ENSSAT
LANNION

Analyse et Traitement d'image

Maël Ortiz - IMR3

Table des matières

Introduction

Nous sommes en pleine révolution numérique, basée sur des outils de plus en plus perfectionnés. Ces outils ont des contenus de plus en plus complexes et tout ceci ne demande qu'à être organisé, ce qui va donc nous amener à différentes classifications.

Nous parlerons donc de classification d'images. Cette dernière consiste à catégoriser des images en fonctions de divers attributs liés à l'image extraite. Cela est donc particulièrement utile par rapport à l'expansion de la révolution numérique.

Ce projet a pour but d'explorer la classification d'images à travers deux approches différentes.

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).

Ce rapport sera donc divisé en 2 parties traitant chacune d'une des classifications mises en œuvre.

Partie 1 : *La classification Old School*

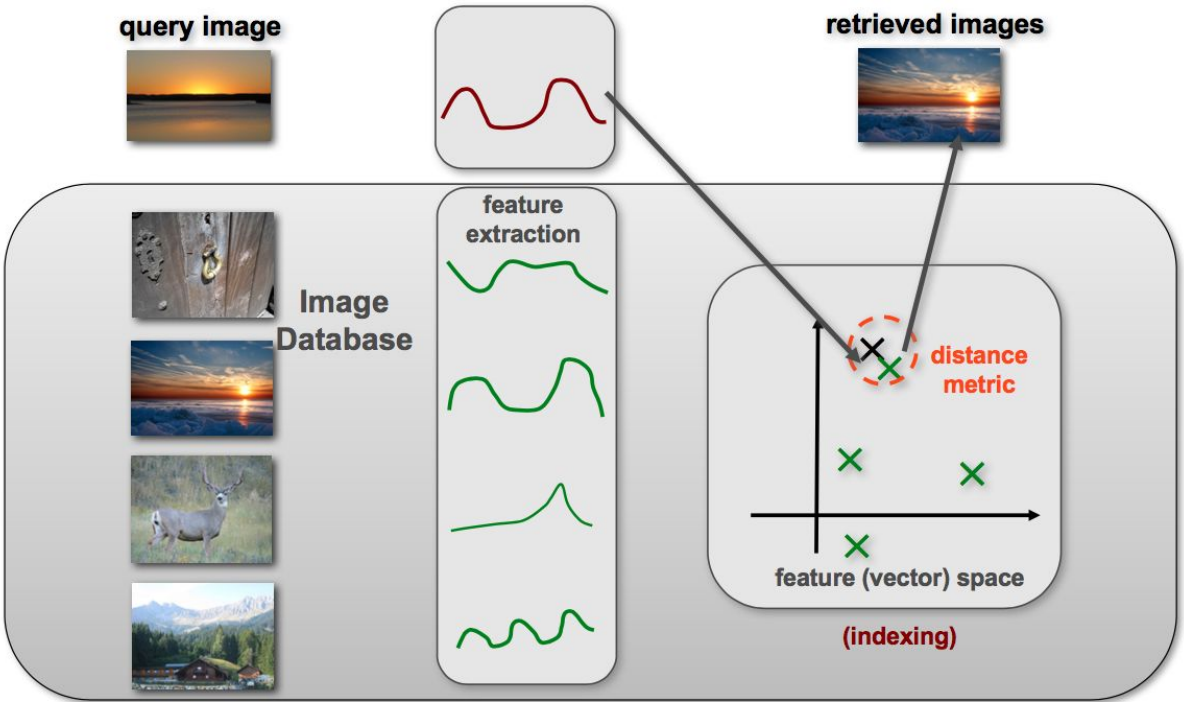
I. Présentation de la méthode

La classification est basée sur des attributs. Les calculs d'attributs suivent différentes méthodes (contours, histogramme des couleurs ou des gradients). On essaye de renvoyer à l'utilisateur les images répondant le mieux à la requête.

Nous nous sommes basé sur un code existant de l'utilisateur Github *pochih* pour réaliser les modifications de cette première classification.

On y retrouve des programmes permettant d'analyser un lot d'images et de prédire si 2 images représentent la même chose. Cette prédiction est réalisée au moyen d'un seul ou plusieurs attributs extrait au moyen d'un autre programme.

Les différents attributs pouvant être extraits sont, par exemple, la couleur, la texture ou les formes/contours.



Classification avec les attributs

II. Solution

A. Fonctionnement de base

Le fonctionnement de base de la solution prévue par le projet récupéré est celui-ci :

Tout d'abord, il faut préparer un dossier de **train**. Ce dossier est celui où se trouve les images qui vont être utilisées. Ensuite, un des programmes utilisant divers attributs comme indiqués ci-dessus va être exécuté. A titre d'exemple, un des programmes est color.py, qui permet de classifier les images en fonction de leurs couleurs. Les images seront alors récupérées et évaluées.

Une fois que cela est effectué, le programme color.py va afficher pour chaque images, un nombre d'images défini au préalable lui ressemblant le plus, en se basant toujours sur l'attribut de la couleur. Ces images "comparées" seront donc les images les plus proches selon leurs couleurs de l'image de base.

B. Modifications apportées

Une des première chose que nous avons effectué est d'ajouter un nouveau dossier intitulé **validation.** Il va servir les même catégories que le dossier **train**, mais les images qu'ils vont contenir seront différentes.

Un des premier fichier que nous avons modifié est le fichier db.py. A l'intérieur de ce programme se trouve la classe Database. Cette classe sert à initialiser la base de données avec les images que nous allons utiliser plus tard. Cette classe a donc été modifiée en utilisant 2 paramètres lors de l'initialisation de cette dernière. Le premier paramètre correspond à l'emplacement du dossier qui contient la base d'images, le second paramètre est l'emplacement du fichier csv contenant les différentes informations liées aux images.

```
14    def __init__(self):
15        self._gen_csv()
16        self.data = pd.read_csv(DB_csv)
17        self.classes = set(self.data["cls"])
18
```

← Première version de db.py

```
14    def __init__(self, DB_dir, DB_csv):
15        self._gen_csv(DB_dir, DB_csv)
16        self.data = pd.read_csv(DB_csv)
17        self.classes = set(self.data["cls"])
18        self.db_type = DB_dir[-4:]
19
```

← Version modifiée de db.py

Le second fichier à être modifié est le programme color.py. Cette modification se retrouve dans la fonction main. Nous avons donc en paramètres, non plus un seul jeu de données, mais deux bases de données. L'une correspondant au train, l'autre à la validation. Nous allons donc récupérer ces bases de données qui ont été définies grâce au répertoire contenant les images stockées par classe et le fichier csv correspondant.

```
171     DB_train_dir = '../dataset/train'
172     DB_train_csv = '../result/train.csv'
173
174     db1 = Database(DB_train_dir, DB_train_csv)
175     print('DB length', len(db1))
176
177     color = Color()
178
179     DB_validation_dir = '../dataset/validation'
180     DB_validation_csv = '../result/validation.csv'
181
182     db2 = Database(DB_validation_dir, DB_validation_csv)
183     print('DB length', len(db2))
```

C'est ensuite que l'on va appeler la fonction d'évaluation, myevalutate, provenant directement du fichier evaluate.py. Cela va permettre, comme son nom l'indique, d'évaluer les bases de données passées en paramètres afin de traiter les images par rapport à l'attribut sélectionné (ici, la couleur, d'où color.py) :

```
185     # evaluate database
186     APs, res = myevaluate(db1, db2, color.make_samples, depth=depth, d_type='d1')
187     #APs, APn = evaluate_class(db, f_class=Color, d_type=d_type, depth=depth)
```

On va alors utiliser deux variables, APs qui correspond aux informations liées aux classes prédites alors que res va correpondre aux informations liées aux images prédites comme étant proches de l'image de base (celle qui va être utilisée comme comparaison).

Une des informations qui ressort est donc la précision pour chacune des classes utilisées par rapport à l'image de base, et c'est ainsi que l'on peut calculer la précision totale de cette prédiction.

Finalement, on va pouvoir afficher les valeurs MAP pour chaque classe utilisée, et trouver la moyenne de ces valeurs.

```
189     cls_MAPs = []
190     for cls, cls_APs in APs.items():
191         MAP = np.mean(cls_APs)
192         print("Class {}, MAP {}".format(cls, MAP))
193         cls_MAPs.append(MAP)
194     print("MMAP", np.mean(cls_MAPs))
```

Les images sont ensuite réparties dans des dossiers qui, selon la prédiction, leur correspondent:

```
196     dir = "../result/results/"
197     for root, dirs, files in os.walk(dir, topdown=False):
198         for name in files:
199             os.remove(os.path.join(root, name))
200         for name in dirs:
201             os.rmdir(os.path.join(root, name))
202
203     for i in range(len(db2)):
204         dir = "../result/results/" + res[i]
205
206         if not os.path.isdir(dir):
207             os.mkdir(dir)
208
209         saveName = dir+ "/" + db2.data.img[i].split('/')[-1]
210         bid = imageio.imread(db2.data.img[i])
211         mpimg.imsave(saveName, bid/255.)
```

Le dernier fichier à avoir été modifié est evaluate.py. C'est le point central de notre méthode old school. Comme vu plus tôt, une des fonction utilisée est nommée myevaluate, elle provient donc d'ici. Une autre fonction dans ce fichier a avoir été modifiée est myevaluate_class.

Afin de faire profiter ces changements, les fonctions de base, evaluate et evaluate_class n'ont pas été supprimées, une comparaison peut être faite en utilisant les différentes fonctions. Les fonction de base infer et AP ont été également modifiées, cette dernière connaîtra son doublon modifié myAP. Ces deux classes modifiées vont faire profiter la précision de notre méthode.

Une des premières modification a été réalisée sur la fonction AP, ou plutôt sur la création de la fonction myAP se basant sur la fonction AP. Cette dernière fonctionnait de manière à donner les images qui vont ressembler à l'image initial passée en paramètre. Après modification, cette fonction permet de vérifier si une image passée en paramètre est attribuée à la bonne classe.

La fonction va ainsi vérifier si l'image est attribuée à la bonne classe, si c'est le cas, la variable hit va passer à 1 et va être ajoutée au tableau nommé precision. Cet tableau regroupe les valeurs de hit pour toutes les images traitées, d'où la boucle sur toutes les images. Une fois que les images ont été traitées, on sort de la boucle et la fonction va retourner le booléen visant à ce que la moyenne des valeurs présentes dans le tableau precision soit supérieure à 0.5. Si c'est le cas, la fonction va renvoyer true, sinon false. True signifierais que plus de la moitié des images ont été placées dans la bonne classe.

Une deuxième modification se trouve dans la fonction infer qui, elle, ne s'est pas vu attribuée de doublon, car c'est une amélioration de la fonction avant tout. Lors de la version initiale de infer, on pouvait voir que cela se basait sur la distance entre 2 images afin de déterminer les différences entre elles et ainsi dire si elles sont similaires ou non. Après de cette amélioration, il est désormais possible de voir que la précision est d'autant plus fine car nous avons rajouté la distance en les classes en plus de la distance entre les images :

```
124     if depth and depth <= len(results):
125         results = results[:depth]
126         print(q_img)
127         list_im = [sub['img'] for sub in results]
128         print(list_im)
129         list_im.insert(0, q_img)
130         pred = [sub['cls'] for sub in results]
131         weig = [sub['dis'] for sub in results]
132         weig = np.reciprocal(weig)
133         pred2 = weighted_mode(pred, weig)
134         pred = np.array_str(pred2[0])[2:-2]
```

On peut voir ça justement grâce à la ligne indiquant pred2 = weighted_mode(pred, weig)

Ce mode va retourner un tableau qui contient les valeurs modales pondérées. Ainsi, lorsque les valeurs sont parcourues, si certaines d'entre elles sont similaires, seule la première est restituée, ce qui nous évite les doublons lors de l'exécution de la méthode.

Lors de la ligne suivante, nous pouvons voir que la valeur de ce weighted_mode, stockée dans la variable pred2, est utilisée pour créer un nouveau tableau de String.

Conclusion sur la méthode *Old School*

Pour cette première méthode, dite old school, la pratique consiste à extraire des attributs (ou appelés "primitives") tels que la couleurs, la texture, les formes/contours etc… Une fois cette extraction réalisée, une classification va être effectuée en s'appuyant sur les critères de décision basés sur les distances entre ces vecteurs d'attributs. C'est ainsi qu'on peut avoir des résultats différents, par exemple pour la couleurs les résultats sont largements satisfaisants, alors que pour l'attribut des formes, les résultats sont moins bons.

Partie 2 : *La classification New School*

I. Présentation de la méthode

La classification new school correspond à une méthode de classification qui s'appuie sur un réseau de neurones. Il n'y est plus question d'attributs, on en vient à un système d'entraînement du réseau de neurones, qui va prendre en paramètres les images et il va ainsi être capable de donner directement la classe a laquelle appartient chacune des images.

Pour créer ce réseau de neurones, j'ai pu m'aider du keras blog blog.keras.io de François Chollet. François Chollet est un ingénieur de Google, il est l'auteur de keras.

La méthode proposée dans cette partie est de mettre en œuvre un réseau de neurones convolutif. On va donc dans un premier temps entraîner le modèle qui contient la structure du réseau à la fin de l'apprentissage. Cette entraînement est ensuite utilisé pour la partie de test qui sera un programme à part entière. Il va permettre de prédire les classes auxquelles appartiennent les images contenues dans le test.

Avant de rentrer plus en détails dans les caractéristiques du réseau mis en place, il faut se rappeler qu'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.

II. Solution

Il faut dans un premier temps utiliser des classes d'images, dans notre cas, ce sera fitness, obj_orbits, pet_cat et sc_indoor. Ces classes d'images sont des dossiers répartis dans les répertoires de validation (**val**), de train (**train**) et de test (**test**). La répartition se fait donc selon un script avec un certain pourcentage, dans notre cas, ce sera 50%/25%/25% comme indiqué ci-dessous :

```
1  import split_folders
2
3  # Cette librairie permet de facilement découper notre dossier src en 3 dossiers train / test / val
4  split_folders.ratio('../dataset/data/', output="../dataset/", seed=1337, ratio=(.5, .25, .25))
5
```

Une fois les jeux de données constitués, on peut s'attaquer au réseau de neurones.

La solution est composée de deux fichier python. Le premier fichier, train.py, va permettre l'entraînement du modèle, le deuxième fichier, test.py, va utiliser ce modèle entraîné pour classifier les images qu'on va lui passer en paramètres.

On va commencer par détailler le fichier train.py. On commence par indiquer les chemins vers les répertoires de validation (**val**) et de train (**train**) comme indiqué :

```
13 # dimensions of our images.
14 img_width, img_height = 64, 64
15 train_data_dir = '../dataset/train'
16 validation_data_dir = '../dataset/val'
```

La taille des images est calculée en fonction de la taille originale des images. Comme certaines images sont réduites par rapport à d'autres, on s'assure qu'en 64x64, il n'y ait pas de soucis d'agrandissement et donc de réduction de qualité à cause de ce dernier. Cela pourrait fausser notre classification.

```
22 epochs = 25
23 batch_size = 8
```

En restant raisonnable tout en ne visant pas le sur-apprentissage, nous avons décidé de réaliser 25 epochs. Il permet d'obtenir une précision raisonnable et le moins de perte possible.

Les lots auront donc une taille de 8 afin d'entraîner au mieux notre modèle, cela lui permettra une souplesse d'apprentissage avec une certaine régularité. Etant donné que nous avons un jeu de donné restreint (seulement 4 classes), il nous faut donc une taille de lot assez restreinte également.

Le modèle est composé de 3 couches, qui vont être affinées au fur et à mesure afin d'avoir la précision de la classification :

```
model = Sequential()

# #1ère couche
model.add(Conv2D(16, (5, 5), input_shape=input_shape, padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))  # groupe 2x2 pixel

#2ème couche
model.add(Conv2D(32, (5, 5)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Dropout(0.25))

#3ème couche
model.add(Conv2D(64, (5, 5)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())  # mise à plat des coeffs des neurones
model.add(Dense(64))  # dense = fully connected
model.add(Activation('relu'))
model.add(Dropout(0.25))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Pour finir ces couches, on compile le modèle en utilisant comme attribut de perte le categorical_crossentropy, car si nous avions utilisé binary_crossentropy, nous aurions du nous limiter à 2 classes.

Pour pallier au problème d'avoir un jeu de données faibles, on utilise un procédé appelé data augmentation. Ce procédé est assez courant en apprentissage de réseau de neurones. Il permet, comme son nom l'indique, d'augmenter la source de données afin d'entraîner le réseau de neurones :

```
57  # this is the augmentation configuration we will use for training
58  train_datagen = ImageDataGenerator(
59          rescale=1./255,
60          shear_range=0.2,
61          zoom_range=0.2,
62          horizontal_flip=True)
63
64  # this is the augmentation configuration we will use for testing:
65  # only rescaling
66  validation_datagen = ImageDataGenerator(rescale=1. / 255)
```

Cela peut par contre rendre le modèle réticent au contact d'images ayant subies des transformations.

On finit le fichier avec une ligne :

```
93  model.save('CorelDB_model.h5')
```

Cela va permettre d'enregistrer notre modèle afin de garder les caractéristiques obtenues lors de son apprentissage/entraînement. Il va donc pouvoir être utilisé avec le deuxième fichier qui va symboliser la phase de test.

En testant tout d'abord le fichier train.py, nous pouvons conclure que la précision est bonne et que les pertes sont moindres. Avec 25 epochs, il n'y a aucun soucis.

Nous arrivons donc à la description du second fichier, test.py. Nous allons, dans un premier lieu, charger le modèle préalablement entraîné :

```
16  model = load_model('CorelDB_model.h5')
```

On indique ensuite, comme dans le fichier de train, les chemins vers les répertoires qui nous intéressent. Dans notre cas le répertoire de test (**test**) et le répertoire de résultats (**res**) :

```
18  img_width, img_height = 64, 64
19
20  test_dir = "../dataset/test"
21  res_dir = "../dataset/res"
```

On garde bien évidemment le même format d'image pour éviter tout problèmes entre l'entraînement de notre modèle et la phase de test.

Nous allons donc indiquer les informations que le modèle a apprises durant la phase d'entraînement :

```
39  y_pred = model.predict_classes(batch_holder)
40
41  classification, confusion = reports(y_pred, y_true, class_names)
42
43  print(classification)
44  print(confusion)
```

Les classes d'images étant assez distantes les unes des autres, cela nous permet d'avoir des résultats satisfaisants sur le nombre d'images que constitue notre jeu de données passées en paramètre de la phase de test. Nous obtenons donc une assez bonne précision.

Conclusion méthode *New School*

La méthode new school se base donc sur l'apprentissage d'un réseau de neurone. Il est donc plus simple à mettre en place que la méthode concurrente, dite old school.

Tout d'abord, il faut donner des images en entrée au réseau de neurones afin qu'il puisse s'entraîner (phase d'entraînement, train.py) puis le modèle va permettre de prédire la classe d'une image et effectuer sa classification (phase de test, test.py).On a donc plus besoin d'extraire des attributs d'une image afin de les comparer à d'autre et classifier cette dernière.

Mais on retrouve un souci lors de l'exécution de cette méthode, les réseaux de neurones. Ce sont des boîtes noires, leur fonctionnement est difficilement compréhensible et cela nous bloque dans les diverses améliorations que l'on pourrait apporter à notre programme.

Conclusion

Nous avons donc mis en place ce projet en utilisant 2 solutions permettant d'analyser, traiter et classifier des images. La première solution se basait sur des attributs (primitives) alors que la seconde se basait sur un réseau de neurones. La première est dite old school car c'était cette méthode qui était utilisée avant que l'apprentissage de réseaux de neurones ne soit vraiment mis en place.

Après divers tests sur ces deux méthodes, il en ressort quelques points intéressants. Les résultats étaient sensiblement similaires après utilisation des méthodes. Même s'ils peuvent être jugés satisfaisant, ces résultats proviennent de deux problèmes, chacun lié à sa méthode. La seconde méthode dite new school est beaucoup plus simple à mettre en place que sa concurrente, la méthode old school. Mais on retrouve ensuite le problème de la seconde méthode, si la première était plus longue, la seconde est moins compréhensive. Le fonctionnement de la seconde se base sur des réseaux de neurones qui sont compliqués à comprendre, ce qui rend les améliorations de code assez difficiles. La première méthode rend ces améliorations plus aisées avec une bonne compréhension du code.