Classification et Détection de logo de marques

Baudouin Fauchier-Magnan, Jessica Cohen, Alexia Hugon

DEEP LEARNING - CentraleSupelec

Abstract

Nous nous proposons ici d'étudier la reconnaissance, la classification et la détection de logos de marques. À l'aide de Keras et Tensorflow, nous construirons un modèle de reconnaissance d'image basé sur ImageNet, puis l'adapterons à la reconnaissance et la classification de logos en partant du dataset FlickLogos-47. Finalement, nous adapterons un système de détection (YOLO) à notre dataset afin d'encadrer nos logos reconnus dans des *bounding boxes*.

1. Introduction

La reconnaissance de logo dans les images - et plus encore dans les vidéos - est devenue une problématique clée des marques. Sensibles à leur présence en ligne, elles doivent développer des outils de traçabilité pour une multitude d'applications diverses : la violation de droits d'auteur, le tracking des fake news ou de leur présence sur les réseaux sociaux, etc.

La détection du logo dans l'image et son encadrement dans des "bounding box" prend également une importance majeure dans la publicité. Grace à l'utilisation de réseaux convolutifs adaptés, on peut imaginer à moyenterme une évolution vers des budgets publicitaires plus optimisés. Prenons la diffusion d'un match de football : combien de temps le logo est-il réellement passé à l'écran (*reconnaissance*) ? Quelle portion de l'écran a-t-il occupé (*detection et encadrement en bouding box*) ? On pourrait en déduire une adaptation optimale des dépenses publicitaires colossales des marques. Nous avons donc souhaité faire un état de l'art des méthodes que l'on pourrait mettre à profit pour répondre à cette problématique.

Dans cette optique et à l'aide d'un dataset de logos labélisé, nous nous sommes d'abord intéressé aux méthodes de reconnaissance de logo existantes ; avant de tenter d'adapter un système de détection (YOLO) à notre dataset.

2. Notre dataset

2.1. FlickrLogos-47

Le dataset FlickrLogos-47 est une collection de photos de logos libre d'accès. Elle est composée de 47 marques différentes. Les images font apparaître les logos de manière plus ou moins décentrée et lisible dans les photos. Il constitue une évolution du dataset FlickrLogos-32 (32 classes) largement utilisé dans d'autres projets de recherche notamment celui de S.Bianco et al. ou celui de JH.Chiam

Ce dataset se prête bien à de la reconnaissance d'image, et, bien qu'annoté correctement, il présente la limite du nombre restraint d'images par classe. Ainsi, la figure 1 répertorie le découpage des données en terme d'apprentissage, validation et test.

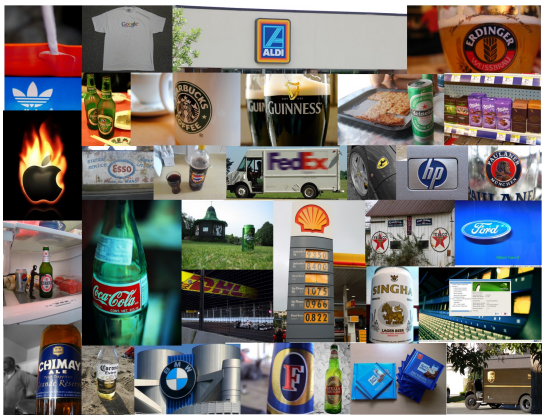

Figure 1: Echantillon des images du dataset étudié

	Nombre d'images
Données d'apprentissage	978
Données de validation	213
Données de test	1382

Table 1: Décompte et Répartition des données du dataset

2.2. Améliorer le dataset

Afin de prendre en main le dataset et d'obtenir l'architecture de données souhaitée, le parseur *parser.py* a été réalisé. Un deuxième parseur *splitter.py* a permis de créer les dossiers de validation à partir du dossier train initial.

Au-delà de l'architecture peu adaptée des données, la taille du dataset "en l'état" n'est pas nécessairement la plus adaptée à une approche Deep Learning. Au vu du nombre limité d'images par classe, il est très vite apparu

pertinent d'avoir recours à de l'augmentation de données afin de gonfler le nombre d'images sur lequel entraîner le réseau.

Au cours de l'expérimentation, nous avons ainsi voulu tester l'intérêt de différentes méthodes d'entrainement afin de les commenter :

- avant et après augmentation de données
- différentes variation des tailles des données
- variation du nombre d'epochs

Nous aurions également pu tester d'autres améliorations du dataset :

- suppression des duplicats pour agrandir la variabilité du dataset
- faire grossir le dataset avec des images tirées de Google pour avoir plus d'exemples contextuels (pour une marque de bière : une bouteille, une canette, un panneau, un bandeau publicitaire, un tee-shirt floqué...)

3. Reconnaissance et classification : méthode proposée

3.1. Travailler avec des réseaux existants et ImageNet

Nous avons rapidement constaté qu'un apprentissage $from\ scratch$, avec aussi peu de données et des contraintes au niveau de la capacité de calcul était peu envisageable. Afin d'obtenir des performances décentes, nous avons fait le choix de réutiliser des réseaux existants et déjà entraînés sur de larges datasets - en l'occurrence celui d'ImageNet.

ImageNet est un ensemble d'images (1,2 millions d'images et mille classes) déjà annotées et utilisables librement. Des modèles de réseaux de convolution comme ResNet50 (developpé par Microsoft Research) et InceptionV3 (developpé par Google Research) sont entrainés sur cette base d'image et sont facilement réutilisables. La librairie Keras est une librairie de réseaux neuronaux (couche d'abstraction audessus de Tensorflow) qui nous permet de faire appelle à ces modèles grace au module $keras.applications$.

À l'aide donc de Keras, Tensorflow et le modèle InceptionV3 (sur ILSVRC) directement accessible à travers Keras, nous avons commencé par construire un court programme python avec différents arguments possibles :

- $ python $model.py$ –image $nameofimage$.jpg
- $ python $model.py$ –imageurl

Une fois le modèle appris, il est possible de lancer le programme python avec n'importe quelle image (parmi les catégories d'ImageNet) et d'obtenir un graphique en baton avec les résultats de prédiction obtenus. On obtient un Top 3 des prédictions de catégorie obtenue ainsi que leurs probabilités.

3.2. Utiliser le transfert d'apprentissage

Une fois les réseaux entraînés sélectionnés, il est nécessaire de réfléchir à quel portion du réseau réutiliser et quelle partie ré-entraîner. En l'occurrence, il est important de trancher entre des méthodes de transfert d'apprentissage ou de réglage fin du réseau.

Par manque de temps, nous avons choisi de concentrer nos efforts sur le transfert d'apprentissage en utilisant l'article de G.Chu.

Les réseaux convolutifs nécessitent des ressources et des quantités de données importantes pour s'entrainer, c'est donc intéressant de passer par des méthodes de $transfer\ learning$ pour transférer les poids du réseau appris lors d'une tâche antérieure (comme ImageNet) vers une nouvelle tâche.

Le $transfer\ learning$ consiste à reprendre notre réseau de convolution pré-entrainé sur ImageNet, retirer la dernière couche $fully - connected$, puis traiter le reste du réseau comme un extracteur de $features$ pour notre nouveau dataset - en l'occurent FlickrLogo47. Une fois que l'on a extrait les $features$ pour toutes les images, on peut entrainer un classifieur pour le nouveau dataset.

Pourquoi avons-nous préférer le $transfer\ learning$ au $fine-tuning$? Deux raisons principales ont motivé notre décision : la taille du dataset et les similitudes entre les deux datasets. En effet, avec un dataset $petit$ et des datasets $similaires$, on fait le choix du $fine\ tuning$ pour éviter l'$overfitting$. Le réglage fin du réseau ne nous aurait pas permi d'atteindre de bons niveaux de généralisation.

Avec le $transfer\ learning$ on freeze l'avant dernière couche et re-entraine la dernière.

Le lancement de l'entraînement via la commande $python\ transfer\_learning.py\ -train\_dir(trainpath)\ -val\_dir(valpath)$ permet de générer un fichier .model très lourd (plus de 100 000 Ko) qui contient les poids enregistrés. Ce fichier .model est ensuite utilisé avec le script test.py afin de prédire la classe d'une image de l'ensemble de test.

3.3. Architecture du réseau

En utilisant le réseau Inception v3, on travaille avec une architecture de réseau relativement complexe, comme on peut le voir dans la figure 2. Le réseau comporte une succession de couches de convolution, d'étapes de pooling, dropout, et finit par une couche softmax.

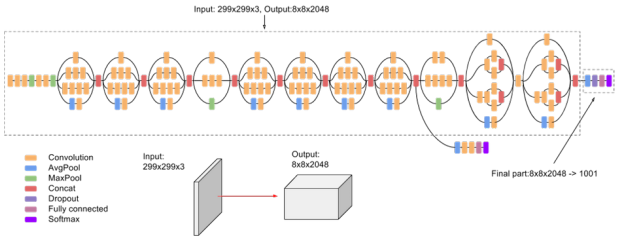

FIGURE 2: Architecture du réseau Inception v3

4. Expérimentation et résultats

Dans la figure 2 nous résumons les différents paramétrages testés et nous reprenons les résultats dans les figures 3, 4, 3 et 6.

	Temps/epoch	Temps/étape	Perte	Précision	Perte (validation)	Précision (validation)
Epoch 1	6765 sec	226 sec	4,5444	0,0453	15,6155	0,0216
Epoch 2	6778 sec	226 sec	3,3846	0,1292	14,7601	0,0315
Epoch 3	7001 sec	233 sec	2,8668	0,2222	16,1126	0

Figure 3: Résultats sur les données d'apprentissage et de validation du paramétrage A

	Temps/epoch	Temps/étape	Perte	Précision	Perte (validation)	Précision (validation)
Epoch 1	6543 sec	218 sec	4,2616	0,0474	15,0165	0,0197
Epoch 2	6483 sec	216 sec	3,2727	0,1540	15,2318	0,0174
Epoch 3	6502 sec	217 sec	2,8118	0,2418	15,6624	0,0282
Epoch 4	6508 sec	217 sec	2,4880	0,3033	16,0822	0
Epoch 5	7251 sec	242 sec	2,2812	0,3301	15,4577	0,023

Figure 4: Résultats sur les données d'apprentissage et de validation du paramétrage B

	Temps/epoch	Temps/étape	Perte	Précision	Perte (validation)	Précision (validation)
Epoch 1	6422 sec	214 sec	4,4111	0,0602	14,4712	0,0234
Epoch 2	6359 sec	226 sec	3,0692	0,2079	15,5024	0,0188
Epoch 3	6368 sec	212 sec	2,4272	0,3200	15,9434	0,0094

Figure 5: Résultats sur les données d'apprentissage et de validation du paramétrage C

	Temps/epoch	Temps/étape	Perte	Précision	Perte (validation)	Précision (validation)
Epoch 1	32027 sec	1068 sec	3,8997	0,0177	14,7155	0,0281
Epoch 2	13485 sec	449 sec	3,8480	0,0198	14,6960	0,0236
Epoch 3	6592 sec	220 sec	3,7944	0,0404	14,6358	0,0284

Figure 6: Résultats sur les données d'apprentissage et de validation du paramétrage D

	A	B	C	D
Nombre d'epochs	3	5	3	3
Taille de batch	32	32	32	32
Taille de FC	1024	1024	1024	1024
Transfert d'apprentissage	Oui	Oui	Oui	Non
Réglage fin de réseau	Non	Non	Non	Oui
Augmentation de données	Oui	Oui	Non	Oui

TABLE 2: Description des différents paramétrages

Nous constatons sans surprise que les meilleurs résultats sont obtenus pour le paramétrage B correspondant au nombre maximisé d'époch avec augmentation de données et utilisant le transfert d'apprentissage. Nous obtenons ainsi une précision de près de 33%. Bien que cette précision ne soit pas totalement satisfaisante, nous pouvons toujours comparer ce chiffre avec l'aléatoire : sachant qu'on a 47 classes, une méthode aléatoire nous donnerait une prédiction avec une précision de 2,13%. Afin d'augmenter cette précision, il s'agirait par exemple d'augmenter le nombre d'images par classe et de rédéfinir les classes. En effet, pour l'instant, pour certaines marques, le dataset fait la distinction entre la classe logo image et la classe logo texte. Certaines images d'entraînement appartiennent donc à plusieurs classes, ce qui peut expliquer nos performances mitigées.

De plus, nous remarquons que la technique de $fine-tuning$ prend beaucoup plus de temps et offre des performances particulièrement médiocres, ce qui confirme notre hypothèse de départ sur la pertinence du $transferlearning$ sur le $fine-tuning$.

Par ailleurs, nous remarquons également que l'évolution de la perte sur l'ensemble de validation n'est pas monotone suivant les différents paramétrages, et, de manière générale, les performances sur l'ensemble de validation sont sont, de manière notable, moins bonnes que sur l'ensemble d'apprentissage.

5. Détection : Approche à l'aide de YOLO

5.1. Introduction

En plus du travail réalisé avec le réseau déjà entraîné sur ImageNet, nous avons choisi une deuxième approche à l'aide de YOLO (You Only Look Once) créé par JC.Redmon et expliqué par M.Chablani et D.Forester les résultats de ce réseau sont très bons et permettent de s'affranchir du problème des logos multiples sur une image (entre autres). Il s'agit d'un réseau convolutionnel de type R-CNN développé par Joseph Redmon composé de 24 couches convolutionnelles et 2 couches entièrement connectées. Les avantages de ce réseau sont les suivants :

- Rapidité (45 images par seconde)
- le réseau reconnaît les objets de façon généralisée (dessin, photos, peinture...)
- open source

YOLO est ainsi capable de traiter des vidéos en live et d'y repérer les différents objets sur lesquels il a été entraîné. Il va encadrer les objets repérés dans une bounding box et les nommer. Il est même possible d'utiliser YOLO à partir de sa webcam si on a une bonne carte graphique. L'application dans notre cas pourrait être de reconnaître les logos présents dans une vidéo, pour de la répartition de droits marketing, compter le nombre de voitures de telle ou telle marque qui circulent sur une portion de route...

FIGURE 7: Exemple de logo utilisé dans le dataset d'entraînement de YOLO

5.2. Rapide théorie

L'input de YOLO est une image (fichier JPEG, png, ou bien extrait d'une vidéo ie d'un fichier mp4). Le système divise alors l'image en une grille de taille S X S. Si le centre d'un objet est dans une cellule de la grille précédente, alors cette cellule est responsable de la détection de cet objet.

Chaque cellule prédit B bounding boxes ainsi qu'un score de confiance pour chacune de ces boxes. Ce score permet d'évaluer la confiance qu'a YOLO dans le fait que la cellule contienne bien le centre d'un objet. La figure 4 détaille le modèle de détection des bounding boxes d'abord, puis des objets qu'elles contiennent.

YOLO est un réseau convolutionnel à 24+2 couches dont le schéma est présenté sur la figure 5.

Une fois que le réseau est entraîné, on n'a plus qu'à le lancer sur une vidéo ou une image, et celui-ci fait de la détection en live.

5.3. Procédé

Après avoir téléchargé YOLO, nous avons dû faire des modifications dans certains documents pour pouvoir l'entraîner sur nos classes (nombre de classes = 47, path vers les données d'entraînement, de validation et de test...). Nous avons également dû préparer nos données, qui devaient se présenter sous la forme d'un fichier .txt contenant la liste des paths vers les images sur lesquelles il allait s'entraîner, et d'un fichier .txt pour chaque image décrivant les classes qu'elle contient, sous la forme :

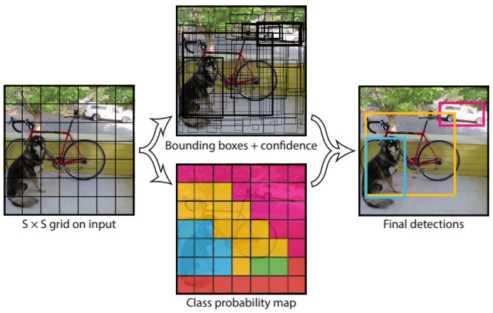

Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.

FIGURE 8: Schéma représentant le modèle de YOLO

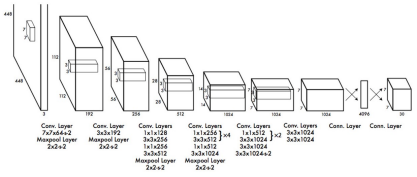

Figure 3: The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

FIGURE 9: Architecture du réseau YOLO

classe x_m y_m *largeur hauteur*

où x_m, y_m, *largeur* et *hauteur* sont relatifs aux dimensions de l'image. xmoy et ymoy représentent respectivement l'abscisse moyenne et l'ordonnée moyenne de la bounding box contenant l'objet. Pour obtenir ces fichiers, il a fallu réaliser un autre script python nommé $parser2.py$. Il a également fallu réduire la taille des images (plus d'un Mo par photo), ce qui a été inclus dans le script. Les poids pré-entraînés ont ensuite été téléchargés (source : ImageNet $darknet19\_448.conv.23$), et nous avons pu entraîner le réseau sur nos données.

Après avoir tenté d'entraîner nos poids sur les 47 classes, nous nous sommes rendus compte que nos jeux de données ne permettraient pas d'avoir de bonnes performances, à moins de réaliser une augmentation de données considérable, ce qui demanderait alors une puissance de calcul que nous n'avions pas. Nous nous sommes donc restreints à 2 classes (logos BMW et Ferrari). Afin de réaliser une augmentation de données rapide et efficace, nous avons également utilisé les données de test fournies dans la base FlickrLogos_47, ce qui a permis d'obtenir une liste de 121 images dans les données d'entraînement et de 32 images dans les données de validation (répartition d'environ 80-20). Les tests seraient ensuite effectués à la main sur des images téléchargées sur internet, ou éventuellement sur un set d'images annotées à la main, selon le temps qu'il nous resterait.

5.4. Entraînement

Nous avons lancé l'entraînement avec les paramètres suivants :

$batch = 64$

$subdivisions = 8$

Ceci signifie que 64 images seront utilisées pour chaque étape d'entraînement, et que ces batchs seront divisés par 8.

Nous avons lancé l'entraînement sur un de nos ordinateurs (CPU et non pas GPU), ce qui rend la tâche extrêmement laborieuse. En effet, pour obtenir un modèle performant, il faudrait réaliser environ 1000 itérations. Or, chaque itération prend sur notre ordinateur environ 870 secondes, ie un peu moins d'un quart d'heure. Il aurait donc fallu attendre 250 heures environ, soit plus de 10 jours, pour que le modèle soit performant. Sur GPU, ces calculs auraient pris moins d'une heure... Nous avons naturellement essayé de passer par des serveurs dédiés type Azure ou Floydhub, mais nous avons alors eu des problèmes de permission : l'exécutable darknet n'est pas utilisable depuis un notebook jupyter par exemple. Nous avons donc mis à disposition dans notre Github un dossier YOLOdetection prêt à être exécuté en ligne de commande. Il n'est pas entraîné encore et les poids pré-entraînés d'ImageNet sont à télécharger (ils pèsent près de 80Mo), mais il n'y a qu'une commande à lancer.

```
Recall: 0.000000,  count: 8
10: 275.707031, 276.233124 avg, 0.000000 rate, 834.070934 seconds, 640 images
Resizing
448
Loaded: 0.305388 seconds
Region Avg IOU: 0.159649, Class: 0.542353, Obj: 0.247555, No Obj: 0.434558, Avg
Recall: 0.125000,  count: 8
```

FIGURE 10: Capture d'écran après 10 itérations

Sur la figure 6, nous pouvons voir que le loss est d'environ 275 après 10 itérations, le loss moyen est de 276.23, le learning rate actuel est nul, l'itération a pris 834 secondes, et 640 images ont été traitées depuis le début de l'entraînement.

6. Expérimentation et résultats

6.1. Performances sur la base d'apprentissage et de validation

La première étape a été d'entraîner le réseau en procédant au transfert d'apprentissage décrit au-dessus. Comparer différents réseaux existants et paramétrages différents performances

L'ensemble de validation permet de réaliser de la cross validation avec les hyper paramètres.

6.2. Performances sur les résultats de test

Une fois le réseau entraîné et les paramètres fixés (et notammenent les poids du réseau), il s'agit de réaliser les prédictions sur les images du dossier test afin de vérifier à quel point le modèle est bien correct.

7. Conclusion

Nous l'avons vu, il est possible d'adapter un réseau appris sur une base d'image très large à la reconnaissance et la classification de logos de marque. Cette adaptation a des *usecases* intéressants dans l'univers de la propriété intellectuelle et de la veille des marques sur les réseaux sociaux ou durant leurs campagnes de publicité. Cela devient d'autant plus intéressant lorsque l'on applique les résultats impressionnants d'un système de détection comme $YOLO$ à la même base d'image de logos de marque. Le repérage en *bounding boxes* des dits-logos permettrait un suivi fin de l'exposition d'un logo lors d'un évènement sportif ou même de reportages et émissions quelconques.

Cela étant dit, plusieurs questions auraient pu être posées pour aller plus loin.

Tout d'abord, le choix d'InceptionV3 a été arbitraire et il aurait été intéressant de comparer les performances des deux modèles (*vs* ResNet50). Ensuite, et même si cette fois notre choix a été motivé, il aurait été judicieux de comparer ou même d'ajouter le $fine\text{-}tuning$ au $transfer\ learning$.

Enfin, si la classification permet de répondre à la question : « à quel logo l'image ressemble-t-elle le plus ? » (sous entendu, quelles caractéristiques de logo retrouvons-nous le plus sur l'image), elle ne permet pas de :

- confirmer la présence d'un logo ou non dans la photo
- faire des prédictions sur des photos présentant plusieurs logos
- définir où est situé le logo dans l'image

Ainsi, si l'on voulait dépasser le cas des images comportant un et un seul logo, il devenait pertinent de se pencher sur des techniques de détection comme $YOLO$. Mais l'on aurait pu également ajouter l'étape d'interrogation : « y a-t-il un logo dans cette image ? » avant de déterminer lequel.

Références

D.Forester, 2018. Guide to using yolov2.
URL `https://github.com/drforester/Guide_to_using_YOLOv2#create-the-traintxt-and-testtxt-files`

G.Chu, 2017. How to use transfer learning and fine-tuning in keras and tensorflow to build an image recognition system and classify (almost) any object.
URL `https://deeplearningsandbox.com/how-to-use-transfer-learning-and-fine-tuning-in-keras-and-tensorflow-to-build-an-image-recognition-94b0b02444f2`

JC.Redmon, 2016. Yolo real time object detection.
URL `https://pjreddie.com/darknet/yolo/`

JH.Chiam, 2015. Brand logo recognition.
URL `http://cs231n.stanford.edu/reports/2015/pdfs/jiahan_final_report.pdf`

M.Chablani, 2017. Yolo : You only look once, real time object detection explained.
URL `https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-detection-explained-492dc9230006`

S.Bianco, M.Buzelli, D.Mazzini, R.Schettini, May 2017. Deep learning for logo recognition.
URL `https://arxiv.org/pdf/1701.02620.pdf`

8. Github

Le lien du Github est disponible ici `https://github.com/alexhug/logo_recognition`.