Projet Traitement et Analyse d'Images

IMR3 – 2019-2020

Classification d'images

La révolution numérique conduit à une croissance exponentielle des documents multimédia (texte, images, audio, vidéo). Parmi des contenus, les données de type image sont ubiquitaires et doivent faire l'objet d'une indexation qui n'est plus seulement textuelle, mais qui s'appuie sur des descripteurs (*features*) qui en sont extraits automatiquement. On parle alors d'indexation par le contenu (*Content-Based Image Retrieval* - CBIR). Pour réaliser ce type de traitement automatique, il convient de procéder à la classification des images, qui consiste à extraire suffisamment d'information des primitives de l'image pour pouvoir la catégoriser (ex : chien, chat, voiture, etc.)

La classification est généralement réalisée de manière supervisée. Dans un premier temps, on sélectionne comme référence un (petit) ensemble d'images déjà labellisées (on connaît leur classes d'appartenance) et représentatives, dont on extrait des primitives (couleur, texture, formes) : c'est l'ensemble d'apprentissage (*learning set*). Dans un second temps, on détermine grâce à cet ensemble un modèle de décision, en faisant en sorte que celui-ci soit suffisamment robuste et conduise aussi à de bonnes décisions dans le cas de données déjà labellisées, mais qui n'ont pas servi à bâtir le modèle de décision : c'est l'ensemble de validation (*validation set*). Enfin, quand le modèle a été jugé suffisamment robuste, il peut être appliqué sur un jeu de test (*test set*), contenant des exemples distincts des deux premiers ensembles.

Le but de ce projet est de mettre en œuvre deux approches de la classification d'images.

La première (*old school*) consiste à extraire des attributs (primitives) classiquement utilisés en traitement d'image, comme la couleur, la texture, etc., et de procéder à une classification d'image s'appuyant sur des critères de décision simples, basés sur les distances entre vecteurs d'attributs.

La seconde (*new school*) fera appel à une modélisation de l'information conjointe entre les attributs d'une image et sa classe d'appartenance via un réseau de neurones (*neural network*, NN). Dans ce cas, nous verrons qu'il ne sera pas nécessaire de passer par une étape d'extraction de primitives (elle se fera naturellement dans le NN).

Le projet, d'une durée de 24 heures, sera découpé en quatre phases.

1. Préparation des outils et données

a. Outils : Python + Jupyter + Spyder

La programmation sera réalisée en langage Python. Si ce dernier n'est pas installé sur votre machine, il est conseillé d'installer l'application Anaconda (disponible pour Windows et Linux) qui vous donnera un accès simplifié (via Anaconda Navigator) aux paquets et librairies scientifiques les plus récents de Python. Avec Anaconda, vous pourrez aussi facilement installer et lancer les deux IDE de base que nous utiliserons : Jupyter et Spyder.

Jupyter est un environnement web de développement en Python qui permet, depuis votre navigateur préféré, de créer, manipuler et exécuter des blocs de code Python, tout en permettant l'insertion d'annotations et de rich media de nature variée (image, vidéo, html, LaTeX, etc.). Les fichiers créés, appelés *notebooks*, ont l'extension .ipynb (IPython notebook)

Spyder est quant à lui un IDE standard prenant en charge une multitude de langages de programmation, et permettant entre autres le déboguage. Il est assez simple de transférer du code de Jupyter vers Spyder ou inversement.

Une alternative intéressante à ces deux IDE est la plateforme Google Colab qui permet l'exécution de code Python à la Jupyter, dans le cloud. Elle ne nécessite aucune installation, tout au plus l'existence d'un compte Google.

b. L'environnement et les paquets nécessaires

Sous Anaconda, vous pouvez créer un ou plusieurs environnements de travail, chacun contenant les librairies et les paquets (*packages*) nécessaires à l'application visée. Durant ce projet, il ne sera pas nécessaire de créer un environnement particulier et vous pourrez ajouter les paquets dans l'environnement 'base (root)'.

Les principaux paquets à installer sont :

- Numpy
- Scipy
- Scikit-learn
- Scikit-image
- Matplotlib
- Tensorflow
- Keras

A chaque installation de paquet, Anaconda vérifie les dépendances et leur compatibilité pour une installation optimale. Il se peut que certains paquets manquent pour l'exécution de commandes Python ; dans ce cas il suffira de les installer dans votre environnement de travail, toujours via Anaconda (plus simple).

c. Les données

Il existe sur Internet de nombreuses bases de données d'images utilisables pour un tel projet. La base COREL en est un bon exemple. Elle comprend 10800 images regroupées en 80 groupes thématiques, pour une taille totale de 40 Mo. Les images sont de petites dimensions (80x120 ou 120x80), ce qui permettra de réduire les temps de traitement.

d. *Premiers pas*

Vous trouverez sur Moodle à l'adresse :

https://foad.univ-rennes1.fr/course/view.php?id=1008557

le notebook IPython `first_steps.ipynb` . Chargez-le dans Jupyter, exécutez-le et implémentez vos propres traitements !

2. Préparation des bases d'apprentissage, validation et test

a. *Organisation standard des données pour un problème de classification*

Lorsque l'on veut s'attaquer à un problème de classification de données de type *supervisé*, il faut prendre garde à bien spécifier les différents jeux de données à utiliser pour que la classification soit consistante et suffisamment robuste dans son exploitation.

Ainsi, un ensemble de données étiquetées (labellisées) sera généralement séparé en trois sous-ensembles disjoints (sans doublon) :

- **un ensemble d'apprentissage** : il va servir de référence, soit pour déterminer directement la classe d'appartenance d'un nouvel objet dans le cas des méthodes basées sur les distances entre plus proches voisins (*nearest neighbors* - NN) ; soit pour construire un *modèle* décisionnel qui sera utilisé ultérieurement pour prédire la classe d'un nouvel objet ;

- **un ensemble de validation** : celui-ci va servir à optimiser et valider les règles de décision, qui dépendent en général d'un ensemble de paramètres (p.ex. le nombre *k* de NNs pour les plus proches voisins, ou bien les multiples paramètres nécessaires au calcul d'un modèle SVM ou réseau de neurones) ;

- **un ensemble de test** : ce dernier va servir à évaluer la qualité de la classification obtenue sur des exemples nouveaux, c'est-à-dire n'ayant jamais été présentés au système une fois validé.

b. *Choix de la base d'images*

La base COREL comprend un grand nombre de classes thématiques, aussi il sera judicieux de choisir parmi ces classes un sous ensemble de taille réduite (de 5 à 10 classes parmi les 80 classes de la base, soit de 1000 à 2000 images environ), ce qui permettra de tester assez rapidement les méthodes d'extraction d'attributs et de classification.

☞**Consigne** : préparer les trois ensembles de données sous la forme de répertoires [train, validation, test], en conservant pour chacun le nommage des sous-répertoires correspondant aux classes thématiques (voir Fig. 1).

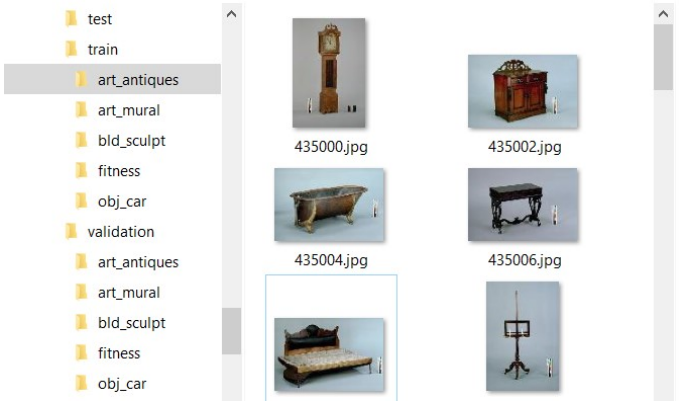

Figure 1 : Organisation des ensembles d'apprentissage (train), validation, et test pour cinq classes issues de la base COREL.

MAJ 2020/06/01

3. Classification d'images basée sur attributs

Télécharger les codes Python disponibles sur :

https://github.com/pochih/CBIR/

Ces codes permettent d'effectuer des tâches de recherche d'images dans une base spécifiée. La recherche est structurée suivant le schéma classique (cf. support cours) :

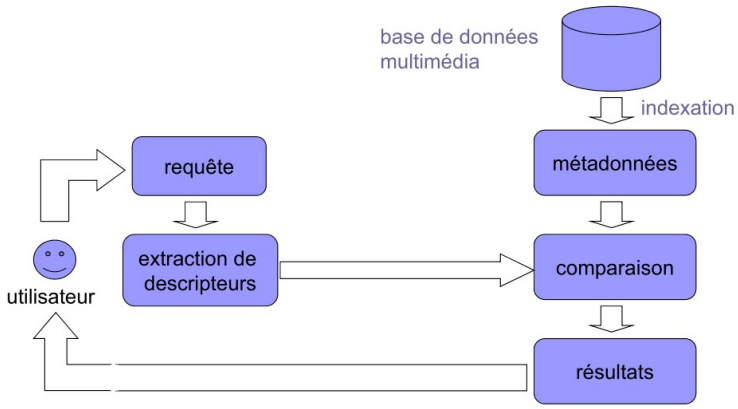

Figure 2 : Principe de la recherche d'images ou de données multimédia.

Chaque instance de la base d'images fait l'objet d'un calcul d'attributs (*features*) suivant différentes méthodes (histogramme des couleurs, contours, daisy, histogramme des gradients, etc.), et le but est de renvoyer à l'utilisateur les images répondant le mieux à la requête. Les attributs basés sur les modèles VGG19 et ResNet ne seront pas étudiés ici.

Avec ces codes Python, le problème est que la base de donnée d'images est unique, et la requête en fait également partie, ce qui ne correspond pas à une situation réaliste. C'est bien pourquoi il faut toujours séparer une base de données en au moins deux sous-ensembles disjoints. Vous pourrez ici

☞**Consignes** : modifier le code de quelques classes de calcul d'attributs (commencer par l'histogramme des couleurs, `color.py`) et des classes dépendantes afin de :

- Prévoir le calcul et le stockage des attributs des images issues de la base d'apprentissage (train) seulement ;
- Transformer le problème de recherche d'images les plus proches d'une requête en problème de classification d'images ; il faudra pour cela modifier le résultat de chaque requête pour lui attribuer une et une seule classe prédite, basé sur le mode pondéré (*weighted mode*) de la réponse à la requête ;
- Etendre cette approche à la classification basée sur plusieurs attributs (code `fusion.py`) ;
- Tester et évaluer plusieurs configurations d'attributs et combinaisons de ceux-ci, le cas échéant en variant les bases de données d'images ;
- Restituer dans une archive l'ensemble de vos codes modifiés et dûment commentés, ainsi qu'une synthèse des résultats obtenus, que vous commenterez. ;

4. Classification d'images basée sur un réseau de neurones

Dans cette partie, il s'agira de mettre en œuvre un réseau de neurones (peu profond), qui prendra en entrée directement des images, sans passer par une étape d'extraction d'attributs.

a. Une première approche empirique d'un réseau de neurones

Afin de vous familiariser avec ce qu'est et comment fonctionne un réseau de neurones, nous vous invitons à consulter (jusqu'au bout) la page web interactive suivante :

http://playground.tensorflow.org

qui résume efficacement une grande partie des concepts qui seront utilisés plus loin pour la classification d'images. Le but est de partitionner au mieux des points en dimension 2 issus d'un ensemble d'apprentissage, en deux classes labellisées, tout en veillant à prédire au mieux la classe d'appartenance des points appartenant à un ensemble de test.

Une bonne règle de classification doit permettre d'éviter :

- **le sur-apprentissage** : en contraignant la méthode pour que la classification des données d'apprentissage suive au plus près les labels qui leur sont affectés, on la rend peu généralisable : appliquée à la base de test, la classification risque d'être peu performante.

- **Le sous-apprentissage** : en diminuant la complexité de l'algorithme ou du modèle de décision, celui-ci sera moins adapté aux données d'apprentissage, et conduira également à de faibles performances en test.

Il existe donc en principe une stratégie optimum de classification.

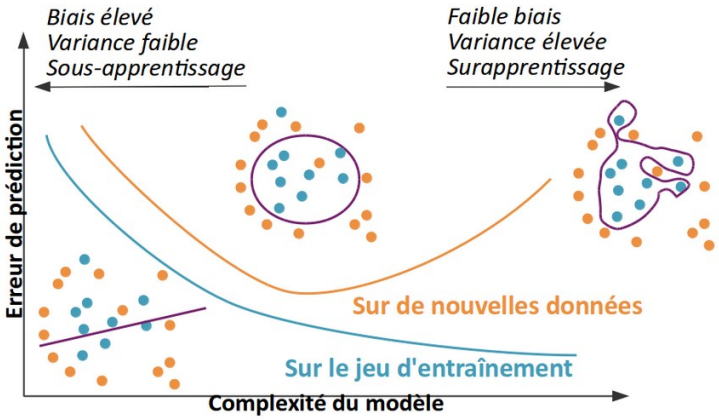

Figure 3 : Performances d'un modèle de classification en fonction de la complexité. Source : https://openclassrooms.com/fr/courses/4297211-evaluez-et-ameliorez-les-performances-dun-modele-de-machine-learning/4297218-comprenez-ce-qui-fait-un-bon-modele-d-apprentissage

☞**Consigne** : déterminer, pour chaque jeu de données à classer, l'ensemble des différents paramètres conduisant au meilleur coût (de mauvaise classification, le plus faible étant le meilleur), en évitant le sur- et le sous-apprentissage.

b. Application à la classification d'images

Vous trouverez sur :

https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

une mise en œuvre simple d'un réseau de neurones convolutif peu profond. Une fois spécifié le réseau, ce code doit vous permettre de produire *un modèle*, qui contiendra la structure globale du réseau à la fin de l'apprentissage (paramètres compris) et sera enregistré au format HDF5 (.h5). Ce dernier sera ensuite utilisé pour la partie test dans un code Python distinct, qui permettra de prédire les classes d'appartenance des images de l'ensemble de test.

☞**Consigne** : adapter ce code (partie apprentissage) au problème de la classification des images utilisées précédemment, et mettre en œuvre la partie test dans un code séparé. Il sera utile de créer quatre répertoires : deux pour la partie apprentissage (*train* et *validation*), et deux autres pour la partie test (*test* et *results*), avec pour chacun d'entre eux un nombre de sous-répertoires correspondant aux classes à apprendre, puis à prédire.

5. Rendu de projet

Le compte-rendu du projet, avec ses deux parties (classification par attributs et classification par réseau de neurone) devra être dématérialisé. Le plus simple est de procéder à un dépôt de l'ensemble sur Git (sous GitHub par exemple), ce qui permet d'inclure les codes (.py ou .ipynb), de décrire les modifications apportées aux codes originaux pour aboutir aux solutions proposées, et d'illustrer le rendu avec les résultats obtenus. Le nombre de signes (blancs non compris) sera compris entre 20000 et 30000 (équivalent de 15 à 20 pages) ; vous pourrez vous inspirer du blog cité plus haut. Vous veillerez également à la qualité de la rédaction (style et orthographe).