# [Utilisez des modèles supervisés non linéaires](https://openclassrooms.com/fr/courses/4470406-utilisez-des-modeles-supervises-non-lineaires)

Table des matières

[Utilisez des modèles supervisés non linéaires 1](#_Toc57817430)

[**Transformez un problème non-linéaire en un problème linéaire** 3](#_Toc57817431)

[**En conclusion** 4](#_Toc57817432)

[Classifiez vos données avec une SVM à noyau 6](#_Toc57817433)

[Apprenez des étiquettes réelles avec une régression ridge à noyau 18](#_Toc57817434)

[Partie 1 27](#_Toc57817435)

[Compétences évaluées 27](#_Toc57817436)

[ Question 1 27](#_Toc57817437)

[ Question 2 27](#_Toc57817438)

[ Question 3 28](#_Toc57817439)

[ Question 4 29](#_Toc57817440)

[ Question 5 29](#_Toc57817441)

[ Question 6 32](#_Toc57817442)

[ Question 7 32](#_Toc57817443)

[Entraînez un réseau de neurones simple 34](#_Toc57817444)

[Résumé 39](#_Toc57817445)

[Empilez les perceptrons 40](#_Toc57817446)

[Partie 2 46](#_Toc57817447)

[Compétences évaluées 46](#_Toc57817448)

[ Question 1 46](#_Toc57817449)

[ Question 2 47](#_Toc57817450)

[ Question 3 47](#_Toc57817451)

[ Question 4 48](#_Toc57817452)

[ Question 5 49](#_Toc57817453)

[ Question 6 49](#_Toc57817454)

[ Question 7 50](#_Toc57817455)

Dans le cours [Entraînez un modèle prédictif linéaire](https://openclassrooms.com/courses/entrainez-un-modele-predictif-lineaire), vous avez appris à construire des modèles linéaires de classification binaire ou multi-classe et de régression.

Mais ceux-ci peuvent ne pas être adaptés à la nature de vos données. Dans ce cours, vous apprendrez à **entraîner des modèles supervisés non-linéaires** sur vos données.

Vous comprendrez comment construire un modèle non-linéaire grâce à une **redescription des données**, et saurez utiliser les méthodes à noyaux, qui permettent d’étendre les notions de **SVM** et de**régression ridge** au cas non-linéaire.

Suivez ce cours pour développer des modèles prédictifs non linéaires puissants !

**Objectifs pédagogiques :**

* Utiliser des **noyaux** pour étendre des méthodes linéaires (**SVM**, **régression ridge**) à des cas **non-linéaires** ;
* Comprendre l'intérêt de l'**astuce du noyau**;
* Comprendre le fonctionnement du **perceptron** ;
* Appréhender les **réseaux de neurones multi-couches** et la **backpropagation.**

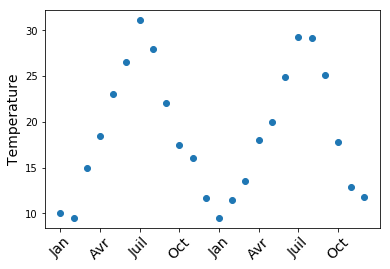
**Prérequis :**

Ce cours fait partie du parcours [**Data Scientist**](https://openclassrooms.com/fr/paths/164-data-scientist). Il se situe au croisement des mathématiques et de l'informatique. Pour en profiter pleinement, n'hésitez pas à vous rafraîchir la mémoire, avant ou pendant le cours, sur :

* **Python** pour le calcul numérique (numpy) et la création de graphiques (pyplot), que nous utiliserons dans les parties TP du cours,
* Quelques notions d'**algèbre linéaire** : manipulation de vecteurs, multiplications de matrices, normes, et valeurs/vecteurs propres,
* Quelques notions de **probabilités et statistiques,** telles que distribution de loi de probabilité et variance,
* Les notions de [**régression linéaire ridge**](https://openclassrooms.com/courses/entrainez-un-modele-predictif-lineaire/reduisez-lamplitude-des-poids-affectes-a-vos-variables)et [**SVM**](https://openclassrooms.com/courses/entrainez-un-modele-predictif-lineaire/maximisez-la-marge-de-separation-entre-vos-classes)**.**

**Transformez un problème non-linéaire en un problème linéaire**

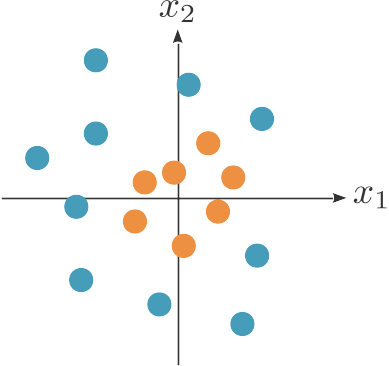
Beaucoup de phénomènes physiques ne sont pas linéaires. Par exemple, la température mensuelle moyenne dans une ville donnée est approximativement périodique en fonction de la saison.



Températures mensuelles (maximales moyennes) sur deux ans pour la ville de Digne-les-Bains.

Il est donc inévitable que, en machine learning aussi, les modèles linéaires ne suffisent pas toujours à bien appréhender nos données. Mais que faire si l'on n'a à sa disposition que des algorithmes linéaires comme la SVM linéaire ou la régression ridge ?

Prenons l'exemple de classification sur l'image ci-dessous : il s'agit de séparer, en deux dimensions, les points bleus des points orange.



Comment séparer les points bleus des points orange quand on ne connait que des méthodes linéaires ?

Il s'agit de séparer les points orange des points bleus, en deux dimensions. Un modèle *linéaire* va donc créer une *droite* comme fonction de décision. Mais on n'aura jamais de bonnes performances avec une droite ! Quelle frontière de décision serait plus appropriée ? Personnellement, je verrais bien un *cercle* centré à l'origine. Si j'appelle **x1** et **x2** mes deux dimensions, un cercle a pour équation , où R est son rayon. J'ai donc 1 unique paramètre à déterminer : le rayon R du cercle. Mais ma boite à outils de modèles linéaires ne permet pas de faire ça...

Comment faire ? En fait, il me suffit de remarquer que si je crée deux nouvelles variables , alors mon équation devient ce qui est une équation... eh oui, *linéaire*! En utilisant une application Φ qui a (x1, x2)' associe (x1², x2²)', j'ai décrit mon problème dans *un nouvel espace*, appelé **espace de redescription**, défini par les coordonnées (Φ(x1), Φ(x2)) et dans lequel mon problème est *linéaire*.

Dans le cas du cercle centré à l'origine, mon espace de redescription fait la même dimension que l'espace dans lequel mes données étaient initialement décrites.

Mais prenons un cas plus général, par exemple celui d'une ellipse (alignée avec les axes). L'ellipse a pour équation  . Pour arriver à la décrire dans un espace linéaire, je vais avoir besoin de *plus* de deux dimensions. Par exemple, je peux prendre . J'utilise maintenant *quatre* dimensions. Ce cas de figure ressemble plus à ce que nous rencontrerons en pratique : l'espace de redescription sera de dimension beaucoup plus grande que l'espace initial.

Par exemple, pour apprendre un polynôme de degré d en p dimensions, il faut utiliser comme nouvelles variables tous les monômes de degrés inférieur ou égal à d, c'est-à-dire tous les polynômes de la forme où .

Étant donné un jeu de données  étiquetées par , je peux maintenant appliquer un algorithme de machine learning linéaire (par exemple, SVM linéaire ou régression ridge)*dans l'espace de redescription* : je vais apprendre une fonction g linéaire sur les données transformées {Φ(x1),Φ(x2),…,Φ(xn)} et leurs étiquettes {y1,y2,…,yn}. Ma fonction de décision sera donc f=g∘Φ : pour faire une prédiction, je commence par calculer l'image Φ(x) de x dans l'espace de redescription, puis je lui applique g. La fonction g est linéaire en Φ(x), mais la fonction f n'est pas linéaire.

Ce processus est bien évidemment valable pour un problème de régression comme de classification.

**En conclusion**

Ainsi, une façon de construire une fonction de décision non-linéaire est de redécrire les données dans un nouvel espace dans lequel on apprend une fonction de décision linéaire. Deux questions se posent maintenant :

* comment *choisir* l'application Φ et l'espace de redescription ?
* comment apprendre *de manière efficace* dans un espace de redescription de dimension potentiellement très grande ?

C'est ce que nous verrons dans le prochain chapitre.

## Classifiez vos données avec une SVM à noyau

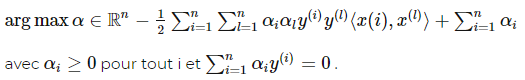
#### SVM à noyau

Prenons un jeu de données  étiquetées par , et une application   qui permet de redécrire ces données dans l'espace de redescription H.

L'espace de redescription H est ainsi nommé car nous allons choisir des applications Φ tel que H soit un espace de Hilbert. En pratique, cela veut dire que H peut être muni d'un produit scalaire, et vous pouvez considérer que H est égal à  (ou  mais en pratique tous nos exemples utiliseront des espaces réels et non pas complexes.)

Plutôt que d'utiliser une [SVM linéaire](https://openclassrooms.com/courses/entrainez-un-modele-predictif-lineaire/maximisez-la-marge-de-separation-entre-vos-classes) pour apprendre un modèle linéaire qui explique y comme une fonction linéaire des coordonnées de x, nous pouvons utiliser ce même algorithme de SVM linéaire dans H pour apprendre un modèle qui explique y comme une fonction linéaire des coordonnées de Φ(x). Nous créons ainsi un modèle non-linéaire dans l'espace initial.

Regardons de plus près ce que cela veut dire, et reprenons pour cela l'expression duale de l'apprentissage du SVM :



Comme nous apprenons notre SVM non pas dans l'espace initial dans lequel les données sont décrites, mais dans l'espace de redescription H, nous remplaçons donc x par  Φ(x) dans le problème d'optimisation en question :



Nous pouvons faire la même manipulation sur la fonction de décision : la prédiction se fait en fonction du signe de



##### Noyau

Appelons maintenant k la fonction qui à  associe . Je peux réécrire et le problème d'optimisation et la fonction de décision en utilisant uniquement k et non plus Φ :



et



La fonction k s'appelle un **noyau** (kernel en anglais, d'où la notation k), et a deux propriétés mathématiques intéressantes :

* elle est définie positive, autrement dit, étant donnés un nombre arbitraires de points , la matrice K de dimensions m x m telle que  est [définie positive](https://www.wikiwand.com/fr/Matrice_d%C3%A9finie_positive).
* quelle que soit la fonction , si h est définie positive, alors il existe un espace de Hilbert H et une application telle que .

##### Astuce du noyau

Le tour de passe-passe mathématique où j'ai remplacé les produits scalaires faisant intervenir Φ par k peut paraître anodin, mais il est en fait très intéressant : il va nous permettre d'apprendre une SVM non-linéaire et de l'utiliser pour faire des prédictions sans calculer explicitement Φ, en utilisant directement k sur l'espace initial. C'est ce que l'on appelle **l'astuce du noyau**, ou kernel trick en anglais.

D'accord, mais quel est l'intérêt de cette astuce ?

Cette astuce est super intéressante parce que l'espace de redescription H peut être de très très grande dimension, et de ne pas avoir à calculer l'image de chaque point par Φ va nous sauver énormément de temps, voire nous permettre d'utiliser des espaces de redescription que nous ne savons pas définir explicitement !

##### Exemples de noyau

Considérons par exemple un espace H qui permette de décrire une fonction de décision qui soit un polynôme de degré d. La fonction de redescription associe à x l'ensemble de tous les monômes de degré d, autrement dit les polynômes de la forme , où . H a pour dimension . Mais prenons maintenant deux points x et x', et calculons le produit scalaire de leurs images : il s'agit en fait tout simplement de . C'est d'ailleurs ce qu'on appelle un **noyau polynomial.**

Plus fou encore, le **noyau gaussien :**



Il correspond à un espace de redescription de dimension infinie ! Autrement dit, il nous permet de résoudre une SVM dans un espace de redescription qu'on ne peut même pas calculer explicitement... C'est fort !

Le fait que n'importe quelle fonction définie positive soit un noyau donne au SVM non-linéaires un grand pouvoir de modélisation : un noyau, comme un produit scalaire, peut en effet être interprété comme une mesure de similarité (deux vecteurs colinéaires ont un produit scalaire élevé, deux vecteurs orthogonaux ont un produit scalaire nul). Cela signifie que l'on peut créer des noyaux appropriés pour un domaine d'application particulier, et qu'il « suffit » pour cela de construire une mesure de similarité entre observations qui soit définie positive.

Cela a par exemple beaucoup été fait pour l'analyse de séquences biologiques. Les séquences en question sont des chaînes de caractères, généralement d'acides aminés (brique de base des protéines), autrement dit sur un alphabet de 23 caractères. De nombreux noyaux pour séquences ([**string kernels**](https://www.wikiwand.com/en/String_kernel)) ont ainsi été construits en se basant sur l'idée que deux chaînes de caractères sont d'autant plus similaires qu'elles ont de sous-chaînes en commun. Par exemple, 'abbatiale' et 'abbaye' qui ont en commun les chaines de 3 caractères 'abb' et 'bba' sont plus proches l'une de l'autre que 'abbaye' et 'abbé' ('abb'), mais moins que 'abbatiale' et 'spatiale' ('ati', 'ial', 'ale'). Remarquez que cela permet de manipuler des objets de tailles différentes...

L'astuce du noyau s'applique aussi à la version « régression » des SVMs, à savoir, la SVR.

#### En pratique avec scikit-learn

Les SVM à noyaux sont implémentées dans scikit-learn dans les classes [sklearn.svm.SVC](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html)pour la classification et [sklearn.svm.SVR](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html) pour la régression. Dans ces deux classes, vous pouvez spécifier un noyau grâce au paramètre « kernel ». Ce noyau peut être un des grands classiques (linéaire, polynomial, RBF), mais vous pouvez aussi définir vos propres noyaux !

Regardons comment utiliser la classe sklearn.svm.SVC en pratique. Nous allons utiliser les données concernant les caractéristiques physico-chimiques de vins blancs portugais disponibles [sur l'archive UCI](https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality). Il s'agit ici de prédire le score (entre 3 et 9) donné par des experts aux différents vins. Chargeons les données et transformons le problème en un problème de classification, pour lequel il s'agira de prédire si le score est supérieur à 6 (vin de bonne qualité) ou non :

import numpy as np

# charger les données

import pandas as pd

data = pd.read\_csv('winequality-white.csv', sep=';')

# créer la matrice de données

X = data[data.columns[:-1]].values

# créer le vecteur d'étiquettes

y = data['quality'].values

# transformer en un problème de classification binaire

y\_class = np.where(y<6, 0, 1)

Avant toute chose, nous allons découper nos données en un jeu d'entraînement (X\_train, y\_train) et un jeu de test (X\_test, y\_test).

from sklearn import model\_selection

X\_train, X\_test, y\_train, y\_test = \

model\_selection.train\_test\_split(X, y\_class, test\_size=0.3)

Nous pouvons maintenant standardiser les variables, c'est-à-dire les centrer (ramener leur moyenne à 0) et les réduire (ramener leur écart-type à 1), afin qu'elles se placent toutes à peu près sur la même échelle.

# standardiser les données

from sklearn import preprocessing

std\_scale = preprocessing.StandardScaler().fit(X\_train)

X\_train\_std = std\_scale.transform(X\_train)

X\_test\_std = std\_scale.transform(X\_test)

OK, nous pouvons enfin entraîner notre première SVM à noyau !

# Créer une SVM avec un noyau gaussien de paramètre gamma=0.01

from sklearn import svm

classifier = svm.SVC(kernel='rbf', gamma=0.01)

# Entraîner la SVM sur le jeu d'entraînement

classifier.fit(X\_train\_std, y\_train)

Comment se comporte-t-elle sur le jeu de test ? Nous allons pour le comprendre regarder la courbe ROC.

# prédire sur le jeu de test

y\_test\_pred = classifier.decision\_function(X\_test\_std)

# construire la courbe ROC

from sklearn import metrics

fpr, tpr, thr = metrics.roc\_curve(y\_test, y\_test\_pred)

# calculer l'aire sous la courbe ROC

auc = metrics.auc(fpr, tpr)

# créer une figure

from matplotlib import pyplot as plt

fig = plt.figure(figsize=(6, 6))

# afficher la courbe ROC

plt.plot(fpr, tpr, '-', lw=2, label='gamma=0.01, AUC=%.2f' % auc)

# donner un titre aux axes et au graphique

plt.xlabel('False Positive Rate', fontsize=16)

plt.ylabel('True Positive Rate', fontsize=16)

plt.title('SVM ROC Curve', fontsize=16)

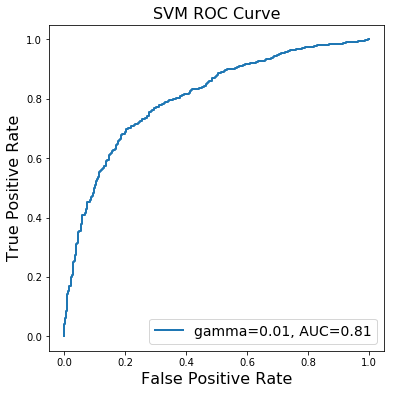
# afficher la légende

plt.legend(loc="lower right", fontsize=14)

# afficher l'image

plt.show()

Et voilà notre courbe ROC, avec une AUC de 0.81, plutôt pas mal !



Courbe ROC d'une SVM avec noyau gaussien sur les données winequality-white.

#### Sélection des hyperparamètres

Comme dans le cas de la SVM linéaire, il faut mettre en œuvre de bonnes pratiques de sélection de modèle (par exemple une validation croisée) pour choisir la meilleure valeur de C !

Nous allons ici utiliser une validation croisée sur le jeu d'entraînement pour sélectionner les valeurs optimales de C et de gamma parmi une grille de valeurs.

# choisir 6 valeurs pour C, entre 1e-2 et 1e3

C\_range = np.logspace(-2, 3, 6)

# choisir 4 valeurs pour gamma, entre 1e-2 et 10

gamma\_range = np.logspace(-2, 1, 4)

# grille de paramètres

param\_grid = {'C': C\_range, 'gamma': gamma\_range}

# critère de sélection du meilleur modèle

score = 'roc\_auc'

# initialiser une recherche sur grille

grid = model\_selection.GridSearchCV(svm.SVC(kernel='rbf'),

param\_grid,

cv=5, # 5 folds de validation croisée

scoring=score)

# faire tourner la recherche sur grille

grid.fit(X\_train\_std, y\_train)

# afficher les paramètres optimaux

print("The optimal parameters are {} with a score of {:.2f}".format(grid.best\_params\_, grid.best\_score\_))

The optimal parameters are {'C': 1.0, 'gamma': 1.0} with a score of 0.85

Nous pouvons maintenant évaluer la performance de notre modèle optimisé sur le jeu de test :

# prédire sur le jeu de test avec le modèle optimisé

y\_test\_pred\_cv = grid.decision\_function(X\_test\_std)

# construire la courbe ROC du modèle optimisé

fpr\_cv, tpr\_cv, thr\_cv = metrics.roc\_curve(y\_test, y\_test\_pred\_cv)

# calculer l'aire sous la courbe ROC du modèle optimisé

auc\_cv = metrics.auc(fpr\_cv, tpr\_cv)

# créer une figure

fig = plt.figure(figsize=(6, 6))

# afficher la courbe ROC précédente

plt.plot(fpr, tpr, '-', lw=2, label='gamma=0.01, AUC=%.2f' % auc)

# afficher la courbe ROC du modèle optimisé

plt.plot(fpr\_cv, tpr\_cv, '-', lw=2, label='gamma=%.1e, AUC=%.2f' % \

(grid.best\_params\_['gamma'], auc\_cv))

# donner un titre aux axes et au graphique

plt.xlabel('False Positive Rate', fontsize=16)

plt.ylabel('True Positive Rate', fontsize=16)

plt.title('SVM ROC Curve', fontsize=16)

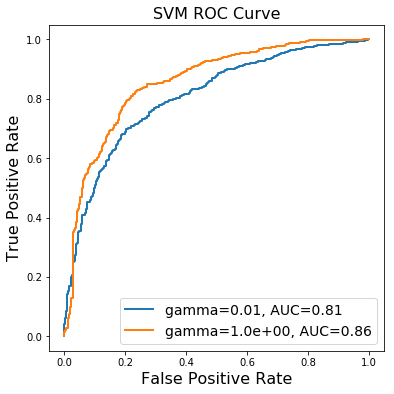
# afficher la légende

plt.legend(loc="lower right", fontsize=14)

# afficher l'image

plt.show()

On obtient alors la courbe ROC suivante, avec une performance optimisée :



Courbe ROC d'une SVM avec noyau gaussien optimisée, sur les données winequality-white

##### Matrice de **Gram**

Quel est l'importance du paramètre gamma ? La documentation nous indique que le noyau RBF gaussien est donné par .

Gamma contrôle la bande passante de la gaussienne : plus gamma est élevé, plus la gaussienne est étroite, autrement dit, plus la distance entre x et x' doit être faible pour que le noyau soit différent de 0.

Pour en visualiser l'effet, nous pouvons visualiser la **matrice de Gram** des données, autrement dit la matrice K telle que .

Il peut arriver que la **matrice de Gram** de vos données, autrement dit la matrice K telle que , est **à diagonale fortement dominante**, autrement dit que les nombres sur sa diagonale ont des valeurs plusieurs ordres de grandeurs plus grandes que les autres. Dans ce cas, du point de vue numérique elle ressemblera à la matrice identité (fois un grand nombre réel), et la SVM ne pourra pas bien apprendre : c'est comme si vous lui disiez que chaque point n'est semblable qu'à lui-même, et extrêmement différent des autres.

Calculons la matrice de Gram obtenue sur notre jeu d'entraînement quand gamma=0.01 :

from sklearn import metrics

kmatrix = metrics.pairwise.rbf\_kernel(X\_train\_std, gamma=0.01)

Nous allons réduire cette matrice à ses 100 premières lignes et 100 premières colonnes pour en faciliter la visualisation :

kmatrix100 = kmatrix[:100, :100]

Nous pouvons maintenant utiliser [plt.pcolor](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.pcolor.html?highlight=matplotlib%20pyplot%20pcolor#matplotlib.pyplot.pcolor) pour visualiser cette matrice :

# dessiner la matrice

plt.pcolor(kmatrix100, cmap=matplotlib.cm.PuRd)

# rajouter la légende

plt.colorbar()

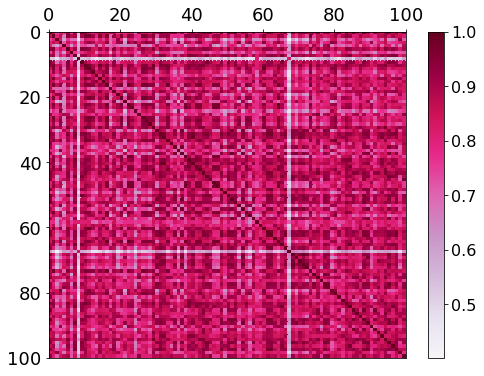
# retourner l'axe des ordonnées

plt.gca().invert\_yaxis()

plt.gca().xaxis.tick\_top()

# afficher l'image

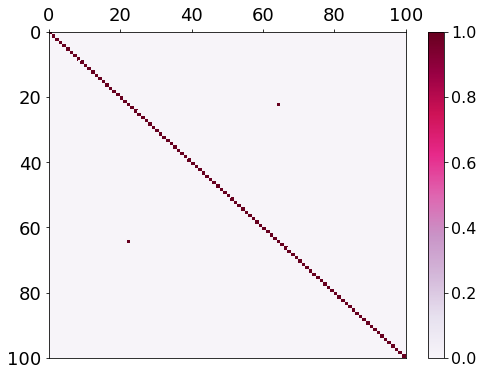
plt.show()



Matrice de Gram pour gamma=0.01

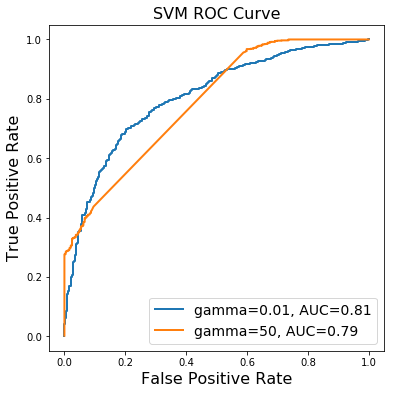
Nous avons ici des valeurs de noyau comprises entre 0.40 et 1.0, avec une diagonale plus forte mais qui n'écrase pas la matrice.

Avec une valeur de gamma plus grande, par exemple, gamma=50, on obtient la matrice de Gram suivante :



Matrice de Gram pour gamma=50

Cette matrice est à diagonale dominante. Si l'on entraîne une SVM avec cette valeur de gamma, la courbe ROC est beaucoup moins bonne :



Courbe ROC d'une SVM avec noyau gaussien, pour plusieurs valeurs de gamma, sur les données winequality-white.

Que faire si la matrice de Gram est dominée par sa diagonale ? Deux choix sont possibles :

* jouer avec les paramètres du noyau pour réduire la différence entre les valeurs diagonales et les valeurs hors diagonale
* remplacer la matrice de Gram K par la matrice M telle que . Les valeurs diagonales de cette matrice seront de 1.

Il est aussi possible que les paramètres du noyau soient tels que toutes les valeurs de la matrice de Gram sont proches les unes des autres. Dans ce cas, la SVM ne peut pas bien apprendre non plus : c'est comme si on lui avait dit que tous les points sont quasi-identiques...

## Apprenez des étiquettes réelles avec une régression ridge à noyau

Il n'y a pas qu'aux SVM que l'on peut appliquer l'astuce du noyau ! Elle marche avec un certain nombre d'algorithmes linéaires, par exemple, la [régression ridge régularisée](https://openclassrooms.com/courses/entrainez-un-modele-predictif-lineaire/reduisez-lamplitude-des-poids-affectes-a-vos-variables).

Prenons n points en p dimensions, décrits par la matrice X (de taille n x p), et leurs étiquettes, décrites par le vecteur y (de taille n).

Pour une nouvelle observation x, la prédiction de la régression ridge est :

**f(x)=xβ**

où β est un vecteur colonne de taille p, donné par . Je note **Ip** la matrice identité de taille p x p.

Ainsi donc



Cette équation peut se réécrire sous la forme



Cela nécessite quelques manipulations mathématiques, que vous trouverez ci-dessous mais qui ne sont pas essentielles à la compréhension, si vous admettez cette réécriture...

En effet, en multipliant à gauche l'expression qui définit β par , on obtient :  et donc  où .

En remplaçant β par dans la définition de α, on obtient  et donc .

Revenons maintenant à f :  et nous avons donc bien notre équation



Supposons maintenant une application ϕ vers un espace de redescription H. Appliquons maintenant la régression ridge dans H :



où Φ est la matrice de taille n x d (d étant la dimension de H) dont la i-ième ligne est l'image du i-ème point du jeu de données dans H, autrement dit le vecteur .

À cette application ϕ correspond un noyau, défini par .

Il suffit maintenant de remplacer le vecteur  par le vecteur κ et la matrice  par la matrice de Gram K, où  et K, comme pour les SVMs est la matrice n x n telle que .

On peut donc écrire



et ici aussi, comme dans le cas des SVMs, entraîner et utiliser une régression ridge dans l'espace de redescription sans avoir à connaître ϕ ni à calculer explicitement l'image d'aucun point par cette application !

Cette méthode, que l'on appelle donc, vous l'aurez deviné, la **régression ridge à noyau**, s'appelle kernel ridge regression en anglais (ou kRR).

On peut de la même façon appliquer l'astuce du noyau à la régression logistique régularisée ridge.

Oui mais attends... qu'est-il arrivé aux coefficients β ?

L'astuce du noyau ne nous permet pas de déterminer β explicitement. D'ailleurs souvenez-vous, dans le cas du noyau RBF gaussien, l'espace de redescription est de dimension infinie, et donc \beta aussi... Ça n'aurait pas grand sens d'essayer de le calculer !

Contrairement à sa version linéaire, la version à noyau de la régression ridge ne nous donne pas une forme explicite de la fonction de décision f en fonction des variables. C'est ce qu'on appelle une approche **non-paramétrique** (eh oui, malgré les paramètres du noyau), comme pour la SVM à noyau.

Il existe bien d'autres approches non-paramétriques, en particulier la méthode des [plus proches voisins](https://openclassrooms.com/courses/initiez-vous-au-machine-learning/tp-entrainez-le-modele-des-k-plus-proches-voisins-k-nn) ou [les arbres de décisions et les forêts aléatoires](https://openclassrooms.com/courses/modelisez-vos-donnees-avec-les-methodes-ensemblistes/reduisez-la-correlation-entre-les-apprenants-faibles-a-laide-des-forets-aleatoires).

La complexité des approches non-paramétriques grandit avec le nombre d'observations (plus il est grand, plus la matrice de Gram K est grande, plus il est compliqué de calculer les k plus proches voisins, plus un arbre de décision peut être profond, etc.), tandis que celle des approches paramétriques (comme la régression linéaire) grandit avec le nombre de variables !

En quoi la kRR diffère-t-elle vraiment de la SVR ?

Les modèles appris par la régression ridge et une SVR ont exactement la même forme, mais pas les mêmes coefficients. En effet, on a choisi d'optimiser des fonctions de perte différentes pour l'une et pour l'autre. Dans le cas de la régression ridge, on choisit de minimiser l'erreur quadratique entre la prédiction et la réalité. Pour la SVR, on utilise en fait une fonction de perte dite insensible à \epsilon, c'est à dire qu'elle vaut la valeur absolue de la différence entre la prédiction et la réalité, sauf si cette différence est faible, comprise entre −ϵ et ϵ , auquel cas on l'estime négligeable et la fonction de perte vaut 0.

En pratique, entraîner une kRR sera plus efficace car la solution est analytique est exacte ; prédire sera par contre plus rapide avec une SVR.

#### En pratique avec scikit-learn

La régression ridge à noyau est implémentée dans scikit-learn dans la classe [kernel\_ridge.KernelRidge](http://scikit-learn.org/stable/modules/generated/sklearn.kernel_ridge.KernelRidge).

Nous allons utiliser les données concernant les caractéristiques physico-chimiques de vins blancs portugais disponibles [sur l'archive UCI](https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality). Il s'agit ici de prédire le score (entre 3 et 9) donné par des experts aux différents vins.

Chargeons les données, séparons-les en un jeu d'entraînement et un jeu de test contenant respectivement 70% et 30% des données, et standardisons les variables sur le jeu d'entraînement.

# charger les données

import pandas as pd

data = pd.read\_csv('winequality-white.csv', sep=';')

# créer la matrice de données

X = data[data.columns[:-1]].values

# créer le vecteur d'étiquettes

y = data['quality'].values

# créer un jeu d'entrainement et un jeu de test (30% des données)

from sklearn import model\_selection

X\_train, X\_test, y\_train, y\_test = \

model\_selection.train\_test\_split(X, y, test\_size=0.3)

# standardiser les données

from sklearn import preprocessing

std\_scale = preprocessing.StandardScaler().fit(X\_train)

X\_train\_std = std\_scale.transform(X\_train)

X\_test\_std = std\_scale.transform(X\_test)

Nous pouvons maintenant entraîner une kRR sur le jeu d'entraînement en utilisant des paramètres par défaut pour le paramètre de régularisation et la bande passante du noyau RBF gaussien :

from sklearn import kernel\_ridge

predicteur = kernel\_ridge.KernelRidge(

alpha=1.0, # valeur par défaut

kernel='rbf', # noyau Gaussien

gamma=0.01) # valeur de 1/(2 \* sigma\*\*2)

# entraîner le classifieur sur le jeu d'entrainement

predicteur.fit(X\_train\_std, y\_train)

# prédire sur le jeu de test

y\_test\_pred = predicteur.predict(X\_test\_std)

# calculer la RMSE sur le jeu de test

from sklearn import metrics

rmse = np.sqrt(metrics.mean\_squared\_error(y\_test, y\_test\_pred))

print("RMSE: {:.2f}".format(rmse))

On obtient une RMSE de 0.72, ce qui est correct pour des nombres entiers allant de 3 à 9.

Pour visualiser les prédictions, on peut utiliser un nuage de points dans lequel la surface de chaque point est proportionnelle au nombre d'observations ayant exactement ces valeurs de score prédit et de score réel. Pour plus de détails, vous pouvez vous référer à la procédure utilisée dans le cours [Comparez votre algorithme à des approches de régression naïves](https://openclassrooms.com/courses/evaluez-et-ameliorez-les-performances-d-un-modele-de-machine-learning/comparez-votre-algorithme-a-des-approches-de-regression-naives).

# créer une figure

fig = plt.figure(figsize=(6, 6))

# Compter, pour chaque paire de valeurs (y, y') où y est un vrai score et y' le score prédit,

# le nombre de ces paires.

# Ce nombre sera utilisé pour modifier la taille des marqueurs correspondants

# dans un nuage de points

sizes = {}

for (yt, yp) in zip(list(y\_test), list(y\_test\_pred)):

if (yt, yp) in sizes.keys():

sizes[(yt, yp)] += 1

else:

sizes[(yt, yp)] = 1

keys = sizes.keys()

# afficher les prédictions

plt.scatter([k[0] for k in keys],

[k[1] for k in keys],

s=[sizes[k] for k in keys],

label="gamma = 0.01: RMSE = {:.2f}".format(rmse))

# étiqueter les axes et le graphique

plt.xlabel('Vrai score', fontsize=16)

plt.ylabel(u'Score prédit', fontsize=16)

plt.title('kernel Ridge Regression', fontsize=16)

# limites des axes

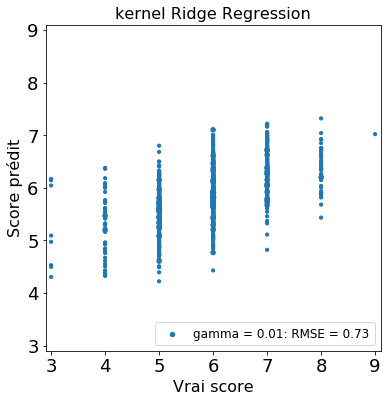
plt.xlim([2.9, 9.1])

plt.ylim([2.9, 9.1])

# afficher la légende

plt.legend(loc="lower right", fontsize=12)

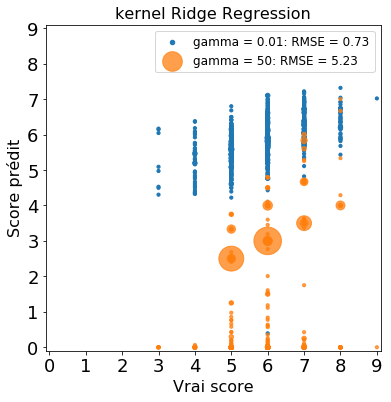
On obtient le graphique suivant :



Score du vin prédit vs. vrai score, sur le jeu de test, pour une kRR.

On observe une corrélation entre les scores prédits et les scores réels.

Comme dans le cas de la SVM à noyau, le paramètre gamma du noyau RBF gaussien joue un rôle important : s'il est trop élevé, la matrice de Gram sur le jeu d'entraînement est dominée par sa diagonale et la kRR ne peut pas apprendre. Si l'on recommence avec gamma=50, on obtient les prédictions suivantes :



Score du vin prédit vs. vrai score, sur le jeu de test, pour une kRR avec différentes valeurs de gamma.

Les prédictions avec gamma=50 sont de très mauvaise qualité!

 Comme dans le cas de la régression ridge linéaire, il faut mettre en œuvre de bonnes pratiques de sélection de modèle (par exemple une validation croisée) pour choisir la meilleure valeur du paramètre de régularisation. Mais il y a maintenant des hyperparamètres supplémentaires : les paramètres du noyau !

Pour optimiser le paramètre de régularisation alpha et le paramètre gamma du noyau gaussien, on peut utiliser une recherche sur grille :

# valeurs du paramètre C

alpha\_range = np.logspace(-2, 2, 5)

# valeurs du paramètre gamma

gamma\_range = np.logspace(-2, 1, 4)

# grille de paramètres

param\_grid = {'alpha': alpha\_range, 'gamma': gamma\_range}

# score pour sélectionner le modèle optimal

score = 'neg\_mean\_squared\_error'

# initialiser la validation croisée

grid\_pred = model\_selection.GridSearchCV(

kernel\_ridge.KernelRidge(kernel='rbf'),

param\_grid,

cv=5,

scoring=score)

# exécuter la validation croisée sur le jeu d'entraînement

grid\_pred.fit(X\_train\_std, y\_train)

# prédire sur le jeu de test avec le modèle sélectionné

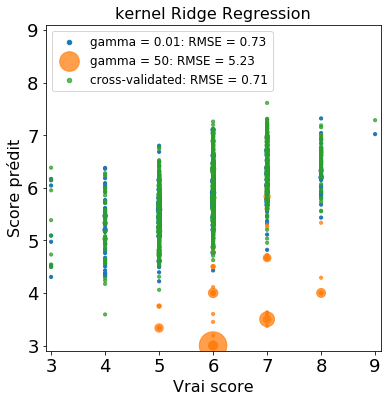
y\_test\_pred\_cv = grid\_pred.predict(X\_test\_std)

# calculer la RMSE correspondante

rmse\_cv = np.sqrt(metrics.mean\_squared\_error(y\_test, y\_test\_pred\_cv))

On obtient une RMSE légèrement meilleure (0.71) en optimisant le modèle.

Les prédictions faites se superposent à peu près à celles obtenues avec gamma=0.01 :



Score du vin prédit vs. vrai score, sur le jeu de test, pour plusieurs kRR.

# Partie 1

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Utiliser des noyaux pour étendre des méthodes linéaires à des cas non linéaires
* Comprendre l'intérêt de l'astuce du noyau

### Question 1

**J'aimerais apprendre une fonction de décision qui soit un polynôme de degré 3. Quelles propositions sont vraies parmi les propositions suivantes ?**

*Attention, plusieurs réponses sont possibles.*

* + 

Je peux utiliser une SVM sur mes données telles quelles.

* + 

Je peux utiliser une régression ridge à noyau sur mes données telles quelles.

* + 

Je peux utiliser une SVM, mais il faut d'abord que je transforme mes données vers un espace avec autant de dimensions que de monômes de degré 3.

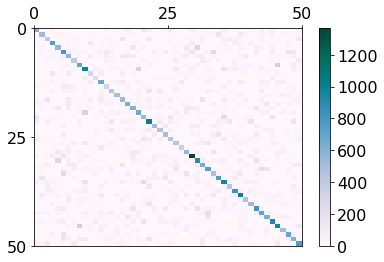
* + 

Je peux utiliser une régression ridge à noyau, mais il faut d'abord que je transforme mes données vers un espace avec autant de dimensions que de monômes de degré 3.

*L'astuce du noyau permet d'appliquer les algorithmes à noyau (SVM, régression ridge) directement dans l'espace initial. Il n'est pas nécessaire de transformer les données.*

### Question 2

**J'utilise un noyau K sur mes données et la matrice de Gram ressemble à :**

Matrice de Gram

* + 

Mes données contiennent 50 variables et 50 échantillons.

* + 

Mes données contiennent 50 variables et je ne sais pas combien d'échantillons.

* + 

Mes données contiennent 50 échantillons et je ne sais pas combien de variables.

* + 

Mes données contiennent 50 échantillons et 1200 variables.

*La matrice de Gram est la matrice K telle que**Elle a donc comme taille n x n où n est le nombre d'échantillons. L'échelle à droite indique l'intensité de chaque point et donc les valeurs prises par k.*

### Question 3

**Sur les mêmes données qu'à la question précédente, que pensez-vous qu'il faille faire avant d'entraîner une SVM:**

* + 

Rien, on a tout ce dont on a besoin pour entraîner une SVM.

* + 

Normaliser la matrice de Gram en divisant chaque entrée par .

* + 

Normaliser les données en divisant chaque entrée  

* + 

Normaliser les données pour les centrer sur 0 et leur donner un écart-type de 1.

*On observe que la diagonale domine (la matrice ressemble à 1200 x la matrice identité). Il faudrait pour bien apprendre soit changer les paramètres du noyau pour que cela soit moins marqué, soit corriger la diagonale en remplaçant .*

### Question 4

**Mes données concernent la fréquentation des boulangeries dans une ville. Cependant, plutôt que d'avoir les coordonnées de chacune des boulangeries, elles sont représentées par leur distance (mesurée en mètres de marche à pied) à chacune des autres boulangeries. Pour entraîner un modèle prédictif, il me faut :**

* + 

Je dois commencer par utiliser un algorithme comme IsoMap pour associer des coordonnées géographiques à mes points. Je pourrai ensuite utiliser n'importe quel algorithme supervisé sur cette description en 2D.

* + 

Je peux entraîner une SVM avec un noyau gaussien.

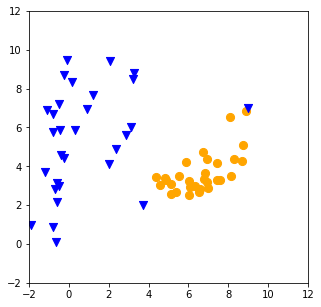
* + 

Je peux entraîner n'importe quel algorithme supervisé directement sur cette description des données.

*Chaque boulangerie x est représentée par . C'est tout ce dont j'ai besoin pour calculer le noyau Gaussien**!*

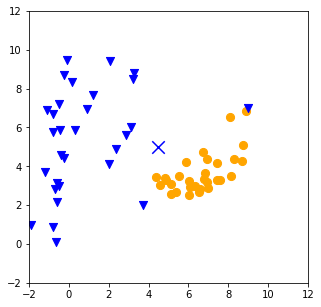
### Question 5

**Voici un problème de classification en 2 dimensions. Dans quel scénario ci-dessous est-on sûrs que la frontière de décision d'une SVM avec noyau quadratique (= polynomial de degré 2) n'est pas affectée ?**



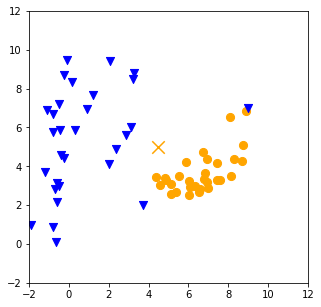
Classification : séparer les points bleus des points orange.

* + 



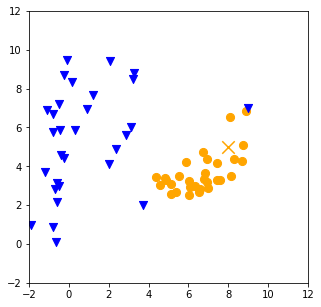
On rajoute l'observation représentée par la croix bleue au jeu de points bleus.

* + 



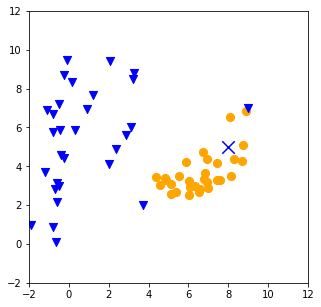
On rajoute l'observation représentée par la croix orange aux données orange.

* + 



On rajoute l'observation représentée par la croix orange aux données orange.

* + 



On ajoute le point représenté par la croix bleue aux données bleues.

*Seuls les points près de la frontière de décision (et, selon la valeur de C, les données aberrantes) peuvent être vecteurs de support et affecter celle-ci.*

### Question 6

**L'intérêt d'utiliser les méthodes à noyau est de pouvoir utiliser des espaces de redescriptions à grande dimensions sans avoir besoin de l'expliciter**

* + 

moins de dimensions dans l'espace de redescription que dans l'espace initial

* + 

autant de dimensions dans l'espace de redescription que dans l'espace initial

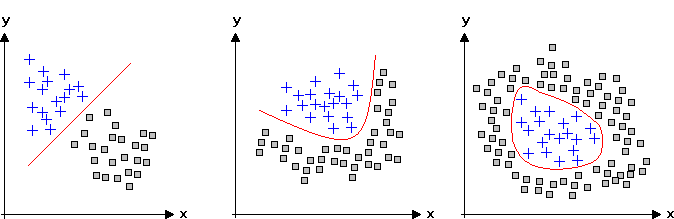
* + 

plus de dimensions dans l'espace de redescription que dans l'espace initial

*Lorsque on redécrit un problème, c'est en général pour capter plus d'informations non-linéaires sur les données que d'informations que l'on possède*

### Question 7

**Lesquels de ces exemples representent un modèle sur les données (x,y) non-linéaires?**

****

*Attention, plusieurs réponses sont possibles.*

* + 

Le premier

* + 

Le second

* + 

Le troisième

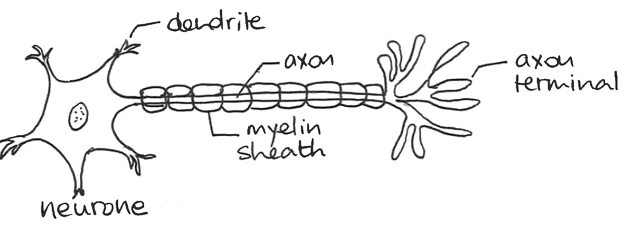
*La linéarité est entre les paramètres à estimer, pas entre les variables indépendantes observées. C'est pourquoi les deux premières figures sont considérés comme des modèles linéaires.*

## Entraînez un réseau de neurones simple

#### Neurones biologiques ou artificiels ?

Les réseaux de neurones, voilà un domaine du machine learning dont on entend beaucoup parler en ce moment... De la reconnaissance vocale à la recherche d'images, en passant par les voitures autonomes et AlphaGo, les récents succès de l'intelligence artificielle sont nombreux à se baser sur les réseaux de neurones profonds, plus connus sous le nom mystérieux de deep learning.

Mais l'histoire des réseaux de neurones artificiels remonte aux années 1950 et aux efforts de psychologues comme Franck Rosenblatt pour comprendre le cerveau humain. Initialement, ils ont été conçus dans le but de modéliser mathématiquement le traitement de l'information par les réseaux de neurones biologiques qui se trouvent dans le cortex des mammifères. De nos jours, leur réalisme biologique importe peu et c'est leur efficacité à modéliser des relations complexes et non linéaires qui fait leur succès.



Un neurone (pas artificiel).

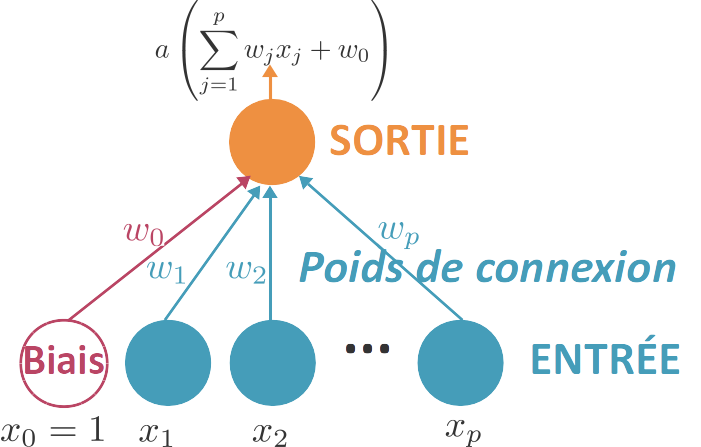
Dans leur principe, les réseaux de neurones ne sont rien d'autre qu'une façon de construire des modèles paramétriques, c'est-à-dire pour lesquels la fonction de décision est explicite. Contrairement à d'autres algorithmes paramétriques comme la régression linéaire, ils permettent de construire facilement des modèles très complexes et non linéaires.

#### Architecture d'un perceptron

Commençons cependant par le commencement et le **perceptron**, ce réseau de neurone à une seule couche inventée par Rosenblatt dont je parlais plus haut.

Le perceptron est formé d'une première **couche** d'**unités**(ou**neurones**) qui permettent de « lire » les données : chaque unité correspond à une des variables d'entrée. On peut rajouter une **unité de biais** qui est toujours **activée** (elle transmet 1 quelles que soient les données). Ces unités sont reliées à une seule et unique **unité de sortie**, qui reçoit la somme des unités qui lui sont reliées, pondérée par des **poids de connexion**. Pour p variables , la sortie reçoit donc . L'unité de sortie applique alors une **fonction d'activation** a à cette sortie.

Un perceptron prédit donc grâce à une fonction de décision f définie par . Cette fonction a une forme explicite, il s'agit bien d'un modèle paramétrique.

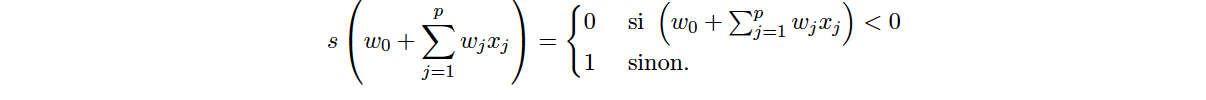


Perceptron

Quelle fonction d'activation utiliser ?

Dans le cas d'un problème de **régression**, il n'est pas nécessaire de transformer la somme pondérée reçue en entrée. La fonction d'activation est la fonction identité, elle retourne ce qu'elle a reçu en entier.

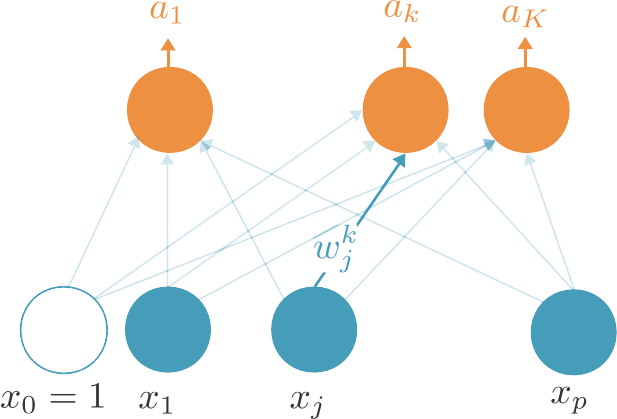
Dans le cas d'un problème de **classification binaire**, on peut utiliser une fonction de **seuil** :



Comme dans le cas de la régression logistique, on peut aussi utiliser une fonction **sigmoïde**pour prédire la probabilité d'appartenir à la classe positive :

.

Dans le cas d'un problème de **classification multi-classe,**nous allons modifier l'architecture du perceptron. Au lieu d'utiliser une seule unité de sortie, il va en utiliser autant que de classes. Chacune de ces unités sera connectée à toutes les unités d'entrée. On aura donc ainsi K.(p+1) poids de connexion, où K est le nombre de classes.



Perceptron multi-classe.

On peut alors utiliser comme fonction d'activation la fonction **softmax.** Il s'agit d'une généralisation de la sigmoïde, qui peut aussi s'écrire  : nous allons utiliser  . Si la sortie pour la classe k est suffisamment plus grande que celles des autres classes, son activation sera proche de 1 tandis que l'activation des autres sera proche de 0. On peut donc aussi considérer qu'il s'agit d'une version différentiable du maximum, ce qui nous aidera grandement pour l'apprentissage.

#### Apprentissage d'un perceptron

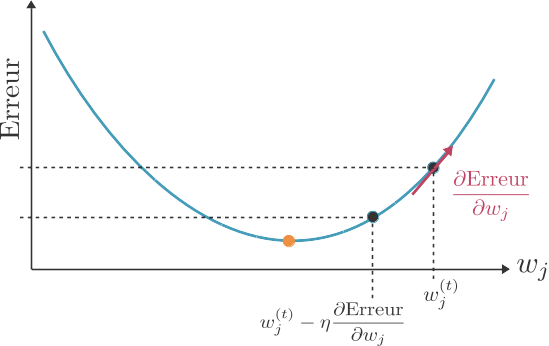
D'accord, mais comment apprendre les poids de connexion ?

Pour entraîner un perceptron, c'est-à-dire apprendre les poids de connexion, nous allons chercher à minimiser l'erreur de prédiction sur le jeu d'entraînement. Nous pourrions faire ça de manière explicite, comme dans le cas de la méthode des moindres carrés pour la régression linéaire ; cependant ce n'est vraisemblablement pas comme ça qu'un réseau de neurones biologiques fonctionne.

De plus, les réseaux de neurones biologiques sont supposés être plastiques, c'est-à-dire qu'ils s'adaptent constamment, en fonction des signaux qu'ils reçoivent. Ainsi, nous allons supposer que nos n observations  ne sont pas observées simultanément mais séquentiellement, l'une après l'autre.

Ainsi les réseaux de neurones sont ils entraînés par des algorithmes dits **incrémentaux** ou **en ligne** (online learning en anglais), par opposition aux modèles que nous avons vus jusqu'à présent, qui sont entraînés par des algorithmes **hors ligne** (batch learning en anglais). Une solution intermédiaire consiste à considérer que les observations arrivent petit paquet par petit paquet, ce qu'on appelle le mini-batch learning.

L'entraînement d'un perceptron est donc un processus **itératif.** Après chaque observation, nous allons ajuster les poids de connexion de sorte à réduire l'erreur de prédiction faite par le perceptron dans son état actuel. Pour cela, nous allons utiliser [l'algorithme du gradient](https://www.wikiwand.com/fr/Algorithme_du_gradient) : le gradient nous donnant la direction de plus grande variation d'une fonction (dans notre cas, la fonction d'erreur), pour trouver le minimum de cette fonction il faut se déplacer dans la direction opposée au gradient. (Lorsque la fonction est minimisée localement, son gradient est égal à 0.)



Un déplacement dans le sens opposé au gradient (flèche rose) rapproche wj de la valeur minimisant l'erreur (point orange).

Ainsi, nous commençons par choisir aléatoirement des valeurs initiales  pour nos poids de connexion. Ensuite, après chaque observation (x(i),y(i))(x(i),y(i)) , nous allons appliquer à chacun des poids la règle de mise à jour suivante :

.

On peut itérer plusieurs fois sur l'intégralité du jeu de données. On itère généralement soit jusqu'à ce que l'algorithme converge (le gradient est suffisamment proche de 0) ou, plus fréquemment, pour un nombre fixé d'itérations.

**η** est un hyperparamètre du réseau de neurones, appelé la **vitesse d'apprentissage** (ou learning rate en anglais).

Si **η** est grand, on s'éloigne beaucoup de . Si ce point était déjà proche de la valeur optimale, on risque de dépasser notre objectif et que  soit plus loin que **** de sa valeur optimale. L'algorithme risque de **diverger**, c'est-à-dire de s'éloigner de la solution optimale.

À l'inverse, si **** est loin de sa valeur optimale et que **η** est faible, l'algorithme va mettre très longtemps à converger.

Il est donc important de **bien choisir la vitesse d'apprentissage.** Il existe des algorithmes permettant d'adapter cette vitesse pour qu'elle soit élevée loin de la solution et plus faible dans son voisinage.

Comment définir la fonction d'erreur ?

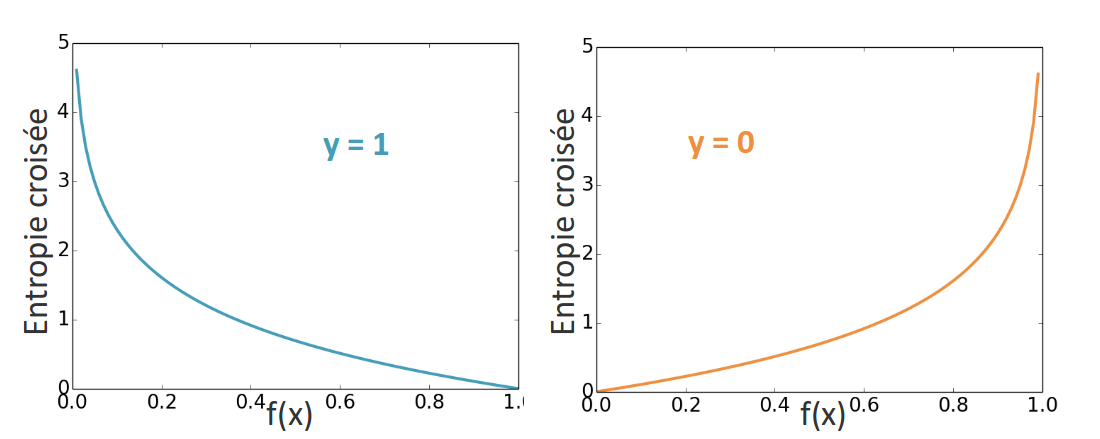
Dans le cas de la **régression**, nous allons choisir l'**erreur quadratique** (comme pour une régression linéaire) :



La règle de mise à jour est donc .

Dans le cas de la **classification**, nous allons choisir l'**entropie croisée.**Dans le cas binaire l'entropie croisée est définie par

.



Quand y=0, l'entropie croisée est d'autant plus élevée que f(x) est proche de 1. Réciproquement, quand y=1, l'entropie croisée est d'autant plus grande que la prédiction est proche de 0.

L'entropie croisée est un peu plus compliquée à différencier que l'erreur quadratique, mais il se trouve après quelques calculs que la règle de mise à jour des poids de connexion est encore une fois  !

Et cela est aussi vrai pour la version multiclasse de l'entropie croisée   : la règle de mise à jour est .

### Résumé

* Le perceptron permet d'apprendre des**modèles paramétriques** basés sur une**combinaison linéaire** des variables.
* Le perceptron permet d'apprendre des modèles de **régression** (la fonction d'activation est l'identité), de **classification binaire** (la fonction d'activation est la fonction **logistique**) ou de **classification multi-classe** (la fonction d'activation est la fonction **softmax**).
* Le perceptron est entraîné par des **mises à jour itératives** de ses poids grâce à **l'algorithme du gradient.** La même règle de mise à jour des poids s'applique dans le cas de la régression, de la classification binaire ou de la classification multi-classe.

## Empilez les perceptrons

#### Architecture d'un perceptron multi-couche

Le perceptron est, essentiellement, un modèle linéaire. Sa capacité de modélisation est donc nécessairement limitée, ce qui a été à l'origine d'un certain désenchantement de la communauté connexionniste au début des années 1970. Dans un célèbre ouvrage de 1969 sur le sujet, le mathématicien et éducateur Seymour Papert et le spécialiste en sciences cognitives Marvin Minsky évoquent même leur intuition de l'inutilité d'étendre le perceptron à des architectures multi-couche... L'histoire leur donnera tort !

Mais comment créer une telle architecture ?

Nous allons créer des **couches intermédiaires** (ou **couches cachées,**hidden layers en anglais) entre la couche d'entrée et celle de sortie. Chaque neurone d'une couche est connecté à tous les neurones de la couche au-dessus de lui. Et voilà ce qu'on appelle un **perceptron multi-couche** (ou multilayer perceptron, souvent abrégé « MLP », en anglais).

Prenons l'exemple d'un perceptron avec une unique couche intermédiaire. La sortie  du h-ème neurone de la couche intermédiaire est obtenue en appliquant la fonction d'activation de ce neurone à une combinaison linéaire des entrées : .

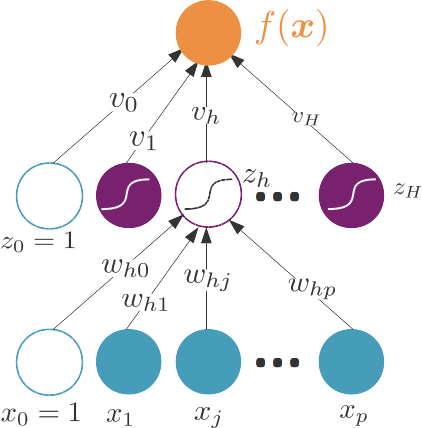
La sortie est ensuite obtenue en appliquant la fonction d'activation du neurone de sortie à la combinaison linéaire des sorties de la couche intermédiaire :



Supposons que l'on utilise la fonction logistique  comme fonction d'activation pour tous les neurones, alors  

Il s'agit donc d'un modèle paramétrique (nous pouvons explicitement écrire la fonction de décision comme une fonction des variables d'entrée), mais il n'est pas linéaire du tout !

Ce modèle paramétrique a d'autant plus de paramètres (i.e. de poids de connexion) qu'il y a de couches intermédiaires et de neurones dans ces couches. Plus il y a de paramètres, plus il faut d'observations pour éviter le sur-apprentissage, et les réseaux de neurones profonds requièrent souvent des quantités massives de données pour bien fonctionner.



Perceptron multi-couche à une couche intermédiaire.

Pour des raisons d'optimisation, plutôt que la fonction logistique, on utilise souvent comme fonction d'activation des couches intermédiaires la fonction [tangente hyperbolique](https://www.wikiwand.com/fr/Tangente_hyperbolique) qui permet de transformer la combinaison linéaire des signaux d'entrée en un nombre entre -1 et 1 plutôt qu'entre 0 et 1.

Cette architecture n'est pas la seule qui soit possible. En particulier, ici les connexions vont toutes des couches basses vers les couches hautes, mais ce n'est pas nécessairement le cas. Ce type d'architecture sans boucle de retour vers les couches basses s'appelle une architecture **feed-forward.**

#### Théorème d'approximation universelle

Un résultat dû à Cybenko en 1989 (raffiné par Hornik en 1991) nous dit que toute fonction f continue définie sur  peut être approximée, avec une précision arbitraire, par un réseau de neurones formé d'une seule couche intermédiaire (à condition d'utiliser un nombre suffisant de neurones dans cette couche).

C'est un résultat extrêmement encourageant : nous pouvions modéliser uniquement des fonctions linéaires, nous pouvons maintenant modéliser n'importe quelle fonction! Cependant, ce résultat est à modérer par la pratique : le nombre de neurones nécessaire dans cette couche intermédiaire est, dans beaucoup de cas, colossal, ce qui rend les réseaux de neurones à une seule couche cachée peu efficaces.

La solution est alors d'empiler les couches cachées, créant ainsi ce que l'on appelle maintenant un **réseau de neurones profond.** Voilà le **deep learning** qui pointe son nez !

On peut interpréter un réseau de neurones multi-couche en considérant que la sortie de chaque couche intermédiaire est une nouvelle représentation des données. La puissance des réseaux de neurones multi-couche vient donc de leur capacité à apprendre une « bonne » représentation, autrement dit une représentation telle que le modèle linéaire (perceptron) entre la dernière couche intermédiaire et la sortie puisse être performant.

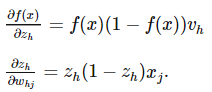
#### Rétropropagation

Comment apprendre les poids d'un MLP ?

Comme pour le perceptron, nous allons utiliser un algorithme itératif basé sur l'algorithme du gradient. Pour faciliter les calculs, on utilise une idée de **rétro-propagation des erreurs** (ou backpropagation), énoncée pour la première fois (hors du domaine des réseaux de neurones) dans les années 1960 et popularisée chez les connexionnistes par Rumelhart, Hinton et Williams en 1986.

Il s'agit en fait de décomposer l'erreur grâce au [théorème de dérivation des fonctions composées](https://www.wikiwand.com/fr/Th%C3%A9or%C3%A8me_de_d%C3%A9rivation_des_fonctions_compos%C3%A9es) : sur le réseau à une couche intermédiaire ci-dessus, la dérivée partielle de l'erreur E par rapport au poids peut s'écrire .

Les deux derniers termes sont simples à calculer car la dérivée dela fonction logistique σ est  :



La dérivée partielle de l'erreur par rapport à un poids d'une couche inférieure peut donc se calculer en calculant la dérivée partielle de l'erreur par rapport à la fonction de décision et les dérivées partielles des sorties (f ou z dans notre exemple à une seule couche) par rapport aux couches précédentes (z ou x dans notre exemple).

La mise à jour des poids peut donc se faire en alternant une phase **forward** (ou montante), dans laquelle les sorties des couches intermédiaires sont mises à jour, et une phase **backward** (ou descendante), dans laquelle le gradient de l'erreur par rapport aux poids d'une couche peut être calculé à partir du gradient de l'erreur par rapport aux poids d'une couche supérieure. La mise à jour des poids se fait au moment de la phase descendante (après que le gradient nécessaire ait été calculé).

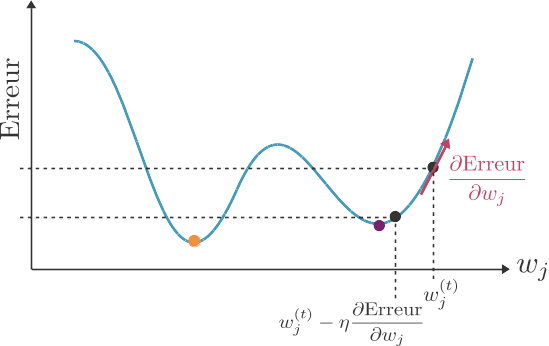
#### Défis de l'apprentissage d'un réseau de neurones multi-couche

L'algorithme de rétropropagation permet d'apprendre des réseaux de neurones multicouche, pour lesquels on ne sait pas optimiser explicitement les paramètres du modèle. Malheureusement, cet algorithme a un certain nombre de limitations et il faut être précautionneux en le manipulant. C'est pour cela que les réseaux de neurones ne sont revenus à la mode qu'en 2006, quand les travaux de Hinton parmi d'autres, combinés à la puissance de calcul des ordinateurs modernes, ont permis de pallier certaines difficultés. C'est à l'heure actuelle encore un domaine en pleine évolution !

##### Conditionnement et minimums locaux

Malheureusement, les dérivées de la fonction d'erreur utilisée dans un réseau de neurones multicouche sont généralement mal conditionnées. Cela veut dire que si l'on perturbe un tout petit peu les conditions initiales de notre algorithme, le résultat sera très différent. En pratique, cela se voit en ce que la fonction d'erreur dessine beaucoup de puits et de montagnes.

Ainsi, la fonction d'erreur d'un réseau de neurones multicouche admet un grand nombre de minimums locaux, c'est-à-dire de points qui sont optimaux dans leur voisinage. Ces points sont situés dans des puits. Ces points ne sont pas des minimums globaux, mais l'algorithme du gradient peut rester « coincé » dans un tel voisinage.



L'algorithme du gradient peut rester coincé dans le puits de droite, et ne détecter que le minimum local (violet) au lieu du minimum global (orange).

L'initialisation des poids de connexion, la standardisation des variables, le choix de la vitesse d'apprentissage et celui de la fonction d'activation ont tous un impact sur la capacité du réseau de neurones multicouche à trouver un bon minimum. L'utilisation d'algorithmes d'apprentissage en mini-batch ou d'une vitesse d'apprentissage adaptative peuvent pallier en partie ce problème.

##### Instabilité du gradient

Dans les architectures multicouches, le gradient a tendance à être de plus en plus petit quand on redescend les couches cachées depuis la sortie vers l'entrée du réseau. Cela signifie que les neurones des couches basses apprennent beaucoup plus lentement que ceux des couches hautes.  C'est ce qu’on appelle en anglais le phénomène de vanishing gradient (« **disparition du gradient** »).

À l'inverse, il est aussi possible que le gradient prenne des valeurs très grandes dans les couches basses, ce que l'on appelle «**l'explosion du gradient** » (exploding gradient).

Ce qu'il faut retenir, c'est que le gradient est **instable.**Cette instabilité est liée au fait que, dans la rétropropagation, le gradient des couches basses est un produit de celui des couches hautes.

##### Saturation

La **saturation** est un phénomène qui apparaît quand la plupart des neurones à activation logistique du réseau ont une sortie à 0 ou 1 (-1 et + 1 pour une activation tangente hyperbolique), autrement dit quand la somme pondérée des signaux qu'ils reçoivent en entrée est trop grande (en valeur absolue). Cela signifie que les poids de connexion sont trop grands. À ce stade, un changement mineur dans les données d'entrée ne va quasiment pas avoir d'impact sur la sortie, et le réseau ne pourra plus apprendre (ou alors très lentement). De plus, un réseau saturé est souvent en situation de sur-apprentissage...

Une des stratégies que l'on peut utiliser pour éviter le sur-apprentissage est la **régularisation**. Eh oui, comme pour la régression linéaire ! Nous allons même utiliser le même régularisateur que pour la régression ridge : la norme l2 des poids de connexion. Contrôler cette norme permet en effet de contrôler la valeur absolue des poids, évitant ainsi que les sommes pondérées pré-activation soient trop grandes. Dans le monde des réseaux de neurones, cette technique s'appelle la **dégradation des pondérations** (« weight decay » en anglais). Il s'agit de remplacer l'erreur par .

#### Résumé

* Empiler des perceptrons en un **réseau de neurones multicouches (feed-forward)** permet de **modéliser des fonctions arbitrairement complexes.** C'est ce qui donne aux réseaux de neurones profonds la puissance prédictive qui fait actuellement leur succès.
* L'entraînement de ces réseaux se fait par **rétropropagation.** Attention, cet algorithme ne converge pas nécessairement, et pas nécessairement vers la solution optimale !
* Plus il y a de paramètres (i.e. de poids de connexion), **plus il faut de données** pour pouvoir apprendre les valeurs de ces paramètres sans risquer le **sur-apprentissage.**
* Il existe de nombreuses autres architectures de réseaux de neurones que celle présentée ici, et qui permettent de modéliser des types de données particuliers (images, sons, dépendances temporelles...)

# Partie 2

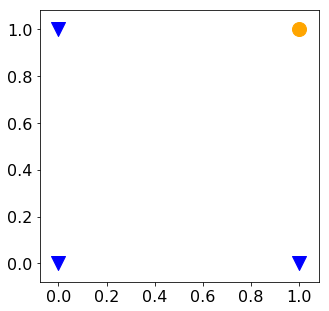
Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Comprendre le fonctionnement du perceptron
* Appréhender les réseaux de neurones multi-couches et la backpropagation

### Question 1

**Considérons la fonction binaire ET, représentée sur le diagramme ci-dessous. Les données sont en 2 dimensions, chacune de ces dimensions peut prendre la valeur 0 ou 1, et l'étiquette y vaut le ET logique entre les deux coordonnées.**

Peut on séparer les points bleus des points orange ?

*Attention, plusieurs réponses sont possibles.*

* + 

Un perceptron peut apprendre cette fonction.

* + 

Un perceptron ne peut pas apprendre cette fonction.

* + 

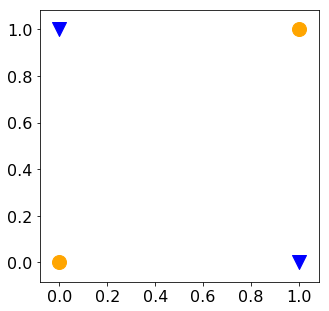
Un perceptron multi-couche peut apprendre cette fonction.

* + 

Un perceptron multi-couche ne peut pas apprendre cette fonction.

*Un perceptron ne peut qu'encoder des fonctions de séparation linéaires. Un perceptron multi-couche peut apprendre des fonctions plus compliquées, et a plus forte raison celle-ci !*

### Question 2

Peut-on séparer les points bleus des points orange ?

**Considérons la fonction binaire XOR, représentée sur le diagramme ci-dessous. La fonction XOR vaut 1 quand les deux variables sont opposées (l'une à 0, l'autre à 1) et 0 si elles sont égales.**

* + 

Un perceptron peut apprendre cette fonction.

* + 

Un perceptron multi-couche avec une couche intermédiaire peut apprendre cette fonction.

* + 

Seul un perceptron multi-couche avec plus d'une couche intermédiaire peut apprendre cette fonction.

* + 

Aucun réseau de neurones ne peut apprendre cette fonction.

*Un perceptron apprend une frontière de séparation qui est une droite. Il n'est pas possible de séparer les points bleus des points orange, il faudra donc ajouter une couche pour pouvoir apprendre cette fonction.*

### Question 3

**J'ai essayé d'appliquer des algorithmes linéaires à mes données, mais leur performance laissait à désirer. Je pense qu'une fonction non-linéaire complexe serait plus appropriée. J'ai quelques centaines d'observations.**

* + 

Je construis un réseau de neurones à 3 couches cachées avec 100 neurones chacun : un tel réseau a certainement suffisamment de flexibilité pour trouver la bonne fonction de décision !

* + 

Je construis un réseau de neurones avec une couche intermédiaire et peu de neurones dedans : je ne veux pas avoir plus de paramètre dans mon réseau que d'observations dans mes données.

* + 

Avec aussi peu d'observations, je n'arriverai jamais à construire un modèle non-linéaire.

*Un réseau de neurones très compliqué va avoir beaucoup de paramètres et donc besoin de beaucoup d'observations pour apprendre. Le réseau à 3 couches intermédiaires avec 100 neurones chacune a 100 x 100 x 100 poids à apprendre juste entre les couches... Cependant, un réseau de taille raisonnable, ou une SVM, peuvent parfaitement apprendre sur quelques centaines d'observations.*

### Question 4

**Cocher les affirmations vraies concernant la qualité du modèle appris par un réseau de neurones multi-couches :**

*Attention, plusieurs réponses sont possibles.*

* + 

Elle dépend du choix des fonctions d'activation.

* + 

Elle dépend du nombre de neurones et du nombre de couches.

* + 

Elle ne dépend pas de la vitesse d'apprentissage.

* + 

Elle ne dépend pas des poids initiaux du réseau.

* + 

Elle dépend de la quantité de données utilisées.

*Parce qu'il s'agit d'un problème d'optimisation complexe dont la solution est recherchée de manière heuristique, et que l'on risque de tomber dans des minimas locaux, le modèle appris par un réseau de neurones multi-couches dépend de la vitesse d'apprentissage, de tout ça !*

### Question 5

**Un perceptron multi-couche permet de résoudre des problèmes de classification multi-classe...**

*Attention, plusieurs réponses sont possibles.*

* + 

... avec un seul réseau de neurones.

* + 

... uniquement en entraînant un réseau de neurones par classe.

* + 

... et d'interpréter facilement le rôle de chaque variable dans la fonction de décision.

* + 

... à condition d'avoir beaucoup de données pour chaque classe.

*On peut utiliser un réseau de neurones avec autant de sorties que de classes + des activations soft-max pour traiter un problème multi-classe. Cela nécessite beaucoup de données, comme pour tous les réseaux de neurones. La fonction de décision peut être écrite explicitement mais est difficile à interpréter.*

### Question 6

**Les réseaux de neurones sont connus pour être des modèles complexes à faire converger. En effet, la fonction d'erreur objectif est dure à**

* + 

optimiser, car l'optimum n'est pas clair

* + 

minimiser car il n'y a pas de minimum global

* + 

maximiser, car il y a des maximum globaux

* + 

minimiser, car il y a des minimum locaux

*Comme expliqué dans le cours, on cherche à minimiser la fonction d'erreur, qui n'est pas convexe et donc n'a pas un seul minimum qui (est global).*

### Question 7

**Les réseaux de neurones sont populaires car ils ont la particularité de**

* + 

étant donné les bons hyperparamètres, pouvoir approximer n'importe quelle fonction continue (approximateur universel)

* + 

avec n'importe quels hyperparamètres et assez de temps et puissance de calcul, pouvoir approximer n'importe quelle fonction continue (approximateur universel)

* + 

avec la bonne fonction d'erreur et algorithme de descente de gradient, pouvoir approximer n'importe quelle fonction continue (approximateur universel)

*C'est un approximateur universel théorique, mais il faut effectivement les bons hyperparamètres (couches, nombre de neurones par couche, etc).*