ROUSSE Jochen
Année universitaire 2019-2020

Formation Informatique Multimédia & Réseau
3ème Année
ENSSAT Lannion

RAPPORT DE PROJET

Projet analyse d'images

Enseignants responsables : M. CARIOU et M. VOZEL

ENSSAT
LANNION

Table des matières

ENSSAT
LANNION

Partie 1 : Préparation des outils et données

1. Outils

Pour ce projet, j'ai utilisé Python3, en version 3.7.6. Pour l'IDE, mon choix s'est porté sur le logiciel PyCharm Professional développé par la société JetBrains. Cet IDE fait partie de la suite IntelliJ, étant déjà familier avec plusieurs de ces IDE j'ai préféré l'utiliser par rapport à Jupyter ou Spyder que je n'avais jamais utilisé.

2. Environnement et paquets

La première étape nécessaire avant le début du projet a été de mettre en place un environnement de travail pour développer en python. Pour cela, il a été nécessaire d'installer Anaconda et plusieurs librairies afin que le code donné en exemple s'exécute correctement.

Parmi ces librairies se trouve notamment :

- Numpy (v1.18.1) ;
- Scipy (v1.3.1) ;
- Scikit-learn (v0.22.1) ;
- Scikit-image (v0.15.0) ;
- Matplotlib (v3.1.1) ;
- Tensorflow (v1.15) ;
- Keras (v2.2.4).

3. Données

Une fois cette première étape d'installation de paquets effectuée, il a fallu récupérer une base de données d'image de test. Pour cela, j'ai utilisé la base de données CorelDB. Cette base de données comprend 10800 images regroupées en 80 groupes thématiques (= classe). Les images sont de petite dimension (120x80 ou 80x120), ce qui permettra de limiter les temps de traitement.

Pour chacune des parties j'ai choisi les mêmes images, à savoir un total de 900 images appartenant aux 6 classes suivantes :

- *art_dino* ; *fitness* ; *obj_decoys* ; *obj_dish* ; *obj_train* ; *pet_cat*.

ENSSAT
LANNION

Partie 2 : Classification par attributs

1. Préparation des données

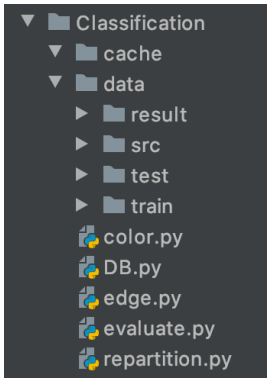


Figure 1 : Architecture du projet

Une fois les données récupérées, il a ensuite fallu les séparer en 2 ensembles distincts :

- Un ensemble d'apprentissage ;
- Un ensemble de test.

Pour cette première partie du projet, il n'était pas nécessaire de créer un ensemble de validation car le code effectuait simplement une classification par attribut.

Afin de s'assurer que les images soient bien réparties, j'ai créé un script de repartition (repartition.py). Ce script copie toutes les images présentes dans le dossier data/src dans un dossier data/train. Il va ensuite choisir 30% des images présentes et les déplacer dans un dossier data/test. Au sein de ce dossier test, les images ne sont pas rangées dans des sous-dossiers contenant le nom de leur classe comme dans le dossier train. Ceci est normal, en effet, si les images étaient déjà placées dans des sous-dossiers contenant le nom de leur classe, le système n'aurait alors aucun travail à effectuer.

Figure 2 : Architecture du dossier data/train

Figure 3 : Architecture du dossier data/test

2. Fonctionnement initial du code

Pour cette partie du projet, nous nous sommes basés sur un code préexistant. Ce code, disponible à cette adresse offrait un système CBIR (Content-Based Image Retrieval).

Son fonctionnement était le suivant :

- Premièrement, il fallait lui donner en entrée l'image recherchée ;
- Ensuite, le code allait extraire des descripteurs de l'image (dépendant de l'algorithme choisie) ;
- Puis, il appliquait le même procédé sur toutes les images de la base et recherchait celles présentant les plus grandes similarités ;
- Finalement, il renvoyait un score (MMAP) permettant d'évaluer le pourcentage d'images de même classe (= vrai positif) renvoyées.

À l'origine, 7 algorithmes étaient implémentés :

- Color : basé sur l'histogramme de couleurs ;
- Gabor : basé sur un filtre de Gabor ;
- Daisy : basé sur des histogrammes de gradients et des fonctions Gaussiennes ;
- Edge : basé sur un edge histogramme ;
- HOG : basé sur un histogramme de gradient orienté ;
- VGG : basé sur un CNN (Convolutional Neural Network) ;
- ResNet : basé sur un ANN (Artificial Neural Network).

Ces 7 algorithmes étaient complétés par une dernière fonction, appelée « fusion », permettant de rechercher une image en combinant plusieurs algorithmes (de 2 à 7) ;

Dans le cadre du projet, j'ai décidé de ne me baser que sur les algorithmes Color et Edge à cause de difficultés à faire fonctionner les autres algorithmes avec le code modifié. Les classes VGG et ResNet étaient exclues d'office du projet car les réseaux de neurones allaient être étudiés dans la seconde partie de celui-ci.

3. Modification du code

Notre objectif pour cette partie du projet était qu'en lançant un algorithme (Color ou Edge dans mon cas), le programme devait se baser sur les images du dossier data/train (où les classes des images étaient connues) pour reconstituer un dossier data/result. Pour constituer ce dossier data/result, il devait utiliser les images présentes dans le dossier data/test (où leur classe est inconnue) et leur attribuer une classe (sous la forme d'un sous-répertoire du dossier data/result).

La première étape pour parvenir à ce résultat a été de modifier le fichier DB.py. Ce fichier permet d'initialiser la base d'images et est le premier instancié dans tous les appels des fonctions de classification.

A l'origine, celui-ci ne gérait qu'un seul dossier, il a donc fallu lui faire gérer notre dossier de train et de test et également faire en sorte qu'il puisse renvoyer les classes présentes dans notre base.

```
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:]
```

Pour cela, j'ai passé en paramètre du constructeur de la classe le chemin vers le dossier train et test et les passons à la fonction gen_csv qui génère les associations entre les images et leur classe. Également, j'ai ajouté un paramètre « db_type », ce paramètre permet de différencier la base de données test de celle de train, et donc de ne pas avoir de conflit leur de la création des fichiers de cache.

La prochaine étape a été de modifier le code d'un algorithme (Color.py) et d'adapter le code d'evaluate.py dans le même temps afin d'obtenir le fonctionnement voulu. Voyons tout d'abord les modifications apportées au code de color.py (les modifications effectuées sur le code du fichier edge.py sont identiques à celles effectuées sur le fichier color.py).

Un des premiers problèmes qui a été corrigé était le chargement de l'image dans la fonction histogram()

```
else:
    # img = scipy.misc.imread(input, mode='RGB')
    img = imageio.imread(input)
```

La fonction scipy.misc.imread ne fonctionnait pas, j'ai donc dû la remplacer par son équivalent avec imageio.

La prochaine modification s'est effectuée dans la fonction make_samples(), cette fonction permet de mettre en cache l'histogramme créé.

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

    if h_type == 'global':
        sample_cache = "histogram_cache-{}-n_bin{}_{}".format(h_type,
n_bin, mystr)
    elif h_type == 'region':
        sample_cache = "histogram_cache-{}-n_bin{}n_slice{}_{}"
.format(h_type, n_bin, n_slice, mystr)
```

Comme expliqué dans la description de la classe DB.py, j'ai utilisé la fonction get_db_type() afin de pouvoir différencier le cache de la base train et celui de la base test.

Finalement, les principales modifications dans ce fichier ont eu lieu dans la fonction main afin d'utiliser les fonctions modifiées d'evaluate.py et de DB.py.

```
DB_train_dir = "data/train"
DB_train_csv = DB_train_dir + "/data_train.csv"

db1 = MyDatabase(DB_train_dir, DB_train_csv)
print("DB length: ", len(db1))
color = Color()

DB_test_dir = "data/test"
DB_test_csv = DB_test_dir + "/data_test.csv"

db2 = MyDatabase(DB_test_dir, DB_test_csv)
print("DB length: ", len(db2))

# evaluate database
APs, res = myevaluate(db1, db2, color.make_samples, depth=depth,
d_type="d1")

for i in range(len(db2)):
    saveName = "./data/result/" + res[i] + "/" +
db2.data.img[i].split('/')[-1]
    bid = imageio.imread(db2.data.img[i])
    if not os.path.exists("./data/result/" + res[i]):
        os.makedirs("./data/result/" + res[i])
    mpimg.imsave(saveName, bid / 255.)
```

Je commence par définir et instancier la base de données de train, ensuite j'instancie la classe Color avant d'instancier la base de données de test.

Ensuite, j'utilise une des nouvelles fonctions de la classe evaluate.py afin d'obtenir les images de ma base de test classées. Finalement, les images sont enregistrées dans le dossier data/result dans le sous-dossier correspondant à la classe déterminée par l'algorithme.

Pour terminer avec cette partie de modification du code, examinons le code du fichier evaluate.py.

Ici, j'ai remplacé les fonctions AP et evaluate par de nouvelles fonctions adaptées au fonctionnement voulu.

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

    return np.mean(precision)
```

Pour la nouvelle fonction myAP, la modification principale est qu'on ne va plus renvoyer une moyenne des résultats mais simplement renvoyer 1 quand la classe de l'image fournie correspond à celle recherchée.

Cette fonction est appelée dans la fonction infer(), qui a également été modifiée. Désormais, les images sont ajoutées au tableau results[] ce qui permet de les enregistrer dans le dossier result.

```
if depth and depth <= len(results):
    results = results[:depth]
    print(q_img)  # image recherchée dans la base de test
    list_im = [sub['img'] for sub in results]
    print(list_im)  # images similaires dans la base de train
    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 = myAP(q_cls, results, sort=False)

return ap, pred
```

La seconde moitié de la function a également été modifiée pour notre but. Tout d'abord, la variable q_img représente l'image auquelle on souhaite attribuer une classe dans notre base de test et le tableau results contient les images similaires à notre image q_img dans la base de train.

Je vais ainsi regrouper toutes les classes de ces images dans la variable pred et également stocker la distance (liée à la comparaison des histogrammes RGB entre les images) des images train par rapport à l'image recherchée dans la variable weig.

Ensuite, la fonction de np.reciprocal est appliquée sur les poids obtenus, ceci permet d'obtenir un tableau trié par ordre décroissant des poids des images train (= ressemblance avec l'image test).

Enfin, j'applique la fonction weighted_mode sur les variables pred et weig. Ceci va nous permettre de retourner l'élément le plus commun en fonction des classes et du poids de chacune d'entre elles.

Par exemple :

```
pred = ['obj_decoys', 'obj_dish', 'obj_decoys']
weig = [0.89669344, 0.89385475, 0.83463745]
pred2 = weighted_mode(pred, weig)
pred = np.array_str(pred2[0])[2:-2]
print(pred) # => 'obj_decoys'
```

Dans l'exemple ci-dessus, on peut constater que pour notre image recherchée, 2 images de la classe 'obj_decoys' et 1 image de la classe 'obj_dish' sont proches. Cependant, en faisant la somme des poids pour chacune des classes il devient assez évident que la classe avec la plus haute valeur sera 'obj_decoys', c'est donc cette classe qui sera retournée par l'appel à la fonction weighted_mode(pred, weig).

Finalement, on stocke cette classe dans la variable pred, qui sera retournée par la fonction.

La dernière fonction que j'ai remplacée dans le fichier evaluate.py est la fonction evaluate. Désormais, la fonction myevaluate prend 2 instances de la classe DB en paramètre.

```
for query in samples2:
    ap, pred = infer(query, samples=samples1, depth=depth, d_type=d_type)
    ret[query['cls']].append(ap)
    predict.append(pred)

return ret, predict
```

Pour chaque image du répertoire data/test, j'appelle la fonction infer afin de pouvoir récupérer la prédiction de la classe de l'image.

4. Analyse des résultats

Sur ma base d'images, les résultats obtenus par les 2 algorithmes sont très variables, par exemple pour l'algorithme color.py sur 270 images de test seulement 3 ont été incorrectement classées par l'algorithme.

Ces très bons résultats sont explicables car j'ai fait très attention de choisir des classes assez « simples » et très éloignées les unes des autres. Ceci facilite donc forcément le travail de l'algorithme.

Par contre, à l'inverse, l'algorithme edge produit des résultats presque inexploitables. Dans mon dossier résultat, seuls 2 classes sont présentes et 90% des images sont concentrées dans le dossier 'obj_decoys', seul 1 image (incorrectement classée) est présente dans le dossier 'obj_dish', le reste des images étant dans le dossier '0.0'.

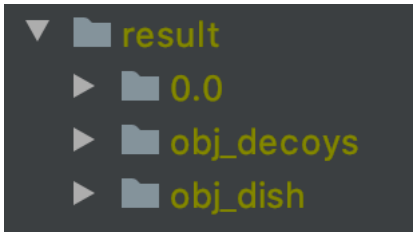


Figure 4 : Structure du dossier résultat après algorithme edge

5. Critique du travail

Un des problèmes que j'ai rencontrés est l'impossibilité de traiter les résultats de la fonction myAP. En effet, on passe en paramètre de cette fonction la classe de notre image recherchée, cependant cette classe est toujours « test » (le nom du dossier de l'image) car les images ne sont pas classées dans le dossier de test.

Ceci fait donc que la fonction myAP renvoie toujours 0 et rend l'analyse du MMAP impossible (toutes les valeurs sont à 0 et uniquement pour la classe 'test'). Il faudrait donc trouver un moyen de corriger ce problème afin de pouvoir produire un score MMAP utilisable.

Partie 3 : Classification par réseau de neurone

1. Préparation des données

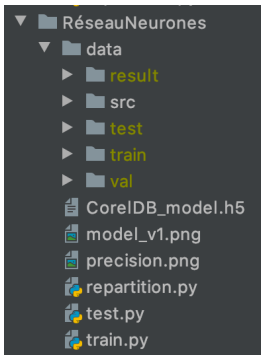


Figure 5 : Architecture du projet

Une fois les données récupérées, il a ensuite fallu les séparer en 3 ensembles distincts :

- Un ensemble d'apprentissage ;
- Un ensemble de test ;
- Un ensemble de validation.

Pour cette seconde partie du projet, il était nécessaire de créer un ensemble de validation car nous allions utiliser un réseau de neurones (prévention du sur-apprentissage).

Afin de s'assurer que les images soient bien réparties, j'ai utilisé un script de repartition (repartition.py). Ce script utilise une librairie (split_folders) qui permet de facilement découper un dossier source en 3 dossiers train / test / val avec des pourcentages choisis.

```
import split_folders

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

J'ai choisi un découpage 50% / 20% / 30% pour mes dossiers train / test / val.

2. Fonctionnement du code

Pour cette seconde partie du projet, nous nous sommes également basés sur un code préexistant. Ce code, disponible à cette adresse offrait un code permettant de distinguer des chats et des chiens.

Son fonctionnement était le suivant :

- Premièrement, il prenait en entrée 2000 images (1000 pour chaque classe) ;
- Ensuite, le code allait effectuer de la data augmentation afin d'augmenter le nombre de données ;
- Puis, il entraînait un réseau neuronal convolutif ;
- Finalement, il affichait la performance du réseau entraîné sur les données de validation.

À partir de ce code, il m'a donc fallu l'adapter pour notre problème de classification d'images avec de nombreuses classes. Il a également fallu créer un fichier de test afin d'évaluer la performance du modèle sur des données de test et pas seulement de validation.

3. Modification du code

En premier lieu, j'ai dû adapter les paramètres du fichier train.py pour mes besoins.

```
img_width, img_height = 120, 120
train_data_dir = 'data/train'
validation_data_dir = 'data/val'
nb_classes = sum([len(d) for r, d, files in os.walk(train_data_dir)])
nb_train_samples = sum([len(files) for r, d, files in
os.walk(train_data_dir)])
nb_validation_samples = sum([len(files) for r, d, files in
os.walk(validation_data_dir)])
epochs = 50
batch_size = 16
```

J'ai donc dû définir les dimensions des images à '120, 120' car les images de la base CorelDB sont de taille 80x120 ou 120x80.

J'ai ensuite défini mes dossiers de train et de validation, puis les 3 lignes suivantes permettent de dynamiquement connaître le nombre de classes présentes, le nombre d'images dans le dossier train et le nombre d'images dans le dossier validation. Finalement, j'ai défini le nombre d'epochs à 50 et la taille de batch à 16, c'est-à-dire que les images sont traitées par lot de 16 et non toutes ensembles.

Les avantages d'utiliser un batch sont que le réseau de neurone va s'entraîner plus vite et va demander moins de puissance au pc.

Ensuite, je définis le modèle qui va être entraîné. Ce modèle est un modèle séquentiel, c'est-à-dire un empilement linéaire de couches (chaque couche a une entrée et une sortie).

```
model = Sequential()
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

model.add(Conv2D(32, (5, 5)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

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

model.add(Conv2D(128, (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(128))  # 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'])
```

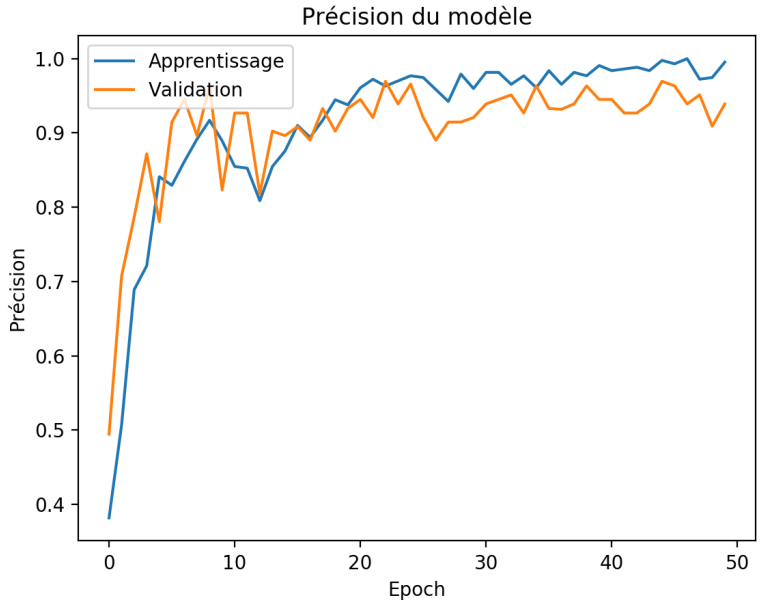
Comme on peut le voir, ce modèle est composé de 6 couches. Les 4 premières couches sont des couches Conv2D, chaque couche Conv2D est composée d'un nombre croissant de filtres, suite à chaque couche on utilise MaxPooling2D pour réduire les dimensions spatiales de la sortie de la couche précédente.

Ensuite, on met à plat la sortie de la dernière couche Conv2D pour pouvoir ajouter une nouvelle couche Dense fully connected (tous les neurones sont connectés à tous les neurones de la couche suivante). Finalement, on ajoute du dropout, c'est-à-dire qu'on ne va pas prendre en compte 25% des neurones et on ajoute une dernière couche avec en nombre de neurones le nombre de classes que l'on cherche à prédire (ici dans mon cas, 6).

Une fois le modèle définit, on peut exécuter la fonction model.compile qui va configurer notre modèle pour l'entraînement.

Avant d'entraîner notre modèle, il nous reste une dernière étape, c'est d'effectuer de la data augmentation afin d'augmenter artificiellement le nombre de données d'entraînement disponibles pour notre modèle.

Suite à cela, on peut entraîner notre modèle et afficher ses performances sur les jeux de données de train et de validation.



Pour la partie test, il a fallu créer un nouveau fichier qui va s'occuper de charger notre modèle obtenu suite à l'exécution du fichier train.py et de vérifier son efficacité sur les images du répertoire de test.

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

img_width, img_height = 120, 120

img_dir = "./data/test"
resu_dir = "./data/result"

if not os.path.exists(resu_dir):
    os.mkdir(resu_dir)

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()
```

La première étape est de définir les paramètres nécessaires à la bonne exécution de notre test. Pour cela, je définis les dimensions des images, les chemins vers mon dossier de test et mon dossier de résultat (où les images seront catégorisées par classe).

Ensuite, je calcule dynamiquement le nombre d'images présentes dans le dossier de test et j'initialise les matrices batch_holder et y_true à zéro.

La dernière étape est de regrouper mes classes dans un tableau et de trier ces classes. L'étape de tri est extrêmement importante, sans elle les résultats obtenus deviennent totalement incohérents.

```
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)
```

Pour la suite du code, il suffit de parcourir le dossier test et de charger chaque image dans la matrice batch_holder, également on initialise la matrice y_true (notre gold standard) avec les valeurs réelles des classes de chaque image.

Finalement, on initialise y_pred avec les résultats de la prédiction des classes de notre modèle sur la matrice batch_holder et on affiche les matrices de classification et de 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
```

L'ultime étape de cette partie du code est de copier les images du dossier test vers le dossier result et de les mettre dans le sous-dossier correspondant à leur classe.

4. Analyse des résultats

Voici le résultat d'exécution du fichier test pour mon modèle :

	precision	recall	f1-score	support
art_dino	1.00	1.00	1.00	30
fitness	0.92	1.00	0.96	60
obj_decoys	0.84	0.87	0.85	30
obj_dish	0.67	0.97	0.79	30
obj_train	1.00	0.86	0.92	90
pet_cat	0.96	0.77	0.85	30
accuracy			0.91	270
macro avg	0.90	0.91	0.90	270
weighted avg	0.92	0.91	0.91	270

Figure 6 : Rapport de classification

Comme on peut le voir sur la figure ci-dessus, les résultats sont globalement très bons, seul la classe obj_dish (représentant de la vaisselle) pose un peu plus de difficultés au modèle notamment pour la différencier par rapport à la classe obj_decoys (représentant des leurres en forme de canard).

5. Critique du travail

Je pense que les classes choisies ont une grande influence sur les résultats du modèle. Je ne suis pas sûr que j'aurai obtenu le même résultat avec des classes choisies aléatoirement et qui pourrait donc potentiellement se ressembler assez fortement. Je n'ai pas fait de comparaison entre les résultats pour un même modèle avec différentes classes données en entrée mais ce serait un axe d'amélioration à envisager.

Partie 4 : Conclusion

En conclusion, j'obtiens des résultats satisfaisants pour chacune des parties du projet, cependant, comme je l'ai dit dans la partie critique pour chaque projet je pense que mes bons résultats sont assez fortement liés au fait que j'ai pris soin de bien choisir des classes très différentes afin de faciliter le travail des algorithmes.

Ce projet était tout de même assez complexe, du le fait que j'étais débutant avec le langage Python et dans l'utilisation des réseaux de neurones et des librairies associés. J'ai également rencontré quelques difficultés liées aux librairies (notamment TensorFlow et Keras) mais j'ai pu régler ces problèmes plutôt rapidement.