Apprentissage statistique : modélisation descriptive et introduction aux réseaux de neurones (RCP208)

Réseaux de neurones

Michel Crucianu
(prenom.nom@cnam.fr)
http://cedric.cnam.fr/vertigo/Cours/ml/

EPN05 Informatique
Conservatoire National des Arts & Métiers, Paris, France

15 janvier 2026

Plan du cours

2 De la description à la prédiction

3 Capacités de représentation

4 Apprendre à partir de données

5 Descente de gradient

6 Amélioration de la convergence

7 Calcul du gradient
- Sans couche cachée
- Rétropropagation

8 Apprentissage et généralisation

9 Représentations apprises par les réseaux de neurones

Prédiction vs description

- Analyse des données : exploration des relations entre multiples variables, mise en évidence des relations saillantes entre groupes de variables ou entre groupes d'observations
  - → Méthodes factorielles, réduction de dimension, classification automatique
- Modélisation descriptive : description synthétique de la distribution des observations
  - → Classification automatique, estimation de densité
- Modélisation prédictive (ou décisionnelle) : prédire les valeurs d'une variable expliquée (ou de sortie) à partir des valeurs de variables explicatives (ou d'entrée)
  - Les décisions s'appuient en général sur des prédictions (et parfois on ne fait pas la différence entre les deux)
  - L'analyse des données et la modélisation descriptive sont souvent des étapes préparatoires de la modélisation prédictive

Applications

- Prédiction : un prêt bancaire sera remboursé ou non ?
- Prédiction : cette partie d'image représente un piéton, un poteau ou une chaise ?
- Prédiction : le piéton traversera-t-il la rue ?
- Prédiction du classement d'un film dans l'ordre de préférence d'un abonné
- Prédiction de la densité du trafic automobile
- Prédiction de la pollution de l'océan à 10 mètres de profondeur
- Prédiction du cours d'une matière première (ou autre actif)

→ Sens élargi du mot « prédiction » : concerne un fait non observé car à venir ou trop coûteux à mesurer ou non observable/caché au système

Types de problèmes de prédiction

1. Classification (ou discrimination, plus traditionnellement appelé « classement ») : la variable expliquée est une variable nominale, chaque observation possède une modalité (appelée en général classe)
   - Exemples : le prêt sera remboursé ou non, une partie d'image représente une personne ou un poteau ou une chaise, etc.
2. Régression : la variable expliquée est une variable quantitative (domaine $\subset \mathbb{R}$)
   - Exemples : densité de trafic, pollution en profondeur, cours d'un actif
3. *Ranking* (« classement » en français, possibilité de confusion avec classification ci-dessus) : la variable expliquée est une variable ordinale
   - Exemple : classement d'un article dans l'ordre de préférence d'un utilisateur
4. Prédiction structurée : la variable expliquée prend des valeurs dans un domaine de données structurées (les relations entre parties comptent)
   - Exemple : extraire une entité nommée d'une phrase

Comment décider ?

- Comment un banquier du dix-septième siècle décidait-il d'accorder un prêt à un armateur pour monter une expédition ?

  Décision d'accorder le prêt si prédiction de bon remboursement
  - L'armateur a de quoi garantir le prêt en cas d'échec de l'expédition ?
  - L'armateur a un navire récent avec un capitaine expérimenté ?
  - (Un peu) plus complexe : l'armateur a déjà bien remboursé un autre prêt et a un navire récent avec un capitaine expérimenté, a-t-il de quoi garantir 50% du prêt ?
  - Règles plus complexes, avec plus de facteurs $\rightarrow$ traitement algorithmique nécessaire
- Comment obtenir une règle permettant de décider (ici, d'accorder ou non un prêt) ?
  1. A partir d'une parfaite compréhension du problème
     - Exemple : prêt accordé si le débiteur donne des gages à hauteur de 100% du montant
     - On ne peut avoir une « parfaite compréhension » que si le nombre de facteurs considérés est très (trop ?) réduit et des aléas exclus $\Rightarrow$ règles résultantes très (ou trop) limitatives
  2. A partir de données : constats réalisés sur un (grand, si possible) nombre de cas antérieurs, pour lesquels on connaît à la fois les valeurs prises par les facteurs qui comptent (taux de garantie, âge du capitaine) et l'issue (remboursement/défaillance)
     - « Facteurs qui comptent » : variables explicatives (ou prédictives, ou d'entrée)
     - Issue : variable expliquée (ou prédite, ou de sortie)

Prédiction et décision

- En général, une décision se base sur une prédiction
  - On décide d'accorder un prêt si on prévoit son bon remboursement
  - On décide d'acheter un actif si on prévoit une hausse de son cours
- Le passage de prédiction à décision n'est pas toujours simple
  - La prédiction peut être un score (valeur entre 0 et 100%) pour une décision binaire (par ex. issue de la comparaison du score à un seuil)
  - La décision peut être humaine à partir d'une prédiction obtenue par un modèle
- Malgré cela, on parle souvent indifféremment de modèle prédictif ou décisionnel
- Par modèle prédictif on entend ici une fonction f qui, à partir des valeurs (observées) des variables explicatives, calcule une valeur (non observée) pour la variable expliquée

$$\text{valeurs variables explicatives} \xrightarrow{f} \text{valeur variable expliquée}$$

Questions majeures

- Un modèle f dans une famille $\mathcal{F}$ est en général défini par un ensemble de paramètres

1. Dans quelle famille $\mathcal{F}$ de fonctions chercher le modèle f ?
   - Quelles dépendances peut représenter (ou approcher) une famille $\mathcal{F}$?
   - Les capacités d'approximation d'une famille $\mathcal{F}$ sont suffisantes pour les données ?
2. Comment trouver le modèle f dans la famille $\mathcal{F}$?
   - Quel critère optimiser pour trouver les paramètres qui définissent le modèle optimal f^* ?
   - Quel algorithme d'optimisation employer ?

Plan du cours

2 De la description à la prédiction

3 Capacités de représentation

4 Apprendre à partir de données

5 Descente de gradient

6 Amélioration de la convergence

7 Calcul du gradient
- Sans couche cachée
- Rétropropagation

8 Apprentissage et généralisation

9 Représentations apprises par les réseaux de neurones

Codage des variables

- Variables quantitatives : codage numérique
- Variables nominales : codage disjonctif (*one-hot*)

Enseignant	1 0 0 ...0 0 0
Médecin	0 1 0 ...0 0 0
Technicien	0 0 1 ...0 0 0
...	...

- Variables ordinales (à nombre limité de modalités) : codage unaire

Pas du tout d'accord	0 0 0 0
Pas d'accord	0 0 0 1
Ni en désaccord ni d'accord	0 0 1 1
Plutôt d'accord	0 1 1 1
Tout à fait d'accord	1 1 1 1

Représenter les dépendances

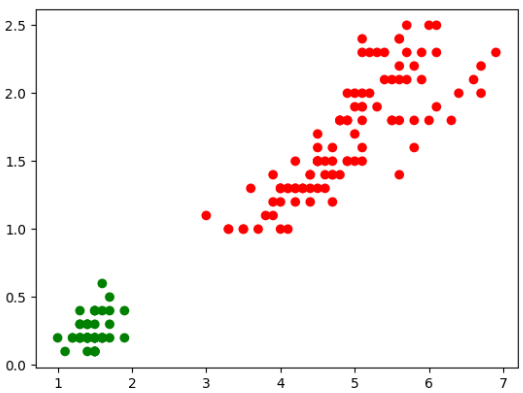
- Notations
  - Variables explicatives (en nombre de d) : $X_1, \cdots, X_d$
  - Variable expliquée : Y
  - Observations (en nombre de n) : $o_i = (x_{i1}, \cdots, x_{id}, y_i)$, $1 \leq i \leq n$
- Familles $\mathcal{F}$ candidates pour trouver f tel que

$$(x_{i1}, \cdots, x_{id}) \xrightarrow{f} y_i \qquad 1 \leq i \leq n$$

  - Régression (Y quantitative) :
    - Linéaires : $f(x_1, \cdots, x_d) = w_0 + \sum_{j=1}^{d} w_j x_j$, où $w_j \in \mathbb{R}, 0 \leq j \leq d$ sont les paramètres
    - Polynomiales, ...
    - Capables d'approximation universelle ?
  - Classification à deux classes (Y binaire) :
    - Linéaires seuillées : $f(x_1, \cdots, x_d) = \theta\left(w_0 + \sum_{j=1}^{d} w_j x_j\right)$, où θ est la fonction de Heaviside
    - ...

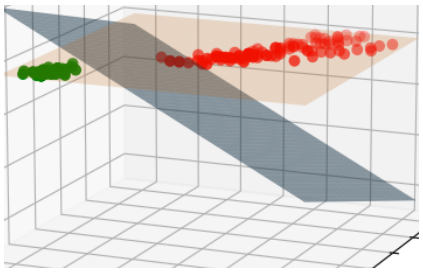
Exemple simple : les iris (Anderson, Fisher)

- Données (bien connues, `https://fr.wikipedia.org/wiki/Iris_de_Fisher`) : 150 observations de fleurs d'iris
  - Variables explicatives (4) : longueurs et largeurs des sépales et des pétales (en cm)
  - Variable expliquée : la variété (*Iris Setosa*, *Iris Versicolo(u)r*, *Iris Virginica*)
- Nous considérons
  - Deux variables explicatives seulement : longueur pétales (X_1, en abscisse dans la figure) et largeur pétales (X_2, en ordonnée)
  - Classification à 2 classes : *Setosa* (vert dans la figure) et *Versicolor,Virginica* (rouge)



Fonctions linéaires seuillées

- Fonction $f(x_1, x_2) = \begin{cases} 1 & \text{si } z = w_0 + x_1 w_1 + x_2 w_2 \geq 0 \\ 0 & \text{sinon} \end{cases}$

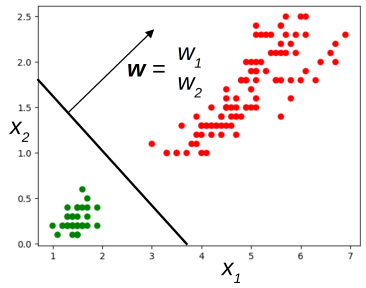


  - plan gris : $z = w_0 + x_1 w_1 + x_2 w_2$
  - plan orange : $z = 0$

→ frontière : intersection entre ces plans ⇒ linéaire

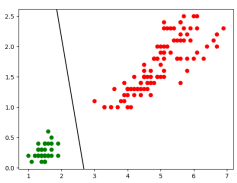
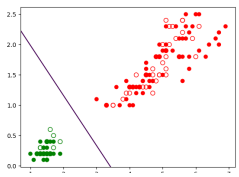
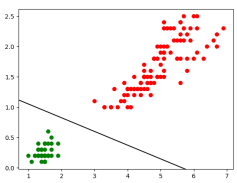
  - Obs. : $x_1 = x_2 = 0 \Rightarrow z = w_0$

- Autre perspective : projection orthogonale de $\mathbf{x} = [x_1, x_2]^\top$ sur $\mathbf{w} = [w_1, w_2]^\top$, addition de $\frac{w_0}{\|\mathbf{w}\|}$ et comparaison à 0 ⇒ frontière linéaire orthogonale à $\mathbf{w}$:



Fonctions linéaires seuillées (2)

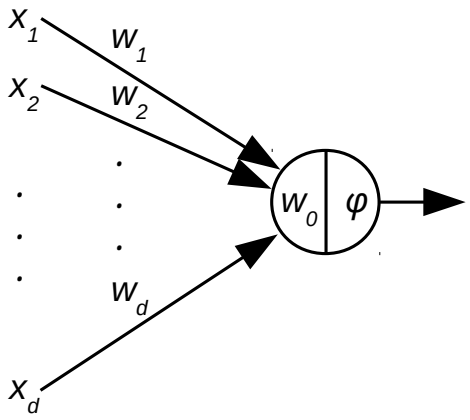
- Obs. 1 : si les observations sont écartées entre les deux classes, plusieurs frontières différentes (même une infinité de frontières) séparent parfaitement les classes :

Modèle 1	Modèle 2	Modèle 3
		
w_0, w_1, w_2 : 25.3, -9.51, -2.69	15.6, -4.58, -5.55	11.8, -2.07, -10.44

- Obs. 2 : une frontière ne change pas si tous les w_j sont multipliés par une meme constante $c \neq 0$

⇒ multiplicité des modèles qui permettent une bonne séparation

Neurone « formel »



- Introduit dans [6] comme modèle simplifié d'un neurone réel
  - Entrées $x_j, 1 \leq j \leq d$
  - Poids $w_j, 1 \leq j \leq d$, « seuil » w_0 ; w_j est le poids de la connexion avec l'entrée j
  - Fonction d'activation ϕ (dans [6] : fonction « marche » θ de Heaviside)
  - Sortie $\hat{y} = \phi\left(\sum_{j=1}^{d} w_j x_j - w_0\right)$

Expression matricielle de la sortie d'un neurone formel

- Notations
  - Vecteur d'entrée $\mathbf{x}_i = [x_{i1}, \cdots, x_{id}]^\top$: composé des valeurs des variables explicatives pour l'observation $o_i = (x_{i1}, \cdots, x_{id}, y_i)$, $1 \leq i \leq n$
  - Vecteur de poids $\mathbf{w} = [w_1, \cdots, w_d]^\top$: composé des coefficients correspondant aux variables explicatives

→ Sortie $\hat{y}_i = \phi\left(\mathbf{w}^\top \mathbf{x}_i - w_0\right)$: projection de $\mathbf{x}_i$ sur $\mathbf{w}$, suivi de soustraction du seuil w_0

⇒ la frontière reste linéaire quelle que soit ϕ

- Éviter de séparer le seuil w_0 des autres paramètres : on introduit une « variable » supplémentaire X_0 dont la valeur est toujours égale à -1 (ou à 1 si on change le signe de w_0), w_0 est le poids de la connexion avec cette entrée
  - Vecteur d'entrée redéfini : $\mathbf{x}_i = [x_{i0} = -1, x_{i1}, \cdots, x_{id}]^\top$
  - Vecteur de poids redéfini : $\mathbf{w} = [w_0, w_1, \cdots, w_d]^\top$

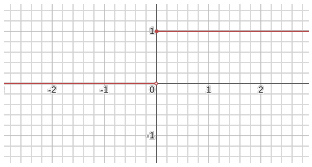
  ⇒ Sortie $\hat{y}_i = \phi\left(\mathbf{w}^\top \mathbf{x}_i\right)$

→ fonction affine (linéaire + translation) des entrées, suivie par l'application de ϕ

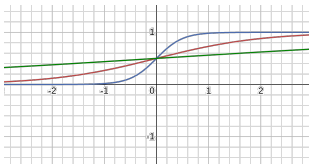
Fonctions d'activation

Fonction identité $\phi(x) = x$

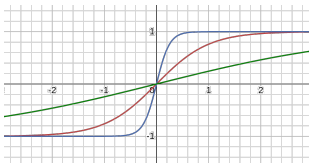
Fonction de Heaviside $\theta(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ 0 & \text{sinon} \end{cases}$



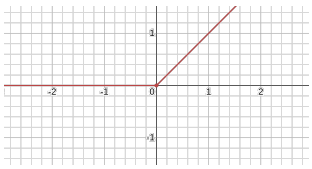
Sigmoïde (asymétrique) $\sigma(ax) = \frac{1}{1+e^{-ax}} \in [0, 1]$



Tangente hyperbolique $\tanh(ax) = \frac{e^{ax}-e^{-ax}}{e^{ax}+e^{-ax}} \in [-1, 1]$

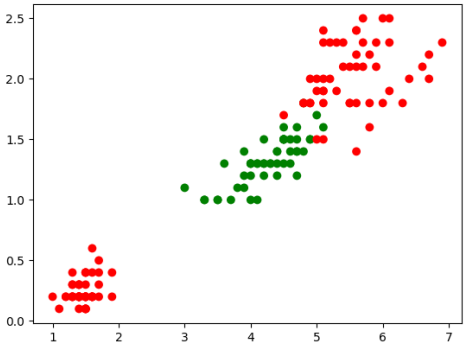


Rectified linear unit (ReLU) $\theta(x) = \begin{cases} x & \text{si } x \geq 0 \\ 0 & \text{sinon} \end{cases}$



Linéaire ne suffit pas

- Et si nous souhaitons séparer Iris Versicolor (en vert) des autres (en rouge) ?



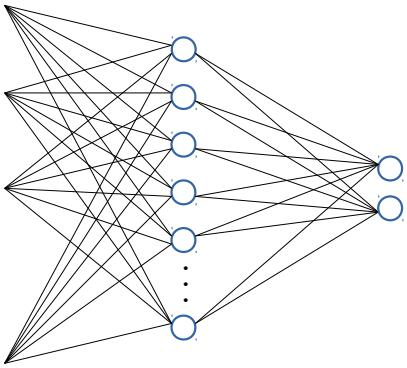
→ une frontière linéaire ne suffit pas

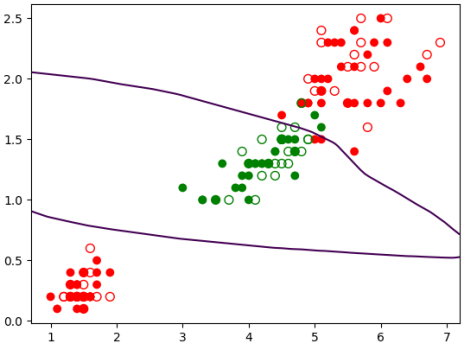
- Il est possible de combiner deux frontières linéaires : une entre versicolor et Setosa (en bas à gauche), l'autre entre Versicolor et Virginica (en haut à droite)
- … possible si la classe « non Versicolor » a un partitionnement favorable ⇒ ce n'est pas une solution en général

Solution : ajouter une couche de neurones (ou plusieurs)

→ perceptron multi-couches (PMC) → couche(s) « cachée(s) » (ni entrée, ni sortie)

- Couche(s) intermédiaire(s) augmente(nt) la capacité de représentation du réseau





… dans tous les cas ?

Calculs dans un PMC avec une couche cachée

- Vecteur d'entrée (dimension d) : $\mathbf{x} = [x_0 = -1, x_1, \cdots, x_d]^\top$
- Matrice de poids des c neurones de la première couche (cachée) :

$$\mathbf{W} = \begin{bmatrix} w_{01} & \cdots & w_{d1} \\ \vdots & \ddots & \vdots \\ w_{0c} & \cdots & w_{dc} \end{bmatrix}$$

→ Vecteur de sorties de la première couche : $\mathbf{h} = \Phi_1(\mathbf{W}\mathbf{x})$

- Φ_1 regroupe l'application des fonctions d'activation aux neurones de la première couche

- Vecteur de poids du neurone de sortie : $\mathbf{u} = [u_1, \cdots, u_h]^\top$

→ Activation du neurone de sortie : $\hat{y} = \phi_2\left(\mathbf{u}^\top \mathbf{h} + u_0\right)$

- u_0 peut être intégré à $\mathbf{u}$ suivant le même procédé que pour la première couche

- Si les neurones de la première couche ont des fonctions d'activation identité, alors $\hat{y} = \phi_2\left(\mathbf{u}^\top \mathbf{h} + u_0\right) = \phi_2\left(\mathbf{u}^\top \mathbf{W}\mathbf{x} + u_0\right) \Rightarrow$ fonction affine (linéaire + translation) des entrées $\Rightarrow$ frontière toujours linéaire !

Que calcule un PMC ?

- Comment déterminer quelles fonctions de transfert entrée → sortie peuvent être calculées suivant
  - le nombre de couches cachées ?
  - le nombre de neurones dans les différentes couches cachées ?
  - les fonctions d'activations des neurones dans différentes couches cachées ?

→ classes de fonctions obtenues par la composition de fonctions linéaires (multiplication vecteur d'entrée par matrice poids) et fonctions d'activation

- Très utile de déterminer les limites éventuelles : ce qui ne peut pas être calculé
  - Par ex., si les fonctions d'activation des neurones cachés sont l'identité, la fonction de transfert du réseau reste affine quel que soit le nombre de couches cachées et de neurones dans ces couches cachées

Résultats d'approximation universelle

- Résultats d'approximation universelle [2, 4, 7] : quelle que soit une fonction continue sur des compacts [1] de $\mathbb{R}^d$, son approximation aussi bonne que souhaité peut être obtenue avec un PMC ayant un nombre fini de neurones dans la couche cachée et une fonction d'activation non polynomiale
  - Une couche cachée suffit si assez de neurones et fonction d'activation appropriée
  - Fonctions d'activation polynomiales ⇒ fonctions de transfert polynomiales de degré borné (or, pour les polynômes des résultats d'approximation universelle exigent que le degré ne soit pas borné)

- D'autres résultats d'approximation existent, par ex. concernant le nombre de couches cachées de taille bornée

[1] Compact : borné et fermé.

Classification : plus de deux classes

- 2 classes : un neurone de sortie peut suffire, son activation indique la classe prédite
  - Fonction d'activation identité ou $\tanh$: une classe si $\hat{y} > 0$, l'autre classe si $\hat{y} < 0$
  - Fonction de Heaviside ou sigmoïde : une classe si $\hat{y} > 0.5$, l'autre classe si $\hat{y} < 0.5$

  $\rightarrow$ Solution non généralisable à plus de deux classes
- Plus de deux classes : codage *one-hot* pour la variable expliquée « classe » $\Rightarrow$ un neurone par classe $\rightarrow$ celui qui a l'activation la plus grande indique à quelle classe est affectée l'observation d'entrée
- Très utile d'avoir une interprétation de probabilités pour les sorties des neurones
  - La sortie de chaque neurone doit rester dans $[0, 1]$
  - La somme des sorties doit être égale à 1 (condition de normalisation)

  $\rightarrow$ même avec 2 classes on emploie souvent 2 neurones de sortie

$\rightarrow$ Fonction d'activation *softmax*

Activation *softmax*

- Les fonctions d'activation précédentes s'appliquent à chaque neurone seul
- *Softmax* s'applique à un ensemble de neurones pour respecter la normalisation
  - K classes $\rightarrow$ K neurones de sortie
  - Soit $\mathbf{x}$ le vecteur d'entrée de tous les neurones de sortie
  - Pour le neurone de sortie k, soit $\mathbf{w}_k$ son vecteur de poids et w_{0k} son seuil
  - On note $s_k = \mathbf{w}_k^\top \mathbf{x} + w_{0k}$ (sortie neurone k avant application *softmax*)
  - La sortie du neurone k avec *softmax* est

$$\hat{y}_k = \frac{e^{s_k}}{\sum_{j=1}^{K} e^{s_j}}$$

  - Chaque sortie est interprétée[2] comme une probabilité *a posteriori* de classe :

$$P(k|\mathbf{x}) = \hat{y}_k(\mathbf{x})$$

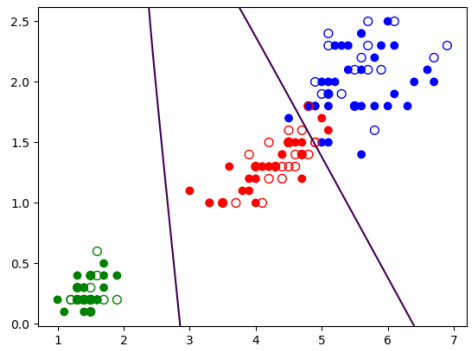
  - *Softmax* garantit la normalisation : $\forall \mathbf{x}, \sum_{k=1}^{K} P(k|\mathbf{x}) = 1$

[2] Si la normalisation autorise de traiter ces valeurs comme des probabilités, rien ne garantit en général qu'après l'apprentissage du réseau ces valeurs soient vraiment les probabilités *a posteriori* des classes !

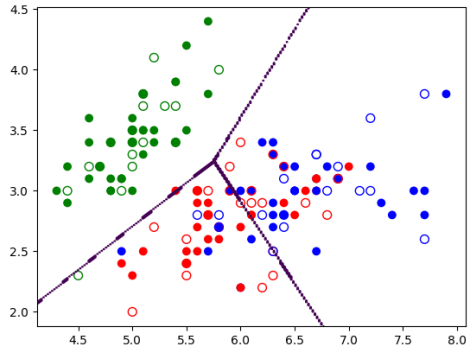
Activation *softmax* : illustrations

- Données Iris avec les trois classes : Setosa, Versicolor, Virginica
- Trois neurones (3 classes) avec activation *softmax* ; pas de couche cachée

Sur longueurs et largeurs pétales



Sur longueurs et largeurs sépales

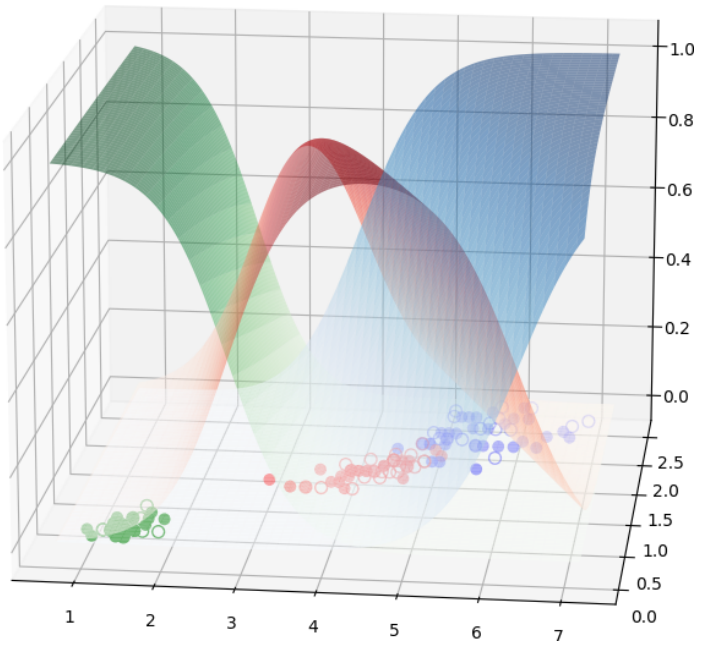


- Remarque : les traits correspondent aux frontières ; sans couche cachée la frontière entre deux classes reste linéaire !

Activation *softmax* : illustrations (2)

- Données Iris avec les trois classes : Setosa, Versicolor, Virginica
- Activations de 3 neurones (3 classes) avec *softmax*, sans couche cachée :

Sur longueurs et largeurs pétales



Fonctions d'erreur

- Observation $(\mathbf{x}, \mathbf{y})$, où $\mathbf{x} = [x_1, \cdots, x_d]^\top$
- Mesurer l'écart entre la prédiction $\hat{\mathbf{y}}(\mathbf{x})$ du modèle et la sortie désirée $\mathbf{y}$:
  - Problème de classification :
    - Erreur 0-1 : $\mathcal{E}_{01}(\hat{\mathbf{y}}, \mathbf{y}) = \begin{cases} 1 & \text{si classe prédite} \neq \text{vraie classe} \\ 0 & \text{sinon} \end{cases}$
    - Entropie croisée : K classes, codage *one-hot*, la classe correcte pour $\mathbf{x}$ est k^* (donc $y_{k^*} = 1$ et $y_k = 0$ pour tout $k \neq k^*$), un neurone de sortie par classe, le neurone k prédit la probabilité d'appartenance de $\mathbf{x}$ à la classe k :

$$\mathcal{E}_{ce}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k = -\log \hat{y}_{k*}$$

  - Problème de régression :
    - Erreur quadratique $\mathcal{E}_2(\hat{y}, y) = \|\hat{y} - y\|^2$ (calculée pour chaque variable expliquée si plusieurs)
- Le choix d'une fonction d'erreur dépend de l'objectif : certaines utiles pour rapporter de façon succincte les performances (erreur 0-1), d'autres permettent aussi d'apprendre le modèle (entropie croisée)

Plan du cours

2 De la description à la prédiction

3 Capacités de représentation

4 Apprendre à partir de données

5 Descente de gradient

6 Amélioration de la convergence

7 Calcul du gradient
- Sans couche cachée
- Rétropropagation

8 Apprentissage et généralisation

9 Représentations apprises par les réseaux de neurones

Obtenir un modèle à partir de données

- Le modèle est obtenu à partir d'observations d'apprentissage, étiquetées (sont connues les valeurs des variables explicatives et la valeur de la variable expliquée)
- « Apprendre » = trouver le modèle, ce qui revient en général à trouver les valeurs des paramètres du modèle
  - Pour un réseau de neurones d'architecture donnée : les poids et les seuils
- Quel critère permet de trouver les paramètres ?
  - Objectif : faire les meilleures prédictions sur de futures observations, inconnues mais supposées issues de la même distribution que les exemples d'apprentissage

  → Minimisation des erreurs faites sur les exemples d'apprentissage, en ajoutant un terme de régularisation pour maîtriser la complexité du modèle
- Comment déterminer les paramètres ?
  - Plupart des cas : déterminés par application d'un algorithme itératif d'optimisation
  - Quelques rares cas (par ex. régression linéaire) : peuvent être calculés analytiquement

Optimisation itérative

- Principe :
  1. Initialiser les paramètres ($\sim$ aléatoirement suivant distribution convenable)
  2. A chaque itération t, modifier les paramètres pour se diriger vers un optimum global ou (plus souvent) local
     - Vecteur de paramètres $\mathbf{w}$ (regroupant les poids et les seuils des différentes couches)

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \Delta\mathbf{w}_t$$

     - Différentes méthodes peuvent être employées pour trouver $\Delta\mathbf{w}_t$
     - Critères d'arrêt : lorsque l'optimum est atteint ($\Delta\mathbf{w}_t \approx 0$), ou après un nombre pré-déterminé d'itérations, ou lorsqu'un autre critère commence à augmenter (par ex. l'erreur sur les données de validation), etc.
- Méthode typique : descente de gradient, sous conditions :
  - L'architecture du réseau est fixée
  - La fonction de transfert (entrée → sortie) du réseau est différentiable
  - La fonction d'erreur est différentiable

Plan du cours

2 De la description à la prédiction

3 Capacités de représentation

4 Apprendre à partir de données

5 Descente de gradient

6 Amélioration de la convergence

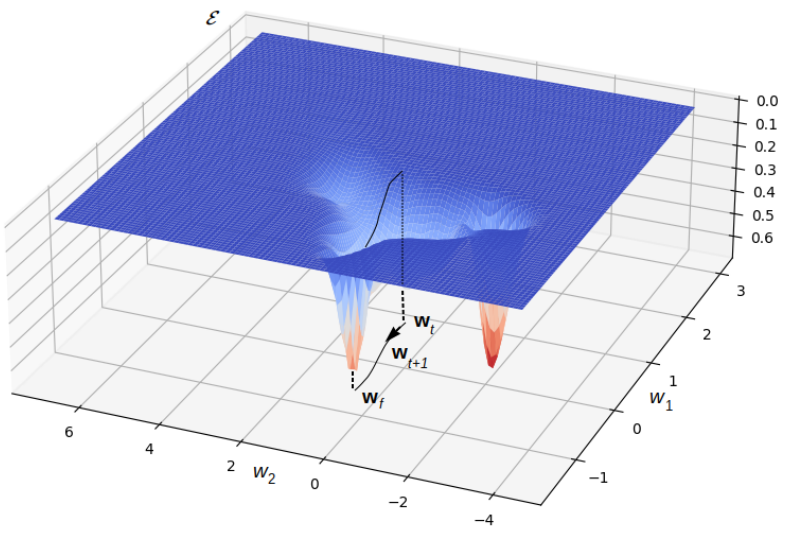
7 Calcul du gradient
- Sans couche cachée
- Rétropropagation

8 Apprentissage et généralisation

9 Représentations apprises par les réseaux de neurones

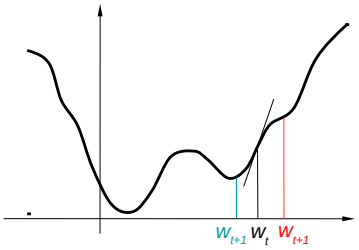
Descente de gradient : illustration simple

- Les 2 paramètres sont représentés par les axes du plan horizontal
- Surface : valeurs du critère à minimiser (par ex. erreur d'apprentissage régularisée)
- A chaque itération t on applique à $\mathbf{w}_t$ une modification $\Delta\mathbf{w}_t$ dans la direction opposée au gradient du critère par rapport aux paramètres
- Courbe dans le plan des paramètres : trajectoire lors des itérations successives
- Courbe sur la surface : évolution du critère lors des itérations successives



Descente de gradient : cas unidimensionnel

- Considérons le cas très simple d'une fonction $g : \mathbb{R} \to \mathbb{R}$, notons son argument par w



- Gradient de g dans $w = w_t$: dérivée de g par rapport à w dans $w = w_t$, $\frac{dg}{dw}(w_t)$
  - La dérivée de g par rapport à w est positive dans $w = w_t$, en appliquant une modification $w_{t+1} = w_t + \Delta w_t$, où $\Delta w_t = \eta \frac{dg}{dw}(w_t)$ avec $\eta > 0$, on augmente la valeur de g (remontée de gradient)

  → Pour se diriger plutôt vers un **minimum** de g, $\Delta w_t = -\eta \frac{dg}{dw}(w_t)$ avec $\eta > 0$ (**descente** de gradient)

Descente de gradient : cas multidimensionnel

- Fonction $g : \mathbb{R}^m \to \mathbb{R}$, d'argument vectoriel $\mathbf{w}$
  - Gradient de g dans $\mathbf{w} = \mathbf{w}_t$: vecteur des dérivées partielles de g par rapport à chaque composante de $\mathbf{w}$ dans $\mathbf{w} = \mathbf{w}_t$, $\nabla g(\mathbf{w}_t) = \left[\frac{\partial g}{\partial w_1}(\mathbf{w}_t), \cdots, \frac{\partial g}{\partial w_m}(\mathbf{w}_t)\right]^\top$
  - Descente de gradient : $\Delta \mathbf{w}_t = -\eta \nabla g(\mathbf{w}_t)$

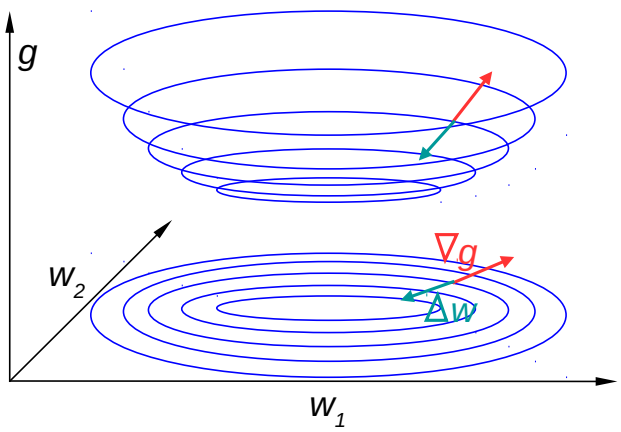
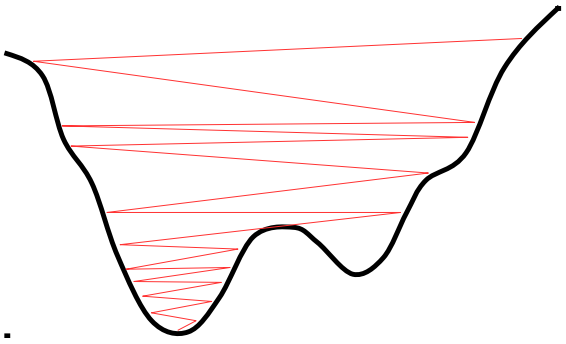


FIG. – Descente de gradient : illustration pour deux paramètres ($g : \mathbb{R}^2 \to \mathbb{R}$) ; courbes de niveau représentées en bleu

Descente de gradient : à quelle vitesse ?

- Descente de gradient $\Delta \mathbf{w}_t = -\eta \nabla g(\mathbf{w}_t)$: convergence (vers minimum global ou local) assurée si η, appelé vitesse (d'apprentissage), est suffisamment faible pour que $\Delta \mathbf{w}$ soit infinitésimal
  - Avec η trop faible la convergence est trop lente
  - Avec η trop élevé la convergence peut être lente aussi (voir figure) et n'est plus assurée



  - Quelle que soit la valeur fixe de η, si le gradient est élevé (= pente forte) alors $\Delta \mathbf{w}$ est (potentiellement trop) élevé, alors que si le gradient est faible (= pente faible) $\Delta \mathbf{w}$ est faible ($\rightarrow$ convergence lente)

$\rightarrow$ Diverses méthodes d'amélioration de la convergence

Gradient de l'erreur sur *n* exemples

- Exemples d'apprentissage : $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_i, \mathbf{y}_i), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$

1. Par lot global (*batch*) : cumul de l'erreur sur la totalité des exemples, $\mathcal{E} = \sum_{i=1}^{n} \mathcal{E}(\hat{\mathbf{y}}_i, \mathbf{y}_i)$, ensuite mise à jour des paramètres $\Delta \mathbf{w}_t = -\eta \nabla \mathcal{E}$; itération suivante : nouveau calcul de l'erreur sur le même batch
2. Stochastique (*online*) : choix aléatoire d'un exemple, mise à jour des paramètres après calcul de l'erreur sur chaque exemple, $\Delta \mathbf{w}_t = -\eta \nabla \mathcal{E}(\hat{\mathbf{y}}_i, \mathbf{y}_i)$; itération suivante : nouveau choix aléatoire d'un exemple
3. Par mini-lot (*mini-batch*) : choix d'un (petit par rapport à *n*) échantillon d'exemples $\mathcal{S}_j$, cumul de l'erreur sur cet échantillon, mise à jour des poids $\Delta \mathbf{w}_t = -\eta \nabla \mathcal{E}(\mathcal{S}_j)$; itération suivante : nouveau choix d'un échantillon
   - En *mini-batch* le gradient de l'erreur globale est suivi de façon très approximative et le nombre d'itérations est plus élevé qu'en *batch* mais le coût global est plus faible (nombre itérations $\times$ coût chaque itération)
   - Descente de gradient stochastique (*Stochastic Gradient Descent*, SGD) : 2 et 3

Plan du cours

2 De la description à la prédiction

3 Capacités de représentation

4 Apprendre à partir de données

5 Descente de gradient

6 Amélioration de la convergence

7 Calcul du gradient
- Sans couche cachée
- Rétropropagation

8 Apprentissage et généralisation

9 Représentations apprises par les réseaux de neurones

Aspect réel d'une fonction à minimiser

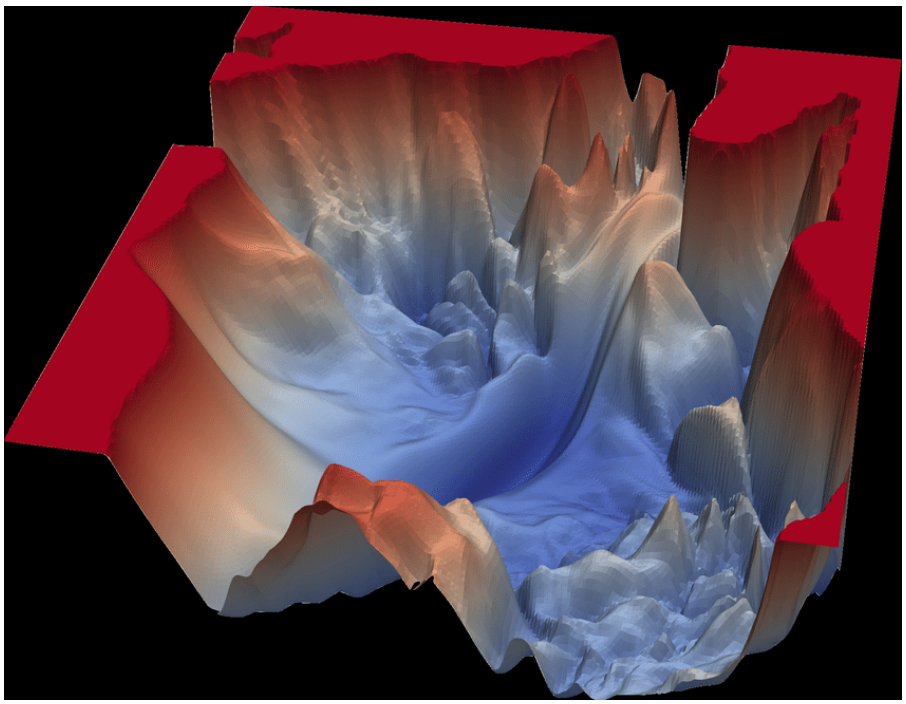


FIG. – Critère de coût à minimiser - cas réel (illustration issue de [9])

Amélioration de la convergence : adaptation de la vitesse η

- Motivations :
  1. La convergence de la descente de gradient est démontrée lorsque les modifications sont infinitésimales ou η tend vers 0
  2. Si η est trop élevé, l'exploration de l'espace est ∼aléatoire et les valeurs successives de $g(\mathbf{w}_t)$ peuvent aussi bien diminuer qu'augmenter → réduire η
  3. Minimum dans un « puits » : η trop élevé ne permet pas de descendre dans le puits → réduire η
  4. Traversée d'un « plateau » : η trop faible rend la traversée trop longue → augmenter η

→ Méthodes très diverses qui prennent en compte une ou plusieurs des motivations

- Scikit-learn avec `learning_rate='adaptive'` : si lors de 2 époques successives f ne diminue pas d'au moins `tol`, diviser η par 5 ; répond aux motivations 1 à 3
- D'autres méthodes ont été intégrées à des approches plus complexes, voir la suite

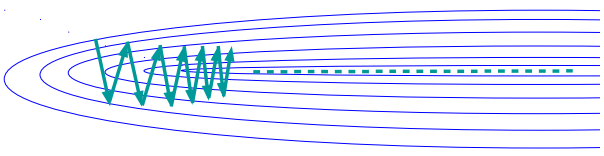
Amélioration de la convergence : inertie (*momentum*)

- Motivations :
  - Pente faible ⇒ gradient faible, → accumuler les modifications précédentes permet d'augmenter la vitesse
  - Direction du gradient change fortement à chaque itération → une « inertie » devrait lisser la trajectoire et accélérer la convergence

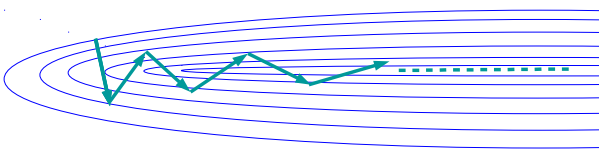
→ Calcul mise à jour : $\Delta\mathbf{w}_t = -\eta\nabla g(\mathbf{w}_t) + \beta\Delta\mathbf{w}_{t-1}$

- Coefficient β : typiquement $\in [0.8, 0.9]$

Trajectoire sans inertie :



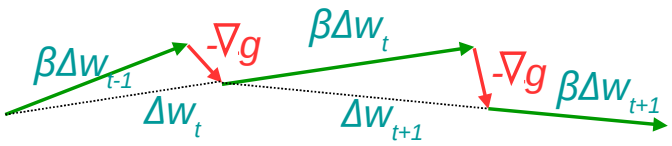
Trajectoire avec inertie :



- SGD : trajectoire suit les gradients par *mini-batch*, non le gradient de l'erreur totale
  → l'inertie a un effet de lissage de la trajectoire et accélération de la convergence

Amélioration de la convergence : méthode de Nesterov

- Motivation (voir par ex. [8]) : au lieu de cumuler la modification donnée par le gradient et l'inertie (utilisation classique de l'inertie), plutôt suivre l'inertie (flèche verte) et ensuite utiliser le gradient (dans le point intermédiaire, flèche rouge) pour corriger



→ Calcul mise à jour : $\Delta \mathbf{w}_t = \beta \Delta \mathbf{w}_{t-1} - \eta \nabla g(\mathbf{w}_t + \beta \Delta \mathbf{w}_{t-1})$

Amélioration de la convergence : Adam [5]

- Motivations issues de caractéristiques des problèmes complexes d'optimisation :
  - Le gradient contient beaucoup de « bruit » (complexité de la fonction à optimiser + aspect stochastique de SGD) → nécessaire de lisser la trajectoire
  - Toutes les composantes de $\mathbf{w}$ ne sont pas modifiées avec la même fréquence (par ex. en raison de vecteurs d'activation creux provenant des couches précédentes) → nécessaire d'individualiser par paramètre la vitesse d'apprentissage
- Solution Adam :
  - Calcul de moyennes décroissantes
    - Des gradients (→ lissage) : $\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \nabla g(\mathbf{w}_t)$
    - Des carrés (par composante) des gradients (→ individualisation vitesse) : $\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) \left(\nabla g(\mathbf{w}_t)\right)^2$
  - Correction des biais de ces estimations : $\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1}$, $\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2}$

→ Mise à jour poids w_i au moment t :

$$\Delta w_{it} = -\frac{\eta}{\sqrt{\hat{v}_{it}} + \epsilon} \hat{m}_{it}$$

- Valeurs typiques (hyper-)paramètres : $\eta = .001$, $\beta_1 = .9$, $\beta_2 = .999$ et $\epsilon = 10^{-8}$

Plan du cours

2 De la description à la prédiction

3 Capacités de représentation

4 Apprendre à partir de données

5 Descente de gradient

6 Amélioration de la convergence

7 Calcul du gradient
- Sans couche cachée
- Rétropropagation

8 Apprentissage et généralisation

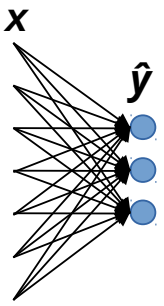
9 Représentations apprises par les réseaux de neurones

Calcul du gradient sans couche cachée

- Cumul sur données apprentissage :

  batch, *online* ou *mini-batch*

  (voir diapositive correspondante)



- Classification K classes (K neurones), activation *softmax*, erreur entropie croisée
  - Calculs présentés pour **un** vecteur d'entrée $\mathbf{x}$, avec seuils w_{0k} intégrés dans $\mathbf{w}_k$
  - Sortie neurone k (notation $s_k(\mathbf{x}) = \mathbf{w}_k^\top \mathbf{x}$) :

$$\hat{y}_k(\mathbf{x}) = \frac{e^{s_k(\mathbf{x})}}{\sum_{j=1}^{K} e^{s_j(\mathbf{x})}}$$

  - Si la classe correcte pour $\mathbf{x}$ est k^*, avec un codage *one-hot* de la sortie désirée $\mathbf{y}$, nous avons $y_{k^*} = 1$ et $y_k = 0$ pour $k \neq k^*$. L'erreur entropie croisée est alors :

$$\mathcal{E}_{ce}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k = -\log \hat{y}_{k^*}$$

Calcul du gradient sans couche cachée (2)

- Dérivation fonction composée $(f \circ g \circ h)(z)$: $\frac{df(g(h))}{dz} = \frac{df}{dg}\frac{dg}{dh}\frac{dh}{dz}$
- Dérivées partielles (points $\mathbf{w}_t$, $\widehat{\mathbf{y}}_t$, et $\mathbf{s}_t$ où la dérivée est calculée, itération t) :

$$\frac{\partial \mathcal{E}_{ce}}{\partial w_{kj}}(\mathbf{w}_t) = \frac{\partial \mathcal{E}_{ce}}{\partial \hat{y}_{k^*}}(\widehat{\mathbf{y}}_t)\frac{\partial \hat{y}_{k^*}}{\partial s_k}(\mathbf{s}_t)\frac{\partial s_k}{\partial w_{kj}}(\mathbf{w}_t) \tag{1}$$

- Il est très utile d'exprimer le gradient en fonction de quantités déjà calculées

→ Pour le premier facteur de l'équation (1) :

$$\mathcal{E}_{ce}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \hat{y}_{k^*} \quad \Rightarrow \quad \frac{\partial \mathcal{E}_{ce}}{\partial \hat{y}_{k^*}} = -\frac{1}{\hat{y}_{k^*}}$$

→ Pour le second facteur de l'équation (1) : $\hat{y}_k(\mathbf{x}) = \frac{e^{s_k(\mathbf{x})}}{\sum_{j=1}^K e^{s_j(\mathbf{x})}} \quad \Rightarrow$

$$\Rightarrow \frac{\partial \hat{y}_{k^*}}{\partial s_k} = \begin{cases} \frac{-e^{s_k} e^{s_{k^*}}}{\left(\sum_{j=1}^K e^{s_j}\right)^2} = -\hat{y}_k \hat{y}_{k^*} & \text{si} \quad k \neq k^* \\ \frac{e^{s_{k^*}}\left(\sum_{j=1}^K e^{s_j}\right) - e^{s_k} e^{s_{k^*}}}{\left(\sum_{j=1}^K e^{s_j}\right)^2} = \hat{y}_{k^*}(1 - \hat{y}_k) & \text{si} \quad k = k^* \end{cases}$$

Calcul du gradient sans couche cachée (3)

$$\Rightarrow \quad \frac{\partial \hat{y}_{k^*}}{\partial s_k} = \hat{y}_{k^*}(\delta_{k^* k} - \hat{y}_k)$$

$$(\delta_{k^* k} = \begin{cases} 1 & \text{si } k = k^* \\ 0 & \text{sinon} \end{cases} \quad \text{permet d'écrire le résultat sous une forme plus compacte})$$

- Pour le troisième facteur de l'équation (1) :

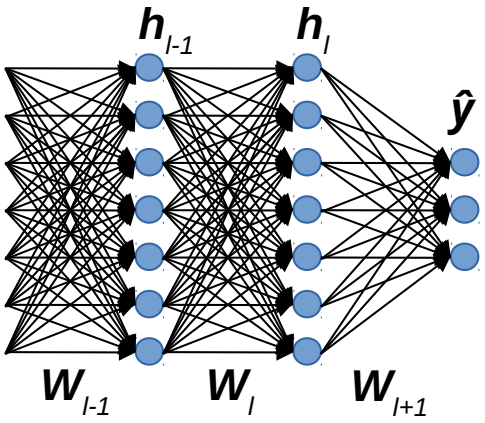
$$s_k(\mathbf{x}) = \mathbf{w}_k^\top \mathbf{x} = \sum_{j=1}^d w_{kj} x_j \quad \Rightarrow \quad \frac{\partial s_k}{\partial w_{kj}} = x_j$$

→ En conséquence

$$\begin{aligned} \frac{\partial \mathcal{E}_{ce}}{\partial w_{kj}}(\mathbf{w}_t) &= -\frac{1}{\hat{y}_{k^*}}(\hat{y}_{k^*})(\delta_{k^* k} - \hat{y}_k)x_j \\ &= (\hat{y}_k - \delta_{k^* k})x_j \end{aligned}$$

Et avec plusieurs couches ?

- Notations :
  - Matrice de poids des neurones de la couche l : $\mathbf{W}_l$ (seuils intégrés à la matrice)
  - Sorties des neurones de la couche l : $\mathbf{h}_l$; fonction activation couche l : ϕ_l



- Propagation de l'activation (entrées $\rightarrow$ sorties, *forward pass*) :
  - Entrées $\mathbf{x} \rightarrow$ couche cachée 1 : $\mathbf{h}_1 = \phi_1(\mathbf{u}_1)$, avec $\mathbf{u}_1 = \mathbf{W}_1\mathbf{x}$
  - Couche cachée $l-1 \rightarrow$ couche cachée l : $\mathbf{h}_l = \phi_l(\mathbf{u}_l)$, avec $\mathbf{u}_l = \mathbf{W}_l\mathbf{h}_{l-1}$
  - Dernière couche cachée $l \rightarrow$ sorties : $\hat{\mathbf{y}} = \phi_{out}(\mathbf{u}_{l+1})$, avec $\mathbf{u}_{l+1} = \mathbf{W}_{l+1}\mathbf{h}_l$

Rétropropagation (*backpropagation*)

$\rightarrow$ Couche de sortie : $w_{kj(l+1)}$ poids neurone j couche $l \rightarrow$ neurone k couche sortie

- ϕ_{out} *softmax*, erreur entropie croisée : $\sim$ cas sans couche cachée mais h_{jl} remplace x_j

$$\frac{\partial \mathcal{E}_{ce}}{\partial w_{kj(l+1)}} = (\hat{y}_k - \delta_{k^* k})h_{jl}$$

- ϕ_{out} sigmoïde $\sigma\left(u_{k(l+1)}\right) = \frac{1}{1+e^{-u_{k(l+1)}}}$, erreur quadratique $\mathcal{E} = \|\hat{\mathbf{y}} - \mathbf{y}\|^2$:

$$\frac{\partial \mathcal{E}}{\partial w_{kj(l+1)}} = 2(\hat{y}_k - y_k)\hat{y}_k(1-\hat{y}_k)h_{jl}$$

- ϕ_{out} quelconque calculée par neurone de sortie, fonction d'erreur $\mathcal{E}$ quelconque :

$$\frac{\partial \mathcal{E}}{\partial w_{kj(l+1)}} = \frac{\partial \mathcal{E}}{\partial \hat{y}_k}\frac{d\phi_{out}}{du_{k(l+1)}}h_{jl}$$

Rétropropagation (2)

→ Couche l, cas général : w_{kjl} poids neurone j couche $l-1$ → neurone k couche l

Pour calculer $\frac{\partial \mathcal{E}}{\partial w_{kjl}}$, en considérant l'activation h_{kl} comme fonction scalaire intermédiaire, nous avons

$$\frac{\partial \mathcal{E}}{\partial w_{kjl}} = \frac{\partial \mathcal{E}}{\partial h_{kl}} \frac{\partial h_{kl}}{\partial u_{kl}} \frac{\partial u_{kl}}{\partial w_{kjl}}$$

or

$$u_{kl} = \sum_i w_{kil} h_{i(l-1)} \quad \Rightarrow \quad \frac{\partial u_{kl}}{\partial w_{kjl}} = h_{j(l-1)}$$

et

$$h_{kl} = \phi_l(u_{kl}) \quad \Rightarrow \quad \frac{\partial h_{kl}}{\partial u_{kl}} = \frac{d\phi_l}{du_{kl}}$$

Comment calculer $\frac{\partial \mathcal{E}}{\partial h_{kl}}$?

→ $\mathcal{E}$ dépend de h_{kl} à travers les activations de tous les neurones de la couche suivante $l+1$, donc à travers la fonction vectorielle $\mathbf{h}_{l+1}$ ⇒

Par une généralisation de la règle de dérivation des fonctions composées nous avons :

$$\frac{\partial \mathcal{E}}{\partial h_{kl}} = \sum_{i \in \text{couche } l+1} \frac{\partial \mathcal{E}}{\partial h_{i(l+1)}} \frac{\partial h_{i(l+1)}}{\partial h_{kl}}$$

Rétropropagation (3)

Or, $\frac{\partial \mathcal{E}}{\partial h_{i(l+1)}}$ ont déjà été calculées pour les neurones de la couche $l+1$

Donc, sachant que $\frac{\partial h_{i(l+1)}}{\partial h_{kl}} = \frac{\partial h_{i(l+1)}}{\partial u_{i(l+1)}} \frac{\partial u_{i(l+1)}}{\partial h_{kl}}$, nous obtenons

$$\frac{\partial \mathcal{E}}{\partial h_{kl}} = \sum_{i \in \text{couche } l+1} \frac{\partial \mathcal{E}}{\partial h_{i(l+1)}} \frac{\partial h_{i(l+1)}}{\partial u_{i(l+1)}} \frac{\partial u_{i(l+1)}}{\partial h_{kl}}$$

⇒ Les gradients de $\mathcal{E}$ par rapport aux activations $h_{i(l+1)}$ sont ainsi rétro-propagés (de la sortie vers l'entrée) pour le calcul du gradient de $\mathcal{E}$ par rapport aux poids :

$$\frac{\partial \mathcal{E}}{\partial h_{kl}} = \sum_{i \in \text{couche } l+1} \frac{\partial \mathcal{E}}{\partial h_{i(l+1)}} \frac{d\phi_{l+1}}{du_{i(l+1)}} w_{ik}$$

car $\frac{\partial h_{i(l+1)}}{\partial u_{i(l+1)}} = \frac{d\phi_{l+1}}{du_{i(l+1)}}$, où ϕ_{l+1} est la fonction d'activation des neurones de la couche $l+1$, et $\frac{\partial u_{i(l+1)}}{\partial h_{kl}} = w_{ik}$, avec w_{ik} le poids de la connexion qui relie la sortie du neurone k de la couche l à une entrée du neurone i de la couche $l+1$

Rétropropagation (4)

→ Couche 1 (d'entrée), w_{kj1} poids entre l'entrée j et le neurone k de la couche 1 :

$$\frac{\partial \mathcal{E}}{\partial w_{kj1}} = \frac{\partial \mathcal{E}}{\partial h_{k1}} \frac{\partial h_{k1}}{\partial u_{k1}} \frac{\partial u_{k1}}{\partial w_{kj1}}$$

avec $\frac{\partial \mathcal{E}}{\partial h_{k1}}$ et $\frac{\partial h_{k1}}{\partial u_{k1}}$ obtenus comme dans le cas général avec $l \leftarrow 1$, et $\frac{\partial u_{k1}}{\partial w_{kj1}} = x_j$ (composante j de l'entrée)

Plan du cours

2 De la description à la prédiction

3 Capacités de représentation

4 Apprendre à partir de données

5 Descente de gradient

6 Amélioration de la convergence

7 Calcul du gradient
- Sans couche cachée
- Rétropropagation

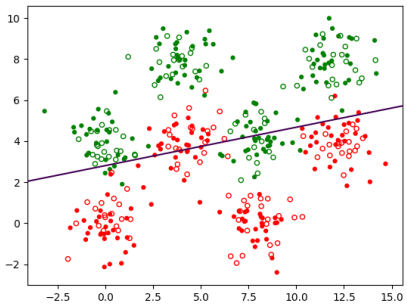
8 Apprentissage et généralisation

9 Représentations apprises par les réseaux de neurones

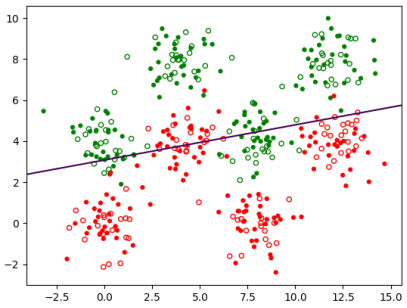
Erreur d'apprentissage et erreur de généralisation

- Modèle $\mathbf{x} \xrightarrow{f} \mathbf{y}, f \in \mathcal{F}$ appris sur données d'apprentissage $\mathcal{D}_n = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{1,\dots,n}$
- Objectif : faire les meilleures prédictions sur de futures observations, inconnues lors de l'apprentissage (mais supposées issues de la même distribution que les exemples d'apprentissage) $\rightarrow$ meilleure généralisation
- Estimation de l'erreur de généralisation par l'erreur sur échantillon de test, différent de l'échantillon d'apprentissage (mais issu de la même distribution)
- Dans quelle famille $\mathcal{F}$ chercher f ?
  - Plus la « capacité » de $\mathcal{F}$ est élevée, plus il est probable d'avoir dans $\mathcal{F}$ un modèle f^* qui approche la bonne dépendance (entrée $\rightarrow$ sortie)
  - Pouvons-nous trouver f^* à partir des observations d'apprentissage ?
- Illustration sur données simples bidimensionnelles :
  - Deux familles de modèles : $\mathcal{F}_l$ linéaires $\leftrightarrow$ $\mathcal{F}_{nl}$ non linéaires (PMC)
  - Quelles frontières obtenues, quelles erreurs d'apprentissage et de test ?

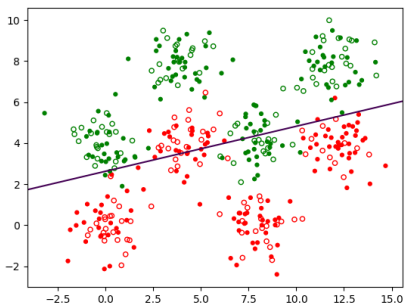
Résultats pour différents échantillons : cas linéaire



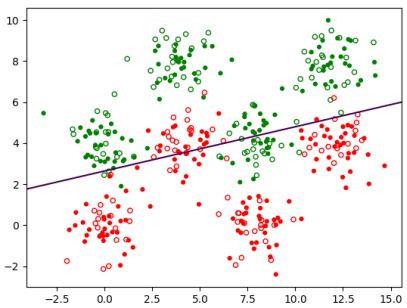
Score apprentissage : 0.8
Score test : 0.856



Score apprentissage : 0.837
Score test : 0.8

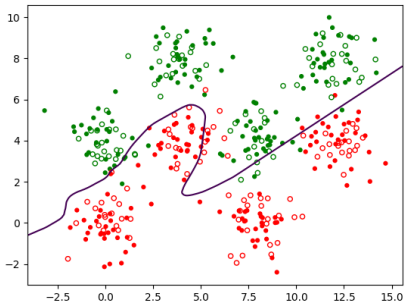


Score apprentissage : 0.837
Score test : 0.81

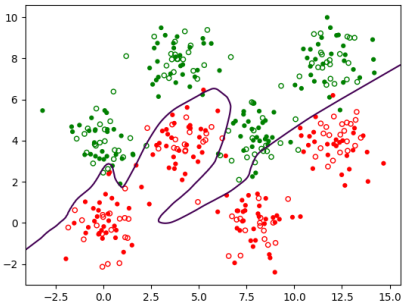


Score apprentissage : 0.86
Score test : 0.775

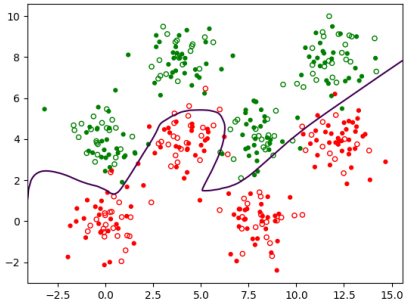
Résultats pour différents échantillons : cas non linéaire (PMC)



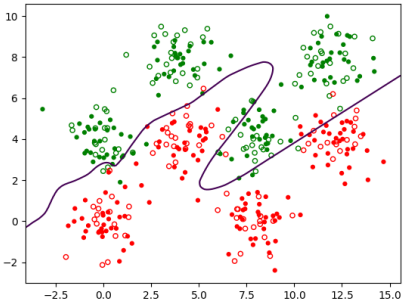
Score apprentissage : 0.946
Score test : 0.906



Score apprentissage : 0.966
Score test : 0.93



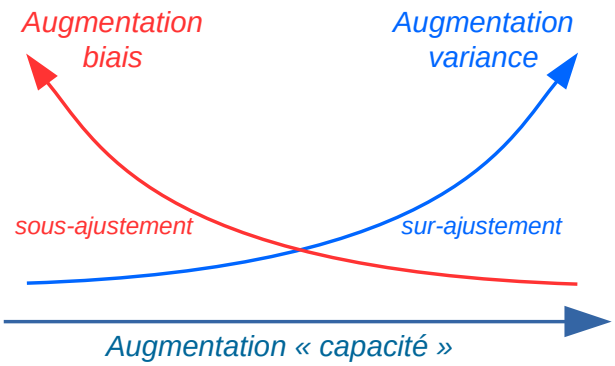
Score apprentissage : 0.96
Score test : 0.94



Score apprentissage : 0.946
Score test : 0.918

Biais et variance

- Observations :
  - Dans $\mathcal{F}_l$: la frontière varie peu avec l'échantillon mais les performances restent médiocres car on n'arrive pas à approcher la bonne frontière ($f \in \mathcal{F}_l \Rightarrow f$ sous-ajusté)
  - Dans $\mathcal{F}_{nl}$: meilleures performances mais la frontière varie beaucoup avec l'échantillon d'apprentissage (le modèle « sur-apprend », ou est sur-ajusté à l'échantillon)
- Ce constat est typique et exprimé souvent par le dilemme biais-variance : comment trouver la capacité qui correspond au bon compromis



Réduire l'écart entre apprentissage et généralisation

→ Chercher le modèle f dans une famille $\mathcal{F}$ de capacité suffisante mais « maîtriser » la complexité du modèle sélectionné, en écartant les modèles trop complexes de $\mathcal{F}$

→ Apprentissage régularisé :

$$f^* = \arg\min_{f \in \mathcal{F}} \left[\mathcal{E}_{\mathcal{D}_n}(f) + \alpha \mathcal{R}(f)\right]$$

- $\mathcal{R}(f)$: terme de régularisation explicite, pondéré par α ; pénalise la « complexité » du modèle f
- Un modèle f plus complexe peut diminuer l'erreur d'apprentissage $\mathcal{E}_{\mathcal{D}_n}(f)$ mais augmente la pénalité $\mathcal{R}(f)$

→ Compromis entre réduction de l'erreur d'apprentissage (risque empirique) et réduction de la complexité

→ Minimisation du Risque Empirique Régularisé (MRER)

Régularisation L_2 ou par « oubli » (*weight decay*)

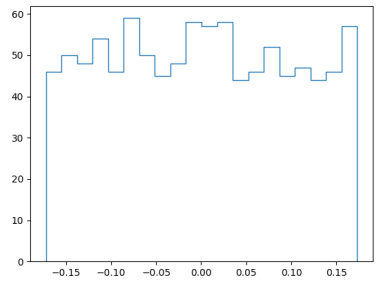
- Expression de $\mathcal{R}(f)$: $\mathcal{R}(f) = \|\mathbf{w}\|^2$

  $\mathbf{w}$: vecteur qui contient tous les poids (et biais) du réseau de neurones

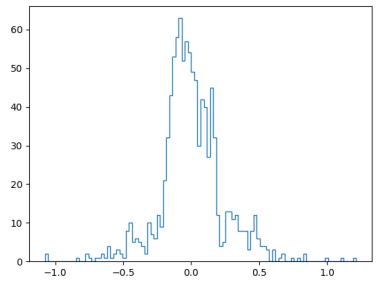
  → On parle de régularisation L_2
- Minimisation par descente de gradient : le gradient de $\mathcal{R}(f)$ par rapport aux poids est très facile à calculer, $\frac{\partial \mathcal{R}}{\partial w} = 2\alpha w$, la modification du poids sera donc $\Delta w_t = -\eta \frac{\partial \mathcal{E}}{\partial w}(w_t) - 2\alpha w_t$, d'où le nom « oubli » (*weight decay*)
- Constats :
  - Concentration autour de 0 de l'histogramme des poids
  - Réduction écart entre erreur d'apprentissage et de test
  - Si α trop élevé ⇒ augmentation des erreurs (régularisation excessive)
- Pourquoi la régularisation L_2 fonctionne ?
  - Cas général : réduction de la sensibilité au « bruit » en entrée
  - En plus, pour réseaux de neurones : les valeurs des poids restent plus proches de 0 ⇒ les fonctions d'activation travaillent plus dans leur partie quasi-linéaire ⇒ la fonction de transfert du réseau est plus « lisse »

Régularisation par « oubli » : histogrammes des poids

Après initialisation :

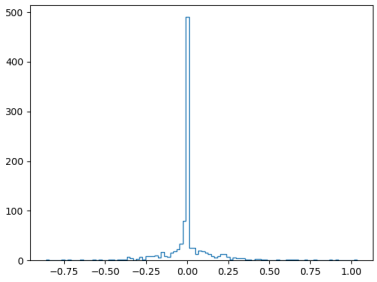


$\alpha = 0.0$



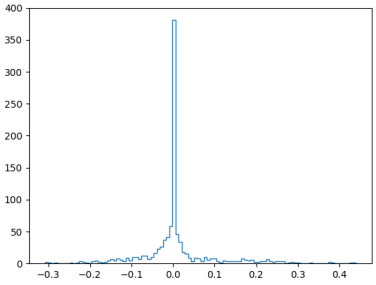
Score apprentissage : 0.844
Score test : 0.683

$\alpha = 0.5$



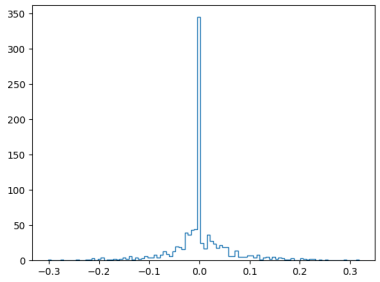
Score apprentissage : 866
Score test : 0.716

$\alpha = 5.0$



0.822
0.733

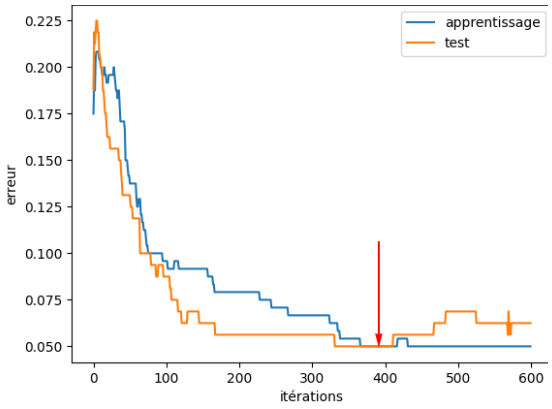
$\alpha = 10.0$



0.744
0.683

Régularisation par « arrêt précoce » (*early stopping*)

- Évolution de l'erreur d'apprentissage et de test (estimation de l'erreur de généralisation) lors des itérations d'apprentissage :

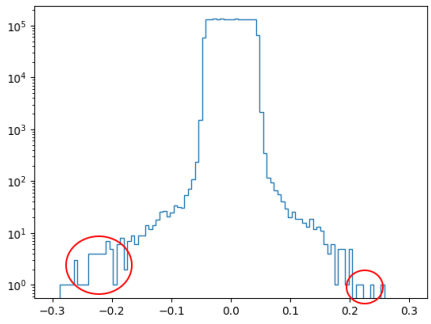
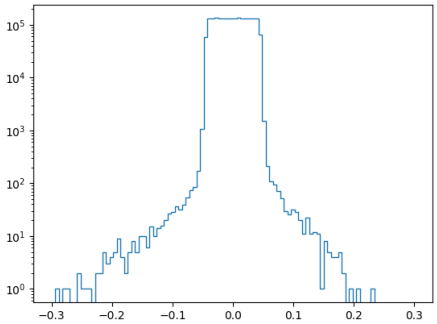


→ Régularisation par « arrêt précoce » : arrêt lorsque l'erreur de test est minimale (flèche dans la figure), non lorsqu'on atteint un minimum de l'erreur d'apprentissage

- Régularisation sans terme $\mathcal{R}(f)$ explicite
- L'échantillon de test devient échantillon de validation, un autre échantillon de test ($\neq$ apprentissage et validation) doit être employé pour estimer l'erreur de généralisation !

Régularisation par « arrêt précoce » (2)

- Pourquoi la régularisation par arrêt précoce fonctionne :
  - Prend explicitement en compte l'objectif : minimiser l'erreur de généralisation, estimée par l'erreur de validation
  - Permet d'arrêter la divergence (l'augmentation en valeur absolue) des valeurs des poids qui serait la conséquence de la poursuite de la minimisation de l'erreur
- Différences entre les histogrammes des valeurs des poids : peu saillantes

500 itérations :	400 itérations (arrêt précoce) :
	
Score apprentissage : 0.951	0.946
Score test : 0.928	0.937

Autres méthodes de régularisation

- Liste non exhaustive :
  1. L'utilisation de *mini-batch* (par opposition à *batch*) : l'introduction d'une dose de stochasticité fait converger vers des minimas plus « larges », associés à une meilleure généralisation
  2. Augmentation de données : différentes transformations appliquées aux données d'entrée ne devraient pas produire de changement sur la sortie → générer de nouvelles observations en transformant les entrées (et en conservant la même sortie)
  3. Normalisation des activations : permet d'éviter l'explosion de certaines valeurs d'activations qui peuvent avoir comme effet la saturation de neurones dans des couches suivantes (→ augmentation de la non linéarité, annulation des gradients rétro-propagés)
  4. *Dropout* (traduit par « décrochage » ou « abandon ») : désactivation aléatoire de neurones, avec une probabilité p, lors de l'apprentissage ; a comme effet la réduction de la coadaptation de différents neurones dans des couches successives
- En général plusieurs méthodes de régularisation sont utilisées conjointement !

Plan du cours

2 De la description à la prédiction

3 Capacités de représentation

4 Apprendre à partir de données

5 Descente de gradient

6 Amélioration de la convergence

7 Calcul du gradient
- Sans couche cachée
- Rétropropagation

8 Apprentissage et généralisation

9 Représentations apprises par les réseaux de neurones

Représentations internes dans les réseaux de neurones

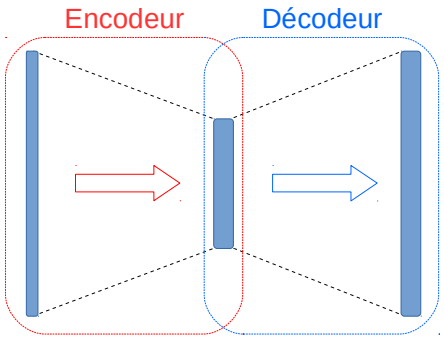
- Dans les perceptrons multi-couches (PMC) les prédictions sont obtenues grâce à des changements de représentation des données

→ utile d'examiner ces changements de représentation pour mieux comprendre le fonctionnement des PMC

- Deux cas analysés dans la suite :
  1. Problème de régression « auto-supervisé » : apprendre à reproduire en sortie le vecteur d'entrée à travers des couches cachées dont au moins une est de dimension inférieure à l'entrée ; on parle d'auto-encodeur
     → Lien avec l'analyse en composantes principales
  2. Problème de classification supervisée multi-classe : apprendre à affecter chaque vecteur d'entrée à une classe parmi plusieurs
     → Lien avec l'analyse factorielle discriminante

Auto-encodeur

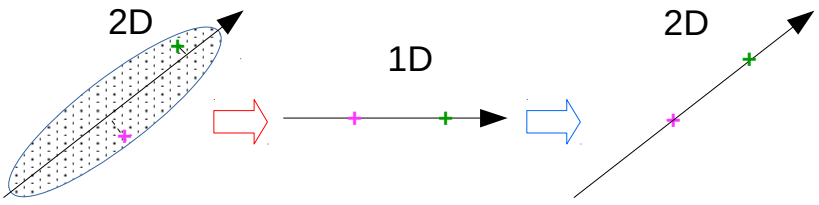
- Objectif : obtenir pour les vecteurs d'entrée des codes de dimension plus faible mais conservant le plus d'information permettant de reconstruire au mieux ces vecteurs
- Architecture : partie encodeur + partie décodeur
  - Encodeur : vecteur d'entrée $\mathbf{x} \rightarrow$ code $\mathbf{h}$
  - Décodeur : code $\mathbf{h} \rightarrow$ vecteur de sortie $\hat{\mathbf{y}} \approx$ vecteur d'entrée $\mathbf{x}$



- Apprentissage auto-supervisé (*self-supervised*) : sortie désirée $\mathbf{y} \equiv$ entrée $\mathbf{x}$

Auto-encodeur linéaire : analyse en composantes principales (ACP)

- Notations :
  - Matrice des données centrées $\mathbf{X}$ ($n \times d$), chaque ligne $\mathbf{x}_i^\top$ est une observation
  - Matrice $\mathbf{U}_k$ ($d \times k$) qui a comme colonnes les vecteurs propres unitaires $\mathbf{u}_j, 1 \leq j \leq k$ associés aux k plus grandes valeurs propres de la matrice des covariances empiriques
  - Codes $\mathbf{h}_i^\top$ de dimension $k < d$, chaque code est une ligne de la matrice $\mathbf{H}_k$ ($n \times k$)
- Auto-encodeur linéaire par ACP :
  - Encodeur : projection $\mathbf{x}_i \rightarrow \mathbf{h}_i$ sur les k composantes principales, $\mathbf{H}_k = \mathbf{X}\mathbf{U}_k$
  - Décodeur : reconstruction (approximative) des données $\mathbf{h}_i \rightarrow \hat{\mathbf{y}}_i \approx \mathbf{x}_i$, $\hat{\mathbf{X}}^* = \mathbf{H}_k\mathbf{U}_k^\top$



$\rightarrow$ $\hat{\mathbf{X}}^*$ est la meilleure approximation de rang k de $\mathbf{X}$: $\hat{\mathbf{X}}^* = \arg\min_{\hat{\mathbf{X}}} \|\hat{\mathbf{X}} - \mathbf{X}\|_F$, $\|\cdot\|_F$ étant la norme de Frobenius (théorème Eckart-Young-Mirsky)

Auto-encodeur : cas non linéaire

- L'approximation des observations par leur projection sur un sous-espace linéaire n'est pas toujours un bon choix :

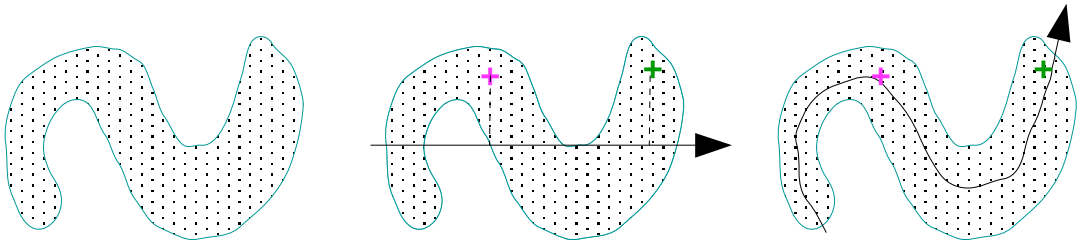


FIG. – Un sous-espace non linéaire (à droite) peut « ajuster » le nuage des observations mieux qu'un sous-espace linéaire (au milieu)

→ chercher un sous-espace non linéaire qui se rapproche plus des observations

- Le sous-espace doit rester assez « simple » pour bien ajuster la distribution des données et non simplement l'échantillon d'apprentissage

Auto-encodeur PMC et ACP

- Auto-encodeur PMC :
  - Couche de sortie de même dimension que l'entrée
  - Code = vecteur d'activation d'une couche cachée de dimension inférieure
  - Problème de régression, avec sortie désirée $\mathbf{y} \equiv$ entrée $\mathbf{x}$
  - Erreur à minimiser : erreur quadratique $\|\hat{\mathbf{x}} - \mathbf{x}\|^2$
- Cas le plus simple : une couche cachée de k neurones, fonction d'activation identité $\Rightarrow \hat{\mathbf{x}} = \mathbf{W}_2\mathbf{h} = \mathbf{W}_2\mathbf{W}_1\mathbf{x}$, on note $\mathbf{W} = \mathbf{W}_2\mathbf{W}_1$

→ Pour ce cas on peut montrer que

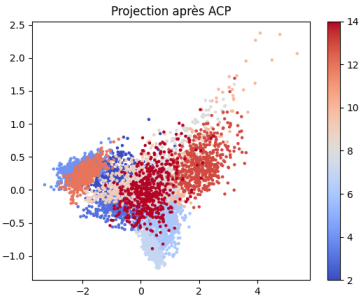
- L'erreur est minimisée lorsque la fonction de transfert entrée → sortie du PMC est la projection orthogonale sur le sous-espace engendré par les vecteurs propres associés aux k plus grandes valeurs propres de l'ACP des observations d'apprentissage
- La représentation des données sur la couche cachée correspond (à une transformation linéaire de rang maximal près) à la projection orthogonale sur ces k vecteurs propres
- Ce minimum de l'erreur est global et unique [1] (la solution $\mathbf{W}$ est unique)

- Cas général : l'encodeur et le décodeur comportent chacun au moins une couche cachée avec une fonction d'activation non linéaire

Auto-encodeur PMC et ACP (2)

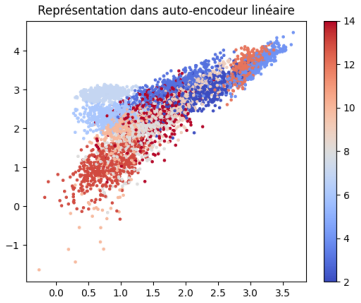
- Comparaison sur les données Textures (11 couleurs différentes ↔ 11 classes) :
  - ACP : projection sur les 2 premiers axes
  - PMC linéaire : 1 couche cachée de 2 neurones, projections sur couche cachée
  - PMC non linéaire : 3 couches cachées 20-2-20, projections sur couche de 2 neurones

ACP



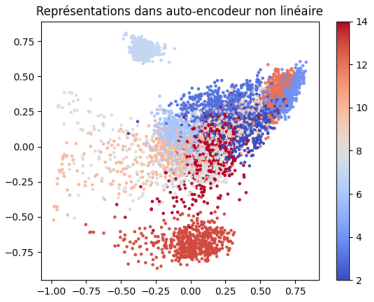
EQM test : 0.0088

PMC linéaire



0.0089

PMC non linéaire 20-2-20



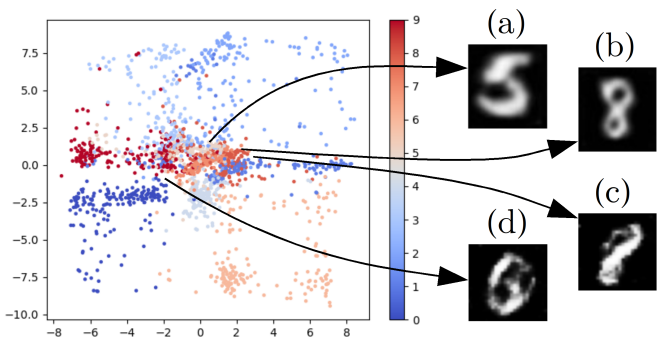
0.0072

→ L'erreur de reconstruction (EQM) est plus faible pour l'auto-encodeur non linéaire

Capacités de génération du décodeur

- Les « codes » développés par un auto-encodeur conservent l'information principale permettant de reconstruire approximativement les entrées correspondantes

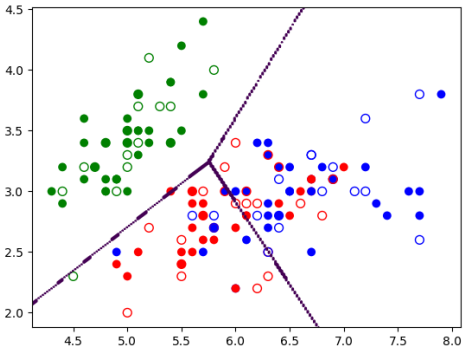
→ Quelles sorties sont obtenues à partir d'autres vecteurs de l'espace des codes, qui ne correspondent à aucune entrée apprise ?



1. Vecteurs très proches de projections d'entrées apprises : en général sorties similaires aux reconstructions de ces entrées (cas a et b)
2. Autres vecteurs de l'espace des codes : sorties intermédiaires (« interpolation »), ou très différentes de toutes les entrées (cas c et d)

Représentations internes développées pour la classification

- La couche de sortie d'un PMC peut être vue comme un réseau de neurones sans couche cachée
  - → Séparations linéaires entre les projections des observations correspondant aux différentes classes sur la (dernière) couche cachée :



  - → Pour qu'une bonne séparation soit possible, dans la dernière couche cachée les classes doivent être linéairement séparables ⇐ les données d'une même classe sont suffisamment bien regroupées entre elles et séparées des autres classes

Classification et analyse discriminante linéaire (AFD)

- Résultats théoriques obtenus dans les conditions suivantes :
  - Problème de classification, codage *one hot* pour les sorties $\mathbf{y}_i$
  - Fonction d'activation identité pour la couche de sortie, donc $\hat{\mathbf{y}}_i = \mathbf{W}_s \mathbf{h}_i$
  - Erreur à minimiser : erreur quadratique $\|\hat{\mathbf{y}} - \mathbf{y}\|^2$
  - On s'intéresse aux représentations $\mathbf{h}_i$ obtenues dans la dernière couche cachée

→ Cas linéaire : une couche cachée de k neurones, activation identité ⇒ $\mathbf{h} = \mathbf{W}_1 \mathbf{x}$

  - L'erreur est minimisée lorsque la fonction de transfert entrée → couche cachée du PMC est la projection orthogonale sur le sous-espace engendré par les vecteurs propres associés aux k plus grandes val. propres de l'AFD des obs. d'apprentissage [10, 3]
  - La représentation des données sur la couche cachée correspond (à une transformation linéaire de rang maximal près) à la projection orthogonale sur ces k vecteurs propres

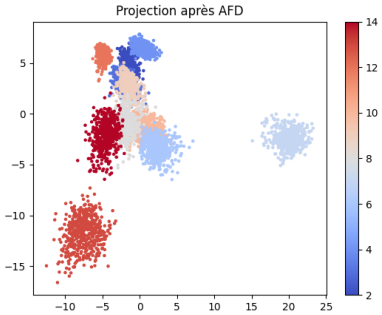
→ Cas général : une ou plusieurs couches cachées, fonctions d'activation non linéaires

  - L'erreur est minimisée lorsque les activations de la dernière couche cachée satisfont $\max \mathrm{Tr}\left(\mathbf{E}_H \mathbf{S}_H^+\right)$ [10, 3]
    - $\mathbf{E}_H$ est la matrice des cov. pondérées interclasse et $\mathbf{S}_H^+$ l'inverse Moore-Penrose de la matrice des cov. totales des vecteurs d'activation de la couche cachée (si $\mathbf{S}_H^{-1}$ existe alors $\mathbf{S}_H^+ = \mathbf{S}_H^{-1}$)

Classification et AFD (2)

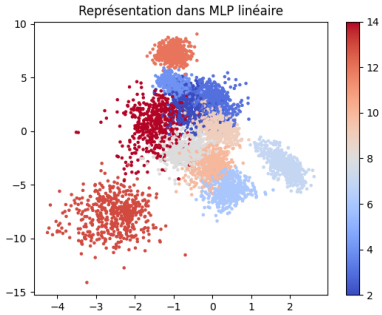
- Comparaison sur les données Textures (11 couleurs différentes ↔ 11 classes) :
  - AFD : projection sur les 2 premiers axes
  - PMC linéaire : 1 couche cachée de 2 neurones, projections sur couche cachée
  - PMC non linéaire : 1 couche cachée de 2 neurones, projections sur couche cachée

AFD



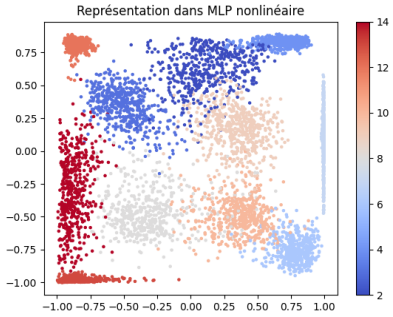
Score test : 0.82

PMC linéaire



0.87

PMC non linéaire



0.93

(score : taux de bon classement, le taux le plus élevé indique les meilleures performances)

Conclusion

- Réseaux de neurones multi-couches : grand potentiel d'approximation de fonctions
- Apprentissage : fonctions non convexes et en général complexes à minimiser
- Descente de gradient : paramétrage difficile, nombreuses tentatives nécessaires dans le cas de problèmes complexes
- Régularisation : indispensable pour une bonne généralisation
- Représentations internes dans les PMC : améliorations non linéaires de celles obtenues par l'analyse factorielle

- La suite :
  - Apprentissage statistique : modélisation décisionnelle et apprentissage profond (RCP209)
  - Intelligence artificielle avancée (RCP211)
  - Intelligence artificielle pour des données multimédia (RCP217)

Références I

[1] P. Baldi and K. Hornik.
Neural networks and principal component analysis : Learning from examples without local minima.
Neural Networks, 2(1) :53–58, 1989.

[2] G. Cybenko.
Approximations by superpositions of sigmoidal functions.
Mathematics of Control, Signals, and Systems, 2 :303–314, 1989.

[3] P. Gallinari, S. Thiria, F. Badran, and F. Fogelman-Soulie.
On the relations between discriminant analysis and multilayer perceptrons.
Neural Networks, 4(3) :349–360, jun 1991.

[4] K. Hornik.
Approximation capabilities of multilayer feedforward networks.
Neural Networks, 4(2) :251–257, Mar. 1991.

Références II

[5] D. P. Kingma and J. Ba.
Adam : A method for stochastic optimization.
In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[6] W. S. McCulloch and W. Pitts.
A logical calculus of the ideas immanent in nervous activity.
Bulletin of Mathematical Biophysics, 5 :115–133, 1943.

[7] A. Pinkus.
Approximation theory of the mlp model in neural networks.
ACTA NUMERICA, 8 :143–195, 1999.

[8] I. Sutskever, J. Martens, G. Dahl, and G. Hinton.
On the importance of initialization and momentum in deep learning.
In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page III–1139–III–1147. JMLR.org, 2013.

Références III

[9] D. Ulmer.

Recoding latent sentence representations – dynamic gradient-based activation modification in RNNs, 01 2021.

[10] A. R. Webb and D. Lowe.

The optimised internal representation of multilayered classifier networks performs nonlinear discriminant analysis.

Neural Networks, 3(4) :367–375, jul 1990.