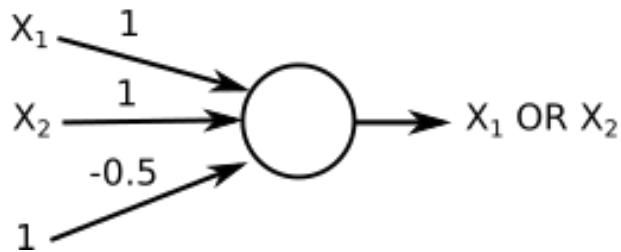
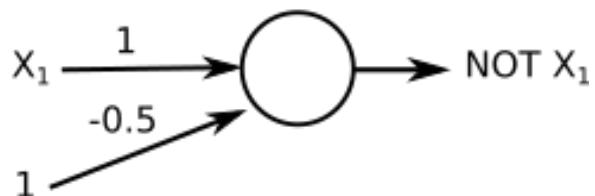
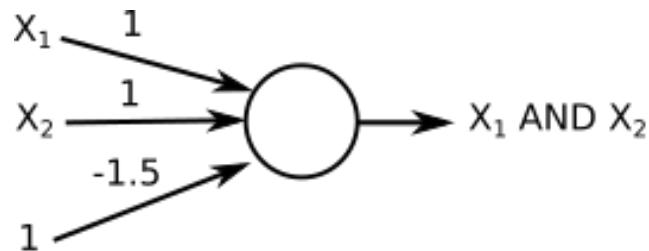
Version apprentissage par "batch"

- "Batch" en anglais : "jeu de plusieurs exemples de données"

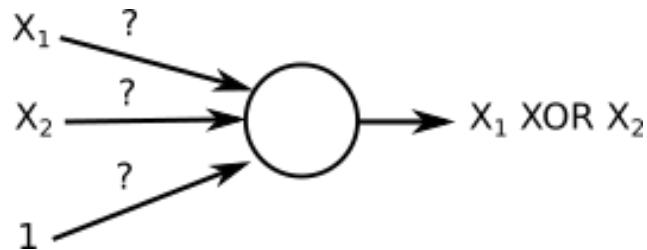
Algorithme perceptron batch ($\mathcal{D}, y^j \in \{-1, +1\}$)

- 1: Initialiser $\mathbf{w} \leftarrow \mathbf{0}$
- 2: FAIRE
- 3: Initialiser $\delta \leftarrow \mathbf{0}$
- 4: POUR j de 1 à m FAIRE
- 5: $s^j \leftarrow \mathbf{w}^t \mathbf{x}^j$
- 6: SI $y^j s^j \leq 0$ ALORS
- 7: $\delta \leftarrow \delta + y^j \mathbf{x}^j$
- 8: $\mathbf{w} \leftarrow \mathbf{w} + \delta$
- 9: TANT QUE $|\delta| > \epsilon$

Le perceptron permet de simuler des fonctions booléennes...



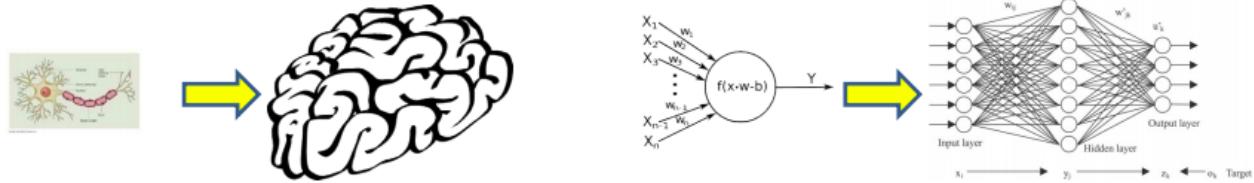
Sauf... La fonction XOR !



⇒ Pas de solution avec le perceptron !

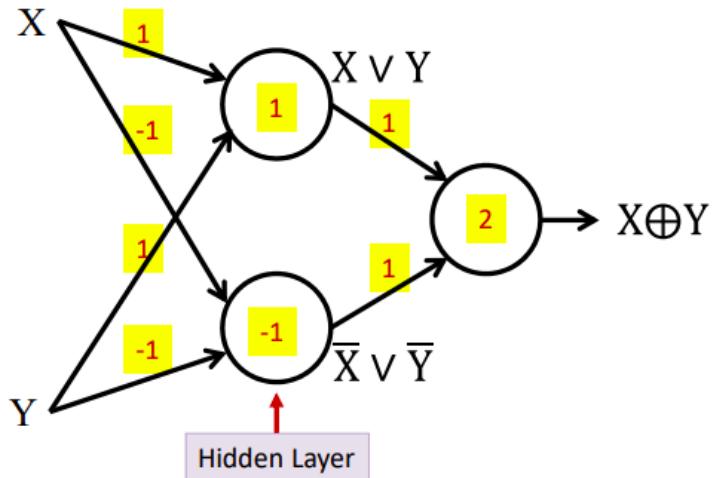
- Minsky and Papert, 1968

Un seul neurone n'est pas suffisant



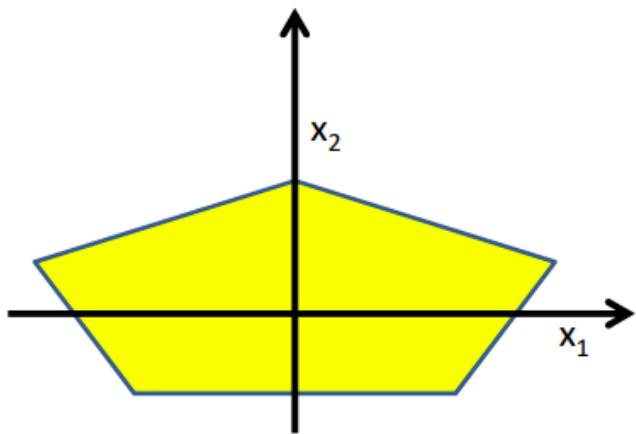
- Minsky and Papert, 1969, *Perceptrons : An Introduction to Computational Geometry*
 - Un neurone seul est un élément faible
 - Il faut des neurones interconnectés \Rightarrow "Réseau de neurones multicouche"

Perceptron Multi-Couche, Multi-Layer Perceptron (MLP)

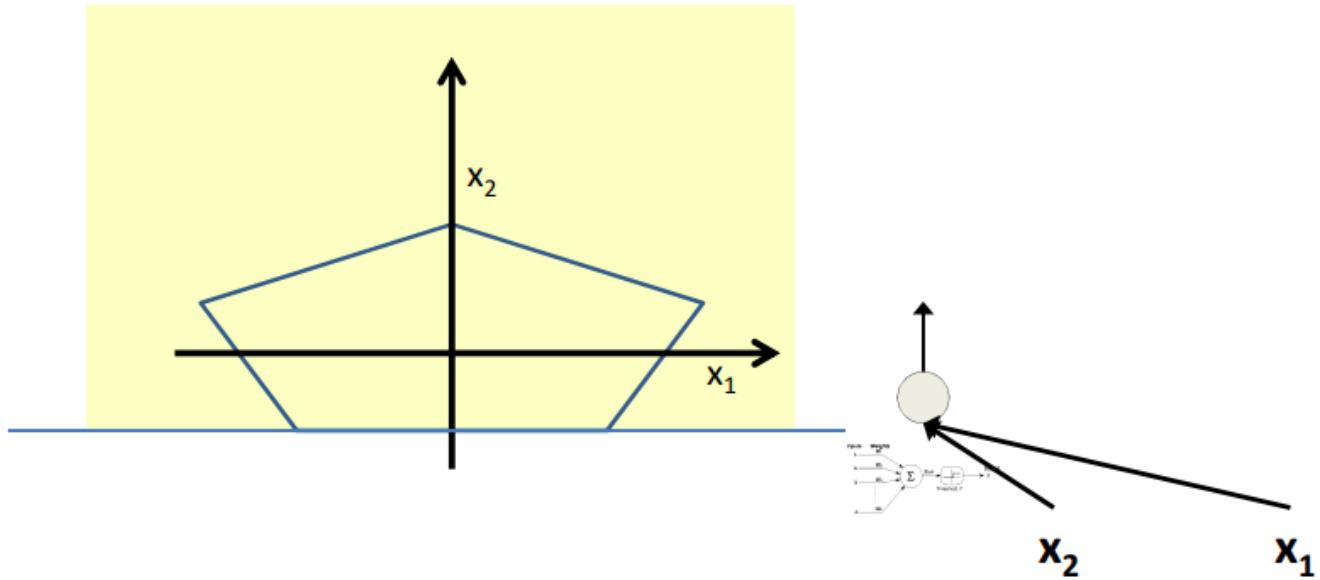


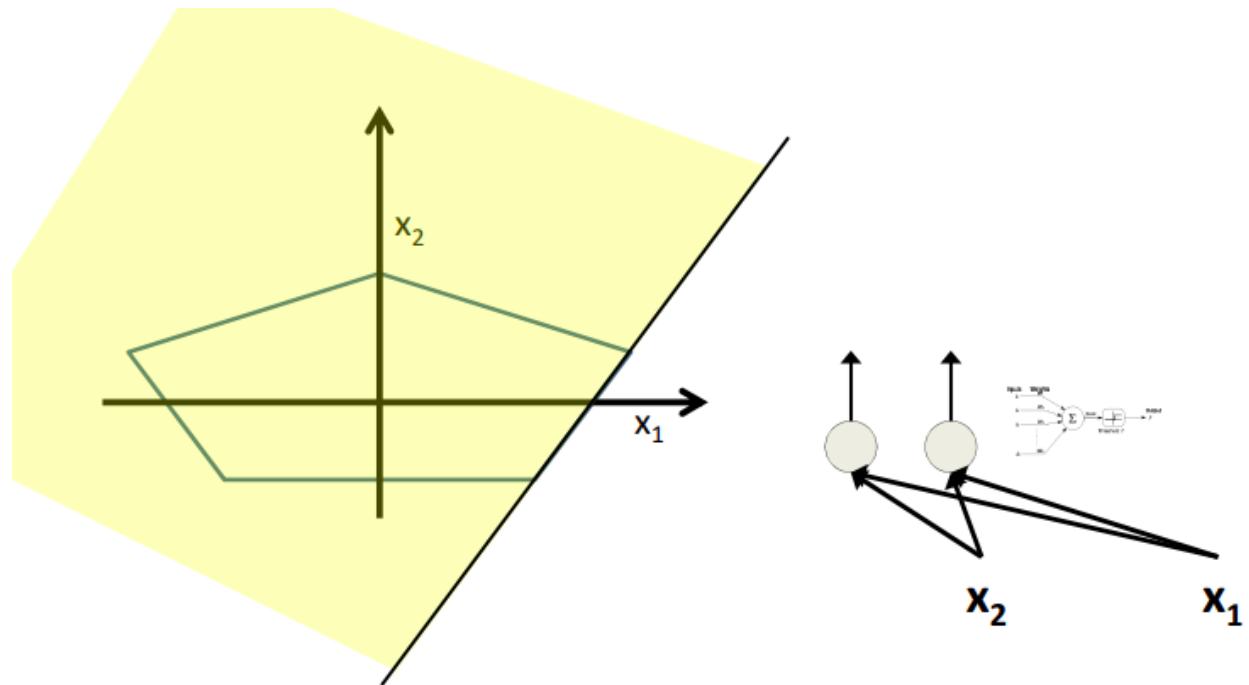
- XOR
 - La première couche est une couche "cachée"
 - Architecture proposée dans Minsky and Papert, 1968

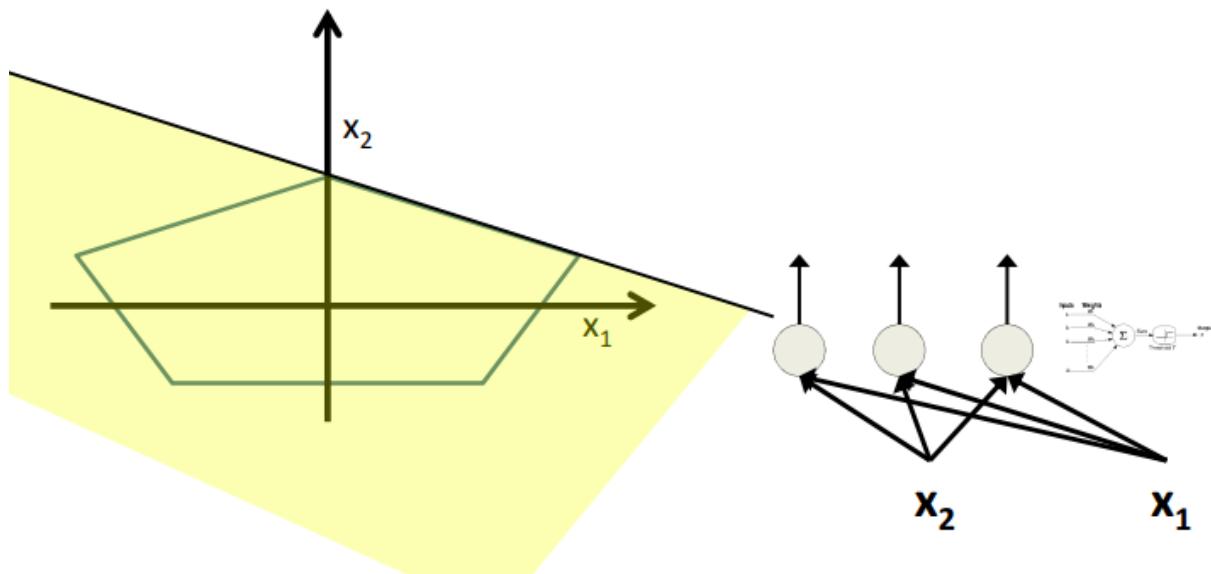
Composer des frontières de décision complexes

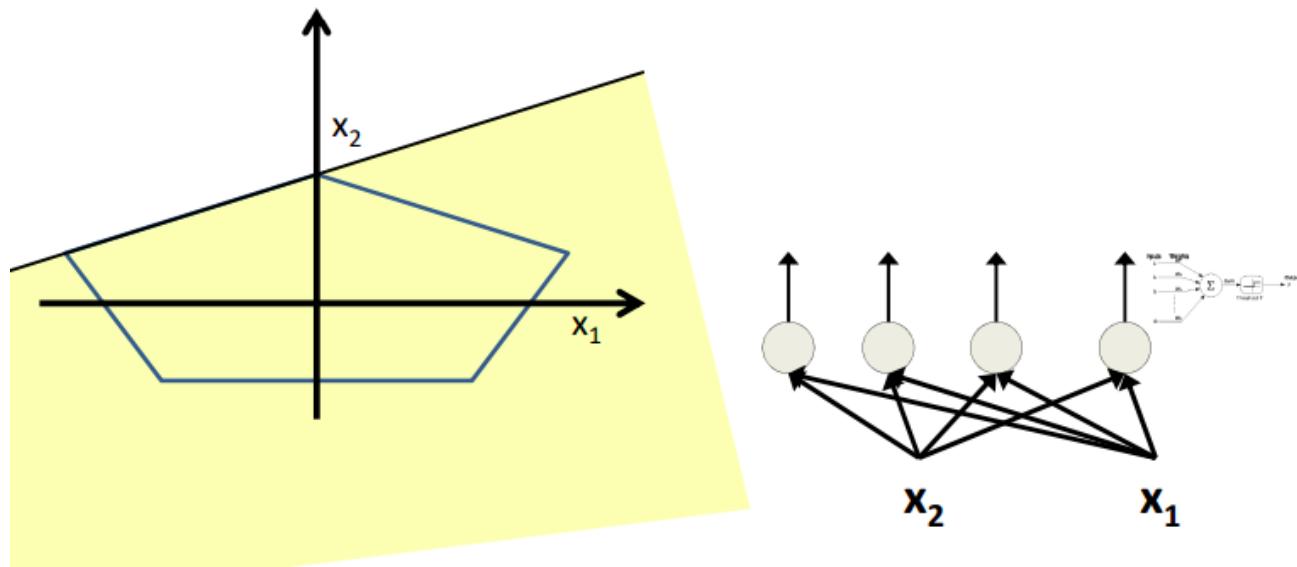


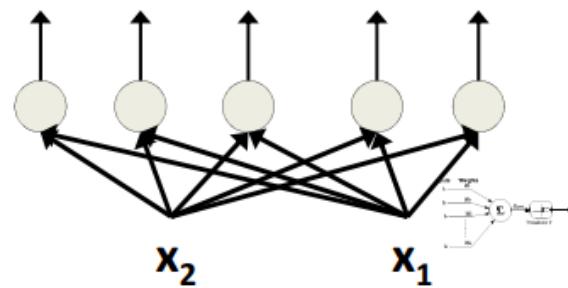
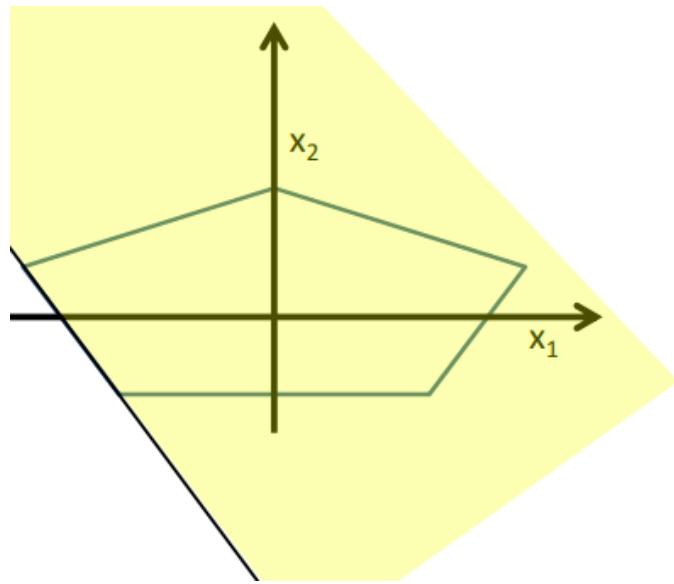
Objectif : construire un réseau de neurones avec un unique neurone de sortie qui "s'active" uniquement pour les points de la région jaune

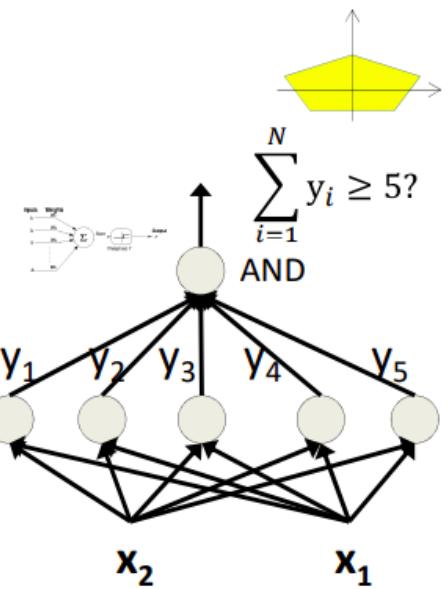
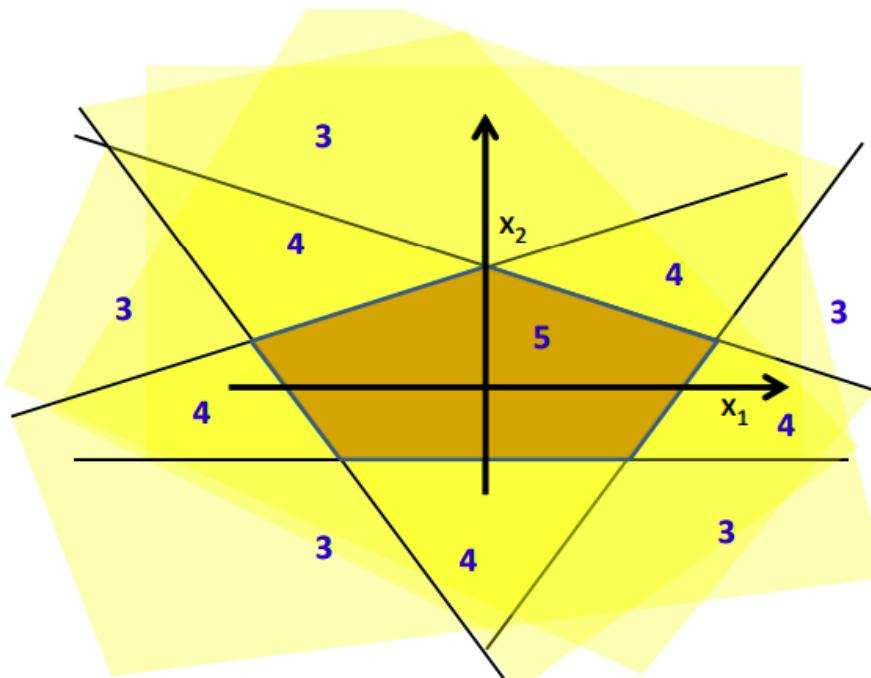




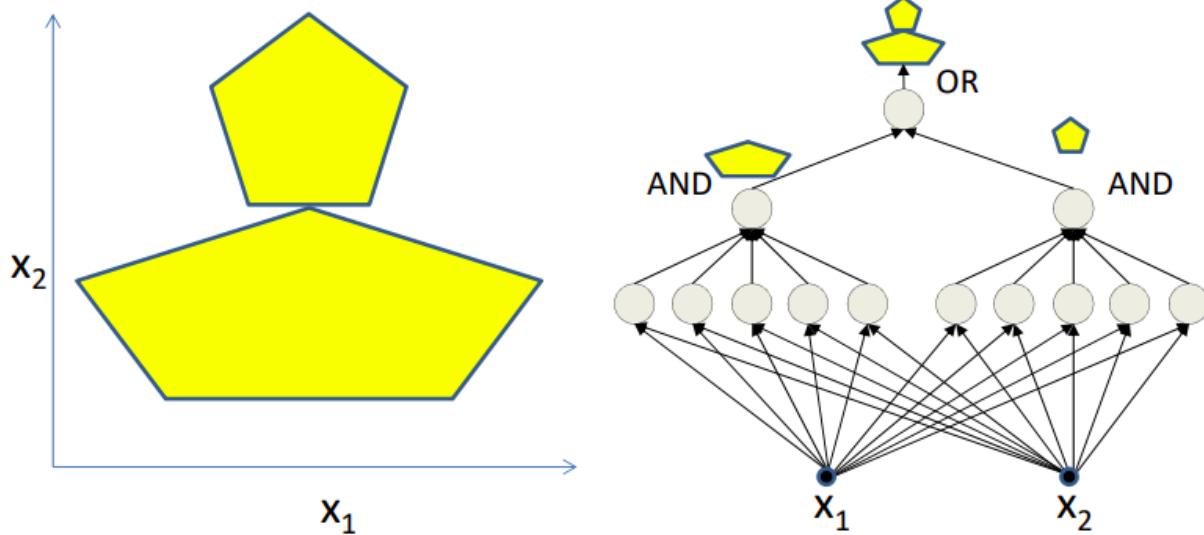








Deux polygones ?

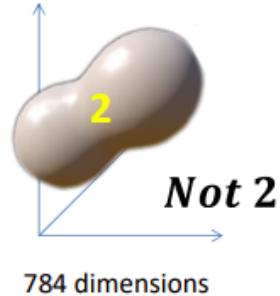
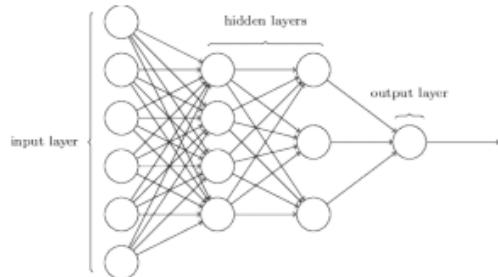


- Le réseau ne doit s'activer que pour la région jaune
- Deux polygones
 - ⇒ Un neurone "OR"
 - ⇒ Trois couches : 2 couches cachées, 1 couche de sortie

Frontières de décision complexes



784 dimensions
(MNIST)



- Problèmes de classification dans la vie réelle : trouver des frontières de décision dans des espaces à grande dimension
 - Faisable par un MLP
 - Un MLP peut prendre en entrée un vecteur de valeurs réelles et donner un probabilité d'appartenance à une ou plusieurs classes

Revenons aux classificateurs linéaires

Un même classifieur peut être formalisé avec différents w puisque c'est le signe qui donne le résultat de classification :

$$\begin{aligned}\hat{y} &= h(x; \mathbf{w}) = \text{signe}(\mathbf{w}^t \mathbf{x}) \\ &= \text{signe}(2 \times \mathbf{w}^t \mathbf{x}) \\ &= \text{signe}(1025.23 \times \mathbf{w}^t \mathbf{x}) \\ &= \dots\end{aligned}$$

⇒ la norme de \mathbf{w} n'est pas pertinente vis-à-vis de la frontière de décision

- Nous utiliserons ce degré de liberté qu'est la norme de \mathbf{w} par la suite ("régularisation")
- Cependant, la valeur du produit scalaire $\mathbf{w}^t \mathbf{x}$ donne une notion de confiance de la classification d'un exemple en point positif ou négatif
- Cette confiance varie uniquement si un point se déplace perpendiculairement à la frontière de décision

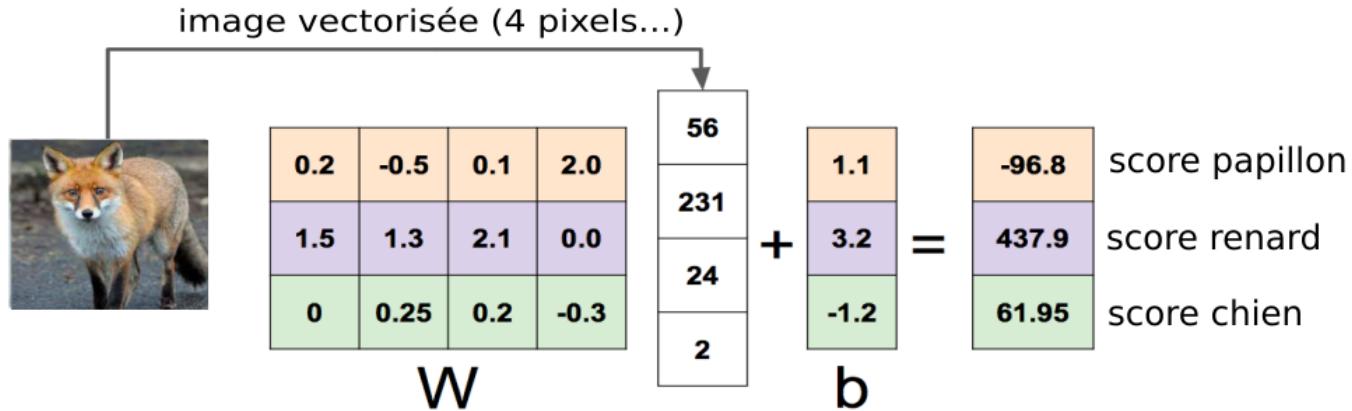
Classification linéaire : cas multiclass

Cas à $k > 2$ classes

$$\begin{bmatrix} \text{score } C_1 \\ \text{score } C_2 \\ \vdots \\ \text{score } C_k \end{bmatrix} = h_w(x) = Wx + b$$

matrice de poids biais (vecteur)

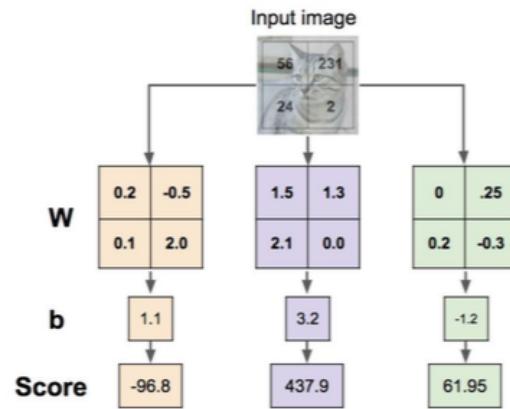
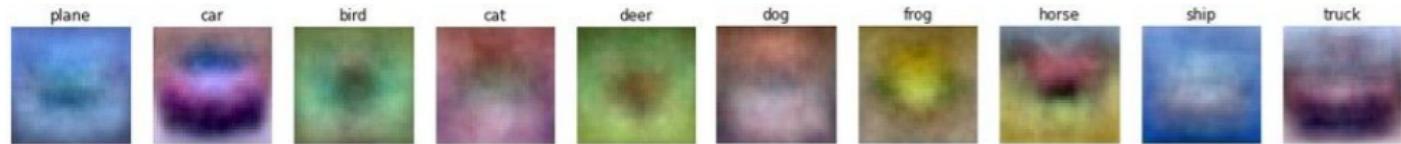
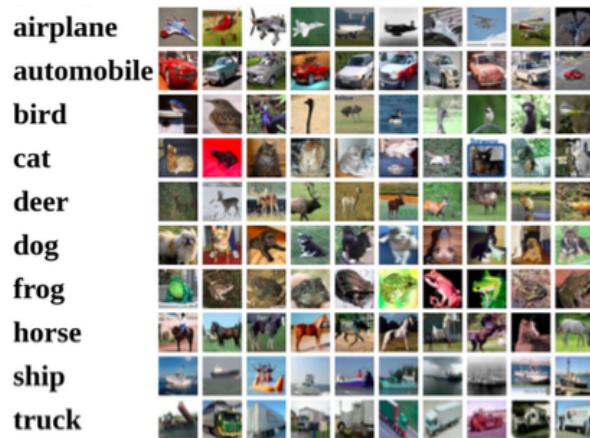
Classification linéaire : cas multiclass



Adapté de <https://cs231n.github.io/linear-classify/>

Classification linéaire : cas multiclass

CIFAR-10



Source : <https://cs231n.github.io/linear-classify/>

Remarque technique

On peut "augmenter" \mathbf{x} et \mathbf{w} pour intégrer le biais w_0 dans le produit scalaire :

$$\mathbf{x} = [1, \quad x_1, \quad x_2, \dots]^t$$

$$\mathbf{w} = [w_0, \quad w_1, \quad w_2, \dots]^t$$

Le produit scalaire a intégré le biais w_0 :

$$\hat{y} = \text{signe}(\mathbf{w}^t \mathbf{x})$$

Cas multiclass :

$$\mathbf{W} = \begin{bmatrix} b_0 & W_{01} & W_{02} & \dots \\ b_1 & W_{11} & W_{12} & \dots \\ \vdots & & & \end{bmatrix}$$

Remarque technique 2

On peut aussi vouloir regrouper tous les exemples des données en une seule matrice :

$$X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_d^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_d^{(2)} \\ \vdots & & & & \\ 1 & x_1^{(N)} & x_2^{(N)} & \dots & x_d^{(N)} \end{bmatrix}$$

Dans le cas binaire (w vecteur), c'est pratique pour obtenir un vecteur de prédictions \hat{y} pour les N exemples d'un coup :

$$\hat{y} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(N)} \end{bmatrix} = \text{signe}(\mathbf{w}^t X)$$

La question importante :

Comment trouver de bonnes valeurs pour les
poids w ?

La question importante :

Comment trouver de bonnes valeurs pour les poids w ?

Réponse courte : il nous faut une fonction qui mesure l'erreur du modèle sur les exemples d'apprentissage et ensuite on trouve les poids w qui minimisent cette fonction

La question importante :

Admettons que l'on ait défini une fonction qui mesure les erreurs, comment peut-on la minimiser ?

La question importante :

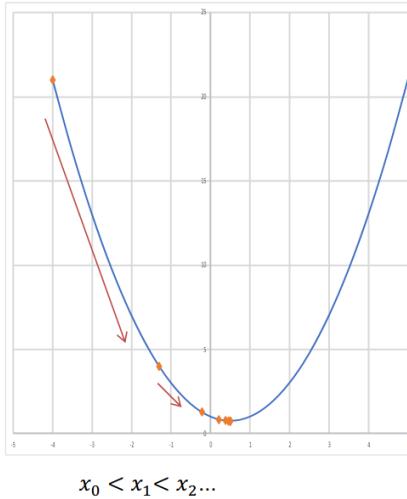
Admettons que l'on ait défini une fonction qui mesure les erreurs, comment peut-on la minimiser ?

Réponse courte : avec l'algorithme de la **descente de gradient**

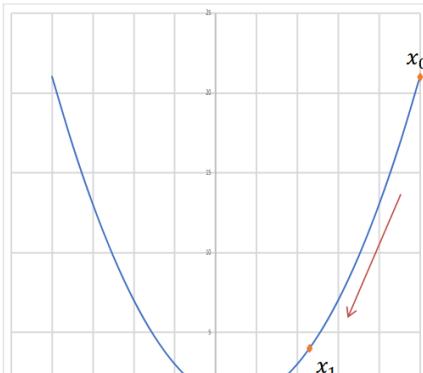
La descente de gradient en un slide

Algorithme DG sur une "loss"
 $\mathcal{L}(y, \hat{y})$:

- ➊ Initialiser \mathbf{w} avec un $\mathbf{w}_{t=0}$ aléatoire
- ➋ Répéter jusqu'à convergence :
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} \mathcal{L}(y, \hat{y})$$
- ➌ $\nabla_{\mathbf{w}} \mathcal{L}(y, \hat{y})$: "gradient" de \mathcal{L} par rapport à \mathbf{w} , i.e. le vecteur des dérivées de \mathcal{L} par rapport à w_0, w_1 , etc.
- ➍ Le gradient donne la direction de descente la plus rapide



$x_0 < x_1 < x_2 \dots$

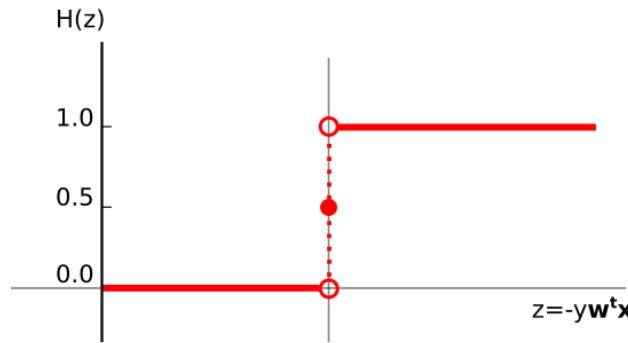


Revenons à l'autre partie de la question : quelle fonction de loss prendre ?

Trouver w : quelle loss ?

- Première idée spontanée : l'erreur binaire que fait le modèle, la "0-1 loss"

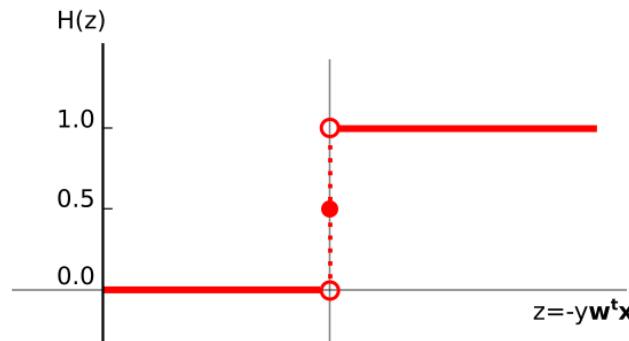
$$\begin{aligned}\mathcal{L}(y, \hat{y}) &= \begin{cases} 0 & \text{si } \hat{y} = y \\ 1 & \text{sinon} \end{cases} \\ &= \text{Heaviside}(-y\mathbf{w}^t \mathbf{x})\end{aligned}$$



Trouver w : quelle loss ?

- Première idée spontanée : l'erreur binaire que fait le modèle, la "0-1 loss"

$$\begin{aligned}\mathcal{L}(y, \hat{y}) &= \begin{cases} 0 & \text{si } \hat{y} = y \\ 1 & \text{sinon} \end{cases} \\ &= \text{Heaviside}(-y\mathbf{w}^t \mathbf{x})\end{aligned}$$



Problème : cette fonction n'a pas une bonne tête ! Pas convexe et pas lisse, donc pas utilisable pour une descente de gradient

Trouver w : quelle loss ?

En fait il nous faut faire deux choses :

- ① Compléter notre modèle linéaire en ajoutant une fonction non-linéaire (une "activation")
- ② Choisir une fonction de perte, une loss dérivable

Trouver w : ajout d'une activation

- ➊ Compléter notre modèle linéaire en ajoutant une fonction non-linéaire (une "activation"), **oui mais laquelle ?**

On a un score z pour la classe positive :

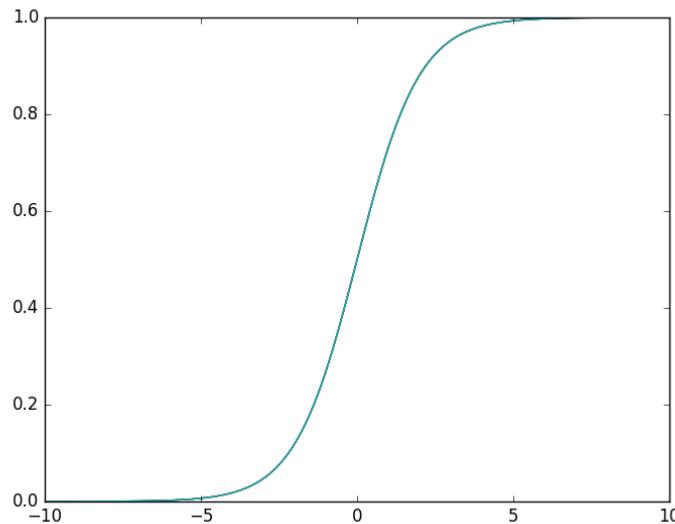
$$z = \mathbf{w}^t \mathbf{x}$$

On veut une fonction d'activation g qui ramène ce score dans l'intervalle $[0, 1]$ pour interpréter la valeur comme la probabilité de prédire la classe positive

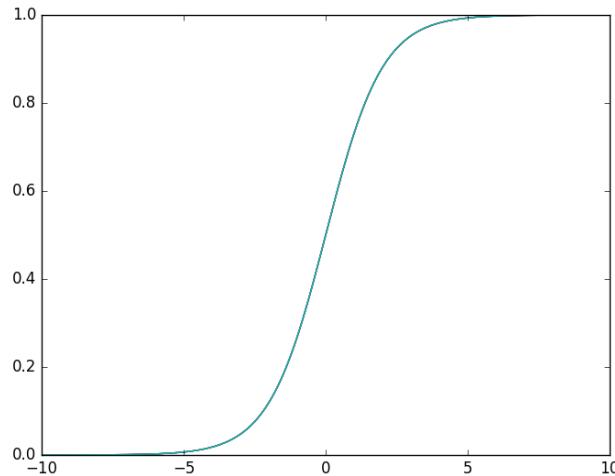
Trouver w : ajout d'une activation

On peut prendre la fonction "sigmoïde" (*aka* fonction "logistique") :

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



Trouver w : ajout d'une activation, la fonction sigmoïde



$$\begin{aligned}\hat{y} &= \begin{cases} 1 & \text{si } z > 0 \\ 0 & \text{sinon} \end{cases} \\ &= \begin{cases} 1 & \text{si } g(z) > 0.5 \\ 0 & \text{sinon} \end{cases}\end{aligned}$$

Remarque importante : les labels des données ne sont plus $\{-1, +1\}$ comme avec le perceptron mais $\{0, +1\}$

Trouver w : quelle loss ?

Critères de choix de la fonction de loss :

- lisse, dérivable et "convexe"
- pénalise de plus en plus fortement à mesure que le score de prédiction s'approche de 0 si $y = 1$
- même chose à mesure que le score de prédiction s'approche de 1 si $y = 0$

Trouver w : quelle loss ?

Critères de choix de la fonction de loss :

- lisse, dérivable et "convexe"
- pénalise de plus en plus fortement à mesure que le score de prédiction s'approche de 0 si $y = 1$
- même chose à mesure que le score de prédiction s'approche de 1 si $y = 0$

On pose que la prédiction est

$$\hat{y} = \text{sigmoide}(z) = \text{sigmoide}(\mathbf{w}^t \mathbf{x})$$

La loss usuelle pour la classification binaire s'appelle :

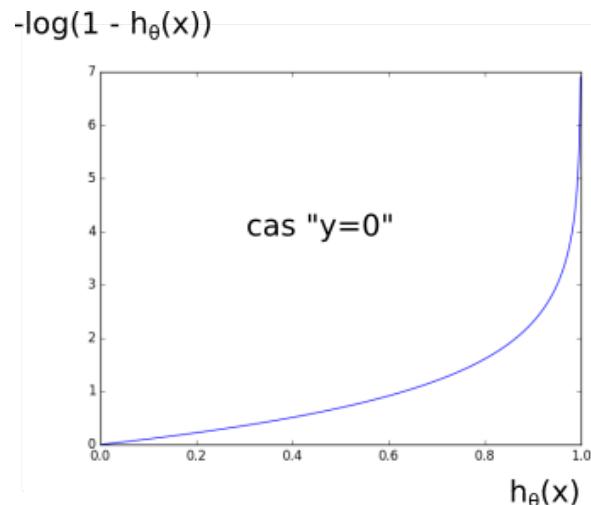
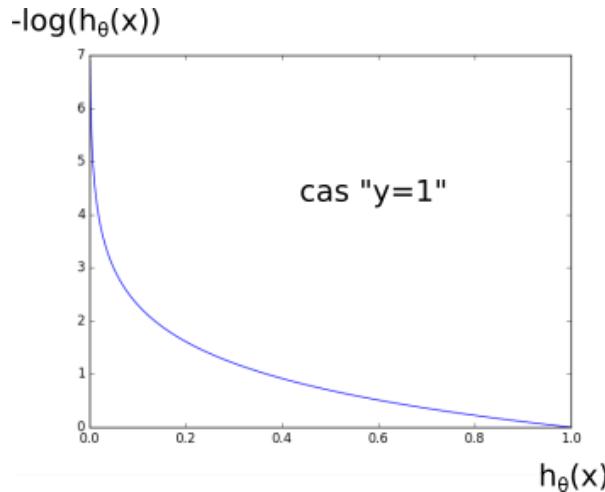
- l'entropie croisée binaire
- la *binary cross-entropy* (BCE) en anglais.

Elle est définie par :

$$\begin{aligned}\text{loss}(y, \hat{y}) &= \text{BCE}(y, \hat{y}) \\ &= -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})\end{aligned}$$

Trouver w : la BCE

$$\begin{aligned}\text{loss}(y, \hat{y}) &= \text{BCE}(y, \hat{y}) \\ &= -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})\end{aligned}$$



Régression logistique

- Le classifieur linéaire (binaire) qui utilise la sigmoïde et la BCE s'appelle une **régression logistique**
- La régression logistique est généralisable à une tâche de classification multiclass (à suivre)

Régression logistique en PyTorch

```
import torch
import torch.nn as nn

class LogisticRegression(nn.Module):

    def __init__(self, input_dim):
        super(LogisticRegression, self).__init__()
        self.linear = torch.nn.Linear(input_dim, 1)

    def forward(self, x):
        a = self.linear(x)
        y_pred = torch.sigmoid(a)
        return y_pred

model = LogisticRegression(10)

input = torch.randn(100, 10)
output = model(input)
print(output.size())

→ torch.Size([100, 1])
```

NEURAL NETWORK

Topics: multilayer neural network

- Could have L hidden layers:

- layer input activation for $k > 0$ ($\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$)

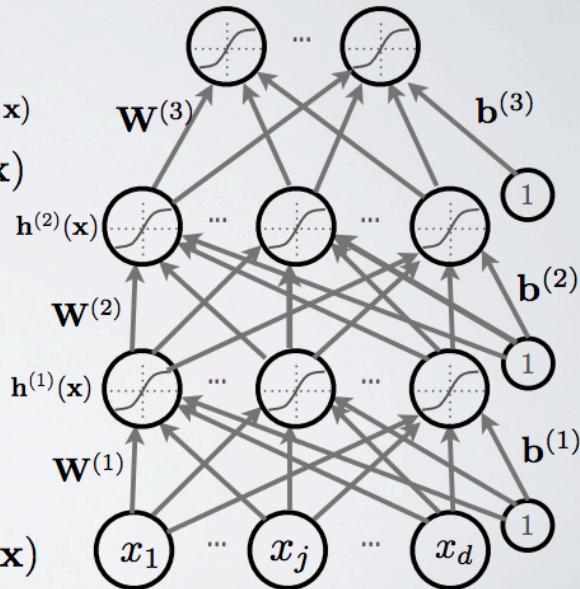
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

- hidden layer activation (k from 1 to L):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

- output layer activation ($k=L+1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



Fonctions d'activation g et o

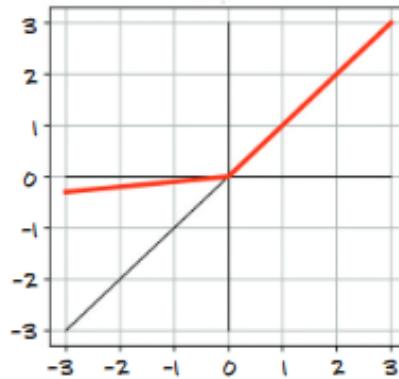
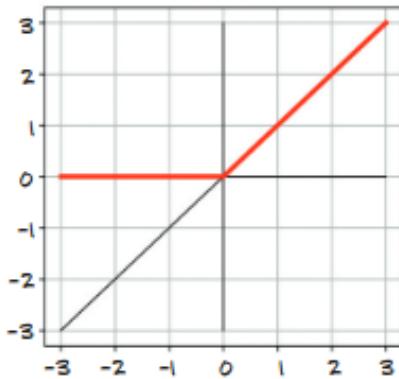
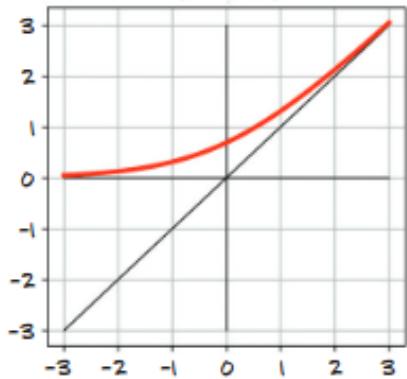
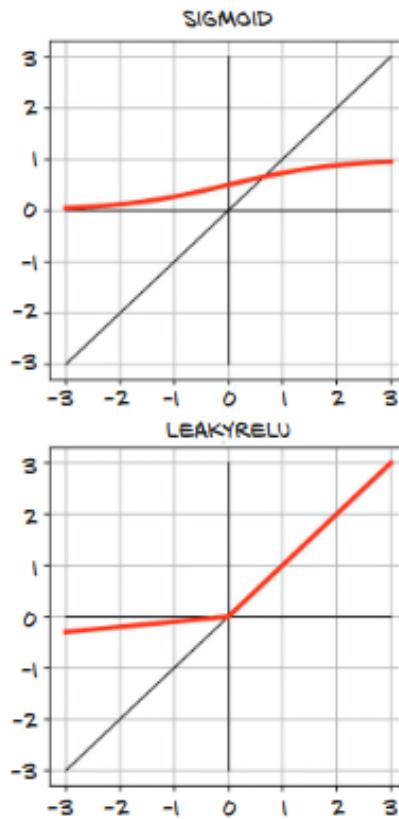
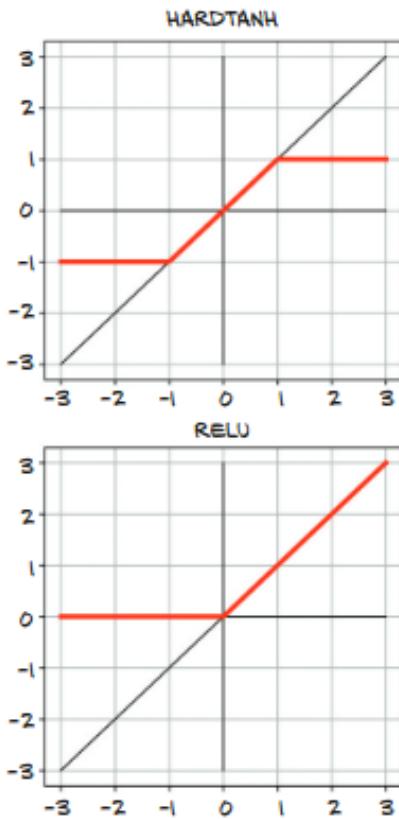
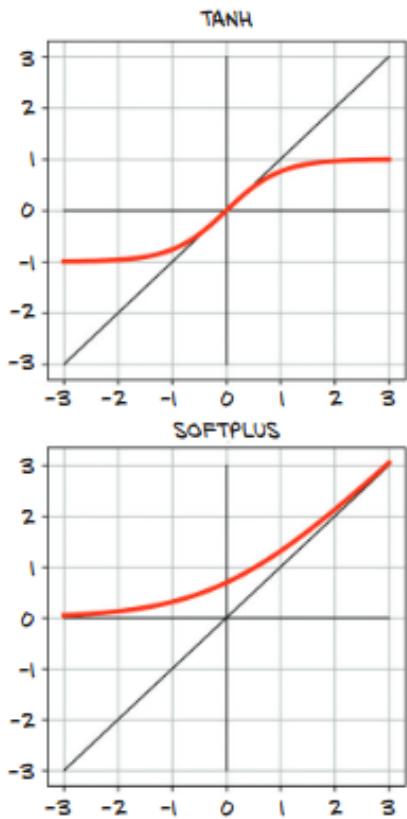
Fonctions d'activation vues dans les slides précédents :

- Tout-ou-rien (hard threshold) : pas utilisée dans les NN (à cause de la DG)
- Sigmoide (soft threshold)

Il en existe beaucoup d'autres...

<https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity>

Fonctions d'activation



NEURAL NETWORK

Topics: softmax activation function

- For multi-class classification:
 - we need multiple outputs (1 output per class)
 - we would like to estimate the conditional probability $p(y = c|\mathbf{x})$
- We use the softmax activation function at the output:

$$\mathbf{o}(\mathbf{a}) = \text{softmax}(\mathbf{a}) = \left[\frac{\exp(a_1)}{\sum_c \exp(a_c)} \cdots \frac{\exp(a_C)}{\sum_c \exp(a_c)} \right]^\top$$

- strictly positive
- sums to one
- Predicted class is the one with highest estimated probability

Comment entraîner les poids du MLP ?

- Les poids w sont initialisés aléatoirement
- Peut-on utiliser la règle du perceptron pour un MLP ?

Comment entraîner les poids du MLP ?

- Les poids w sont initialisés aléatoirement
- Peut-on utiliser la règle du perceptron pour un MLP ?

Réponse rapide : NON

Comment entraîner les poids du MLP ?

- Les poids w sont initialisés aléatoirement
- Peut-on utiliser la règle du perceptron pour un MLP ?

Réponse rapide : NON

Réponse rapide mais un peu moins : nous disposons des données d'entrée et des étiquettes désirées en sortie du réseau seulement. Il faudrait des étiquettes "intermédiaires" pour entraîner les neurones des couches couchés un par un...

Comment entraîner les poids du MLP ?

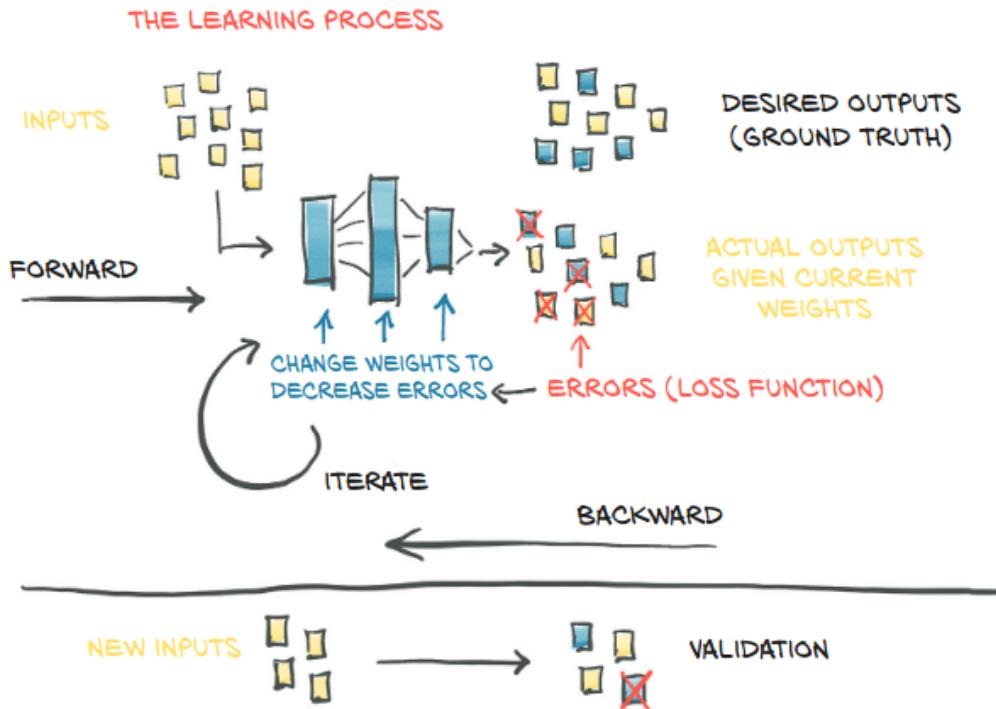
- Les poids w sont initialisés aléatoirement
- Peut-on utiliser la règle du perceptron pour un MLP ?

Réponse rapide : NON

Réponse rapide mais un peu moins : nous disposons des données d'entrée et des étiquettes désirées en sortie du réseau seulement. Il faudrait des étiquettes "intermédiaires" pour entraîner les neurones des couches couchés un par un...

On va plutôt faire une descente de gradient comme pour le modèle linéaire précédent

Apprentissage d'un réseau de neurones



Deep-Learning-with-PyTorch, p.107

Algorithme :

- On fait passer un jeu d'exemples (un "batch") dans le réseau : passe "forward"

Algorithme :

- On fait passer un jeu d'exemples (un "batch") dans le réseau : passe "forward"
- On calcule l'écart aux étiquettes désirées avec une "loss" (fonction de "coût" ou de "perte")

Algorithme :

- On fait passer un jeu d'exemples (un "batch") dans le réseau : passe "forward"
- On calcule l'écart aux étiquettes désirées avec une "loss" (fonction de "coût" ou de "perte")
- On calcule les gradients de la loss par rapport à chaque poids et biais, grâce à un algorithme astucieux appelé "rétropropagation", *backpropagation* : passe "backward"

Algorithme :

- On fait passer un jeu d'exemples (un "batch") dans le réseau : passe "forward"
- On calcule l'écart aux étiquettes désirées avec une "loss" (fonction de "coût" ou de "perte")
- On calcule les gradients de la loss par rapport à chaque poids et biais, grâce à un algorithme astucieux appelé "rétropropagation", *backpropagation* : passe "backward"
- On modifie les poids et biais avec les valeurs des gradients, selon l'une des nombreuses variantes d'optimisation (SGD, Adam, RAdam, etc)

Question : quelle quantité est rétropropagée ?

Algorithme :

- On fait passer un jeu d'exemples (un "batch") dans le réseau : passe "forward"
- On calcule l'écart aux étiquettes désirées avec une "loss" (fonction de "coût" ou de "perte")
- On calcule les gradients de la loss par rapport à chaque poids et biais, grâce à un algorithme astucieux appelé "rétropropagation", *backpropagation* : passe "backward"
- On modifie les poids et biais avec les valeurs des gradients, selon l'une des nombreuses variantes d'optimisation (SGD, Adam, RAdam, etc)

Question : quelle quantité est rétropropagée ?

⇒ le gradient de la fonction de perte

Classification : quelle loss utiliser ?

Une dizaine de choix possibles...

Voir <https://pytorch.org/docs/stable/nn.html#loss-functions>

Classification : quelle loss utiliser ?

- Cas de la classification : $y \in \{0, 1, 2, \dots, K - 1\}$

Question : comment mesurer l'erreur entre nos prédictions \hat{y} et la "vérité terrain" ou *groundtruth* ?

⇒ La fonction de perte dite "entropie croisée" ou "*cross-entropy loss*"

Il faut distinguer deux variantes :

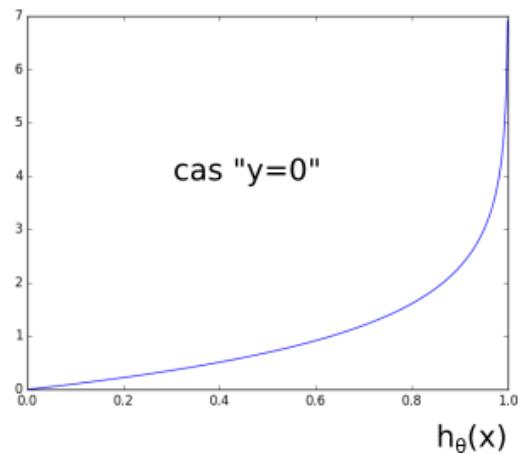
- entropie croisée binaire, *binary cross-entropy loss*
- entropie croisée, *cross-entropy loss* qui est une généralisation de la version binaire

Classification binaire : Binary Cross-entropy loss

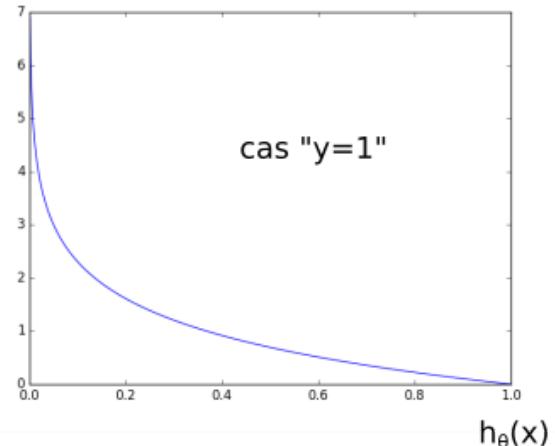
- Mesure l'erreur de probabilité pour des tâches de classification discrètes
- On a deux classes K=2 ou plus
- Les classes doivent être indépendantes et non-mutuellement exclusives
- On peut faire de la classification multi-catégorie comme, par exemple, détecter un éléphant et un chien dans une image

Classification binaire : Binary Cross-entropy loss (2)

$$-\log(1 - h_{\theta}(x))$$



$$-\log(h_{\theta}(x))$$



- Forte pénalisation des prédictions fausses faites avec une grande confiance !
- Pénalise les deux types d'erreur

Classification binaire : Binary Cross-entropy loss (3)

Formule avec

Vérité terrain : $y \in \{0, 1\}$

Prédiction du réseau : $\hat{y} = P(y = 1|x)$

$$\text{loss} = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

Code naïf :

```
def CrossEntropy(yHat, y):  
    if y == 1:  
        return -log(yHat)  
    else:  
        return -log(1 - yHat)
```

Classification binaire : Binary Cross-entropy loss (4)

Allons voir le code de cette fonction dans Keras :

<https://keras.io/losses/>

<https://github.com/keras-team/keras/blob/master/keras/losses.py>

```
def binary_crossentropy(y_true , y_pred):  
    return K.mean(K.binary_crossentropy(y_true , y_pred) , axis=-1)
```

Pas très informatif...

Classification binaire : Binary Cross-entropy loss (5)

Allons voir le code de cette fonction dans Tensorflow :

https://www.tensorflow.org/api_docs/python/tf/nn/sigmoid_cross_entropy_with_logits

For brevity, let `x = logits, z = labels`. The logistic loss is

$$\begin{aligned} & z * -\log(\text{sigmoid}(x)) + (1 - z) * -\log(1 - \text{sigmoid}(x)) \\ &= z * -\log(1 / (1 + \exp(-x))) + (1 - z) * -\log(\exp(-x) / (1 + \exp(-x))) \\ &= z * \log(1 + \exp(-x)) + (1 - z) * (-\log(\exp(-x)) + \log(1 + \exp(-x))) \\ &= z * \log(1 + \exp(-x)) + (1 - z) * (x + \log(1 + \exp(-x))) \\ &= (1 - z) * x + \log(1 + \exp(-x)) \\ &= x - x * z + \log(1 + \exp(-x)) \end{aligned}$$

For $x < 0$, to avoid overflow in $\exp(-x)$, we reformulate the above

$$\begin{aligned} & x - x * z + \log(1 + \exp(-x)) \\ &= \log(\exp(x)) - x * z + \log(1 + \exp(-x)) \\ &= -x * z + \log(1 + \exp(x)) \end{aligned}$$

Hence, to ensure stability and avoid overflow, the implementation uses this equivalent formulation

$$\max(x, 0) - x * z + \log(1 + \exp(-\text{abs}(x)))$$

`logits` and `labels` must have the same type and shape.

Classification binaire : Binary Cross-entropy loss (6)

Allons voir le code de cette fonction dans Tensorflow :

https://github.com/tensorflow/tensorflow/blob/r1.10/tensorflow/python/ops/nn_impl.py

```
def sigmoid_cross_entropy_with_logits(  
    labels=None,  
    logits=None):  
  
    # The logistic loss formula from above is  
    #   x - x * z + log(1 + exp(-x))  
    # For x < 0, a more numerically stable formula is  
    #   -x * z + log(1 + exp(x))  
    # Note that these two expressions can be combined into the following:  
    #   max(x, 0) - x * z + log(1 + exp(-abs(x)))  
    # To allow computing gradients at zero, we define custom versions of max and  
    # abs functions.  
    zeros = array_ops.zeros_like(logits, dtype=logits.dtype)  
    cond = (logits >= zeros)  
    relu_logits = array_ops.where(cond, logits, zeros)  
    neg_abs_logits = array_ops.where(cond, -logits, logits)  
    return math_ops.add(  
        relu_logits - logits * labels,  
        math_ops.log1p(math_ops.exp(neg_abs_logits)))
```

Remarque "pour aller plus loin"

Divergence de Kullback-Leibler

Pour deux distributions de probabilités discrètes P et Q , définies sur le même espace de probabilité \mathcal{X} , la divergence de Kullback–Leibler (KL) de P par rapport à Q est définie par :

$$D_{\text{KL}}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right)$$

La BCE est en fait la divergence de Kullback-Leibler entre les densités de probabilité :

- de la vérité terrain : $[d, 1 - d]$ avec $d=0$ ou 1 (probabilité dite "certaine")
- et de la sortie du réseau \hat{y} : $[\hat{y}, 1 - \hat{y}]$

$$\text{BCE}(d, \hat{y}) = \text{KL}(d, \hat{y})$$

Classification multiclass exclusive : Cross-entropy loss

- Vérité terrain : $y \in \{0, 1, \dots, K - 1\}$, classes **exclusives**
- Représentation de y utilisée : **one-hot** ($\Rightarrow \text{CE} = \text{KL}$)
- Prédiction du réseau :
 $\hat{y}_0 = P(y_0 = 1|x), \dots, \hat{y}_{K-1} = P(y_{K-1} = 1|x)$
- La loss, appelée CE, est la divergence de Kullback-Leibler entre les distributions y et \hat{y} :

$$\begin{aligned}\text{KL}(y, \hat{y}) &= \sum_{k=0}^{M-1} y_k \log\left(\frac{y_k}{\hat{y}_k}\right) = \sum_{k=0}^{M-1} y_k \log(y_k) - \sum_{k=0}^{M-1} y_k \log(\hat{y}_k) \\ &= -\log(\hat{y}_c)\end{aligned}$$

Classification multiclassse exclusive : Cross-entropy loss

$$\text{KL}(y, \hat{y}) = \text{CE}(y, \hat{y}) = -\log(\hat{y}_c)$$

$$\frac{d\text{KL}(y, \hat{y})}{d\hat{y}_i} = \begin{cases} -1/\hat{y}_c & \text{pour la composante } c \\ 0 & \text{pour les autres composantes} \end{cases}$$

- La dérivée est négative par rapport à \hat{y}_c donc si \hat{y}_c augmente, la loss diminue
- Remarque : quand $\hat{y}_c = y_c$, la loss est nulle mais pas la dérivée qui vaut -1

Différence CE et KL

- Divergence KL entre les distributions \hat{y} et y :

$$\text{KL}(y, \hat{y}) = \sum_{k=0}^{M-1} y_k \log(y_k) - \sum_{k=0}^{M-1} y_k \log(\hat{y}_k)$$

- Cross-entropy entre les distributions \hat{y} et y :

$$\text{CE}(y, \hat{y}) = - \sum_{k=0}^{M-1} y_k \log(\hat{y}_k)$$

- CE est simplement KL - entropie :

$$\text{CE}(y, \hat{y}) = \text{KL}(y, \hat{y}) - \sum_{k=0}^{M-1} y_k \log(y_k) = \text{KL}(y, \hat{y}) - H(y, \hat{y})$$

Différence CE et KL

- CE est égale à KL - entropie :

$$\text{CE}(y, \hat{y}) = \text{KL}(y, \hat{y}) - H(y)$$

- $H(y)$ ne dépend pas de \hat{y} donc minimiser CE revient à minimiser KL
- Avec y en one-hot, $H(y) = 0$ et $CE = KL$
- En pratique on minimise CE, et bien que CE atteigne son min en $\hat{y} = y$, sa valeur min n'est pas 0

Régression : erreur quadratique moyenne

Formule avec

Vérité terrain : $y \in \mathcal{R}$

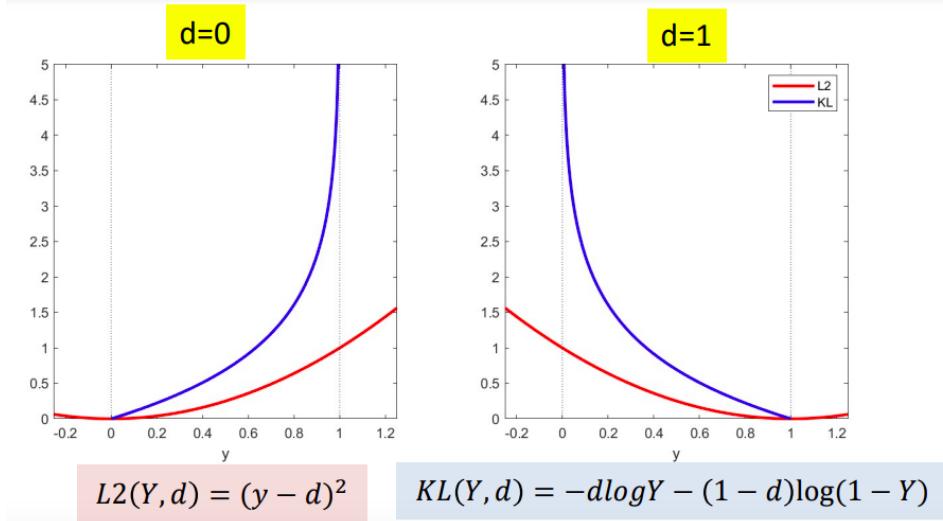
Prédiction du réseau : \hat{y}

$$\text{loss} = \frac{1}{2}(y - \hat{y})^2$$

Dans Keras : "mean_squared_error", alias "mse"

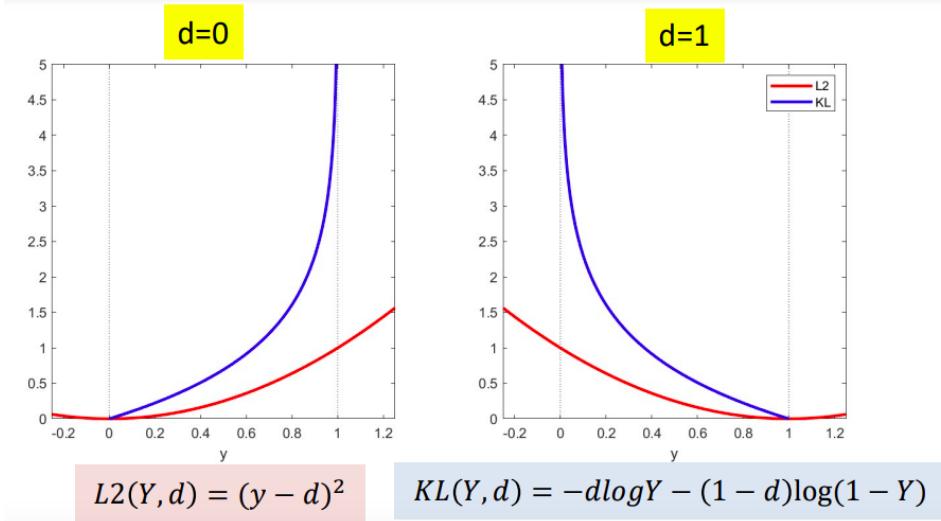
```
def mean_squared_error(y_true , y_pred):
    return K.mean(K.square(y_pred - y_true) , axis=-1)
```

KL (BCE) versus MSE (L2)



- KL et MSE ont leur minimum quand y est égal à la cible d
- KL augmente beaucoup plus vite que MSE quand y s'éloigne de d
 - Encourage une convergence plus rapide de la descente de gradient
- La dérivée de KL au minimum ne vaut pas 0, pour MSE oui

KL (BCE) versus MSE (L2)



- La dérivée de KL au minimum ne vaut pas 0, pour MSE oui

$$\frac{d\text{KL}(\hat{y}, y)}{d\hat{y}} = \begin{cases} -1/\hat{y} & \text{si } y = 1 \\ 1/(1 - \hat{y}) & \text{si } y = 0 \end{cases}$$

Descente de gradient pour la régression logistique

- Classifieur linéaire avec une sigmoïde pour obtenir $P(\hat{y} = 1|x)$:

$$z = \mathbf{w}^t \mathbf{x} + b$$

$$P(\hat{y} = 1|x) = \text{sigmoid}(z)$$

- Fonction de perte : *binary cross-entropy*
- Pourquoi ne pas prendre la fonction MSE ?

Bilan : quelle fonction de coût et quelle activation pour la couche de sortie ?

Les trois incontournables :

Classification binaire / multi-label

- Activation : sigmoïde, *sigmoid* ou *logistique*
- Coût : entropie croisée binaire, *binary cross-entropy*

Classification Multiclasse (classes exclusives)

- Activation : softmax
- Coût : entropie croisée, *categorical cross-entropy*

Régression

- Activation : linéaire, tanh ou sigmoïde, autre...
- Coût : erreur quadratique moyenne, *mean square error*

Idée :

- **Vu** : On fait passer un exemple dans le réseau : passe "forward"
- **Vu** : On calcule l'écart à l'étiquette désirée : fonction de "coût" ou de "perte", *loss function*
- **On calcule les gradients de la loss par rapport à chaque poids et biais, grâce à un algorithme astucieux appelé "rétropropagation", *backpropagation* : passe "backward"**
- On modifie les poids et biais avec les valeurs des gradients

Comment calculer les gradients de la loss ?

ARTIFICIAL NEURON

Topics: connection weights, bias, activation function

- Neuron pre-activation (or input activation):

$$a(\mathbf{x}) = b + \sum_i w_i x_i = b + \mathbf{w}^\top \mathbf{x}$$

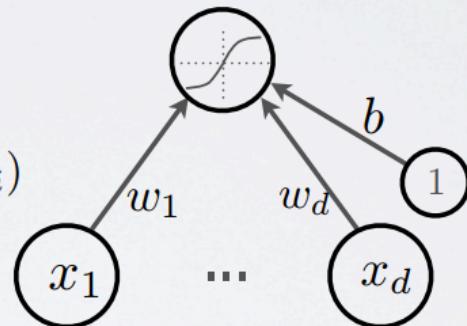
- Neuron (output) activation

$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

- \mathbf{w} are the connection weights

- b is the neuron bias

- $g(\cdot)$ is called the activation function



Calcul des gradients pour 1 neurone

- Avec activation sigmoïde et loss "mse"

a = préactivation

$$a = \sum_i w_i x_i + b$$

$$\frac{\partial a}{\partial w_i} = x_i$$

h = activation

$$h = \text{sigmoid}(a) = \frac{1}{1 + e^{-a}}$$

$$\frac{dh}{da} = h(1 - h)$$

Calcul des gradients pour 1 neurone

On applique la règle des dérivées à la chaîne, la "*chain rule*"

$$\frac{\partial h}{\partial w_i} = \frac{dh}{da} \frac{\partial a}{\partial w_i} = h(1 - h)x_i$$

$$\begin{aligned}\frac{\partial \text{loss}}{\partial w_i} &= \frac{\partial \text{loss}}{\partial h} \frac{\partial h}{\partial w_i} \\ &= -(y - h)h(1 - h)x_i\end{aligned}$$

Quels termes reconnaissiez-vous dans cette expression ?

Calcul des gradients pour 1 neurone

On applique la règle des dérivées à la chaîne, la "*chain rule*"

$$\frac{\partial h}{\partial w_i} = \frac{dh}{da} \frac{\partial a}{\partial w_i} = h(1 - h)x_i$$

$$\begin{aligned}\frac{\partial \text{loss}}{\partial w_i} &= \frac{\partial \text{loss}}{\partial h} \frac{\partial h}{\partial w_i} \\ &= -(y - h)h(1 - h)x_i\end{aligned}$$

Quels termes reconnaissiez-vous dans cette expression ?

la règle du perceptron et la pente de la sigmoïde

Notations : MLP à plusieurs couches cachées

NEURAL NETWORK

Topics: multilayer neural network

- Could have L hidden layers:

- layer pre-activation for $k > 0$ ($\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$)

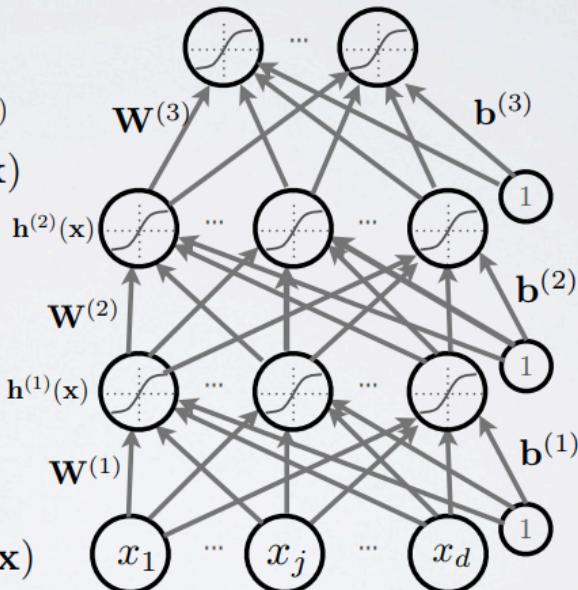
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

- hidden layer activation (k from 1 to L):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

- output layer activation ($k = L+1$):

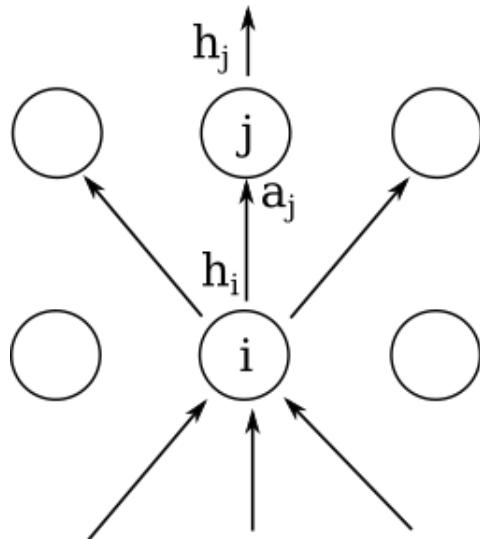
$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



Slide de Hugo Larochelle

Calcul des gradients pour un MLP

- Avec activation sigmoïde



$$\frac{\partial \text{loss}}{\partial a_j} = \frac{\partial \text{loss}}{\partial h_j} \frac{dh_j}{da_j} = h_j(1 - h_j) \frac{\partial \text{loss}}{\partial h_j}$$

$$\frac{\partial \text{loss}}{\partial h_i} = \sum_j \frac{da_j}{dh_i} \frac{\partial \text{loss}}{\partial a_j} = \sum_j w_{ij} \frac{\partial \text{loss}}{\partial a_j}$$

$$\frac{\partial \text{loss}}{\partial w_{ij}} = \frac{\partial a_j}{\partial w_{ij}} \frac{\partial \text{loss}}{\partial a_j} = h_i \frac{\partial \text{loss}}{\partial a_j}$$

Slide adapté de Geoffrey Hinton, Neural Networks for Machine Learning, Lecture 3d, The backpropagation algorithm

en pratique, comment calculer les gradients de la loss ?

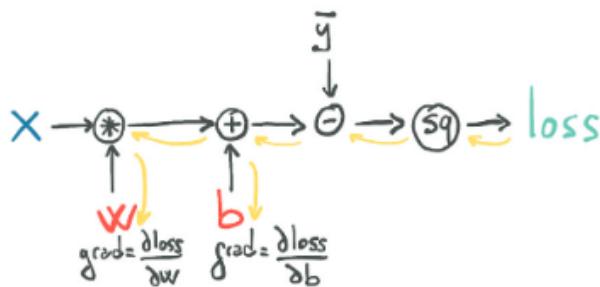
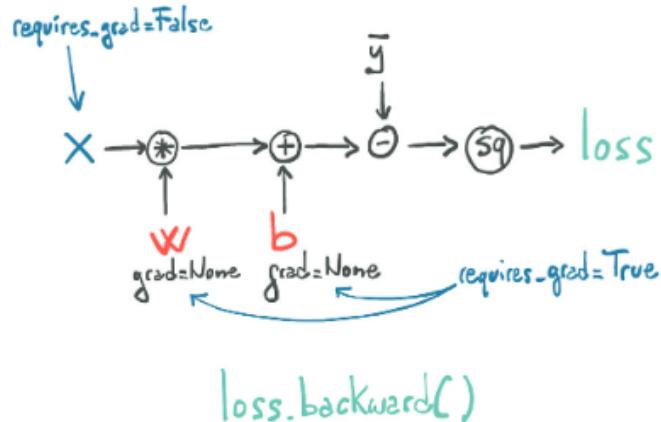
Réponse : on ne les calcule pas à la main, c'est automatique avec PyTorch !

```
▶ import torch

x = torch.ones(1, requires_grad=True)
y = x + 2
z = y * y * 2
z.backward()      # calcule le gradient automatiquement
print(x.grad)   # ∂z/∂x = 12

⇨ tensor([12.])
```

Comment calculer les gradients de la loss ?



- Quand la loss est calculée (passe forward), Pytorch crée un graphe avec le module Autograd
- Quand on appelle ensuite `loss.backward()`, Pytorch (Autograd) parcourt le graphe dans le sens opposé (passe "backward") pour calculer les gradients.

Graphes computationnels : exemple fonction sigmoide

Graphes computationnels

- Les graphes computationnels : utiles pour faire les calculs des passes forward et backward
- Les librairies de deep learning les utilisent
- Ainsi, les formules "directes" des gradients/dérivées ne sont pas utilisées en pratique
- Ils sont calculés avec les graphes, cela s'appelle l'auto-différenciation, *automatic differentiation*
- C'est le mode "reverse" de l'AD qui est utilisée, car la dimension des entrées est souvent beaucoup plus grande que la dimension de la sortie des réseaux

Actualisation des poids d'un réseau

- Les données de Train sont utilisées par "mini-lots", on parle de "minibatchs"
- Il est important que ces minibatchs soient choisis aléatoirement pour avoir une estimation non-biaisée du gradient moyen d'un ensemble d'exemples
- Beaucoup de techniques possibles pour l'actualisation des poids : Stochastic Gradient Descent (SGD), SGD avec moment, Adam et ses variantes, etc
- En deep learning, en pratique on n'utilise que des techniques du premier ordre (utilisant le gradient uniquement)
- Des techniques du second ordre (utilisant la Hessienne) nécessiteraient de trop grandes tailles de minibatchs.

Actualisation des poids d'un réseau

- SGD avec moment, *SGD with momentum*

$$v_t = \beta v_{t-1} + (1 - \beta) \nabla_W \text{loss}(y, \hat{y})$$

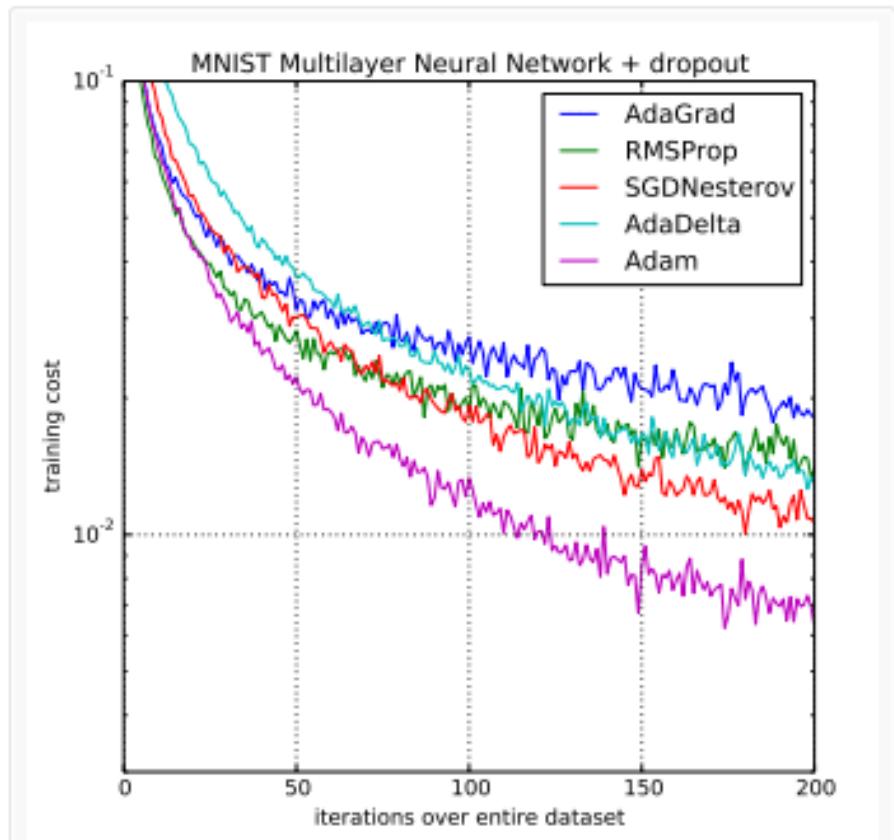
$$W = W - \alpha v_t$$

Formulé plus souvent par :

$$v_t = \beta v_{t-1} + \alpha \nabla_W \text{loss}(y, \hat{y})$$

$$W = W - v_t$$

Actualisation des poids d'un réseau



Importance de l'initialisation des poids

- L'initialisation peut avoir un impact significatif sur la convergence dans la formation des réseaux neuronaux profonds

L'initialisation des poids avec :

- une unique valeur constante (zéro ou autre) fait que tous les neurones apprennent la même chose : mauvais
- des valeurs trop petites : convergence trop lente voire nulle
- des valeurs trop grandes : divergence
- <http://www.deeplearning.ai/ai-notes/initialization/>

Importance de l'initialisation des poids

Dans PyTorch, méthodes d'initialisation par défaut des poids et biais :

- Couche linéaire : $U([-\sqrt{k}, \sqrt{k}])$, avec $k = 1/\text{in_features}$
- Couche conv2d : $U([-\sqrt{k}, \sqrt{k}])$, avec $k = 1/(C_{\text{in}} \times h \times w)$, où C_{in} est le nombre de channels d'entrée, h, w la taille des noyaux de convolution

Il y a des raisons théoriques de choisir telle ou telle méthode d'initialisation. Les plus utilisées sont les méthodes "Xavier" (appelée aussi "Glorot") et "He" du nom de leur inventeur.

- Xavier : quand la fonction d'activation est tanh
- He : ReLU et variantes

Importance de l'initialisation des poids

Distribution normale :

- Xavier : $N(0, 1/\text{in_features})$ or
 $N(0, 1/(\text{in_features} + \text{out_features}))$
- He : $N(0, 2/\text{in_features})$ or
 $N(0, 2/(\text{in_features} + \text{out_features}))$

<http://playground.tensorflow.org/>