NEURONES ARTIFICIELS

Par Kévin Bouchard Ph.D.
Professeur titulaire en intelligence artificielle et apprentissage automatique
Laboratoire d'Intelligence Ambiante pour la reconnaissance d'activités (LIARA)
Directeur de l'Espace innovation en technologies numériques Hydro-Québec
Président du Regroupement québécois des maladies orphelines (RQMO)
Université du Québec à Chicoutimi
www.Kevin-Bouchard.ca Kevin_Bouchard@uqac.ca

CONTENU DE LA LEÇON #2

Vous apprendrez:

- Comment fonctionne un seul neurone artificiel (ou réviserez)
- Comment on entraîne un Adaline et le passage vers l'optimisation
- Nous essaierons également de définir l'apprentissage profond comme un sous-domaine de l'intelligence artificielle

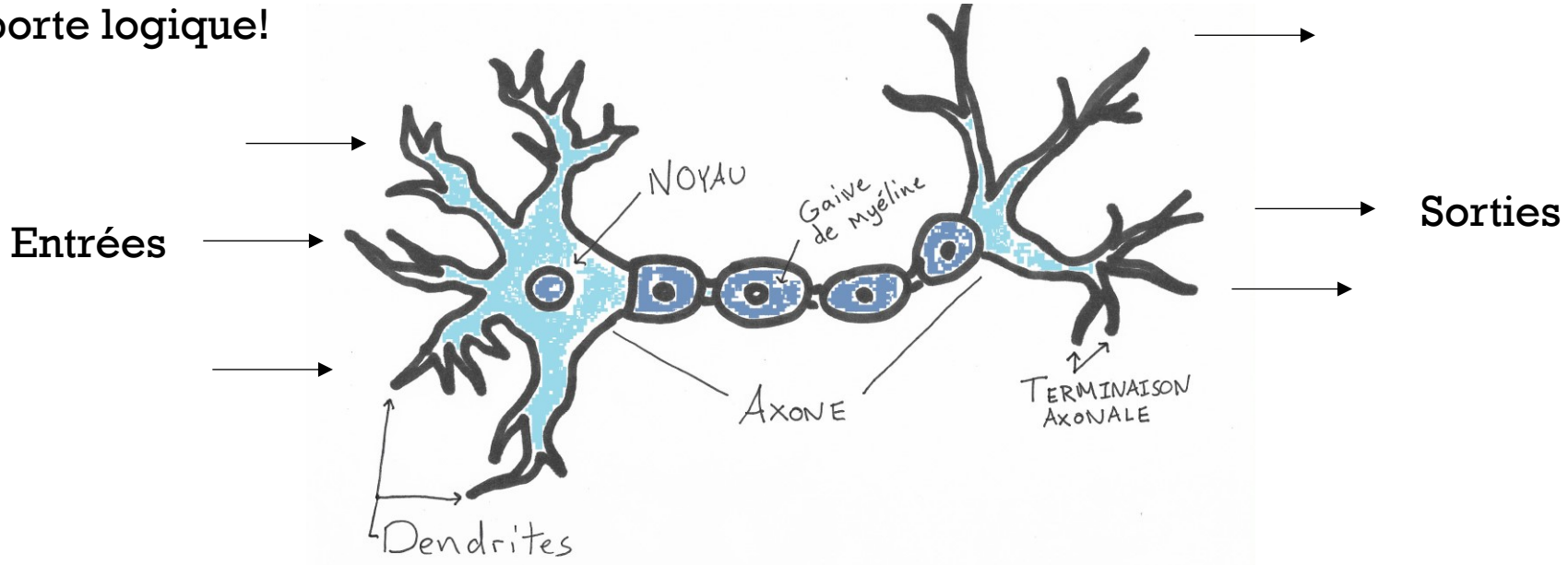
Contenu spécifique:

- Retour sur le Perceptron
- Exercices
- Adaline et le gradient
- Retour sur Logistic Regression
- Exemples de code

UN BREF HISTORIQUE DES PERCEPTRONS

- Retour en 1943: McCullock & Pitts publient le *MCP Neuron*
  - *A logical Calculus of the Ideas Immanent in Nervous Activity*

- Cellules nerveuses interconnectées
  - Transmettent des signaux électriques et chimiques
  - Simple porte logique!



PERCEPTRON

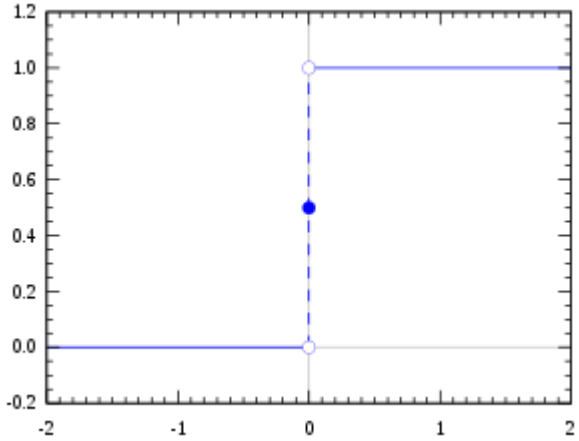
- Rosenblatt 1957: Apprendre les poids optimaux à multiplier avec les entrées afin de déterminer si le neurone s'active ou non
  - Utile pour la classification binaire (ML supervisé)
    - 1 positif
    - -1 négatif
- z est l'entrée nette composé d'une combinaison linéaire d'entrées x et de poids w (somme pondérée)

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \qquad z = w_1 x_1 + \cdots + w_m x_m$$

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

- La classification se définie par une fonction d'activation $\phi(z)$ avec un threshold θ

PERCEPTRON



- x représente l'entrée
  - Grosso modo, l'instance avec ses m features
  - Il peu s'agir d'une instance d'apprentissage ou d'une instance à classer
- La classe est déterminée en function de ce qu'on appelle une fonciton ***d'activation***
- La fonction d'activation du Perceptron s'appelle *Heaviside* ou encore function par palier (step-wise function)
- Elle est représentée par $\phi()$ et prend z en entrée avec un seuil θ

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

$$z = w_1 x_1 + \cdots + w_m x_m$$

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

PERCEPTRON

- L'équation peut être écrite plus simplement apprendre le seuil θ comme un paramètre supplémentaire
- Pour ce faire, nous définissons un paramètre $w_0 = \theta$ avec une caractéristique fictive $x_0 = 1$

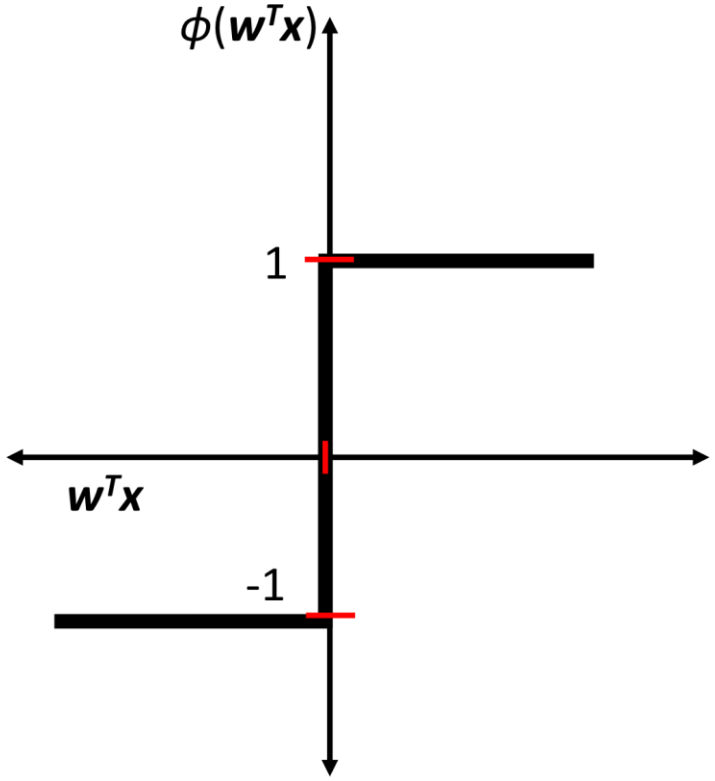
$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \mathbf{w^T x} = \sum_{j=0}^{m} \mathbf{w_j x_j} = \mathbf{w}^T \mathbf{x}.$$

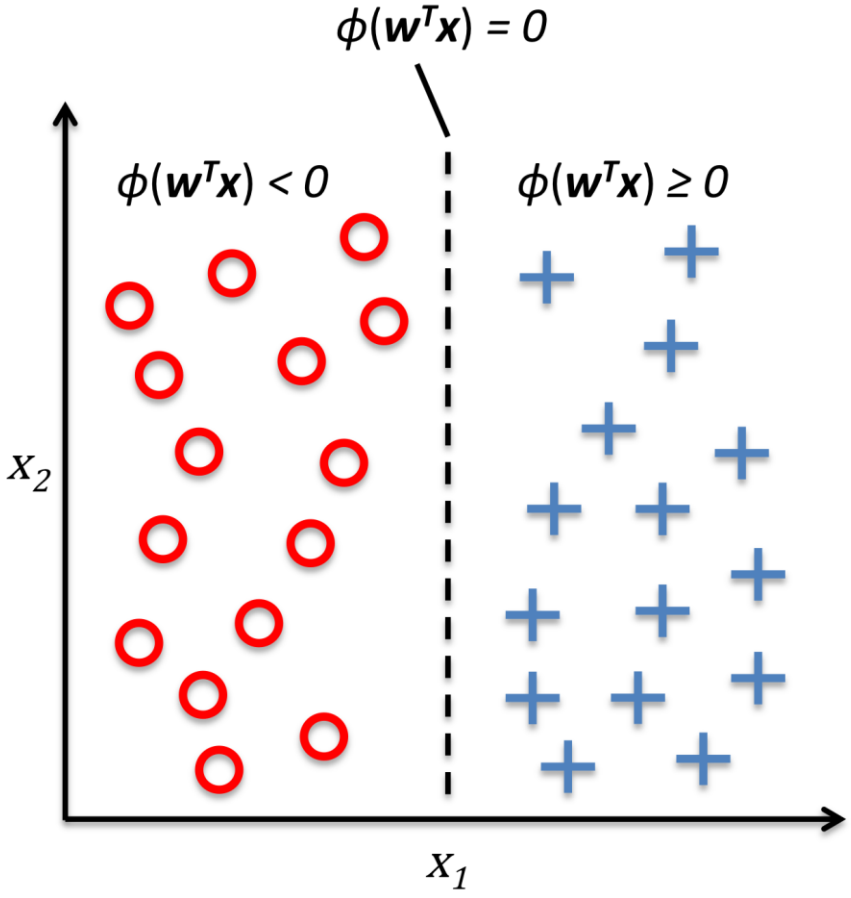
  - T est ajouté dans la forme vectorielle pour signifier transposé
  - E.g.: $z =$

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32.$$

  - Note: w_0 est ce qu'on appelle le *biais*

PERCEPTRON





FONCTIONNEMENT DE L'APPRENTISSAGE

1. Initialisation des poids à 0 (ou un petit nombre aléatoire)
2. Tant qu'il y a des mauvaises classifications:
    1. Pour chaque exemple d'entraînement $x^{(i)}$:
        1. Classer $x^{(i)}$ avec les modèles courant pour obtenir la sortie estimée $\hat{y}^{(i)}$
        2. Mettre à jour les poids

- La mise à jour des poids $w_j \epsilon w$ est $w_j = w_j + \Delta w_j$
- Δw_j calculé selon la règle d'apprentissage du perceptron:

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

- η est le fameux « *Learning rate* » dans l'intervalle]0.0,1.0]

FONCTIONNEMENT DE L'APPRENTISSAGE

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$

- Exemples:
  - $\Delta w_1 = 1(1-1)*1 = 0$
  - $\Delta w_1 = 1(-1--1)*1 = 0$
  - $\Delta w_1 = 1(1--1)*4 = 8$
  - $\Delta w_1 = 1(-1-1)*0.5 = -1$
  - $\Delta w_1 = 0.1(1--1)*4 = 0.8$

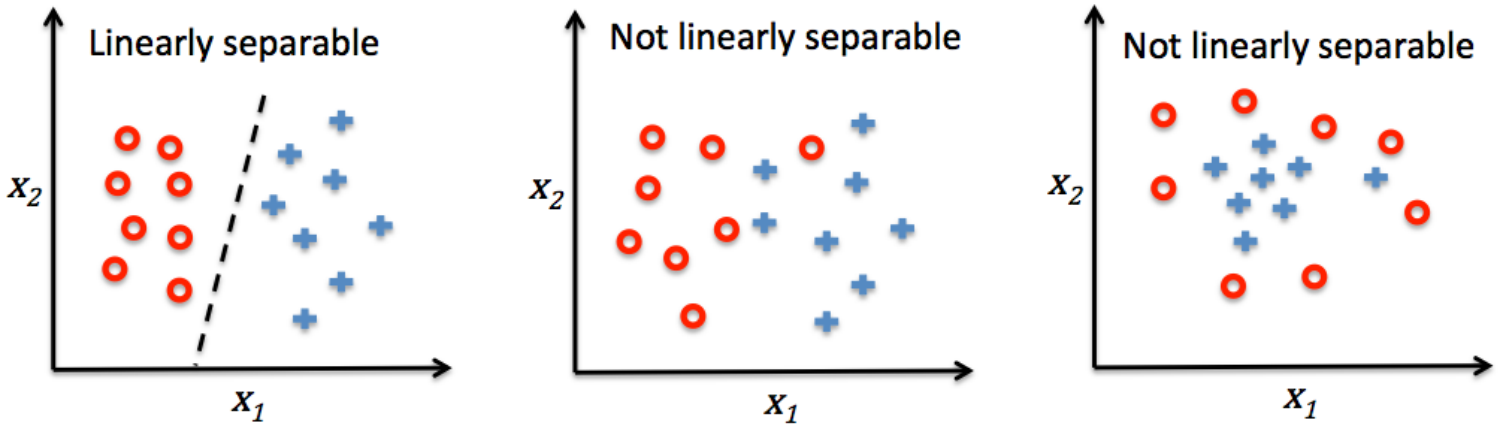
Prédiction positive correcte!

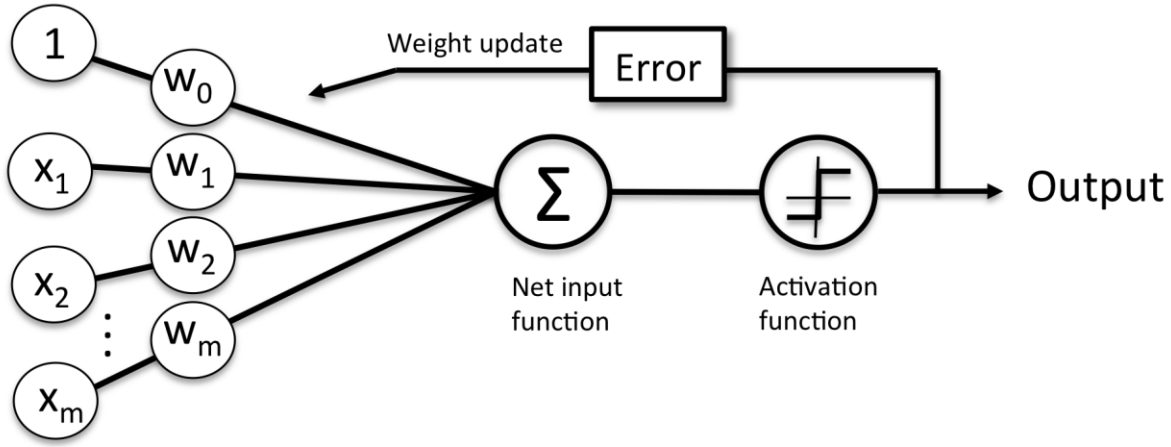
Prédiction négative correcte!

- Le poids ne changera pas si la prédiction est correcte!
- Il varie autrement en fonction de la valeur de $x_j^{(i)}$ dans le vecteur d'entraînement *i*

BILAN

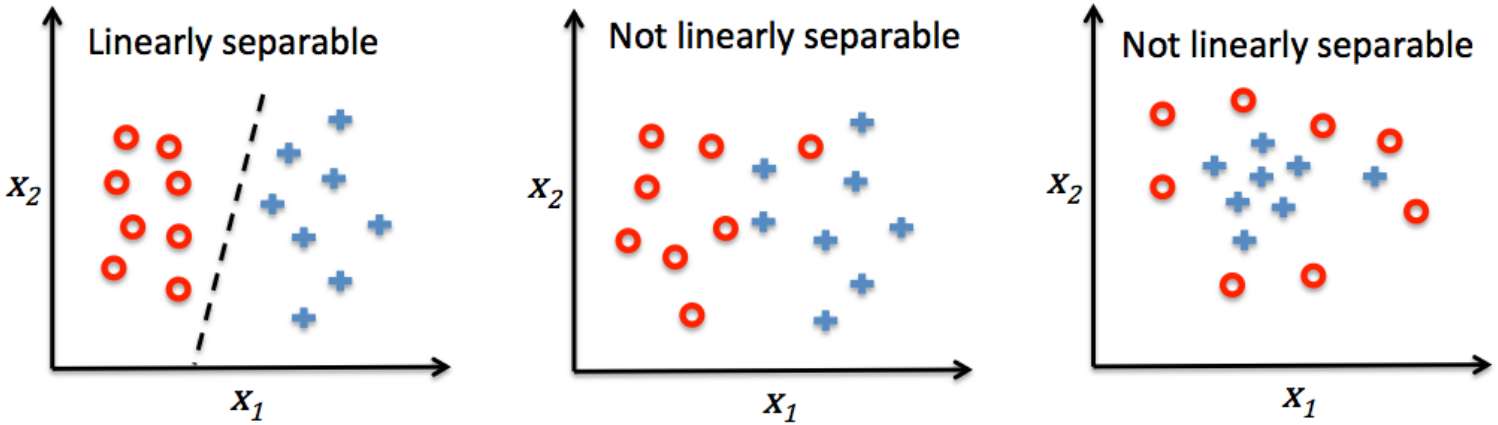
- Si pas séparable linéairement, Perceptron MàJ à l'infini
  - Nombre max de passes à travers l'ensemble (*epochs*)

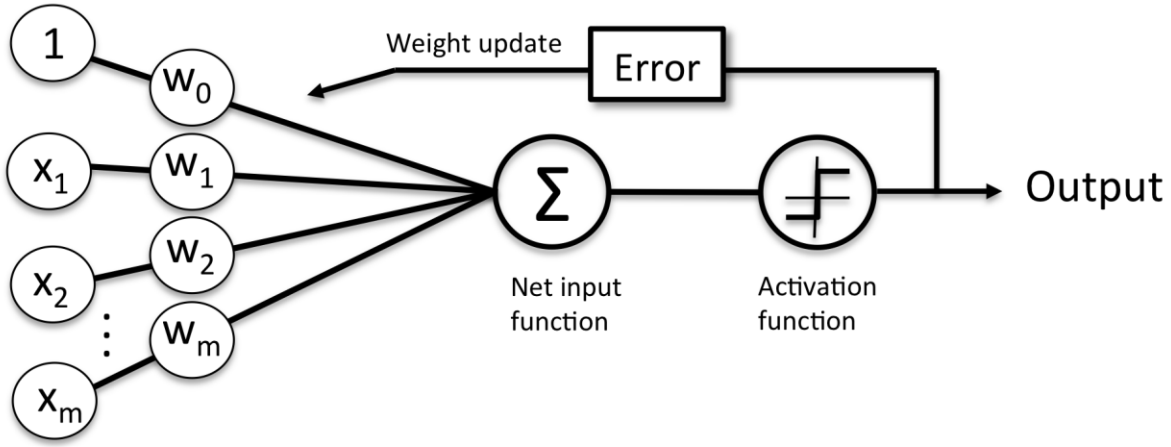




BILAN

- Dans l'ensemble, le perceptron original met en place les principaux éléments des réseaux de neurones
- Le perceptron est un modèle linéaire
- Si pas séparable linéairement, Perceptron MàJ à l'infini
  - Nombre max de passes à travers l'ensemble (*epochs*)





EXERCICES

- Faisons des entraînements de Perceptron ensemble

- Supposons un Perceptron entraîné sur l'ensemble des Iris
  - Poids : [-0.311, -3.091, 3.443, -3.292]
  - Biais : -1.0

- Tentez le calcul avec les instances suivantes (et votre ordinateur!!!)
  - Instance Versicolor :$[0.311, -0.592, 0.535, 0.001]$
  - Instance Virginica: $[-0.174, 1.71, -1.17, -1.184]$
- $z = -1 + 0.311 * -0.311 - 3.091 * -0.592 + 3.443 * 0.535 + 0.001 * -3.292 = 2.572$
- $z = -1 - 0.311 * -0.174 - 3.091 * 1.71 + 3.443 * -1.17 - 3.292 * -1.184 = -6.362$

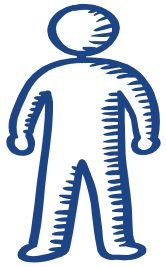
EXEMPLE AVEC LES IRIS

- À partir du code, voici un moment spécifique dans l'exécution

```
Modèle actuel:  [ 0.   -0.24  0.14 -0.7  -0.3 ]
Fleur:  [4.6 3.1 1.5 0.2]  Type:  Iris-setosa
Z= 0.0 *1+ [-0.24  0.14 -0.7  -0.3 ] * [4.6 3.1 1.5 0.2]  =  -1.7800000000000005
Update =  0.1  * ( 1  -  -1 ) =  0.2
Poids MàJ:  [-0.24  0.14 -0.7  -0.3 ]  +  [4.6 3.1 1.5 0.2]  *  0.2  =  [ 0.68  0.76 -0.4  -0.26]
```

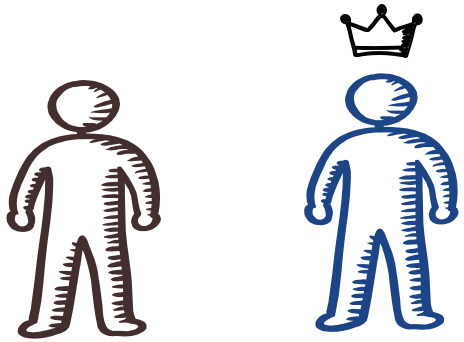
- On voit le calcul de l'entrée nette (en prenant le biais)
- Le calcul de la mise à jour
- La mise à jour elle-même

ONE-VS-ALL



- Les algorithmes de classification binaires tel que le perceptron peuvent être étendu aux problèmes multi classes par diverses stratégies
  - Le ***One-vs-All*** consiste à créer un classeur par classe où toute autre instance est considérée de classe négative
  - La classification consiste ensuite à passer un nouvel exemple dans tous les classeurs de façon à trouver celui qui se déclenche
  - Attention! En général, on préfère plutôt avoir une sortie en termes de niveau de confiance où l'on cherche le max
    - Difficile avec des données mal balancées
    - Difficile de s'assurer que la gamme de niveaux de confiance ne varie pas trop d'un classeur à l'autre

ONE-VS-ONE



- Il existe une autre stratégie populaire pour d'autres types de classeurs qui ne sont pas multiclasses par défaut
- Le One-vs-One est une des stratégies populaires utilisées dans Scikit-Learn
  - Un classeur par pair de classe
  - Vote sur la totalité des classeurs
- Le vote se fait sur l'entièreté des $T_c = \frac{c(c-1)}{2}$ classeurs
  - E.g.: pour 10 classes→$10(10-1)/2 = 45$ classeurs
- Scikit-Learn explique de long en large les stratégies implémentées selon l'algorithme:
  - https://scikit-learn.org/stable/modules/multiclass.html

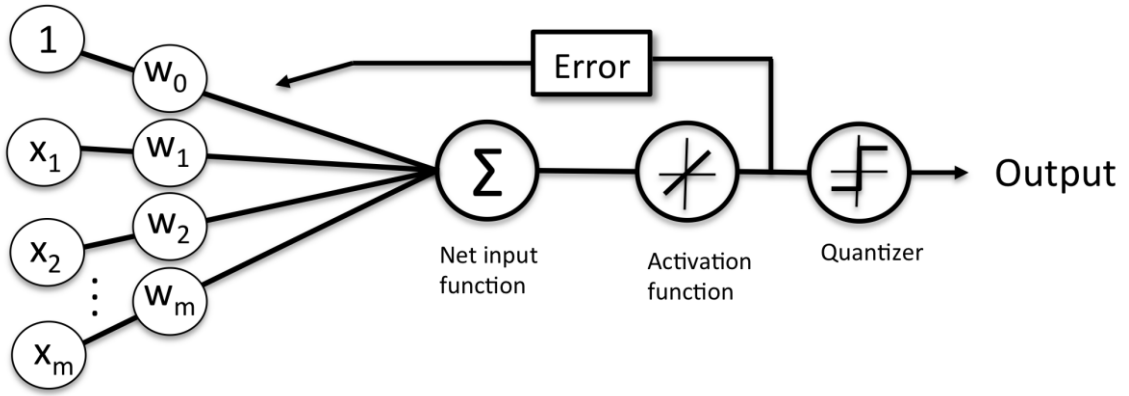
16

ADAPTIVE LINEAR NEURONS

Adaline: Un autre type de NN simple couche

-MàJ des poids selon une fonction d'activation linéaire!

ADALINE



- La fonction d'activation linéaire $\phi(z)$ est simplement la fonction d'identité de l'entrée nette $\phi(w^T x) = w^T x$
- Elle sert à mettre à jour les poids
- Cependant, un élément similaire à la fonction *Heaviside* parfois nommé « *Quantizer* » permet la prédiction de la classe

- Les sorties sont des valeurs **continues**! (plutôt que binaires)

FONCTION DE COÛTS

- Clé en ML: optimisation d'une fonction objective (souvent *cost ou loss function*)
- Ceci n'est pas un cours d'optimisation, mais nous devrons comprendre quelques éléments
- Pour Adaline, fonction de coûts à minimiser
  - Apprendre les poids en tant que ***Sum of Squared Errors*** (SSE) entre les sorties et les vraies classes

$$J(\mathbf{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

*Le ½ est ajouté pour faciliter le calcul du gradient (1/n classes)

ALGORITHME DU GRADIENT

- Puisque $J(w)$ est une fonction convexe, nous pouvons faire l'optimisation grâce à l'algorithme *gradient descent*.
- Regarde la « pente » des états voisins
  - Bouge dans la direction la plus abrupte
  - Trouvée par la dérivée partielle (voir diapo 15)
- Learning rule: $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}$

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$$

Hill-climbing (Rappel)

- **Exploration locale**
- **G(n) (cost) de chaque voisin**
- **On choisit le voisin qui améliore le plus**

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node
   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
   end
```

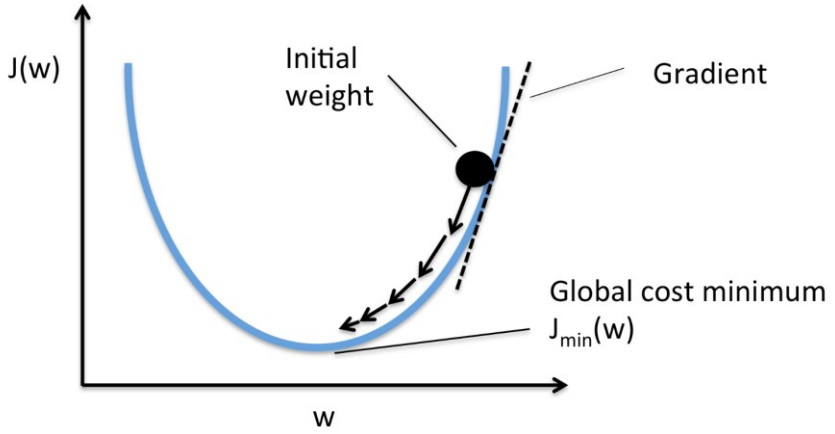
ALGORITHME DU GRADIENT

- La dérivée d'une fonction mesure comment elle change à un point donné
- Elle quantifie le **taux de variation** de la fonction par rapport à une ou plusieurs variables
- La dérivée de $f(x)$ à un point spécifique $x = a$ est directement la pente de la tangente à ce point (taux de variation instantané)

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

- Par exemple, si $f(x) = 3x^2 + 4x$ et $x = 1$, alors pour h vers 0 nous aurons ≈ 2
- E.g. $h = 0.1, f'(1) = 2.3$ mais $h = 0.00001, f'(1) = 2.00003$

POURQUOI EST-CE PERTINENT?



- La différentiation peut nous dire comment varier les paramètres d'une fonction
- Évidemment, nos fonctions sont plus complexes, car elles contiennent $|w|$ paramètres
- Le gradient d'une fonction $f(x_1, x_2, \ldots, x_n)$ est un vecteur qui contient les dérivées partielles de f par rapport à chacune des variables x_i

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

  - Par exemple, pour $f(x, y) = x^2 + y^2$ le gradient serait le vecteur $\nabla f = [2x, 2y]$

- Bref, le gradient de notre fonction de coûts est le vecteur de taille $|w|$!
  - Donc, $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}$

DÉRIVÉE PARTIELLE DE SSE

$$J(\mathbf{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi(z^{(i)})\right)^2$$

- Pour trouver la dérivée partielle $\frac{\partial J}{\partial w_j}$ (comment J change en fonction du poids)
- On applique la règle de la chaîne pour le terme au carré:

$$\frac{\partial J}{\partial w_j}\left(y^{(i)} - \phi(z^{(i)})\right)^2 = 2\left(y^{(i)} - \phi(z^{(i)})\right) * \frac{\partial J}{\partial w_j}\left(y^{(i)} - \phi(z^{(i)})\right)$$

$$\frac{\partial J}{\partial w_j} = \frac{1}{2}\sum_i 2\left(y^{(i)} - \phi(z^{(i)})\right) * \frac{\partial J}{\partial w_j}\left(y^{(i)} - \phi(z^{(i)})\right)$$

- De plus le 2 annule le ½

$$\frac{\partial J}{\partial w_j} = \sum_i \left(y^{(i)} - \phi(z^{(i)})\right) * \frac{\partial J}{\partial w_j}\left(y^{(i)} - \phi(z^{(i)})\right)$$

DÉRIVÉE PARTIELLE DE SSE

$$J(\mathbf{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

- La fonction $\phi(z^{(i)})$ peut être remplacée par la somme pour toutes les instances

$$\frac{\partial J}{\partial w_j} = \sum_i \left(y^{(i)} - \phi(z^{(i)})\right)\frac{\partial J}{\partial w_j}\left(y^{(i)} - \sum_i w_j^{(i)} x_j^{(i)}\right)$$

- Comme $y^{(i)}$ est une constante, nous cherchons plutôt $\frac{\partial \phi(z^{(i)})}{\partial w_j}$

$$\frac{\partial \phi(z^{(i)})}{\partial w_j} = \phi'\left(z^{(i)}\right) * \frac{\partial z^{(i)}}{\partial w_j}$$

- Puisque $z^{(i)} = \sum_j w_j^{(i)} x_j^{(i)}$, alors $\frac{\partial \phi(z^{(i)})}{\partial w_j} = x_j^{(i)}$

$$\frac{\partial J}{\partial w_j} = \sum_i \left(y^{(i)} - \phi(z^{(i)})\right)\left(-x_j^{(i)}\right) = -\sum_i \left(y^{(i)} - \phi(z^{(i)})\right)x_j^{(i)}$$

RETOUR AU GRADIENT

- Bref nous avons maintenant: $\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi(z^{(i)})\right) x_j^{(i)}$
- Nous avons dit précédemment que la mise à jour des poids $\Delta w = -\eta \frac{\partial J}{\partial w_j}$
- Donc ultimement, si vous souhaitez implémenter Adaline avec le gradient:

$$\Delta w = \eta \sum_i \left(y^{(i)} - \phi(z^{(i)})\right) x_j^{(i)}$$

- En code c'est encore plus simple:

```
output = self.activation(X)
errors = (y - output)
self.w_[1:] += self.eta * X.T.dot(errors)
self.w_[0] += self.eta * errors.sum()
```

EXPLICATION DU CODE

```
output = self.activation(X)
```

- Calcul l'entrée nette pour chaque instance

```
errors = (y - output)
```

- Deux vecteurs de la taille du dataset
- Donne en retour un vecteur d'erreurs de la même taille où chaque vraie classe se voit soustraire l'entrée
- E.g. y=1, z=0.234 alors 1-0.234

```
self.w_[1:] += self.eta * X.T.dot(errors)
```

- On multiplie le learning rate avec X
- X est un tableau de dimension instances par features
- .T transpose X afin de pouvoir le multiplier par notre vecteur d'erreurs
  - Ça donne un vecteur de taille features, soit 1 élément par w!!!

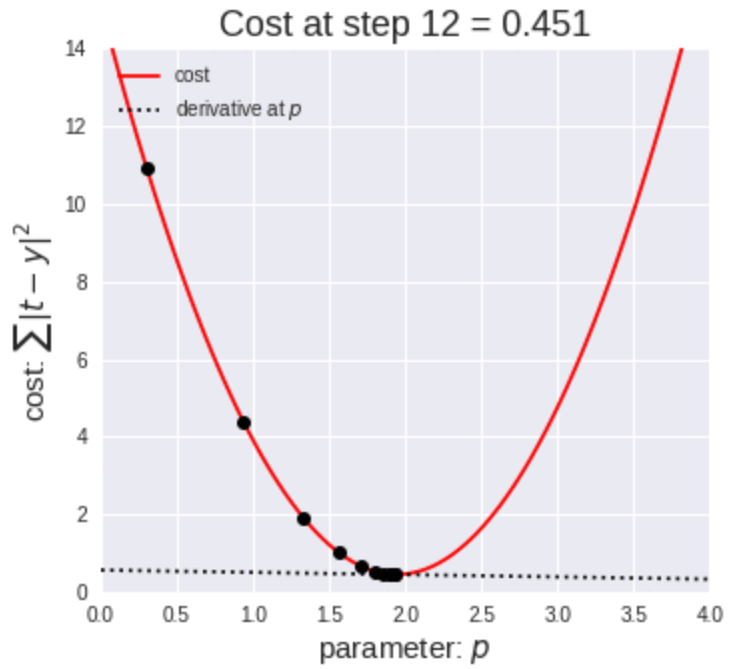
```
self.w_[0] += self.eta * errors.sum()
```

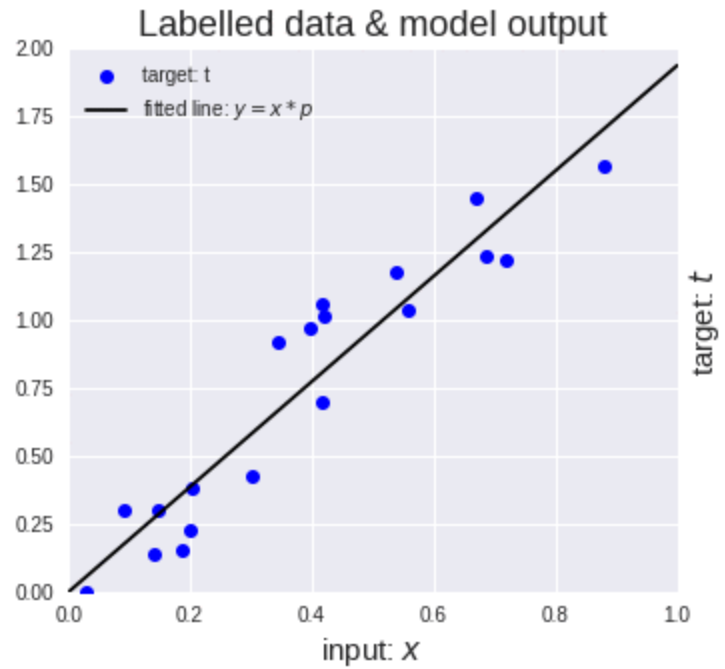
- Fait la somme des erreurs * le learning rate et l'addition au biais

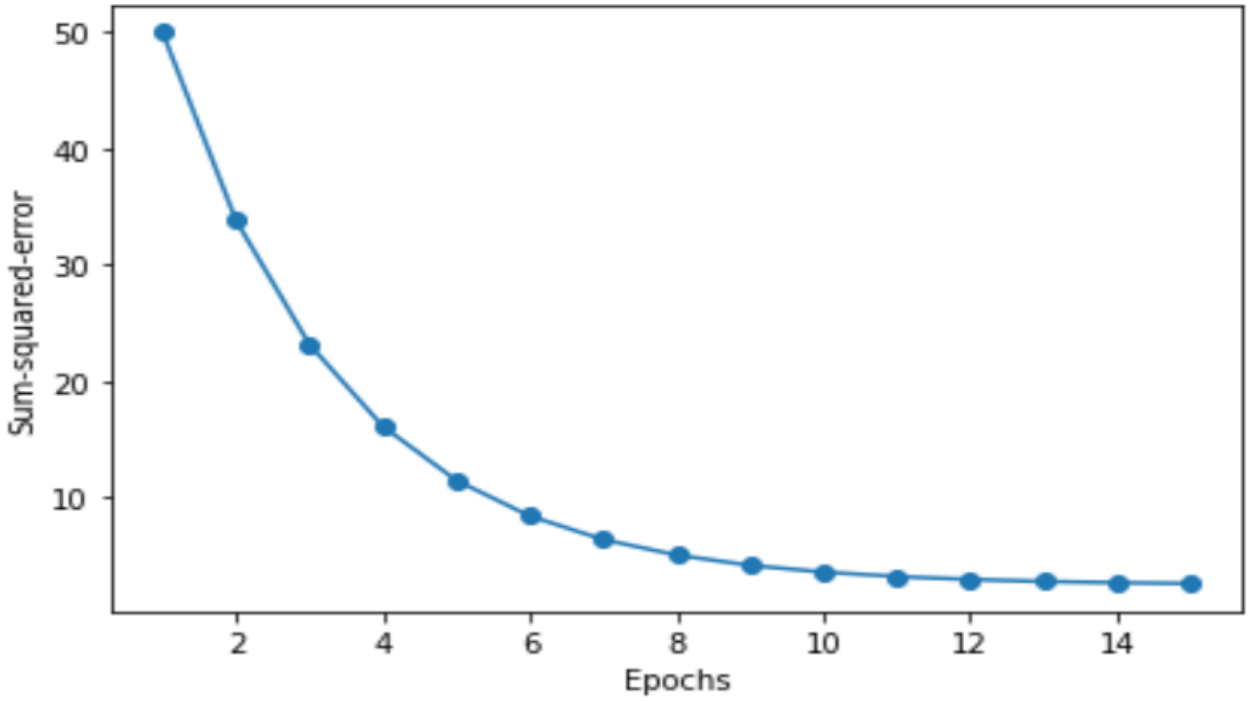
PRODUIT CROISÉ (DOT) - RAPPEL

- Attention! Cette ligne ne donne pas la même chose: `X.T*errors`
  - On multiplie chaque élément du vecteur par les éléments de la matrice et on reste en instances x features!!!
- Tandis que le produit croisé change la dimension
- Si $X \epsilon \mathbb{R}^{n \times f}$ et errors $e \epsilon \mathbb{R}^{n}$ alors $X^T e$:

$$X^T e = \begin{bmatrix} \sum_{i=1}^{n} x_{i,1} e_i \\ \sum_{i=1}^{n} x_{i,2} e_i \\ \dots \\ \sum_{i=1}^{n} x_{i,f} e_i \end{bmatrix}$$







GRADIENT

Haut: Visualisation de l'algorithme du gradient à 1 paramètre

Bas: SSE en fonction des epochs

*Image

towardsdatascience.com

RÉSUMÉ

- Même si la règle d'apprentissage $\phi(z^{(i)})$ ressemble à celle du perceptron, $z^{(i)} = w^T x$ est un nombre réel
  - Dans le perceptron, c'est en entier naturel de classe 1 ou -1

- Enfin, la MàJ des poids se fait sur le dataset en entier!
  - Dans le perceptron, c'est plutôt incrémental (instance par instance)

- L'algorithme du gradient bénéficie du *feature scaling*
  - Le calcul est plus rapide!!!
  - sklearn.preprocessing.scale (standardisation)
  - Sinon avec un tenseur Torch:

```
data = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
means = data.mean(dim=0, keepdim=True)
stds = data.std(dim=0, keepdim=True)
normalized_data = (data - means) / stds
```

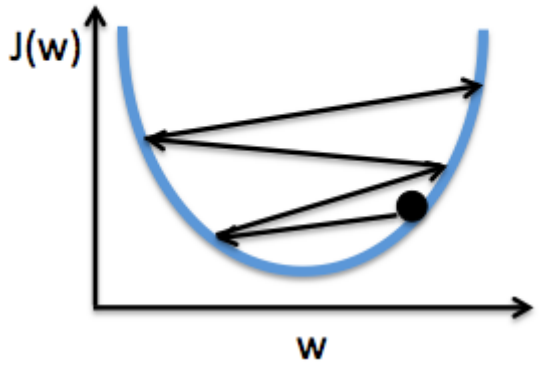
ALLER UN PEU PLUS LOIN…

HYPERPARAMÈTRES

- η, le « Learning rate » et le nombre d'« epochs » sont ce qu'on appelle des hyperparamètres
  - Il n'y a pas de valeurs parfaites
  - Un learning rate trop haut peut empêcher de converger
  - Un learning rate trop faible fera tourner l'algorithme longtemps (besoin de rouler un grand nombre d'epochs)

- Des méthodes existent pour automatiquement calibrer les hyperparamètres

- Nous en rediscuterons plus tard



STOCHASTIC GRADIENT DESCENT

- Nous avons vu comment minimiser une fonction de coûts avec un gradient calculé à partir de l'ensemble **complet**
  - Difficile si nous avons des millions d'instances!
  - Réévaluation de l'ensemble en entier à chaque étape (!!!)

- Plus populaire dans le ML:
  - *Stochastic Gradient Descent* (parfois *iterative* ou *online)*
  - Incrémentation des poids pour chaque échantillon de l'ensemble d'entraînement!

$$\Delta \mathbf{w} = \eta \sum_i \left(y^{(i)} - \phi(z^{(i)}) \right) \mathbf{x}^{(i)}. \quad \Rightarrow \quad \Delta \mathbf{w} = \eta \left(y^{(i)} - \phi(z^{(i)}) \right) \mathbf{x}^{(i)}.$$

STOCHASTIC GRADIENT DESCENT (SUITE)

- C'est une approximation de l'algorithme du gradient
  - Converge plus rapidement en général! (+ de MàJ)
  - L'erreur contient plus de bruit

- IMPORTANT: Les données doivent être présentées en ordre aléatoire pour obtenir de bons résultats
  - Puisqu'on calcule sur chaque échantillon sans remise à 0

- Le learning rate est souvent adaptatif avec SGD
- Voir code Adaline – SGD (fit, partial_fit et shuffle)

LOGISTIC REGRESSION

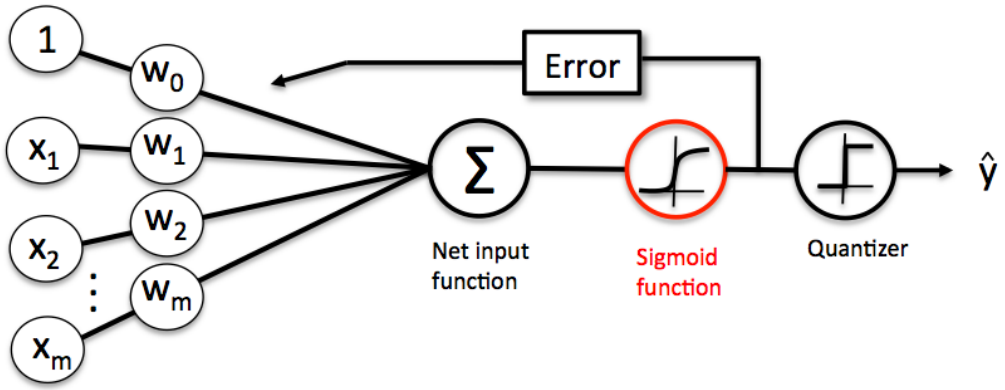
Un autre algorithme pour la classification

-Étonnamment, pas pour la régression!!!!

LOGISTIC REGRESSION

- Similairement au Perceptron et à Adaline
  - Classification binaire
  - Très populaire, peut utiliser la stratégie One vs All

- Modèle probabiliste
- La fonction d'activation est le Sigmoid



ODDS RATIO (RAPPORT DES CHANCES)

- Le rapport des chances en faveur d'un certain événement
  - Le ratio utilise p la probabilité qu'un événement se produise
  - $^{p}/_{(1-p)}$
- La fonction suivante est le logarithme d'un odds ratio

$$logit(p) = \log \frac{p}{(1-p)}$$

- Elle permet de prendre en entrée des valeurs entre 0 et 1 et de les transformer afin d'exprimer une relation linéaire avec les log-odds
  - Ici $p(y = 1|x)$ est la probabilité conditionnelle qu'un échantillon appartienne à la classe 1 étant donné ses attributs x

$$logit(p(y = 1|\mathbf{x})) = w_0 x_0 + w_1 x_1 + \cdots + x_m w_m = \sum_{i=0}^{m} w_i x_i = \mathbf{w}^T \mathbf{x}.$$

FONCTION D'ACTIVATION

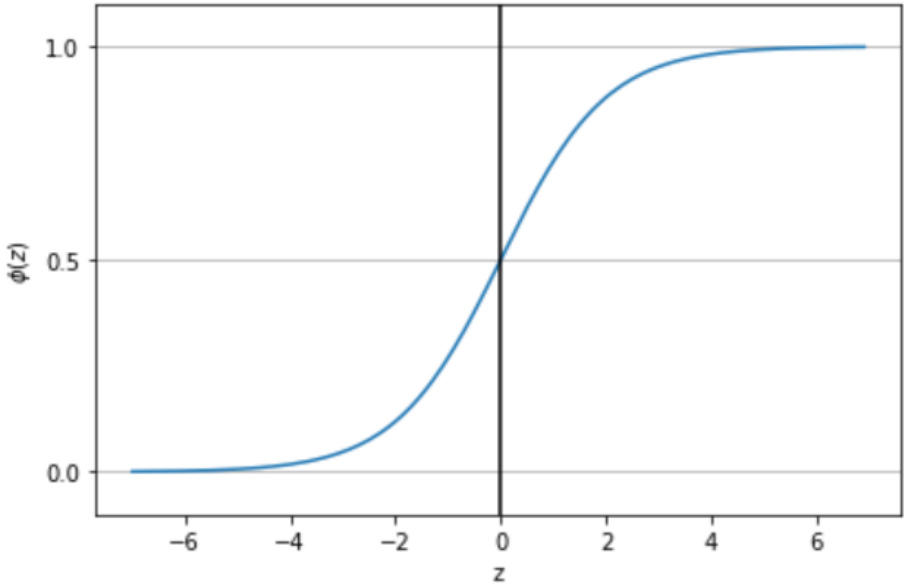
- Nous sommes plutôt intéressés à :
  - Prédire les probabilités qu'un échantillon appartienne à une classe **spécifique**
  - C'est la fonction inverse, la fonction **sigmoïde** (ou la fonction logistique)!

  - z est l'entrée nette (tel que précédemment) → $z = w^T x$

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

INTERPRÉTATION

- La fonction d'activation tend vers
  - 1 lorsque z tend vers ∞
  - 0 lorsque z tend vers $-\infty$
  - Elle vaut exactement 0.5 lorsque $z = 0$



- On interprète la $\phi(z)$ comme la probabilité qu'un échantillon appartienne à une classe spécifique
  - E.g. $\phi(z) = 0.8$ pourrait vouloir dire 80% de chance que l'échantillon soit de type Iris-Versocolor
  - Il resterait dans ce cas 20% d'appartenance à l'autre classe (binaire!)
- On peut utiliser un Quantizer pour une réponse discrète

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0.5 \\ 0 & \text{otherwise .} \end{cases}$$

APPRENTISSAGE DES POIDS

- Sum-Squared-Error (SSE) en tant que fonction de coûts:
  - On a minimisé pour apprendre!

$$J(\mathbf{w}) = \frac{1}{2}\sum_i \left(\phi(z^{(i)}) - y^{(i)}\right)^2.$$

- Fonction de coûts pour Logistic Regression
  - La probabilité à maximiser en supposant que les échantillons de notre dataset sont indépendants:

$$L(\mathbf{w}) = P(\mathbf{y}|\mathbf{x};\mathbf{w}) = \prod_{i=1}^{n} P(y^{(i)}|x^{(i)};\mathbf{w}) = \prod_{i=1}^{n} \left(\phi(z^{(i)})\right)^{y^{(i)}} \left(1-\phi(z^{(i)})\right)^{1-y^{(i)}}$$

  - Plus facile d'utiliser les logarithmes (fonction Log-Likelyhood)

$$l(\mathbf{w}) = \log L(\mathbf{w}) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi(z^{(i)})\right) + \left(1-y^{(i)}\right) \log\left(1-\phi(z^{(i)})\right)\right]$$

APPRENTISSAGE DES POIDS

- Les logs permettent d'éviter un « underflow » (soupassement arithmétique)
  - E.g.: 0.0000001*0.003*…
- De plus, on transforme le produit en somme! (plus facile à dériver)
- On peut donc optimiser de la même façon qu'avec Adaline en dérivant pour trouver le gradient
  - Attention, il faut utiliser gradient ascent avec la fonction
  - Ou inverser celle-ci:

$$J(\mathbf{w}) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi(z^{(i)})\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi(z^{(i)})\right) \right]$$

EXEMPLE

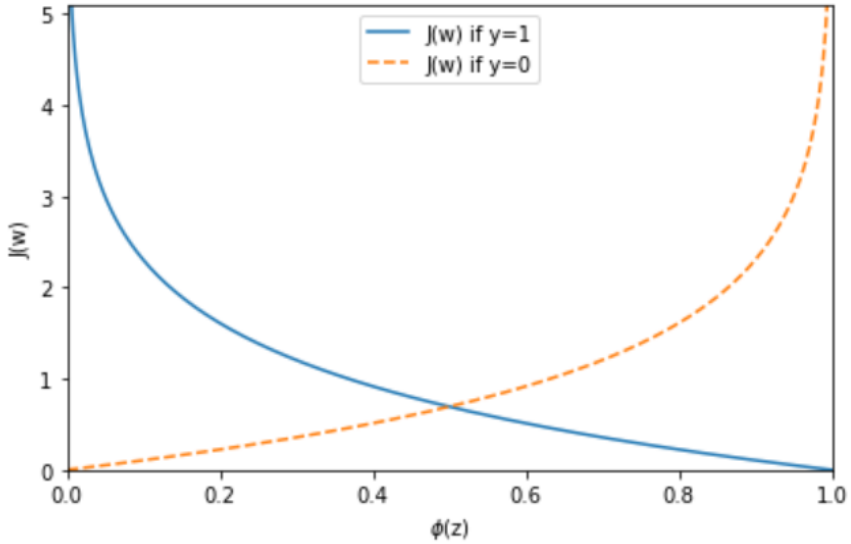
- Supposons la fonction de coûts pour un seul échantillon:

$$J\big(\phi(z), y; \mathbf{w}\big) = -y \log\big(\phi(z)\big) - (1-y) \log\big(1 - \phi(z)\big).$$

- Si l'échantillon est 0 ou s'il est 1, nous avons:

$$J\big(\phi(z), y; \mathbf{w}\big) = \begin{cases} -\log\big(\phi(z)\big) \text{ if } y = 1 \\ -\log\big(1 - \phi(z)\big) \text{ if } y = 0 \end{cases}$$

- Pour différentes valeurs de la fonction d'activation:



DÉRIVÉE DE LOG-LIKELYHOOD

Dérivée de sigmoïde:

$$\frac{\partial}{\partial w_j}\phi(z) = \frac{\partial}{\partial z}\frac{1}{1+e^{-z}} = \frac{1}{(1+e^{-z})^2}e^{-z} = \frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right) = \phi(z)(1-\phi(z))$$

Dérivée de log-likelyhood

$$\frac{\partial}{\partial w_j}l(w) = \left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\frac{\partial}{\partial w_j}\phi(z)$$

$$= \left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\phi(z)(1-\phi(z))\frac{\partial}{\partial w_j}z$$

$$= \Big(y\big(1-\phi(z)\big) - (1-y)\phi(z)\Big)x_j$$

$$= (y-\phi(z))x_j$$

Forme connue!!!

LEARNING RULE

- La mise à jour des poids fonctionne de cette façon (comme précédemment):

$$w_j := w_j + \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)} \qquad \mathbf{w} := \mathbf{w} + \Delta \mathbf{w}$$

- Delta *w* est trouvé grâce au gradient de la fonction J(*w*)

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)}$$

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \; \Delta \mathbf{w} = -\eta \nabla J(\mathbf{w})$$

OPTIMISEURS DES LIBRAIRIES CONNUES

- Nous reviendrons sur l'optimisation dans les prochains cours, mais en attendant voici vos options.
- Dans Scikit-Learn, peu de choix: https://scikit-learn.org/stable/modules/neural_networks_supervised.html
- Dans Tensorflow: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers
  - Adam et RMSProp sont particulièrement populaires
- PyTorch offre encore plus de choix que Keras: https://pytorch.org/docs/stable/optim.html

RÉFÉRENCES ORIGINALES

Perceptron	Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6), 386.
Adaline	Widrow, B. (1960). Adaptive" adaline" Neuron Using Chemical" memistors.".
Logistic Regression	Cox, David R. (1958). "The regression analysis of binary sequences (with discussion)". J R Stat Soc B. 20 (2): 215–242.
Gradient Descent	Cauchy, A. (1847). Méthode générale pour la résolution des systemes d'équations simultanées. Comp. Rend. Sci. Paris, 25(1847), 536-538.
SGD	Robbins, H., & Monro, S. (1951). A stochastic approximation method. The annals of mathematical statistics, 400-407.
One vs All & One vs One	Multiple papiers, pas d'auteurs spécifiques!