



CentraleSupélec

Projet Navee

Moteur Inversé de Recherche

Bruno MUHLMANN HOLANDA

Hippolyte SAULNIER

Lucie CLEMOT

Matheus KLEDEGLAU JAHCCHAN ALVES

Tony WU

Table des matières

1 Résumé	3
2 Introduction	3
3 Travaux Similaires	3
4 Concepts Théoriques Nécessaires	3
4.1 Convolutional Neural Networks	3
4.2 Hyperparameters Tuning	4
4.3 Transfer Learning	4
4.4 Image Embedding	5
4.5 K-Nearest Neighbours	6
4.6 LSHashing	7
4.7 Learning to Hash	8
4.8 Triplet Loss	8
4.9 YOLO	10
5 Base de Données	11
5.1 Présentation	11
5.2 Prise en main	12
5.3 Utilisation et Prétraitement	13
5.4 Data Generator	13
5.5 Data Generator pour l'entraînement	13
5.6 Images corrompues ou invalides	13
6 Travail réalisé	13
6.1 Embedding	14
6.1.1 Training	14
6.2 LSHashing	14
7 Résultats	14
8 Travaux Futurs	16
9 Conclusion	17
Bibliographie	18

1 Résumé

Nous proposons un moteur de recherche capable de rechercher des images similaires dans une base de données. Ce moteur fait appel à des concepts récents et largement utilisés, afin de pouvoir prendre en compte une large base de données d'images complexes : les œuvres d'art. Enfin, la finalité du projet ne reste pas cloisonnée à cette problématique puisque le but de Navee est la détection de fraudes sur Internet.

2 Introduction

Dans le cadre de notre projet de première année DataScience, il nous a été demandé de programmer un moteur de recherche inversé d'images. Notre client, l'entreprise Navee, travaille en effet dans la recherche de fraudes à partir d'images dupliquées. Un moteur de recherche tel que celui qu'on nous a demandé permettrait à l'entreprise de lutter contre la fraude d'images similaires : si quelqu'un modifie légèrement l'image volée, notre programme devrait le reconnaître. En particulier, le programme devait être efficace sur une base de données spécialisée en d'œuvres d'art, particulières à cause de leur potentielle abstraction, et de leurs différents modes de réalisation (différents types de peinture, photographie, dessin...). Le client nous a demandé trois versions différentes, de difficulté croissante. Différentes thèses nous ont été fournies pour nous apporter des connaissances et affiner le programme, en augmentant son efficacité et notre connaissance des techniques de machine learning.

3 Travaux Similaires

Recherche d'image par le contenu. Dans (11) les auteurs réalisent une recherche dans la littérature sur les différents approches proposées. Il est important de noter que juste récemment l'utilisation du "deep-learning" dans ce domaine a augmenté sa pertinence.

Recherche d'image profonde. Dans (12), les auteurs proposent un méthodologie générale pour utiliser les réseaux profonds avec pour finalité une recherche d'image se basant sur le contenu. Un modèle présenté dans cet article a une approche très similaire de celui traité en (7).

La différence essentielle entre ces travaux et le nôtre est que notre modèle est formé spécifiquement pour les œuvres d'art et il est formé en pensant qu'une image peut appartenir à différentes classes.

Utilisation de LSH Comme est expliqué dans la section 4.6 l'utilisation des algorithmes de hachage est pertinent pour améliorer le temps de réponse du modèle (la vitesse qui le modèle envoie une image étant donné une requête). Dans (13) et (10) les auteurs abordent l'algorithme de LSH, le plus utiliser pour cette finalité

Autres Méthodes de Hachage En plus de LSH, autres méthodes de hachage sont explorés dans la littérature. Comme l'utilisation d'une fonction de hachage engendre généralement une perte d'exactitude il est nécessaire l'implantation d'autres types de hachage pour améliorer la performance du modèle. Ainsi, par exemple dans (14) et (9) les auteurs proposent une méthode de "learning to hash", aussi expliquée dans la section 4.7 pour établir une fonction de hachage. Le premier utilise une approche pour cela de "Deep Supervised Hashing" et le deuxième utilise autoencoders.

4 Concepts Théoriques Necessaires

Nous introduisons d'abord les concepts théoriques nécessaires à la compréhension du projet. Bien que longue, cette partie est la base de celui-ci.

4.1 Convolutional Neural Networks

Dès le début du projet, nous nous sommes rapidement orientés vers le choix d'un réseau de neurones. Cependant, il restait encore à décider de son architecture. Etant donné que les données à traiter sont des images, nous avons opté pour un Convolutional Neural Network (CNN) qui se caractérise par la présence de couches de convolution. Ce sont ces opérations de convolution qui permettent de détecter localement au niveau des premières couches certaines caractéristiques de l'image telles que des bords, des formes, etc...

Le rôle des couches supérieures est alors de se servir de ces résultats pour renvoyer en sortie un vecteur dit d'embeddings qui caractérisera au mieux l'image initiale.

En outre, les couches de convolution sont souvent suivies de *pooling layers* dont le rôle est de ne garder que les features dominants, souvent par l'application d'un *max pooling* ou d'un *average pooling*.

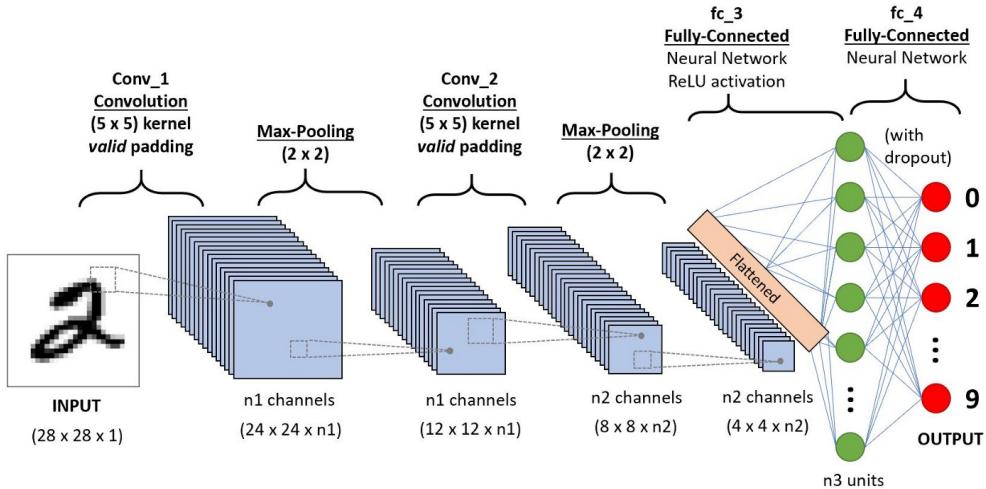


FIGURE 1: Exemple de la structure d'un CNN

4.2 Hyperparameters Tuning

Un aspect non négligeable de l'entraînement par réseau de neurones est le choix des hyperparamètres. En effet, la performance de ce premier en dépend grandement. En pratique, on choisit le jeu optimal d'hyperparamètres de manière empirique, c'est-à-dire en prenant le modèle qui a donné les meilleures performances.

Le problème est alors de trouver un moyen simple et pratique de lancer ces tests et de regrouper les résultats de manière claire. Nous avons opté ici pour l'utilisation de HParams (Tensorflow) qui nous a permis de regrouper les résultats dans un tableau Tensorboard comme celui qui suit.

Session Group Name.	Show Metrics	num_units	dropout	optimizer	Accuracy
3df0d7cf35bec5a...	<input type="checkbox"/>	32.000	0.20000	sgd	0.77550
3ec2aed9e07589f...	<input type="checkbox"/>	32.000	0.20000	adam	0.82650
53bf5bece9190fa...	<input type="checkbox"/>	16.000	0.20000	adam	0.81540
5b97f3c2967245b...	<input type="checkbox"/>	16.000	0.10000	adam	0.83210
6826c7fa3322d82...	<input type="checkbox"/>	32.000	0.10000	adam	0.83950
7684dcc13358fd0...	<input type="checkbox"/>	16.000	0.20000	sgd	0.76830
7b29a731e3daca...	<input type="checkbox"/>	32.000	0.10000	sgd	0.78530
ae235909ec4e4d...	<input type="checkbox"/>	16.000	0.10000	sgd	0.77700

FIGURE 2: Tableau synthétisant les résultats du hyperparameter tuning.

4.3 Transfer Learning

C'est une méthode dans laquelle un modèle utilisé pour effectuer une certaine tâche sert de point de départ à un second modèle pour effectuer une tâche similaire (5). De manière intuitive, l'idée est que quelqu'un qui veut apprendre le piano apprend plus vite s'il sait déjà jouer d'un autre instrument comme la guitare. Cette technique est largement utilisée, surtout dans le deep learning car elle permet de ne pas recommencer un travail depuis le début mais de s'appuyer pour les premières phases d'entraînement sur un réseau dont les performances dans le domaine d'étude sont reconnues.

Pour être plus précis, il est d'usage de "figer" (*freeze* en anglais) les premières couches du réseau construit par Transfer Learning, c'est-à-dire de bloquer les poids associés à ses neurones. Ainsi, lors de

l'entraînement du nouveau réseau, seuls les poids des couches plus en amont varieront. En effet, dans le cas de la reconnaissance d'image, il est prouvé (cf (6)) que les couches de niveaux les plus bas sont plus à même de s'activer lorsqu'ils reconnaissent des formes basiques telles que des segments, des ellipses, etc... tandis que les couches les plus hautes utilisent les informations recueillies par les couches précédentes pour reconnaître des patterns plus complexes.

Dans le contexte de la reconnaissance d'image, nous utiliserons ImageNet.

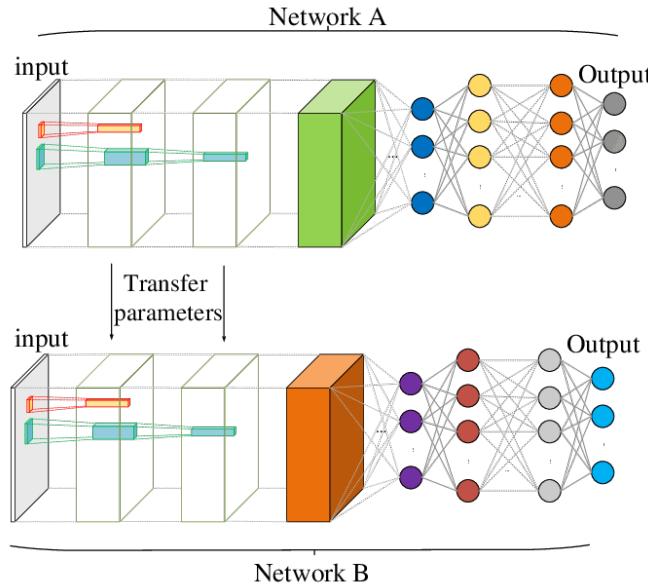


FIGURE 3: Illustration du processus du "transfer learning", où le modèle A préformé, transfère ses deux premières couches au modèle B afin qu'il puisse apprendre plus rapidement et plus efficacement à effectuer une tâche similaire.

4.4 Image Embedding

Embedding en Machine Learning est la création d'un espace vectoriel de dimension réduite à partir d'éléments de plus grande dimension dans lequel les éléments qui ont des caractéristiques similaires sont représentés avec une distance moindre entre eux que les éléments ayant des caractéristiques plus différentes.

Cela consiste alors à représenter une image, représentée par $N \times M$ pixels, par un vecteur plus petit dont la distance pour les images qui sont similaires est plus petite en comparaison aux images qui ne sont pas considérées comme similaires. Plus intuitivement, il s'agirait de créer un espace vectoriel avec les images dans lequel, le vecteur représentant une image représentant une première personne est plus proche du vecteur représentant une image avec une deuxième personne, que d'une image avec un chien.

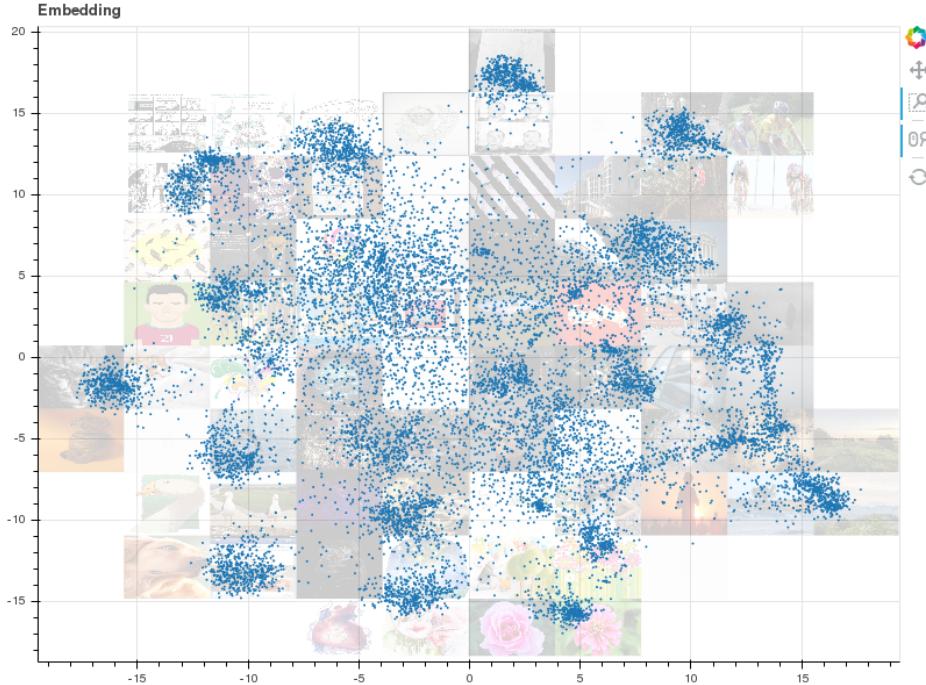


FIGURE 4: Représentation de l'embedding. *Source : <https://bam-dataset.org>*

Pour les images, le vecteur d'embedding peut être créé en utilisant un modèle de classification avec des réseaux neuronaux convolutifs (CNN), les classes de classification étant les images qui seront naturellement les plus proches et le vecteur utilisé serait alors le résultat de l'avant-dernière couche.

4.5 K-Nearest Neighbours

Pour trouver les images similaires, après les avoir convertis en vecteurs d'embedding, il faut trouver dans cet espace vectoriel les vecteurs proches de l'embedding de l'image qui a été mise dans la requête de recherche. Ainsi donc il faut proposer un algorithme de Nearest-Neighbor (NN) Search Problem :

Definition. *Étant donné un ensemble P de points dans un espace de dimension d R^d , construire une structure de données qui étant donné un point q comme requête retourne les n points les plus proches de q dans P .*

Remarque : Le problème n'est pas complètement spécifié sans une distance définie.

Dans le cas d'une base de données très large, la dimensions d de l'espace vectoriel doit être suffisamment petite pour qu'il soit possible de stocker cette structure de données dans la mémoire des ordinateurs.

L'approche classique pour résoudre ce problème est un algorithme naïf, où on calcule les distances de q à chaque point dans P et donne les plus proches. Le problème de cette approche est qu'on a une complexité, pour un espace de dimension d avec n points en $O(dn)$. Cette approche est suffisante pour des bases de données petites, mais pour des bases de données plus larges, le temps d'une requête peut être long.

Ainsi, pour résoudre ce problème, des fonctions de hachages sont proposées dans la littérature, ce que nous avons utilisées.

Definition. *Une fonction de hachage est une fonction d'un espace $f : \Omega \rightarrow V$, où on utilise les points de V pour identifier les points de Ω . En particulier, on peut avoir $\Omega = R^d$ et $V = R^c$ avec $c < d$. Les points de V qui sont dans l'image de f sont appelés "code".*

Ainsi, avec une fonction de hachage on peut réduire la dimension en passant d'un espace de dimension d à un espace de dimension c , avec $c < d$. Cela réduit ainsi les temps de calcul et l'espace de stockage alloué. De plus, en utilisant des codes binaires, la réduction du temps de calcul et de stockage est encore plus forte. Ainsi le problème ne sera plus (NN), mais un problème de Approximative Nearest Neighbor :

Definition. Étant donné un ensemble P des points dans un espace de dimension d R^d , construire une structure de données qui étant donné un point q comme requête retourne n points dans P dont la distance à q est au moins c fois la distance de q à w , w étant le point plus proche de q .

Dans (9), selon Sovann En et al. les deux principales façons de générer des fonctions de hachage sont avec le Locality Sensitive Hashing (LSH) et le Learning to Hash. L'approche choisie est le LSH car c'est un algorithme peu gourmand en temps de calcul et qui répond rapidement à une requête.

4.6 LSHashing

Locality Sensitive Hashing (LSH) est un algorithme qui se base sur l'utilisation de plusieurs fonctions de hachage pour assurer que, pour chaque fonction de hachage, la probabilité que deux points aient le même code est plus grande pour des points qui sont proches dans l'espace de dimension d .

Definition. Une famille H de fonctions de hachage $f : R^d \rightarrow V$ est dite (r, cr, P_1, P_2) -sensitive si $\forall p, q \in R^d$:

- $\|p - q\| \leq r \Rightarrow P_H[h(q) = h(p)] \geq P_1$
- $\|p - q\| \geq cr \Rightarrow P_H[h(q) = h(p)] \leq P_2$

Et on a $P_1 > P_2$

Les algorithmes LSH sont des algorithmes capables de résoudre le problème de Aproximative Nearest Neighbors dont la définition est la même que la définition du problème de Nearest Neighbor. L'algorithme est basé sur la procédure suivante :

1. On partitionne notre espace de dimension d avec K hyperplans aléatoires représentés par les vecteurs perpendiculaires n_1, \dots, n_k
2. On génère les coordonnées du code du vecteur p de l'espace :

$$[hachage(p)]_i = \begin{cases} 1 & n_i^T p > 0 \\ 0 & n_i^T p \leq 0 \end{cases}, i = 1, \dots, K$$

3. On établit une table de hachage où la clé de p sera $hachage(p)$
4. On répète la procédure L fois, en construisant L tables de hachages.

A la fin de l'algorithme de LSH, le vecteur de l'espace de dimension d sera représenté par L codes de taille k . La Figure 12 montre un exemple d'un code généré par l'algorithme.

Ainsi, une méthode non plus exacte mais approchée, peut être mise en œuvre en utilisant les tables de hachage qui ont été construites. Il suffit de trouver le code de hachage, pour chaque table, de l'image présentée en requête, trouver les mêmes codes dans les L différentes tables et comparer l'image avec chaque objet ayant le même code (en utilisant par exemple des distances euclidiennes). Ainsi, il suffit de prendre les c images les plus proches.

Dans ce projet, le LSH a été utilisé en vue de la taille de la base de données utilisée.

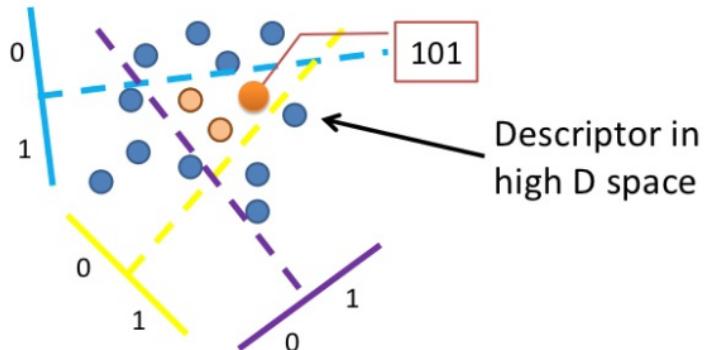


FIGURE 5: Exemple d'un code de hachage basé sur la procédure LSH

4.7 Learning to Hash

Une autre approche qui est présente dans la littérature est l'algorithme de learning to hash. Cette approche est basée sur l'utilisation d'un réseau de neurones qui produira un code binaire directement. Ce type d'approche est basé sur des algorithmes d'auto-encoders.

Dans un algorithme d'auto-encoders, il y a une partie du réseau responsable du codage et une autre partie responsable du décodage. La différence avec un auto-encoder normal ici est que la fonction de perte du réseau prendra en compte les valeurs du code pour qu'ils soient proches de valeurs binaires.

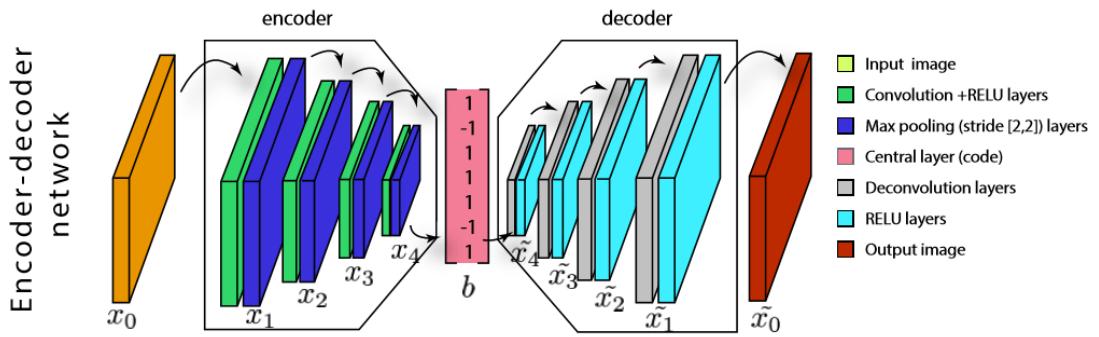


FIGURE 6: Exemple d'un code de hachage basé sur la procédure de Learning to Hash

Selon Sovann En et al., voilà ci-dessous une fonction de perte que l'on peu utiliser :

$$L = \frac{1}{N} \sum_{j=1}^N \|\hat{x}_0 - x_0\|^2 + \frac{\alpha}{N} \sum_{j=1}^N \sum_{i=1}^d \||b_i^j| - 1\|$$

avec

- N la quantité d'images prises en entrée
- x_0 la matrice représentant l'image avant le réseau
- \hat{x}_0 la matrice représentant l'image après un passage par le réseau
- $|b_i^j|$ les valeurs du code (qui seront arrondies à 1 ou -1)
- α une constante

4.8 Triplet Loss

Le problème de similarité d'image est très différent d'un simple problème de classification. Pour ce dernier, pour tout dataset il existe un nombre fini de classes, tandis que dans la reconnaissance d'image, ce nombre n'est pas constant. L'exemple pour lequel est né le concept de triplet loss est celui de reconnaissance de visage, sa première apparition étant dans la thèse de FaceNet (15).

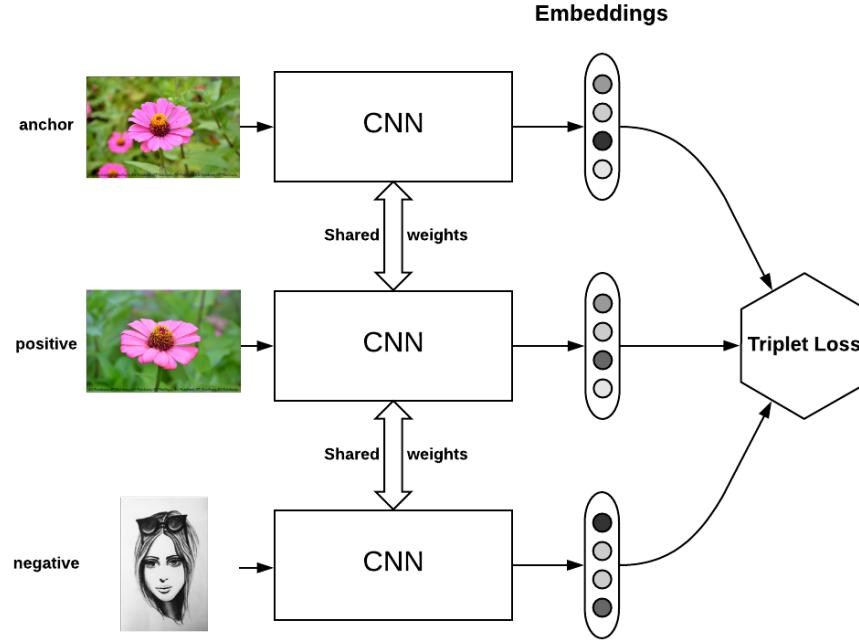


FIGURE 7: Illustration d'un triplet (le triplet provient de notre algorithme)

Ci-dessous est expliqué plus en détails la méthode du Triplet Loss :

Objectifs Faire en sorte que les embeddings de chaque "type d'image" forment de petits clusters cohérents et de les éloigner le plus possible les uns des autres.

Principe L'idée est ici d'entraîner le réseau de neurones par paquets de 3 exemples, chaque paquet étant appelé triplet. Un triplet est constitué de :

1. Un anchor, qui est l'image de référence
2. Un positif, c'est-à-dire une image de la classe de l'anchor
3. Un négatif, c'est-à-dire une image d'une classe différente de celle de l'anchor.



FIGURE 8: Exemple de triplets générés à partir des données de BAM

L'objectif de l'entraînement du réseau sera de minimiser le loss défini de la manière suivante, où nous désignons par a , p et n respectivement l'anchor, le positive et le negative et où d est la distance dans l'espace des embeddings en sortie du réseau de neurones :

$$\mathcal{L} = \max(d(a, p) - d(a, n) + \alpha, 0)$$

où $\alpha > 0$ est la marge.

L'équation est à comprendre dans ce sens : nous voulons avoir la distance anchor-positive très faible devant la distance anchor-negative. La présence de la marge α permet d'empêcher le réseau de renvoyer le même embedding pour toutes les images car pour $\alpha = 0$, l'équation est trivialement vérifiée si le réseau a un output constant. Ainsi, plus α sera grand et plus les negatives seront éloignées de l'anchor au sens de la distance euclidienne dans l'espace des embeddings. Enfin, la présence du max permet de ne pas modifier les poids du réseau si anchor, positive et negative sont convenablement éloignés les uns par rapport aux autres.

Ainsi, une fois l'entraînement terminé, nous nous attendons à avoir la représentation 2D des embeddings suivante :

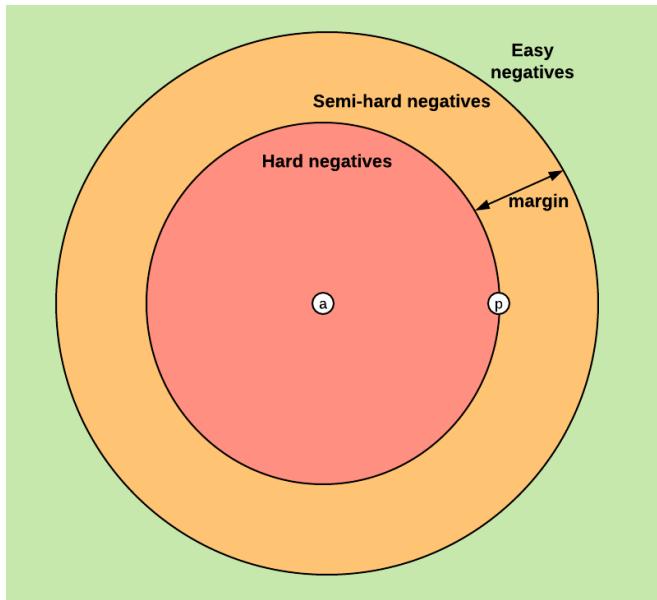


FIGURE 9: Interprétation géométrique de la marge

Pour cette étude, nous ne distinguerons pas les triplets hard des triplets semi-hard, c'est-à-dire qu'en pratique nous nous contenterons de prendre comme *negative* une image de classe différente de l'*anchor*.

4.9 YOLO

Pour rechercher des images similaires, il est judicieux de connaître les éléments essentielles qui composent une image, puisque naturellement des images similaires ont en commun ces éléments. On souhaite s'intéresser en priorité à eux et non pas à l'environnement qui les entoure car celui-ci peut brouiller la recherche. C'est ainsi qu'entre en jeu l'algorithme YOLO.

YOLO i.e. You Only Look Once est un algorithme de détection d'objet dans des images. Dans la littérature, il y aussi d'autres algorithmes tels que RetinaNet-50 et RetinaNet-101. Néanmoins, le YOLO est beaucoup plus rapide et efficace, d'où notre choix envers celui-ci.

Son fonctionnement fait appel à un CNN à 106 couches, qui est pré-entraîné sur 80 classes distinctes. Ici, l'image est découpé afin de former un grille $S \times S$ carrés. Dans chaque carré, il met d'abord en place des *bounding boxes* lorsqu'un objet est détecté. Il peut par ailleurs détecter plusieurs objets au sein d'un même carré grâce aux *anchor boxes*, des lots de bounding boxes. D'un autre côté, il assigne à chaque carré, les classes correspondantes aux objets. Finalement, il regroupe ces deux travaux et ne garde qu'une seule bounding box pour chaque objet, en supprimant celles les moins adaptés, celles qui ont une probabilité de détection faible et celles ne contenant pas l'objet convenablement.

Le fonctionnement de l'algorithme est expliquée à l'aide de l'exemple ci-dessous :

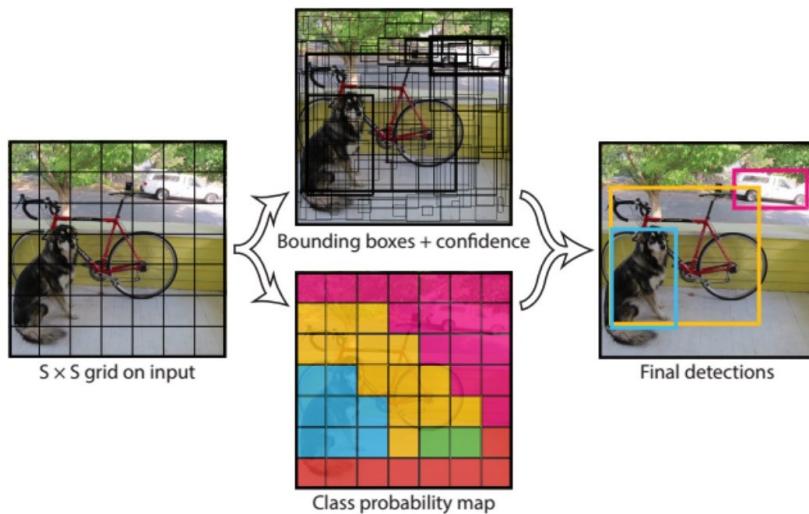


FIGURE 10: Fonctionnement et résultats du YOLO

5 Base de Données

5.1 Présentation

Il existe de nombreuses bases de données pour le traitement des images. On peut citer par exemple "ImageNet" l'une des bases de données les plus utilisées pour former les algorithmes "state of the art" pour la reconnaissance d'images. Cependant, peu de bases de données présentent une vision artistique de l'image comme le fait "The Behance Artistic Media Dataset" (1). C'est pour cette raison que cette base de données était un choix naturel pour la poursuite de ce travail.

"The Behance Artistic Media Dataset" (1) est une base de données construite à partir de <http://behance.net>, un site de portofolio pour les artistes professionnels et commerciaux. Behance compte plus de 65 millions d'images et chaque image est associée à un projet et chaque projet a un titre et une description.

Dans la base de données, chaque image contient 3 principaux types d'attributs : *Médias*, *Émotions* et *Contenu de l'image*. Un exemple de cette représentation est montré dans la Figure 11 ci-dessous.

- **Médias** : Ce sont des attributs binaires qui représentent si une image est de ce type de média ou non. Les types de médias catalogués sont les suivants : graphiques 3D sur ordinateur (media_3d), bandes dessinées (media_comics), peinture à l'huile (media_oil_painting), encre de stylo (media_pen_ink), dessins au crayon (media_graphite), art vectoriel (media_vector_art) et aquarelle (media_watercolor).
- **Émotions** : Ce sont aussi des attributs binaires qui représentent si une image apporte cette émotion ou non. Les types d'émotions catalogués sont les suivants : calme/pacifique, content / joyeux, triste / lugubre et effrayant.
- **Contenu** : Encore des attributs binaires mais cette fois-ci, ils représentent si l'image a le contenu indiqué. Les types de contenus sont : vélos, oiseaux, bâtiments, voitures, chats, chiens, fleurs, personnes et arbres

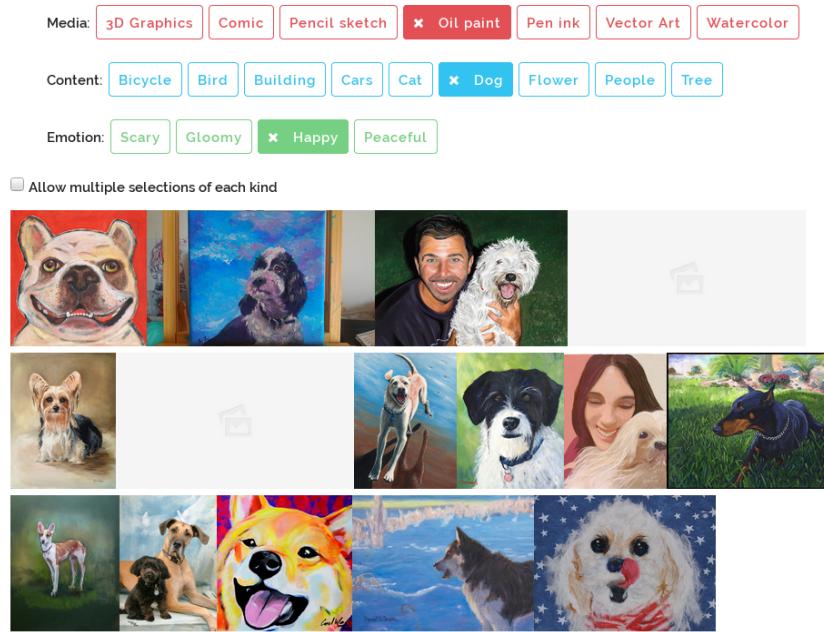


FIGURE 11: Exemple d'images de la base de données et représentation par leurs attributs. *Source : <https://bam-dataset.org>*

5.2 Prise en main

Nous avons réalisé un jupyter notebook pour explorer cette base de donnée. Nous avons créé une classe `data_base` adaptée à nos données. Dedans, nous avons différentes fonctions : sélectionner des données selon un label, rendre l'ID avec le lien en entrée ou réciproquement, puis des fonctions créant une sous base de donnée respectant différents critères. C'est grâce à ces fonctions, notamment celle intitulée `get_images`, que nous avons parcouru les données. Dans un premier temps, cette sous-base de données était transformée en dataframe pandas et affichait l'ID, le lien de l'image, les différents labels et le nombre de labels associés à l'image. Ensuite, nous avons utilisé l'outil interactif de ipywidgets pour explorer plus finement la db. Tout d'abord nous avons procédé à un affichage d'images : une sous database donnée, l'utilisateur peut choisir dans une liste déroulante l'ID de l'image qu'il souhaite s'afficher puis peut la visualiser. Ensuite, dans cette même sous database, l'utilisateur peut choisir au maximum un label par catégorie (émotions, techniques de réalisation, contenus) et la liste d'images (et leur caractéristiques) correspondant à tous ces labels s'affichent. Cela nous a donc permis de pouvoir exploiter une plus petite quantité d'images que les 10 000 initiales pour explorer la base de données, d'avoir une idée de la diversité d'images disponibles, et de mieux appréhender l'intersection de ces différentes catégories de labels.

e	emotion_scary			
m	All			
c	All			
MID	emotions	media labels	amount of labels	link to image
155 67093	emotion_scary	content_dog, emotion_scary	2	https://mir-s3-cdn-cf.behance.net/project_modu...
189 81389	emotion_scary	emotion_scary	1	https://mir-s3-cdn-cf.behance.net/project_modu...
190 95336	emotion_scary	content_people, emotion_scary	2	https://mir-s3-cdn-cf.behance.net/project_modu...
191 84131	emotion_scary	emotion_scary	1	https://mir-s3-cdn-cf.behance.net/project_modu...
192 5680	emotion_scary	emotion_scary	1	https://mir-s3-cdn-cf.behance.net/project_modu...
193 95349	emotion_scary	emotion_scary	1	https://mir-s3-cdn-cf.behance.net/project_modu...
194 112955	emotion_scary	emotion_scary	1	https://mir-s3-cdn-cf.behance.net/project_modu...
195 1559	emotion_scary	emotion_scary	1	https://mir-s3-cdn-cf.behance.net/project_modu...
196 52546	emotion_scary	emotion_scary	1	https://mir-s3-cdn-cf.behance.net/project_modu...

FIGURE 12: Capture d'écran d'une des fonctions

5.3 Utilisation et Prétraitement

Dans ce travail, une partie de cette base de données sauvegardée en format SQL a été utilisée, dans laquelle chaque image avait une clé unique, ses attributs et une url pour récupérer l'image. Dans ce base de données, il y avait plus de 1 million d'images.

Pour traiter cette forme de base de données, le bibliothèque python "sqlite3" (2) a été utilisée pour se connecter à la base de données en SQL et pouvoir utiliser ses commandes, principalement le "Select". La bibliothèque "requests" a été utilisée pour faire des demandes GET et récupérer des images à partir de son url, et enfin la bibliothèque "PIL" pour traiter le code binaire reçu comme une image.

Certaines étapes de nettoyage étaient nécessaires car de nombreuses urls n'étaient plus associées à l'image et leur suppression était alors nécessaire pour le bon déroulement du travail.

5.4 Data Generator

Pour faciliter la manipulation des images, une classe "Data Generator" a été créée afin qu'il ne soit pas nécessaire de télécharger toutes les images qui seront utilisées en même temps, ce qui serait très coûteux en termes de mémoire et très difficile pour la phase d'apprentissage des algorithmes.

Cette classe consiste à télécharger les images de manière dynamique et uniquement sur demande, puis à permettre la création d'une liste dynamique avec les images. Un autre point important à souligner de cette classe est qu'elle provient de la bibliothèque de keras. Son utilisation pour la phase d'entraînement des algorithmes vient donc de façon très naturelle.

À des fins d'optimisation, le "Data Generator" fonctionne par lots (dans ce travail, des lots de 32 images ont été faits). Par conséquent, le téléchargement des images se fait également par lots, c'est-à-dire que pour utiliser un lot, l'ensemble des images qui s'y rapportent sont téléchargés. L'une des conséquences négatives de cette situation est que si l'on souhaite utiliser une seule image de ce lot, le lot entier sera téléchargé. Il faut ainsi réduire le nombre d'opérations qui n'utilisent qu'une seule image et l'adapter pour utiliser des lots.

5.5 Data Generator pour l'entraînement

L'étape d'entraînement du CNN demande d'avoir à disposition un grand nombre d'images. Certes, les avantages de la classe "DataGenerator" sont indéniables une fois le modèle entraîné, mais il reste cependant peu pratique d'avoir à télécharger à chaque fois ces images pour l'entraînement.

Ainsi, nous avons créé une classe DataGenerator_training qui hérite de DataGenerator mais qui, au lieu de télécharger les images sur Internet, les récupère dans un dossier où ont été préalablement téléchargées les images du training set.

Le notebook "DataBase Downloader" permet de télécharger les images directement dans le dossier évoqué précédemment.

5.6 Images corrompues ou invalides

Un problème qui s'est rapidementposé à nous est de devoir exclure de l'entraînement et de la recherche d'image les images corrompues ou invalides. Nous avons donc utilisé un algorithme dans les notebooks "DataBase Downloader" et "Clean" qui permet d'obtenir une liste d'IDs renvoyant une image correcte.

6 Travail réalisé

Dans notre approche, l'idée est de créer un embedding avec les images (voir section 4.4), puis d'utiliser les vecteurs de dimension réduite pour pouvoir calculer la distance entre l'image d'entrée et celles de la base de données afin de renvoyer les images K les plus proches. Pour réduire le temps de recherche dans la base de données, la technique du Local Sensitive Hashing a été utilisée.

6.1 Embedding

L'embedding avec les images a été fait en utilisant la base de données décrite dans la section 5 "Base de données" en dimension de 256. Un classificateur a été réalisé à l'aide de Resnet50 pré-entraîné dans la base de donnée ImageNet, avec la dernière couche non utilisée. La technique de transfer learning a ensuite été utilisée et des couches supplémentaires ont été ajoutées pour entraîner le réseau. L'embedding a donc été créé en utilisant l'avant dernière couche du réseau. Le description du modèle est montrée ci-dessous :

Layer (type)	Output Shape	Param #
resnet50 (Model)	(None, None, None, 2048)	23587712
dropout_1 (Dropout)	(None, None, None, 2048)	0
conv2d_1 (Conv2D)	(None, None, None, 1024)	18875392
batch_normalization_1 (Batch Normalization)	(None, None, None, 1024)	4096
global_average_pooling2d_1 (Global Average Pooling2D)	(None, 1024)	0
dropout_2 (Dropout)	(None, 1024)	0
dense_1 (Dense)	(None, 1024)	1049600
dense_2 (Dense)	(None, 512)	524800
dense_3 (Dense)	(None, 256)	131328
dense_4 (Dense)	(None, 20)	5140

FIGURE 13: Description du modèle utilisé pour classer les images

Remarque : On se rend d'abord compte qu'un réseau ResNet50 formé a été utilisé dans la base de données ImageNet. Puis des couches de dropout ont été ajoutées pour limiter le phénomène d'overfitting : des couches convolutives et enfin 4 couches denses (dense_1, dense_2, dense_3 et dense_4). Il est important de voir que la couche qui servira de vecteur pour représenter l'image est la troisième.

6.1.1 Training

Le modèle a été entraîné en utilisant les 20 classes décrites dans la section "Base de données". Toutes les couches de Resnet50 sauf les dix dernières ont été maintenues fixes (c'est-à-dire non mises à jour par l'entraînement) et les dix dernières ont également été entraîné avec les autres couches. Le modèle a été compilé en utilisant la fonction de perte "binary crossentropy" et la métrique utilisée était la "accuracy".

Pour entraîner le modèle, 3000 images ont été utilisées et celles-ci étaient divisées en 2 groupes : un groupe de entraînement (qui représente 80 % des images) et un groupe test pour évaluer le modèle. Après 10 epoch, le modèle a obtenu une accuracy de 88 % dans le groupe de test.

6.2 LSHashing

Après la création de l'embedding, la technique LSHashing décrite dans les sections précédentes a été utilisée pour trouver les images les plus proches. Ici, l'implémentation a été utilisée en utilisant la bibliothèque python LSHash. La taille du hachage était de 10 bits et le nombre de tables était égal à 5 .

7 Résultats

L'ensemble des résultats du projet est le développement d'un modèle fonctionnel capable de réaliser la recherche d'image inversée. Le modèle se base sur l'approche décrite précédemment, afin de retourner des images de la base de données qui sont similaires à l'image en requête.

Notre approche se base sur les résultats empiriques : une série de 100 images a été donnée comme test et les résultats obtenus ont été, empiriquement, concluants. Le choix empirique est une conséquence de la difficulté de définir ce qu'est une image similaire et ce que cela n'est pas.

Un exemple de métrique afin de choisir le meilleur modèle évoqué dans la littérature consiste à dire que deux images sont similaires si elles appartiennent à la même classe. Cette métrique, pour le cas de la base de données BAM!, est particulièrement peu adaptée puisqu'une image peut être présente dans différentes classes.

Les résultats ont été rassemblés dans un site Internet pour que l'utilisation du modèle soit facilitée. La figure 14 illustre le site.

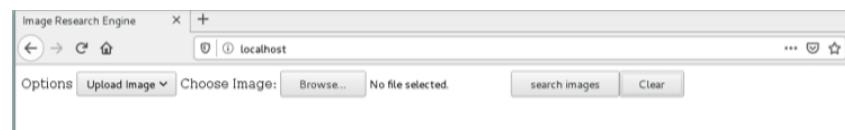


FIGURE 14: Capture d'écran du site élaboré

Ci-dessous, quelques résultats obtenus par le modèle sont présentés dans les figures 15, 16 et 17.



FIGURE 15: Exemple avec une peinture abstraite

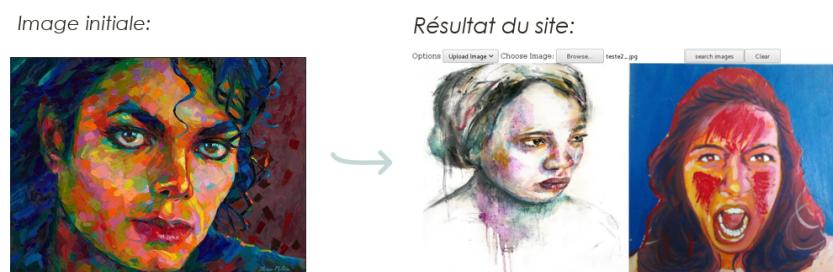


FIGURE 16: Exemple avec un visage humain

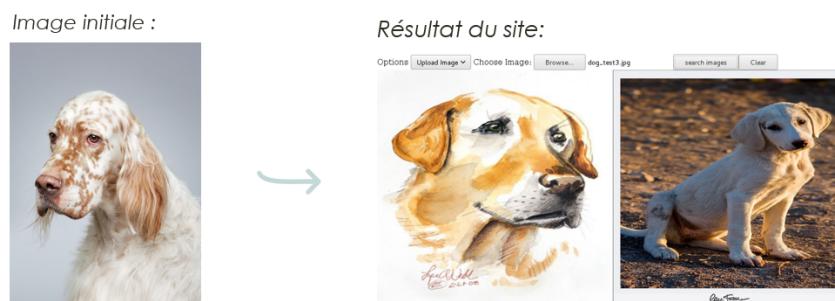


FIGURE 17: Exemple avec un animal

8 Travaux Futurs

Ayant passé un certain temps à prendre en main les notions et outils nécessaires à la reconnaissance d'image, nous n'avons pas pu finaliser toutes les fonctionnalités que nous souhaitions implémenter. Nous avons néanmoins livré un algorithme se fondant sur l'approche définie au début, mais le lecteur pourra également retrouver le code dans certains modules Python permettant de réaliser plus ou moins exactement certaines tâches qu'on détaillera ci-dessous.

Les travaux futurs peuvent être séparés en différents axes. Nous pouvons notamment considérer les axes suivants :

- **Test du module YOLO** L'algorithme de YOLO a été implémenté à l'aide de la bibliothèque *OpenCV* sur Python. Il a été greffé au reste du projet et utilisait ainsi le travail effectué en amont pour fonctionner. Nous avons donc dans un premier temps renvoyé une liste de sous-image de l'image en requête, les sous-images étant les objets détectés à l'aide du YOLO. Ensuite, comme ce qui a été expliqué précédemment, nous avons calculé pour chacune de ces sous-images les images de la Base de Données les plus proches de celles-ci. La distance des images trouvées a enfin été mise à jour en prenant en compte la taille de la sous-image correspondante et sa probabilité qu'elle a d'appartenir à sa classe. Il reste finalement à comparer les résultats à l'ajout de ce module avec les résultats précédents, afin de tester son efficacité.

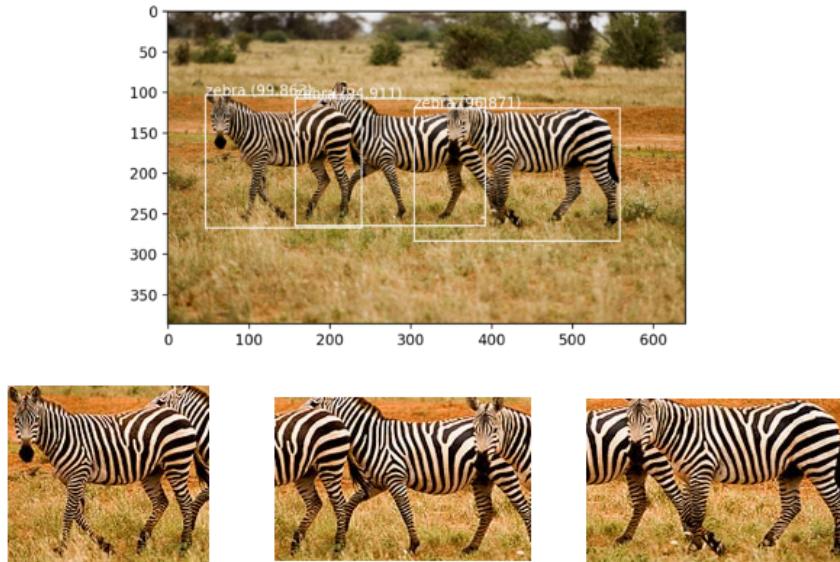


FIGURE 18: Mise en place du YOLO

- **Développement d'une métrique efficace montrant la qualité du modèle.** Elle prendrait en compte la sémantique de l'image donné en requête et des images renvoyées. La métrique que l'on a utilisée prend simplement en compte les classes de chaque image de la base de données.
- **Hyperparamètres Tuning.** Essayer plus de configurations de réseaux de neurones et d'hyperparamètres de l'algorithme de LSH pour améliorer la qualité des images renvoyées.
- **Construction d'un moteur de recherche avec différentes fonctions de hachage.** Utiliser la technique de "Learning to Hash" pour construire des codes qui seront égaux pour les images similaires, dans le but d'améliorer les images qui sont données lors d'une requête.
- **Ajouts de fonctionnalités au moteur de recherche.** Ajouter un mécanisme qui soit capable de rechercher des images dans le Web et les additionner dans la base de données. Ainsi le moteur de recherche serait amélioré mais il serait aussi applicable à la détection d'images identiques ou similaires présentes sur Internet.

9 Conclusion

Le projet proposé par Navee consistait à construire un moteur de recherche d'images similaires. Ce moteur de recherche devait être basé sur le contenu d'une image proposée en requête, afin de renvoyer des images similaires présentes dans une base de données. La complexité du problème est notable suite à la grande quantité d'images dans la base de données et au "gap" sémantique, c'est-à-dire comment obtenir des concepts de sémantiques complexes (les contenus des images) à partir d'un groupe de pixels, qui sont la façon de stocker l'image.

Plusieurs approches ont été proposées pour résoudre ce problème dans la bibliographie. En particulier, les modèles de "deep-learning" ont été récemment très utilisés pour construire des moteurs de recherches efficaces. Les différents couches avec la présence de couches de convolution sont capables d'obtenir les "features" d'une image et, ainsi, résoudre le problème de "gap" sémantique.

Après l'extraction de ces features il fallait obtenir une représentation plus compact de l'image pour qu'il soit plus facile de réaliser une recherche dans la base de données. Pour cela des fonctions de hachage que sont sensitives localement sont recommandés. Elles produisent des codes de hachages qui sont égaux pour des images qui ont des features similaires. En particulier le LSH est utilisé pour générer ces codes qui permettent une recherche efficace même dans une très large base de données.

Le moteur inversé de recherche construit dans le projet a été capable de faire la recherche des images similaires présentes dans une base de données d'oeuvre d'arts, la base BAM. Ce moteur utilise les algorithmes de deep learning et LSH pour faire cela.

Ce moteur a plusieurs applications. La principale contribution du moteur de recherche proposé est de montrer que l'approche réseaux de neurones couplée à l'utilisation d'une fonction de hachage est fonctionnel pour la détection d'images similaires. Ainsi dans un contexte de recherche d'images similaires ou dupliquées pour la détection de fraudes, cette approche est valable et le moteur de recherche pourrait être appliquée. Si on ajoute une base de données prenant des images de sites internet, ce moteur de recherche pourra être utilisé afin de détecter des images qui ont été soit légèrement modifiées, soit dupliquées.

Pour conclure, le projet peut servir de base pour le développement d'un algorithme pour Navee, qui peut l'utiliser pour son business modèle.

Bibliographie

- [1] Wilber, Michael J. and Fang, Chen and Jin, Hailin and Hertzmann, Aaron and Collomosse, John and Belongie, Serge *BAM! The Behance Artistic Media Dataset for Recognition Beyond Photography*. The IEEE International Conference on Computer Vision (ICCV), 2017, pp. 1202-1211.
- [2] Documentation de la bibliothèque : <https://www.sqlite.org>.
- [3] Sumit Saha *A Comprehensive Guide to Convolutional Neural Networks*, Dec 15, 2018, <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>
- [4] Tensorflow *Hyperparameter Tuning with the HParams Dashboard*, https://www.tensorflow.org/tensorboard/hyperparameter_tuning_with_hparams
- [5] Jason Brownlee *A Gentle Introduction to Transfer Learning for Deep Learning*, on December 20, 2017 in Deep Learning for Computer Vision
- [6] Andrew NG *Deep Learning Specialization - ML Strategy*, 2018, <https://www.coursera.org/specializations/deep-learning>.
- [7] Aayush Agrawal *Finding similar images using Deep learning and Locality Sensitive Hashing*. on 18/03/2019 in Towards DataScience.
- [8] Yona Gal *Fast Near-Duplicate Image Search using Locality Sensitive Hashing*. on 05/05/2019 in Towards DataScience.
- [9] Sovann En,Bruno Crémilleux,Frédéric Jurie *Unsupervised Deep Hashing With Stacked Convolutional Autoencoders*. IEEE International Conference on Image Processing, Sep 2017, Beijing, China.
- [10] Gregory Shakhnarovich, Trevor Darrell and Piotr Indyk *Nearest-Neighbors Methods in Learning and Vision. Theory and Practice* MIT Press, Octobre 13, 2006, 260
- [11] Wengang Zhou, Houqiang Li, and Qi Tian Fellow *Recent Advance in Content-based Image Retrieval : A Literature Survey* IEEE, arXiv, Septembre 02, 2017
- [12] Albert Gordo, Jon Almazan, Jerome Revaud, and Diane Larlus *Deep Image Retrieval : Learning global representations for image search* arXiv, Juin 28, 2016
- [13] A. Gionis, P. Indyk, and R. Motwani *Similarity search in high dimensions via hashing* In VLDB, pages 518–529, 1999. 2, 7, 8
- [14] Haomiao Liu^{1,2}, Ruiping Wang¹, Shiguang Shan¹, Xilin Chen¹ *Deep Supervised Hashing for Fast Image Retrieval*. Int J Comput Vis 127, 1217–1234 (2019)
- [15] Schroff, Florian, Dmitry Kalenichenko, and James Philbin *FaceNet : A unified embedding for face recognition and clustering* . “FaceNet : A Unified Embedding for Face Recognition and Clustering.” 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015) : n. pag. Crossref. Web.
- [16] Olivier Moindrot *Triplet Loss and Online Triplet Mining in TensorFlow*, Mar 19, 2018, <https://omoindrot.github.io/triplet-loss>
- [17] Jason Brownlee *How to Perform Object Detection With YOLOv3 in Keras*, October 19, 2019, <https://machinelearningmastery.com/how-to-perform-object-detection-with-yolov3-in-keras/>
- [18] Manish Chablani *YOLO — You only look once, real time object detection explained*, August 21, 2017, <https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-detection-explained-492dc9230006>
- [19] Joseph Redmon *YOLO : Real-Time Object Detection*, <https://pjreddie.com/darknet/yolo/>