ENSG Géomatique

ÉCOLE NATIONALE
DES SCIENCES
GÉOGRAPHIQUES

Ecole Nationale des
Sciences Géographiques

Rapport d'analyse du projet informatique

Cycle des ingénieurs diplômés de l'ENSG 3ème année

Application de méthodes de "Deep Learning" pour l'appariement de points d'intérêt dans de grands jeux de données

Guillemette Fonteix

Commanditaires : Ewelina Rupnik & Marc Pierrot-Deseilligny

Novembre 2018

☒ Non confidentiel ☐ Confidentiel IGN ☐ Confidentiel Industrie ☐ Jusqu'au ...

Ecole Nationale des Sciences Géographiques
6-8 Avenue Blaise Pascal - Cité Descartes - 77420 Champs-sur-Marne
Téléphone 01 64 15 31 00 Télécopie 01 64 15 31 07

Table des matières

Chapitre 1 Reformulation du sujet

1.1 Commanditaires

Je tiens avant toute chose à remercier mes commanditaires Ewelina Rupnik et Marc Pierrot-Deseilligny (LaSTIG, IGN) de me faire confiance pour la réalisation de ce projet informatique.

1.2 Contexte

L'appariement de points d'intérêt dans les images est la première étape dans la chaîne de traitement photogrammétrique. En photogrammétrie, SIFT (Scale Invariant Feature Transform) est l'algorithme typiquement utilisé. Il est invariant aux rotations, à l'échelle et partiellement aux transformations affines. Cependant, son temps de calcul est long et sa licence d'utilisation est contraignante.

C'est pourquoi, pour parer à ces désavantages et dans le cadre du développement d'un nouvel algorithme d'extraction (détection et appariement) de points homologues (AIME) au sein de l'IGN, il m'a été demandé d'utiliser des méthodes de Deep Learning afin d'apparier des points d'intérêt dans de grands jeux de données.

Les algorithmes de détection des points d'intérêt (détails remarquables dans les images) ont déjà été implémentés. De plus, des descripteurs caractérisant ces points ont été calculés. On peut voir ci-dessous une détection de points d'intérêt dans deux photos différentes.

Figure 1.1 – Exemple de détection de points d'intérêt sur deux photos différentes

Suite à la détection des points d'intérêt et à la création de descripteurs, il sera possible de retrouver les points homologues pour chaque couple d'images.

1.3 Problématique

Etant donné un point d'intérêt (i_1, j_1) de l'image 1 (image de référence), on peut retrouver son point homologue (i_2, j_2) dans l'image 2 (image secondaire). En effet en intersectant (i_1, j_1) avec la géométrie de la scène, on obtient un point 3D correspondant au point d'intérêt. Puis, en reprojetant le point 3D dans l'image secondaire on aura une position (x, y) d'un point homologue.

Cependant reste à savoir si l'implémentation d'un algorithme de Deep Learning peut être une alternative efficace pour l'appariement de ces points d'intérêt dans de grands jeux de données.

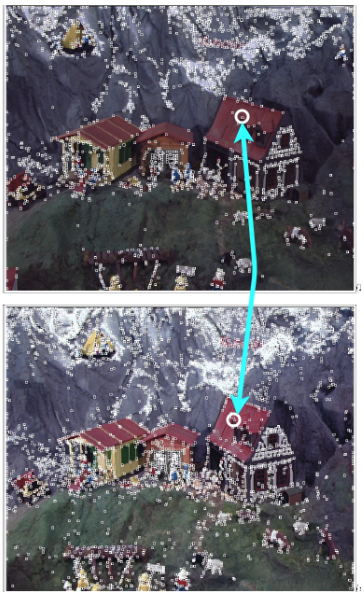

FIGURE 1.2 – Appariement de deux points homologues

Chapitre 2
Analyse du besoin

2.1 Objectifs

L'objectif de ce projet est de mettre en correspondance des points d'intérêt entre différentes images par l'intermédiaire d'algorithmes de Deep Learning.

Il me faudra réaliser un programme python qui indiquera à l'utilisateur, suite à un apprentissage, si deux points sont homologues par l'intermédiaire de leurs descripteurs.

2.1.1 Le Deep Learning : une solution pour l'appariement de points homologues ?

Qu'est-ce que le Deep Learning ?

Le Deep Learning est une forme d'intelligence artificielle, dérivée du Machine Learning.

Il repose principalement sur les algorithmes de réseaux de neurones artificiels. Ces algorithmes tendent à reproduire le fonctionnement interne du cerveau humain, en tentant de coller aux différentes étapes qui interviennent dans la prise de décision. C'est un système composé de plusieurs unités de calculs simples fonctionnant en parallèle, dont la fonction est déterminée par la structure du réseau, la solidité des connexions, et l'opération effectuée par les éléments ou nœuds (DARPA Neural Network Study Group (1988)). Plus le nombre de neurones est élevé, plus le réseau est « profond ».
Il ressemble au cerveau sur deux aspects :
1. La connaissance est acquise par le réseau au travers d'un processus d'apprentissage.
2. Les connexions entre les neurones, connues sous le nom de poids synaptiques, servent à stocker la connaissance. (S. Haykin (1994))

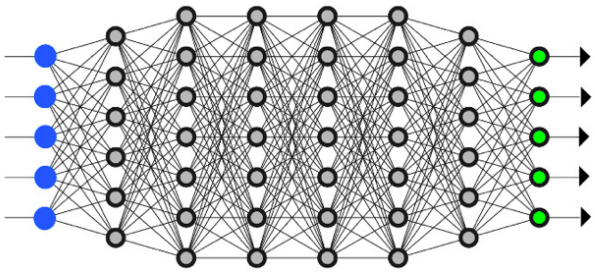

Figure 2.1 – Structure d'un réseau neuronnal artificiel

Comment l'utiliser dans notre cas ?

Pour apprendre, le réseau doit savoir qu'il a commis une erreur, et doit connaître la réponse qu'il aurait dû donner. De ce fait, on parle d'apprentissage supervisé. Il faudra donc dans un premier temps entraîner le réseau de neurones. Il est nécessaire de compiler un ensemble d'images d'entraînement pour pratiquer le Deep Learning. Cet ensemble va regrouper des centaines voire des milliers de photos de paires de descripteurs de points homologues mélangés avec des paires de descripteurs de points non-homologues. Ces images seront ensuite converties en données et transférées sur le réseau. Les

neurones artificiels assigneront alors des poids aux différents éléments. La couche finale de neurones rassemblera les différentes informations pour déduire si l'image correspond à un point homologue ou non.

Le réseau de neurones va alors comparer cette réponse aux bonnes réponses indiquées par l'apprentissage. Si les réponses correspondent, le réseau garde cette réussite en mémoire et s'en servira plus tard pour reconnaître les autres points. Dans le cas contraire, le réseau prend note de son erreur et ajuste le poids placé sur les différents neurones pour corriger son erreur. Le processus est répété des centaines de fois jusqu'à ce que le réseau soit capable de reconnaître les points homologues dans toutes les circonstances.

A partir de quelles données va t-on apprendre ?

Pour chaque couple d'images :
Une image au format .tif de taille n * 10* 20 pixels (n correspondant au nombre total de points homologues entre les deux images). Cette image associe les descripteurs des points homologues pour chaque paire d'images. Chaque descripteur est de taille 10 * 10 pixels.

FIGURE 2.2 – Descripteurs des points homologues entre les photos MG_0012 et MG_0022

FIGURE 2.3 – Zoom sur un couple de descripteurs

Comment ces descripteurs ont été trouvés et appariés ?

(1) Mise en place de l'ensemble des images (orientation interne et externe).
(2) Mise en correspondance (appariement dense).
(3) Détection des points d'intérêt avec AIME.
(4) Calcul des descripteurs pour chaque point d'intérêt avec AIME.
(5) Pour chaque point d'intérêt :

a. On l'intersecte avec la géométrie de la scène : on obtient un point 3D correspondant au point d'intérêt.

b. On projette ce point 3D dans des images voisines : à l'issue de cette opération on aura une position (x, y) d'un point homologue avec l'image de référence.

c. Il est probable que ça ne tombe pas exactement sur un point d'intérêt calculé alors on prend le point d'intérêt le plus proche. On connaît les descripteurs calculés à l'étape (4) et on les apparient.

On obtient alors des paires de descripteurs de points homologues qui permettront de réaliser l'apprentissage de la classification supervisée.

Résultats attendus

Après avoir entraîné le réseau de neurones nous obtiendrons alors un algorithme permettant de classer les points homologues et non-homologues dans n'importe quel jeu de données. Il nous permettra donc de réaliser l'appariement des points homologues au travers des descripteurs.

Plus précisément, l'algorithme indiquera avec quelle probabilité des points sont homologues.

2.2 Utilisateurs

Le programme sera utilisé par les développeurs de MicMac. Il s'agit ici d'un programme prototype, le but final étant de l'intégrer au nouvel algorithme de détection et d'appariement des points d'intérêt AIME conçu pour le logiciel de Photogrammétrie MicMac.

Chapitre 3
Analyse fonctionnelle

3.1 Base de l'outil

L'algorithme d'apprentissage peut se décomposer en quatre grandes parties comme indiqué sur le schéma 3.1.
Premièrement, il faudra préparer les données en créant des classes pour les points homologues et non-homologues. Puis subdiviser les données en échantillons d'apprentissage, de test et d'évaluation.
Ensuite, il s'agira de créer l'architecture du réseau neuronal (choix du nombre des couches de convolutions, pooling...) et de choisir les différentes caractéristiques de l'algorithme d'apprentissage (fonction d'activation, fonction de perte et algorithme d'optimisation). Nous pourrons aussi utiliser le Transfer Learning qui nous permettra de gagner du temps de calcul.
Les données pourront alors être entraînées et validées grâce à une validation croisée.
La dernière étape consistera à utiliser des données d'évaluation pour tester la performance des réseaux en concurrence et faire le choix du meilleur modèle.

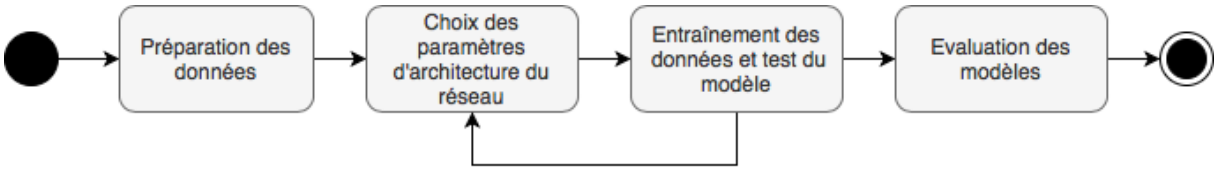

Figure 3.1 – Base de l'algorithme d'apprentissage

La flèche retournant à l'étape 2 indique que nous testerons différents paramètres pour créer nos modèles et si le temps nous le permet différents algorithmes pré-entraînés afin d'aboutir au final à un algorithme s'adaptant au mieux à notre problème.

Une description plus approfondie de ces fonctionnalités est donnée dans les prochaines parties.

3.2 Etape 1. Préparation des données

• Création d'imagettes :
Homologues de taille 10*20 (comportant les descripteurs des points homologues).
Non-homologues de taille 10*20 : on récupère deux descripteurs ne correspondant pas à un point homologue et on les associe. On réalisera le même nombre de paires non-homologues qu'il existe de paire de points homologues.
-> On obtient 2 classes : homologue et non-homologue.

• Transformation de ces imagettes en matrice (array ou tenseur selon la librairie utilisée).

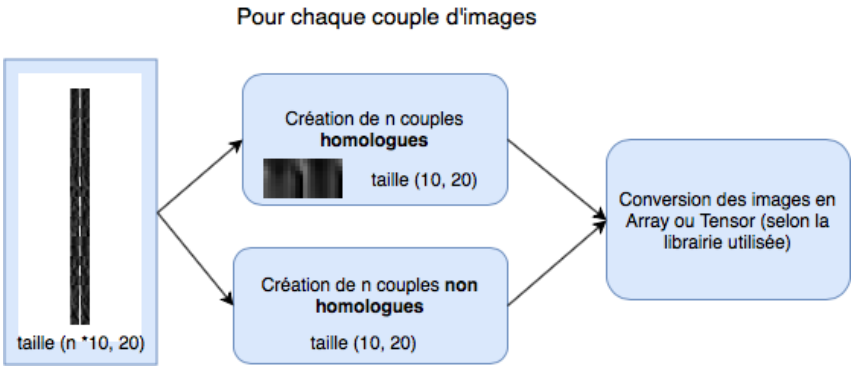

FIGURE 3.2 – Création des classes homologue et non-homologue

• Création de Xtrain, Ytrain, Xtest, Ytest, Xevaluation, Yevaluation.
Xtrain et **Ytrain** correspondent aux données **d'apprentissage/d'entraînement** (training set) qui vont servir à apprendre au réseau. On utilisera 80% des données globales.
Xtest et **Ytest** correspondent aux données de **test** (test set), qui servent à évaluer les progrès de notre algorithme. Elles vont permettre d'affiner les paramètres du modèle. Elles sont présentes à 20%.
XEvaluation et **YEvaluation** correspondent à données mises de côté dès le départ (données d'évaluation ou de validation). Elles ne sont utilisées qu'une fois le modèle complètement formé. Ces données serviront à évaluer les modèles en concurrence. On prendra ici 20% des données d'apprentissage (il faudra penser à les mettre de côté avant l'apprentissage).

Xtrain, Xtest, XEvaluation = matrice de pixels correspondant aux images.
Ytrain, Ytest, YEvaluation = labels (spécifiant si l'image correspond à des points homologues ou non-homologues).

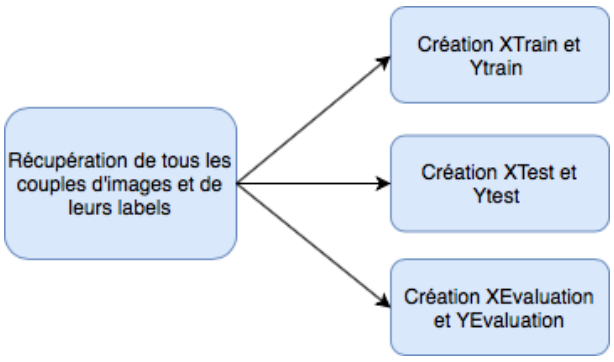

FIGURE 3.3 – Subdivision en échantillons d'apprentissage, de test et d'évaluation

Pour nous assurer que notre modèle ne souffre pas de sur-apprentissage, et qu'il saura faire des prédictions sur de nouvelles données on pourra utiliser une méthode de **validation croisée** (cross-validation). La validation croisée va nous permettre d'utiliser l'intégralité de notre jeu de données pour l'entraînement et pour les tests. Si le modèle s'exécute aussi bien sur l'échantillon test que sur l'échantillon d'apprentissage, nous pouvons dire que la validation croisée est bonne ou plus simplement, qu'il y a validation croisée.

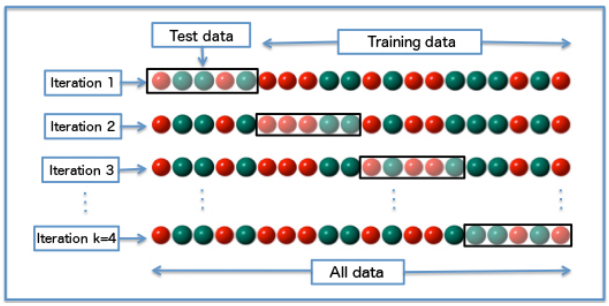

FIGURE 3.4 – Cross validation

3.3 Etape 2. Choix des paramètres d'architecture du réseau

Les algorithmes de Neural Network utilisent des fonctions d'activation et de perte ainsi qu'un algorithme d'optimisation. Voyons à quoi correspondent ces fonctions avant de rentrer dans les détails des architectures des algorithmes.

3.3.1 Les différents paramètres

La fonction d'activation

Cette fonction reçoit en entrée un nombre et effectue une opération propre à sa représentation associée (dans le cas de la fonction ReLU, l'opération est max(0, output)).

Voici quelques exemples de fonctions d'activation :

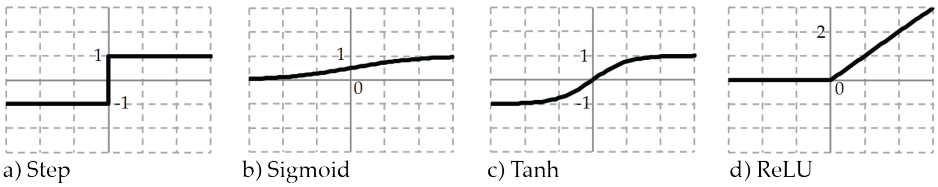

FIGURE 3.5 – Exemples de fonctions d'activation

La fonction de perte (loss function)

Elle quantifie l'**écart entre les prévisions du modèle et les observations réelles** du jeu de données utilisé pendant l'apprentissage.

C'est la fonction qui est minimisée dans la procédure d'ajustement d'un modèle. La phase d'apprentissage consistera à ajuster les paramètres.

Les moindres carrés sont utilisés en régression linéaire. Pour un problème de classification comme le nôtre, une des fonctions les plus utilisées actuellement est le « softmax loss », qui convertit le résultat de la dernière couche de neurones en probabilité pour chaque classe. La prédiction du modèle

correspond donc à la classe la plus probable.

Nous rencontrons cette fonction sur toutes les couches du réseau, hormis sur la dernière (appelée full-connected), car c'est celle-ci qui nous renseigne des « résultats » finaux et affecte une classe à une image.

L'algorithme d'optimisation

Les algorithmes d'optimisation nous aident à minimiser (ou à maximiser) la fonction de perte.

La **descente de gradient** a fait ses preuves pour l'optimisation des paramètres d'un réseau de neurones. Cependant, en apprentissage profond, la fonction objective que l'on cherche à minimiser est souvent non convexe et non régulière. La convergence de la descente du gradient vers le minimum global n'est donc pas garantie et la convergence même vers un minimum local peut être extrêmement lente. L'utilisation de l'algorithme de **descente de gradient stochastique** en est la solution. L'idée est de chercher à minimiser une fonction qui peut être écrite sous la forme de la somme de fonctions différentiables. Ce processus est alors réalisé de manière itérative sur des lots de données tirés aléatoirement. Chaque fonction objective minimisée de cette manière est une approximation de la fonction objective globale.

Il existe aussi plusieurs variantes de l'algorithme de descente de gradient dont Adam que l'on testera dans nos algorithmes. **Adam** est l'un des algorithmes les plus récents et les plus efficaces pour l'optimisation par descente de gradient. Il adapte automatiquement le taux d'apprentissage pour chaque paramètre.

3.3.2 Les différentes architectures

Dans un premier temps, nous allons essayer de comprendre ce qu'est un réseau de neurones « classique », **le perceptron simple**. Nous nous pencherons ensuite sur la structure du perceptron multicouche et enfin nous passerons à un réseau neuronal plus complexe, le **CNN (convolutional Neural Network)** plus adapté à la classification d'images. C'est ce dernier que nous chercherons à implémenter pour classifier nos images.

Structure générale d'un réseau de neurones

Avant toute chose il faut comprendre la structure générale d'un réseau de neurone.

Un réseau de neurones se compose de la manière suivante :
- 1 couche d'entrée
- N couche(s) cachée(s)
- 1 couche de sortie

Il faudra obligatoirement une couche d'entrée et une couche de sortie. Entre les deux, nous sommes libre du nombre de couches que nous souhaitons créer.

Nous allons maintenant voir quelques exemples de réseaux neuronaux.

Perceptron simple

Pour spécifier un perceptron simple, nous ajoutons une couche qui relie directement la couche d'entrée avec la couche de sortie, avec une fonction d'activation.

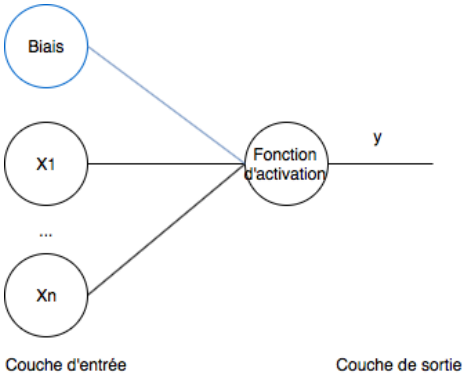

FIGURE 3.6 – Architecture du perceptron simple

Perceptron multicouche

Pour spécifier un perceptron multicouche, nous ajoutons successivement deux couches. La première fait la jonction entre la couche d'entrée et la couche cachée, la seconde entre cette couche cachée et la sortie. Nous avons une fonction d'activation sigmoïde dans les deux cas.

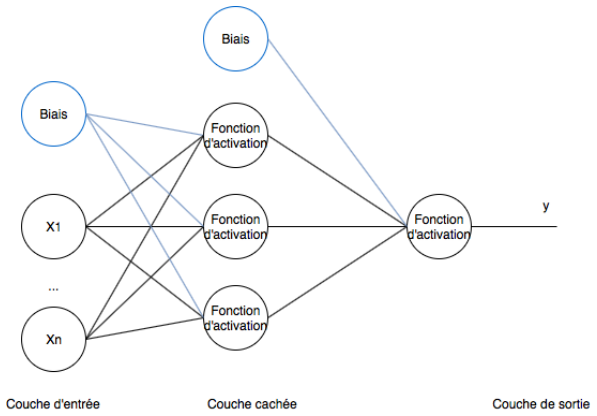

FIGURE 3.7 – Architecture du perceptron multicouche

Convolutional Neural Network (CNN)

Quelques éléments sur la structure d'un CNN

L'entraînement d'un CNN consiste à optimiser les coefficients du réseau, à partir d'une initialisation aléatoire, pour minimiser l'erreur de classification en sortie.

• **Backpropagation** :
En pratique, les coefficients du réseau sont modifiés de façon à corriger les erreurs de classification rencontrées. Ces erreurs sont rétropropagés dans le réseau depuis la couche de sortie, d'où le nom rétropropagation (backpropagation) donné aux algorithmes de réseaux de neurones.

• **Batch** :
On utilisera un entraînement par **batch** qui consiste à rétropropager l'erreur de classification par groupes d'images. Cette méthode est plus rapide qu'en calculant l'erreur sur tout le jeu d'entraînement à chaque itération. Elle est plus stable, car les gradients d'erreurs ont moins de variance. À noter qu'un nombre trop important d'images par batch peut engendrer des problèmes de mémoire lors de l'exécution du code.

• **Sur-détection et sous-détection** :

La sur-détection fait référence à un modèle qui modélise les erreurs d'observations sur les données d'entraînement. Cela se produit lorsqu'un modèle apprend les détails et le bruit dans les données de formation dans la mesure où cela affecte négativement les performances du modèle sur les nouvelles données. Le **« dropout»** est une stratégie implémentée afin de limiter l'overfitting. Il supprime aléatoirement et temporairement des liens entre les neurones, en fixant les poids de sortie à zéro.

La sous-détection fait référence à un modèle qui ne peut ni modéliser les données d'entraînement ni généraliser aux nouvelles données. Un modèle d'apprentissage automatique qui sous-détecte sera facilement détectable car les performances des données d'entraînement seront médiocres.

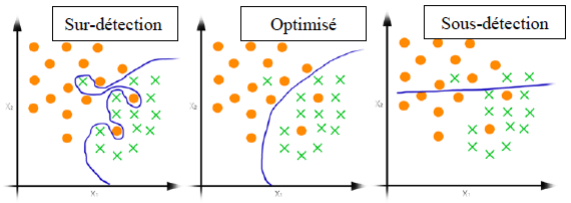

FIGURE 3.8 – Exemple des phénomènes de sur-détection et de sous-détection

Architecture

Un CNN est décomposé en 3 catégories distinctes :

• 1 couche d'entrée : qui contient l'image.
• n couche(s) convolutive(s) : elles vont mettre en valeur quelques caractéristiques bien choisies dans l'image source. Une image est passée à travers une succession de filtres, ou noyaux de convolution, visant chacune un aspect particulier de l'image, créant de nouvelles images appelées cartes de convolutions.

Un filtre va se déplacer sur toute l'image avec une certaine foulée (ou **stride** : correspond au nombre de pixels de déplacement du champ récepteur, évoluant en translation sur l'image) et calculer à chaque position les produits scalaires entre les entrées du filtre et l'entrée à la position i.

De plus, l'utilisation du **0-padding** qui consiste à ajouter des pixels supplémentaires en dehors de l'image permettra de faire des convolutions complètes.

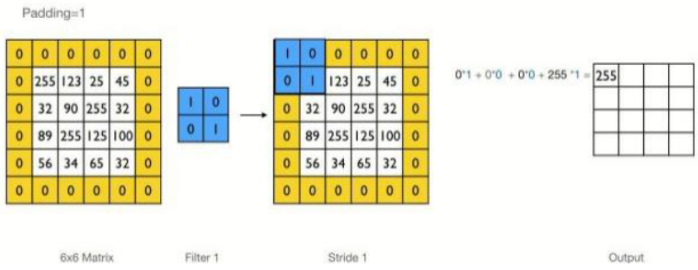

FIGURE 3.9 – Principe du 0-padding

Autre élément constituant notre CNN :

• N couche(s) pool : cette couche se localise entre les couches convolutives. Elles réduisent progressivement la quantité de variables, ce qui consiste en la réduction des dimensions des images (matrices). Le but de cette étape est de garder le maximum d'informations pertinentes même en réduisant les dimensions. Le plus commun est d'utiliser un pool de 2*2 -> stride de 2.

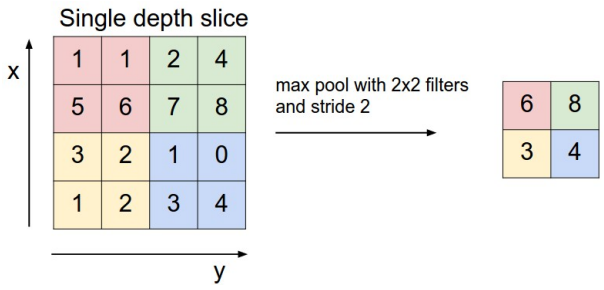

FIGURE 3.10 – Principe d'une couche pool

Un CNN est donc un empilement de plusieurs couches de convolution, pooling, correction (ou fonction d'activation) et fully-connected (voir figure 3.11). Chaque image reçue en entrée va être filtrée, réduite et corrigée plusieurs fois, pour finalement former un vecteur. Dans le problème de classification, ce vecteur contient les probabilités d'appartenance aux classes.

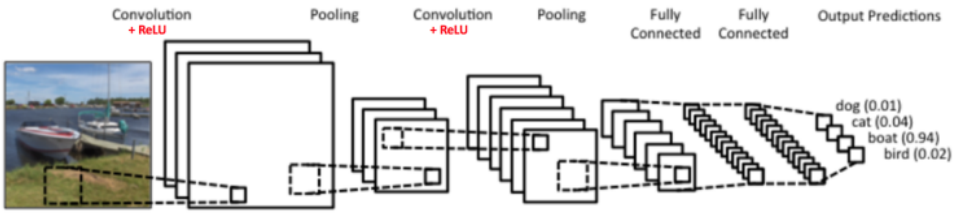

FIGURE 3.11 – Structure générale d'un CNN

Dans le cadre de ce projet, nous chercherons donc à créer une architecture d'un CNN s'adaptant à notre problème, on pourra tester différents paramètres afin de nous adapter au mieux à notre problème.

De plus, nous pourrons tester une structure de CNN prêt à l'emploi, que nous entraînerons sur notre problématique via le principe du **Transfer Learning**.

Autre technique : le Transfer Learning

Entraîner un réseau de neurones convolutif est très coûteux : si le nombre de couches est important, le nombre de convolutions et de paramètres à optimiser le sera aussi. L'ordinateur doit être en mesure de stocker plusieurs gigaoctets de données et de faire efficacement les calculs.

Le Transfer Learning (ou apprentissage par transfert) permet de réduire les temps de calculs et permettra d'entraîner le réseau plus rapidement.

Principes et utilités

Le principe est d'utiliser les connaissances acquises par un réseau de neurones lors de la résolution d'un problème afin d'en résoudre un autre plus ou moins similaire. On réalise ainsi un transfert de connaissances. Pour des usages pratiques, il est possible d'exploiter la puissance des CNN en adaptant des réseaux pré-entraînés disponibles publiquement. En plus d'accélérer l'entraînement du réseau, le Transfer Learning permet d'éviter le sur-apprentissage (overfitting).

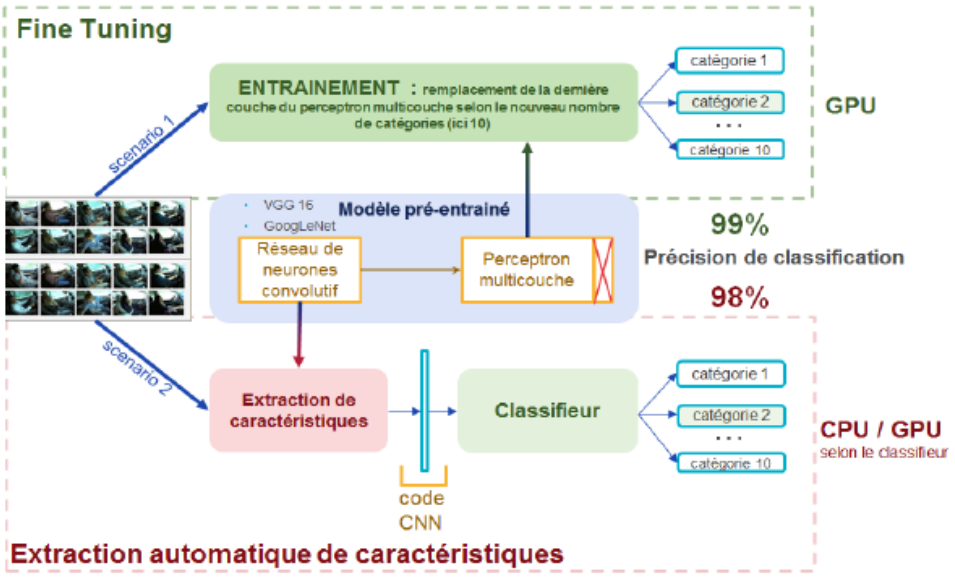

FIGURE 3.12 – Méthodologie du Transfer Learning

3 stratégies s'offrent à nous :

Stratégie 1 : fine-tuning total

On remplace la dernière couche fully-connected du réseau pré-entraîné par un classifieur adapté au nouveau problème et initialisé de manière aléatoire. Toutes les couches sont ensuite entraînées sur les nouvelles images.
La stratégie 1 doit être utilisée lorsque la nouvelle collection d'images est grande.

Stratégie 2 : extraction automatique de caractéristiques

Cette stratégie consiste à se servir des caractéristiques du réseau pré-entraîné pour représenter les images du nouveau problème. Pour cela, on retire la dernière couche fully-connected et on fixe tous les autres paramètres. Ce réseau tronqué va ainsi calculer la représentation de chaque image en entrée à partir des features déjà apprises lors du pré-entraînement. On entraîne alors un classifieur, initialisé aléatoirement, sur ces représentations pour résoudre le nouveau problème.
La stratégie 2 doit être utilisée lorsque la nouvelle collection d'images est petite et similaire aux images de pré-entraînement.

Stratégie 3 : fine-tuning partiel

Il s'agit d'un mélange des stratégies 1 et 2 : on remplace à nouveau la dernière couche fully-connected par le nouveau classifieur initialisé aléatoirement, et on fixe les paramètres de certaines couches du réseau pré-entraîné. Ainsi, en plus du classifieur, on entraîne sur les nouvelles images les couches non-fixées, qui correspondent en général aux plus hautes du réseau.
On utilise cette stratégie lorsque la nouvelle collection d'images est petite mais très différente des images du pré-entraînement.

Dans les trois cas, il faut remplacer les dernières couches fully-connected par un classifieur plus adapté à notre problème. Dans notre cas il ne sera pas possible de trouver des images de réseaux pré-entraînés similaires à nos descripteurs de points homologues. C'est pourquoi nous choisirons la **stratégie 3**.

3.4 Etape 3. Entraînement des données

Après avoir défini les caractéristiques de notre réseau nous pouvons lancer l'estimation des poids synaptiques (coefficients) du réseau à partir des données d'entraînement. On choisira alors un nombre maximum d'itérations et un nombre d'observations que l'on fait passer avant de remettre à jour les poids synaptiques.

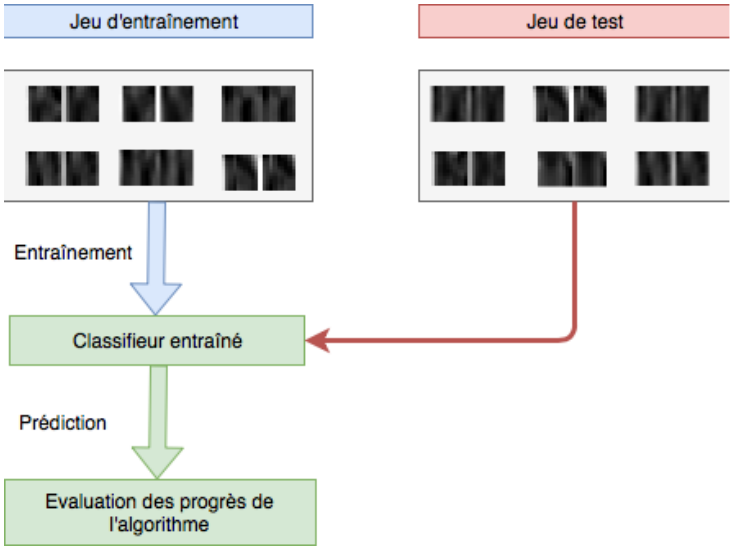

FIGURE 3.13 – Récapitulatif des étapes

Une fois l'apprentissage finalisé, nous pouvons afficher les classes estimées sur les données test et les comparer avec les classes réelles.

Nous pourrons aussi construire la **matrice de confusion** pour l'analyse des résultats de l'algorithme. C'est un outil servant à mesurer la qualité d'un système de classification. Chaque colonne de la matrice représente le nombre d'occurrences d'une classe estimée, tandis que chaque ligne représente le nombre d'occurrences d'une classe réelle (ou de référence).

3.5 Etape 4. Evaluation des modèles

Dans cette dernière étape nous évaluerons nos algorithmes. Le but est ici de comparer les différents algorithmes obtenus entre eux.

Dans les étapes précédentes, on testera plusieurs réseaux en faisant varier les paramètres et en changeant les données pré-entraînées pour le Transfer Learning.

Les données d'évaluation (ou validation) vont ainsi permettre de retenir le meilleur modèle.

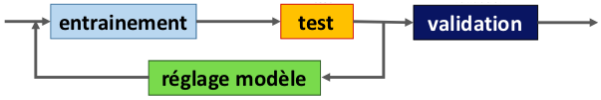

FIGURE 3.14 – Enchaînement de l'utilisation des données d'entraînement, de test et de validation

On pourra notamment utiliser la courbe ROC (Receiving Operator Characteristic) qui est communément utilisée pour mesurer la performance d'un classifieur.

Chapitre 4 Analyse technique

4.1 Description des données utiles

Pour entraîner un réseau de neurones nous avons besoin d'un grand nombre de données. Nous utiliserons donc toutes les descripteurs des couples d'images. Nous construirons le même nombre de paire de descripteurs non-homologue qu'il existe de descripteurs de points homologues.

4.2 Choix techniques

Le projet sera développé en Python qui met à disposition de nombreuses librairies performantes de Deep Learning.

4.2.1 Choix des librairies

J'ai envisagé d'utiliser différentes librairies pour le projet.

• **Tensorflow** est une librairie open-source développée par l'équipe Google Brain. Elle implémente des méthodes d'apprentissage automatique basées sur le principe des réseaux de neurones profonds.

Figure 4.1 – Logo TensorFlow

• **Keras** est une API de haut niveau Python qui encapsule l'accès aux fonctions proposées par plusieurs librairies de machine learning, en particulier Tensorflow. De fait, Keras n'implémente pas nativement les méthodes. Elle sert d'interface avec Tensorflow simplement. Elle propose des fonctions et procédures relativement simples à mettre en œuvre. Depuis sa publication initiale en mars 2015, il a été favorisé pour sa simplicité d'utilisation et sa simplicité syntaxique, facilitant ainsi son développement rapide. C'est une librairie plus ancienne, donc plus documentée.

Figure 4.2 – Logo Keras

• **PyTorch**, est une API de niveau inférieur axée sur le travail direct avec les expressions de tableau. Il a suscité un vif intérêt et est devenu une solution privilégiée pour la recherche et les

applications d'apprentissage en profondeur nécessitant l'optimisation des expressions personnalisées. Il est en effet parfaitement optimisé pour utiliser la puissance des cartes graphiques (GPU) pour les calculs longs et difficiles.

PyTorch est une bibliothèque d'apprentissage machine open-source pour Python basée sur Torch (autre bibliothèque). Elle est principalement développée par le groupe de recherche d'intelligence artificielle de Facebook.

PyTorch offre un environnement d'expérimentation avec plus de liberté pour écrire des couches personnalisées et rentrer dans les détails des tâches d'optimisation numérique.

FIGURE 4.3 – Logo Pytorch

Quel choix de librairie ?

Keras : API plus concise et plus simple.

PyTorch : plus flexible, encourageant une compréhension plus profonde des concepts d'apprentissage en profondeur.

Nous utiliserons donc dans un premier temps Pytorch qui semble être plus performant et nous permettra de développer un algorithme s'adaptant au mieux aux données d'entrées.

Cependant, l'utilisation de Keras reste envisageable si je rencontre des problèmes de programmation avec Pytorch.

4.3 Un peu plus de détails sur les librairies

4.3.1 Liste des paquets PyTorch

Voici l'ensemble des paquets disponibles au sein de cette librairie :

• **Torch** : bibliothèque TENSOR comme NumPy, avec un support puissant du GPU.
• **Torch.autograd** : bibliothèque de différenciation automatique qui prend en charge toutes les opérations de Tensor dans Torch.
• **Torh.nn** : bibliothèque de réseaux de neurones.
• **Torch.optim** : bibliothèque d'optimisation à utiliser avec Torch.nn contenant les fonctions d'optimisations standard telles que SGD, RMSProp, LBFGS, Adam.
• **Torch.multiprocessing** : multitraitement en python. Il est souvent utilisé pour le chargement de données ou bien l'entraînement de réseaux.
• **Torch.utils** : bibliothèque composée de DataLoader, Trainer, et de tout un ensemble de fonctions utiles pour plus de commodité.
• **Torch.legacy** : bibliothèque composée de fonctions compatibles avec PyTorch.

4.3.2 Liste des réseaux pré-entraînés via le package torchvision.models

• **AlexNet** : est le nom d'un réseau de neurones convolutif. La profondeur du modèle est essentielle pour sa haute performance, il est donc coûteux en termes de calcul.
• **VGG** : Le réseau VGG se caractérise par sa simplicité, n'utilisant que 3 3 couches convolutives

empilées les unes sur les autres en profondeur croissante. Les deux principaux inconvénients de VGGNet sont que la phase d'apprentissage est considérable et que les poids de l'architecture réseau sont assez importants.
• **ResNet** : Un réseau résiduel profond (Deep ResNet) est un type de réseau neuronal spécialisé qui aide à gérer des tâches et des modèles d'apprentissage en profondeur plus sophistiqués.
• **SqueezeNet** : Initialement décrit dans un article « SqueezeNet : précision au niveau d'AlexNet avec 50x moins de paramètres et une taille de modèle <0.5MB ». AlexNet est un réseau neuronal profond qui a 240 Mo de paramètres alors que SqueezeNet n'en a que 5 Mo. Ces deux réseaux peuvent atteindre à peu près le même niveau de précision lors de l'évaluation sur le jeu de données ImageNet.
• **DenseNet** : Réseau dit « dense », qui se caractérise par son système de propagation des cartes de caractéristiques ainsi que les connexions entre les couches.

Il est envisagé d'utiliser **ResNet et/ou VGG** lors du Transfer Learning.

4.3.3 Structure d'un CNN dans Pytorch

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.drop_out = nn.Dropout()
        self.fc1 = nn.Linear(7 * 7 * 64, 1000)
        self.fc2 = nn.Linear(1000, 10)
```

FIGURE 4.4 – Exemple d'architecture d'un CNN dans Pytorch

4.3.4 Structure d'un CNN dans Keras

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1),
                 activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(1000, activation='relu'))
model.add(Dense(2, activation='softmax'))
```

FIGURE 4.5 – Exemple d'architecture d'un CNN dans Keras

Chapitre 5 Conclusion et planning prévisionnel

5.1 Conclusion

Nous utiliserons donc dans un premier temps la librairie **Pytorch**, flexible et performante, pour créer notre **Convolutional Neural Network** et nous ferons varier les paramètres pour aboutir à un algorithme s'adaptant au mieux à notre problématique.

Il peut aussi être envisagé d'utiliser les connaissances acquises par un réseau de neurones pré-entrainé lors de la résolution de notre problème via le **Fine Tuning partiel**.

De plus, le but étant d'aboutir à un algorithme focntionnel la librairie **Keras** pourra être utilisée si je rencontre des problèmes avec la librairie Pytorch, puisque celle-ci est plus simple d'utilisation.

5.2 Planning prévisionnel

Cette phase de programmation a été organisée selon un calendrier prévisionnel, qui pourra évoluer au cours du processus (voir figure 5.1).

Il se décompose en plusieurs phases :

Etape 1. Préparation des données.
Etape 2.
a. Création de notre CNN.
b. Entraînement des données et test du modèle.
Etape 3.
a. Modification d'un réseau CNN pré-entraîné s'adaptant à notre problème grâce au Transfer Learning.
b. Entraînement des données et test du modèle.
Etape 4. Evaluation des algorithmes et choix du meilleur modèle.

Les étapes 2 et 3 seront réitérées afin d'affiner les paramètres du modèle, nous pourrons de plus créer différents modèles à partir de différents algorithmes pré-entraînés afin d'obtenir l'algorithme s'adaptant le mieux à notre cas si le temps nous le permet.

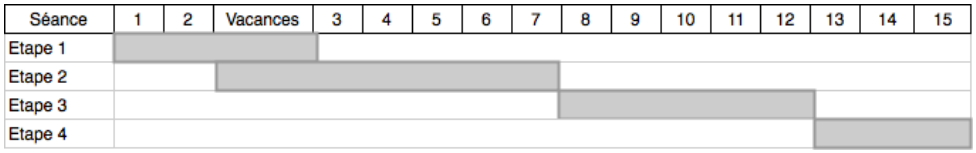

Figure 5.1 – Planning prévisionnel

Pendant toute la durée du développement le rapport d'utilisateur, le rapport développeur et le rapport de développement seront réalisés. De plus, un dépôt GitHub va être créé pour permettre les sauvegardes et le suivi du projet.

Bibliographie

Généralités sur le Deep Learning :

- https://www.lebigdata.fr/deep-learning-definition
- http://penseeartificielle.fr/focus-reseau-neurones-artificiels-perceptron-multicouche/
- https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7
- https://deepsense.ai/keras-or-pytorch/
- https://rfiap2018.ign.fr/sites/default/files/ARTICLES/RFIAP_2018/RFIAP_2018_Gillot_Algorithmes.pdf

Documentation des librairies :

- https://keras.io
- https://pytorch.org/docs/stable/index.html

Tutoriels pouvant être utiles à la programmation :

Tutoriel Pytorch :

- https://github.com/pytorch/tutorials/blob/master/beginner_source/transfer_learning_tutorial.py
- https://pytorch.org/tutorials/

Tutoriel Keras :

- Apprenez à construire un CNN et gagnez du temps avec le Transfer Learning
- Image Classification in Python
- Tutorial: Optimizing Neural Networks using Keras
- Implémentation de perceptrons simples et multicouches

Table des figures