Analyse et traitement d'images

Rapport de projet

Johann Carfantan

IMR3 2019-2020

Table des matières

Introduction

L'objectif de ce rapport est de présenter les deux approches utilisées pour le projet d'analyse et traitement d'images. Dans un premier temps nous verrons comment les données ont été préparées, nous verrons ensuite l'approche par classification d'attributs et nous terminerons par la classification par réseau de neurones.

1 - La préparation

1.1 - L'environnement de développement

Avant une quelconque préparation de données il est important de préciser l'environnement de développement que j'ai utilisé si vous souhaitez reproduire mes résultats. La version de python utilisée est 3.7.6, et n'étant pas familier avec la suite Anaconda, Jupyter ou Spyder j'ai principalement utilisé Visual Studio Code développé par Microsoft en tant qu'IDE pour le développement du projet.

1.2 - Les données utilisées

Ce que l'on cherche à réaliser avec ce projet consiste à classifier des images en fonction de leurs attributs, cela implique que nous devons avoir une base de données d'images classifiées pour le projet. La base de données CorelDB proposée par les enseignants semble être un exemple parfait pour ce projet. Cette base contient une grande quantité d'images (plus de 10 000) regroupées par classe. Cette classe indique si notre image est un "dog", un "iceberg" etc… L'autre avantage de cette base de données est la taille réduite des images qui sont au format 120x80, permettant ainsi un traitement rapide des images avec les machines mises à notre disposition. Dans le cadre du projet ce format d'image convient totalement étant donné que l'objectif est de traiter le maximum d'images en un minimum de temps afin d'avoir des résultats rapides. L'idée est de réduire la résolution

des images sans pour autant en dégrader le contenu ou en perdant les informations principales contenues dans ces images, ce qui pourrait nuire aux traitements futurs que nous allons réaliser sur ces images. La base de données CorelDB a donc été sélectionnée pour le projet.

Dans mon cas, et pour tous les tests expliqués dans ce rapport, je n'ai pas utilisé toute la base de donnée CorelDB, ce qui aurait été beaucoup trop long, j'ai préféré ne sélectionner que certaines classes de photos. Ces classes sont :

- art_antiques : 102 images
- bld_sculpt : 200 images
- fitness : 200 images
- obj_dish : 100 images
- obj_orbits : 100 images
- pl_foliage : 100 images
- sc_mountain : 270 images

Ces classes représentent donc un échantillon de la base de données CorelDB assez fourni pour permettre un traitement correct des données par la suite.

2 - La classification par attributs

2.1 - Présentation

La première approche utilisée est une approche CBIR (Content-Based Image Retrieval) inspirée par un repo github fourni par les enseignants. L'objectif de cette partie du projet est d'utiliser le code fourni afin de traiter les images de notre base de données, notre travail dans un premier temps à donc été de lire et de comprendre le fonctionnement du code fourni.

Le code fourni propose le traitement et la classification d'images par le biais de 7 algorithmes différents, chacun se focalisant sur un attribut spécifique, on notera par exemple l'histogramme de couleurs, un filtre de gabor, un edge histogramme, un histogramme de gradient avancé, etc… Ces algorithmes fonctionnent séparément, néanmoins un algorithme permettant une utilisation de plusieurs algorithmes en simultané est aussi implémenté, il s'appelle "fusion". Les algorithmes fonctionnent tous de la même façon, on leur fournit une

image en entrée, de cette image sont extraits les attributs nécessaire à l'algorithme pour la classification, et ces attributs sont comparés aux attributs de toute la base de données fournie (déjà classifiée) afin de retrouver la classe la plus "proche" permettant de classifier l'image fournie au départ.

Après manipulation du code proposé, j'ai décidé de me focaliser principalement sur le code "color" et "edge" qui s'occupent respectivement du traitement par histogramme de couleurs et par edge histogramme. Ce choix a été fait premièrement grâce à la facilité d'exploitation de ces algorithmes, mais surtout par la difficulté à faire marcher les autres algorithmes une fois le code modifié pour correspondre à notre projet, j'ai d'ailleurs remarqué que plusieurs élèves de la classe n'ont pas réussi non plus à faire fonctionner tous les algorithmes. Seuls color et edge ont donc été gardés, les autres algorithmes ont été supprimés.

2.2 - Architecture du projet

Image 1 : capture d'écran de l'architecture du projet

L'architecture du projet a été modifié comme suit : un dossier "database" a été ajouté ainsi que 3 scripts "clear.sh", "color.sh" et "edge.sh". Le dossier database est composé de 4 sous dossiers et d'un script python "repartition.py". Le sous-dossier "src" contient toutes les

images de notre base de données (réduite aux 7 classes sélectionnées), voir la photo suivante.

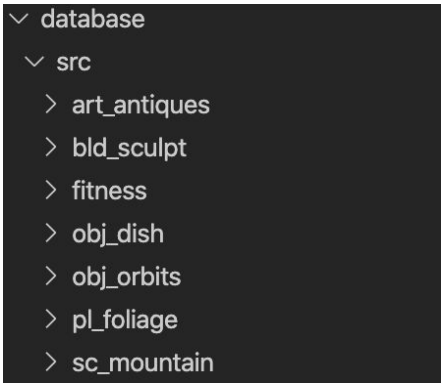

Image 2 : capture d'écran de l'architecture du dossier "database"

Le script "repartition.py" est responsable de la séparation des images de cette base de données "src" en trois sous dossiers "train", "validation" et "test" avec les proportions respectives suivantes 70%, 15% et 15%.

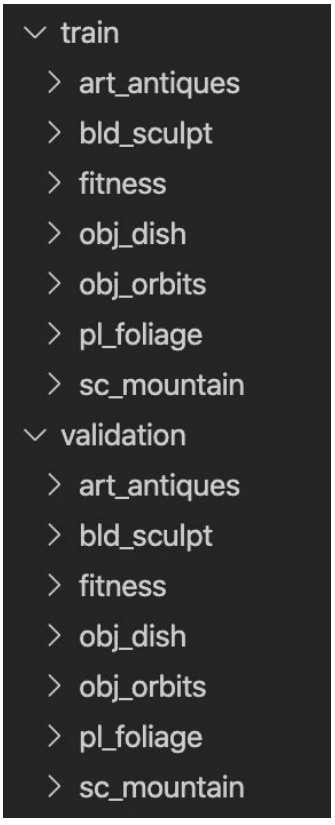

Image 3 : capture d'écran du dossier "train" et "validation"

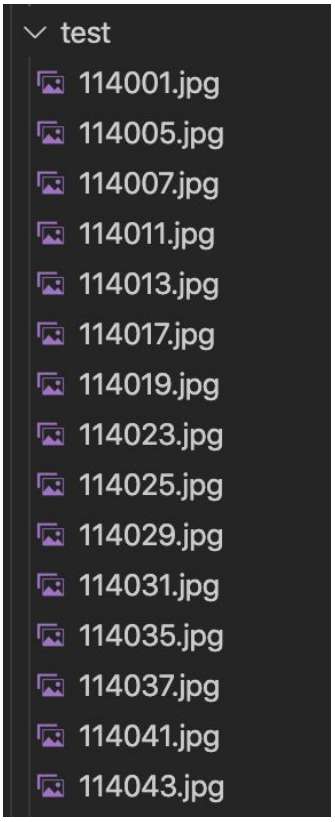

Image 4 : capture d'écran du dossier "test"

Comme nous pouvons le remarquer les dossiers "train" et "validation" conservent l'architecture par classe des images, c'est tout à fait logique étant donné que c'est sur ces dossiers que l'on va baser notre apprentissage pour la classification des images. Le dossier test quand à lui ne contient que les images, et c'est normal puisque l'objectif est de retrouver la classe des images dans ce dossier, on va donc supprimer la classification de ces images en les retirant de leur dossier de classe.

Les algorithmes utilisés ne vont utiliser en réalité que le dossier "train" et "test", le dossier "validation" n'est pas du tout utilisé par le code fourni, il aurait néanmoins pu être intégré afin de vérifier l'apprentissage réalisé, mais cela ne rentrait pas dans le cadre du projet. Alors pourquoi le script "repartition.py" sépare-t'il le dossier "src" en 3 sous-dossiers plutôt qu'en deux? La réponse est qu'il nous a été demandé de le faire afin d'anticiper la partie suivante sur les réseaux de neurones, qui eux vont utiliser ces 3 sous-dossiers.

Le dossier "prédictions" est le dossier qui contiendra les résultats de nos algorithmes à savoir la classification par dossier de classe des images dans le dossier "test". L'idée est

donc de reconstruire l'architecture par classe des dossiers "train" et "validation" à partir des images de "test" qui n'ont pas de classe.

Finalement, les trois script ont pour rôle :

- clear.sh : effacer tous les fichiers de cache python, effacer les prédictions des exécutions précédentes, effacer les fichiers data_train.csv et data_test.csv qui conservent des informations d'une exécution à l'autre.
- color.sh : exécution du script clear.sh et lancement de l'algorithme color.py
- edge.sh : exécution du script clear.sh et lancement de l'algorithme edge.py

Afin de réaliser nos tests en utilisant les 3 scripts il est important de les rendre exécutables en faisant la commande "*chmod u+x ./le_script_à_rendre_executable.sh*" sur chaque fichier, suite à quoi nous pourrons par exemple exécuter la commande "*./color.sh*".

2.3 - Les modifications du code

Dans cette partie le code ne sera pas expliqué en détail mais uniquement les parties modifiées afin d'arriver à notre résultat (une partie du code étant fournie et expliquée par les professeurs en cours), si toutefois vous souhaitez regarder l'implémentation réalisée vous pouvez vous référer aux sources du projet disponibles sur github.

La première étape a consisté à modifier le code du fichier "DB.py" afin qu'il puisse prendre en compte plusieurs dossiers, en effet ce dernier ne prenait en compte qu'un seul dossier à la base. Pour cela il a fallu rajouter au constructeur l'attribut "DB_dir" qui contient en réalité le chemin vers le dossier que nous traitons, dans notre cas le chemin vers le répertoire "train" ou "test". En ajoutant le paramètre "self.db_type" cela permet de différencier les deux bases lors de l'exécution du script et donc de ne pas interférer entre les fichiers de cache générés par l'exécution de "train" et de "test".

```
class MyDatabase(object):

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

Image 5 : capture d'écran du constructeur de la classe MyDatabase

Ensuite, peu importe l'algorithme sélectionné nous utilisons le fichier "evaluate.py" qui sert à déterminer la classe la plus probable choisie par l'algorithme et à enregistrer le résultat. Il faut garder en tête qu'avant la classification se faisait par la classe qui présentait le plus de pourcentage de ressemblance avec notre image, et nous souhaitons implémenter l'algorithme des plus proches voisins. Cela explique certains choix qui ont été réalisés, notamment la modification de la fonction "customAP".

```python
def customAP(label, results, sort=True):
    if sort:
        results = sorted(results, key=lambda x: x['dis'])
    precision = []
    for i, result in enumerate(results):
        cpt = 0
        if result['cls'] == label:
            cpt = 1.
            precision.append(cpt)

    return np.mean(precision)
```

Image 6 : capture d'écran de la fonction customAP

Cette fonction retourne désormais 1 si la classe de l'image fournie correspond à la classe de l'image recherchée. Cette fonction "customAP" est appellée dans la fonction "infer" qui a été modifiée afin de prendre en compte les modifications de la fonction "customAP" mais aussi afin d'implémenter la méthode des plus proches voisins.

```python
def infer(query, samples=None, db=None, sample_db_fn=None, depth=None, d_type='d1'):
    assert samples is not None or (db is not None and sample_db_fn is not None), "need to give either samples or db plus sample_db_fn"
    if db:
        samples = sample_db_fn(db)

    q_img, q_cls, q_hist = query['img'], query['cls'], query['hist']
    results = []
    for idx, sample in enumerate(samples):
        s_img, s_cls, s_hist = sample['img'], sample['cls'], sample['hist']
        if q_img == s_img:
            continue
        results.append({
            'dis': distance(q_hist, s_hist, d_type=d_type),
            'cls': s_cls,
            'img': s_img
        })
    results = sorted(results, key=lambda x: x['dis'])
    if depth and depth <= len(results):
        results = results[:depth]
        # list_images = [sub['img'] for sub in results]
        pred = [sub['cls'] for sub in results]
        weig = [sub['dis'] for sub in results]
        weig = np.reciprocal(weig)
        pred2 = weighted_mode(pred, weig)
        pred = np.array_str(pred2[0])[2:-2]

    ap = customAP(q_cls, results, sort=False)

    return ap, pred
```

Image 7 : capture d'écran de la fonction infer

La fonction “infer” comporte un tableau d’objets “results” qui contient la liste des objets correspondant à toutes les images de la base “train”. Chaque objet contient le nom de l’image concernée dans la variable “img”, sa classe dans la variable “cls” et la distance de ressemblance avec l’image à classifier dans la variable “dis”. Ce tableau est ensuite trié en fonction de la distance de ressemblance avec l’image à classifier “dis” et les classes se voient attribuer un poids grâce à la fonction “weighted_mode”. On sélectionne ensuite la classe qui apparaît le plus dans les résultats les plus proches en distance de notre image à classifier (c’est donc l’algorithme des plus proches voisins), donc celle qui a le poids le plus élevé, et c’est cette classe qui sera sélectionnée comme étant la classe de l’image à classifier dans la variable “pred”.

```
for query in testSamples:
    ap, pred = infer(query, samples=trainSamples, depth=depth, d_type=d_type)
    print(query['img'], "    ==>    ",pred)
    ret[query['cls']].append(ap)
    predict.append(pred)
```

Image 8 : capture d’écran de la fonction myevaluate

Finalement, la fonction “myevaluate” a été modifiée afin d’avoir le comportement adapté à notre besoin. Nous voulons pour chaque image du dossier “test” réaliser une classification grâce aux images du dossier “test”, nous devons donc faire une boucle adaptée. Le résultat de chaque prédiction est ensuite affiché dans le terminal, comme sur l’image ci-dessous.

```
database/test/282053.jpg     ==>     sc_mountain
database/test/282047.jpg     ==>     sc_mountain
database/test/617095.jpg     ==>     sc_mountain
database/test/617081.jpg     ==>     sc_mountain
database/test/470075.jpg     ==>     sc_mountain
database/test/435065.jpg     ==>     sc_mountain
database/test/435071.jpg     ==>     sc_mountain
```

Image 9 : capture d’écran de la sortie de l'exécution d’un script dans le terminal

Finalement, les fichiers “color.py” et “edge.py” ont été modifiés afin de prendre en compte toutes les modifications faites auparavant, et les modifications faites dans les fichiers sont identiques.

Dans un premier temps il a fallu corriger une erreur de librairie sur la fonction “imread”, en remplaçant l’appel à “scipy” par la librairie “imageio”.

```
def make_samples(self, db, verbose=True):
    mystr = db.get_db_type()
```

Image 10 : capture d’écran de la fonction make_samples

La fonction “make_samples” a ensuite été modifiée afin de prendre en compte l’attribut “db_type” de la classe “MyDatabase” ajouté auparavant afin de bien séparer les fichiers générés entre la base “train” et “test”.

```
if __name__ == "__main__":

    dbTrain = MyDatabase("database/train", "database/train/data_train.csv")
    print("Train db length: ", len(dbTrain))
    color = Color()

    dbTest = MyDatabase("database/test", "database/test/data_test.csv")
    print("Test db length: ", len(dbTest))

    # evaluate database
    APs, res = myevaluate(dbTrain, dbTest, color.make_samples, depth=depth, d_type="d1")

    # add pictures in prediction folder under the predicted class folder
    path = "/Users/johanncarfantan/Documents/ENSSAT/IMR3/AnalyseDimages/Partie 1/predictions/"
    for i in range(len(dbTest)):
        saveName = path + res[i] + "/" + dbTest.data.img[i].split('/')[-1]
        bid = imageio.imread(dbTest.data.img[i])
        if not os.path.exists(path + res[i]):
            os.makedirs(path + res[i])
        mpimg.imsave(saveName, bid / 255.)
```

Image 11 : capture d’écran de la fonction main

Les plus grandes modifications se trouvent en réalité dans la fonction “main” qui permet à toutes nos modifications précédentes de bien fonctionner correctement. On récupère les bases “train” et “test”, on appelle ensuite la fonction “myevaluate” qui retourne un résultat, et finalement on créé une copie de notre image à classifier dans le sous-dossier correspondant du dossier “prédictions”.

2.4 - Analyse

En vérifiant les résultats dans le dossier prédictions on se rend vite compte que l'algorithme "color" est bien plus efficace que l'algorithme "edge". Cela s'explique surement par la diversité des images choisies, là où une étude sur les formes d'un objet sont compliquées à généraliser un algorithme basé sur l'analyse de l'histogramme des couleurs est plus performant de l'ordre de 86% environ (calcul fait à la main).

3 - La classification par réseau de neurones

3.1 - Présentation

Dans cette seconde partie nous nous sommes aussi basés sur un code fourni par les enseignants qui implémente un simple réseau de neurones permettant le traitement d'images qui différencie des chats et des chiens. Ce code fonctionne de la façon suivante, il réalise un apprentissage en prenant en entrée plusieurs images déjà classées après avoir réalisé de la data augmentation sur cette base de donnée afin d'augmenter la quantité de données, il réalise ensuite une validation du modèle généré par le réseau de neurone sur une base de donnée différente afin d'évaluer les capacités du modèle sur une autre base de données. Les capacités du modèle fournissent ensuite un score, et l'algorithme recommence du début un certain nombre de fois en changeant un certain nombre de paramètres afin d'essayer d'avoir un score optimal. Une fois le nombre d'époques réalisés, il enregistre le modèle dans un fichier dédié et génère un graphique présentant les performances du modèle tout au long de son apprentissage.

3.2 - Architecture du projet

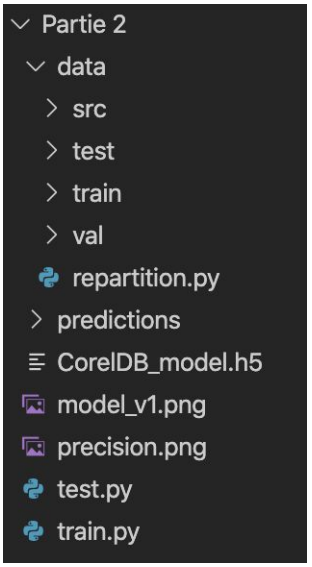

Image 12 : capture d'écran de l'architecture du projet

Comme pour la partie précédente la base de données d'image se situe dans le dossier "data" ou on retrouve le même nombre de sous-dossiers et le fichier "repartition.py". Cependant les sous-dossiers ne portent pas le même nom, cela s'explique par le changement du fonctionnement du fichier "repartition.py" qui utilise désormais une librairie appellée "split_folders" qui est plus efficace, plus précise et plus pertinente pour la séparation des images en 3 sous-dossiers. Cette librairie m'a été conseillée par un autre étudiant de la classe. Dans mon cas, la séparation se fait de la manière suivante : 80% des données dans le jeu "train", 10% dans le jeu "val" (pour validation) et 10% dans le jeu "test" pour l'évaluation finale du modèle.

3.3 - Les modifications du code

Premièrement la taille des images de la base corel sont de 120x80, ou bien 80x120 en mode portrait, il a donc fallu saisir les dimensions des images dans le fichier "train.py". Le modèle qui va être entraîné et utilisé est un empilement de couches, ou modèle séquentiel. Le code utilisé est celui détaillé dans le lien fourni dans le sujet du projet, je n'ai pas modifié ce fichier outre le changement des dimensions des images.

Pour la suite de notre étude des réseaux de neurones il est important de réaliser une phase d'évaluation de notre modèle avec une base de données totalement séparée de

celles de l'entraînement du modèle, c'est le rôle de la base de test. Il nous a donc été nécessaire de créer un fichier "test.py" pour s'occuper de cela.

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

img_width, img_height = 120, 120

img_dir = "data/test"
resu_dir = "predictions"

nb_test_samples = sum([len(files) for r, d, files in os.walk(img_dir)])

batch_holder = np.zeros((nb_test_samples, img_width, img_height, 3))
y_true = np.zeros(nb_test_samples)

class_names = next(os.walk(img_dir))[1]
class_names.sort()

i = 0
for dirpath, dirnames, filenames in os.walk(img_dir):
    for imgnm in filenames:
        img = image.load_img(os.path.join(dirpath, imgnm), target_size=(img_width, img_height))
        batch_holder[i, :] = img
        y_true[i] = int(class_names.index(os.path.relpath(dirpath, img_dir)))
        i = i + 1

y_pred = model.predict_classes(batch_holder)

classification, confusion = reports(y_pred, y_true, class_names)
```

Image 13 : capture d'écran d'une partie du fichier test.py

Premièrement il faut définir la dimension des images, le chemin d'accès vers le fichier du modèle entraîné précédemment et les chemins vers la base "test" et le dossier "predictions" dans notre cas. C'est dans le dossier "predictions" que nous retrouverons comme pour la première partie le résultat de la classification du modèle. Ensuite on initialise correctement les variables "batch_holder" et "y_true" à un tableau rempli de 0 de la bonne dimension. On récupère ensuite la liste des classes que l'on doit prédire dans la variable "class_names". Pour chaque image du dossier test on récupère la prédiction du modèle grâce à la méthode "model.predict_classes". Et on extrait correctement les classifications et la matrice de confusion grâce à la méthode "reports".

```
print(classification)
print(confusion)

for dirpath, dirnames, filenames in os.walk(img_dir):
    structure = os.path.join(resu_dir, os.path.relpath(dirpath, img_dir))
    if not os.path.isdir(structure):
        os.mkdir(structure)
    else:
        print("Folder already exists")

i = 0
for dirpath, dirnames, filenames in os.walk(img_dir):
    structure = os.path.join(resu_dir, os.path.relpath(dirpath, img_dir))
    for imgnm in filenames:
        shutil.copy(dirpath + '/' + imgnm, resu_dir + '/' + class_names[y_pred[i]] + '/' + imgnm)
        i = i + 1
```

Image 14 : capture d'écran de la fin du fichier test.py

Pour terminer on affiche nos matrices de confusion et le résultat de notre classification, puis on copie les images dans le dossier correspondant à la classe prédite dans le dossier "predictions".

3.4 - Analyse

3.4.1 - L'entraînement

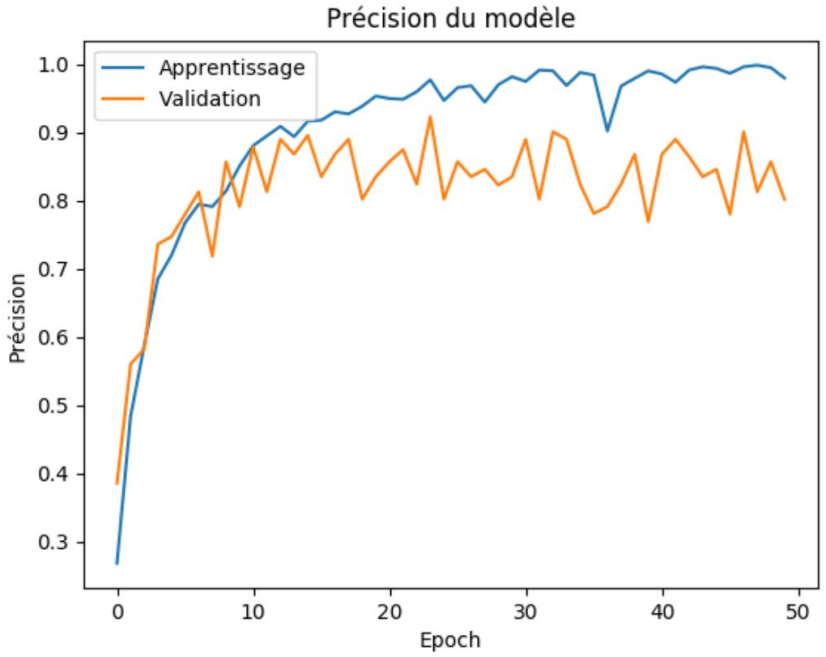

Image 15 : amélioration de la précision du modèle au cours de l'entraînement du modèle

L’entraînement du modèle sur ma base de données s’est amélioré de façon assez conséquente au fur et à mesure des exécutions. On observe toutefois qu’un léger écart existe toujours entre l’apprentissage et la validation.

3.4.2 - L’évaluation

	precision	recall	f1-score	support
art_antiques	0.89	0.73	0.80	11
bld_sculpt	0.44	0.35	0.39	20
fitness	0.49	1.00	0.66	20
obj_dish	0.86	0.60	0.71	10
obj_orbits	1.00	0.80	0.89	10
pl_foliage	0.75	0.60	0.67	10
sc_mountain	0.95	0.67	0.78	27
accuracy			0.68	108
macro avg	0.77	0.68	0.70	108
weighted avg	0.74	0.68	0.68	108

Image 16 : précision de la classification du modèle suite à l'entraînement

Suite à l’exécution du script “test.py” on peut observer que notre modèle est assez performant dans la majorité des cas, voir même excellent pour la classe “obj_orbits”. Par contre, les classes “bld_sculpt” et “fitness” ont des scores de précision assez bas. Celà est peut-être dû à la quantité d’images trop petite permettant au modèle d’être plus précis, ou au fait que ces classes d’images ont tendance à ressembler à d’autres classes.

Conclusion

Ce projet m’a permis de mettre réellement en pratique différentes méthodes d’analyse et de traitement d’images. Je suis satisfait des résultats obtenus suite au développement des deux parties du projet, néanmoins j’aurais pensé que le réseau de neurones serait vraiment plus performant, dans notre cas l’algorithme color de la première partie était plus précis mais je ne suis pas sur qu’avec une très grande quantité de données le résultat serait identique.