DEEP LEARNING AVEC KERAS

**Table des matières**

[I. INTRODUCTION](#_gjdgxs) **3**

[Introduction](#_p77kz9rspbg4) 3

[Objectif](#_khb9jks60ufq) 3

[Contexte](#_s7ub2k6q3laf) 3

[Compétences requises](#_nvktmaab6ux7) 4

[II. Premier essai avec un Random Forest](#_tyjcwt) **8**

[Contexte et objectif](#_yriwzif7ei8d) 8

[Compétences requises](#_h3otig7p0q19) 8

[III. Concepts fondamentaux](#_44sinio) **13**

[Objectif](#_l2fg9ph2xc2c) 13

[Classification grâce au produit scalaire usuel.](#_eis3grng7nu0) 13

[Séparabilité Linéaire](#_7cyh0jnpj7or) 14

[Fonction de Perte](#_sumkiop6b5fc) 14

[L'algorithme du Perceptron](#_zcw7y36ch3pm) 15

[Entraînement par Descente de Gradient](#_kfiiaml0es7b) 17

[Limites de la descente de gradient](#_co7drywnjkja) 18

[Perceptron Multicouche](#_le2btc9bpo3f) 19

[Ce qu'il faut retenir:](#_o1t9mcdr07l0) 19

[Fin](#_8x1uur3uumly) 20

[IV. Prédiction à l’aide des Dense Neural Networks](#_3mmzoe9) **21**

[Contexte et objectif](#_ry84izdh0759) 21

[Compétences requises](#_cvkv2mxh5558) 21

[V. Types de couches](#_30x7rho) **29**

[Rappel : Les couches Dense](#_qecwmhvdifvf) 29

[Les couches de Convolution](#_mxnqjedgubi1) 29

[Couches de Régularisation](#_689a4f3ddwtb) 31

[Couche de Max-Pooling](#_tfkxv4cyd4vu) 32

[Couche de Average-Pooling](#_e3rxs63qac6w) 32

[Couche de Dropout](#_bdh22acdkp1g) 32

[VI. Prédiction à l’aide des CNN](#_2rf19tf) **34**

[Contexte et objectif](#_lyyhn57n9tn2) 34

[Compétences requises](#_uuml2dbl7l40) 34

[Construction de l'architecture CNN](#_6hjyfmvn2f42) 35

[**VII. Résolution avec l’architecture LeNet**](#_9ahheoi4o61m) **41**

[Contexte et objectif](#_7s4ztng54hon) 41

[Ce qu'il faut retenir :](#_h101v3fy7voz) 47

[**VIII. Transfer Learning**](#_3ygridtkca6u) **49**

[Contexte et objectif](#_f0nxl9ct6868) 49

[Compétences requises](#_4mqr05l2acaw) 49

[Exploration des données](#_2g9kmefjfuag) 50

[Générateur de données](#_xex9667glmq1) 53

[Transfer Learning](#_fndli5px7dqc) 56

[Principe](#_sa8ns1hx97ms) 56

[Modèle](#_o1xz82r94t3d) 57

[Features extraction](#_vu21yuftvtv8) 61

[Ce qu'il faut retenir :](#_b4pakzs9mnv0) 62

# I. INTRODUCTION

## Introduction

Depuis 2012, les algorithmes à base de deep learning (apprentissage profond) semblent prêts à résoudre bien des problèmes : reconnaître des visages comme le propose DeepFace, vaincre des joueurs de go ou de poker ou bientôt permettre la conduite de voitures autonomes ou encore la recherche de cellules cancéreuses.

Pourtant, les fondements de ces méthodes ne sont pas si récents : le deep learning a été formalisé en 2007 à partir des nouvelles architectures de réseaux de neurones dont les précurseurs sont McCulloch et Pitts en 1943. Suivront de nombreux développements comme le perceptron (neurone artificiel), les réseaux de neurones convolutifs de Yann Le Cun et Yoshua Bengio en 1998 et les réseaux de neurones profonds qui en découlent en 2012 et ouvrent la voie à de nombreux champs d’application comme la vision (images), le traitement du langage (NLP) ou la reconnaissance de la parole (audio).

**Pourquoi maintenant ?**

Parce que ces nouvelles techniques de machine learning profitent de la montée massive des données, ainsi que de capacités de calcul phénoménales grâce aux processeurs graphiques

## Objectif

L'objectif de cette formation est de familiariser avec des méthodes de machine learning plus poussées. Les méthodes de *deep learning* ont fait leurs preuves récemment et grâce à des packages comme **Keras** que nous utiliserons dans cette formation, ces outils sont maintenant dans les mains du plus grand nombre. Nous avons choisi le framework **Keras** pour sa facilité à implémenter des modèles complexes de Deep Learning.

Nous allons travailler avec la base de données *MNIST*. Cette base contient des images de chiffres manuscrits que nous tenterons de classifier suivant plusieurs méthodes de plus en plus poussées, en partant du simple Random Forest jusqu'à l'algorithme LeNet-5 développé spécifiquement pour cette base.

## Contexte

La base MNIST ("Modified National Institute of Standards and Technology") est une base de données de chiffres écrits à la main développée pour le problème de reconnaissance de chiffres manuscrits par trois chercheurs très réputés dans ce domaine: Yann LeCun, Corinna Cortes et Christopher Burges.

Elle regroupe 70 000 images, issues d'une base de données antérieure, appelée NIST. Ces images sont en échelle de gris, normalisées centrées et de taille 28x28 pixels. Les images sont données ici sous forme de vecteurs unidimensionnels de taille 784 (= 28 x 28).

Plus de détails sur les données MNIST dans le lien suivant: [MNIST](http://yann.lecun.com/exdb/mnist/index.html).  
Plus de détails sur le problème de reconnaissance de chiffres écrits à la main [ici](http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf).

## Compétences requises

* Scikit-learn
* Matplotlib
* Exécutez la cellule ci-dessous pour importer les modules nécessaires à l'exercice.

%matplotlib inline

import numpy as np # Pour la manipulation de tableaux

import matplotlib.pyplot as plt # Pour l'affichage d'images

from matplotlib import cm # Pour importer de nouvelles cartes de couleur

from sklearn.model\_selection import train\_test\_split # Pour répartir les données

from keras.datasets.mnist import load\_data # Pour importer le datasets mnist de Keras

Le module datasets de **keras** comprend un ensemble de sous module de jeu de données ([plus d'informations](https://keras.io/datasets/)). Pour charger un jeu de données de ses sous-modules, nous pouvons utiliser la fonction suivante :

from keras.datasets import cifar10

(X\_train, y\_train), (X\_test, y\_test) = cifar10.load\_data()

* Charger le dataset **mnist** à l'aide de la fonction load\_data du sous module **mnist**. Appeler l'échantillon d'entraînement **X\_train** et **y\_train**, et l'échantillon de validation **X\_test** et **y\_test**.
* Afficher la forme de **X\_train** et **y\_train**

# Pour importer le datasets mnist de Keras

from keras.datasets.mnist import load\_data

# Chargement des données MNIST

(X\_train, y\_train), (X\_test, y\_test) = load\_data()

# Shape of X\_train and y\_train

print('Shape of X:', X\_train.shape)

print('Shape of y:', y\_train.shape)

Pour afficher des graphiques sur une grille de figures, on utilise la fonction [subplot](http://matplotlib.org/api/pyplot_api.html?highlight=subplot#matplotlib.pyplot.subplot) de *matplotlib.pyplot*.

Les images originales sont en niveau de gris. Lors de leur affichage, il faut préciser à la fonction [imshow](https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.imshow) que l'on souhaite visualiser les pixels en nuance de gris, en choisissant la carte de couleur (**cmap**) correspondante . Ici, on choisira cm.binary parmi les *colormaps* de matplotlib que nous avons importées.

* Afficher, dans une grille de figures, 6 images tirées aléatoirement de l'échantillon **X\_train**, avec en titre les labels correspondants de **y\_train**. Pour chaque image il faudra :
  + Sélectionner son emplacement dans la grille de figures.
  + Afficher l'image en niveaux de gris en précisant la *colormap* **binary**.
  + Afficher le label correspondant à l'image dans le titre.

  Les labels des images sont contenus dans le vecteur y\_train.

  Ici, il est nécessaire de préciser le paramètre 'cmap = cm.binary' dans la fonction imshow afin d'afficher l'image en niveaux de gris sur fond blanc.

j = 1

for i in np.random.choice(np.arange(0, len(y\_train)), size = 6):

# Image

img = X\_train[i]

# Subplot nous permet de sélectionner une des sous-figures parmi une grille. Dans notre cas la grille

# a 2 lignes et 3 colonnes.

plt.subplot(2, 3, j)

j = j + 1

# Suppression des axes

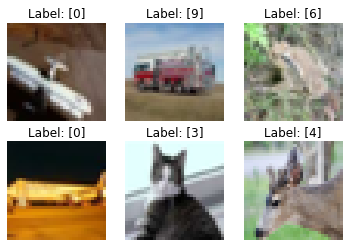
plt.axis('off')

# Affichage de la figure en niveaux de gris

plt.imshow(img, cmap=cm.binary, interpolation='None')

# Modification du titre de la figure

plt.title('Label: ' + str(y\_train[i]))



Pour se donner une idée plus précise de la forme des différents chiffres écrits, il est possible de procéder à quelques statistiques basées sur les pixels des images.

* Pour chaque chiffre de 0 à 9, afficher **l'image moyenne** de l'échantillon X\_train. Pour cela, pour chaque chiffre allant de 0 à 9, utiliser la méthode mean d'un *array* numpy pour calculer la moyenne de chaque pixel, en précisant 'axis = 0' en paramètre de la méthode mean pour que la moyenne se fasse sur les images et non sur les lignes ou les colonnes.
* Faire une boucle allant de 0 à 9 permettant de calculer l'image moyenne pour chaque chiffre séparément. À l'intérieur de la boucle, sélectionnez les images qui correspondent au bon label et calculez ensuite la moyenne. Enfin, affichez-la comme précédemment.

for i in range(10):

# Selection des lignes de X\_train correspondant au label i

t = X\_train[y\_train[y\_train == i]]

# Calcul de l'image moyenne

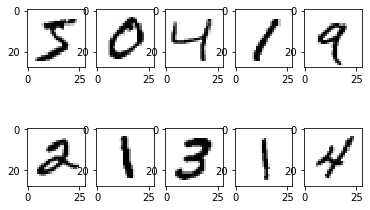
img = t.mean(axis=0)

# Affichage de l'image dans le i+1-ème emplacement d'une grille de figures

# à 2 lignes et 5 colonnes.

plt.subplot(2, 5, i+1)

plt.imshow(img, cmap=cm.binary, interpolation='None')



* Pour se donner une idée de la façon dont chaque chiffre varie dans son échantillon respectif, afficher pour chaque label entre 0 et 9 son **écart type** dans l'échantillon X\_train , grâce à la méthode std d'un *array* numpy.

for i in range(10):

# Selection des lignes de X\_train correspondant au label i

t = X\_train[y\_train == i]

# Calcul de l'image moyenne

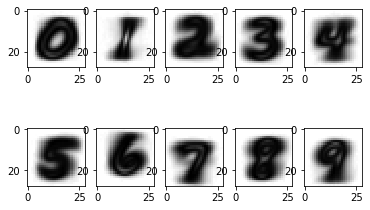
img = t.std(axis=0)

# Affichage de l'image dans le i+1-ème emplacement d'une grille de figures

# à 2 lignes et 5 colonnes.

plt.subplot(2, 5, i+1)

plt.imshow(img, cmap=cm.binary, interpolation='None')



# II. Premier essai avec un Random Forest

## **Contexte et objectif**

Le principal objectif de l'exercice est d'implémenter une méthode de reconnaissance des chiffres écrits à la main grâce à un modèle Random forests (RM). Cette étape servira de baseline pour la comparaison des résultats produits dans la suite de ce module.

Plus de renseignements sur les Random forests et leur utilisation avec le package *scikit-learn* [ici](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html)

## Compétences requises

* Scikit-learn
* Matplotlib
* Exécutez la cellule ci-dessous pour importer les modules nécessaires à l'exercice.

%matplotlib inline

import numpy as np # Pour la manipulation de tableaux

import matplotlib.pyplot as plt # Pour l'affichage d'images

from matplotlib import cm # Pour importer de nouvelles cartes de couleur

from sklearn.model\_selection import train\_test\_split # Pour répartir les données

from sklearn.model\_selection import GridSearchCV, cross\_val\_score # Pour évaluer un modèle

from sklearn import metrics

from keras.datasets.mnist import load\_data # Pour charger le dataset MNIST

from sklearn.ensemble import RandomForestClassifier # Pour instancier et entraîner un modèle Random Forest

import itertools # Pour créer des iterateurs

* Récupérer, comme dans l'exercice précédent, les données **mnist** en un échantillon d'entraînenement (**X\_train**, **y\_train**) ainsi qu'un échantillon de validation (**X\_test**, **y\_test**)
* Changer la forme des images de **X\_train** et de **X\_test** en des vecteurs de tailles 28\*28 à l'aide de la méthode reshape.

# Pour importer le datasets mnist de Keras

from keras.datasets.mnist import load\_data

# Chargement des données MNIST

(X\_train, y\_train), (X\_test, y\_test) = load\_data()

# Changer la forme de X\_train et X\_test

X\_train = X\_train.reshape([-1, 28\*28])

X\_test = X\_test.reshape([-1, 28\*28])

# Shape of X\_train and y\_train

print('Shape of X:', X\_train.shape)

print('Shape of y:',y\_train.shape)

Shape of X: (60000, 784)

Shape of y: (60000,)

* Construire un modèle classique de classification **Random Forest** appelé **model** à l'aide du constructeur RandomForestClassifier. Préciser dans la définition du **Random Forest** n\_jobs = -1 pour réaliser les calculs avec l'ensemble des cœurs du processeur.
* Entraîner le modèle à partir de l'échantillon **X\_train** et des labels **y\_train**.

model = RandomForestClassifier(n\_jobs = -1)

model.fit(X\_train, y\_train)

* Prédire avec ce modèle la classification de l'échantillon *X\_test* dans un tableau appelé **test\_pred**.
* Afficher le score de précision de ces prédictions grâce à la fonction accuracy\_score contenue dans le sous-module metrics de *scikit-learn*.

# Prédiction sur l'échantillon de test

test\_pred = model.predict(X\_test)

# On évalue le niveau de précision de notre prédiction.

print("Précision de la prédiction:", metrics.accuracy\_score(y\_test, test\_pred)\*100, '%')

# Il devrait être aux alentours de 95%.

Précision de la prédiction: 96.78999999999999 %

* Afficher un compte-rendu évaluatif détaillé de la méthode, grâce à la fonction classification\_report du sous-module metrics.
* En déduire les chiffres les mieux reconnus sur cet échantillon.

print("Evaluation détaillée de la Classification par RDF :\n \n" ,

(metrics.classification\_report(y\_test, test\_pred)))

# On observe que les chiffres 0, 1 et 6 ont le meilleur rappel (recall), tandis que le chiffre 1 a la meilleure précision.

# Le F1-score, qui calcule la moyenne harmonique de la précision et du rappel, est un bon indicateur lorsqu'on cherche à prendre

# en compte ces deux mesures.

Evaluation détaillée de la Classification par RDF :

precision recall f1-score support

0 0.98 0.99 0.98 980

1 0.99 0.99 0.99 1135

2 0.96 0.96 0.96 1032

3 0.96 0.96 0.96 1010

4 0.97 0.97 0.97 982

5 0.97 0.96 0.96 892

6 0.97 0.98 0.97 958

7 0.97 0.96 0.97 1028

8 0.96 0.96 0.96 974

9 0.96 0.95 0.95 1009

accuracy 0.97 10000

macro avg 0.97 0.97 0.97 10000

weighted avg 0.97 0.97 0.97 10000

* Calculer et afficher la matrice de confusion de **y\_test** et **test\_pred**, appelée **cnf\_matrix**, pour avoir un aperçu des erreurs réelles de la prédiction. La matrice de confusion se calcule grâce à la fonction confusion\_matrix du sous-module **metrics** de **sklearn**.
* Afficher les paires de chiffres qui ont souvent été confondus (plus de 15 cas d'erreurs), sous la forme du message suivant :"Le chiffre *i* a souvent été pris pour le chiffre *j*"

#Réponse valable:

cnf\_matrix = metrics.confusion\_matrix(y\_test, test\_pred)

print(cnf\_matrix)

###Optionnel: Afficher une matrice de confusion sous forme de tableau coloré

classes = range(0,10)

plt.figure()

plt.imshow(cnf\_matrix, interpolation='nearest',cmap='Blues')

plt.title("Matrice de confusion")

plt.colorbar()

tick\_marks = np.arange(len(classes))

plt.xticks(tick\_marks, classes)

plt.yticks(tick\_marks, classes)

for i, j in itertools.product(range(cnf\_matrix.shape[0]), range(cnf\_matrix.shape[1])):

plt.text(j, i, cnf\_matrix[i, j],

horizontalalignment = "center",

color = "white" if cnf\_matrix[i, j] > ( cnf\_matrix.max() / 2) else "black")

plt.ylabel('Vrais labels')

plt.xlabel('Labels prédits')

plt.show()

#####

for i, j in itertools.product(range(cnf\_matrix.shape[0]), range(cnf\_matrix.shape[1])):

if(cnf\_matrix[i,j] >15 and i!=j):

print("Le chiffre {0} a souvent été pris pour le chiffre {1}".format(i,j))

[[ 970 0 1 0 0 2 3 1 3 0]

[ 0 1124 2 3 0 2 3 0 1 0]

[ 7 0 995 6 4 0 4 8 8 0]

[ 0 0 11 968 0 9 0 9 10 3]

[ 1 0 1 0 950 0 6 1 3 20]

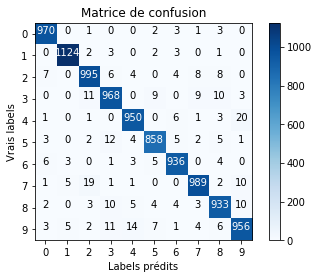
[ 3 0 2 12 4 858 5 2 5 1]

[ 6 3 0 1 3 5 936 0 4 0]

[ 1 5 19 1 1 0 0 989 2 10]

[ 2 0 3 10 5 4 4 3 933 10]

[ 3 5 2 11 14 7 1 4 6 956]]



Le chiffre 4 a souvent été pris pour le chiffre 9

Le chiffre 7 a souvent été pris pour le chiffre 2

* Afficher aléatoirement 6 images de l'échantillon X\_test, ainsi que pour chaque image la prédiction effectuée, en titre de celle-ci.

  Vous pouvez supprimer les axes d'une figure avec l'instruction **plt.axis('off')** pour ne pas gêner l'affichage des titres.

j = 1

for i in np.random.choice(np.arange(0, len(y\_test)), size=6):

img = X\_test[i]

# Redimensionnement de l'image

img = img.reshape(28, 28)

# Sélection de la sous-figure

plt.subplot(2, 3, j)

j = j + 1

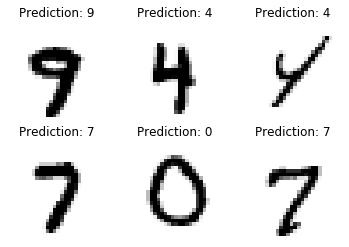
# Suppression des axes

plt.axis('off')

# Affichage de l'image en niveaux de gris

plt.imshow(img, cmap = cm.binary)

plt.title('Prediction: ' + str(test\_pred[i]))



# III. Concepts fondamentaux

Ce module ne compte pas d'exercices pratiques comme les autres et ne sert qu'à illustrer les concepts fondamentaux du *deep learning* le plus clairement possible.

Il n'y a donc aucun pré-requis technique pour aborder ce module à part un niveau très basique en algèbre linéaire et calcul différentiel.

## Objectif

L'objectif de ce module de formation est de vous familiariser avec les concepts fondamentaux du *deep learning*. Ces concepts vous permettront de comprendre les algorithmes qui nous permettent de construire et entraîner un réseau de neurones.

## Classification grâce au produit scalaire usuel.

Le produit scalaire est utilisé pour la classification grâce à ses propriétés géométriques. Le résultat produit par le produit scalaire entre deux vecteurs est très facile à interpréter.

Dans la figure interactive suivante, vous pouvez apercevoir un point et un vecteur. Le point **x** de couleur bleue et de coordonnées **(*****x*****1****,*****x*****2****)** est le point que nous devons classifier. Le vecteur **w** de couleur verte et de coordonnées **(*****w*****1****,*****w*****2****)** est le vecteur qui nous permettra de le classifier.

La classification avec le produit scalaire se fait ainsi:

* Si le produit scalaire entre **x** et **w** est **positif**, **x sera classifié à 1** (de couleur bleue).
* Si le produit scalaire entre **x** et **w** est **négatif**, **x sera classifié à 0** (de couleur rouge).

La **ligne verte perpendiculaire à w** représente tous les points du domaine tels que **le produit scalaire avec w vaut 0**. On appelle cette ligne la **frontière de décision** du problème de classification.

* Exécuter la cellule suivante pour afficher l'interaction

from interaction\_dense import show\_dotProduct

show\_dotProduct()

Le résultat du produit scalaire entre x et w peut s'interpréter géométriquement:

* Si le produit scalaire entre x et w est positif, alors x est **"au dessus"** (dans la direction de w) de la frontière de décision.
* Si le produit scalaire entre x et w est négatif, alors x est **"en dessous"** (dans la direction de w) de la frontière de décision.

## Séparabilité Linéaire

La notion de séparabilité linéaire d'une base de données est fondamentale pour utiliser la classification par produit scalaire.

La figure suivante correspond à la base de données **Iris** qui contient 2 variables: Sepal Width et Sepal Length correspondant à la largeur et longueur de sépale de deux espèces d'iris différentes. Le problème de classification est le suivant:

* **Pouvons-nous à partir de ces deux variables déterminer l'espèce d'une fleur ?**

Les **points verts** de la figure correspondent aux fleurs d'espèce **iris setosa** et les **points oranges** correspondent aux fleurs d'espèce **iris virginica**. (La base de données a été normalisée pour obtenir une meilleure visualisation).

* Exécuter la cellule suivante pour afficher l'interaction

from interaction\_dense import show\_dataset

show\_dataset()

Pour résoudre ce problème **géométriquement** avec la classification par produit scalaire, nous pouvons reformuler la question de la manière suivante:

* **Existe-t-il une frontière de décision linéaire qui nous permettrait de séparer les deux espèces ?**
* Trouver à l'aide de la figure interactive suivante un vecteur w définissant une frontière de décision **séparant les points verts des points oranges**.

from interaction\_dense import show\_data

show\_data()

Une solution possible est le vecteur w = (-1.8, 0.95) qui définit une frontière de décision **linéaire** séparant **parfaitement** les deux groupes d'individus. On dit alors que la base de données est **linéairement séparable**.

Dans ce cas particulier, la frontière de décision porte le nom d'**hyperplan séparateur**. Nous utilisons cette dénomination car les points définissant la frontière de décision sont les points satisfaisant l'**équation de plan** {*x*=(*x*1,*x*2)∈ℝ2:⟨*x*,*w*⟩=*x*1*w*1+*x*2*w*2=0}

## Fonction de Perte

Nous avons vu que nous pouvions trouver pour la base de données Iris un hyperplan séparant parfaitement les deux espèces de fleur. Cependant, cette solution a été trouvée visuellement.

* **Comment trouver mathématiquement un hyperplan séparateur?**

Il faut d'abord trouver un moyen de quantifier la qualité de la séparation des espèces. Une des possibilités les plus simples est de **compter le nombre d'erreurs de classification que nous ferions en utilisant un vecteur spécifique**.

Supposons que notre base de données contient *n* points *X*=(*x**i*)*i*=1,2,...,*n*∈ℝ*d* et qu'à chacun de ses points sont associées les valeurs *Y*=(*y**i*)*i*=1,2,...,*n*∈{0,1} correspondant au groupe auquel appartient le point *x**i*.

Dans notre exemple, la classe 1 correspondrait à l'espèce *iris setosa* et la classe 0 à l'espèce *iris virginica*.

Mathématiquement, la classification d'un individu *x**i* par un vecteur *w* se ferait par une fonction *f* définie ainsi:



Ainsi, le nombre d'erreurs peut être calculé par une fonction de *w*=(*w*1,*w*2), *X* et *Y* qui s'écrirait ainsi:



Cette fonction nous permet de définir un critère pour déterminer la meilleure solution à notre problème de classification. Les fonctions de ce type s'appellent des **fonctions de perte** (*loss function* en anglais).

Plus la valeur de cette fonction est basse, plus notre fonction de classification est performante. **Minimiser cette fonction de perte par rapport au vecteur w est donc équivalent à trouver un hyperplan séparateur**.

* A l'aide de la figure interactive suivante, visualiser la fonction de perte associée à notre problème de classification.

from interaction\_loss import show\_loss

show\_loss()

## L'algorithme du Perceptron

Les étapes que nous venons de suivre peuvent être automatisées dans un algorithme qui s'appelle l'algorithme du **Perceptron** inventé en 1957 par Frank Rosenblatt à l'université de Cornell. Le vocabulaire spécifique à cet algorithme est le suivant:

* Le vecteur *w* définissant l'hyperplan séparateur porte le nom de **vecteur de poids** (*weight vector* en anglais).
* Le vecteur *x* correspondant à un individu à classifier porte le nom de **vecteur d'entrée** (*input vector* en anglais).

En suivant cette terminologie, le produit scalaire entre *x* et *w* peut être interprété comme une somme pondérée des features de x.

Une autre subtilité de l'algorithme du perceptron est que le premier feature de *x* aura **systématiquement la valeur** **1**. Ce feature artificiel sert en fait à décaler l'hyperplan séparateur dans l'espace d'une certaine valeur que l'on appellera **"biais"** (*bias term* en anglais).

L'équation définissant l'hyperplan séparateur devient alors:

⟨*x*,*w*⟩+*b**i**a**i**s*=0

Une autre particularité de l'algorithme du Perceptron est que la fonction de classification n'est pas tout à fait le produit scalaire avec le biais. Dans de nombreux cas, nous allons utiliser une fonction nommée d'**activation** qui va nous permettre d'utiliser une fonction de perte plus adaptée à notre problème.

Les fonctions d'activation les plus utilisées pour l'algorithme du Perceptron sont la tangente hyperbolique **tanh** et la fonction logistique **sigmoid**.

Supposons que les (*y**i*)*i*=1,..,*n*∈{−1,1} et que nous utilisons la fonction *tanh* comme activation. La fonction de classification devient:

*f*(*x**i*,*w*)=*t**a**n**h*(⟨*x*,*w*⟩+*b**i**a**i**s*)

La fonction de perte devient:



La fonction de classification et la fonction de perte **ne contiennent plus de fonctions indicatrices** et sont maintenant **dérivables en tout point**. Cette nuance est très importante pour la suite. De plus, la fonction de perte est **convexe** (si les données sont linéairement séparables).

Une autre raison pour laquelle nous utiliserons la fonction *tanh* est que

*t**a**n**h*(⟨*x*,*w*⟩+*b**i**a**i**s*)=0⟺⟨*x*,*w*⟩+*b**i**a**i**s*=0

car

*t**a**n**h*(*x*)=0⟺*x*=0

C'est-à-dire que l'équation de l'hyperplan séparateur optimal est la même que si nous n'utilisions pas de fonction d'activation.

## Entraînement par Descente de Gradient

Grâce à la fonction d'activation, la fonction de perte est dérivable. Nous pouvons utiliser un algorithme de minimisation nommé **algorithme de descente de gradient** pour trouver le vecteur *w* optimal.

L'algorithme de descente de gradient est très simple. Le cas le plus simple à illustrer est le cas à une dimension. Dans la figure suivante, la fonction représentée est *f*(*x*)=*x*2 et sa dérivée est *f*′(*x*)=2*x*.

* Exécuter la cellule suivante pour afficher l'interaction

from interaction\_dense import show\_optimization\_square

show\_optimization\_square()

Lorsque *f*′(*x*)<0 (en rouge), alors *f* est **décroissante** au voisinage de *x*.

Lorsque *f*′(*x*)>0 (en vert), alors *f* est **croissante** au voisinage de *x*.

Ainsi, un point *x**m**i**n* est un minimum d'une fonction **convexe** que si *f*′(*x**m**i**n*)=0, c'est-à-dire que *f* doit être croissante au voisinage de tout point *x*>*x**m**i**n* et décroissante au voisinage de tout point *x*<*x**m**i**n*

Soit *x*0 un point aléatoire du domaine de définition de *f*. L'algorithme de descente de gradient consiste alors à choisir un point dans la direction **opposée** au gradient. C'est-à-dire que:

* Si *f*′(*x*0)<0, *f* est décroissante au voisinage de *x*0, ce qui veut dire que le minimum *x**m**i**n* est forcément supérieur à *x*0.
* Si *f*′(*x*0)>0, *f* est croissante au voisinage de *x*0, ce qui veut dire que le minimum *x**m**i**n* est forcément inférieur à *x*0.

On définit alors *x*1=*x*0−*λ**f*′(*x*0), où *λ* est appelé **le pas de descente**. Dans le contexte de l'algorithme du Perceptron et du *deep learning* en général, *λ* est appelé **taux d'apprentissage** (**learning rate** en anglais).

On répète l'opération jusqu'à obtenir un point *x**k* tel que *f*′(*x**k*)<*t**o**l* où *t**o**l* est la **tolérance**, une constante très petite.

Etape 0 : Définir un point initial *x*0 et une tolérance *t**o**l*.

Etape k : Tant que *f*′(*x**k*)>=*t**o**l* : *x**k*+1=*x**k*−*f*′(*x**k*).

* Que se passe-t-il lorsque le pas de descente est trop petit? Lorsqu'il est trop grand?
* Pour l'initialisation *x*0=−10, trouver le plus petit pas de descente tel que *f*′(*x**k*)≤0.001 au bout de 20 étapes.
* Trouver un pas de descente tel que l'algorithme de descente converge vers le minimum en 1 étape pour toute initialisation *x*0.

from interaction\_dense import show\_gradient\_descent

show\_gradient\_descent()

## Limites de la descente de gradient

Comme vous pouvez le voir, l'algorithme de descente de gradient muni du bon pas de descente est très efficace pour trouver le minimum global d'une fonction. Néanmoins, cet algorithme possède un point faible colossal: **Il n'est efficace que lorsque la fonction à minimiser est strictement convexe, ce qui n'est pas toujours le cas (comme nous allons le voir dans la suite)**.

Dans la figure interactive suivante, nous avons tracé la fonction *f*(*x*)=(*x*5)4+(*x*5)3−6(*x*5)2+1.

Cette fonction contient un minimum global (celui que nous voulons approcher) et un minimum local (que nous voulons éviter).

* Que se passe-t-il si nous appliquons l'algorithme de descente de gradient avec l'initialisation *x*0=11 et un pas de descente de 0,1?
* Que se passe-t-il si nous appliquons l'algorithme de descente de gradient avec l'initialisation *x*0=0 et un pas de descente quelconque?
* Que se passe-t-il si nous appliquons l'algorithme de descente de gradient avec l'initialisation *x*0=−1 et un pas de descente de 0,1?
* Que se passe-t-il si nous appliquons l'algorithme de descente de gradient avec l'initialisation *x*0=−1 et un pas de descente de supérieur à 0,54?

from interaction\_dense import show\_optimization

show\_optimization()

Lorsque la fonction à minimiser n'est pas convexe, l'algorithme de descente de gradient devient imprévisible et ne donne pas de résultats consistants. **Les résultats de l'algorithme sont très sensibles au variation du pas du gradient**.

Dans la grande majorité des cas en *deep learning*, **la fonction de perte à minimiser n'est jamais convexe et l'algorithme de descente de gradient convergera vers un minimum local**.

Malheureusement, l'algorithme de descente de gradient est l'un des seuls algorithmes pouvant être utilisés en pratique car il est le seul algorithme d'optimisation efficace en temps de calcul dont nous disposons.

Comme vous le verrez plus tard dans des modules pratiques, **le pas du gradient est l'un des hyperparamètres les plus influents sur la performance d'un modèle de** *deep learning*.

## Perceptron Multicouche

L'algorithme de Perceptron simple n'est plus utilisé en pratique. L'algorithme du *Support Vector Machine*, aussi connu sous le nom de **Perceptron à stabilité optimale**, est bien plus performant.

L'interêt de l'algorithme du Perceptron vient d'une technique démontrée en 1989 par George Cybenko qui consiste à empiler plusieurs perceptrons qui auront la même entrée sur une couche appelée **couche cachée** (*hidden layer* en anglais).

La sortie de cette couche de perceptrons sera ensuite donnée en entrée à un perceptron qui fera la classification binaire. Ce perceptron forme ce que l'on appelle la **couche de sortie** (*output layer* en anglais).

Un algorithme de ce type s'appelle **Perceptron Multicouche** (*Multilayer Perceptron* en anglais), souvent abrégé par l'acronyme **MLP**.

Dans l'exemple suivant, nous illustrons un MLP avec une couche cachée de 3 perceptrons et une couche de sortie à 1 perceptron. Vous pouvez appuyer sur le bouton *play* pour lancer l'animation, et le bouton *stop* pour la relancer depuis le début.

* Exécuter la cellule suivante pour afficher l'interaction

from interaction\_dense import show\_dense

show\_dense()

Dans un problème de classification, ce type d'algorithme permet d'approximer une frontière de décision non-linéaire, mais pour cela **il faut absolument que les fonctions d'activations des perceptrons de la couche cachée soient non-linéaires**. En pratique, nous utiliserons les fonctions *t**a**n**h* ou *R**e**L**U* définie par *R**e**L**U*(*x*)={*x*0si *x*≥0sinon .

Pour illustrer les effets de ces fonctions d'activation, nous allons appliquer l'algorithme du MLP sur le dataset *moons*.

* Exécuter la cellule suivante pour afficher l'interaction

from interaction\_dense import show\_dataset\_moon

show\_dataset\_moon()

Le comportement de la descente de gradient est imprévisible car la fonction de perte n'est pas convexe. Un pas de gradient trop grand ou trop petit ne permettra pas à l'algorithme de converger vers une solution satisfaisante, même si la solution existe.

### Ce qu'il faut retenir:

* Le produit scalaire est le principal outil que nous utilisons pour faire la classification. **Cette classification est purement géométrique**.
* L'objectif d'un Perceptron est de **trouver un hyperplan qui sépare les différentes classes d'individus**.
* L'atteinte de cet objectif se fait en **minimisant la fonction de perte par descente de gradient**.
* Si la base de données n'est pas linéairement séparable, il n'existe pas forcément un unique minimum global.
* Si la base de données n'est pas linéairement séparable, **il est quand même possible de trouver un hyperplan séparateur non-linéaire en utilisant une approche multicouche.**
* **Un pas de gradient trop grand ou trop petit ne permettra pas à l'algorithme MLP de converger vers une solution satisfaisante**. Trouver le bon pas de gradient et la bonne initialisation est tout le challenge du *deep learning*.

### Fin

Merci d'avoir suivi cet exercice introductif sur le *deep learning*!

Dans le prochain exercice, nous verrons comment créer et entraîner un réseau de neurones sur le problème de classification des chiffres manuscrits avec la package *keras* en Python.

# IV. Prédiction à l’aide des Dense Neural Networks

## Contexte et objectif

Le principal objectif de cet exercice est de créer votre premier réseau de neurones artificiels (*Neural Network*) pour la reconnaissance de chiffres écrits à la main avec le module **Keras**. Le module **Keras** est accompagné d'une riche documentation que vous pouvez consulter [ici](https://keras.io/#you-have-just-found-keras)).

Nous allons construire un réseau de neurones artificiels simple avec deux couches de neurones cachées (*hidden layers*). On rappelle que les couches cachées sont toutes les couches qui se trouvent entre la couche d'entrée (*input layer*) et la couche de sortie (*output layer*).

## Compétences requises

* Scikit-learn
* Matplotlib
* Pandas pour la Data Science
* Exécutez la cellule ci-dessous pour importer les modules nécessaires à l'exercice.

import numpy as np # Pour la manipulation de tableaux

import pandas as pd # Pour manipuler des DataFrames pandas

import matplotlib.pyplot as plt # Pour l'affichage d'images

from matplotlib import cm # Pour importer de nouvelles cartes de couleur

%matplotlib inline

from keras.models import Sequential # Pour construire un réseau de neurones

from keras.layers import Dense # Pour instancier une couche dense

from keras.utils import np\_utils

import itertools # Pour créer des iterateurs

from sklearn import metrics # Pour évaluer les modèles

* Exécutez la cellule ci-dessous pour charger les échantillons d'entraînement et de test de la base de données MNIST du premier exercice.

# Pour importer le datasets mnist de Keras

from keras.datasets.mnist import load\_data

# Chargement des données MNIST

(X\_train, y\_train), (X\_test, y\_test) = load\_data()

# Changement de forme

X\_train = X\_train.reshape([-1, 28\*28])

X\_test = X\_test.reshape([-1, 28\*28])

# Shape of X\_train and y\_train

print('Shape of X:', X\_train.shape)

print('Shape of y:', y\_train.shape)

Shape of X: (60000, 784)

Shape of y: (60000,)

* Pour une meilleure performance du modèle, réduire les pixels des données **X\_train** et **X\_test** afin qu'ils soient compris entre 0 et 1.

Pour réduire les pixels, il suffit de diviser l'échantillon entier par 255. Ainsi tous les pixels seront compris entre 0 et 1.

X\_train = X\_train / 255

X\_test = X\_test / 255

Transformer les labels de **y\_train** et **y\_test** en vecteurs catégoriels binaires (*one hot*) grâce à la fonction to\_categorical du sous-module **np\_utils** de **keras**.

* Extraire dans des variables appelées respectivement **num\_pixels** et **num\_classes** le nombre de colonnes (pixels) de **X\_train** et le nombre de colonnes (classes) de **y\_test**.

y\_train = np\_utils.to\_categorical(y\_train, dtype = 'int')

y\_test = np\_utils.to\_categorical(y\_test, dtype = 'int')

num\_pixels = X\_train.shape[1]

num\_classes = y\_test.shape[1]

Nous allons construire notre modèle **séquentiellement**, c'est-à-dire que nous allons le construire couche par couche depuis la couche d'entrée jusqu'à la couche de sortie.

La construction séquentielle d'un modèle avec **Keras** se fait très facilement avec les étapes suivantes:

* **Étape 1** : Instancier un modèle avec le constructeur Sequential que nous avons importé.
* **Étape 2** : Instancier les couches qui composeront le modèle avec leur constructeur. Pour instancier une couche dense, il faut utiliser le constructeur Dense, que nous avons aussi importé.
* **Étape 3** : Ajouter les couches au modèle grâce à sa méthode add.

L'instanciation de couches contient plusieurs nuances:

* La première couche que nous allons ajouter au modèle doit être instanciée **en précisant les dimensions du vecteur d'entrée** avec le paramètre **input\_size**. Cette précision n'est pas nécessaire pour les couches suivantes.
* Le **nombre de neurones** dans une couche se définit avec le paramètre **units**.
* L'**initialisation des vecteurs de poids des neurones** se fait avec le paramètre **kernel\_initializer**.
* Pour ajouter une fonction activation à une couche de neurones, on peut soit instancier une couche d'activation puis l'ajouter au modèle, ou bien définir la fonction d'activation dans le paramètre **activation** du constructeur d'une couche.

Plus d'infos sur les layers et leurs paramètres [ici](https://keras.io/layers/core/). Plus d'infos sur les fonctions d'activations [ici](https://keras.io/activations/).

* Instancier un modèle séquentiel qui sera appelé **model** grâce au constructeur Sequential.
* Instancier une couche dense appelée **first\_layer** avec 20 neurones telle que la dimension de son entrée soit le nombre de pixels de chaque image. Cette couche aura comme fonction d'activation la fonction tanh. Le vecteur de poids de cette couche seront initialisés aléatoirement selon la loi normal.
* Instancier une deuxième couche dense appelée **second\_layer** contenant autant de neurones que la base contient de classes, c'est-à-dire 10. Les poids de cette couche seront aussi intialisés aléatoirement selon une loi normal et la couche aura comme fonction d'activation softmax.
* Ajouter au modèle les couches que nous avons instanciées. Il faudra les ajouter dans l'ordre.

# Instanciation du modèle

model = Sequential()

# Instanciation de la première couche dense

first\_layer = Dense(units = 20, # nombre de neurones

input\_dim = num\_pixels, # dimension du vecteur d'entrée

kernel\_initializer ='normal', # loi d'initialisation du vecteur de poids

activation ='tanh') # fonction d'activation

# Instanciation de la seconde couche dense

second\_layer = Dense(units = num\_classes,

kernel\_initializer ='normal',

activation ='softmax')

# Ajout des couches aux modèle

model.add(first\_layer)

model.add(second\_layer)

Notre modèle est maintenant défini.

Avant de pouvoir l'entraîner, il faut configurer son processus d'entraînement avec la méthode compile, qui reçoit **trois arguments**:

* Un **optimiseur** (paramètre **optimizer**) qui définit l'algorithme d'optimisation que nous allons utiliser pour faire la descente de gradient de la fonction de perte.
* Une **fonction de perte** (paramètre **loss**) à optimiser.
* Une **liste de métriques** (paramètre **metrics**) utilisées pour évaluer la performance du modèle au fur et à mesure qu'il s'entraîne. La métrique *'accuracy'* est utilisée pour évaluer la précision de la classification.
* Compiler le modèle grâce à sa méthode compile. La fonction de perte utilisée sera **'categorical\_crossentropy'**, l'optimiseur sera **'adam'** et la métrique sera **['accuracy']**.

Plus d'infos sur la compilation [ici](https://keras.io/getting-started/sequential-model-guide/#compilation)

model.compile(loss = 'categorical\_crossentropy', # fonction de perte

optimizer = 'adam', # algorithme d'optimisation

metrics = ['accuracy']) # métrique d'évaluation

Contrairement aux modèles classiques de Machine Learning, l'entraînement du modèle doit se faire ici par "*batchs*" : On divise la base de données d'entraînement en plusieurs *batchs* (ou lots) de la taille que l'on souhaite sur lesquels nous effectuons la descente de gradient. L'entraînement par *batchs* est nécessaire pour contourner les **problèmes de mémoire** des ordinateurs que nous utilisons de nos jours.

Lorsque toutes les données ont été utilisées, on dit que l'entraînement a complété une "*epoch*". On peut choisir le nombre d'*epochs* sur lesquelles le modèle va s'entraîner.

Par exemple, si nous choisissons de faire notre entraînement sur des *batchs* de taille 200 alors que la base de données contient 2000 entrées, chaque *epoch* consistera à faire la descente de gradient sur 2000/200 = 10 *batchs*.

Malheuresement, il n'y a pas de moyen analytique permettant de déterminer à l'avance le nombre d'*epochs*, l'*optimizer* ou la taille de batch qui donnera le meilleur résultat.

Pour l'instant, la méthode la plus efficace pour choisir est d'entraîner le modèle plusieurs fois en faisant varier un de ces hyperparamètres à la fois et de choisir la combinaison de paramètres qui donne le meilleur résultat sur l'échantillon de validation.

* Entraîner le modèle sur les données X\_train et y\_train grâce à la méthode fit du modèle:
  + L'entraînement devra se faire sur 20 *epochs* (paramètre **epochs**)
  + Les *batchs* devront avoir une taille de 200 (paramètre **batch\_size**)
  + La perfomance du modèle devra être évaluée sur un échantillon de validation contenant 20% des données (paramètre **validation\_split**)
  + La sortie de l'entraînement devra être stockée dans une variable nommée **training\_history**.

Plus d'informations sur les paramètres de la méthode fit [ici](https://keras.io/models/sequential/).

training\_history = model.fit(X\_train, y\_train, # données d'entraînement

epochs = 20, # nombre d'epochs

batch\_size = 200, # taille des batchs

validation\_split = 0.2) # proportion de l'échantillon de validation

La méthode fit renvoie en fait un objet de la classe **History**. Cet objet contient de nombreuses informations sur le déroulement de l'entraînement, en particulier les précisions sur les échantillons d'entraînement et de validation à la fin de chaque époque. Nous pouvons ainsi tracer une courbe représentant leur évolution tout au long de l'entraînement.

L'objet **History** dispose d'un attribut **history** qui est un dictionnaire contenant dans ses clés les précisions obtenues par le modèle.

* Lancer la cellule suivante pour stocker les précisions d'entraînement et de validation obtenues pendant l'entraînement.

train\_acc = training\_history.history['accuracy']

val\_acc = training\_history.history['val\_accuracy']

* Tracer l'évolution des précisions tout au long de l'entraînement.

# Labels des axes

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

# Courbe de la précision sur l'échantillon d'entrainement

plt.plot(np.arange(1 , 21, 1),

training\_history.history['accuracy'],

label = 'Training Accuracy',

color = 'blue')

# Courbe de la précision sur l'échantillon de validation

plt.plot(np.arange(1 , 21, 1),

training\_history.history['val\_accuracy'],

label = 'Validation Accuracy',

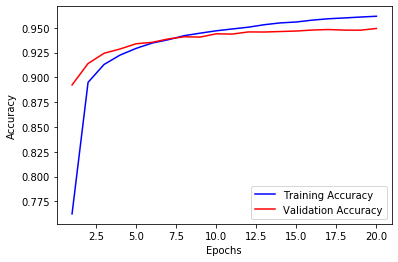
color = 'red')

# Affichage de la légende

plt.legend()

# Affichage de la figure

plt.show()



* Prédire grâce à la méthode predict du modèle la classification de l'échantillon **X\_test** dans un tableau appelée **test\_pred**.
* Évaluer le modèle sur les données de test grâce à sa méthode evaluate. Cette méthode renvoie une liste dont le premier élément est la valeur de la fonction de perte et le second est la précision du modèle.

test\_pred = model.predict(X\_test)

score = model.evaluate(X\_test, y\_test)

score

Nous voulons maintenant effectuer un diagnostique de notre modèle. Pour cela, nous allons calculer une matrice de confusion sur l'échantillon de test.

Néanmoins, les labels de **y\_test** sont des **vecteurs binaires** à cause de la transformation *one-hot* que nous avons effectuée. De plus, si nous essayons de prédire la classe de l'échantillon de test, la méthode predict du modèle renvoie un **vecteur de probabilités** où chaque élément est la probabilité d'appartenance à la classe correspondant à son indice.

Pour utiliser la fonction classification\_report du sous-module **metrics** de **scikit-learn**, il faut que le vecteur de la prédiction et le vecteur de la classe réelle soient composés d'entiers.

Nous allons alors utiliser la méthode argmax d'un *array* **numpy** pour savoir à quelle classe correspondent les vecteurs binaires et les vecteurs de probabilites.

* Prédire les classes de l'échantillon **X\_test** à l'aide de la méthode predict du modèle. Stocker le résultat dans un tableau nommé **test\_pred**.
* Appliquer la méthode argmax sur les tableaux **test\_pred** et **y\_test** pour obtenir des vecteurs d'entiers correspondant aux classes prédites et réelles. Il faudra passer l'argument 'axis = 1' pour que l'argmax soit calculée sur les colonnes et non les lignes. Stocker les sorties des appels de la méthode argmax dans des tableaux nommés **test\_pred\_class** et **y\_test\_class**.
* Afficher un compte-rendu évaluatif détaillé de la perfomance du modèle grâce à la fonction classification\_report du sous-module **metrics** de **scikit-learn**.

test\_pred = model.predict(X\_test)

test\_pred\_class = test\_pred.argmax(axis = 1)

y\_test\_class = y\_test.argmax(axis = 1)

print(metrics.classification\_report(y\_test\_class, test\_pred\_class))

precision recall f1-score support

0 0.96 0.98 0.97 980

1 0.98 0.99 0.98 1135

2 0.95 0.94 0.95 1032

3 0.93 0.94 0.93 1010

4 0.94 0.95 0.95 982

5 0.93 0.92 0.93 892

6 0.95 0.96 0.95 958

7 0.95 0.94 0.94 1028

8 0.93 0.93 0.93 974

9 0.94 0.91 0.92 1009

accuracy 0.95 10000

macro avg 0.95 0.95 0.95 10000

weighted avg 0.95 0.95 0.95 10000

On observe facilement que les chiffres 0, 1 et 6 sont toujours les mieux classés, malgré un écart beaucoup moins important avec les autres chiffres qu'avec la méthode **random forest**.

De plus, la précision a augmenté d'environ 3% par rapport à l'éxercice précedent. Ceci nous invite à considérer les modèles à réseaux de neurones, même simples, sont pour ce problème plus performants que les techniques de **random forest**.

On s'intéresse maintenant aux erreurs réelles du modèle sur l'échantillon de test.

* Calculer et afficher la matrice de confusion entre **y\_test\_class** et **test\_pred\_class**, appelée **cnf\_matrix**, grâce à la fonction confusion\_matrix du sous-module **metrics** de **scikit-learn**.

#Réponse valable:

cnf\_matrix = metrics.confusion\_matrix(y\_test\_class, test\_pred\_class)

print(cnf\_matrix)

###Optionnel: Afficher une matrice de confusion sous forme de tableau coloré

classes = range(0,10)

plt.figure()

plt.imshow(cnf\_matrix, interpolation='nearest',cmap='Blues')

plt.title("Matrice de confusion")

plt.colorbar()

tick\_marks = np.arange(len(classes))

plt.xticks(tick\_marks, classes)

plt.yticks(tick\_marks, classes)

for i, j in itertools.product(range(cnf\_matrix.shape[0]), range(cnf\_matrix.shape[1])):

plt.text(j, i, cnf\_matrix[i, j],

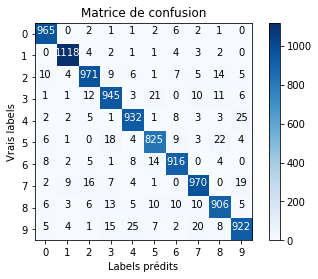
horizontalalignment="center",

color="white" if cnf\_matrix[i, j] > ( cnf\_matrix.max() / 2) else "black")

plt.ylabel('Vrais labels')

plt.xlabel('Labels prédits')

plt.show()



On remarque que les chiffres 5, 3 et 8 sont souvent confondus. Ces chiffres semblent être les points faibles de notre modèle.

Dans l'exercice suivant, nous essaierons d'améliorer notre modèle avec des couches de *convolution*.

# V. Types de couches

L'objectif de ce module est de vous familiariser avec les types de couches les plus utilisés dans la construction de réseaux de neurones profonds.

## Rappel : Les couches *Dense*

Les couches *Dense* ou *Fully-connected* correspondent aux couches que nous trouvons dans le modèle MLP vu dans le module théorique précédent.

Chaque neurone de cette couche possède un vecteur de poids de la même taille que le vecteur qui lui est passé en entrée et un terme de biais.

Les opérations effectuées par cette couche sont:

* **Produit scalaire** entre les vecteur de poids de chaque neurone et le vecteur d'entrée **auquel on ajoute un terme de biais** spécifique à chaque neurone.
* **Activation** des résultats des produits scalaires grâce à une fonction d'activation **non-linéaire** telle que *R**e**L**U*, *t**a**n**h* ou *s**i**g**m**o**i**d*.
* **Concaténation des activations** pour former un nouveau vecteur qui sera passé en **entrée de la couche suivante**.

Dans l'animation interactive suivante, nous avons illustré pour vous chaque étape de ces opérations:

* La couche d'entrée contient 5 neurones, donc la taille du vecteur d'entrée et du vecteur de poids de chaque neurones de la couche dense suivante sera de 5.
* La première couche dense contient 3 neurones, donc la taille de son vecteur de sortie sera de 3 et la taille du vecteur de poids de chaque neurone de la couche dense suivante sera de 3.
* La deuxième couche dense contient 4 neurones, donc la taille de son vecteur de sortie sera de 4.
* Pour interagir avec la figure, cliquez sur le bouton *Next* pour aller à l'étape suivante et *Previous* pour revenir à l'étape précédente.

from interaction\_cnn import show\_dense

show\_dense()

## Les couches de Convolution

Les couches de convolution sont très utilisées pour la *classification d'images*. Avant d'aborder directement les opérations effectuées par une couche de convolution, il est important de comprendre l'opération à la base du neurone de cette couche, **le produit de convolution**.

Le produit de convolution est une opération qui ressemble beaucoup au produit scalaire, sauf qu'il ne s'effectue pas sur un vecteur mais sur une matrice:

D'un côté nous avons une matrice d'entrée, souvent appelée **tuile** ou *convolution patch* en anglais, et d'un autre nous avons une matrice de convolution, souvent appelée **noyau de convolution**, **filtre** ou *convolution kernel* en anglais.

Ces deux matrices **doivent absolument avoir les mêmes dimensions** pour calculer leur produit de convolution.

Les étapes du produit de convolution sont les mêmes que celles du produit scalaire:

* Produit terme à terme des deux matrices.
* Somme des produits.

Dans l'animation interactive suivante, nous illustrons le produit de convolution entre une tuile et un noyau de convolution de dimensions 3x3.

* Pour interagir avec la figure, cliquez sur le bouton *Next* pour aller à l'étape suivante et *Reset* pour redémarrer l'animation.

from interaction\_cnn import show\_one\_operation\_conv

show\_one\_operation\_conv()

Une image est une matrice de pixels. Nous pouvons lui appliquer le produit de convolution.

Pour effectuer cette opération sur une image entière, **il faut d'abord la découper en plusieurs tuiles de mêmes dimensions que le noyau de convolution que nous voulons utiliser.**

Ensuite, nous pouvons effectuer le produit de convolution entre chaque tuile et le noyau.

Enfin, nous allons concaténer tous ces produits pour obtenir une nouvelle image dite *convolée* ou *filtrée*. Lorsque nous appliquons cette technique en *deep learning*, nous allons aussi utiliser une fonction d'activation non-linéaire pour les mêmes raisons que dans l'algorithme MLP.

Dans la figure interactive suivante, nous illustrons la convolution d'une image de dimensions 4x4 par un noyau de convolution de dimensions 3x3:

* Etape 1: Découpage de l'image en plusieurs tuiles convolables. Chaque tuile a une couleur différente.
* Etape 2: Produits de convolutions entre chaque tuile et le noyau.
* Etape 3: Activation et concaténation des produits en une nouvelle matrice.
* Exécuter la cellule suivante pour afficher l'interaction

from interaction\_cnn import show\_operation\_conv

show\_operation\_conv()

Pour quelle raison faisons-nous cette opération? Quels sont les avantages à utiliser cette technique?

Comme vous allez le voir, la convolution d'une image est très facilement interprétable.

Dans la figure interactive suivante, nous illustrons la convolution d'une photo du Taj-Mahal par un noyau de convolution de dimensions 3x3.

La photo est en noir et blanc pour qu'elle corresponde à une matrice de pixels simple, mais l'opération peut être réalisée sur les matrices des composantes rouges, vertes et bleues d'une photo en couleur.

* Pour interagir avec cette figure, vous pouvez soit sélectionner un des noyaux de convolution de la liste, soit modifier manuellement les coefficients du noyau.

from interaction\_cnn import show\_conv

show\_conv()

La convolution d'une image peut être vue comme une extraction de features. Par exemple, les noyaux servant à détecter les bords pourraient être utiles pour classifier certaines formes géométriques.

L'intérêt des neurones de convolution est que par l'entraînement il pourront trouver eux mêmes les **meilleurs noyaux de convolution à utiliser pour extraire des features à partir d'une image**.

## Couches de Régularisation

Les réseaux de neurones contiennent **énormément** de paramètres, en particulier les couches denses. C'est pour cette raison qu'**ils sont très susceptibles au surapprentissage**, c'est-à-dire qu'ils auront une très bonne performance sur l'échantillon d'entraînement, mais cette performance ne pourra pas se généraliser sur l'échantillon de test.

De plus, dans certains problèmes comme la classification d'images ou de sons, le nombre de variables est tellement élevé que les données se retrouvent éparpillées dans un espace de très grande dimension. Il est alors très difficile d'entraîner un modèle sur des données ayant autant de variance. Ce problème est connu comme le **fléau** ou **la malédiction de la dimensionnalité** (*Curse of Dimensionality*).

Pour cela il existe plusieurs techniques spécifiques au *deep learning* qui permettront de réduire simultanément le nombre de variables et le nombre de paramètres du modèle tout en préservant au maximum l'essentiel des caractéristiques des données et la performance du modèle.

Ces techniques s'appellent des **techniques de régularisation**.

Les opérations que nous verrons dans la suite sont souvent illustrées par des "couches" car elles se font sur la sortie d'une couche de neurones et le résultat sera transmis à la couche de neurones suivante.

## Couche de *Max-Pooling*

La couche de *Max-Pooling* s'utilise lorsque les données sur lesquelles nous travaillons sont des matrices. Le concept du *max-pooling* est très simple:

* La matrice est découpée en plusieurs petites tuiles contenant des valeurs de la matrice.
* De chaque tuile on extrait la **valeur maximale**.
* On recompose une matrice ne contenant que les valeurs maximales de chaque tuile.

Nous avons illustré l'opération de *max-pooling* **avec une taille de tuile 2x2** dans la figure interactive suivante. La matrice contient des valeurs représentées par la couleur bleue. Plus la cellule de la matrice est claire, plus sa valeur est élevée.

* Pour interagir avec cette figure, cliquer sur le bouton *Next* pour aller à l'étape suivante et *Previous* pour aller à l'étape précédente. Le bouton *Shuffle Colors* vous permet de changer au hasard les valeurs de la matrice.

from interaction\_cnn import show\_maxpool

show\_maxpool()

## Couche de *Average-Pooling*

Comme pour le *max-pooling*, l'*Average-Pooling* s'effectue sur des matrices. L'opération se déroule ainsi:

* La matrice est découpée en plusieurs petites tuiles contenant des valeurs de la matrice.
* De chaque tuile on extrait la **moyenne**.
* On recompose une matrice ne contenant que les moyennes de chaque tuile.

Comme pour la couche précédente, nous illustrons l'opération de *average-pooling* avec une taille de tuile 2x2 avec la figure interactive ci-dessous. Pour mieux illustrer le concept de moyenne, nous représentons les valeurs de la matrice par des couleurs de toutes les régions du spectre.

* Pour interagir avec cette figure, cliquer sur le bouton *Next* pour aller à l'étape suivante et *Previous* pour aller à l'étape précédente. Le bouton *Shuffle Colors* vous permet de changer au hasard les couleurs de la matrice.

from interaction\_cnn import average\_pool

average\_pool()

## Couche de *Dropout*

La technique du *Dropout* (ou Abandon en français) consiste à couper certaines connexions entre neurones de couches consécutives. Cette technique réduit considérablement la quantité de paramètres à entraîner et permet de "renforcer" les liens entre neurones consécutifs

La quantité de connexions conservées dépend d'un paramètre **p** qui définit la **proportion de connexions à être gardées**. Chaque couche de *dropout* d'un modèle est définie par ce paramètre.

Dans la figure interactive suivante, nous illustrons deux couches de dropout entre 3 couches denses. La première va garder 4/5 des connexions tandis que la deuxième va en garder 2/3.

* Pour interagir avec cette figure, cliquer sur le bouton *Next* pour aller à l'étape suivante et *Previous* pour aller à l'étape précédente.

from interaction\_cnn import show\_dropout

show\_dropout()

# VI. Prédiction à l’aide des CNN

## Contexte et objectif

Le principal objectif de l'exercice est d'apprendre à construire un réseau de neurones convolutif (*Convolutional Neural Network - CNN*) à l'aide du package **Keras**, et de montrer que cette architecture est plus adaptée à notre problème de classification.

La principale différence entre les réseaux classiques (vus à l'étape précédente) et les réseaux de neurones convolutifs réside dans le type d'input et dans la transition entre les couches.

Les CNN ont été construits pour traiter de la donnée de type **image** et les couches de convolution sont aujourd'hui vues comme les **meilleurs extracteurs de features** pour des problèmes de classification liés à l'image.

Nous allons construire un réseau de neurones convolutif comprenant plusieurs types de couches: des couches de **convolution** ainsi que des couches de **pooling** et de **dropout** pour régulariser notre modèle.

## Compétences requises

* Scikit-learn
* Matplotlib
* Pandas pour la Data Science
* Exécutez la cellule ci-dessous pour importer les modules nécessaires à l'exercice.

%matplotlib inline

import numpy as np

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Dropout

from keras.layers import Flatten

from keras.layers.convolutional import Conv2D

from keras.layers.convolutional import MaxPooling2D

from keras.utils import np\_utils

from sklearn import metrics

import matplotlib.pyplot as plt

from matplotlib import cm

import itertools

* Récupérer les données d'entraînements et de validations en exécutant la cellule suivante.

# Pour importer le datasets mnist de Keras

from keras.datasets.mnist import load\_data

# Chargement des données MNIST

(X\_train, y\_train), (X\_test, y\_test) = load\_data()

# Changer la forme de X\_train et X\_test

X\_train = X\_train.reshape([-1, 28\*28])

X\_test = X\_test.reshape([-1, 28\*28])

# Shape of X\_train and y\_train

print('Shape of X:', X\_train.shape)

print('Shape of y:',y\_train.shape)

Shape of X: (60000, 784)

Shape of y: (60000,)

De manière générale les images en couleurs sont composées de 3 canaux rouge vert et bleu de 28x28 pixels superposés les uns sur les autres. Nos images étant en niveaux de gris, elles n'ont qu'un seul canal (profondeur = 1).

* Transformer les données X\_train en un tableau à 4 dimensions (nb\_images, largeur, hauteur, profondeur). Chacune des images sera ainsi redimensionnée au format (28,28,1).
* Faire de même pour les données X\_test.

X\_train = X\_train.reshape((-1, 28, 28, 1))

X\_test = X\_test.reshape((-1, 28, 28, 1))

* Diviser les pixels des données X\_train et X\_test par 255 afin qu'ils soient compris entre 0 et 1.
* Transformer les labels de y\_train et y\_test en vecteurs catégorielles binaires (*one hot*), grâce à la fonction to\_categorical du sous-module np\_utils de keras.

X\_train = X\_train / 255

X\_test = X\_test / 255

y\_train = np\_utils.to\_categorical(y\_train)

y\_test = np\_utils.to\_categorical(y\_test)

## Construction de l'architecture CNN

On rappelle qu'un neurone de convolution contient un noyau avec lequel il effectue le produit de convolution avec les différents *patchs* de l'image.

Dans une couche de neurones de convolution, chaque neurone possède son propre noyau de convolution. L'entraînement de cette couche consiste à trouver les noyaux les plus performants pour la classification.

En général, les couches de convolution sont suivies par des couches de *pooling* afin de réduire progressivement la dimension de l'entrée. Ceci permet à la fois de régulariser le modèle (en diminuant les risques d'overfitting) et d'améliorer la performance de l'algorithme en termes de temps de calcul et d'utilisation de mémoire.

Pour ajouter une couche de convolution à un modèle séquentiel, il suffit comme pour les couches denses de l'instancier avec le constructeur correspondant. Pour les couches de convolution, il s'agit du constructeur Conv2D qui contient différents paramètres:

* **filters** : Un entier correspondant au nombre de matrices de convolution dans la couche.
* **kernel\_size** : Un couple d'entiers correspondant aux dimensions du noyau de convolution.
* **padding** : détermine si le noyau a le droit de dépasser le bord de l'image.
* **input\_shape** : n-uplet correspondant aux dimensions de l'image d'entrée. Doit absolument être précisé dans la première couche.
* **data\_format** : détermine quelle dimension de l'input correspond à quelle dimension de l'image.
* **activation** : fonction d'activation qui sera appliquée à chaque case de la sortie du neurone.
* Instancier un nouveau modèle séquentiel appelé **model** à l'aide du constructeur Sequential.
* Instancier une couche de convolution appelée **first\_layer** à l'aide du constructeur Conv2D avec les paramètres suivants:
  + filters = 32 : La couche sera composée de 32 matrices de convolution.
  + kernel\_size = (5, 5) : le noyau de convolution sera de dimension 5x5 (25 neurones par filtre).
  + padding = 'valid' : pour que le noyau ne puisse pas dépasser les bords de l'image.
  + input\_shape = (28, 28, 1) : l'image passée en entrée aura 28 pixels de hauteur, 28 pixels de largeur et 1 canal.
  + activation = 'relu'.
* Instancier une couche de *max-pooling* appelé **second\_layer** à l'aide du constructeur MaxPooling2D avec le paramètre pool\_size = (2, 2) pour que le maximum soit calculé sur des morceaux de dimensions 2x2.
* Ajouter les couches au modèle à l'aide de sa méthode add. Il faudra les ajouter dans l'ordre.

model = Sequential()

first\_layer = Conv2D(filters = 32,

kernel\_size = (5, 5),

padding = 'valid',

input\_shape = (28, 28, 1),

activation = 'relu')

second\_layer = MaxPooling2D(pool\_size = (2, 2))

model.add(first\_layer)

model.add(second\_layer)

* Instancier une couche de *dropout* appelée **third\_layer** à l'aide du constructeur Dropout avec le paramètre rate = 0.2 pour couper 20% des connexions entre la couche précédente et la couche suivante.
* Instancier une couche d'aplatissement appelée **fourth\_layer** à l'aide du constructeur Flatten. Cette couche ne prend pas de paramètre et permet juste d'aplatir un vecteur en une matrice. Cette transformation est nécessaire pour la suite car les couches denses ne prennent que des vecteurs en entrée et non des matrices.
* Instancier une couche dense appelée **fifth\_layer** à l'aide du constructeur Dense avec 128 neurones et la fonction d'activation *ReLU*.
* Instancier une couche dense appelée **output\_layer** avec 10 neurones (pour 10 classes d'output) et une fonction d'activation *softmax* pour renvoyer des prédictions de probabilité pour chaque classe.
* Ajouter toutes ces couches au modèle.

third\_layer = Dropout(rate = 0.2)

fourth\_layer = Flatten()

fifth\_layer = Dense(units = 128, activation = 'relu')

output\_layer = Dense(units = 10, activation = 'softmax')

model.add(third\_layer)

model.add(fourth\_layer)

model.add(fifth\_layer)

model.add(output\_layer)

* Compiler le modèle en utilisant la fonction de perte 'categorical\_crossentropy', l'algorithme de descente de gradient 'adam' comme optimizer, ainsi que la métrique ['accuracy'].
* Entraîner le modèle avec les données d'entraînement sur 10 *epochs*, des *batchs* de taille 200 et un *split* de validation de 0,2. Stocker la sortie de l'entraînement dans une variable nommée **training\_history**.

model.compile(loss='categorical\_crossentropy', # fonction de perte

optimizer='adam', # algorithme d'optimisation

metrics=['accuracy']) # métrique d'évaluation

training\_history = model.fit(X\_train, y\_train,

validation\_split = 0.2,

epochs = 10,

batch\_size = 200)

* Lancer la cellule suivante pour stocker les précisions d'entraînement et de validation obtenues pendant l'entraînement.

train\_acc = training\_history.history['accuracy']

val\_acc = training\_history.history['val\_accuracy']

* Tracer l'évolution des précisions tout au long de l'entraînement.

# Labels des axes

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

# Courbe de la précision sur l'échantillon d'entrainement

plt.plot(np.arange(1 , 11, 1),

training\_history.history['accuracy'],

label = 'Training Accuracy',

color = 'blue')

# Courbe de la précision sur l'échantillon de validation

plt.plot(np.arange(1 , 11, 1),

training\_history.history['val\_accuracy'],

label = 'Validation Accuracy',

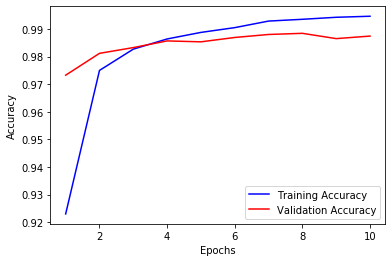
color = 'red')

# Affichage de la légende

plt.legend()

# Affichage de la figure

plt.show()



* Prédire les classes de l'échantillon **X\_test** à l'aide de la méthode predict du modèle. Stocker le résultat dans un tableau nommé **test\_pred**.
* Appliquer la méthode argmax sur les tableaux **test\_pred** et **y\_test** pour obtenir des vecteurs d'entiers correspondant aux classes prédites et réelles. Il faudra passer l'argument 'axis = 1' pour que l'argmax soit calculée sur les colonnes et non les lignes. Stocker les sorties des appels de la méthode argmax dans des tableaux nommés **test\_pred\_class** et **y\_test\_class**.
* Afficher un compte-rendu évaluatif détaillé de la performance du modèle grâce à la fonction classification\_report du sous-module **metrics** de **scikit-learn**.

test\_pred = model.predict(X\_test)

test\_pred\_class = test\_pred.argmax(axis = 1)

y\_test\_class = y\_test.argmax(axis = 1)

print(metrics.classification\_report(y\_test\_class, test\_pred\_class))

precision recall f1-score support

0 0.98 0.99 0.99 980

1 0.99 0.99 0.99 1135

2 0.98 0.98 0.98 1032

3 0.99 0.99 0.99 1010

4 0.99 0.99 0.99 982

5 0.99 0.99 0.99 892

6 0.99 0.99 0.99 958

7 0.98 0.99 0.99 1028

8 0.99 0.98 0.98 974

9 0.99 0.97 0.98 1009

accuracy 0.99 10000

macro avg 0.99 0.99 0.99 10000

weighted avg 0.99 0.99 0.99 10000

La performance de notre modèle est encore plus robuste que celle du modèle dense de l'exercice précédent. La précision est maintenant de 99%, soit 4% de plus que le modèle précédent.

* Calculer et afficher la matrice de confusion entre **y\_test\_class** et **test\_pred\_class**, appelée **cnf\_matrix**, grâce à la fonction confusion\_matrix du sous-module **metrics** de **scikit-learn**.

#Réponse valable:

cnf\_matrix = metrics.confusion\_matrix(y\_test\_class, test\_pred\_class)

print(cnf\_matrix)

###Optionnel: Afficher une matrice de confusion sous forme de tableau coloré

classes = range(0,10)

plt.figure()

plt.imshow(cnf\_matrix, interpolation='nearest',cmap='Blues')

plt.title("Matrice de confusion")

plt.colorbar()

tick\_marks = np.arange(len(classes))

plt.xticks(tick\_marks, classes)

plt.yticks(tick\_marks, classes)

for i, j in itertools.product(range(cnf\_matrix.shape[0]), range(cnf\_matrix.shape[1])):

plt.text(j, i, cnf\_matrix[i, j],

horizontalalignment="center",

color="white" if cnf\_matrix[i, j] > ( cnf\_matrix.max() / 2) else "black")

plt.ylabel('Vrais labels')

plt.xlabel('Labels prédits')

plt.show()

Le modèle a toujours un peu de mal avec les chiffres 3 et 5, mais si on regarde les images sur lesquelles le modèle s'est trompé, il devient difficile de lui en vouloir.

* Lancer la cellule suivante pour afficher des images sur lesquelles le modèle s'est trompé. Vous pouvez la relancer plusieurs fois pour afficher d'autres images.

error\_indexes = []

for i in range(len(test\_pred)):

if (test\_pred\_class[i] != y\_test\_class[i]):

if(y\_test\_class[i] == 5 or y\_test\_class[i] == 3):

if(test\_pred\_class[i] == 5 or test\_pred\_class[i] == 3):

error\_indexes += [i]

j = 1

for i in np.random.choice(error\_indexes, size = 3):

img = X\_test[i]

img = img.reshape(28, 28)

plt.subplot(1, 3, j)

j = j + 1

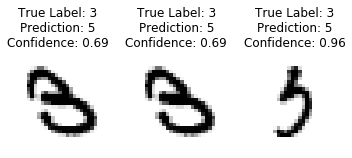
plt.axis('off')

plt.imshow(img, cmap=cm.binary, interpolation='None')

plt.title('True Label: ' + str(y\_test\_class[i]) \

+ '\n' + 'Prediction: '+ str(test\_pred\_class[i]) \

+ '\n' + 'Confidence: '+ str(round(test\_pred[i][test\_pred\_class[i]], 2)))



# VII. Résolution avec l’architecture LeNet

## Contexte et objectif

Le principal objectif de l'exercice est de réaliser une reproduction de l'algorithme de Réseau de Neurones Convolutif **LeNet5** sur Python avec le module Keras. Plus d'infos sur LeNet [ici](http://yann.lecun.com/exdb/lenet/).

L’architecture LeNet a été introduite par Yann LeCun en Novembre 1998 dans le journal [*Proceedings of the IEEE*](http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf). Nous allons reconstruire l'architecture du réseau LeNet qui est plus élaborée que celle créée dans l’exercice précédent. Le réseau LeNet comprend notamment deux couches de convolution, deux couches de *pooling*, ainsi que des layers de *dropout*.

* Exécutez la cellule ci-dessous pour importer les modules nécessaires à l'exercice.

%matplotlib inline

import numpy as np

from keras.models import Sequential

from keras.layers import Dense, Activation

from keras.layers import Dropout

from keras.layers import Flatten

from keras.layers.convolutional import Conv2D

from keras.layers.convolutional import MaxPooling2D

from keras.utils import np\_utils

import matplotlib.pyplot as plt

from matplotlib import cm

from sklearn import metrics

import itertools

* Exécutez la cellule suivante pour charger les données obtenues à l'Exercice 1.

# Pour importer le datasets mnist de Keras

from keras.datasets.mnist import load\_data

# Chargement des données MNIST

(X\_train, y\_train), (X\_test, y\_test) = load\_data()

# Changer la forme de X\_train et X\_test

X\_train = X\_train.reshape([-1, 28\*28])

X\_test = X\_test.reshape([-1, 28\*28])

# Shape of X\_train and y\_train

print('Shape of X:', X\_train.shape)

print('Shape of y:',y\_train.shape)

Shape of X: (60000, 784)

Shape of y: (60000,)

* Transformer les données **X\_train** en un tableau à 4 dimensions (nb\_images, largeur, hauteur, profondeur). Chacune des images sera ainsi redimensionnée au format (28, 28, 1).
* Faire de même pour les données **X\_test**.

X\_train = X\_train.reshape((-1, 28, 28, 1))

X\_test = X\_test.reshape((-1, 28, 28, 1))

* Diviser les pixels des données **X\_train** et **X\_test** par 255 afin qu'ils soient compris entre 0 et 1.
* Transformer les labels de **y\_train** et **y\_test** en vecteurs catégorielles binaires (*one hot*), grâce à la fonction to\_categorical du sous-module **np\_utils** de **keras**.

X\_train = X\_train / 255

X\_test = X\_test / 255

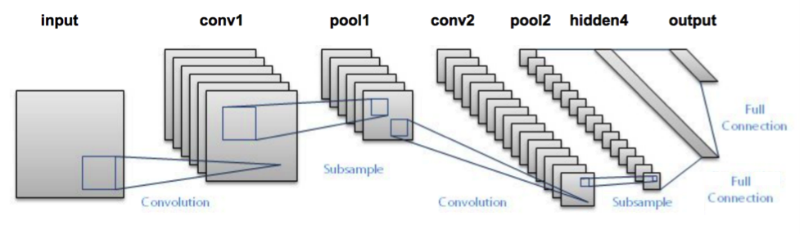
y\_train = np\_utils.to\_categorical(y\_train)

y\_test = np\_utils.to\_categorical(y\_test)

L'architecture du LeNet est constituée des couches suivantes:

* **Convolution 1** : 30 filtres, dimension d'entrée (28, 28, 1), dimension du noyau (5, 5), fonction d'activation *ReLU*, pas de dépassement du noyau.
* **Max-Pooling 1** : dimension du pooling (2, 2).
* **Convolution 2** : 16 filtres, dimension du noyau (3, 3), fonction d'activation *ReLU*, Pas de dépassement du noyau.
* **Max-Pooling 2** : dimension du pooling (2, 2).
* **Dropout** : Connexions coupées: 20%.
* **Aplatissement**
* **Dense 1** : 128 neurones, fonction d'activation *ReLU*.
* **Dense 2** : 10 neurones, fonction d'activation *softmax*.

En image:



* Instancier l'ensemble de ces couches et les ajouter à un modèle séquentiel.
* Compiler le modèle avec la fonction de perte *'categorical\_crossentropy'*, l'optimiseur *'adam'* et la métrique *["accuracy"]*.
* Entraîner le modèle avec les données d'entraînement sur 16 *epochs* avec des *batchs* de taille 200 et un *split* de validation de 0,2. Stocker la sortie de l'entraînement dans une variable nommée **training\_history\_lenet**.

# Architecture du modèle

lenet = Sequential()

conv\_1 = Conv2D(filters = 30, # Nombre de filtres

kernel\_size = (5, 5), # Dimensions du noyau

padding = 'valid', # Mode de Dépassement

input\_shape = (28, 28, 1), # Dimensions de l'image en entrée

activation = 'relu') # Fonction d'activation

max\_pool\_1 = MaxPooling2D(pool\_size = (2, 2))

conv\_2 = Conv2D(filters = 16,

kernel\_size = (3, 3),

padding = 'valid',

activation = 'relu')

max\_pool\_2 = MaxPooling2D(pool\_size = (2, 2))

flatten = Flatten()

dropout = Dropout(rate = 0.2)

dense\_1 = Dense(units = 128,

activation = 'relu')

dense\_2 = Dense(units = 10,

activation = 'softmax')

lenet.add(conv\_1)

lenet.add(max\_pool\_1)

lenet.add(conv\_2)

lenet.add(max\_pool\_2)

lenet.add(dropout)

lenet.add(flatten)

lenet.add(dense\_1)

lenet.add(dense\_2)

# Compilation

lenet.compile(loss='categorical\_crossentropy', # fonction de perte

optimizer='adam', # algorithme de descente de gradient

metrics=['accuracy']) # métrique d'évaluation

# Entraînement\_1

training\_history\_lenet = lenet.fit(X\_train, y\_train, # données

validation\_split = 0.2, # split de validation

epochs = 16, # nombre d'epochs

batch\_size = 200) # taille des batchs

* Lancer la cellule suivante pour extraire de **training\_history\_lenet** les précisions sur les bases d'entraînement et de validation obtenues pendant l'entraînement.

train\_acc\_lenet = training\_history\_lenet.history['accuracy']

val\_acc\_lenet = training\_history\_lenet.history['val\_accuracy']

Nous voulons comparer les trois réseaux que nous avons construits jusqu'à maintenant: le réseau dense, le CNN et LeNet.

* Lancer la cellule suivante pour instancier, compiler et entraîner les modèles des exercices précédents. L'opération peut prendre quelques minutes.

# Réseau Dense

dense = Sequential()

dense\_0 = Flatten()

dense\_1 = Dense(units = 20, input\_dim = 784, kernel\_initializer ='normal', activation ='tanh')

dense\_2 = Dense(units = 10, kernel\_initializer ='normal', activation ='softmax')

dense.add(dense\_0)

dense.add(dense\_1)

dense.add(dense\_2)

# CNN

cnn = Sequential()

cnn\_1 = Conv2D(filters = 32, kernel\_size = (5, 5), padding = 'valid', input\_shape = (28, 28, 1), activation = 'relu')

cnn\_2 = MaxPooling2D(pool\_size = (2, 2))

cnn\_3 = Dropout(rate = 0.2)

cnn\_4 = Flatten()

cnn\_5 = Dense(units = 128, activation = 'relu')

cnn\_6 = Dense(units = 10, activation='softmax')

cnn.add(cnn\_1)

cnn.add(cnn\_2)

cnn.add(cnn\_3)

cnn.add(cnn\_4)

cnn.add(cnn\_5)

cnn.add(cnn\_6)

# Compilation

dense.compile(loss = 'categorical\_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

cnn.compile(loss = 'categorical\_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

# Entraînement

training\_history\_dense = dense.fit(X\_train, y\_train, validation\_split = 0.2, epochs = 16, batch\_size = 200, verbose = 1)

training\_history\_cnn = cnn.fit(X\_train, y\_train, validation\_split = 0.2, epochs = 16, batch\_size = 200, verbose = 1)

Nous avons stocké la sortie de leurs entraînements dans les variables **training\_history\_dense** et **training\_history\_cnn**.

* Extraire de ces deux dictionnaires les précisions sur la base de validation obtenues pendant l'entraînement de chacun des modèles.
* À l'aide du module *matplotlib.pyplot*, tracer les évolutions des précisions sur la base de validation des trois modèles. On rapelle que l'entraînement s'est fait sur *16 epochs* pour les trois modèles.
* Quel modèle vous paraît être le plus performant?

ax2 = plt.subplot(2, 2, 2)

ax2.xlabel('Epochs')

ax2.ylabel('Accuracy')

ax2.title('Modèle Dense')

ax2.plot(np.arange(1, 17, 1),

train\_acc\_dense,

label = 'Training accuracy',

color = 'blue')

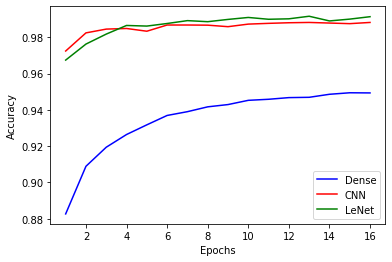
ax2.plot(np.arange(1, 17, 1),

val\_acc\_dense;

label = 'Validation accuracy',

color = 'red')

ax2.legend()



* Prédire les classes de l'échantillon **X\_test** à l'aide de la méthode predict du modèle LeNet. Stocker le résultat dans un tableau nommé **test\_pred\_lenet**.
* Faire de même pour les deux autres modèles et stocker le résultat dans **test\_pred\_dense** et **test\_pred\_cnn**.
* Appliquer la méthode argmax sur les tableaux **test\_pred\_lenet**, **test\_pred\_dense**, **test\_pred\_cnn** et **y\_test** pour obtenir des vecteurs d'entiers correspondant aux classes prédites et réelles. Il faudra passer l'argument 'axis = 1' pour que l'argmax soit calculée sur les colonnes et non les lignes. Stocker les sorties des appels de la méthode argmax dans des tableaux nommés **test\_pred\_lenet\_class**, **test\_pred\_dense\_class**, **test\_pred\_cnn\_class** et **y\_test\_class**.

test\_pred\_lenet = lenet.predict(X\_test)

test\_pred\_dense = dense.predict(X\_test)

test\_pred\_cnn = cnn.predict(X\_test)

test\_pred\_lenet\_class = test\_pred\_lenet.argmax(axis = 1)

test\_pred\_dense\_class = test\_pred\_dense.argmax(axis = 1)

test\_pred\_cnn\_class = test\_pred\_cnn.argmax(axis = 1)

y\_test\_class = y\_test.argmax(axis = 1)

* Afficher un compte-rendu évaluatif détaillé de la performance du modèle **lenet** à l'aide de la fonction classification\_report du sous-module **metrics** de **scikit-learn**.

print(metrics.classification\_report(y\_test\_class, test\_pred\_lenet\_class))

precision recall f1-score support

0 0.99 1.00 1.00 980

1 1.00 1.00 1.00 1135

2 0.99 1.00 0.99 1032

3 0.99 1.00 0.99 1010

4 0.99 0.99 0.99 982

5 1.00 0.99 0.99 892

6 1.00 0.99 0.99 958

7 0.99 0.99 0.99 1028

8 0.99 0.99 0.99 974

9 0.99 0.99 0.99 1009

accuracy 0.99 10000

macro avg 0.99 0.99 0.99 10000

weighted avg 0.99 0.99 0.99 10000

* Lancer la cellule suivante pour voir les images de la base de test où les trois modèles se sont trompés. Vous pouvez la relancer plusieurs fois.

error\_indexes = []

for i in range(len(test\_pred\_cnn)):

if (test\_pred\_lenet\_class[i] != y\_test\_class[i]):

if(test\_pred\_dense\_class[i] != y\_test\_class[i]):

if(test\_pred\_cnn\_class[i] != y\_test\_class[i]):

error\_indexes += [i]

j = 1

for i in np.random.choice(error\_indexes, size = 3):

img = X\_test[i]

img = img.reshape(28, 28)

plt.subplot(1, 3, j)

j = j + 1

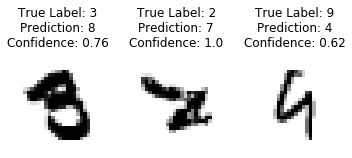
plt.axis('off')

plt.imshow(img,cmap = cm.binary, interpolation='None')

plt.title('True Label: ' + str(y\_test\_class[i]) \

+ '\n' + 'Prediction: '+ str(test\_pred\_lenet\_class[i]) \

+ '\n' + 'Confidence: '+ str(round(test\_pred\_lenet[i][test\_pred\_lenet\_class[i]], 2)))



* On remarque que certains des chiffres mal identifiés sont parfois très mal écrits, certains ne sont même pas reconnaissables.
* Avec un taux de précision dépassant les 99% pour quelques minutes d'entraînement, on peut affirmer qu'au moins un des trois modèles a rempli son objectif.

### Ce qu'il faut retenir :

Les réseaux de neurones construits séquentiellement sont des outils de machine learning désormais accessibles offrant des résultats pouvant surpasser de loin les algorithmes classiques sur des tâches non-triviales.

Le schéma pour implémenter un modèle avec keras est très simple:

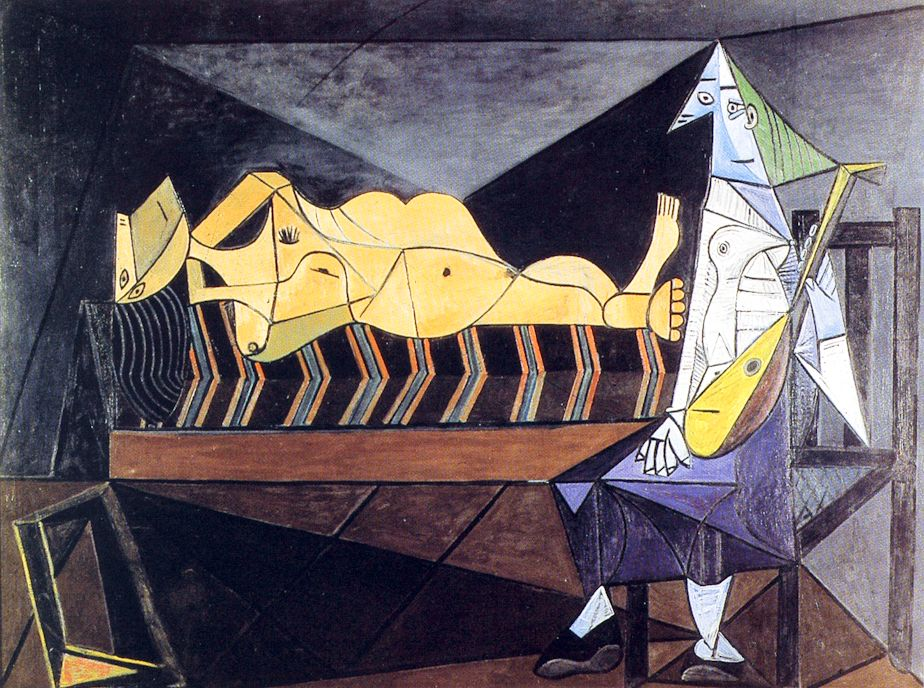
* Architecture du modèle
* Compilation du modèle
* Entraînement du modèle
* Diagnostic de l'entraînement
* Evaluation des prédictions

# VIII. Transfer Learning

## Contexte et objectif

Dans les exercices précédents, nous nous sommes essentiellement intéressés à classifier les chiffres manuscrits. Nous avons pu remarquer qu'un algorithme classique comme un **SVM** ou un **RandomForest** nous donnait une estimation proche de la réalité. Ce n'est plus le cas pour des problèmes de classification plus complexes (ex : classifier des images de chats ou de chiens).

Dans cet exercice, nous allons travailler sur des images de tableaux publiées sur Kaggle. Le jeu de données sous forme de CSV comprend le chemin de l'image du tableau et le nom du peintre.



L'objectif de cet exercice est d'avoir un aperçu plus large du *transfer learning* et des possibilités du Deep Learning. Nous allons effectuer dans un premier temps une exploration des données. Puis, nous allons utiliser un modèle convolutionnel déjà pré-entrainé avec le jeu de données **imageNet**. Dans la conclusion de l'exercice, nous allons tester un modèle hybride combinant une partie du modèle convolutionnel précédent (extraction de caractéristiques) et un SVM (classification).

## Compétences requises

* Scikit-learn
* Matplotlib ou Seaborn
* Pandas pour la Data Science
* Exécutez la cellule ci-dessous pour importer les modules nécessaires à l'exercice.

%matplotlib inline

import numpy as np

import matplotlib.pyplot as plt

import time, cv2

import seaborn as sns

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from keras.layers import Dense, Dropout, Flatten, GlobalAveragePooling2D

from keras.optimizers import Adam

from keras.models import Model, Sequential

from keras.preprocessing.image import ImageDataGenerator

import keras

from keras import backend as K

## Exploration des données

* Importer le fichier *artists.csv* dans un DataFrame appelé artists
* Afficher les 5 premières valeurs.

artists = pd.read\_csv('artists.csv')

artists.head()

|  | **id** | **name** | **years** | **genre** | **nationality** | **bio** | **wikipedia** | **paintings** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0 | Amedeo Modigliani | 1884 - 1920 | Expressionism | Italian | Amedeo Clemente Modigliani (Italian pronunciat... | http://en.wikipedia.org/wiki/Amedeo\_Modigliani | 193 |
| **1** | 1 | Vasiliy Kandinskiy | 1866 - 1944 | Expressionism,Abstractionism | Russian | Wassily Wassilyevich Kandinsky (Russian: Васи́... | http://en.wikipedia.org/wiki/Wassily\_Kandinsky | 88 |
| **2** | 2 | Diego Rivera | 1886 - 1957 | Social Realism,Muralism | Mexican | Diego María de la Concepción Juan Nepomuceno E... | http://en.wikipedia.org/wiki/Diego\_Rivera | 70 |
| **3** | 3 | Claude Monet | 1840 - 1926 | Impressionism | French | Oscar-Claude Monet (; French: [klod mɔnɛ]; 14 ... | http://en.wikipedia.org/wiki/Claude\_Monet | 73 |
| **4** | 4 | Rene Magritte | 1898 - 1967 | Surrealism,Impressionism | Belgian | René François Ghislain Magritte (French: [ʁəne... | http://en.wikipedia.org/wiki/René\_Magritte | 194 |

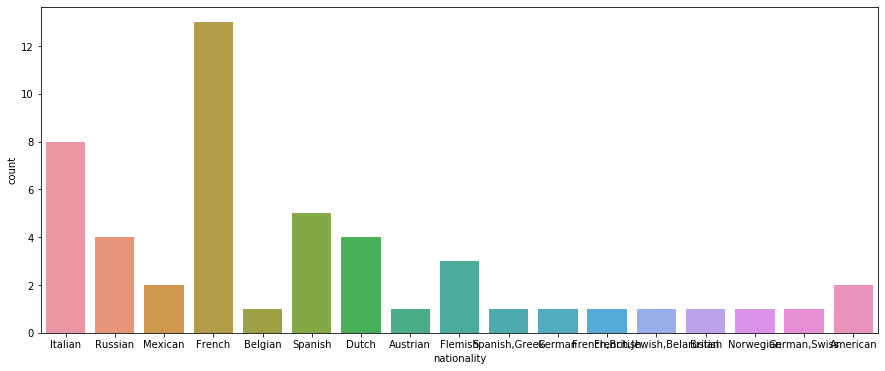
* Afficher le **nombre de peintres** en fonction de leur nationalité sous forme d'un graphique en barre.

plt.figure(figsize=(15,6))

sns.countplot(artists["nationality"]);

# Or

# artists["nationality"].value\_counts().plot.bar()

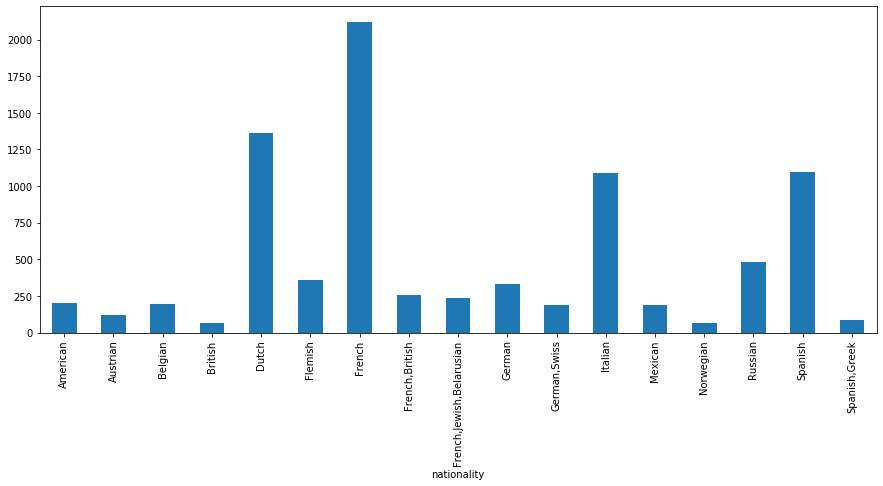


* Afficher le **nombre de tableaux** en fonction de la nationalité des peintres sous forme d'un graphique en barre.

plt.figure(figsize=(15,6))

df2 = artists.groupby("nationality").sum()

df2["paintings"].plot.bar();



* Importer le fichier **file.csv** dans un DataFrame appelé **df**.
* Afficher les 5 premières valeurs.

df = pd.read\_csv('file.csv')

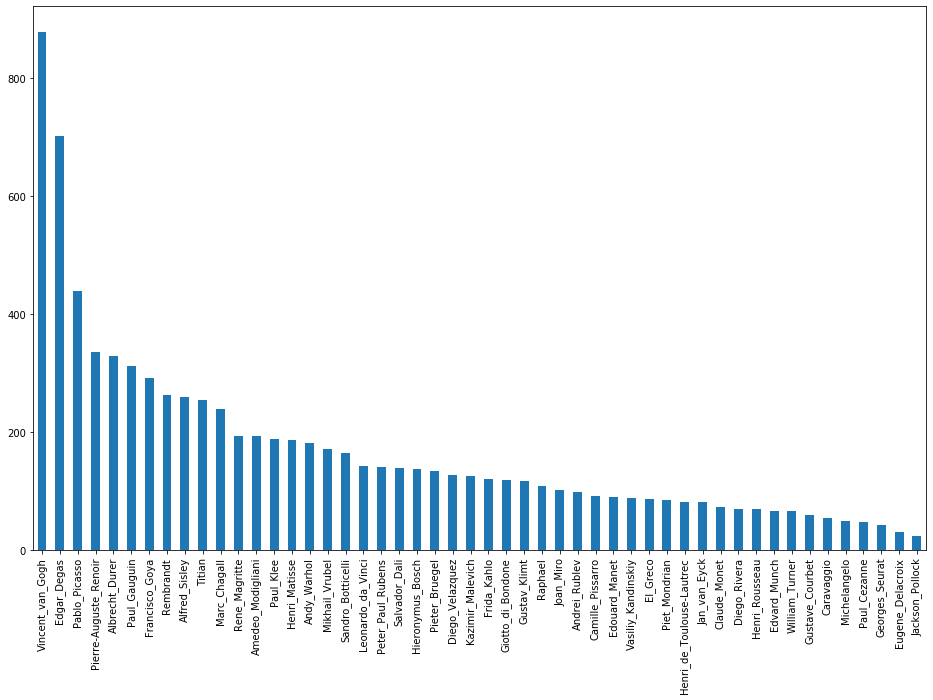
df.head()

|  | **img\_paths** | **artists** | **class** |
| --- | --- | --- | --- |
| **0** | images/Albrecht\_Durer/Albrecht\_Durer\_1.jpg | Albrecht\_Durer | 0 |
| **1** | images/Albrecht\_Durer/Albrecht\_Durer\_10.jpg | Albrecht\_Durer | 0 |
| **2** | images/Albrecht\_Durer/Albrecht\_Durer\_100.jpg | Albrecht\_Durer | 0 |
| **3** | images/Albrecht\_Durer/Albrecht\_Durer\_101.jpg | Albrecht\_Durer | 0 |
| **4** | images/Albrecht\_Durer/Albrecht\_Durer\_102.jpg | Albrecht\_Durer | 0 |

* Afficher le **nombre de tableaux** en fonction du peintre sous forme de graphique en barre.

plt.figure(figsize=(16,10))

df["artists"].value\_counts().plot.bar();



Pour une question de puissance de calcul, nous n'allons que prendre les 10 premiers peintres du jeu de données.

* Définir un nouveau dataframe sous le nom **data** ne prenant que les **10 premières classes**.
* Séparer le jeu de données **df** en un ensemble d'entraînement **data\_train** et en un ensemble de test **data\_test**. Nous choisirons un rapport de 80% pour les données d'entraînements.

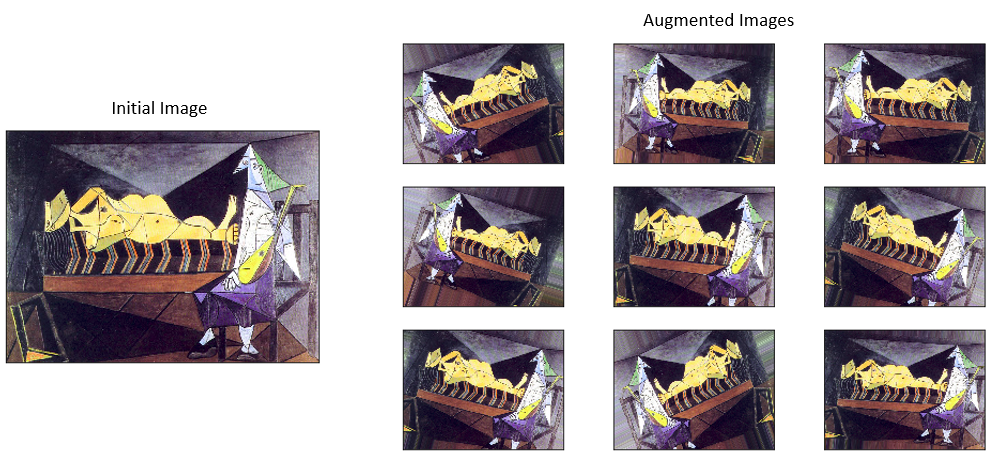
n\_class = 10

data = df[df["class"].isin(range(n\_class))]

data\_train, data\_test = train\_test\_split(data, test\_size=0.2, random\_state=123)

## Générateur de données

L'un des meilleurs moyens d'améliorer la performance d'un modèle de Deeplearning consiste à ajouter davantage de données à l'ensemble d'entraînement. **Dans le cadre des images**, la génération de nouvelles données peuvent être faite en appliquant des transformations géométriques sur nos images (zoom, rotation, redimensionnement, changement de la luminosité ...) originales.



Les générateurs de données permettent de charger les images lorsque le réseau en a besoin, et de leur appliquer des transformations différentes (rotation, zoom, décalage, ...) pour chaque itération. Cette approche permet d'augmenter notre jeu de données et d'éviter au maximum le sur-apprentissage.

La fonction [ImageDataGenerator](https://keras.io/preprocessing/image/) de la classe **keras.preprocessing.image** génère des lots de données d’images avec une augmentation des données en temps réel (transformation). Cette fonction a notamment comme arguments :

* **preprocessing\_function**: la fonction sera exécutée une fois l'image redimensionnée et augmentée. Elle va permettre de prétraiter les images.
* **rotation\_range**: Plage de degrés pour la rotation aléatoire de l'image.
* **width\_shift\_range**: plage pour la translation horizontale de l'image. Si l'argument est inférieur à 1, l'unité de la plage est une fraction de la largeur de l'image. Sinon, l'unité est en pixel.
* **height\_shift\_range**: plage pour la translation verticale de l'image. Si l'argument est inférieur à 1, l'unité de la plage est une fraction de la largeur de l'image. Sinon, l'unité est en pixel.
* **brightness\_range**: Tuple ou liste de deux floats. Plage de sélection d'une valeur de décalage de luminosité.
* **zoom\_range**: Plage de zoom aléatoire. L'argument peut être soit un float ou une liste [inférieur, supérieur].
* **horizontal\_flip**: si True retourne aléatoirement (*p*=12) l'image horizontalement.
* **vertical\_flip**: si True retourner aléatoirement (*p*=12) l'image verticalement.
* Exécuter la cellule suivante pour afficher l'interaction

%matplotlib inline

from interaction\_tl import show\_generator

show\_generator()

* Implémenter sous le nom **train\_data\_generator** un générateur d'image. Les transformations appliquées sur notre image seront :
  + une fonction de prétraitement du modèle VGG16 : preprocess\_input.
  + une rotation aléatoire sur une plage de 10 degrées.
  + une translation verticale et horizontale sur une plage de 10% des dimensions de l'image.
  + un agrandissement sur une plage 10% de l'image.
  + un retournement aléatoirement horizontalement de l'image.
* Implémenter sous le nom **test\_data\_generator** un générateur de test. On appliquera sur notre image uniquement la fonction de prétraitement preprocess\_input.

from keras.applications.vgg16 import preprocess\_input

train\_data\_generator = ImageDataGenerator(

preprocessing\_function = preprocess\_input,

# data augmentation

rotation\_range = 10,

width\_shift\_range = 0.1,

height\_shift\_range = 0.1,

zoom\_range = 1.1,

horizontal\_flip = True

)

test\_data\_generator = ImageDataGenerator(

preprocessing\_function = preprocess\_input)

Dans la cellule précédente, nous avons créé un générateur d'images prenant un lot d'images et leur appliquant des transformations. Nous allons maintenant créer un itérateur permettant de générer à chaque itération (lors de l'apprentissage) un nouveau lot de données. On utilisera la méthode flow\_from\_dataframe de notre générateur d'images. La méthode flow\_from\_dataframe a notamment comme argument :

* **dataframe**: dataframe Pandas contenant le chemin et la classe des images.
* **directory**: chemin du répertoire des images (par défaut: aucun).
* **x\_col**: nom de la colonne du dataframe contenant les chemins des images (par défaut: aucun).
* **y\_col**: nom la colonne du dataframe contenant les classes des images (par défaut: 'class')
* **class\_mode**: le mode d'acception de l'ensemble des classes :
  + 'binary': Tableau numpy 1D binaire des classes. Exemple : [0,1,0,1,1]
  + 'categorical': Tableau numpy 2D du one-hot encoding (dichotomisation) des classes.
  + 'sparse': Tableau numpy 1D du numéro des classes. Exemple : [22,13,1,3]
* **target\_size**: Taille (hauteur, largeur) du redimensionnement des images.
* **batch\_size**: Taille du jeu de données généré à chaque itération.

Cette méthode permettra de charger les images, de générer pour chaque itération des lots de données (batchs) transformés par le générateur d'images.

* Implémenter sous le nom **train\_generator** un iterator à l'aide de la méthode flow\_from\_dataframe de l'objet **train\_data\_generator**. On choisira des images de taille (224,224) et la taille des lots de données (batchs) de 32.

  La colonne **class** des dataframes est sous la forme de numéro de classe.

* Implémenter sous le nom **test\_generator** un iterator à l'aide de la méthode flow\_from\_dataframe de l'objet **test\_data\_generator**. On choisira des images de taille (224,224) et la taille des lots de données (batchs) de 32.

  Pour optimiser le temps de calcul, il est préférable de charger les images en mémoire RAM en utilisant la méthode flow à la place de flow\_from\_dataframe. Mais une telle approche demande d'avoir la mémoire RAM nécessaire.

batch\_size = 32

data\_train["class"] = data\_train["class"].astype(str)

data\_test["class"] = data\_test["class"].astype(str)

train\_generator = train\_data\_generator.flow\_from\_dataframe(dataframe=data\_train,

directory="",

x\_col = "img\_paths",

class\_mode ="sparse",

target\_size = (224 , 224),

batch\_size = batch\_size)

test\_generator = test\_data\_generator.flow\_from\_dataframe(dataframe=data\_test,

directory="",

x\_col = "img\_paths",

class\_mode ="sparse",

target\_size = (224 , 224),

batch\_size = batch\_size)

Found 1181 validated image filenames belonging to 10 classes.

Found 296 validated image filenames belonging to 10 classes.

## Transfer Learning

## Principe

L'apprentissage par transfert est le phénomène par lequel un apprentissage nouveau est facilité grâce aux apprentissages antérieurs partageant des similitudes. Par exemple, les connaissances acquises lors de l’apprentissage de la reconnaissance des voitures peuvent s’appliquer lorsqu’on essaie de reconnaître des camions.

* Exécuter la cellule suivante pour afficher l’interaction.

%matplotlib inline

from interaction\_tl import show\_tl

show\_tl()

Les modèles existants (VGG, ResNet, ...) sont composés de deux grandes parties. La première est un ensemble de convolution permettant l'extraction des features de l'image. La seconde est une succession de dense layer qui a pour but de classifier.

Le **nouveau problème de classification** doit être assez semblable avec le jeu de données utilisé pour le pré-entrainement. Dans ce cas, nous allons initialiser les poids de la partie d'extraction de features de notre modèle par les poids du modèle pré-entrainé. Les couches de la partie de classification seront remplacées et initialisées de manière aléatoire.

Lors du début de l'apprentissage, il est nécéssaire de "freezer" (bloquer) les poids de la partie pré-entrainée puisqu'ils sont proches des poids optimaux. Puis, au cours de l'entraînement, on peut "unfreeze" les couches pour affiner les poids du modèle :

## Modèle

La fonction VGG16 de la classe **keras.applications.vgg16** permet de charger un modèle VGG16. Cette fonction a notamment comme argument :

* **include\_top**: ajouter les couches denses de classification.
* **weights**: si None, les poids sont initialisés de manière aléatoire. Et si **'imagenet'**, les poids sont initialisés au poids de ImageNet.

Un exemple pour créer un modèle VGG16 et "freezer" ses couches:

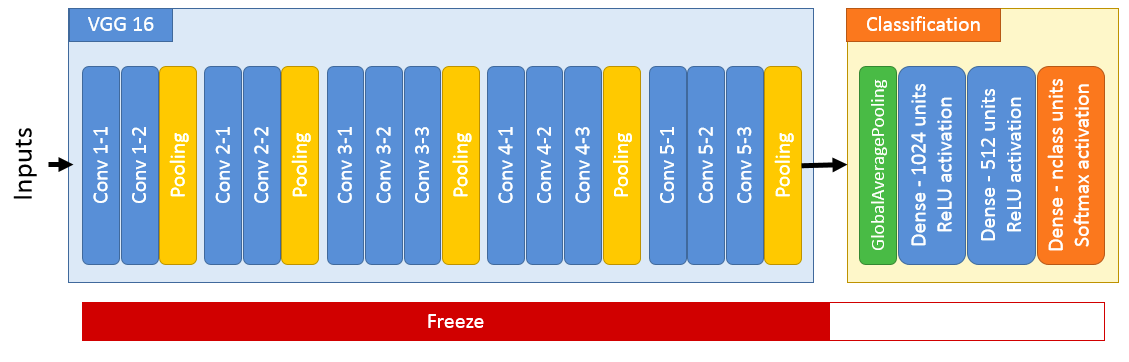
base\_model = VGG16(weights='imagenet', include\_top=False)

for layer in base\_model.layers:

layer.trainable = False

model = Sequential()

model.add(base\_model)



from keras.applications.vgg16 import VGG16

# Modèle VGG16

base\_model = VGG16(weights='imagenet', include\_top=False)

# Freezer les couches du VGG16

for layer in base\_model.layers:

layer.trainable = False

model = Sequential()

model.add(base\_model) # Ajout du modèle VGG16

model.add(GlobalAveragePooling2D())

model.add(Dense(1024,activation='relu'))

model.add(Dropout(rate=0.2))

model.add(Dense(512, activation='relu'))

model.add(Dropout(rate=0.2))

model.add(Dense(n\_class, activation='softmax'))

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['acc'])

Dans les parties précédentes, nous avons implémenté les génerateurs et le modèle. Il ne reste plus qu'à entraîner le modèle.

La méthode fit\_generator appliquée au modèle permet d'entraîner le modèle avec des générateurs. La méthode a comme argument :

* **generator**: Générateur d'entraînement.
* **steps\_per\_epoch**: Nombre d'itération pour chaque époque lors de l'entraînement
* **epochs**: Nombre d'époques.
* **workers**: Nombre de cœurs du processeur utilisés (-1 : tous) pour réaliser l'entraînement.
* **validation\_data**: Générateur pour la validation.
* **validation\_steps**: Nombre d'itération lors du test.
* Entraîner le modèle avec la méthode fit\_generator. Stocker l'historique de l'apprentissage dans la variable **history**.

history = model.fit\_generator(generator=train\_generator,

epochs = 5,

steps\_per\_epoch = len(data\_train)//batch\_size,

validation\_data = test\_generator,

validation\_steps = len(data\_test)//batch\_size

)

* Afficher la courbe de la fonction de coût et de précision en fonction de l'epoch.

plt.figure(figsize=(12,4))

plt.subplot(121)

plt.plot(history.history['loss'])

plt.plot(history.history['val\_loss'])

plt.title('Model loss by epoch')

plt.ylabel('loss')

plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='right')

plt.subplot(122)

plt.plot(history.history['acc'])

plt.plot(history.history['val\_acc'])

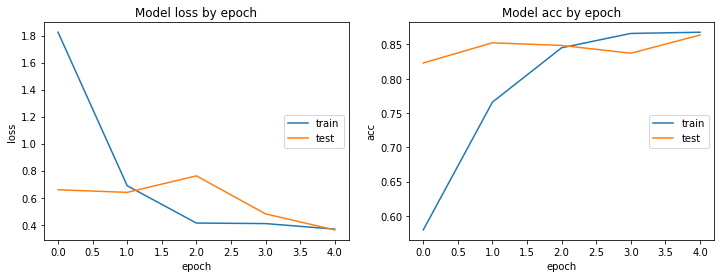
plt.title('Model acc by epoch')

plt.ylabel('acc')

plt.xlabel('epoch')

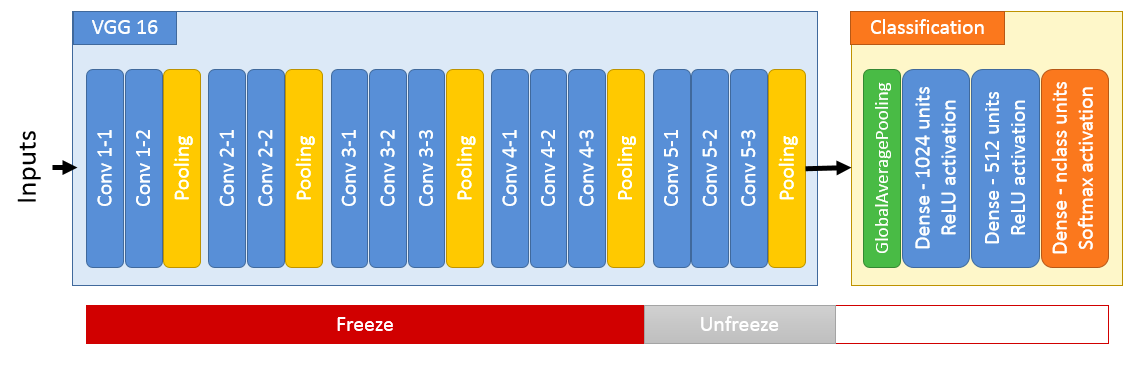
plt.legend(['train', 'test'], loc='right')

plt.show()



Lors de l'apprentissage précédent, nous avons "freezé" les poids du modèle **VGG16 CNN** puisqu'ils étaient proches des poids optimaux et que les poids du classifieur étaient initialisés de manière aléatoire.

Nous allons maintenant rechercher une meilleure solution en reprenant notre modèle appris précédemment et "unfreezer" les couches les unes après les autres du **vgg16**.



Le code suivant permet de "unfreeze" les 4 dernières couches du modèle **base\_model** :

for layer in base\_model.layers[-4:]:

layer.trainable = True

* "Defreezer" les 4 dernières couches du modèle **vgg16**.

for layer in base\_model.layers[-4:]:

layer.trainable = True

A l'aide de la méthode compile appliquée au modèle global, utiliser l'optimizer Adam avec un learning rate **lr** de 10−4. Nous allons toujours utiliser la fonction de coût **'sparse\_categorical\_crossentropy'** et une métrique **['accuracy']**.

* Entraîner le modèle avec la méthode fit\_generator. Stocker l'historique de l'apprentissage dans la variable **history**.

model.compile(optimizer=Adam(lr=1e-4), loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

history = model.fit\_generator(generator=train\_generator,

epochs = 5,

steps\_per\_epoch = len(data\_train)//batch\_size,

validation\_data=test\_generator,

validation\_steps=len(data\_test)//batch\_size

)

* Afficher la courbe de la fonction de coût et de précision en fonction de l'epoch à l'aide de la cellule suivante.

plt.figure(figsize=(12,4))

plt.subplot(121)

plt.plot(history.history['loss'])

plt.plot(history.history['val\_loss'])

plt.title('Model loss by epoch')

plt.ylabel('loss')

plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='right')

plt.subplot(122)

plt.plot(history.history['accuracy'])

plt.plot(history.history['val\_accuracy'])

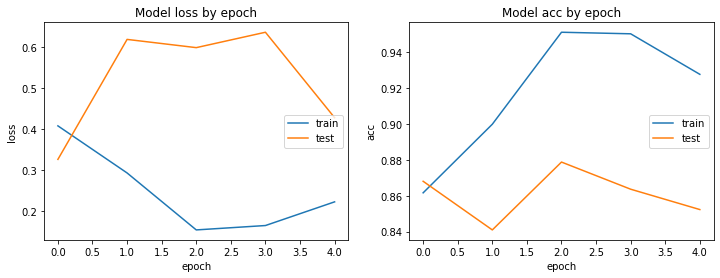
plt.title('Model acc by epoch')

plt.ylabel('acc')

plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='right')

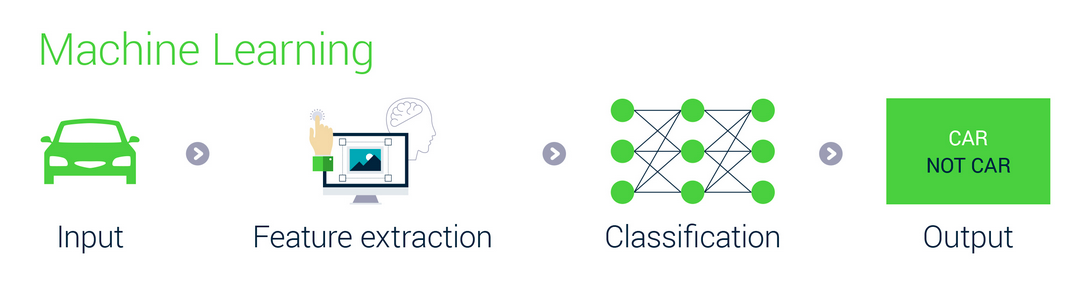
plt.show()



## Features extraction

Avec les méthodes de classifications classiques (SVM, RF ...) donnent de très mauvais scores si nous utilisons les pixels de notre image comme caractéristiques.

Une autre approche combinant les méthodes classiques de machine learning et les méthodes de Deep Learning est l'extraction de features. En utilisant la partie d'extraction des features du modèle de deep learning et en la combinant avec un modèle de classification classique, nous pouvons réussir à avoir de meilleur résultat :



Pour extraire la valeur de sortie d'une couche intermédiaire d'un modèle, une méthode est de définir un nouveau Model en précisant en arguments :

* **input** : l'input de notre ancien modèle avec old\_model.input.
* **output** : l'output d'une couche de notre ancien modèle avec model.layers[num\_layer].output. Puis, on peut extraire la valeur de sortie en utisant la méthode predict sur notre nouveau modèle :

intermediate\_layer\_model = Model(input=model.input, output=model.layers[num\_layer].output)

output\_num\_layer = intermediate\_layer\_model.predict(inputs)

  L'attribut **layers** appliqué au modèle retourne la liste des couches du modèle.

Pour pouvoir extraire les features, nous allons devoir charger les images en mémoire.

* Exécuter la cellule suivante pour charger les images des **n\_class** classes en mémoire.

def convert\_image(X):

X\_img=[]

for image in X:

# Load image

img=cv2.imread(image)

# Resize image

img=cv2.resize(img,(224,224))

# for the black and white image

if img.shape==(224, 224):

img=img.reshape([224,224,1])

img=np.concatenate([img,img,img],axis=2)

# cv2 load the image BGR sequence color (not RGB)

X\_img.append(img[...,::-1])

return np.array(X\_img)

# Load the images train

X\_train\_img = convert\_image(data\_train.img\_paths)

Y\_train = data\_train['class']

# Load the images test

X\_test\_img = convert\_image(data\_test.img\_paths)

Y\_test = data\_test['class']

* En utilisant la méthode d'extraction des features présentée dans la partie précédente, extraire les features de la couche 2 sous le nom **X\_train\_features** et **X\_test\_features**.
* Appliquer un SVM sur ces nouvelles données.

from sklearn.svm import SVC

intermediate\_layer\_model = Model(input=model.input, output=model.layers[2].output)

X\_train\_features = intermediate\_layer\_model.predict(preprocess\_input(X\_train\_img))

X\_test\_features = intermediate\_layer\_model.predict(preprocess\_input(X\_test\_img))

svm = SVC(C=100)

svm.fit(X\_train\_features, data\_train["class"])

svm.score(X\_test\_features,data\_test["class"])

0.9155405405405406

Nous remarquons que notre approche hybride rapporte de meilleures performances. Nous pouvons classifier l'artiste en fonction de ses œuvres.

### Ce qu'il faut retenir :

Les réseaux de neurones construits séquentiellement sont des outils de machine learning désormais accessibles offrant des résultats pouvant surpasser de loin les algorithmes classiques sur des tâches non-triviales.

Le schéma pour implémenter un modèle avec keras est très simple:

* Architecture du modèle
* Compilation du modèle
* Entraînement du modèle
* Diagnostique de l'entraînement
* Evaluation des prédictions

Les **générateurs de données** permettent de charger les images lorsque le réseau en a besoin, et de leur appliquer des transformations différentes (rotation, zoom, décalage, ...) pour chaque itération. Cette approche permet d'augmenter notre jeu de données et d'**éviter au maximum le sur-apprentissage**.

Les réseaux convolutions (CNN) ont deux parties:

* La partie des **convolutions** permet d'extraire les caractéristiques de nos images.
* La partie **Fully Connected Layers** permet de classifier nos images.

Le **transfer learning** consiste à ré-utiliser la partie convolutionnelle d'un modèle pré-entrainer sur une autre tâche de classification.