Réseaux de neurones et deep learning : Utilisation et méthodologie

GEOFFREY DANIEL

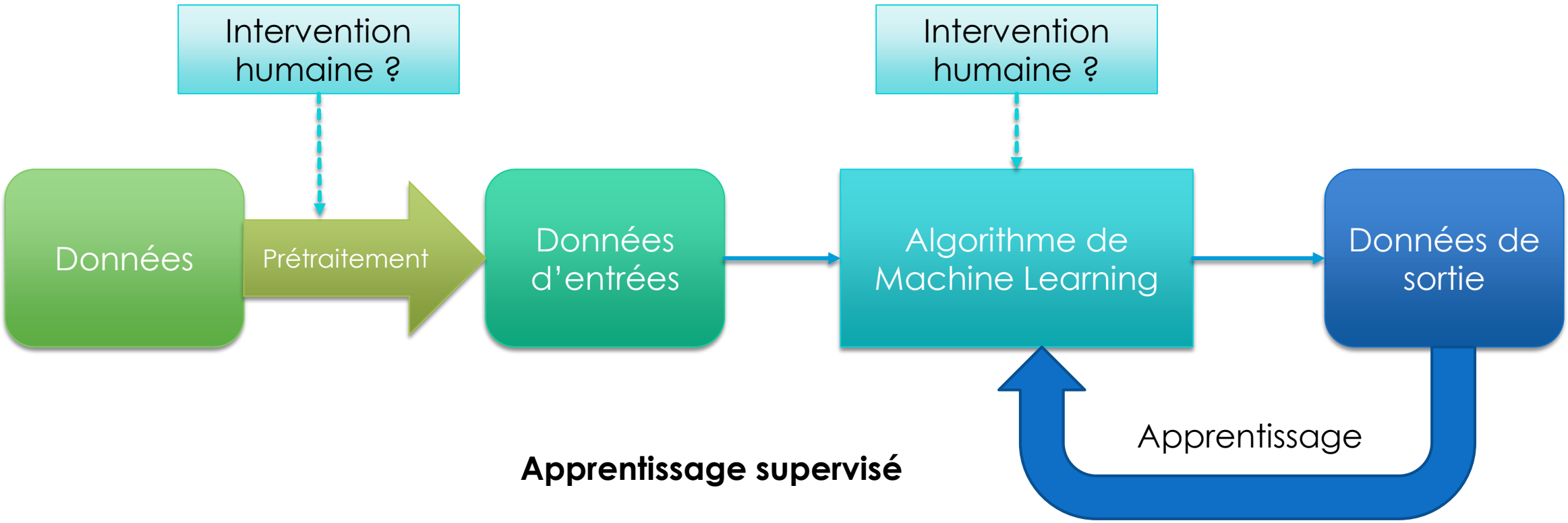
CEA/DES/ISAS/DM2S/STMF/LGLS

GEOFFREY.DANIEL@CEA.FR

Réseaux de neurones et deep learning

- **Partie 1 : Introduction aux réseaux de neurones**
  - **Utilisations courantes du deep learning**
  - **Bases générales**
- Partie 2 : Architecture des réseaux, hyperparamètres et évaluation des performances
  - Comment construire mon réseau et adapter la phase d'apprentissage ?
  - Comment évaluer les performances de mon réseau de neurones ?
- Partie 3 : Construction de la base de données
  - Éléments méthodologiques sur la mise en place du problème à résoudre potentiellement par deep learning
  - Comment utiliser l'évaluation des performances pour améliorer la base de données et le réseau ?
- Partie 4 : Réseaux de neurones convolutifs
  - Introduction à des structures plus avancées

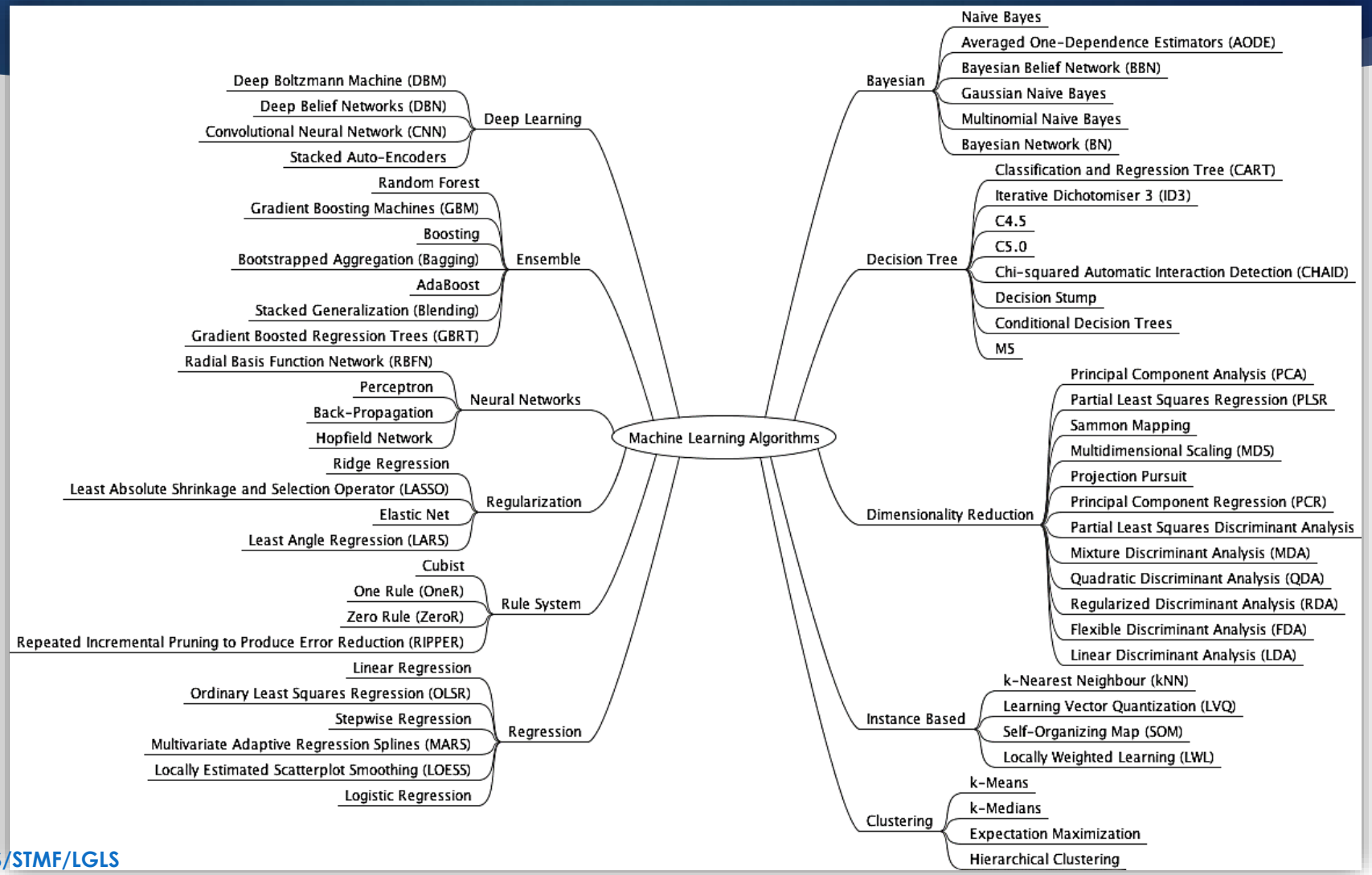
« Philosophie » du Machine Learning



Apprentissage supervisé

La forêt du Machine Learning

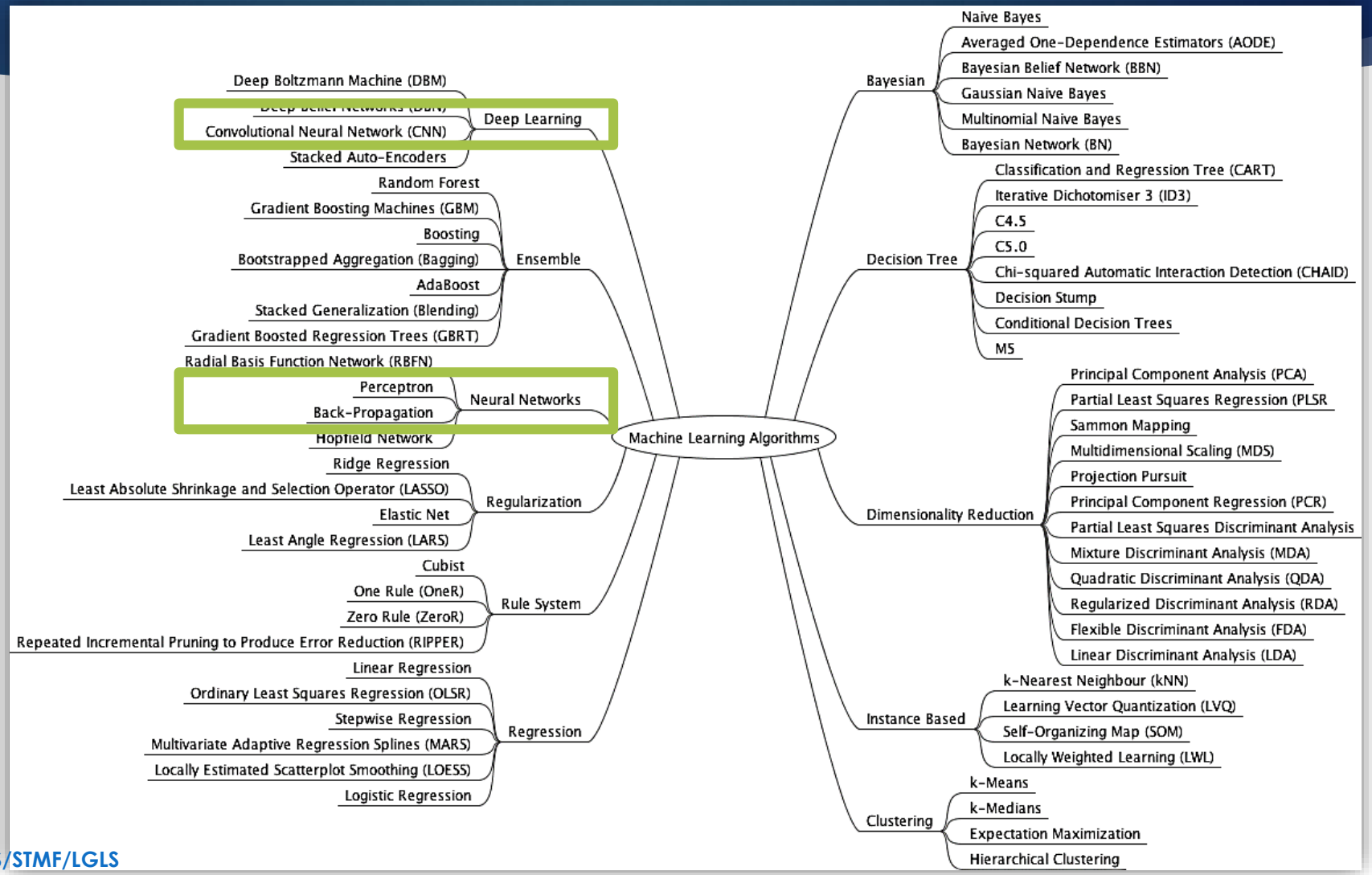
Et encore, ce n'est qu'une partie



Jason Brownlee
2013

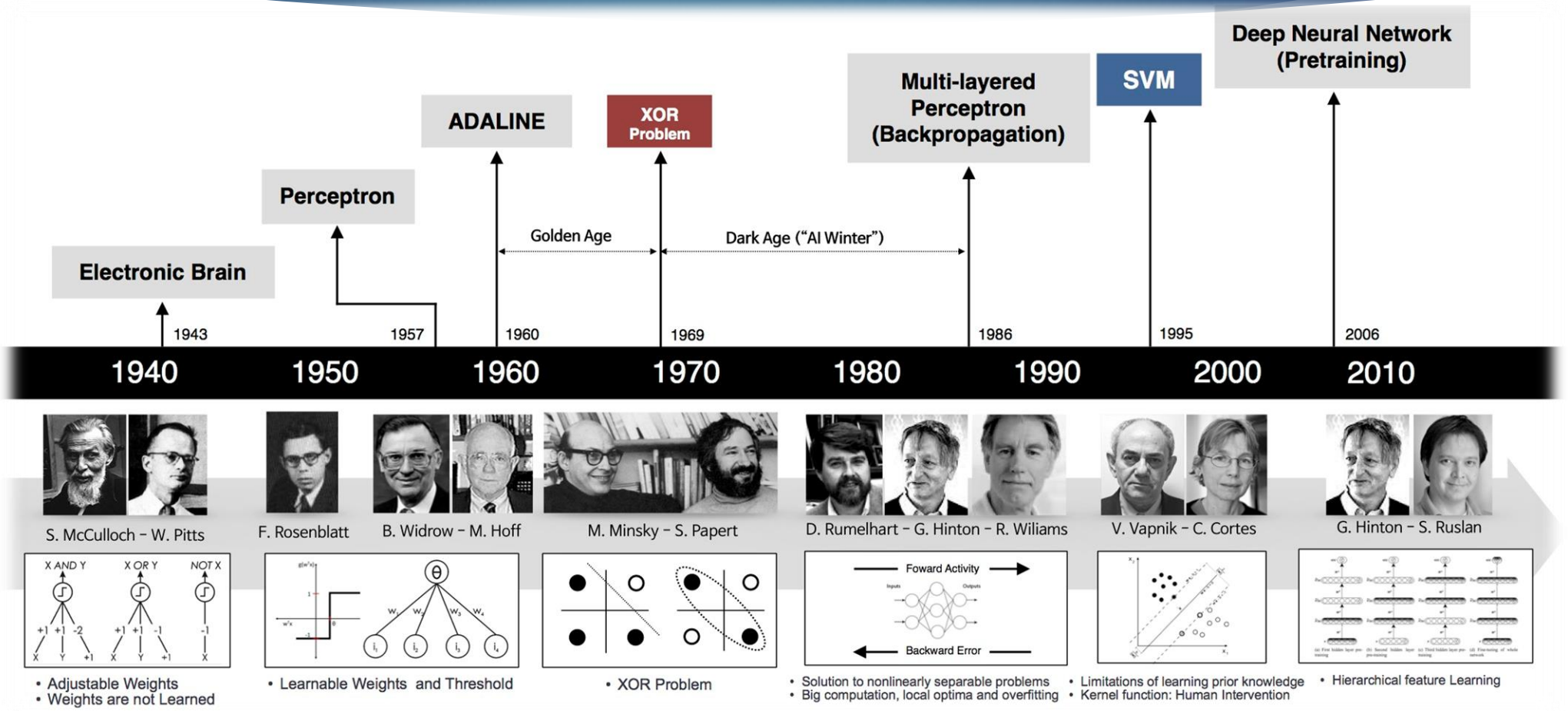
La forêt du Machine Learning

Et encore, ce n'est qu'une partie



Jason Brownlee
2013

Deep learning : un peu d'histoire

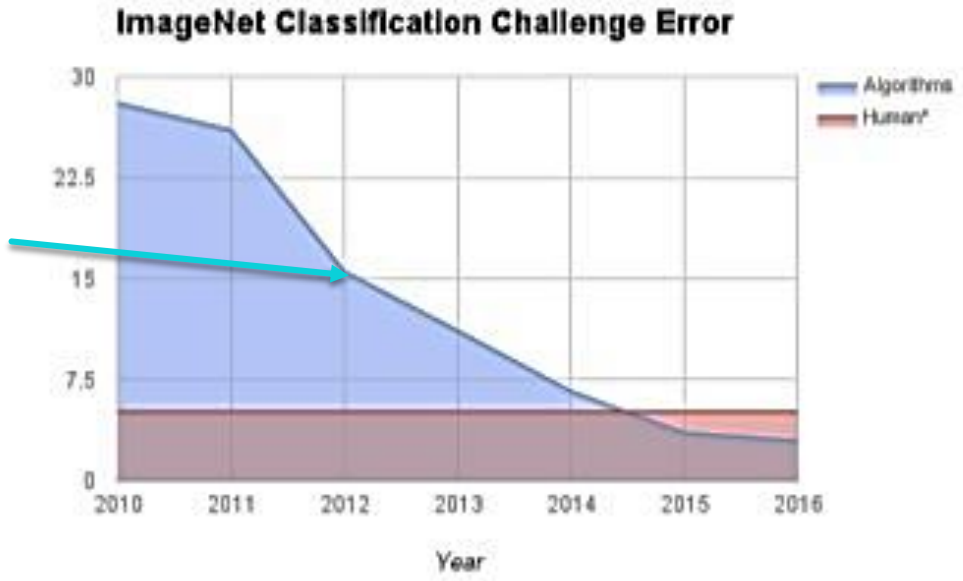


https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html

Deep learning : l'avènement

ImageNet Classification Challenge Error

Première utilisation du Deep learning dans la compétition

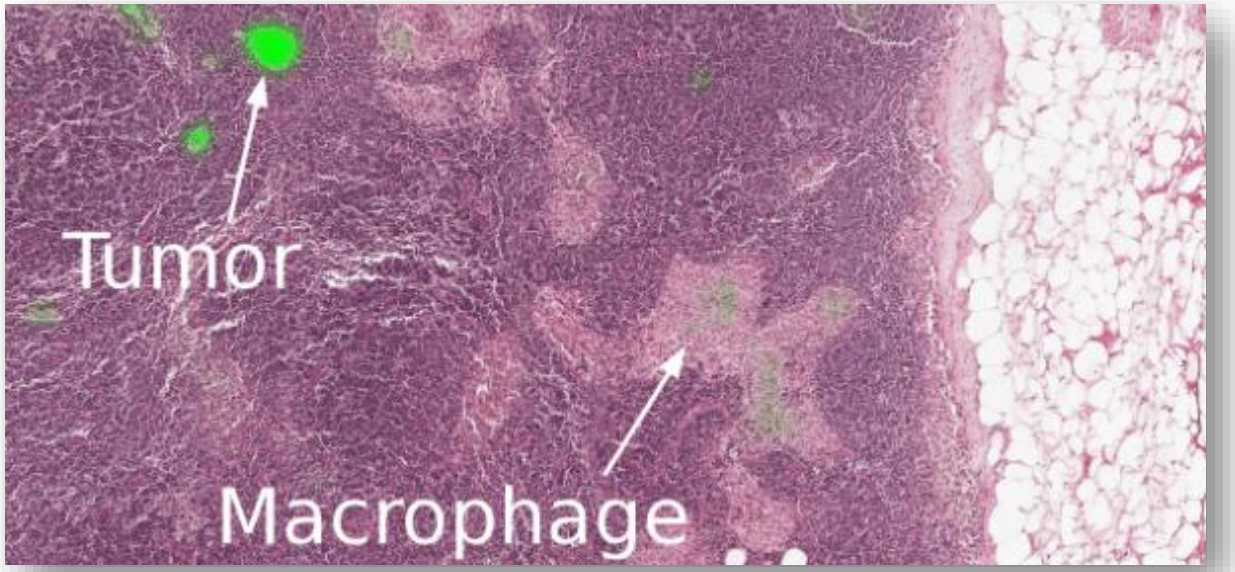


* Human Performance based on analysis done by Andrej Karpathy.
More details here.

Applications du deep learning

https://www.re-work.co/blog/deep-learning-daniel-mcduff-affectiva

Reconnaissance d'émotions

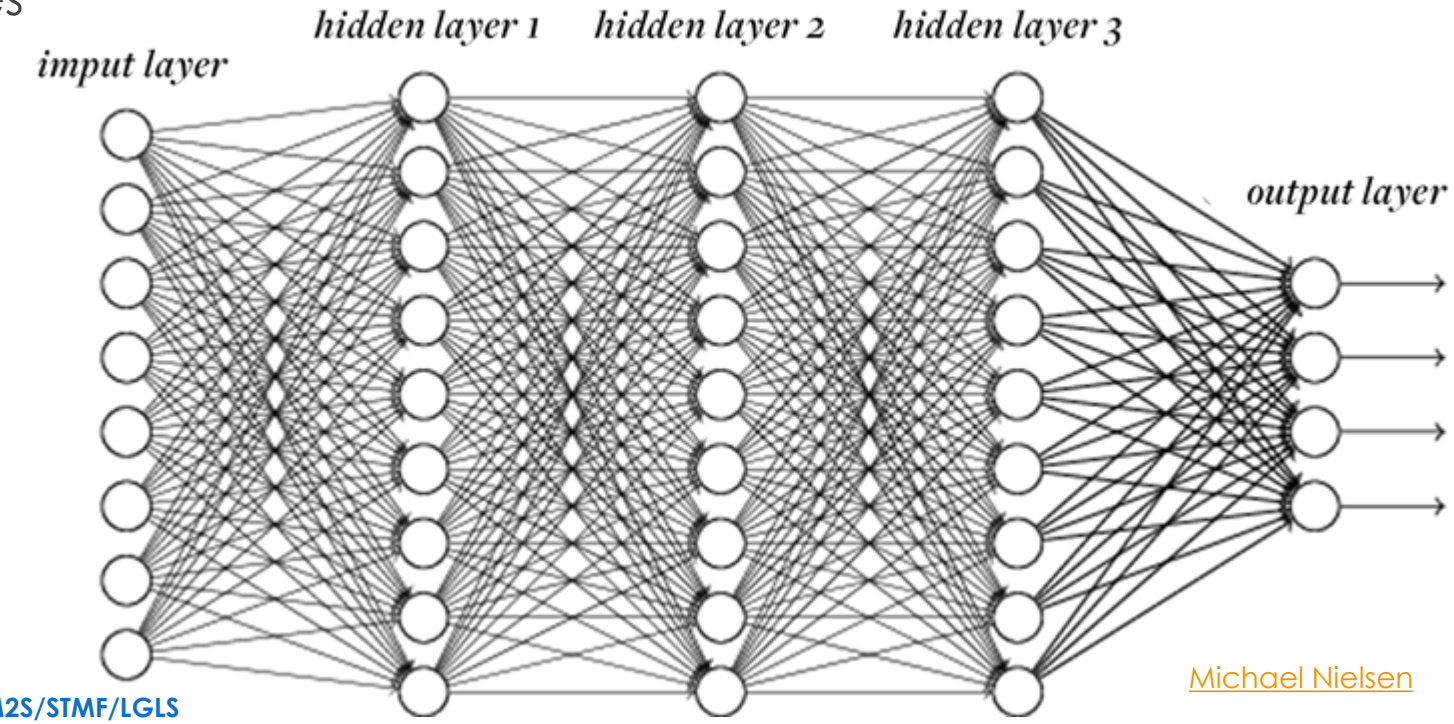


General News, Medical

Détection de cancers

Réseaux de neurones et deep learning

- Réseaux de neurones : Structure constituée d'un ensemble (couches) de briques élémentaires (neurones) effectuant chacune des opérations simples



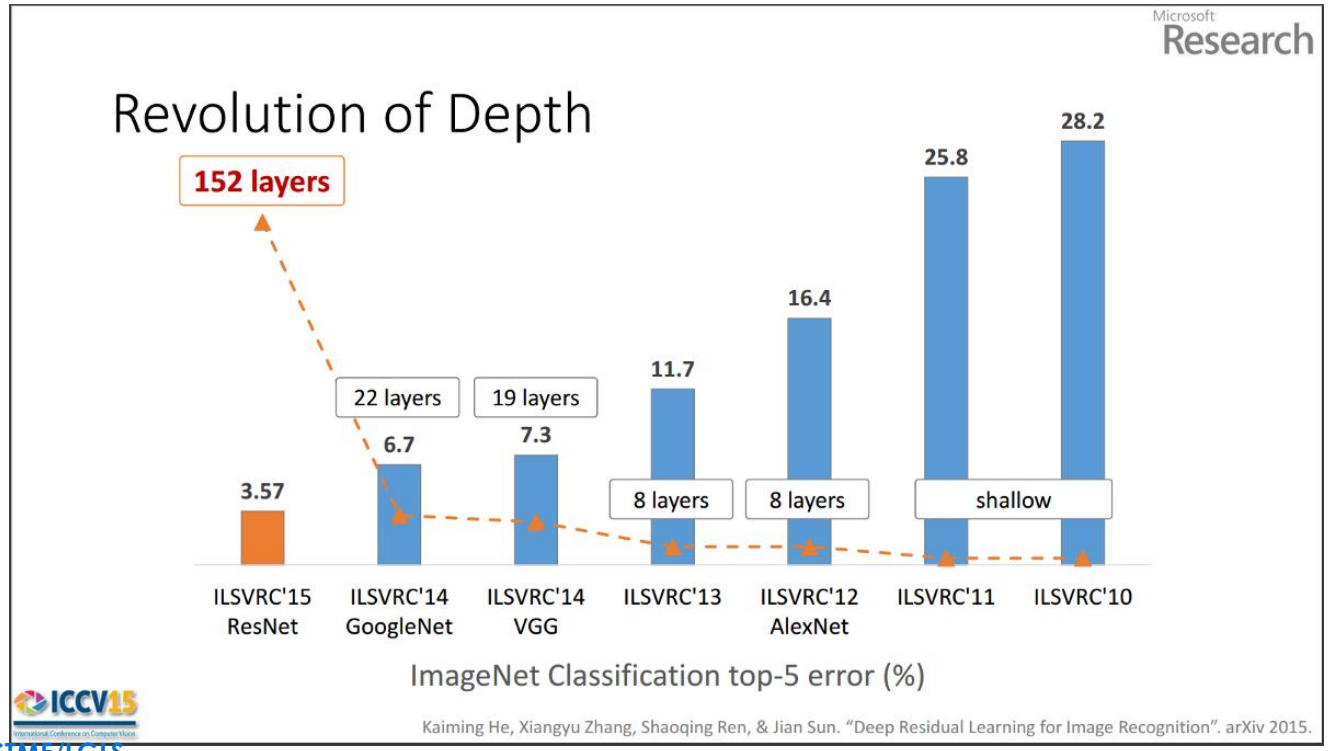
Michael Nielsen

Classification :
$\text{Output} \in \{0,1\}$

Régression :
$\text{Output} \in \mathbb{R}, [0,1], \mathbb{R}^{+*}$ ...

Réseaux de neurones et deep learning

- Apprentissage **profond :** nombre de couches élevé



Mathématiquement : le calcul d'un neurone

Vocabulaire :

X : données d'entrée (ou couche précédente)

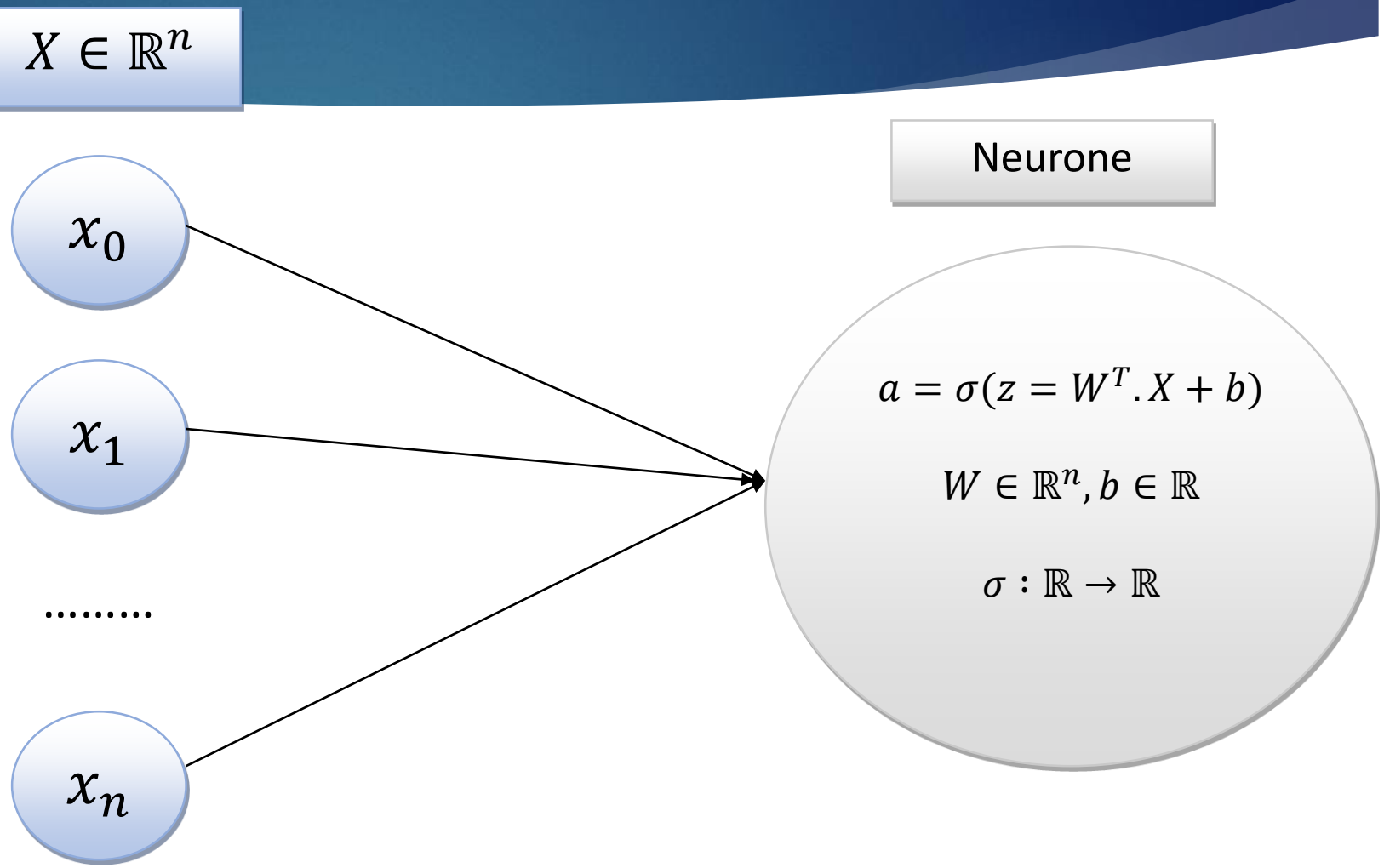
W : poids (weight)

b : biais (bias)

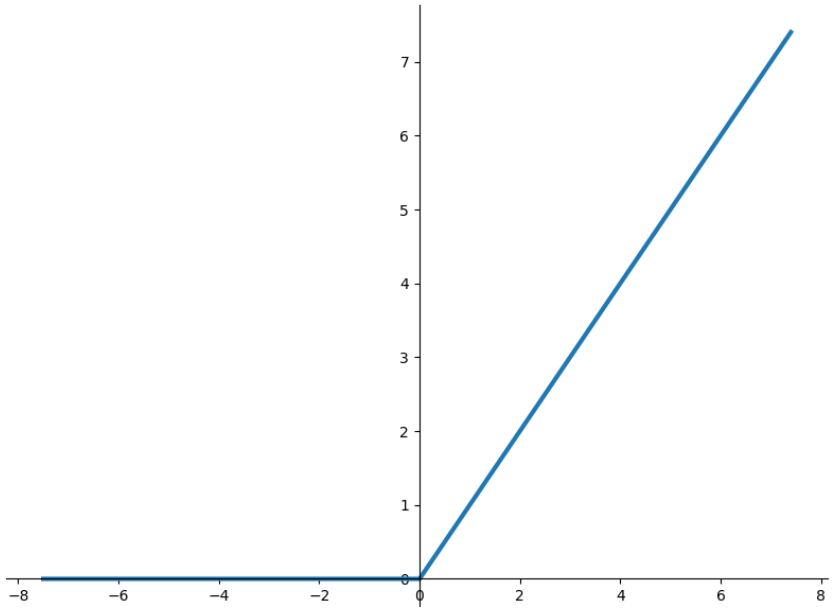
a : output du neurone

σ : fonction d'activation

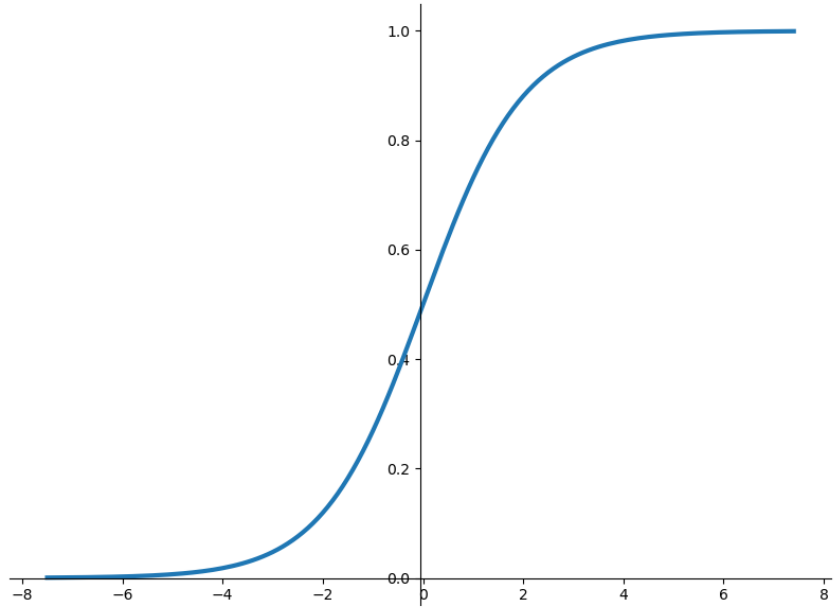
z : calcul intermédiaire



La fonction d'activation



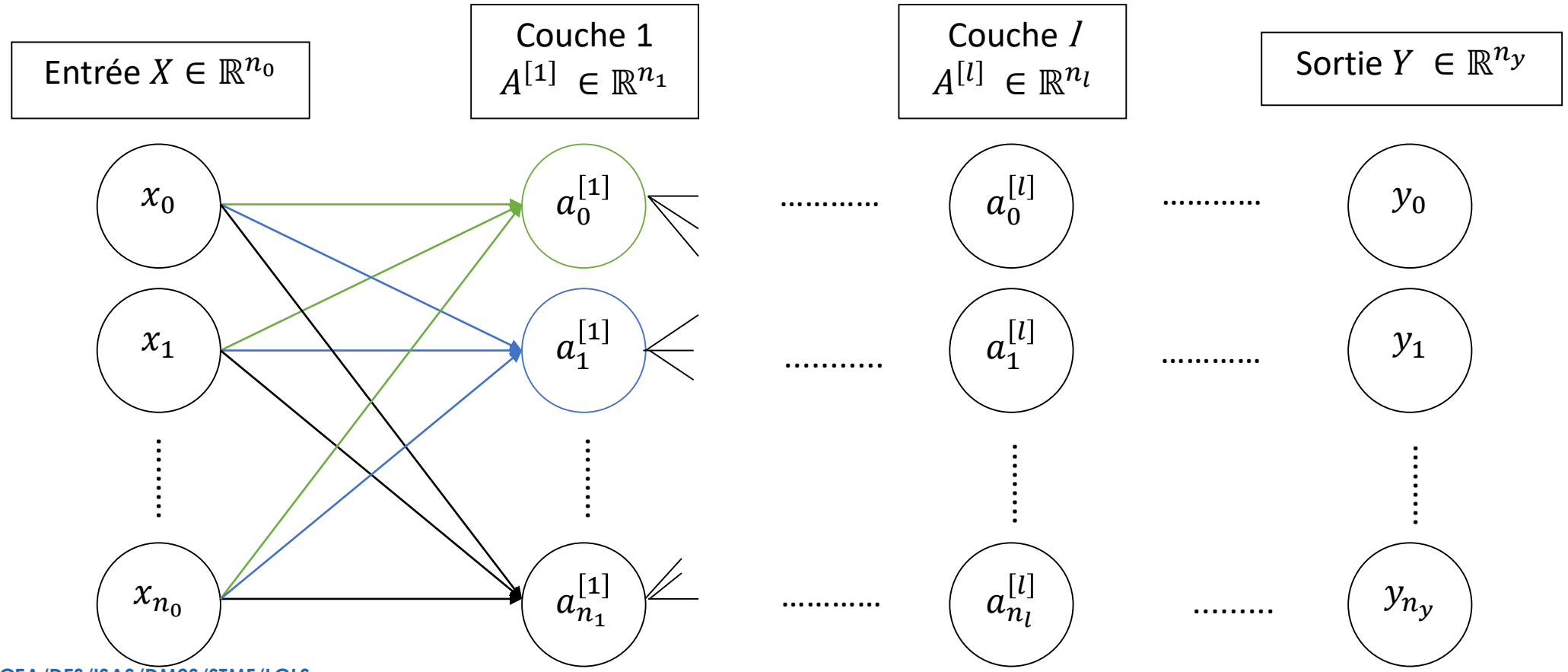
$\mathrm{ReLU}(x) = \max(x, 0)$: La plus utilisée
Simplicité de calcul, gradient non évanescent



$\mathrm{Sigmoïde}(x) = \frac{1}{1+e^{-x}}$: Pour la classification entre 0 et 1 en output

Et d'autres : $\tanh(x)$ (variante de la sigmoïde), $\mathrm{softmax}(x) = \frac{e^{-z}}{\sum_{z_0 \in \mathrm{output}} e^{-z_0}}$ (multi-classes exclusives),…

Et maintenant, un réseau



Théorème d'approximation universelle

- Soit $f : [0,1]^n \rightarrow [0,1]^m$. Pour tout $\varepsilon > 0$, il existe un réseau de neurones à une seule couche intermédiaire NN tel que $\|f - \mathrm{NN}\|_\infty < \varepsilon$
- Cela signifie que toute fonction bornée peut être approximée par un réseau de neurones.

- Condition nécessaire pour le fonctionnement des réseaux de neurones
  - Montre l'intérêt des réseaux de neurones
- Condition non suffisante en pratique :
  - Le théorème ne dit rien sur le nombre de neurones : en fait, pour un réseau monocouche, énormément de neurones peuvent être nécessaires selon la fonction f à approximer

La fonction de coût ou *loss function*

- **Évaluer** la qualité de la prédiction sur un jeu de données connues
- Notation : θ, paramètres du réseau (poids + biais) ; $\hat{Y}(\theta) = \left(\hat{y}_{ij}(\theta)\right)_{ij}$, output du réseau j pour l'exemple i ; $Y = \left(y_{ij}\right)_{ij}$, données réelles pour la valeur j du vecteur de sortie associé à l'exemple i
- *Loss function* :

$$L(\theta) = f\left(\hat{Y}(\theta), Y\right)$$

- Pendant l'apprentissage, la fonction de coût doit être minimisée : $\theta_{\text{optimaux}} = \underset{\theta}{\text{argmin}}\ \left(L(\theta)\right)$
- Monitoring de l'apprentissage :
  - On peut vérifier que la fonction de coût décroît bien à chaque itération sur notre jeu de données (voir parties 2 et 3)

Exemples de fonctions de coût

- Construction à partir de la vraisemblance des données
- Exemple en régression 1D :
  - On suppose que les données suivent une loi normale $\mathcal{N}(\hat{y}(\theta), \sigma^2)$, où $\hat{y}(\theta)$ est la prédiction du réseau de neurones de paramètres θ et σ est un écart-type, supposé identique sur l'ensemble des données
  - La vraisemblance des paramètres θ s'écrit alors :

$$\text{Likelihood}(\theta|Y) = \prod_i \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(\hat{y}_i(\theta) - y)^2}{\sigma^2}\right)$$

  - On va chercher les paramètres θ^* qui maximise cette vraisemblance. Comme la fonction log est strictement croissante, cela revient aussi à minimiser la négative log-vraisemblance :

$$-\log\big(\text{Likelihood}(\theta|Y)\big) = \sum_i \log\big(\sigma\sqrt{2\pi}\big) + \frac{1}{2}\frac{(\hat{y}_i(\theta) - y)^2}{\sigma^2}$$

  - En délaissant les termes constants qui n'ont pas d'influence sur le problème d'optimisation, on ne retient que la fonction de coût suivante :

$$L(\theta) = \sum_i (\hat{y}_i(\theta) - y)^2$$

  - On retrouve un problème de minimisation par moindre carré.
  - À dimension plus grande, cela conduit à la **distance euclidienne au carré ou Mean Squared Error** :

$$\boldsymbol{L(\theta)} = \sum_i \|\hat{\boldsymbol{y}}_i(\boldsymbol{\theta}) - \boldsymbol{y}\|_2^2$$

Exemples de fonctions de coût

- Construction à partir de la vraisemblance des données
- Exemple en classification avec k classes exclusives :
  - On suppose que les données suivent une loi de Bernoulli généralisée, de paramètres $\hat{y}_k(\theta)$ (prédiction du réseau de neurones).
  - $\hat{y}_k(\theta)$ est un nombre compris entre 0 et 1, et $\sum_k \hat{y}_k(\theta) = 1$. Ces propriétés peuvent être garanties en utilisant la fonction d'activation softmax en sortie du réseau de neurones.
  - La vraisemblance des paramètres θ s'écrit alors :

$$\text{Likelihood}(\theta|Y) = \prod_i \prod_k \hat{y}_{ik}(\theta)^{y_{ik}}$$

Ici y_{ik} vaut 1 si l'exemple i est associé à la classe k et vaut 0 sinon,

  - De nouveau, on va chercher à minimiser la négative log-vraisemblance, qui donne directement la fonction de coût :

$$\boldsymbol{L(\theta) = -\sum_i \sum_k y_{ik} \log\left(\hat{y}_{ik}(\theta)\right)}$$

  - Cette fonction de coût s'appelle **l'entropie croisée catégorielle ou categorical cross-entropie**

Exemples de fonctions de coût

- Plusieurs exemples
  - En régression, Mean Squared Error pour estimer la moyenne : $\boldsymbol{L(\theta) = \sum_i \|\hat{y}_i(\theta) - y\|_2^2}$
  - En régression, Mean Absolute Error pour estimer la médiane : $\boldsymbol{L(\theta) = \sum_i \|\hat{y}_i(\theta) - y\|_1^2}$
  - En classification multi-classes exclusives, categorical cross-entropy : $\boldsymbol{L(\theta) = -\sum_i \sum_k y_{ik} \log\left(\hat{y}_{ik}(\theta)\right)}$
  - En classification multi-classes non exclusives, binary cross-entropy : $\boldsymbol{L(\theta) = -\sum_i \sum_k y_{ik} \log\left(\hat{y}_{ik}(\theta)\right) + (1 - y_{ik}) \log\left(1 - \hat{y}_{ik}(\theta)\right)}$
  - D'autres fonctions de coût existent, elles doivent être adaptées au problème à résoudre

- Fonctions généralement non convexes !!! Minima locaux possibles, mais :
  - Plusieurs minima locaux aussi « bons » vis-à-vis de la fonction de coût (Yann Le Cun)
  - On peut tomber dans un mauvais minimum, mais ceci est rare : différentes techniques permettent d'éviter cela (dropout, régularisation… voir partie 2)

L'apprentissage

- Supposons que nous avons un jeu de données d'entrées X_i et de sorties Y_i et un réseau de neurones avec les paramètres $\theta = (W, B)$ qui prédit les sorties $\hat{Y}_i$ à partir des données X_i
- Minimisation de la fonction de coût L par descente de gradient (itérations) :

$$\theta := \theta - \lambda \nabla \mathrm{L}(\theta)$$

λ est le taux d'apprentissage : valeur définie ou adaptée à chaque itération pour assurer la convergence

- Intérêt des réseaux de neurones : le gradient de la fonction de coût L se calcule « facilement », par succession de calculs élémentaires appelée backpropagation

Calcul du gradient : backpropagation

- Pour chaque poids $w_k^{[l]}$ et biais $b_k^{[l]}$ du neurone k de la couche l, on veut calculer pour chaque exemple i :

$$\frac{\partial L_i}{\partial w_k^{[l]}} ; \frac{\partial L_i}{\partial b_k^{[l]}}$$

- **Pour la dernière couche n** :

$$\frac{\partial L_i}{\partial w_k^{[n]}} = \frac{\partial L_i}{\partial \widehat{Y_{ik}}} \frac{\partial \widehat{Y_{ik}}}{\partial w_k^{[n]}} ; \frac{\partial L_i}{\partial b_k^{[n]}} = \frac{\partial L}{\partial \widehat{Y_{ik}}} \frac{\partial \widehat{Y_{ik}}}{\partial b_k^{[n]}}$$

$$\widehat{Y_{ik}} = \sigma\left(z_k^{[n]} = {w_k^{[n]}}^T a^{[n-1]} + b_k^{[n]}\right)$$

$$\frac{\partial \widehat{Y_{ik}}}{\partial w_k^{[n]}} = \frac{\partial \widehat{Y_{ik}}}{\partial z_k^{[n]}} \frac{\partial z_k^{[n]}}{\partial w_k^{[n]}} = \sigma'\left(z_k^{[n]}\right) a^{[n-1]} \in \mathbb{R}^{[m_{n-1}]} ; \frac{\partial \widehat{Y_{ik}}}{\partial b_k^{[n]}} = \frac{\partial \widehat{Y_{ik}}}{\partial z_k^{[n]}} \frac{\partial z_k^{[n]}}{\partial b_k^{[n]}} = \sigma'\left(z_k^{[n]}\right) \in \mathbb{R}$$

Calcul du gradient : backpropagation

$$\frac{\partial L_i}{\partial w_k^{[n]}} = \frac{\partial L_i}{\partial \widehat{Y_{ik}}} \sigma'\left(z_k^{[n]}\right) a^{[n-1]} \; ; \; \frac{\partial L_i}{\partial b_k^{[n]}} = \frac{\partial L_i}{\partial \widehat{Y_{ik}}} \sigma'\left(z_k^{[n]}\right)$$

- Exemple avec :

$$L_i = \left(\hat{Y}_i(\theta) - Y_i\right)^2$$

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

- On obtient :

$$\sigma'\left(z_k^{[n]}\right) = \sigma\left(z_k^{[n]}\right)\left(1 - \sigma\left(z_k^{[n]}\right)\right) = \widehat{Y_{ik}}\left(1 - \widehat{Y_{ik}}\right)$$ Propriété du sigmoïde

$$\frac{\partial L}{\partial \widehat{Y_{ik}}} = 2\left(\widehat{Y_{ik}} - Y_{ik}\right)$$

Calcul du gradient : backpropagation

- **Pour les autres couches $l \neq n$** : de manière récursive

En rouge : par forward pass
En bleu : par récursivité

$$\frac{\partial L_i}{\partial w_k^{[l]}} = \frac{\partial L_i}{\partial a_k^{[l]}} \frac{\partial a_k^{[l]}}{\partial w_k^{[l]}} = \frac{\partial L_i}{\partial a_k^{[l]}} \sigma'\left(z_k^{[l]}\right) a^{[l-1]} \; ; \; \frac{\partial L_i}{\partial b_k^{[l]}} = \frac{\partial L_i}{\partial a_k^{[l]}} \frac{\partial a_k^{[l]}}{\partial b_k^{[l]}} = \frac{\partial L_i}{\partial a_k^{[l]}} \sigma'\left(z_k^{[l]}\right) ; \; (\text{pour } l = 1, a^{[0]} = X)$$

$$\frac{\partial L_i}{\partial a_k^{[l]}} = \sum_j \frac{\partial L_i}{\partial a_j^{[l+1]}} \frac{\partial a_j^{[l+1]}}{\partial a_k^{[l]}}$$

$$a_j^{[l+1]} = \sigma\left(z_j^{[l+1]} = \sum_{k\prime} \left(w_j^{[l+1]}\right)_{k\prime} a_{k\prime}^{[l]} + b_j^{[l+1]}\right) \Rightarrow \frac{\partial a_j^{[l+1]}}{\partial a_k^{[l]}} = \left(w_j^{[l+1]}\right)_k \sigma'\left(z_j^{[l+1]}\right)$$

Ajustement des paramètres

- On connaît $\frac{\partial L_i}{\partial w_k^{[l]}}$ et $\frac{\partial L_i}{\partial b_k^{[l]}}$ pour chaque exemple i
- Finalement :

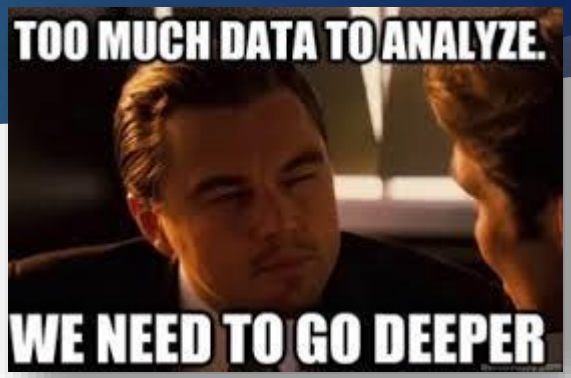
$$w_k^{[l]} := w_k^{[l]} - \lambda \frac{1}{N_{\text{exemples}}} \sum_i \frac{\partial L_i}{\partial w_k^{[l]}}$$
$$b_k^{[l]} := b_k^{[l]} - \lambda \frac{1}{N_{\text{exemples}}} \sum_i \frac{\partial L_i}{\partial b_k^{[l]}}$$

- On peut ne travailler simultanément que sur des sous-ensembles de la base de données (mini-batch), cela peut accélérer les calculs (voir parties 2 et 3)

Résumé des points importants

- Réseaux de neurones : calculs élémentaires $a = \sigma(z = W^T.X + b)$
- Chercher les paramètres $\theta = (W, b)$ qui minimisent une fonction de coût sur la base de données d'apprentissage : $\theta_{\text{optimaux}} = \underset{\theta}{\text{argmin}} \left(f\left(\hat{Y}(\theta), Y\right)\right)$
- Apprentissage par descente de gradient : $\theta \coloneqq \theta - \lambda \nabla \mathrm{L}(\theta)$

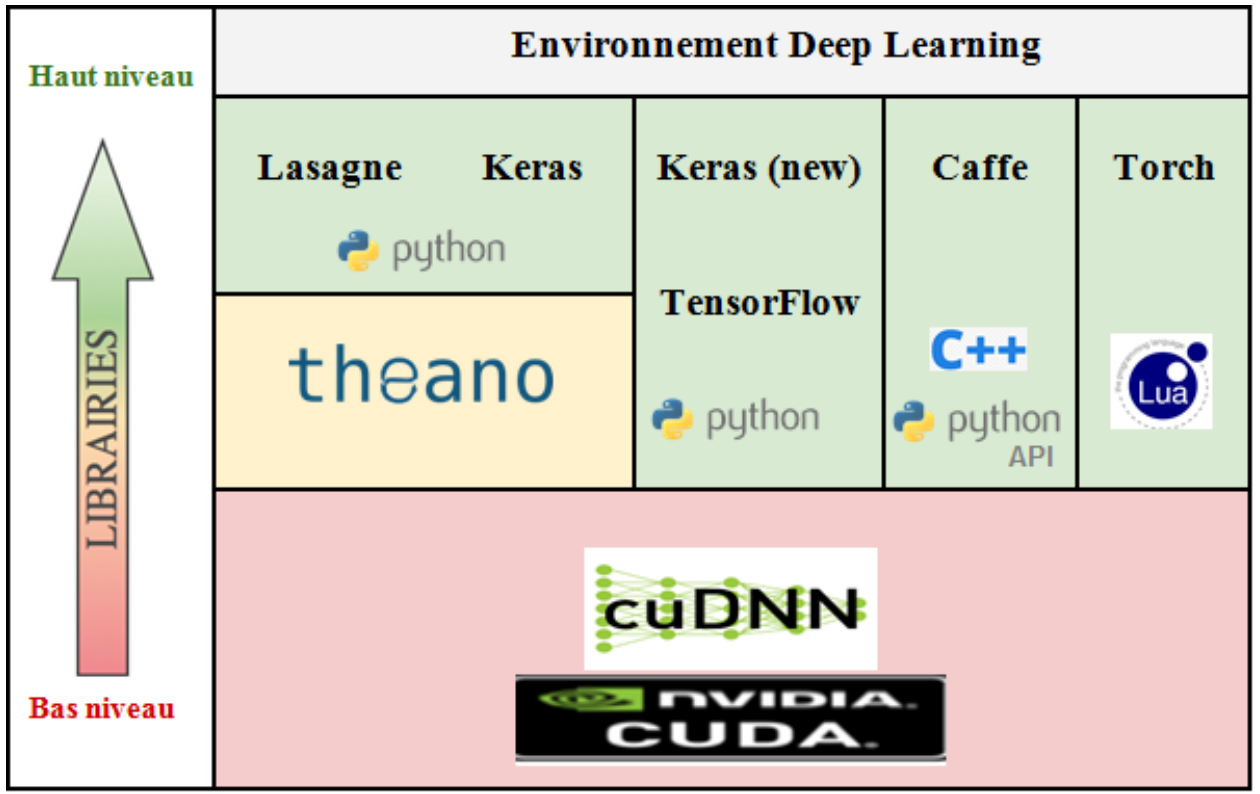
Réseaux de neurones et deep learning



- Partie 1 : Introduction aux réseaux de neurones
  - Utilisations courantes du deep learning
  - Bases générales
- **Partie 2 : Architecture des réseaux, hyperparamètres et évaluation des performances**
  - **Comment construire mon réseau et adapter la phase d'apprentissage ?**
  - **Comment évaluer les performances de mon réseau de neurones ?**
- Partie 3 : Construction de la base de données
  - Éléments méthodologiques sur la mise en place du problème à résoudre potentiellement par deep learning
  - Comment utiliser l'évaluation des performances pour améliorer la base de données et le réseau ?
- Partie 4 : Réseaux de neurones convolutifs
  - Introduction à des structures plus avancées

Quelques librairies

Pour Python et C++ :



https://blog.octo.com/classification-dimages-les-reseaux-de-neurones-convolutifs-en-toute-simplicite/

Matlab : toolbox « deep learning »

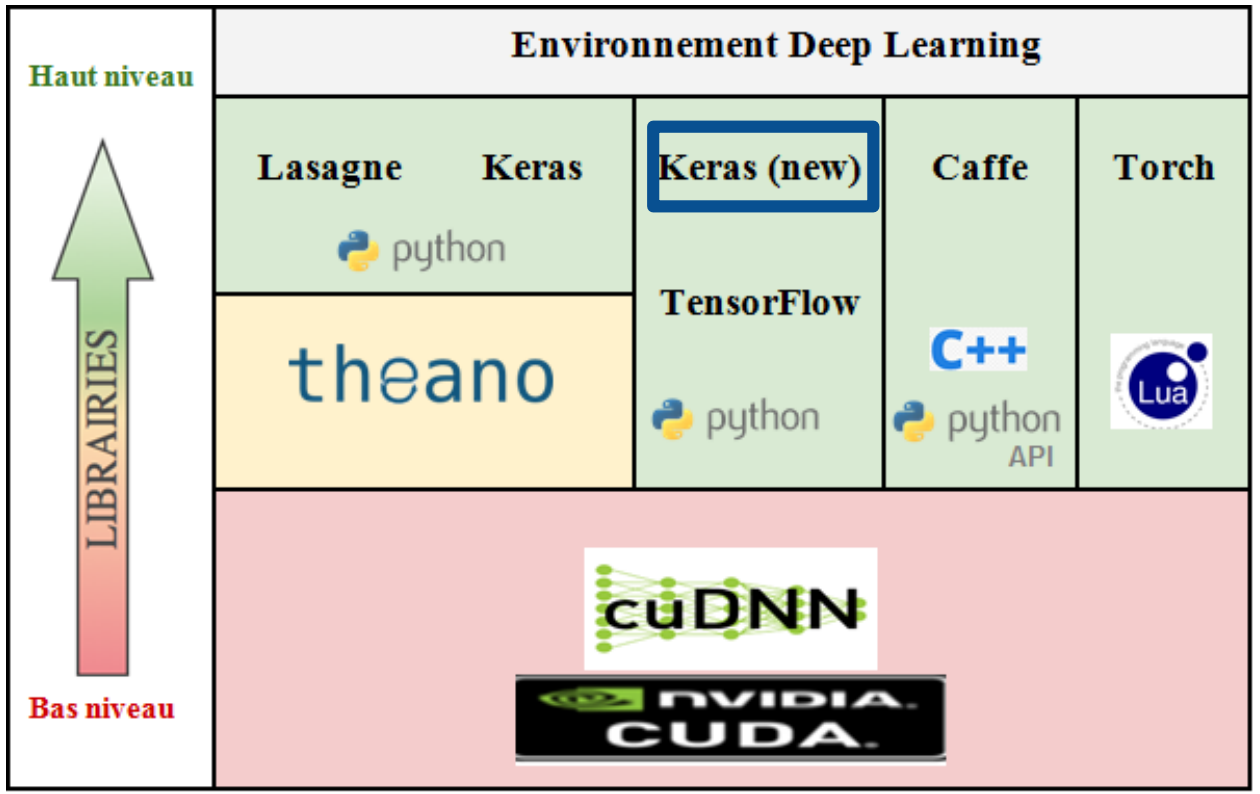
Java : https://deeplearning4j.org/
Possibilité de sauvegarder des exécutables contenant les réseaux de neurones

C : https://github.com/codeplea/genann

Et sûrement bien d'autres !

Quelques librairies

Pour Python et C++ :



Matlab : toolbox « deep learning »

Java : https://deeplearning4j.org/
Possibilité de sauvegarder des exécutables contenant les réseaux de neurones

C : https://github.com/codeplea/genann

Et sûrement bien d'autres !

https://blog.octo.com/classification-dimages-les-reseaux-de-neurones-convolutifs-en-toute-simplicite/

Pour Keras (et sûrement les autres)

- Bien documenté : https://keras.io/
- Beaucoup d'aide disponible (https://stackoverflow.com/)
- Très simple à mettre en place… quand on connaît ce que font les différentes fonctions
- Permet l'utilisation de plusieurs CPU ou GPU
  - Parallélisation des calculs : accélère grandement la vitesse d'exécution
  - Avec Keras, parallélisation automatique et transparente

Un peu de vocabulaire

- $\theta = (W, b)$: les poids et biais à apprendre (ou non, certains peuvent être fixes) sont appelés les **paramètres**

- Tout le reste qui est fixé par l'utilisateur est appelé les **hyperparamètres** :
  - Le nombre de couches
  - Le nombre de neurones par couche
  - Le type de couche
  - Les fonctions d'activation
  - Le taux d'apprentissage ($\theta \coloneqq \theta - \boldsymbol{\lambda}\nabla \mathrm{L}(\theta)$)
  - …

Comment construire mon réseau ?

- Première possibilité : s'inspirer de la littérature
  - Reprendre des architectures construites pour des problèmes similaires : reconnaissance d'image, de signaux audio, de spectres…
  - Adapter l'architecture à son problème : format des données, modifier le réseau suite aux premiers essais d'apprentissage puis aux tests

- Deuxième possibilité : feuille blanche, construction de son propre réseau
  - Commencer par une architecture simple (une ou deux couches fully-connected, quelques neurones)
  - Complexifier peu à peu : ajouter des couches et des neurones, privilégier la profondeur (ajout de couches), utiliser des architectures plus « avancées » (convolution…)
  - La taille est aussi guidée par la vitesse d'exécution, les performances de l'ordinateur… Chaque itération peut être très longue à s'exécuter.

L'heure des premiers choix

Les fonctions d'activation

- En output : en fonction du problème à résoudre
  - Sigmoïde si classification non exclusive : 0 ou 1 avec plusieurs classes pouvant contenir 1
  - Softmax si classification exclusive
  - ReLU si nombre positif
  - Éventuellement None ! Si on veut un nombre réel → Typiquement pour la régression
- Pour les couches intermédiaires :
  - Conseil : privilégier ReLU, apprentissage plus rapide (calculs plus simples), moins de problème d'évanescence ou d'explosion du gradient

L'heure des premiers choix

Usage of activations

Activations can either be used through an `Activation` layer, or through the `activation` argument supported by all forward layers:

```
from keras.layers import Activation, Dense

model.add(Dense(64))
model.add(Activation('tanh'))
```

This is equivalent to:

```
model.add(Dense(64, activation='tanh'))
```

You can also pass an element-wise TensorFlow/Theano/CNTK function as an activation:

```
from keras import backend as K

model.add(Dense(64, activation=K.tanh))
```

Implémentation séquentielle : on ajoute des couches (très courant, plus simple, mais on peut faire autrement aussi)

Nombre de neurones dans la couche

L'heure des premiers choix

L'initialisation des paramètres

- Ne pas oublier d'initialiser les poids !!! Sinon, chaque neurone d'une même couche va propager la même erreur !
- Initialisation aléatoire :
  - Tirée suivant une loi uniforme (choix des bornes de l'intervalle)
  - Tirée dans une gaussienne (choix de la moyenne et écart-type)
  - D'autres initialisations plus sophistiquées : he_normal : *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,* He et al., 2015
- Il n'est pas necessaire en général d'initialiser les biais aléatoirement : les poids suffisent pour donner des comportements différents aux neurones

L'heure des premiers choix

Usage of initializers

Initializations define the way to set the initial random weights of Keras layers.

The keyword arguments used for passing initializers to layers will depend on the layer. Usually it is simply `kernel_initializer` and `bias_initializer`:

```
model.add(Dense(64,
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))
```

On peut mettre son propre initializer

TruncatedNormal

```
keras.initializers.TruncatedNormal(mean=0.0, stddev=0.05, seed=None)
```

Initializer that generates a truncated normal distribution.

These values are similar to values from a `RandomNormal` except that values more than two standard deviations from the mean are discarded and re-drawn. This is the recommended initializer for neural network weights and filters.

Arguments

mean: a python scalar or a scalar tensor. Mean of the random values to generate. stddev: a python scalar or a scalar tensor. Standard deviation of the random values to generate. seed: A Python integer. Used to seed the random generator.

Fonction de coût et optimisation

La fonction de coût

- C'est la fonction que l'on va chercher à optimiser :

$$\theta_{\text{optimaux}} = \underset{\theta}{\operatorname{argmin}} \left(f\left(\hat{Y}(\theta), Y\right) \right)$$

- Beaucoup de possibilités suivant le problème à traiter :
  - Classification exclusive ou non
  - Régression
  - Utilisation de la distance euclidienne, distance en norme 1…

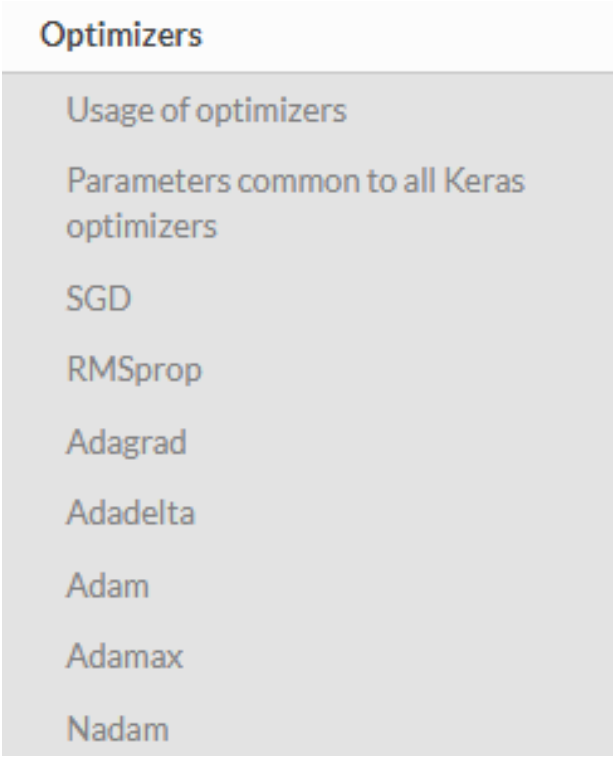
Losses

- Usage of loss functions
- Available loss functions
- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- squared_hinge
- hinge
- categorical_hinge
- logcosh
- categorical_crossentropy
- sparse_categorical_crossentropy
- binary_crossentropy
- kullback_leibler_divergence
- poisson
- cosine_proximity

Fonction de coût et optimisation

Différents algorithmes d'optimisation

- Basés principalement sur la descente de gradient : $\theta \coloneqq \theta - \lambda \nabla \mathrm{L}(\theta)$
- Le plus classique : Stochastic Gradient Descent (SGD)
- D'autres utilisent les moments : ADAM…
  - Conserve en mémoire les gradients précédemment calculés
  - Prend en compte la moyenne de ces gradients dans la mise à jour du nouveau gradient, avec une importance exponentiellement décroissante
  - *Adam: A Method for Stochastic Optimization,* Kingma & Ba, 2014



Petite digression : Algorithme ADAM

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Adam: A Method for Stochastic Optimization, Kingma & Ba, 2014

Choix de l'optimizer

https://www.dlology.com/blog/quick-notes-on-how-to-choose-optimizer-in-keras/

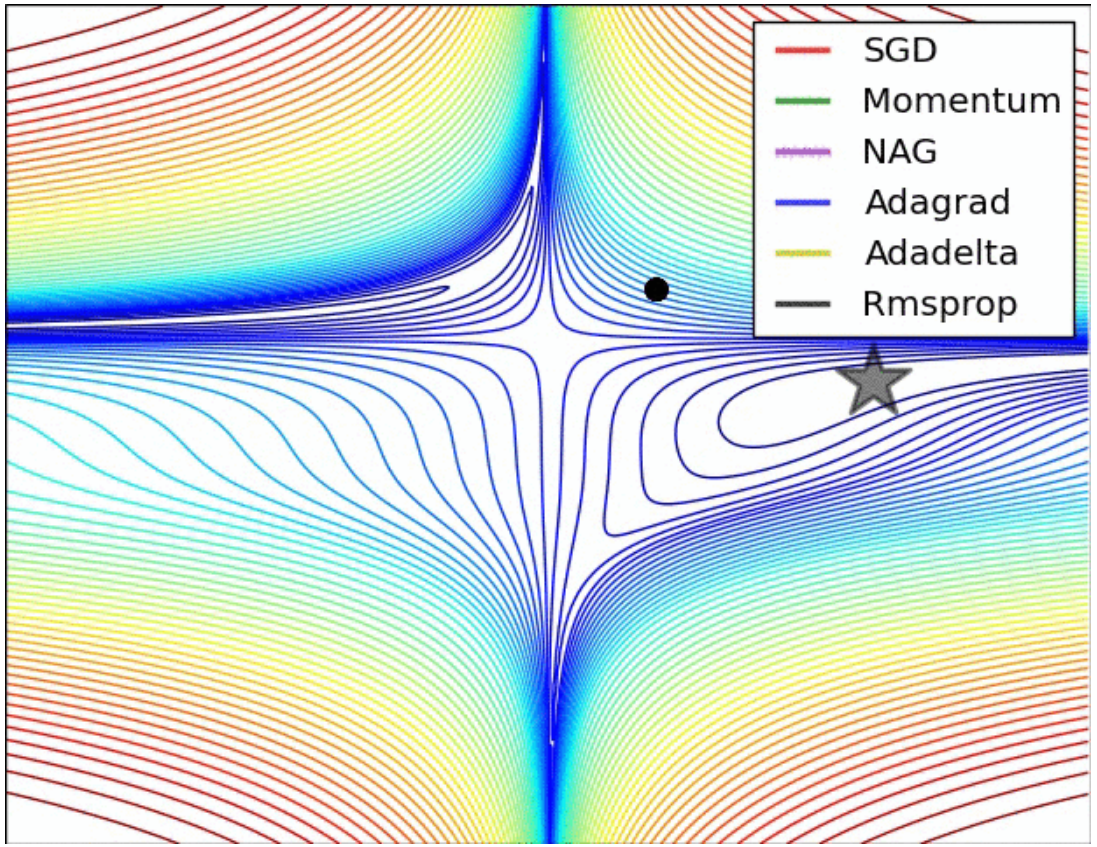


Image Credit: CS231n

Fonction de coût et optimisation

Usage of optimizers

An optimizer is one of the two arguments required for compiling a Keras model:

```
from keras import optimizers

model = Sequential()
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(Activation('softmax'))

sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

Préciser le format des inputs pour la première couche

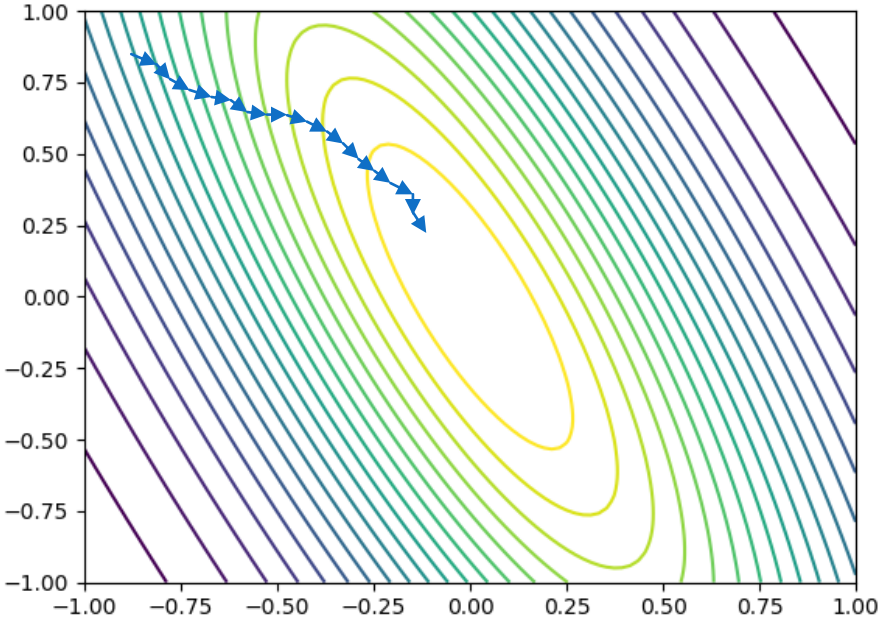
Divers paramètres à préciser

You can either instantiate an optimizer before passing it to `model.compile()`, as in the above example, or you can call it by its name. In the latter case, the default parameters for the optimizer will be used.

```
# pass optimizer by name: default parameters will be used
model.compile(loss='mean_squared_error', optimizer='sgd')
```

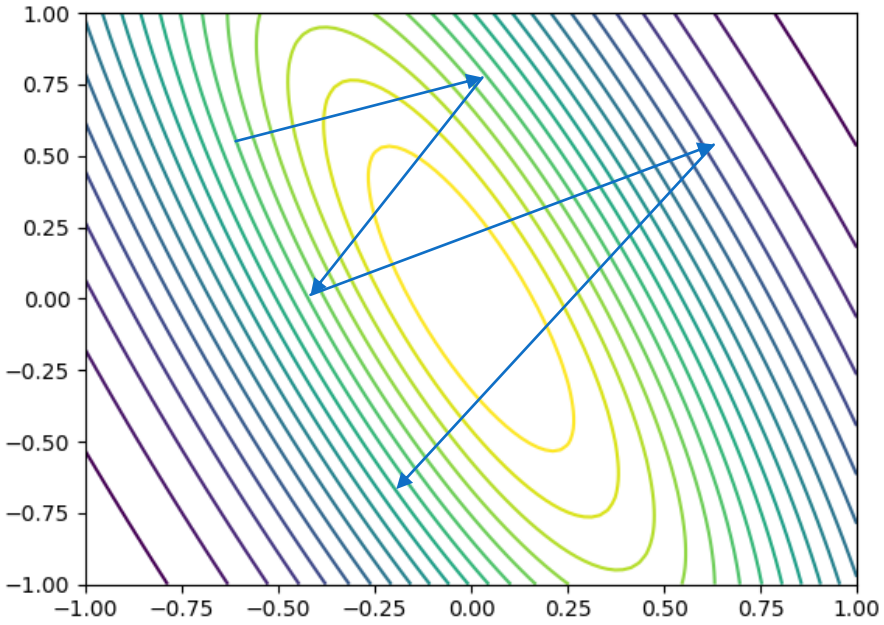
Choix du *learning rate* λ

- En général, valeur fixée au début par l'utilisateur : les valeurs par défaut de Keras sont un bon début
- Possibilité d'utiliser un « *decay* » : à chaque itération, λ devient un peu plus petit
- Certains *optimizers* adaptent le *learning rate* : Adagrad (*Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,* Duchi, Hazan, Singer, 2011)



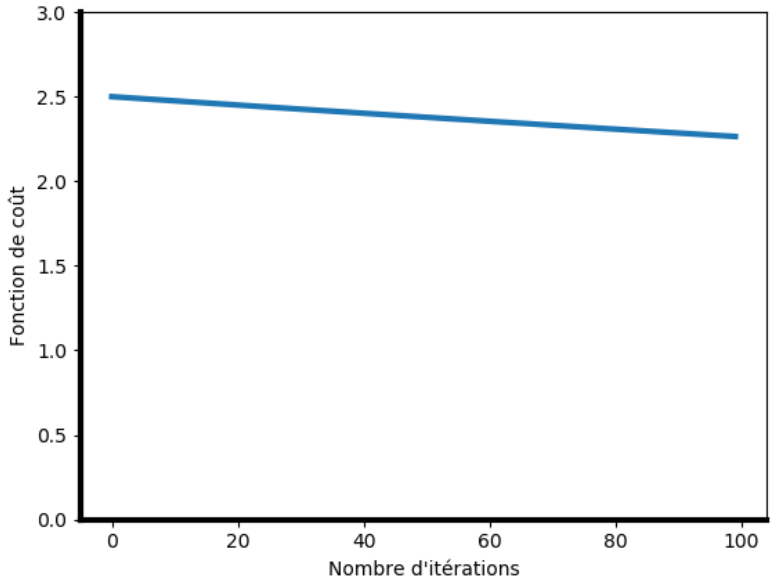
λ trop petit

λ trop grand



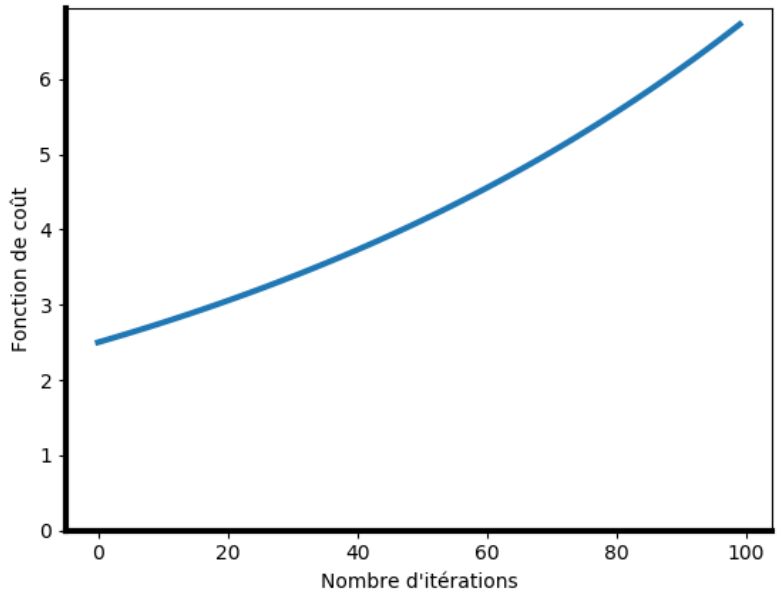
Choix du *learning rate* λ

- Surveiller l'évolution de la **fonction de coût évaluée sur la base d'apprentissage** !
- Sur Keras, directement depuis la console Python : affichage à chaque itération de la valeur de la fonction de coût



λ trop petit

λ trop grand

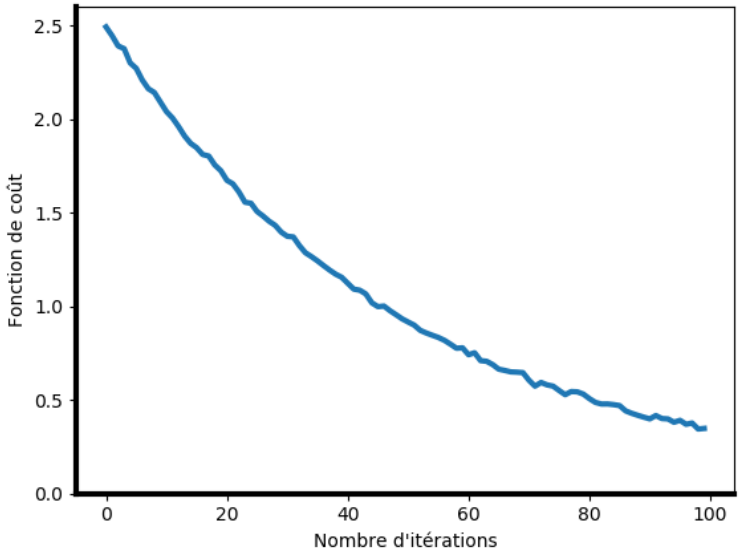


Utilisation des mini-batchs

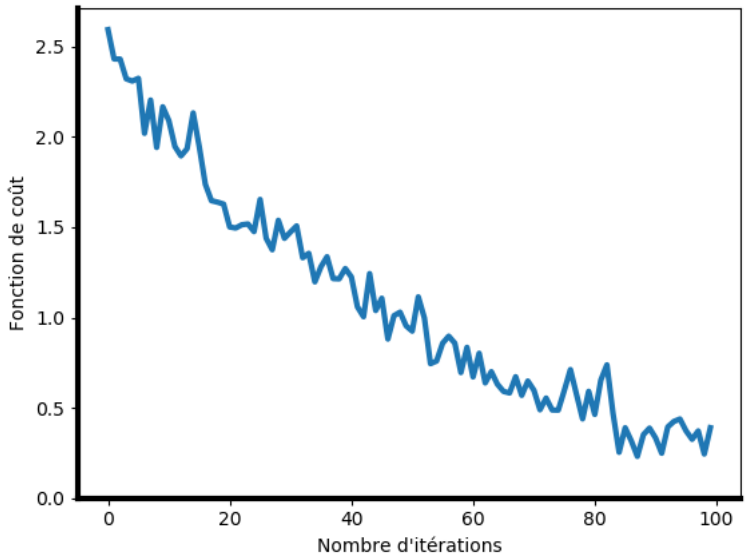
- Le calcul du gradient peut se faire :
  - Sur toute la base de donnée : meilleure convergence, mais chaque itération peut être beaucoup trop longue (**Batch Gradient Descent**)
  - Un exemple par itération : convergence très lente mais chaque itération est très rapide à calculer (**Stochastic Gradient Descent :** attention, c'est différent du SGD de Keras qui peut fonctionner par mini-batchs)
  - Par mini-batchs : sur une partie des exemples (mélangés aléatoirement), permet de tirer au mieux les capacités de parallélisation (**Mini-Batch Gradient Descent**)

- De manière générale, les mini-batchs sont privilégiés : tailles de 32, 64, 128, 256 éléments.

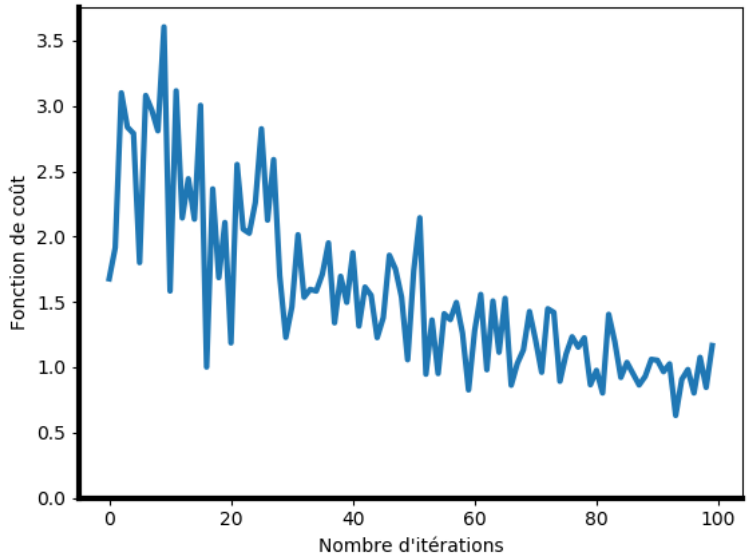
Utilisation des mini-batchs



Batch Gradient Descent

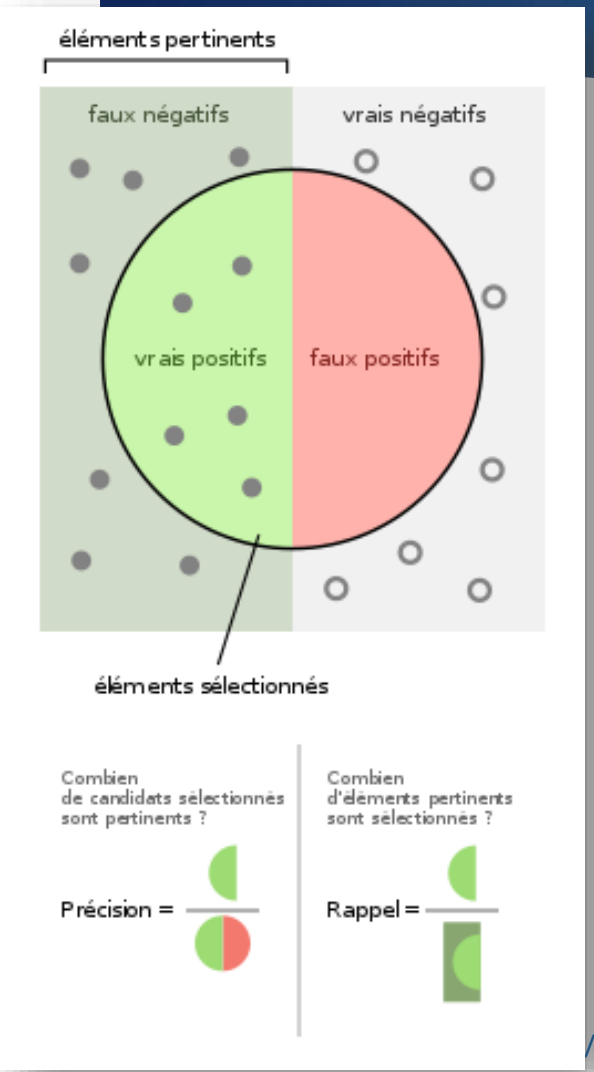


Mini-batch gradient descent



Stochastic gradient descent

Évaluation des performances



- Régression : en général à partir de la fonction de coût directement (moyenne des erreurs en norme 2 ou en norme 1)
- Classification : utilisation d'autres métriques
  - Notations : TP (True Positive), FP (False Positive), TN (True Negative), FN (False Negative)

Attention au seuil à partir duquel on considère une identification de la classe

$\text{Precision} = \frac{\text{TP}}{\text{TP+FP}}$ → caractérise l'erreur due aux faux positifs

$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}$ → caractérise l'erreur due aux faux négatifs

$F_1\text{score} = \frac{2\times\text{precision}\times\text{recall}}{\text{precision+recall}}$ → moyenne harmonique des deux précédents

$\textbf{Accuracy} = \frac{\text{TP+TN}}{\text{TP+FP+TN+FN}}$ → **taux de bonnes réponses**

Metrics

- Usage of metrics
- Arguments
- Returns
- Available metrics
- binary_accuracy
- categorical_accuracy
- sparse_categorical_accuracy
- top_k_categorical_accuracy
- sparse_top_k_categorical_accuracy
- Custom metrics

Résumé des points importants

- Utilisation des librairies pour construire son réseau de neurones
- Bien prendre soin de regarder l'évolution de la fonction de coût au court de l'apprentissage
- Beaucoup d'hyper-paramètres à fixer soi-même : procéder méthodiquement
  - Partir d'un réseau de neurones simple puis complexifier
  - Bien adapter le taux d'apprentissage
  - Penser à utiliser les mini-batchs