# RECONNAISSANCE D'ENVIRONNEMENT POUR LA DETECTION DE STATIONS METRO

*Alex Auternaud, Maria Veizaga*

INSA de Lyon, Novembre 2018

## ABSTRACT

L'objectif était de développer un algorithme pour la reconnaissance d'environnement pour la détection de stations de métro. Nous avons implémenté un algorithme de classification d'images pré-entrainé (VGG16), dont les trois dernières couches ont été supprimées et remplacées par quatre couches supplémentaires. Cet algorithme a été entrainé et testé sur une base de données que nous avons construite sur la ligne de métro D à Lyon, composée de 15 stations (soit 28 quais au total pour l'aller-retour).

Ce document présent les bases théoriques sur lesquelles nous nous sommes basés ainsi que la démarche suivi. Les résultats obtenus sont décrits à la fin. Notre solution est simple mais elle s'est montré assez prometteuse : 24/28 quais ont été détectés correctement (soit 85% de taux de réussite).

## 1. INTRODUCTION

L'accès restreint aux systèmes électroniques dû à leur ancienneté empêche la récupération de données sur la position et trajectoire des métros. Pour s'affranchir de ce problème il faudrait implémenter un système indépendant qui puisse détecter en temps-réel grâce à la reconnaissance d'environnement la station où le métro se trouve, de même que déterminer si le métro est à l'arrêt. L'affluence variable de passagers dans les stations, la ressemblance des stations entre elles, ainsi qu'une base de données réduite ont été les principales difficultés lors de ce projet. Nous aborderons donc tout au long de ce document la mise en œuvre réalisée, les résultats obtenus, les difficultés rencontrées, les limites de l'algorithme, et finalement les axes d'amélioration pour la suite du projet.

## 2. CONTEXTE ET SOLUTIONS EXPLOREES

Entre les approches précédentes les plus récents pour la reconnaissance visuelle d'environnements, nous avons trouvé des méthodes qui se focalisaient sur l'extraction de caractéristiques avec des descripteurs à base de CNN [2] ou sur l'identification des régions particulières à travers des datasets avec des conditions variables et de prises de vue différentes [3]. Cependant ces méthodes étant très sophistiquées et leur code n'étant pas disponible pour l'implémenter, nous avons décidé d'explorer des solutions dont le code était facilement accessible et qui semblaient pouvoir répondre à nos besoins. Les deux solutions essayées sont : l'algorithme de reconnaissance d'objets Yolo et un algorithme de classification d'images basé sur le réseau VGG16 sur Keras.

Yolo est une des meilleures solutions disponibles actuellement pour la détection et classifications d'objets en temps réel [1]. Le code d'origine a été développé sur C, ce qui empêchait son utilisation sur Floydhub, qui supporte du code sur Python uniquement. Nous avons ainsi utilisé une version de Yolo non officielle sur Python : *darkflow*. A cette étape nous avons été confrontés aux problèmes liés aux bases de données, étant donné que cet algorithme trouve toute sa performance sur des bases de données assez larges (autour de quelques centaines d'images par classe pour l'entrainement). Par contre nous avions conformé une base de données d'à peine une cinquantaine d'images par station (ou classe). Yolo étant conçu principalement pour la reconnaissance d'objets, nous avons essayé de trouver d'objets dits « caractéristiques » de chaque station, mais ceci a réduit encore plus la taille de notre base de données. Finalement, après avoir réalisé quelques entraînements, les résultats en test ne se sont pas montrés exploitables. Cette solution a été donc laissée de côté.

L'objectif étant la reconnaissance et classification d'une image entière plutôt que des objets sur l'image, nous avons décidé de tester les performances d'un algorithme de classification. La librairie Keras était la plus adaptée dans cette première étape de par sa clarté et facilité d'utilisation, en particulier pour des débutants comme nous. Grâce à un tutoriel trouvé en ligne, nous avons utilisé le réseau VGG16 pré-entraîné comme base, en remplaçant les dernières couches par d'autres adaptées pour la classification des stations de métro.

## 3. CLASSIFICATION D'IMAGES EN BASE AU RESEAU VGG16

Il est évident que pour résoudre notre problème, un réseau de neurones convolutionnel ou CNN est l'architecture la plus appropriée. A différence d'un réseau de neurones classique, ces réseaux dits aussi « ConvNets » ont été conçus pour recevoir en entrée des images, ce qui permet de rendre plus efficace l'implémentation de la fonction « forward » et de réduire considérablement les paramètres du réseau [5].

Pour pouvoir construire un modèle suffisamment puissant il faut un réseau plus ou moins complexe et entraîné sur une base de données de taille conséquente. Le réseau qui a été choisi est le VGG16, pré-entraîné sur la base de données ImageNet.

### 3.1. Modèle VGG16

Le modèle VGGNet développé par Karen Simonyan et Andrew Zisserman a remporté deux prix au Challenge de ImageNet (ILSVRC) en 2014. Leur contribution la plus significative était de démontrer que la profondeur du réseau est un élément critique pour atteindre une bonne performance. Leur meilleur et dernier réseau prend en entrée des images de taille 224x224x3 (RGB) qui ensuite passent par 16 couches CONV/FC [6]. Cette architecture effectue des convolutions avec des filtres d'une taille de seulement 3x3 et des poolings à 2x2 dès le début jusqu'à la fin. La pile de couches convolutionnelles est suivie par trois couches dites « fully-connected », la toute dernière étant une couche de type « soft-max ». Le réseau contient un total de 140M de paramètres [5].

### 3.2. Base de données ImageNet

ImageNet est une base de données qui a été développée pour la recherche. Cette base est conformée par près de 14 de millions d'images et contient une dizaine de milliers de classes. Nous pouvons implémenter cette base de données depuis Keras, mais nous n'avons pas un accès direct aux images d'entraînement.

### 3.3. Utilisation des dernières cartes d'activation d'un modèle pré-entraîné

Le modèle VGG16 entraîné sur ImageNet était un bon point de départ car il aura appris des caractéristiques rélevantes à notre problème de classification. La stratégie proposée par l'exemple du blog [4] que nous a servi de guide tout au long de notre projet, était d'instancier seulement la partie convolutionnelle du modèle VGG16, soit toutes les couches supérieures avant les dernières couches dites « fully-connected ». Ensuite nous avons entraîné ce modèle seulement une fois avec les données d'entraînement et de validation. La sortie étant les dernières cartes d'activation, nous les avons enregistrées comme « bottleneck features » du modèle sur deux vecteurs du type « numpy ». De cette manière nous devions juste entraîner un modèle plus petit conformé par des couches de type « fully-connected » juste en bas des premières couches du modèle complet. Ceci est fait dans le but d'optimiser le temps d'entraînement, car utiliser le modèle en entier à chaque entraînement prendrait beaucoup de temps.

### 3.4. Optimisation fine des dernières couches d'un modèle pré-entraîné

Pour pouvoir améliorer la précision, l'exemple [4] nous proposait de faire une optimisation fine ou « fine-tunning » du dernier block convolutionnel du réseau VGG16 et des dernières couches ou « top layers ». Pour ceci il a fallu d'abord instancier le modèle VGG16 (inclus dans la librairie *applications* de Keras) et charger ses poids. Puis ajouter le modèle conformé par les dernières couches de type « fully-connected » que nous avons entraîné et charger les poids enregistrés précédemment. Finalement avant de faire l'entraînement, il a fallu congeler les couches du modèle VGG16 pour ne modifier que le dernier block convolutionnel lors du prochain entraînement.

Pour pouvoir faire une bonne optimisation il faut avoir les poids enregistrés des entraînements précédents. De même un « fine-tunning » est fait que sur les dernières couches au lieu de sur tout le modèle pour éviter un « overfitting ». Ainsi les premières couches auront appris des caractéristiques plus générales et les dernières couches des caractéristiques plus spécifiques. Le pas d'apprentissage ou « learning-rate » doit rester lent pendant cette étape, pour assurer que les modifications restent faibles et qu'elles n'écrasent pas les caractéristiques apprises dans les couches du haut.

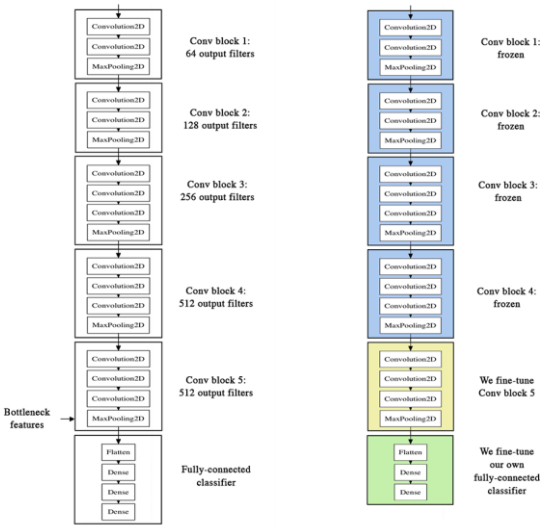


Figure 1: (Gauche) Architecture du réseau VGG16. (Droite) « Fine-tunning » de deux derniers blocks du modèle [4]

## 4. MISE EN OEUVRE REALISEE

La mise en œuvre de notre fonctionnement se découpe en deux parties. Une partie entraînement et une partie prédiction.

### 4.1. Entraînement

Dans le fichier *keras_classier_2.py* se réalise l'entraînement de notre model à partir du modèle pré-entraîné VGG16 de Keras. Dans le fichier *real_time_prediction_and_motion_detect_2.py* nous effectuons une prédiction en temps réel de la station dans laquelle se trouve le métro et nous signalons si le métro est à l'arrêt ou non.

Nous défissions deux fonction : *save_bottlebeck_features()* et *train_top_model()*. Dans *save_bottlebeck_features()*, nous récupérons les poids du model pré-entraîné VGG16 de Keras sans les « top layers » qui sont les trois dernières couches. Ensuite, nous appliquons une « data augmentation » qui génère les batches (de taille 16) de données améliorées à partir de nos données ; c'est-à-dire que nous découpons nos données en lots, les mettons sous le bon format (seulement les images de taille 224*224*3 sont traitées sous VGG16) avec des coefficients entre 0 et 1. Nous aurions pu utiliser d'autres fonctions comme le shuffle, le décalage, le rescale, le zoom, l'horizontal_flip ou le vertical_flip de nos données pour élargir la base de données. A la fin de l'entraînement, nous obtenons une précision de plus de 90% d'où le fait que n'avons pas fait plus de « data augmentation ». De plus, nous avions déjà réalisé un shuffle de nos données au moment de choisir dans chaque classe quelles données vont être utilisées pour entraîner et quelles données pour valider.

Nous passons toutes nos données d'entrainement à travers le modèle pré-entraîné sans les dernières couches ou « top_layers ». Nous obtenons et sauvegardons la sortie *bottleneck_features_train.npy* qui est un vecteur numpy contenant la dernière carte d'activation avant le « top layers ». Ensuite nous extrayons les étiquettes d'entraînement et les sauvons sous un vecteur numpy : *train_label_train.npy*. Ensuite, nous réalisons les deux étapes précédentes de « data augmentation » et de cartes d'activation, mais avec la base de validation. Nous obtenons les array numpy *bottleneck_features_validation.npy* et *validation_label_train.npy*.A la fin de cette fonction, nous créons un fichier *json* qui sauvegarde comment sont reliées chaque nom de station à un numéro.

Dans la seconde fonction, *train_top_model()*, nous récupérons les cartes d'activation *bottleneck_features_train.npy* et *bottleneck_features_validation.npy* et les étiquettes *train_label_train.npy et validation_label_train.npy*. Nous créons un modèle de quatre couches. La première couche, « flatten », charge la dernière carte d'activation d'entraînement ; la seconde, « dense », est une couche de convolution classique « fully connected » où les neurones sont activés ; la troisième, « dropout », est une couche dont seulement 50% des neurones sont activés (afin d'éviter un « overfitting ») ; la quatrième, « dense », est une couche d'activation de type « softmax » pour les modèles multiclasses. Ensuite, nous compilons et entrainons ce modèle en faisant de nouveau un shuffle et avec nombre de 50 epochs.

Par la suite, nous sauvegardons les poids de ce nouveau modèle. Nous imprimons aussi son architecture et son résumé (figure 2). La précision atteinte lors de l'entraînement sur Floydhub est de 0,983 pour le dataset d'entraînement et de 0,983 pour celui de validation.

### 4.2. Prédiction

#### *4.2.1 Détection de mouvement (métro à l'arrêt)*

Premièrement, nous lançons la vidéo et nous sauvegardons les trois premiers frames. Puis nous lisons frame par frame la vidéo. Nous calculons la distance Pythagorean de deux frames séparés par un frame. Ensuite, nous appliquons un lissage Gaussien à cette distance et en calculons la déviation standard. Si cette valeur est inférieure à un minimum, nous estimons que le tram est à l'arrêt. Nous afficherons sur le frame suivant le résultat pour ce frame analysé. Nous avons fixé le seuil à 13 mais nous pouvons le remettre en cause, notamment lorsqu'il y a trop d'individus en mouvement ou qu'un d'entre eux est en train de courir.

#### *4.2.2 Prédiction du quai*

Chaque 20 frames, nous réalisons une prédiction ; c'est-à-dire, nous prenons le frame en cours de lecture, nous le passons à travers le modèle de VGG16 ainsi que des couches rajoutées précédemment. Nous obtenons de nouveau une carte d'activation. Ensuite, nous chargeons le modèle que nous avons entraîné dans la partie précédente et générons une prédiction à partir de cette carte d'activation. Nous affichons en résultat seulement les valeurs supérieures à 0.2, nous recherchons leur index pour remonter au nom de la station de métro grâce au fichier « json » contenant le lien entre un numéro et le nom de la station. Cela signifie que nous sélectionnons seulement les stations qui ont une probabilité supérieure à 0.2 pour le frame analysé. Ensuite nous affichons sur le frame suivant le frame analysé le nom de la station ayant la plus grande probabilité, sa probabilité.

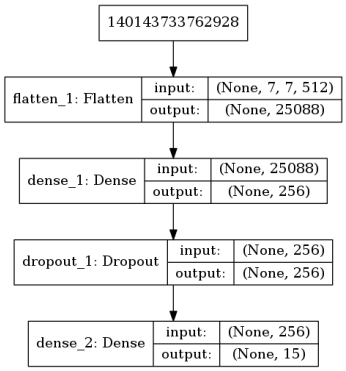


Figure 2 : Dernières couches ou « top layers » rajoutées au modèle VGG16 à la place des 4 dernières couches d'origine.

### 4.3. Base de données

Nous avons effectué nos propres acquisitions pour construire notre base de données. Ces acquisitions ont été prises à trois occasions différentes. La première base de données (correspondant à la première acquisition) nous a servi à faire les premiers tests de notre algorithme. La deuxième base a est composée par des vidéos de meilleure qualité (deuxième acquisition). La troisième et dernière base de données utilisée n'est plus qu'une optimisation de la deuxième base. La troisième acquisition nous a servi à conformer la base de test.

La fonction *video2img.py* prend une vidéo en entrée et sélectionne un nombre de frames défini. Nous avons extrait 50 images par vidéo (100 au total par station), puis nous avons fait le tri pour ne garder que les meilleures pour le dataset.

Pour la base d'entraînement nous avons pris au hasard 85% du total de la base de données, et 15% pour la validation grâce à *shuffle_img_(rotate).py* qui en plus nous remet l'image dans le bon sens pour respecter les notions de perspective appris par la réseau pré-entraîné. Ainsi, le dossier *dataset train* et le dossier *dataset validation* contient chacun 15 dossiers avec des images correspondantes aux 15 stations, les deux sens de circulation confondus.

## 5. RESULTATS

### 5.1. Résultats et critères d'acceptabilité

Dans cette section nous allons évaluer les performances de notre algorithme à reconnaître les stations de métro. Une prédiction est faite sur une image de la vidéo chaque 20 frames. La station avec le pourcentage de ressemblance le plus haut est affiché sur la vidéo ainsi que la déviation standard entre les derniers 3 frames pour la détection de mouvement. Un message est affiché lorsque le métro se trouve à l'arrêt.


Figure 3 : Prédictions faites par l'algorithme sur un vidéo test (Gauche) Sans Souci à 0,903 – Métro en mouvement (Droite) Laennec à 0,999 – Métro à l'arrêt

Nous considérons qu'une station a été correctement identifiée si le nombre de prédictions faites avant l'arrêt du métro en quai est supérieur à 50%. En plus, une fois le métro arrêté, la prédiction doit converger vers une unique solution.

### 5.2. Influence de la base de données sur les résultats

Nous avons effectué au total 3 jeux d'acquisitions différents sur les 15 stations (ou 28 quais au total sur l'aller-retour). Le premier jeu de données nous a servi à faire le Dataset No.1 sur lequel nous avons effectué les premiers entraînements et tests. Le la ligne de métro D n'ayant pas de conducteur, nous avons pu obtenir une vue frontale pour nos acquisitions. Cette base de données comportait une cinquantaine d'images par station, cependant les images n'étant pas d'une bonne qualité (40% de l'image sur les rails), nous avons obtenu des résultats médiocres et pas concluants : 6 quais détectés correctement sur 28. Pour le deuxième jeu de données Dataset No.2 (cinquantaine d'images par station), la prise de vue choisie était frontal-diagonale, ce qui a nettement amélioré les résultats : 20 quais sur 28. Finalement après une optimisation de cette dernière base de données nous n'avons laissé qu'entre une vingtaine et une trentaine d'images par classe, en ne gardant que les plus représentatives de chaque station. Le taux de précision s'étant amélioré de manière globale pour toutes les stations et le résultat final étant de **24 quais détectés correctement sur un total de 28.**

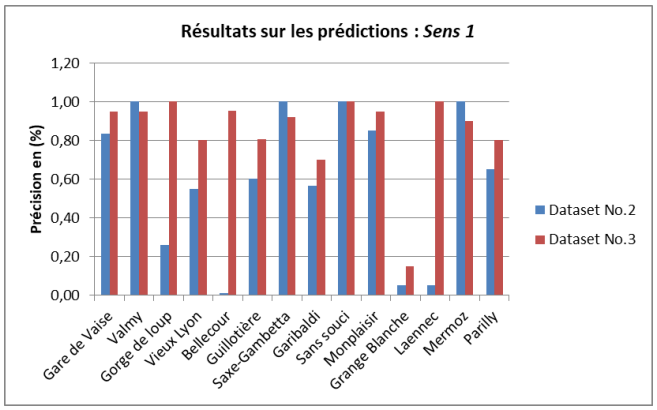


Figure 4: Précision de la prédiction de l'algorithme dans le *Sens 1*.

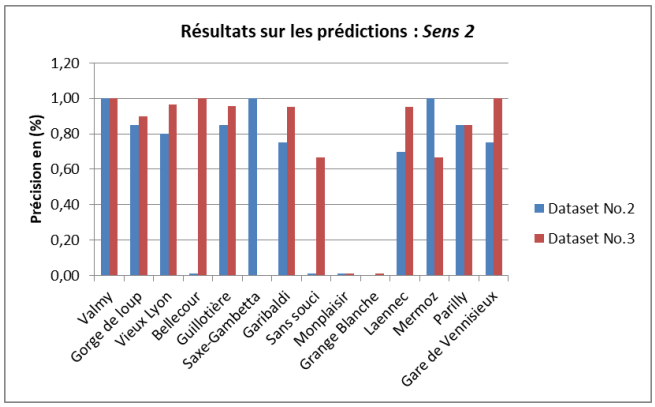


Figure 5 : Précision de la prédiction de l'algorithme dans le *Sens 2*.

### 5.3. Analyse des résultats obtenus

Les résultats obtenus avec la dernière base de données montrent que nous n'avons pas réussi à identifier correctement 4 quais sur 28. Le taux de précision pour ces quais est très mauvais, notamment pour la station Grange Blanche dans les deux sens. Nous croyons que ce problème vient du fait que les images pour l'entraînement de cette station sont de très mauvaise qualité (reflet sur la vitre et mauvaise prise de vue). De plus, les images d'entraînement ont été prises lors d'une affluence importante d'usagers, alors que sur la vidéo de test le quai est quasiment vide. Il faut noter que nous n'avons pas pris en compte à aucun moment la prédiction des quais précédents dans la logique de l'algorithme, de façon à pouvoir évaluer uniquement sa précision sur la reconnaissance d'images.

## 6. CONCLUSION

La solution qui a été développée lors de ce challenge a généré des résultats assez satisfaisants avec **un taux de précision de 85%**. **L'algorithme détecte aussi quand le métro est à l'arrêt, grâce à une valeur de seuil réglable.** Cette première étape du projet nous a permis d'approfondir nos connaissances sur les CNNs et ainsi que de prendre en main les outils nécessaires pour le développement d'algorithmes à base de deep learning. Nous avons aussi été sensibilisés à l'influence de la qualité des bases de données sur les résultats.

Nous avons identifié quelques points à améliorer et développer lors de la prochaine étape de ce projet. Tout d'abord pour augmenter les performances de notre algorithme il faudrait élargir notre base de données, avec des images prises plutôt du haut du métro de manière à s'affranchir le plus possible du problème dû au flux d'usagers. La nouvelle base de données devrait comporter des images des stations vides et puis d'autres avec des personnes. Finalement il faudrait envisager l'implémentation d'un CNN du type LSTM, de manière à rajouter un caractère Spatiotemporel à notre algorithme et ainsi effectuer d'inférences sur des séquences d'images.

## 7. REFERENCES

[1] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, You Only Look Once : Unifed, Real-Time Object Detection. *arXiv preprint arXiv: 1506.02640v5*, 2016.

[2] B. Ivanovic, Visual Place Recognition in Changing Environments with Time-Invariant Image Patch Descriptors. Standford University, 2016

[3] Z. Chen, F. Maffra, I. Sa, M. Chli, Only Look Once, Mining Distinctive Landmarks from ConvNet for Visual Place Recognition. ETH Zurich, Switzerland, 2017.

[4] F. Chollet, Building powerful image classification models using very little data, *The Keras Blog*. 2016

[5] CS231n Convolutional Neural Networks for Visual Recognition, Standford University. 2018

[6] K. Simonyan, A. Zisserman. Very Deep Convolutional Networks for Large Scale Image Recognition. Visual Geometry Group, University of Oxford. *arXiv:1409.1556*, 2015

## 8. ANNEXES

| | Dataset No.2 | | Dataset No.3 | |
|---|---|---|---|---|
| **Station** | **Sens 1** | **Sens 2** | **Sens 1** | **Sens 2** |
| Gare de Vaise | 0,83 | - | 0,95 | - |
| Valmy | 1,00 | 1,00 | 0,95 | 1,00 |
| Gorge de loup | 0,26 | 0,85 | 1,00 | 0,90 |
| Vieux Lyon | 0,55 | 0,80 | 0,80 | 0,97 |
| Bellecour | 0,01 | 0,01 | 0,95 | 1,00 |
| Guillotière | 0,60 | 0,85 | 0,80 | 0,95 |
| Saxe-Gambetta | 1,00 | 1,00 | 0,92 | - |
| Garibaldi | 0,57 | 0,75 | 0,70 | 0,95 |
| Sans souci | 1,00 | 0,01 | 1,00 | 0,67 |
| Monplaisir | 0,85 | 0,01 | 0,95 | 0,01 |
| Grange Blanche | 0,05 | - | 0,15 | 0,01 |
| Laennec | 0,05 | 0,70 | 1,00 | 0,95 |
| Mermoz | 1,00 | 1,00 | 0,90 | 0,67 |
| Parilly | 0,65 | 0,85 | 0,80 | 0,85 |
| Gare de Vennisieux | - | 0,75 | - | 1,00 |

Figure 5: Résultats de deux datasets testés, dans les deux sens de circulation


Figure 6: “Training metrics” sur Floydhub - Dataset No.3