Détection et reconnaissance des panneaux de signalisation

Aude Pertron, Jérémy Le Joncour

1. Introduction

Le but de ce projet est de créer un modèle permettant de détecter et de reconnaitre des panneaux de signalisation. Un premier modèle a été réalisé afin de distinguer les différents panneaux, et un deuxième pour la détection des panneaux en général. Les deux sont ainsi combinés afin de répondre à la problématique.

Nous remercions Paul, Pereg et Thomas de nous avoir transmis leur modèle de détection de panneaux basé sur le guide de Gilbert Tanner. Plusieurs obstacles, notamment sur la compatibilité des versions de Tensorflow et Numpy, ne nous ont pas permis de poursuivre l'entrainement du modèle.

2. Modèle de reconnaissance des panneaux

2.1. Définition de nos paths

A la différence du Brief sur la détection des masques, le nombre de classe des types de panneau est supérieur à 2 (43 comptabilisés). Un travail a donc été réalisé afin de définir les catégories de nos *datasets*. Afin de s'assurer que les paths ont bien été inscrit, un affichage des images a été effectué :

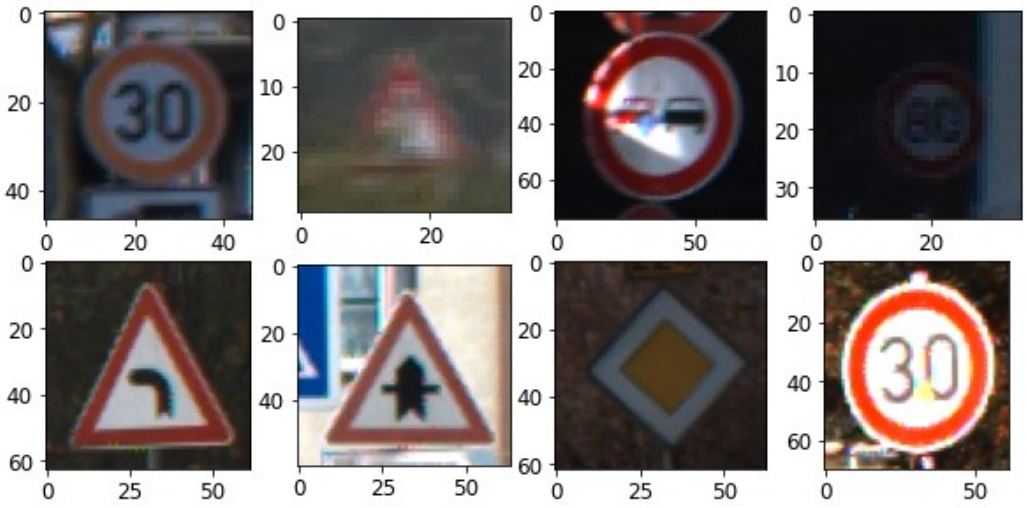


Figure 1. Visualisation d'images aléatoire (test) via matplotlib.image

2.2. Préparation de nos données d'entrainement

Une double boucle a été faite afin d'incorporer des images traitées (*features*) et leur label dans deux listes *image_data* et *image_labels.* La première boucle permet de créer un chemin vers un des dossiers de classe de panneau (de 0 à 42), et une deuxième boucle pour traiter chaque image (sous CV2) et les injecter dans les deux listes citées précédemment. Ces listes ont ensuite été transformées en *array* et leur dimension ont été vérifiées.

2.3. Création des *features* x et *target* y Apprentissage (app) et Evaluation (val)

Les listes ont été randomisées par un *random.shuffle* et les données ont ainsi été séparées en deux sets d'entrainement (80%) et de test (20%). Les dimensions ont été vérifiées après traitement avant de les utiliser pour notre modèle de classification.

```
x_app.shape :     (31367, 23, 23, 3)
x_val.shape :     (7842, 23, 23, 3)
y_app.shape :     (31367,)
y_val.shape :     (7842,)
```

One Hot Encoding sur les labels y

Lors du lancement du premier modèle, une erreur est survenue sur les dimensions des Targets. Nous utilisons le *One hot encoding* de *Keras* pour redimensionner les labels.

```
y_app.shape :     (31367, 43)
y_val.shape :     (7842, 43)
```

2.4. Création du modèle CNN

Notre modèle contient 3 couches :

- Pour l'initialisation de notre modèle, les couches Conv2D sont utilisées sur les traitements d'objets bidimensionnelles, et contiennent différentes classes comme les filtres (dont le nombre augmente vers les couches de sortie), avec une définition de kernel de dimension (3,3) idéale pour des images ayant une taille inférieure à *128x128* pixels. L'input_shape reprend les dimensions des images que nous fournissons à notre modèle (*23x23* pixels en 3 couleurs RGB). Enfin, nous avons définis une activation reLU, l'activation linéaire standard qui permet de fixer les valeurs négatives de nos matrices à 0. MaxPooling2D réduit les dimensions des images injectées et conserve les traits principaux.

- Les dernières couches Flatten et Dense font la liaison entre les couches précédentes et convertissent les données en matrice à 1 dimension. Le but de Dropout est d'aider le réseau à se généraliser et peut régler les problèmes de sur-apprentissage. Les neurones de la couche actuelle, avec une probabilité ici de 0.5, déconnecteront de manière aléatoire des neurones de la couche suivante afin que le réseau doive s'appuyer sur les connexions existantes.

Nous utilisons enfin l'activation Softmax, performant pour la classification multiple. Enfin, nous compilons le modèle avec Adam.

Augmentation du set de données d'apprentissage

ImageDataGenerator réalise de la data augmentation, permettant l'augmentation de la taille du set d'apprentissage en rajoutant des effets sur les images existantes (inclinaisons, flou, changements de luminosité...).

Evaluation du modèle (Historique)

Notre Score de Précision atteint près de 98%, et celui réalisé sur nos données de Validation près de 1. Obtenant des résultats satisfaisants, nous pouvons tester notre modèle sur nos données de Test, n'ayant pas servi à l'entraînement de ce dernier.

```
Epoch 9/10
981/981 [==============================] - 105s 107ms/step - loss: 0.0803 - accuracy: 0.9762 - val_loss: 0.0385 - val_accuracy:
0.9932
Epoch 10/10
981/981 [==============================] - 104s 106ms/step - loss: 0.0706 - accuracy: 0.9783 - val_loss: 0.0143 - val_accuracy:
0.9960
```

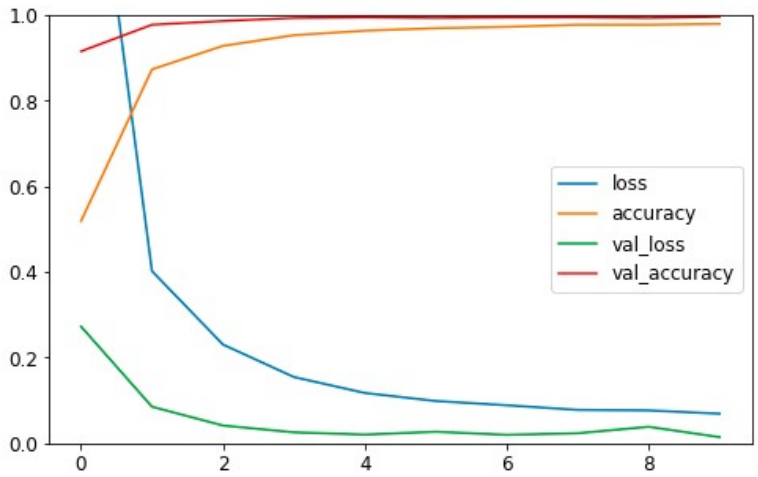


Figure 2. Historique de l'entrainement du modèle CNN

2.5. Chargement des données Test et Evaluation du modèle

Le même processus a été réalisé pour créer une liste nommée *data* contenant les images Test (*resize*) et leur Label.

Une prédiction a été réalisée sur les images Test à partir de notre modèle. La précision de notre modèle sur nos données Test atteint 97%. Nous pouvons visualiser la classification de ces images à travers une matrice de confusion. Celle-ci représente la répartition des images en fonction de leur classe avérée et la prédiction du modèle.

Une visualisation sur des images Test choisies aléatoirement peut être présentée :

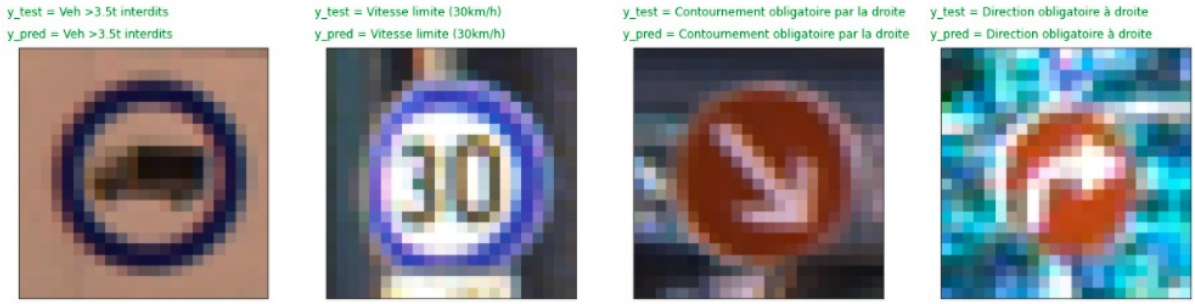
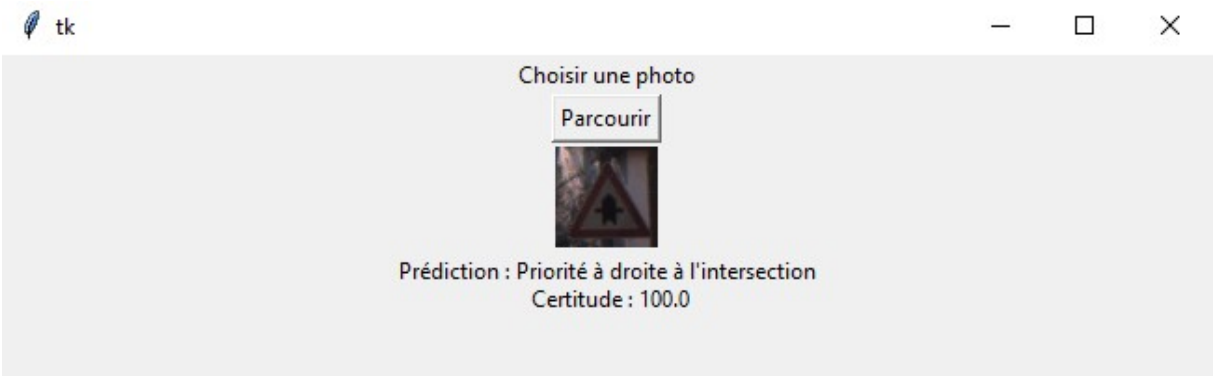


Figure 3. Visualisation d'images Test aléatoires avec leur Label et prédiction du modèle (en vert : correspond à une bonne prédiction de ce dernier, au contraire, le Label et la prédiction seraient affichés en rouge).

2.6. Interface graphique (sur la classification de panneau)

Afin de facilité la vérification de notre modèle, une interface Tkinter (*test_tk.ipynb*) a été mise au point par les soins de Aude et présente la probabilité de la du classement du panneau. L'association du modèle de détection sur le *dataset* du **The German Traffic Sign Detection Benchmark** (au chapitre 3.2) n'a pas pu être testée.



3. Modèle de détection des panneaux

Maintenant que notre premier modèle classifie correctement les différents panneaux, la deuxième partie consiste à créer un modèle capable de détecter les panneaux dans le paysage routier/urbain. Après un travail de recherche sur la documentation faisant mention de la détection d'objet, nous nous sommes tournés sur la création d'un modèle de détection d'objet personnalisé à partir de l'API de détection d'objets Tensorflow 2.

Le guide de Gilbert Tanner (lien du blog) nous a permis de créer un tel modèle de détection de panneau.

3.1. Installation

Après clonage du package dans l'invite de commande d'Anaconda (Conda Prompt en environnement virtuel), nous avons configuré les différents fichiers et script. Les fichiers *protos* ont ensuite été convertis en *python*, toujours en suivant la démarche. Le Protocol Buffers est un langage de définition de données créé par Google. Sa syntaxe, basée sur celle du langage C, évoque celle de JSON, avec pour différence l'utilisation de variables typées.

```
cd models/research
# Compile protos.
protoc object_detection/protos/*.proto --python_out=.
# Install TensorFlow Object Detection API.
cp object_detection/packages/tf2/setup.py .
python -m pip install .
```

```
python use_protobuf.py <path to directory> <path to protoc file>
```

```
python object_detection/builders/model_builder_tf2_test.py
```

```
[ RUN      ] ModelBuilderTF2Test.test_session
[  SKIPPED ] ModelBuilderTF2Test.test_session
[ RUN      ] ModelBuilderTF2Test.test_unknown_faster_rcnn_feature_extractor
[       OK ] ModelBuilderTF2Test.test_unknown_faster_rcnn_feature_extractor
[ RUN      ] ModelBuilderTF2Test.test_unknown_meta_architecture
[       OK ] ModelBuilderTF2Test.test_unknown_meta_architecture
[ RUN      ] ModelBuilderTF2Test.test_unknown_ssd_feature_extractor
[       OK ] ModelBuilderTF2Test.test_unknown_ssd_feature_extractor
----------------------------------------------------------------------
Ran 20 tests in 91.767s

OK (skipped=1)
```

3.2. Les données utilisées

Les données proviennent du site The German Traffic Sign Detection Benchmark (lien du site)**.** Le *dataset* (FullIJCNN2013) contient entre autre de nombreuses images de panneaux routiers au format ppm ainsi qu'un fichier texte (*gt.txt*) qui comprend les différents labels de chaque image.

Pour que ces images soient traitées, un notebook (*label_csv.ipynb*) a été réalisé. Le fichier *gt.txt* a été converti en deux fichiers csv *test_labels.csv* et *train_labels.csv* et représenté ainsi :

	filename	xmin	ymin	xmax	ymax	class	width	height
0	00000.jpeg	774	411	815	446	1	1360	800
1	00001.jpeg	983	388	1024	432	1	1360	800
2	00001.jpeg	386	494	442	552	1	1360	800
3	00001.jpeg	973	335	1031	390	1	1360	800
4	00002.jpeg	892	476	1006	592	1	1360	800
...	...	...	...	...	...	...	...	...
1024	00746.jpeg	1138	537	1182	579	1	1360	800
1025	00747.jpeg	298	489	345	538	1	1360	800
1026	00747.jpeg	1148	506	1196	553	1	1360	800
1027	00749.jpeg	301	551	346	596	1	1360	800
1028	00749.jpeg	1153	546	1200	593	1	1360	800

1029 rows × 8 columns

Les images ont été converties en format jpeg afin qu'elles soient traitées sur *LabelImg*. *LabelImg* est un logiciel permettant d'étiqueter les images en procédant à l'encadrement des objets à détecter.

3.3. Préparation des données pour l'API OD

Les données étiquetées sont converties dans un format que l'API Tensorflow OD peut utiliser. L'API OD fonctionne avec des fichiers au format TFRecord, un format simple pour stocker une séquence d'enregistrements binaires.

Le processus de conversion des données au format TFRecord varie pour différents formats d'étiquettes. A travers le guide de Gilbert Tanner, nous avons généré un TFRecord qui créer deux fichiers (*train.record* et *test.record*), qui peuvent être utilisés pour entraîner notre détecteur d'objet.

3.4. Création du Labelmap pour l’entrainement du modèle

Nous avons utilisé 2 modèles différents, nous n'avons donc qu'une seule catégorie dans notre Labelmap (« Panneau »). Nous le placerons dans un dossier appelé *training*, qui se trouve dans le répertoire *object_detection*. Le numéro d'identifiant de chaque élément doit correspondre à l'identifiant spécifié dans le fichier *generate_tfrecord.py*.

3.5. Création de la configuration de l’entrainement du modèle

Un fichier de configuration a été utilisé. Comme modèle de base, *EfficientDet*, une famille récente de modèles SOTA découverts à l'aide de Neural Architecture Search, a été choisie. Nous avons modifié certaines lignes afin de diriger les paths de nos images et de modifier le size batch (8).

```
train_input_reader: {
  label_map_path: "C:/Users/utilisateur/Desktop/Tutorial/models/research/object_detection/training/label_map.pbtxt"
  tf_record_input_reader {
    input_path: "C:/Users/utilisateur/Desktop/Tutorial/models/research/object_detection/train.record"
  }
}

eval_config: {
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
  batch_size: 8;
}

eval_input_reader: {
  label_map_path: "C:/Users/utilisateur/Desktop/Tutorial/models/research/object_detection/training/label_map.pbtxt"
  shuffle: false
  num_epochs: 1
  tf_record_input_reader {
    input_path: "C:/Users/utilisateur/Desktop/Tutorial/models/research/object_detection/test.record"
  }
}
```

Le modèle est ensuite lancé via l’invite de commande :

```
python model_main_tf2.py \
    --pipeline_config_path=training/ssd_efficientdet_d0_512x512_coco17_tpu-8.config \
    --model_dir=training \
    --alsologtostderr
```

Pour des raisons d’incompatibilité de Tensorflow et Numpy, malgré les downgrades et les upgrades des différents packages, le modèle de Pereg, Thomas et Paul a été utilisé. Ainsi l’historique de l’entrainement n’a pas pu être vérifié. L’état du modèle aurait été enregistré sur Tensorboard.

4. Evaluation des Modèles

Un script modifié du guide utilisé nous a permis de tester nos 2 modèles. Les modèles ont tout d'abord été importé (*model_detection* et *model_classification*).

- Une première fonction (*run_inference_for_single_image*) a permis de modifier les images pour que notre modèle de détection puisse fonctionner. Elles sont converties en Tensor grâce au *tf.convert_to_tensor* en entrée. La sortie est de retour transformée en array Numpy.

- La deuxième fonction (*create_boxes*) permet de créer les cadres entourant les panneaux de signalisation détectés par le modèle. L'encadrement de l'image au niveau des panneaux détectés est fixé en taille 23x23 pour utiliser le modèle Classification.

- La dernière fonction (*process_image*) permet simplement d'afficher l'image traitée avec nos deux modèles.

5. Conclusion

La problématique de la détection et la classification de panneaux de signalisation nous aura incité à concevoir deux modèles distinctes l'une pour la détection d'objet, à savoir les panneaux, et leur classification. Ce dernier modèle était plus simple à concevoir grâce au *dataset* que nous disposions avec de nombreuses images répertoriées. La précision de ce modèle s'élevait d'ailleurs à 97% ce qui était convenable. Ces résultats sont d'ailleurs représentés sous une interface graphique Tkinter.

Le deuxième modèle, axé sur la détection d'objet, nous aura demandé davantage de temps afin de trouver la documentation nécessaire pour répondre à la problématique du projet. Le guide de **Gilbert Tanner** mentionné précédemment présentait la configuration d'un modèle personnalisable. Avec un *dataset,* répertoriant de nombreuses images de panneaux, téléchargé sur **The German Traffic Sign Detection Benchmark**, nous avons pu configurer nos fichiers pour l'API Tensorflow OD. Toutefois, le lancement de l'entrainement n'a pas pu aboutir du fait de l'incompatibilité de certains packages tel que Tensorflow et Numpy. Le modèle de détection utilisé provient des travaux de nos collègues **Thomas, Pereg** et **Paul** qui nous l'ont gracieusement partagé. Cela nous a permis de poursuivre notre projet et de répondre en partie au Brief.

L'association des deux modèles nous donne des résultats mitigés : Le modèle de détection des panneaux n'est pas robuste. Certaines images présentant plusieurs panneaux n'aboutissent qu'à la détection d'un seul panneau avec une reconnaissance de la classe plutôt hasardeux (malgré une précision importante de notre modèle de classification). Ces résultats ne nous ont pas conforté à poursuivre vers l'intégration vidéo.

Afin de palier à ces problèmes (compatibilité du modèle de détection, vérification de la classification), plusieurs solutions sont présentées. La résolution des soucis de compatibilité des différents packages peut être palier par l'utilisation d'environnement virtuel comme l'empaquetage **Docker**, ou encore une utilisation de **Colab**. La vérification de la prédiction des panneaux peut être aussi réalisée par un *predict_classes* sur les photos des différents paysages routiers.

Enfin, d'autres solutions pourront être apportées en créant un modèle « tout en un » qui pourrait potentiellement effectuer une reconnaissance plus fine de ces objets. La documentation importante des recherches qui ont pu être faites spécifiquement sur la détection de panneaux de signalisation routière, un cas classique de vision par ordinateur, pourrait répondre à cette problématique (modélisation géométrique, détecteur photométrique proposé par Timofte et *al.*, 2009, *AdaBoost Forest-ECOC* etc...).