Rapport de projet

Classification d'images basée sur les approches de Deep Learning

Sylvain RAJKOUMAR – Thomas BRASEY – William ZERBATO
Tuteur : M. Madjid MAIDI
ESME SUDRIA 2017 – 2018
2C

Sommaire

I. Introduction

1. Problématique

De nos jours, grâce à l'émergence du Big Data dans le monde, la problématique actuelle est de collecter le plus de données possibles et plus vite, afin d'en tirer un maximum d'informations utiles pour des applications diverses et variées.

2. Objectifs

C'est donc dans cette optique qu'il nous a été demandé de mettre au point une application mobile permettant de classifier et traiter des images de milieu urbain. Ces données extraites des images pourront être utilisées par exemple dans une application Big Data pour détecter une hausse de trafic sur une route, ou une affluence de piétons dans une rue ou zone piétonne.

Pour cela on nous propose d'utiliser des outils d'apprentissage automatique, ou Deep Learning tels que les Réseaux de Neurones Convolutifs (CNN) en plein essor ces temps-ci.

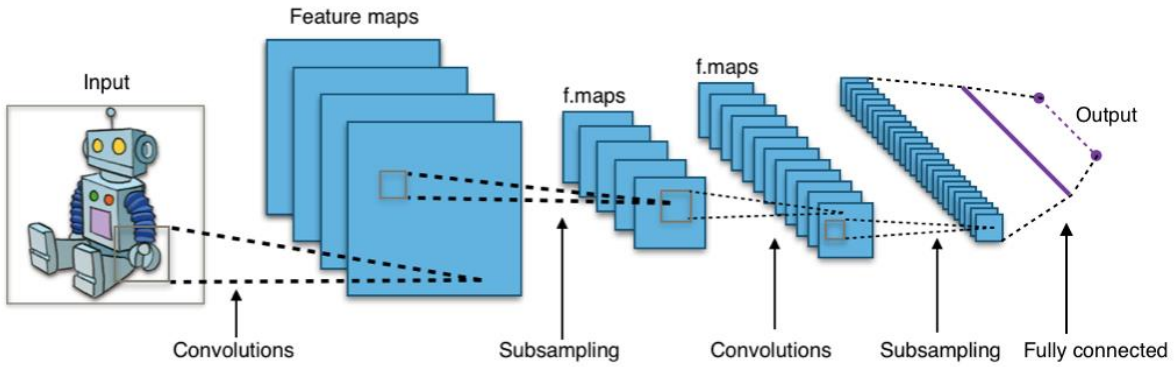
II. Méthodes

1. Réseaux de neurones convolutifs

Un réseau de neurones artificiel est un système/une structure qui reproduit le fonctionnement des neurones du cerveau humain. Il existe actuellement différents types de réseaux de neurones qui sont subdivisée en deux grandes catégories d'apprentissages : supervisé et non supervisé.

Dans notre cas où la classification d'images est requise, il faut utiliser un réseau de neurones convolutifs qui est une méthode utilisant l'apprentissage profond/Deep Learning (où la structure est composée de plusieurs couches intermédiaires). Un réseau de neurones a une matrice de probabilité de sortie qui lui est propre, il a été entrainé pour reconnaitre un certain nombre d'objets.

Le réseau de neurones convolutifs prend généralement en entrée une image et renvoie en sortie une matrice de probabilité. Cela permet donc de savoir avec précision la classe de l'objet présent dans l'image.



Représentation d'un réseau de neurones convolutifs (Wikipedia)

Cependant, ces types de réseaux de neurones ont des performances assez limitées et ne permettent pas de faire du traitement en temps réel et avec précision. Avec ces structures, des milliers d'itérations sont effectués sur l'image ce qui consomme une quantité importante de ressources.

Ces frameworks tels que Tensorflow, Caffe et Torch ont donc certaines faiblesses. Un nouveau modèle a récemment été introduit, le framework YOLO, il permet d'effectuer un nombre très réduit d'itérations sur l'image et est ainsi 100 à 1000 fois plus rapide que ses prédécesseurs.

Ce modèle permet en plus de la détection d'objets multiples, la localisation de ces objets dans l'images. Cette localisation apporte une information très importante dans le milieu urbain (figure 1).

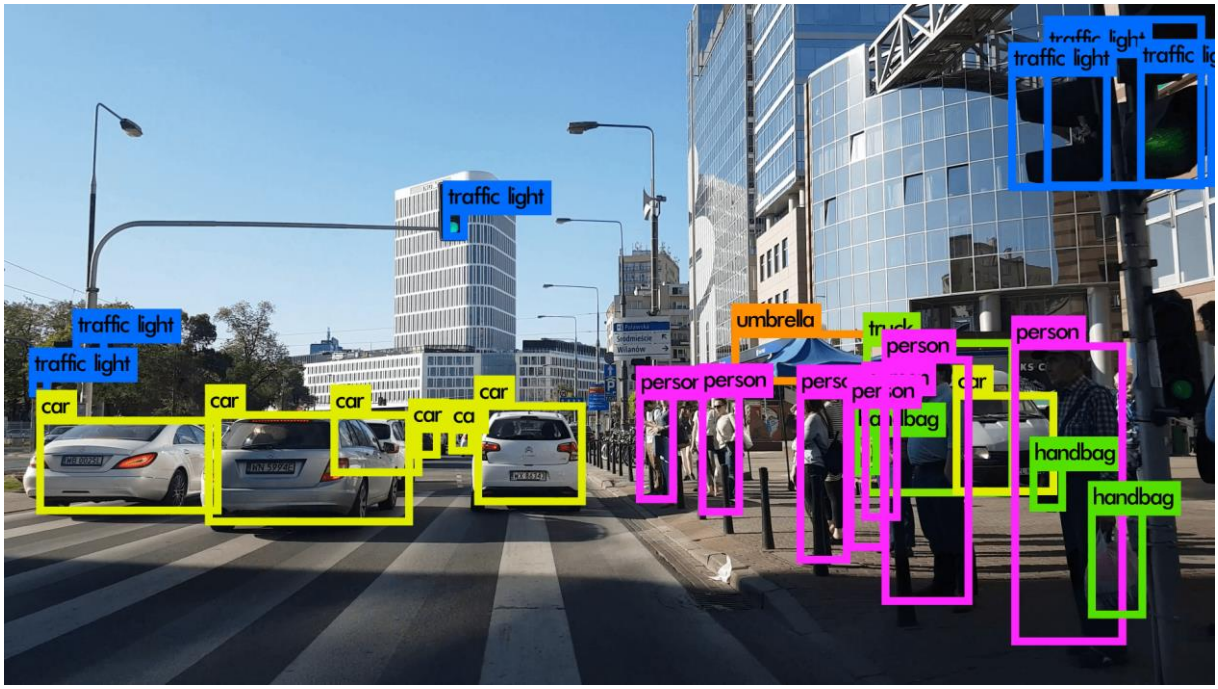


Figure 1 - Résultat de détection en utilisant Yolo.

2. Classifieur Haar cascades

Un classifieur Haar cascade moins complexe qu'un réseau de neurones. Ce classifieur repose sur un ensemble de masques définis pour détecter différentes caractéristiques de l'image (figure 2). Un masque est de 20x20 pixels, il est appliqué à toutes les positions de l'image à différentes échelles.
La convolution de ces masques avec les imagettes de l'objet d'intérêt permet de coder l'image et déterminer ainsi la caractéristique de Haar de l'image.

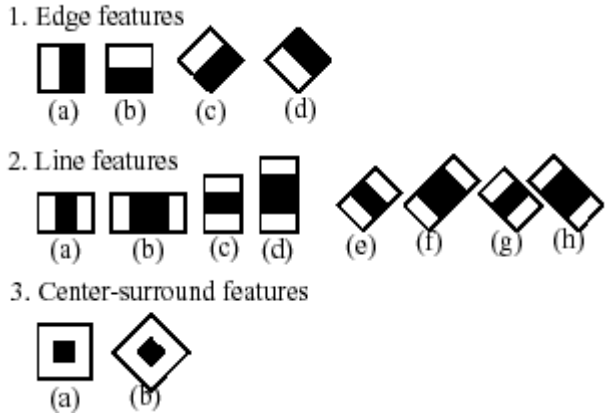


Figure 2 - Caractéristiques de Haar

Le classifieur de visage a été implémenté en utilisant la méthode de Viola et Jones basée sur une approche par recherche de visage dans toutes les positions et à plusieurs échelles (figure 3). Afin de réduire le temps de calcul, l'algorithme de détection applique plusieurs classifieurs en cascades qui permettent de rejeter ou d'accepter le résultat de la détection et de passer ou non classifieur suivant.

Le modèle Haar cascade du visage a déjà été entrainé et le codage de Haar est disponible dans un fichier xml. Cependant cette détection n'est pas robuste aux variations d'angle de l'objet et possède donc des limites.

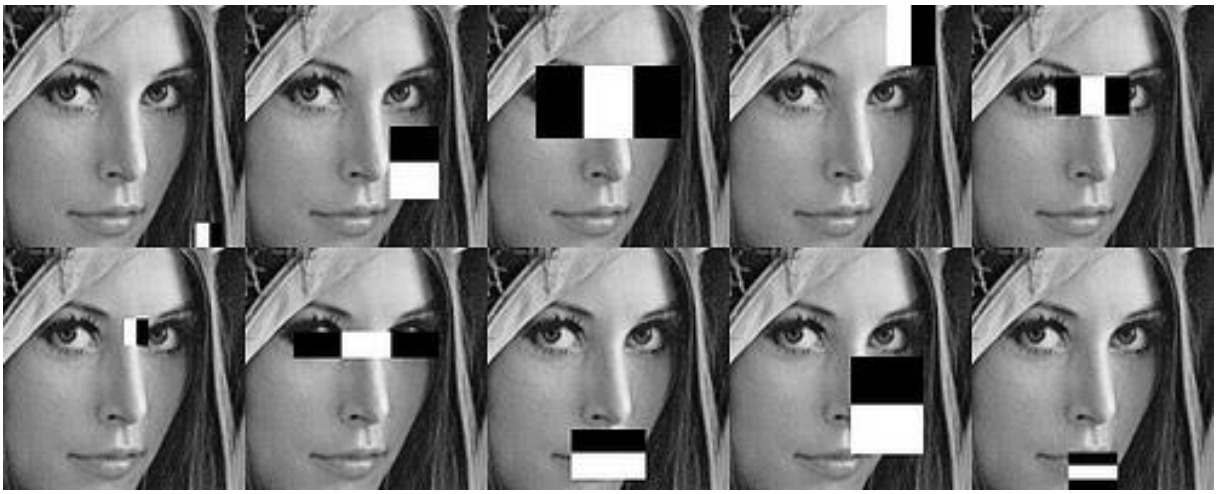


Figure 3 - Détéction de visage en utilisant le Haar cascade.

III. Outils

1. Environnement de développement et langage

Concernant les choix techniques, nous avons utilisé des langages performants dont la documentation et implémentation sur OpenCV est complète.

Langages

- **C++ :**

  Le C++ est un langage de programmation compilé. C'est un langage bas niveau et très typé qui offre donc de bonnes performances en termes de vitesse d'exécution. Cela en fait un bon langage dans le milieu scientifique et/ou pour des applications temps réel dans lesquels la précision et la rapidité par l'optimisation sont de mise.

- **Python :**

  Le Python est un langage de programmation interprété et haut niveau, ce qui permet un portage des applications d'une plateforme à une autre plus facilement. Cependant il n'est pas aussi performant que le C++ dans le domaine du calcul scientifique. Dans notre projet il a principalement été utilisé pour la réalisation d'un « Image Crawler » permettant la récupération d'images depuis le web. D'autres scripts ont aussi été réalisés afin d'obtenir un fichier .xml. Sa syntaxe simple permet donc un prototypage assez rapide.

Environnements

- **Visual Studio :**

  Microsoft Visual Studio est un environnement de développement intégré (IDE). Il possède de nombreux modules pour développer toutes sortes d'application en C, C++, C#, Visual Basic, etc. De plus, il permet une gestion simple des dépendances, un environnement de débogage avancé et un environnement de travail complet.

- **VirtualBox – VM Ubuntu :**

  VirtualBox est un outil de virtualisation développé par Oracle. Il permet sur une machine hôte de faire tourner plusieurs autres systèmes. Dans notre cas, les scripts en Python ont été réalisés sous une machine virtuelle ayant pour système d'exploitation Ubuntu. De plus nous avons générer des classifieur à l'aide d'exécutables OpenCV.

- **Android Studio :**

  Android Studio est un environnement de développement intégré (IDE) similaire à Eclipse dédié à la création et au développement d'applications sous Android. Cet environnement nous a été utile au déploiement de l'application sous Android. Il nous a permis d'intégrer des fichiers sources C++ au projet Android à l'aide de divers modules tels que le Java Native Interface et Android Native Development Kit.

2. Librairie OpenCV

OpenCV est une bibliothèque logicielle open source écrite principalement en C++, Python et Java et destinée aux plateformes Windows, Android, IOS, Linux, Mac OS, etc.

Créée en 1999 par Intel, elle est reprise par une organisation à but non lucratif en Aout 2012 et passe dans le domaine open source.

Cette bibliothèque propose de nombreuses fonctionnalités permettant le traitement d'images et vidéos. Elle est aussi utile au domaine scientifique grâce aux divers algorithmes d'apprentissage et calculs matriciels.

Exemples d'application : Reconnaissance faciale, Reconstruction 3D, Réalité augmentée, etc.

- **CMake :**

  CMake est un logiciel permettant de générer, à partir d'un fichier de configuration (appelé CMakeLists.txt), une librairie. Dans notre cas, à partir des fichiers sources d'OpenCV et OpenCV_contrib disponibles sur leur GitHub (+ GitHub_contrib), nous avons généré une librairie nous permettant d'avoir accès à tous les modules disponibles.

IV. Déroulement du projet

1. Différentes phases du projet

Installation des outils

La première étape de notre projet à été l'installation des outils nécessaires cités au chapitre précédent. Nous avons généré les fichiers binaires .dll et .lib (représentant une bibliothèque de lien dynamiques) à partir des fichiers sources de OpenCV et OpenCV_Contrib disponible sur GitHub.

Au début du projet, certains modules dont le module Deep Neural Network utile à notre projet, n'était pas présent dans les fichiers sources d'OpenCV de base mais était présent dans OpenCV_Contrib.

Le dépôt GitHub OpenCV_Contrib contient des modules en cours de développement et donc potentiellement instable et incomplet. Cela explique qu'ils ne font pas partie du dépôt principal.
Nous avons donc utilisé l'outil CMake afin de générer une nouvelle librairie contenant tous les modules nécessaires.

Phase de recherches

Ayant très peu de connaissance dans le domaine des réseaux de neurones, nous avons donc chercher à comprendre la plupart des concepts de bases sans partir dans les détails assez complexes.
Nous avons trouvé une série de 3 vidéos qui nous a semblé être le meilleur format afin d'en apprendre plus sur la phase d'apprentissage et d'utilisation des réseaux de neurones. Ces vidéos vulgarisent très bien les différents concepts.

Passage à la pratique et aux premiers tests

Le sujet compris il nous fallait maintenant confronter la théorie à la pratique. Nos diverses recherches nous ont menés à l'utilisation du framework Caffe dans un premier temps.

L'implémentation dans le code s'est faite très rapidement durant la première semaine. Il nous a fallu réunir 3 fichiers, le fichier de structure du réseau de neurones, le fichier contenant les différents poids de chaque neurone composant la structure et un dernier fichier regroupant toutes les classes pour lesquelles le réseau de neurones a été entrainé. Ces fichiers étant présent en open-source sur différents GitHub, il n'a pas été complexe de les récupérer.

Nos premiers tests ont été effectués à l'aide de la webcam Logitech HD C270. Nous avons basiquement récupéré le flux vidéo provenant de la webcam afin d'appliquer à chaque image une reconnaissance d'objet à l'aide du réseau de neurones Caffe.

L'affichage du résultat à ce stade était assez basique : une fenêtre contenant l'image et la classe d'objet détecté affiché dans la console accompagnée de sa probabilité renvoyée par le réseau de neurones Caffe.

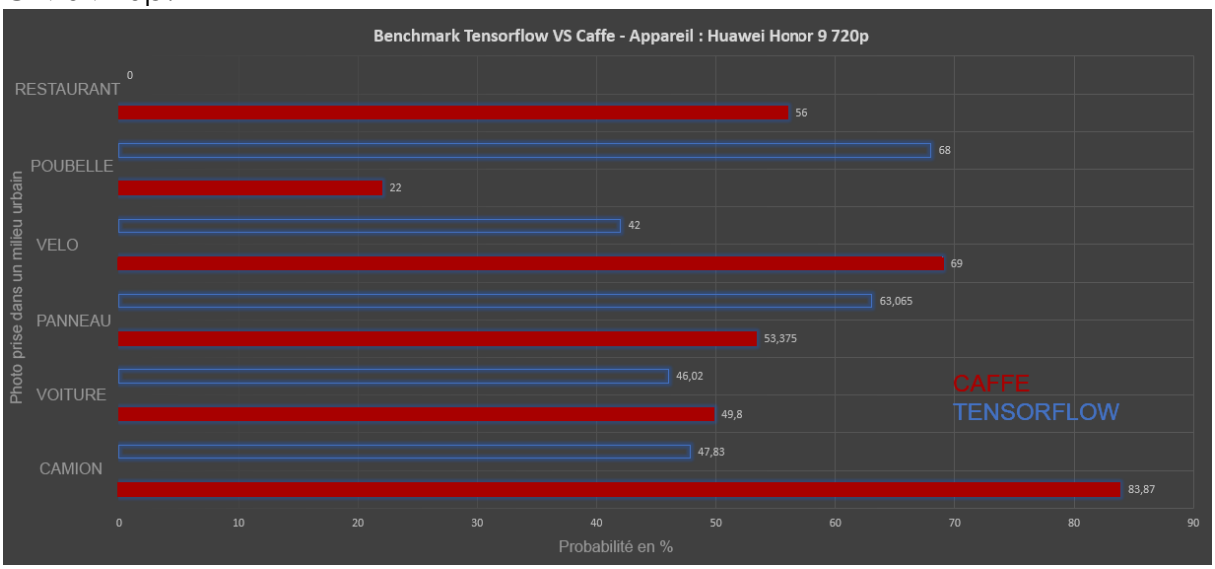
Ces tests ont été réalisés sur divers objets accessible facilement et présents dans la liste des objets pouvant être détectés par Caffe tels qu'un ordinateur portable, une souris d'ordinateur, un écran, une voiture, un animal, etc.

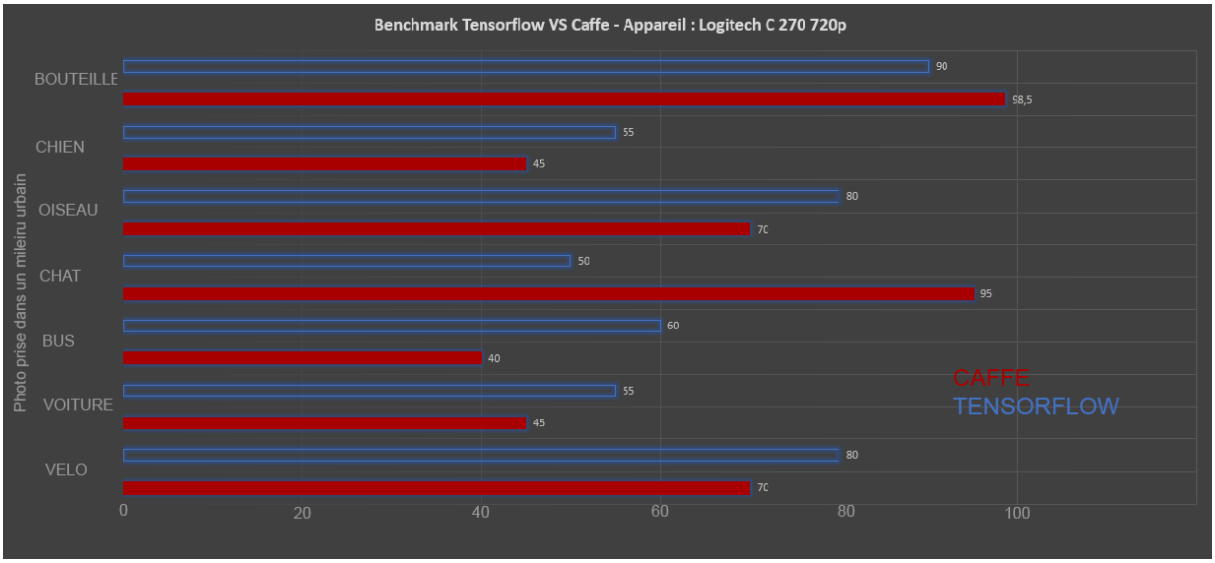
Les résultats étaient parfois approximatifs (~60% de sûreté), mais dans l'ensemble, le framework Caffe a été capable de détecter avec précision une multitude de classes en nous donnant précisément la race d'un chien, modèle d'une voiture, modèle d'un avion par exemple.

Comparaison Caffe/Tensorflow

Après avoir manipulé en profondeur le framework Caffe, nous avons décidé de comparer ses performances avec un autre framework.

Les frameworks Torch et Tensorflow étaient disponibles cependant certains soucis de compatibilité avec Torch nous ont forcés à l'abandonner.
La comparaison Caffe vs Tensorflow a été faite sur deux appareils, un téléphone portable Huawei Honor 9 720p et une webcam Logitech C270 720p.





Après une étude de ces résultats, nous avons choisis de continuer le projet avec le framework Caffe.

Amélioration du projet et utilisation d'un réseau de neurones convolutifs très récent

À la suite de quelques recherches, nous avons découvert qu'une nouvelle méthode de reconnaissance d'objets YOLO (YouOnlyLookOnce) était en cours de développement et avait déjà fait ses preuves.

Ce nouveau réseau de neurones est bien plus rapide (environ 100 à 1000 fois) que ses prédécesseurs. Nous avons donc proposé à notre encadrant de réorienter l'objectif du projet et ainsi d'améliorer le projet de base via l'utilisation de ce nouveau réseau de neurones convolutifs.

Cette nouvelle méthode de reconnaissance est accompagnée d'un réseau de neurones convolutifs possédant une structure spécifique permettant en plus de la reconnaissance d'objets, de fournir la localisation de ces derniers (figure 4).

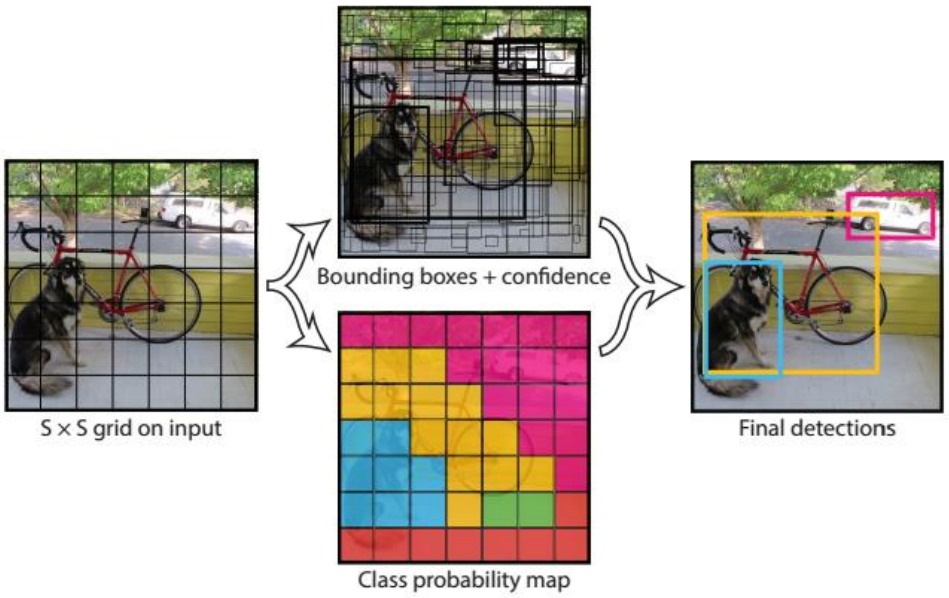


Figure 4 - Résultat de détection en utilisant YOLO.

C'est une amélioration assez considérable car le framework Caffe permettait seulement de fournir le nom de l'objet présent dans l'image, contrairement à la structure YOLO permettant de détecter de multiples objets en fournissant la zone de détection autour des objets.
Cette information est d'autant plus importante dans notre contexte de détection d'objets en milieu urbain.

Nous avons gardé le framework Caffe en association avec le framework YOLO mais Caffe n'étant plus utile et pertinent, nous l'avons abandonné pour se concentrer sur le fonctionnement de YOLO.

Premiers pas avec YOLO

Ce réseau de neurones existe sous différentes variantes. La première que nous avons voulu utiliser étant Yolo9000 capables de détecter 9000 classes différentes. Cependant la structure de ce réseau n'étant pas supporté par OpenCV, nous avons dû nous tourner vers la structure YoloV2 pouvant détecter un nombre de classes différentes très limité mais suffisant à notre projet. YoloV2 permet de détecter la plupart des objets présents dans le milieu urbain.
Nos premiers essais ont été réalisés à l'aide d'une webcam pointée sur une rue :

Figure 5 - Résultat de détection en utilisant YOLO première version.

Cependant nous nous sommes rendu compte que la résolution et la qualité des images fournis par la webcam étaient trop faible. Nous avons donc décidé de remplacer la webcam par des images de meilleures qualités prises par téléphone ou sur le web.
Nous avons aussi privilégié la précision par rapport au temps de calcul. La détection des objets est ainsi devenue plus précise et importante.
A ce stade, la reconnaissance YOLO est satisfaisante.

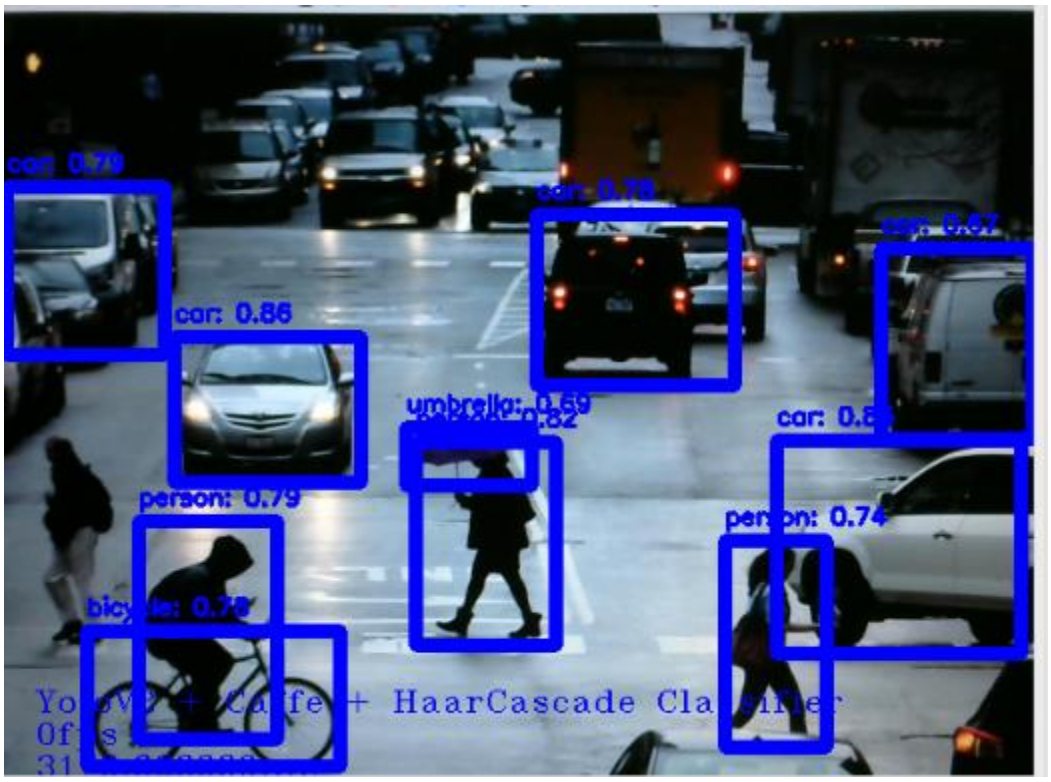


Figure 6 - Résultat de détection en utilisant YOLO deuxième version.

Améliorations de YOLO

Dans le but d'obtenir un meilleur rendu visuel, plus lisible et compréhensible, nous avons apporté quelques modifications à l'affichage. Cet affichage est accompagné du temps de calcul et de l'occurrence des différentes classes.

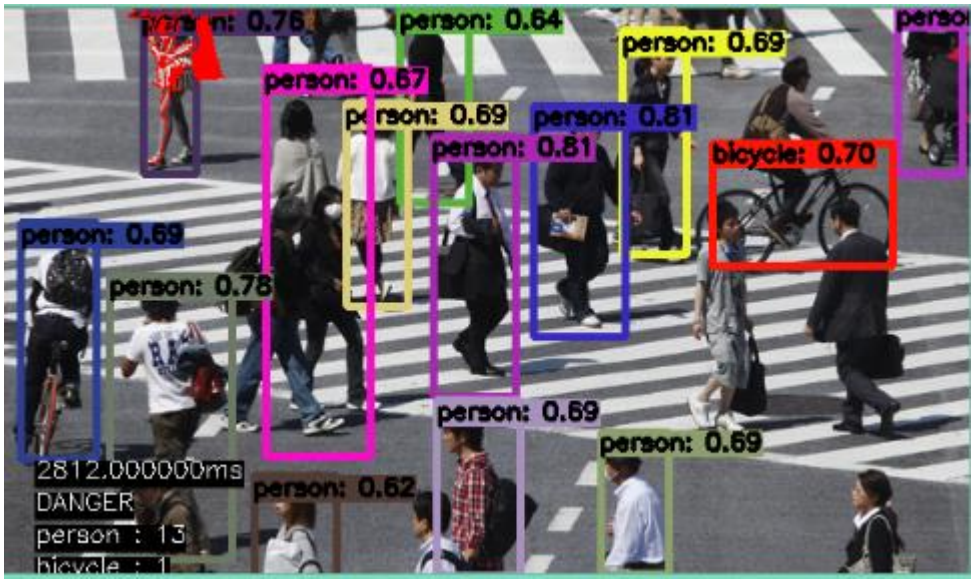


Figure 7 - Résultat de détection en utilisant YOLO dernière version.

Détection de la route

L'objectif étant de récupérer un maximum d'informations à partir d'une image, la détection de la route est donc pertinente.
Cependant tous les pré-traitements de l'image qui s'accompagne avec sont assez complexes.

Afin d'obtenir un résultat valide, il faut passer par diverses calibrations de l'outils de capture, gérer les différentes situations d'éclairement, d'exposition, saturation, teinte de l'image, balances des blancs, détection des formes et transformée de Hough.

Nous avons donc opté pour un traitement assez simple consistant à récupérer les couleurs correspondantes aux marquages de la route, le blanc et le jaune.
Ces différents traitements sont réalisés sur une image en couleur et une autre en niveau de gris. Après récupération de ces pixels, nous utilisons un filtre de Canny permettant la détection de contours et affichons sur l'image originale les lignes de la route dans une couleur spécifique.

Figure 8 - Résultat de détection des lignes de la route

En revanche, le résultat nous a paru peu concluant après de multiples essais, nous avons donc choisis de nous concentrer sur la détection de panneaux.

Haarcascade et détection des panneaux

Le framework YOLO permettant seulement de détecter les panneaux stop, nous avons décidé de rajouter diverses méthodes dans le but de détecter les panneaux de vitesse et les panneaux de danger triangulaire.

Un fichier .xml dédié aux panneaux de vitesse étant déjà présent à l'installation de OpenCV, l'implémentation de ce dernier au code fu rapide.

Figure 9 - Résultat de détection des panneaux de vitesse..

Cette détection permet rapidement d'obtenir la position des différents panneaux de vitesse et comble le manque du framework YOLO. Ces deux méthodes s'associent bien dans notre cas et nous permettent d'obtenir bien plus d'informations depuis l'image.

Quant à la détection des panneaux de danger, nous avons utilisé la détection de formes triangulaires. Plusieurs fonctions présente dans OpenCV permettent de réaliser cela très facilement. Une fois un triangle détecté, il est simplement rempli d'une certaine couleur et nous pouvons afficher ou non un message d'avertissement.

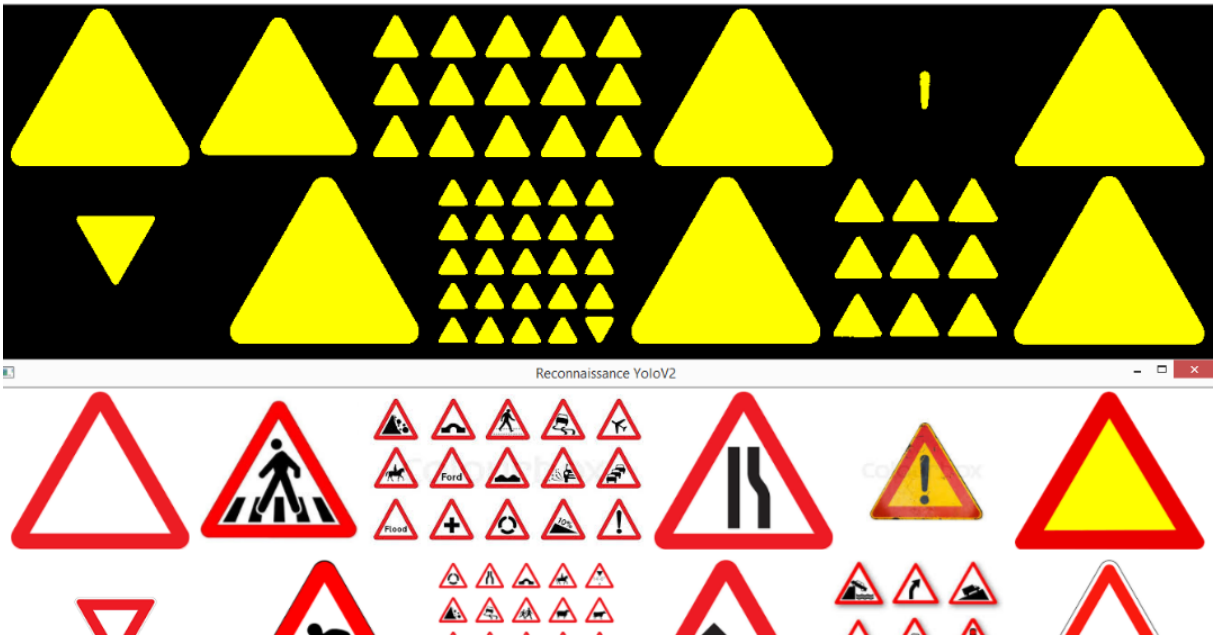


Figure 10 - Résultat de détection des panneaux triangulaires.

Cependant nous avons voulu réaliser cette détection des panneaux de danger en créant nos propres classifieur haarcascade.
Pour ce faire, nous avons donc dû générer un fichier .xml à l'aide de diverses commandes sous une machine virtuelle Ubuntu.

Afin de générer ce fichier, il a fallu collecter un certain nombre d'images positives (50*50) et négatives (100*100). Les images positives contiennent l'objet souhaitant être détecté, les images négatives ne le contiennent pas. Cette collecte d'images a été effectué à l'aide un crawler (script python) depuis un lien de base de données d'image (image-net).
Nous appliquons un autre script pour retirer les images non valide(lien-mort).

Ensuite nous utilisons des commandes permettant de générer des échantillons pour entrainer notre classifieur de panneaux de danger.

En revanche, le résultat n'est pas parfait comme vous pouvez le voir :

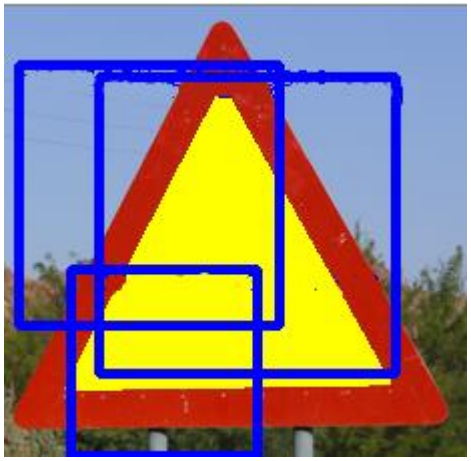


Figure 11 - Résultat de détection de panneaux avec notre propre Haar Cascade

On n'obtient pas précisément la zone désirée comme avec la détection des panneaux de vitesse. Afin d'obtenir un détecteur utilisable et pertinent, il faudrait utiliser une quantité plus importante d'images et de meilleures ressources.

C'est une des principales difficultés que nous avons rencontrés au cours de ce projet. Après plusieurs essais, le détecteur fournissait toujours de mauvais résultat.

Déploiement sous Android

Le déploiement sous Android s'est réalisé avec l'aide de notre encadrant. Nous avons dû lier les différentes dépendances et ajouter nos fichiers sources C++.

Cependant la principale difficulté durant le déploiement a été la compatibilité du programme ancien fourni par notre encadrant avec notre version d'Android Studio récente. Certaines configurations du programme étaient obsolètes et d'autres parties ont nécessité plusieurs modifications.

2. Résultat final et plus-values

Notre programme final sous Android permet une utilisation facile, interactive et naturelle avec une installation simple de l'application ne nécessitant que quelques validations d'étapes sans avoir besoin d'aucune dépendance externe. Le passage à un nouveau type de réseau de neurones convolutifs plus récent apporte une plus-value très intéressante à notre projet.

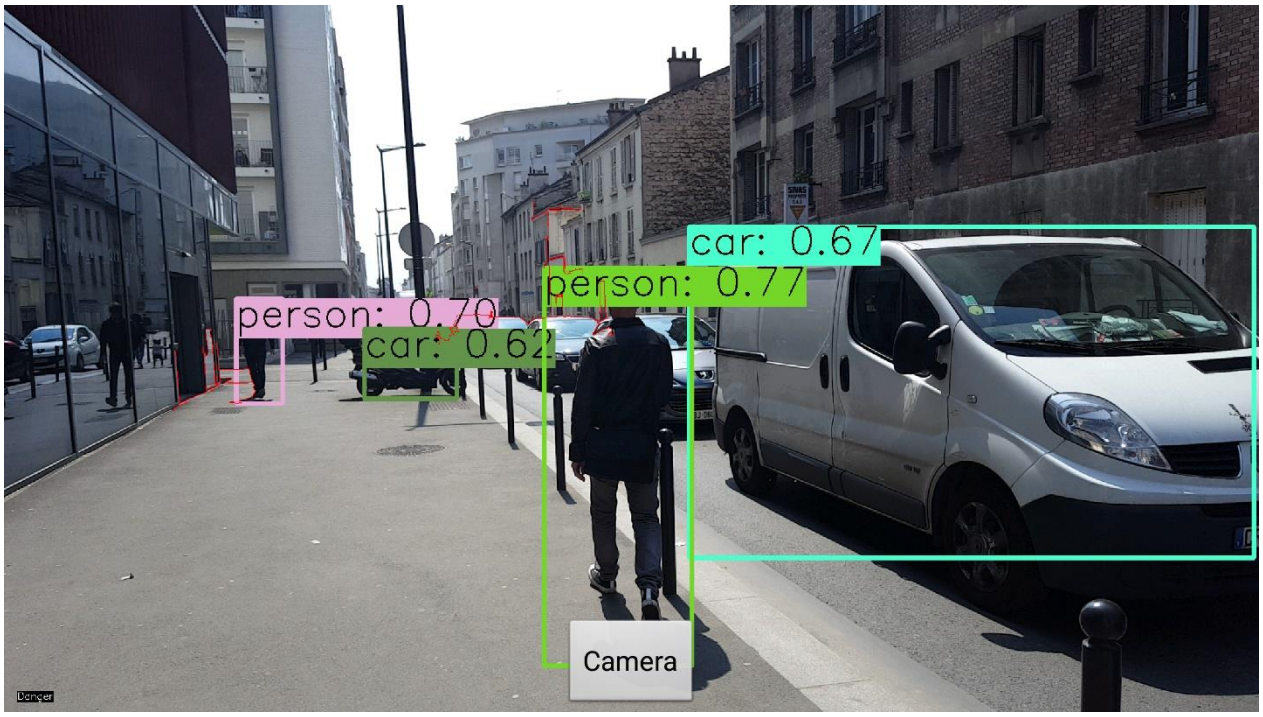
Le CNN implémenté permet de détecter les personnes, vélo, voiture, avion, bus, train, camion, bateau, feu rouge, panneau stop, banc, oiseau, chat et chien.

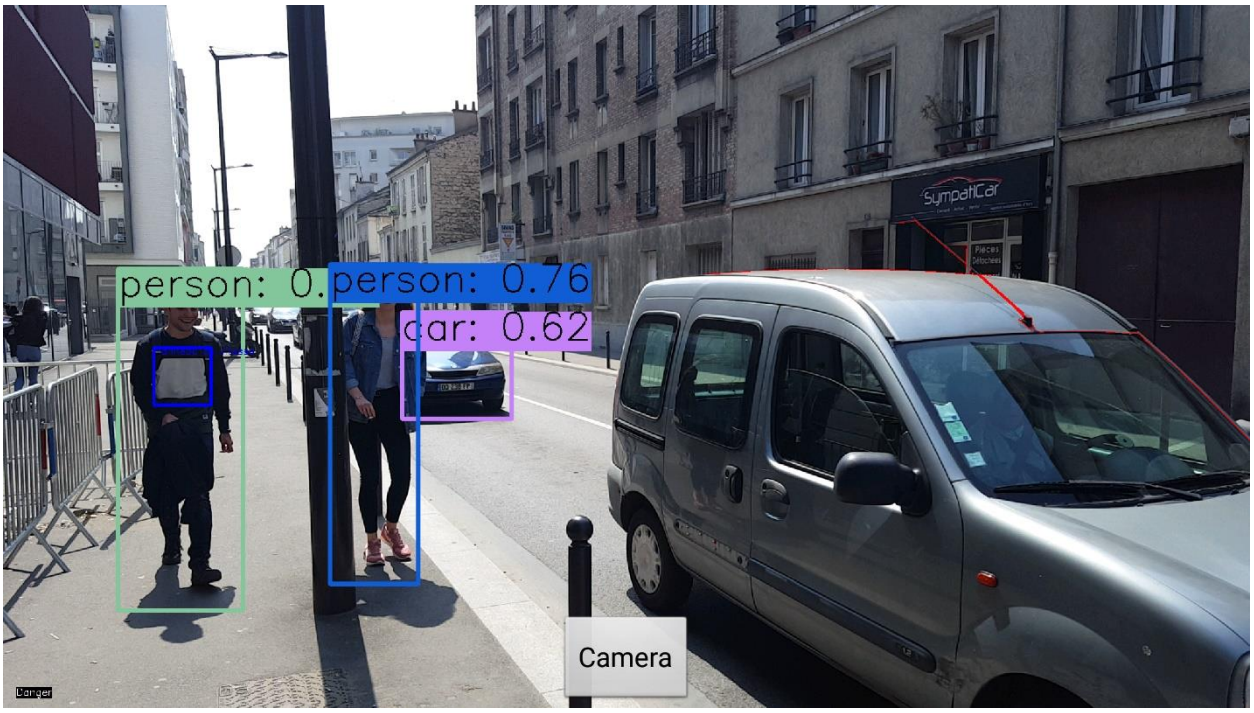
La localisation des objets dans l'image, la détection des panneaux de vitesse et des panneaux de danger constitue une amélioration importante à notre projet. Ces différentes informations peuvent donner lieu à de nombreuses idées d'application. Il y a une grande marge d'améliorations à apporter à notre logiciel tant au niveau de l'architecture du logiciel que l'interface.

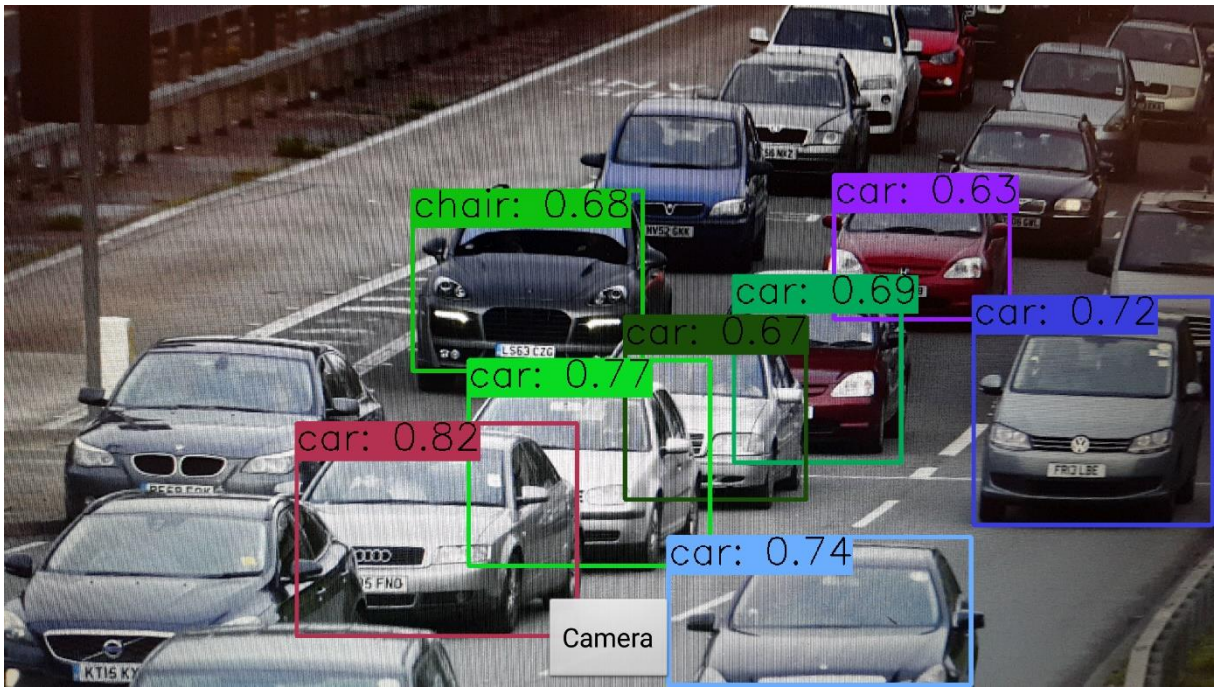
Le rendu final nous semble suffisamment abouti pour un prototype, il nous a permis d'étudier une multitude de techniques dans le domaine du traitement de l'images et du machine learning de manière générale.

Notre programme a ainsi respecté et rempli tous les objectifs de bases fournis par notre encadrant au début du semestre dans le cahier des charges. Nous avons en plus rajouté diverses fonctionnalités apportées par le framework YOLO et divers classifieurs.

Voici ci-dessous différents tests réalisés sur des véhicules, personnes et signalisations. La plupart des objets détectés ont une probabilité d'environ 70 % quelque soit le type d'objet.

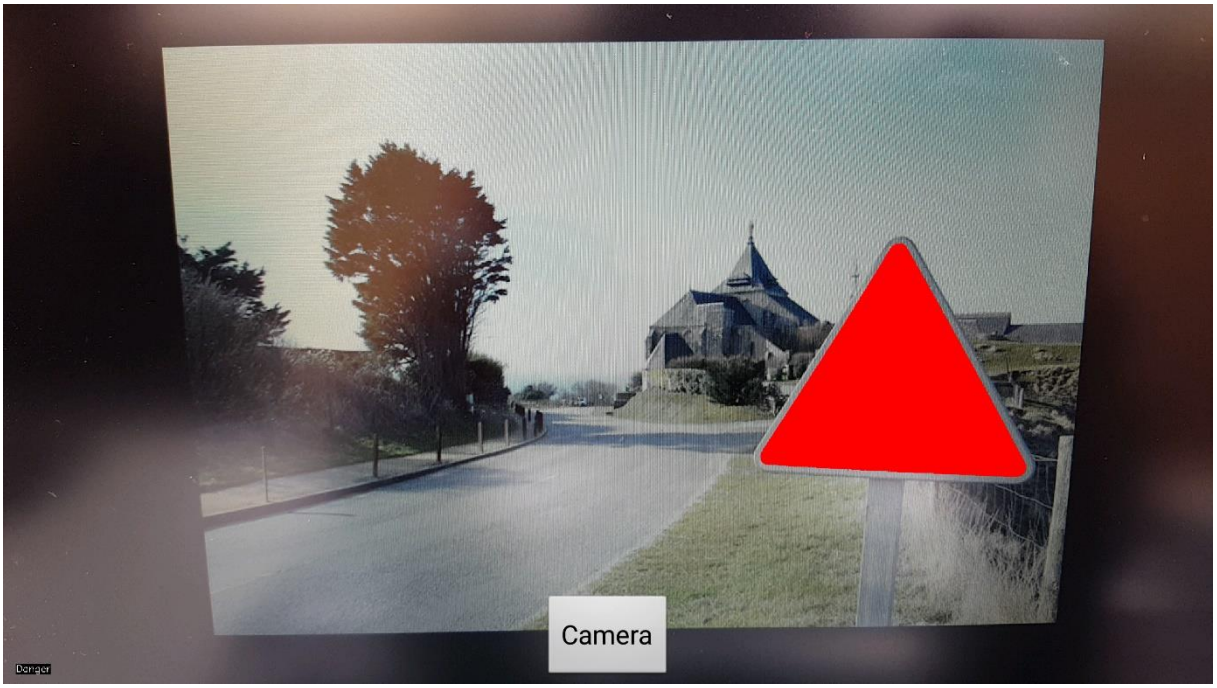





chair: 0.68
car: 0.63
car: 0.69
car: 0.72
car: 0.67
car: 0.77
car: 0.82
car: 0.74
Camera


70
Camera


traffic
traffic light: 0.72
traffic light:
Camera


Camera

3. Améliorations possibles

Comme cité précédemment, notre logiciel a une grande marge d'amélioration.
Nous avons implémenté une détection des panneaux de vitesse mais nous n'avons pas récupéré la valeur de cette vitesse. Or c'est cette information qui devrait être récupéré. Pour cela il faudrait réaliser un nouvel apprentissage sur le réseau de neurones YOLO afin de lui ajouter d'autres classes pouvant être détectées ou utiliser d'autres technique de reconnaissance de chiffres ou de reconnaissance d'images texturées. De la même manière, l'une des perspectives du projet est de distinguer avec précision les autres panneaux de signalisation routière en combinant des approches géométriques et descriptives ou bien en entrainant un CNN.

Par ailleurs, pour identifier une nouvelle classe d'objets, il faut entrainer ce réseau de neurones convolutifs. Cela nécessite beaucoup plus de ressources de calculs qu'un simple ordinateur portable. En effet, l'apprentissage de Yolo requiert 100 heures pour avoir une première identification de paramètres sur une carte GPU GeForce GTX 880M.

Pour la partie test, il est aussi souhaitable d'avoir un PC ou un téléphone puissant pour pourvoir exécuter le programme en temps réel. Toutefois, ce n'est pas une exigence forte du projet car on ne peut pas demander à un utilisateur d'avoir un téléphone performant pour tester l'application, c'est la précision qui importe le plus et le test peut se faire sur des images statiques. A défaut d'exiger un smartphone performant pour tester l'application, il est envisageable de réaliser une application client-serveur pour réduire le temps de calcul et avoir un rendu visuel dans un délai plus court. Cependant, une telle application est tributaire de la connexion réseau et nécessite la gestion de connexions lorsque l'application est déployée sur plusieurs terminaux.

Ayant réalisé un traitement basique de détection de la route expliqué précédemment, une des améliorations majeures aux projets pourrait être d'implémenter une détection du marquage de la route bien plus poussés et complexes. La quantité d'informations récupérée serait donc bien plus importante et détaillée, par exemple nous pourrions obtenir la déviation de la route, la position des autres véhicules, les dangers sur la route etc. Pour ensuite fournir ces informations à un logiciel implémenté dans une voiture autonome.

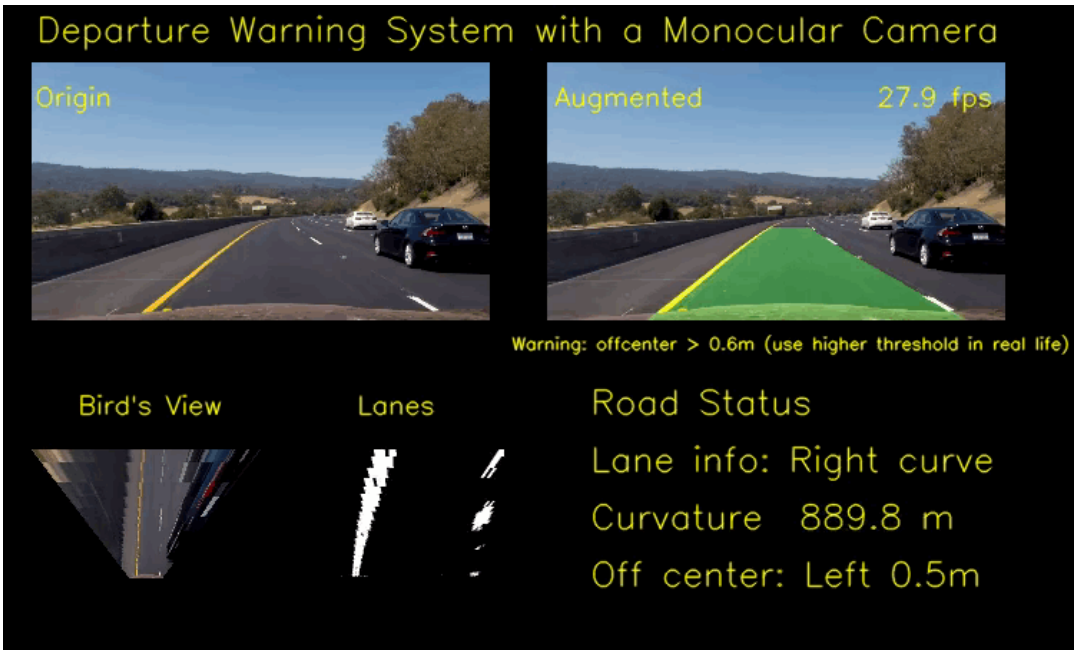


Figure 12 – Exemple de détection avancée de la route

V. Annexes

Vidéos

Building Opencv + Contribute with CMake https://www.youtube.com/watch?v=fIpTks0G2m0
Haarcascade Visualization https://www.youtube.com/watch?v=hPCTwxF0qf4
Canny Edge Detector https://www.youtube.com/watch?v=sRFM5IEqR2w
Finding the Edges (Sobel Operator) https://www.youtube.com/watch?v=uihBwtPIBxM
K-means & Image Segmentation https://www.youtube.com/watch?v=yR7k19YBqiw
What is a Neural Network ? https://www.youtube.com/watch?v=aircAruvnKk
How neural networks learn https://www.youtube.com/watch?v=IHZwWFHWa-w
What is backpropagation really doing ?https://www.youtube.com/watch?v=Ilg3gGewQ5U
Tariq Rashid – Introduction to Neural Network https://www.youtube.com/watch?v=2sevic5Vy4E
Neural Network 3D Simulation https://www.youtube.com/watch?v=3JQ3hYko51Y
Opencv Basics https://www.youtube.com/playlist?list=PLAp0ZhYvW6XbEveYeefGSuLhaPlFML9gP
Real-Time Object Tracking https://www.youtube.com/watch?v=bSeFrPrqZ2A
Haarcascade Object Detection https://www.youtube.com/watch?v=88HdqNDQsEk

Documentation

Histoire du traitement d'images https://interstices.info/jcms/c_5952/histoire-du-traitement-d-images
Base d'images http://www.image-net.org/
Dépôt OpenCV https://github.com/opencv/opencv
Dépôt OpenCV Contribute https://github.com/opencv/opencv_contrib
OpenCV Tutorial C++ https://www.opencv-srf.com/p/introduction.html
Caffe Framework Tutorial https://docs.opencv.org/3.4.0/d5/de7/tutorial_dnn_googlenet.html
YOLO Tutorial https://docs.opencv.org/3.4.0/da/d9d/tutorial_dnn_yolo.html
Darknet YOLO https://pjreddie.com/darknet/yolo/