TP n°5: Automatisation de la détection de chiffres manuscrits

Le compte-rendu doit être rédigé en Jupyter-Notebook, Markdown ou LaTeX. Il doit comporter à la fois les codes mais également **et surtout** *des commentaires et interprétations clairs et pertinents concernant (i) vos choix, (ii) vos résultats en rapport avec les notions vues en cours.*

1 Descriptif du TP

L'objectif du TP est d'aller un cran plus loin que le TP4 en automatisant maintenant la détection des chiffres manuscrits par vos soins à l'aide d'une **fenêtre glissante** qui va (idéalement!) s'arrêter sur chaque chiffre et l'identifier. La difficulté majeure de ce TP ne parait pas évidente à première vue mais elle est fondamentale: comme nous avons entrainé des machines à identifier si la donnée d'entrée ressemble plutôt à un 0 ou à un 1 (ou d'autres chiffres), l'algorithme **doit toujours choisir** parmi 0 ou 1 même quand aucun chiffre n'est présent. Nous aurons donc toujours une détection de chiffre, quoiqu'il arrive!

Deux possibilités s'offrent à nous:

- une approche naïve, celle que nous allons adopter dans un premier temps, consiste à ajouter à la base de données des **faux chiffres** (étiquetés `'z'` par exemple): pour nous il s'agira notamment de chiffres mal centrés que la fenêtre glissante va inévitablement observer et que nous souhaitons exclure, ainsi que de cases vides de chiffres.
- l'approche la plus naturelle cependant consiste à développer un algorithme de **détection** qui évalue la probabilité que l'objet observé est bien un chiffre: cela peut s'effectuer notamment grâce à un SVM simple-classe (*one-class SVM* en anglais). Un SVM simple-classe construit un hyperplan (ou avec un noyau, une surface non-linéaire) au voisinage de tous les points de la classe unique: si un point s'en trouve trop éloigné, il est considéré être en dehors de la classe et donc exclu. Nous pouvons implémenter le SVM simple classe en considérant comme **"classe unique" l'ensemble des chiffres** 0 **et** 1 (ou l'ensemble des chiffres de 0 à 9): tout point distinct d'un chiffre sera alors considéré en dehors de "la" classe. Une fois filtré, il ne reste alors que les points de "la" classe dont on cherchera à détecter la valeur grâce à un algorithme de classification standard.

2 Approche par extension de la base de données

2.1 Extension de la base

Dans cette partie, nous allons créer de fausses images que nous ajouterons à la base de données initiale `X` et que nous étiquèterons `'z'`. Comme d'habitude maintenant, nous commencerons par travailler sur l'ensemble restreint des 0 et 1 avant de généraliser à l'ensemble complet des chiffres.

En vous rappelant de la fonction `roll()` fournie par `numpy` et manipulée au cours du TP1, suggérez une façon de créer très facilement des chiffres mal centrés, tels qu'une fenêtre glissante en rencontrera inévitablement. Il pourra être en particulier judicieux de créer plusieurs niveaux de "décentrage". Ensuite, au moyen de la fonction `concatenate()` de `numpy`, ajoutez à la base de donnée `X` les nouveaux chiffres décentrés ainsi créés et au vecteur `y` associé les valeurs `'z'`.

Si cela vous semble nécessaire, vous pouvez de la même manière ajouter une série d'images totalement noires à la base de donnée `X`: pour cela, on pourra créer plusieurs vecteurs de zéros de taille 784 accolés dans une matrice en utilisant la fonction `zeros((dim_x,dim_y))` fournie par `numpy`.

Astuce: l'opération `['z']*n` *crée le vecteur* `['z',...,'z']` *de taille* `n`.
Attention: la fonction `concatenate` *fonctionne sous le format* `concatenate([A,B,C,...])` *où*

`A`, `B`, *etc., sont des matrices (ou vecteurs) ayant le même nombre de lignes. N'oubliez pas les crochets.*

Avant de pouvoir effectuer notre classification, il nous reste un dernier point à résoudre: les fausses images ont été ajoutées **à la suite des images** de `X`, de sorte que, si l'extraction de `X_train` et `X_test` est effectuée séquentiellement, seul `X_test` contiendra les fausses images. Il nous faut donc "mélanger" la base de données `X`, mais attention, tout en conservant le même ordre de mélange pour `y`. Une possibilité pour cela est d'opérer ainsi: on enregistre la "graine" aléatoire courante fournie par `random.get_state()` de `numpy` (appelons la `seed`), on mélange alors `X` grâce à `random.shuffle(X)` fournie par `numpy` et, avant de mélanger `y` aléatoirement (le mélange serait alors différent!), on récupère la graine sauvegardée via `random.set_state(seed)` de `numpy` et on mélange `y` avec `random.shuffle(y)`.

2.2 Classification par fenêtre glissante

À partir de la nouvelle base de données, commencez par relancer vos classifieurs préférés comme dans les TPs précédents. Attention, si vous avez codé en dur la taille des étiquettes (2 dans le cas binaire, 10 dans le cas complet), il faut désormais vous ajuster, étant donné qu'une nouvelle classe `'z'` a été ajoutée. Vérifiez que les scores d'entrainement et de test obtenus sont toujours bons.

Nous allons maintenant construire la fenêtre glissante: comme au cours du TP précédent, repérez manuellement le coin supérieur gauche (nous l'avions appelé `top_left`) de l'image de vos chiffres manuscrits ainsi que la taille `size` d'un chiffre standard. Effectuez également les opérations de seuillage du fond noir et gardez de côté pour l'instant l'opération de mise à l'échelle de vos chiffres.

Cette fois-ci, au lieu de d'utiliser une boucle `for` qui effectuerait de grands sauts de chiffre en chiffre, nous allons utiliser une boucle `for` qui progresse à petits pas (par exemple de 5 pixels en 5 pixels) et qui:

- estime l'étiquette de la zone courante
- si l'étiquette décidée est `'z'`, se déplace plus loin, sinon identifie le chiffre.

2.3 Visualisation

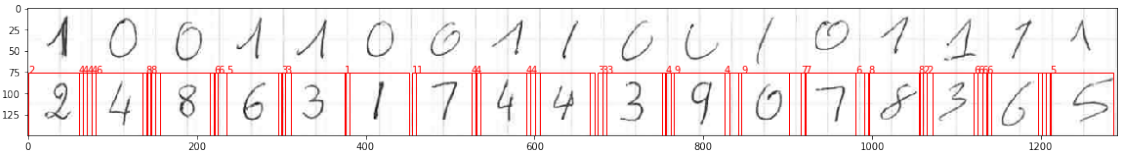
Afin de mieux visualiser les résultats obtenus, nous pouvons ajouter des carrés rouges en surimpression au-dessus de l'image originelle (avant pré-traitement) des chiffres scannés. Pour cela, nous pourrons utiliser la librairie `matplotlib.patches` comme suit:

```
import matplotlib.patches as patches
fig, ax = plt.subplots(figsize=(20,30))

ax.imshow(img.imread('digits.png'),cmap='gray')

xpos = ...
ypos = ...
size = ...
rect = patches.Rectangle((ypos,xpos), size, size, linewidth=1,
    edgecolor='r', facecolor='none')
ax.add_patch(rect)
```

Utilisez cette approche pour ajouter des carrés rouges lorsqu'un chiffre est détecté. On pourra alors ajouter également l'étiquette identifiée au dessus du carré rouge comme suit:

```
label = ...
ax.text(ypos,xpos,label,color='r')
```

Le résultat final pourrait ressembler à cela:



Bien évidemment, commentez et discutez vos résultats! Votre méthode est-elle efficace? Est-elle longue à entrainer ou coûteuse? Quels en sont les défauts majeurs? Comment pourrait-on améliorer les choses?

3 SVM simple-classe

Comme précisé en introduction, la méthode par ajout de fausses images est particulièrement naïve: le cerveau ne détecte pas en permanence des “non chats” ou des “non chiens” ou des “non chevaux”, etc., mais parvient plutôt à (i) détecter ou non un animal puis (ii) à identifier l'animal en question. C'est ce qu'un classifieur **simple-classe** va également faire: entrainé à reconnaitre un “chiffre”, il va accepter ou refuser l'image observée comme tel. Le SVM simple-classe va précisément créer un hyperplan (dans l'espace originel $\mathbb{R}^{784}$ ou dans l'espace des représentations si on utilise un noyau: par exemple $\mathbb{R}^{\infty}$ avec un noyau 'rbf') qui passe au plus près des points de la base de données. Grâce à une fonction de score, le SVM évaluera alors la proximité de tout point $x \in \mathbb{R}^{784}$ (ou sa représentation dans $\mathbb{R}^{\infty}$) à l'hyperplan. Ceci nous permettra alors, par seuillage, d'accepter ou de refuser le point.

Le fonctionnement général du SVM simple-classe sous Python est comme suit:

```
from sklearn.svm import OneClassSVM

# Training
X_train = ...
ocsvm = OneClassSVM(kernel='rbf')
ocsvm.fit(X_train)

# Score
x = ...
ocsvm.score_samples(x)
```

Reprenez votre code précédent de fenêtre glissante mais désormais, **au lieu d'ajouter des images fausses et de filtrer les étiquettes `'z'`**, travaillez directement avec la base de données `(X,y)` originelle et utilisez un SVM simple-classe pour détecter les chiffres identifiés comme tels. Pour y parvenir, il vous faudra fixer (à tâtons) un seuil pour le score `ocsvm.score_samples(x)` pour chaque image `x`: seuls les chiffres associées aux images dont le score est supérieur au seuil seront identifiés.

Comparez les résultats obtenus avec la solution SVM simple-classe et avec l'approche par extension des données, à le fois en termes de performance, de temps de calcul, mais également de pertinence théorique. Quelles solutions conseilleriez-vous à un non-expert qui souhaiterait réaliser un projet similaire?

Comme vous en avez maintenant l'habitude, rassemblez votre code sous une forme compacte, claire, facilement paramétrable et non redondante. L'évaluation CC2 du prochain TP, qui clôt le projet, s'appuiera en grande partie sur le code mis en place ici.