



TP1 et 2 : Détection d'anomalies :

On a 2 manières de détecter des anomalies :

Détection supervisée

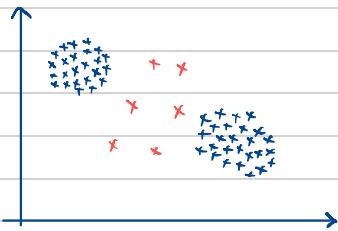
- SVM
- Forêt aléatoire ...

Détection non supervisée

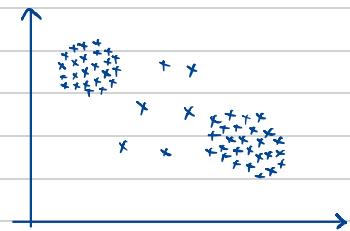
- DBSCAN
- LOF
- One class SVM

Dans le cas où nos données sont à plusieurs dimensions, on va commencer à faire une ACP (que ce soit pour les méthodes supervisées ou non).

La différence entre ces deux approches est la suivante :



Supervisée : On connaît au préalable les classes de nos données

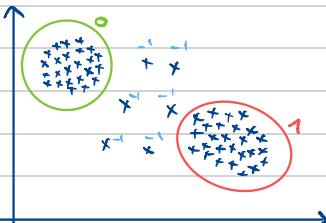


Non supervisée : On ne connaît pas au préalable les classes de nos données.

① DBSCAN :

Concrètement, le principe de DBSCAN est de trouver des clusters de données, associer 1 à la première classe et 0 à l'autre et donner le label -1 aux données bruitées (noise) qui n'appartiennent pas à une de ces 2 classes.

Exemple :



Dans l'exemple des nouvelles que l'on traite, nous ne sommes pas intéressés par diviser les données en classes différentes mais plutôt par différencier anomalies et cas normaux.

Pour ce faire, on dit que les données avec un label ≥ 0 sont normales et les données avec un label < 0 sont anomalies. Ceci nous amène au code suivant :

```
➊ from sklearn.cluster import DBSCAN

# Appel de l'algorithme DBSCAN
clustering = DBSCAN(eps=0.5)
labels = clustering.fit(X).labels_
} appel à l'algorithme Obscan
# Les prédictions sont contenues dans la variable clustering.labels_
# Les anomalies (classe 0) correspondent aux points pour lesquels DBSCAN assigne la valeur -1
y_pred = labels>=0 ↪ prendre uniquement les labels positifs.

# Affichage des trajectoires dans le plan des 2 premiers axes de l'ACP
plt.scatter(X_pca[y_pred==1, 0], X_pca[y_pred==1, 1], color='green', label='Trajectoires normales', alpha=0.5)
plt.scatter(X_pca[y_pred==0, 0], X_pca[y_pred==0, 1], color='red', label='Trajectoires anomalies', alpha=0.5)

# Labels d'axes et titre
plt.xlabel('1e Composante Principale')
plt.ylabel('2e Composante Principale')
plt.title('Classification non supervisée des anomalies par DBSCAN')

# Légende
plt.legend()

# Affichage de la figure
plt.show()
```

② Local Outlier Factor

Comme expliqué plus haut, Obscan n'est pas forcément adapté à la détection d'anomalies. Pour cela on introduit un nouvel algorithme (LOF) expliqué dans le cours et dans le sujet de TP.

La méthode LOF (Local Outlier Factor) vue en cours, permet justement d'améliorer cet aspect. Elle consiste à calculer un score $LOF(x_i)$ associé à chaque donnée. Plus ce score est élevé, plus la donnée a de chance d'être une anomalie.

Pour chaque point x_i on commence d'abord par calculer :

$$\mu(x_i) = \left(\frac{1}{|\mathcal{N}_k(x_i)|} \sum_{x_j \in \mathcal{N}_k(x_i)} d_k(x_i, x_j) \right)^{-1},$$

où $\mathcal{N}_k(x_i)$ désigne les k plus proches voisins de x_i

Puis :

$$LOF_k(x_i) = \frac{\frac{1}{|\mathcal{N}_k(x_i)|} \sum_{x_j \in \mathcal{N}_k(x_i)} \mu(x_j)}{\mu(x_i)}$$

On va procéder de la manière suivante

1. Calculer une matrice des distances de tous les points à tous les points
2. Trouver, pour chaque point, les indices de ses k plus proches voisins
3. Calculer $\mu(x_i)$ pour tous point x_i
4. Calculer $LOF(x_i)$ pour tout point x_i
5. Diviser $LOF(x_i)$ par la valeur LOF maximale de la base de données pour obtenir des valeurs entre 0 et 1.

```
[ ] import numpy as np

def calculate_lof(X, k=5):
    n = X.shape[0]

    # Calcul de la matrice des distances de tous les points à tous les points
    x = X[:, :, 0].np.newaxis - X[np.newaxis, :, 0] # ce code nous donne une
    y = X[:, :, 1].np.newaxis - X[np.newaxis, :, 1] # matrice de la forme :
    x_2 = np.square(x) # matrice avec la distance euclidienne
    y_2 = np.square(y) # telle qu'on la connaît
    D = np.sqrt(x_2 + y_2)
    print(D)

    # Détermination des indices des k plus proches voisins de chaque point
    indices = np.argsort(D, axis=1) # On tire la matrice D pour avoir au départ les distances les plus petites
    kppv = indices[:, 1:k+1] # on ne prend pas la première coordonnée parce qu'il n'en va que des O

    # Calcul de la "reachability distance" ( $\mu$ ) de chaque point
    d = np.zeros(n)
    mu = np.zeros(n)
    for i in range(n):
        d[i] = (1/k) * np.sum(D[i, kppv[i, :]]) # on met 1/k parce que c'est le cardinal de l'ensemble
        mu[i] = 1/d[i]

    # Calcul du LOF pour chaque point
    LOF = np.zeros(n)
    for i in range(n):
        LOF[i] = ((1/k) * np.sum(mu[kppv[i, :]])) / mu[i] # résultat final tel que demandé

    # Normalisation du LOF entre 0 et 1
    max = np.max(LOF)
    lof_normalized = LOF / max

    return lof_normalized
```

Δx et y sont les 2 composantes de l'ACP

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

idem pour y

les commandes permettent de prendre les éléments de matrices aux indices de vecteur $kppv$ et elle renvoie que les éléments. Pas une matrice avec des 0 et les éléments.

③ One Class-SVM :

TP3-4: Réseaux de Neurones:

$$x = \begin{pmatrix} m \\ u \\ \vdots \\ n \end{pmatrix}$$

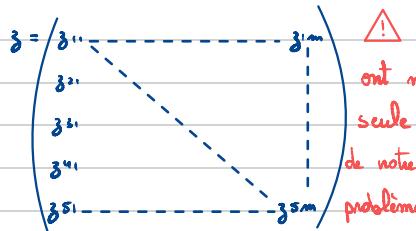
$$W_{xy} \times x \sim \overbrace{5 \times m}^{\text{dimensions}}$$

$$W_{xy} = \begin{pmatrix} 5 \\ 4 \\ \vdots \\ n \end{pmatrix}$$

J. la fonction perte.

$$h = w_0y_0 + w_1x_1 + b$$

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial x}$$



⚠ Dans l'exemple du TP y, z, \dots sont des matrices (i.e. ont m colonnes et pas une seule) car on ne donne pas une seule donnée en entrée mais un batch de données (ex. 10 données de notre ensemble d'apprentissage). Ce qui permet de régulariser le problème.

D'habitude on donne une seule donnée en entrée et $m=1$.

Dans le cas d'une fonction d'activation.

$$\text{linéaire : } \frac{\partial \hat{y}}{\partial z} = 1$$

$$\text{ReLU : } \frac{\partial \hat{y}}{\partial z} = \begin{cases} 1 & \text{si } z > 0 \\ 0 & \text{si } z \leq 0 \end{cases}$$

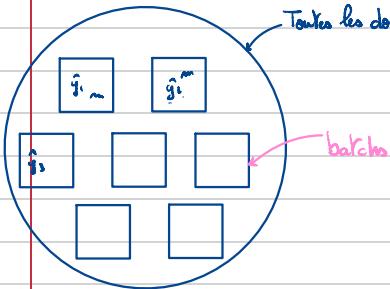
$$\text{sigmoïde : } \hat{y} = \frac{1}{1 + \exp(-z)} \rightarrow \frac{\partial \hat{y}}{\partial z} = \frac{\exp(-z)}{(1 + \exp(-z))^2}$$

Commande python :

Si je veux sommer sur les colonnes d'une matrice M sur python, je fais `S = sum(M, axis=1, keepaxis=True)`

$$M = \begin{pmatrix} m_{11} & \cdots & m_{1n} \\ m_{21} & \cdots & m_{2n} \\ m_{31} & \cdots & m_{3n} \end{pmatrix}$$

$$S = \begin{pmatrix} \sum m_{11} \\ \sum m_{21} \\ \sum m_{31} \end{pmatrix}$$



Toutes les données qu'on a :

batch

Pour entraîner notre modèle, nous devons mettre en place un optimiseur. Nous implémenterons la descente de gradient stochastique avec mini-batch. Il nous faut cependant au préalable implémenter la fonction de coût que nous utiliserons pour évaluer la qualité de nos prédictions.

Pour le moment, nous allons nous contenter d'une erreur quadratique moyenne, qui associe à une fonction d'activation linéaire (l'identité) permet de résoudre les problèmes de régression.

La fonction de coût prend en entrée deux paramètres : la vérité-terrain y_{true} et la prédiction du modèle $y_{pred} (\hat{y})$. Ces deux matrices sont de dimension $n_y \times m$ (où m désigne le nombre d'éléments du batch, et n_y le nombre de neurones de la couche de sortie). La fonction retourne deux grandeurs : J_{mb} , qui correspond à l'erreur quadratique moyenne des prédictions par rapport aux vérités-terrains, et $\frac{\partial J}{\partial \hat{y}}$ au gradient de l'erreur quadratique moyenne par rapport aux prédictions. Autrement dit :

$$\frac{\partial J}{\partial \hat{y}}$$

où \hat{y} correspond à y_{pred} , et J_{mb} à la fonction objectif calculée sur un mini-batch mb de données.

Dans le cas de l'erreur quadratique moyenne, on a :

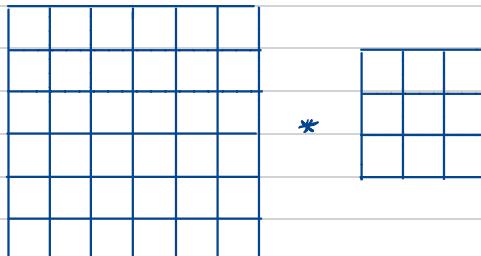
$$J_{mb} = \frac{1}{mn_y} \sum_{i=1}^m \sum_{j=1}^{n_y} (y_{ij} - \hat{y}_{ij})^2$$

et

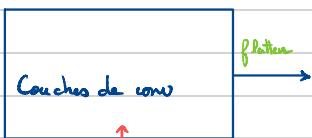
$$\frac{\partial J_{mb}}{\partial \hat{y}} = -\frac{2}{mn_y} (y - \hat{y})$$

Dans cette partie du TP on calcule l'erreur avec une fonction linéaire dans un premier temps pour ensuite calculer avec le log.

TP 6:



①



cette partie comporte
les différentes couches
de réseau de neurones
avec les conv2d et
le max pooling

Après les CNN je renvoie un
vecteur avec toutes les données vectorisées

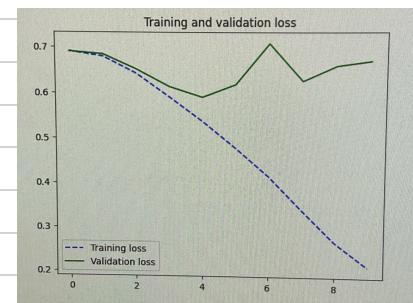
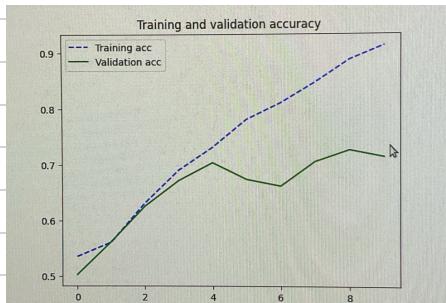
sortie finale (chien ou chat).

ensuite je peux
faire une sorte de couche intermédiaire
pour éviter de passer directement
d'un nombre de sorties très grand
(20k8 p.en) à une seul sortie
qui sera chien ou chat

② Quand on a une classification binaire, il faut utiliser une sigmoïde pour la sortie finale.

Sinon, la sigmoid ne pourra pas être la fonction d'activation de la sortie finale.

Après avoir entraîné nos modèles on observe les courbes suivantes:



On remarque que au départ les données de test collent parfaitement aux données de test mais ensuite ça fait n'importe quoi.

⇒ Ça c'est parce que notre modèle surapprend.

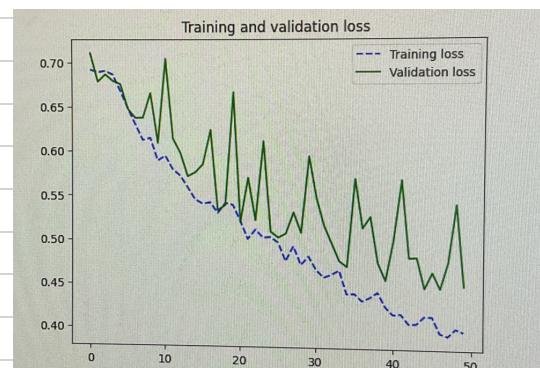
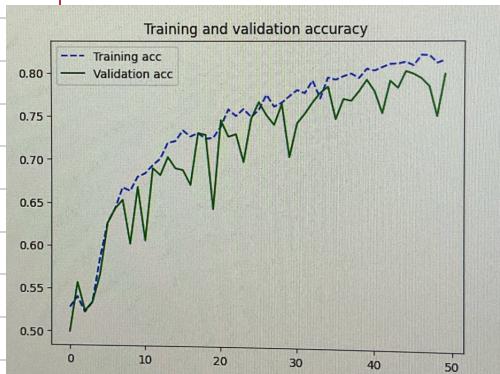
Pour régler les problèmes de surapprentissage on peut :

1. Rajoutez beaucoup de photos en entrée pour augmenter le nombre de données qui entraînent notre modèle.
2. Rajoutez des couches de neurones.

Le problème avec ces deux approches c'est que pour l'une on a besoin d'une base de données très importante, chose à laquelle on a pas facilement accès, et pour l'autre les calculs deviennent plus coûteux.

Par suite, pour régler ce problème on fait ce que on appelle de la "Data augmentation". Ceci veut dire qu'on prend une photo et on la déforme / rotation, décoloration, taille, effet gaufrage ce qui nous fournit une base de photos plus importante sans avoir à récupérer de données.

On remarque qu'en faisant cette petite modification on obtient des résultats beaucoup plus corrects mais qui ne sont pas tout à fait satisfaisants.



L'autre approche pour augmenter l'accuracy ce serait d'utiliser une bibliothèque qui s'appelle VGG.

L'idée ici c'est d'utiliser un réseau de neurones qui existe déjà et qui est entraîné avec des milliards et des milliards de valeurs existantes.

Comment ça marche ?

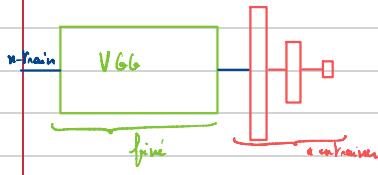


l'idée ici c'est de diviser le réseau de neurones en deux parties, une première qui connaît à donner des x en entrées à vgg et de lui donner des features (voir slide 36 CHV) et une deuxième qui prend ces features et les donne à notre réseau que nous avons créé nous-même et il va renvoyer la y en sortie.

C'est comme si on avait deux parties différentes. (avec l'approche ci-dessous on ne peut pas introduire l'augmentation de données)

On, les résultats obtenus par cette méthode là ne sont pas terrible. Une manière d'améliorer les résultats serait de greffer notre mini réseau à Vgg. On, entraîner Vgg prend des semaines et des mois, chose qu'on a pas forcément envie de faire.

Donc l'idée serait de faire Vgg et d'entraîner seulement notre réseau à nous.



Pour faire ça sur numpy on écrit :

conv_base.trainable = False

c'est ↑ Vgg

Finalement,

la dernière approche c'est d'entraîner quand même Vgg mais pas bc je n'entraîne les dernières couches à ce moment là, on écrit :

conv_base.trainable = True

learning_rate = 1e-5 (On apprend moins avec un taux d'apprentissage bas)

TP7: Classification et génération de texte:

① Dans ce TP, la problématique que nous sommes en train d'étudier est la classification de texte. Ici on devra utiliser un réseau de neurones récursif parce que la taille des mots que nous sommes en train d'apprendre varie. En effet, on peut avoir des mots de quelques lettres et d'autres beaucoup plus grands, contrairement aux images qui ont toutes la même taille.

Le problème de cette approche c'est que l'étude du texte est faite token après token (les tokens peuvent être des lettres, des mots des syllabes...) et cela peut être très long sans qu'on puisse arriver à paralleliser le traitement.

② Dans notre base de données on a plusieurs caractères. On peut avoir l'idée de donner un entier à chaque caractère :

par exemple: $\begin{matrix} a & b & c & d \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 0 & 1 & 2 & 3 \end{matrix}$

On, le problème qui s'impose dans ce genre d'approche c'est que pour l'ordinateur, d est 3 fois plus grand que b , ce qui correctement n'a pas vraiment de sens.

Une approche serait de dire, on considère des vecteurs de \mathbb{R}^n au lieu des entiers. Donc par exemple :

$$a = [1 \ 0 \ 0 \ \dots]$$

$$b = [0 \ 1 \ 0 \ \dots]$$

$$c = [0 \ 0 \ 1 \ \dots]$$

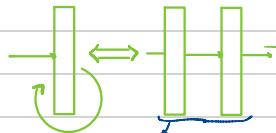
Mais le problème dans cette approche c'est que plus on a de caractères plus les vecteurs seront grands ce qui n'est pas forcément pratique.

La solution à cela est d'introduire une couche d'embedding qui a pour rôle de transformer chaque token en un vecteur de dimension "embedding-size" que nous fixons nous-mêmes.

Notre réseau sera donc composé de :



C'est ce qu'on a expliqué
juste avant



c'est comme ça qu'on modélise
les couches récursives

la couche dense est toujours la
dernière couche du réseau de
neurones, elle prend en entrée tous
les neurones précédents (contrairement
aux autres couches) et fournit la sortie
attendue.

```
Q ✓ 1s ① import tensorflow as tf  
from tensorflow import keras  
from keras import layers  
from keras import models  
  
# A COMPLETER  
model = models.Sequential()  
② model.add(layers.Embedding(VOCAB_SIZE, 128))  
③ model.add(layers.SimpleRNN(128))  
④ model.add(layers.Dense(10, 'softmax'))  
model.summary()  
  
Model: "sequential_1"
```

- 1) C'est la couche d'embedding qui prend en entrée la vocab_size = les nombres de caractères.
- 2) Simple.RNN = c'est la couche recursive
- 3) Dense : 10 = c'est le nombre de sortie (les pays)

softmax : c'est une fonction d'activation qui donne en sortie les probas d'appartenir à chacune des nationalités.

