Rapport d'IMPACT
A Deep Learning Approach To Universal Image Manipulation Detection Using a New Convolution Layer

ÉCOLE CENTRALE DE LILLE
OPTION DÉCISION ET ANALYSE DE DONNÉES

Tuteurs : Patrick BAS et John KLEIN
Étudiant : Ludovic DARMET

Année 2016-2017

Table des matières

Les forensics sont un sous-domaine de la cryptographie dont le but est d'arriver à détecter des manipulations d'images. Typiquement ces modifications ne sont pas visibles à l'œil nu et demande donc l'aide de la vision par ordinateur.

Le problème qui nous intéresse ici est de pouvoir détecter automatiquement des modifications du type bruit gaussien additif à faible variance dans une image. Dans le cadre de l'apprentissage statistique, il s'agit d'une tâche de classification. Il faut pouvoir dire si les images appartiennent à la classe originale ou modifiée. On parle plus précisément de classification supervisée puisque notre outil sera construit en utilisant une base d'exemples labellisés, que l'on appelle base d'entraînement.

Cette base d'entraînement provient d'images scrapées sur Flickr. Il y en a au total 400 000 originales. Les images perturbées sont produites à partir de cette base. Elles sont toutes en noir et blanc et de taille 256x256 pixels.

Pour attaquer ce problème, nous allons utiliser ici des réseaux de neurones convolutifs (Convolutional Neural Network ou CNN) avec une couche de convolution particulière comme suggéré dans l'article "A Deep Learning Approach to Universale Image Manipulation Detection Using a New Convolution Layer"[1]. Cette approche est relativement récente (l'article date de 2016) et a été peu explorée pour le moment (2 citations en mars 2017). L'objectif a donc été de reproduire le protocole décrit dans l'article puis de faire une étude de sensibilité sur différents paramètres et de tenter quelques améliorations.

Chapitre 1

Les réseaux de neurones convolutifs

1.1 Principe des réseaux de neurones

L'idée et le début de développement des réseaux de neurones date du XXème siècle (Figure 1.1). L'inspiration provient de la biologie et des neurones du cerveau humain. Mais ce n'est qu'une inspiration, le but n'étant pas reproduire le fonctionnement du cerveau, de la même manière qu'une aile d'avion s'inspire des ailes d'oiseau mais ne prétend pas en reproduire le fonctionnement.

Cette technique a connu une première vague de popularité jusqu'à la fin des années 90, où les SVM (Support Vector Machine) ont pris le dessus. Le domaine est ensuite resté relativement confidentielle jusqu'en 2012. Il ne parvenait pas à concurrencer les autres méthodes d'estimation statistique en terme de résultats et demandait de toute façon beaucoup trop de puissance de calcul. Puis a eu lieu un véritable retournement et c'est maintenant la technique la plus en vogue. Ce retournement provient principalement de trois facteurs :

- les moyens techniques ont très franchement progressé : développement des GPU et de serveurs de calculs bien plus puissants,
- des bases d'exemples très larges ont pu être construites (ImageNet,...),
- l'introduction de la couche de convolution par Yann Lecun qui partage certains paramètres sur plusieurs neurones.

Le glissement vers ces techniques a débuté par une victoire de réseaux de neurones convolutifs en 2012 à la compétition ImageNet Large-Scale Visual Recognition Challenge, qui est en quelque sorte les jeux olympiques de la vision par ordinateur. Cette année là les résultats ont progressé de 10% quand ils ne progressaient que de quelques pourcents les années précédentes.

Table 1: Major milestones that will be covered in this paper

Year	Contributer	Contribution
300 BC	Aristotle	introduced Associationism, started the history of human's attempt to understand brain.
1873	Alexander Bain	introduced Neural Groupings as the earliest models of neural network, inspired Hebbian Learning Rule.
1943	McCulloch & Pitts	introduced MCP Model, which is considered as the ancestor of Artificial Neural Model.
1949	Donald Hebb	considered as the father of neural networks, introduced Hebbian Learning Rule, which lays the foundation of modern neural network.
1958	Frank Rosenblatt	introduced the first perceptron, which highly resembles modern perceptron.
1974	Paul Werbos	introduced Backpropagation
1980	Teuvo Kohonen	introduced Self Organizing Map
	Kunihiko Fukushima	introduced Neocogitron, which inspired Convolutional Neural Network
1982	John Hopfield	introduced Hopfield Network
1985	Hilton & Sejnowski	introduced Boltzmann Machine
1986	Paul Smolensky	introduced Harmonium, which is later known as Restricted Boltzmann Machine
	Michael I. Jordan	defined and introduced Recurrent Neural Network
1990	Yann LeCun	introduced LeNet, showed the possibility of deep neural networks in practice
1997	Schuster & Paliwal	introduced Bidirectional Recurrent Neural Network
	Hochreiter & Schmidhuber	introduced LSTM, solved the problem of vanishing gradient in recurrent neural networks
2006	Geoffrey Hinton	introduced Deep Belief Networks, also introduced layer-wise pretraining technique, opened current deep learning era.
2009	Salakhutdinov & Hinton	introduced Deep Boltzmann Machines
2012	Geoffrey Hinton	introduced Dropout, an efficient way of training neural networks

FIGURE 1.1 – Historique des réseaux de neurones [4]

L'idée de base est d'utiliser plusieurs opérateurs basiques, des neurones, qui effectuent chacun une opération non-linéaire. Ces neurones sont organisés en couches successives. L'enchaînement de ces non-linéarités permet d'approcher des fonctions de décisions complexes. Cependant les fondements mathématiques de ces techniques restent encore une question ouverte à l'heure actuelle. L'avantage principal, outre la performance, est que le réseau peut apprendre à extraire et organiser des descripteurs, au lieu de devoir les construire de manière plus « artisanale », avec une intervention humaine dans une démarche classique d'apprentissage statistique.
Chacun des neurones possèdent plusieurs paramètres à régler, que l'on appelle les poids du réseau. Ils décrivent comment chacune des entrées va être prise en compte. Il faut pouvoir apprendre ces poids pour s'adapter à un problème. Classiquement ces poids sont appris grâce à une base d'apprentissage et grâce à la back-propagation.
De plus pour traiter les images, on ne fait pas appel à un simple neurone qui prend juste une valeur entrée, mais à des filtres de convolution qui prennent des entrées en 2 dimensions ou plus.

1.2 Les filtres de convolution

L'expression analytique d'un filtre de convolution est relativement simple :

$$\mathbf{h}_j^{(n)} = \sum_{k=1}^{K} \mathbf{h}_k^{(n-1)} * \mathbf{w}_{kj}^{(n)} + b_j^{(n)}$$

Avec $\mathbf{h}_j^{(n)}$ la $j^{ième}$ colonne de la sortie de la couche n, $h_k^{(n-1)}$ la $k^{ième}$ colonne du filtre (kernel) de la couche précédente, $w_{kj}^{(n)}$ le poids d'indice (k,j) dans la matrice des poids w (à apprendre pour chaque neurone) du filtre de convolution et $b_j^{(n)}$ un terme de biais (à apprendre également). Les poids et les biais sont partagés entre neurones.
On peut voir une image comme une fonction à 2 dimensions, tout comme le noyau (kernel) qui va être convolué avec l'image. Le noyau (plus petit que l'image) se déplace sur toute l'image d'entrée, pour effectuer à chaque fois une convolution qui donne la valeur d'un pixel en sortie (en rouge sur le schéma). La valeur d'un pixel est une moyenne pondérée des valeurs sur lequel le noyau est venu effectué une convolution avec l'image d'entrée.

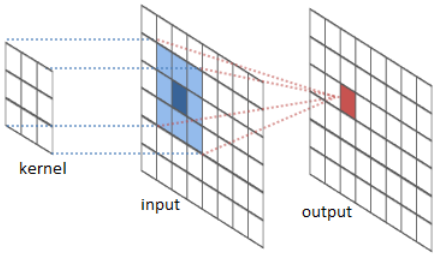

FIGURE 1.2 – Un noyau et une convolition [2]

Après le passage par une convolution, les valeurs de sortie vont passer par une fonction d'activation. Le plus souvent cette fonction d'activation est non-linéaire. Elle a pour but de gommer les valeurs extrêmes et laisser plus de variations aux valeurs moyennes, en plus d'introduire une non-linéarité bien sûr. Le principe est le même que pour le passage par une sigmoïde dans une régression logistique.

Cette étape est généralement suivi d'une étape de « pooling » où chaque matrice à la sortie de la couche de convolution est réduite à une taille inférieure, soit en prenant la moyenne par zone ou le maximum. Par exemple un max-pooling 4x4 va faire passer d'une matrice 8x8 à un matrice 4x4 en prenant le maximum sur le coin supérieur gauche de taille 4x4, le maximum sur le coin supérieur droit de taille 4x4 et ainsi de suite. C'est une forme de régularisation.

Pour des images 256x256, les filtres de convolutions ont habituellement une

taille inférieur à 10x10 et se réduisent au fur et à mesure que les convolutions s'enchaînent. Une cellule de pooling à une taille de l'ordre de 3x3, 4x4. En plus on trouve habituellement du « stride » ou recouvrement, c'est à dire que les filtres de convolutions ne s'appliquent pas à des zones disjoints de l'image mais se recouvrent de 1 ou 2 pixels.

1.3 La back-propagation

Comme nous l'avons vu précédemment dans chaque couche de convolution il y a des poids et des biais à régler. Cela va être fait durant la phase d'entraînement grâce à la back-propagation.
La fonction globale que l'on cherche à minimiser est la cross-entropy ou negative log likelihood :

$$E = \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{c} y_i^{*(k)} log(y_i^{(k)})$$

Avec y_i^*le véritable label et $y_i^{(k)}$la probabilité d'appartenance à la classe k pour la $i^{ème}$image. C'est elle qui décrit à quel point notre réseau classe bien les exemples qu'il a disposition.
Pour cela, on va faire une passe « backward », c'est à dire à partir du label d'une image on va chercher à trouver les poids optimaux en « remontant » dans le réseau. On part du problème à optimiser sur la dernière couche qui va dépendre de la couche précédente et ainsi de suite, d'où le nom de "BackPropagation". On résout alors un par un tous ces problèmes imbriqués. C'est un passage de backpropagation et plusieurs vont se succéder. On enchaîne ainsi entre passage forward pour tester et backward pour régler le réseau. Chaque groupe d'image envoyé dans le réseau est appelé un batch.

Le plus souvent cette optimisation est faite par descente de gradient stochastique pour deux raisons :

- une méthode plus évoluée type Newton serait beaucoup trop énorme en coup de calcul pour un gain marginal
- l'espace des solutions contient un très grand nombre de minima locaux dont il faut pouvoir se sortir.

La différence entre une descente de gradient classique et une descente de gradient stochastique est que dans le cas stochastique, le gradient n'est pas remis à jour entièrement à chaque itération. On sélectionne juste un batch de données qui va mettre à jour le gradient. Cette apprentissage se fait donc « en ligne » c'est à dire qu'on ne présente à chaque fois que quelques exemples pour mettre à jour le gradient.
La mise à jour à chaque itération se formule de la manière suivante :

$$\begin{aligned} \mathbf{w}_{ij}^{(n)} &= \mathbf{w}_{ij}^{(n)} + \eta \nabla \mathbf{w}_{ij}^{(n)} \\ \nabla \mathbf{w}_{ij}^{(n)} &= m * \nabla \mathbf{w}_{ij}^{(n)} - d * \epsilon * \mathbf{w}_{ij}^{(n)} - \epsilon * \frac{\partial E}{\partial \mathbf{w}_{ij}^{(n)}} \end{aligned}$$

Avec $w_{ij}^{(n)}$le poids d'indice i et j dans la $n^{ième}$couche de convolution, η est le « learning rate », m est le « momentum » et le « decay ». Le momentum et le decay sont des hypers-paramètres qu'il convient de régler pour permettre une convergence plus rapide. Le terme de biais est mis à jour de la même façon. Comme discuté précédemment le terme $\nabla w_{ij}^{(n)}$ va dépendre des couches précédentes.

L'avantage de la descente de gradient stochastique est double, tout d'abord c'est beaucoup moins coûteux en temps de calcul et en mémoire (apprentissage en ligne). De plus, le côté stochastique va aussi permettre de se sortir de minima locaux. Notre problème à optimiser est très complexe et comporte de nombreux minima locaux dans lesquels on ne veut pas tomber.

Un problème majeur de la backpropagation est l'évanouissement du gradient. Vu que la résolution du problème d'optimisation dépend de la résolution des couches précédentes, si le nombre de couche est trop important ou le learning rate (pas avec lequel on avance à chaque étape de la descente de gradient) est trop faible alors pour les couches au sommet le gradient a une norme très petite et les poids ne bougent presque pas. L'apprentissage se fait au mieux très lentement ou reste bloqué dans des minima locaux.

Chapitre 2

A Deep Learning Approach to Universal Image Manipulation Detection Using a New Convolution Layer

L'article a été publié en 2016. Il a été rédigé par Belhassen Bayar et Matthew C.Sstamm de l'université Drexler à Philadelphie aux États-Unis. A l'heure actuelle il a inspiré deux autres travaux. Ce papier a depuis évolué (notamment concernant l'architecture et le solveur) pour être présenté à la conférence ICASSP 2017 (International Conference on Speech and Signal Processing) organisé par l'IEEE.

2.1 Architecture et solveur

AlexNet est un des réseaux de référence sur ImageNet, c'est aussi l'un des premiers. Il se trouve que son architecture semble fonctionner de manière convenable pour d'autres problèmes de classification qu'ImageNet. C'est sans doute pour cela que les auteurs ont choisi de la reprendre. A ce réseau ils ont ajouté en entrée une couche de convolution un peu particulière puisqu'elle correspond à un filtre dérivateur. (5×5 avec un -1 au milieu et normalisé autour). L'idée est d'également « apprendre » ce filtre c'est pourquoi il est intégré dans le réseau et pas placé en amont. L'envie de dériver notre entrée est tout à fait légitime puisque l'on veut s'intéresser aux termes de second ordre de l'image. D'ailleurs on verra par la suite l'importance de cette couche dans le réseau. Le solveur choisi avec le réseau est des plus basiques puisque c'est une descente de gradient stochastique (SGD) avec pas constant. De plus le pas est très très fin (10^{-6}). D'après les auteurs c'est pour apprendre convenablement les poids de la première couche de dérivation.

Le réseau possède 8 couches : la couche de convolution pour les résidus, 2 convolutions, 2 max-polling et 3 couches denses (dont celle de sortie). Avec seulement 2 couches de convolutions on ne parle pas forcément de « deep learning » car le réseau reste peu profond.

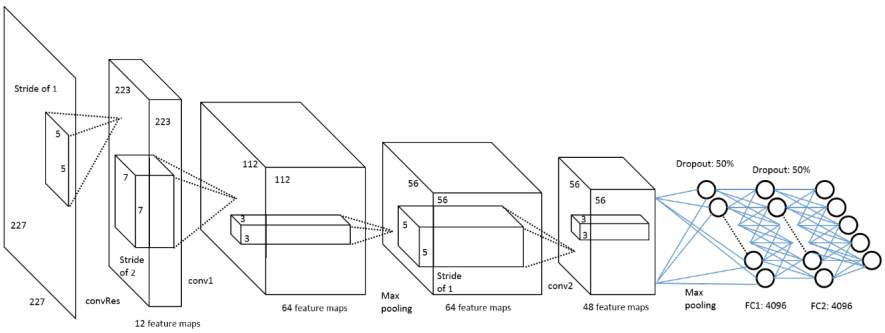

FIGURE 2.1 – Architecture du réseau CNN Architecture [1]

Les fonctions d'activations sont des ReLU (rectifier linear unit) dont la valeur vaut $f(x) = max(0, x)$. C'est une opération non-linéaire. La première couche n'est pas suivie par une fonction d'activation. Ils justifient cette absence de fonction d'activation par le pré-sentiment que l'opérateur non linéaire derrière détruirait une partie des descripteurs. La dernière couche est suivie classiquement d'une fonction soft-max ou exponentielle normalisée et non ReLU afin d'obtenir en sortie une probabilité d'appartenance à chaque classe.
Dans les deux couches denses (à la fin), nous avons un « dropout » de 50%. Cela signifie que 50% des neurones de la couche (parmi les 4096) sont tirés selon une loi uniforme à chaque passage de batch et mis à 0, donc non pris en compte pour le calcul à la couche suivante. C'est une forme de régularisation. Cette technique peut sembler surprenante vu que l'on tire aléatoirement mais elle s'avère très efficace en terme de vitesse de convergence et précision atteinte. C'est une technique tout à fait classique actuellement.

Concernant les paramètres expérimentaux, le momentum choisi était de 0.9 avec un decay de 0.0005 et un learning rate fixe de $\epsilon = 10^{-6}$ et 32 images par batch.

2.2 Une convolution particulière

L'idée de cette couche provient de Chen et al. [4] sur la détection de modification par filtre médian. Ils ont observé qu'un CNN n'est pas capable de

classifier des images manipulées si l'on prend comme entrée des images brutes. Ils avaient donc procédé en premier lieu à une extraction d'un grand nombre de descripteurs classiques pour des résidus de filtre médian servant d'entrée à un CNN.

On cherche ici un outil "universel", il faut donc s'affranchir de cette extraction de caractéristiques préalables qui reste spécifique. Cela est fait grâce à une première couche de convolution dérivatrice, qui joue un rôle d'extraction de descripteurs. Elle est apprise en même temps que le reste du réseau. Cette dérivation va permettre de faire ressortir les relations entre voisins. C'est cette information de base qui va permettre aux couches de convolutions de créer des attributs pour les manipulations d'image.

Pour cela la première couche est contrainte durant l'apprentissage pour avoir la forme d'un filtre de prédiction d'erreur. C'est à dire de manière explicite :

$$\begin{cases} \mathbf{w}_{0,0}^{(1)} = -1 \\ \sum_{l,m \neq 0} \mathbf{w}_{(l,m)}^{(1)} = 1 \end{cases}$$

avec les $\mathbf{w}$ les poids de chaque filtre et (l,m) les indices respectivement de ligne et colonne. Chaque filtre est initialisé aléatoirement puis contraint. La contrainte est ré-appliquée après chaque itération sur le réseau qui met à jour les poids, c'est à dire la back-propagation et descente de gradient.

Voici un exemple sur une image très simple et de même taille que le kernel, de l'action d'un tel filtre :

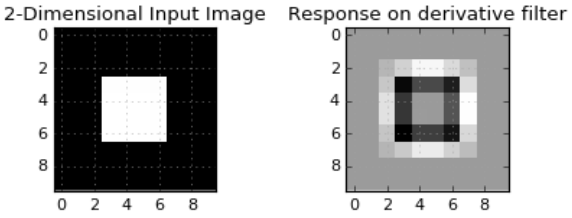

FIGURE 2.2 – Exemple de convolution avec noyau dérivateur

2.3 Résultats obtenus

2.3.1 Classification binaire

Dans ces travaux, les images proviennent de 12 appareils photos différents dont seulement le canal vert a été conservé afin d'avoir des images en niveau de gris. Puis ces images ont été cropé en bloc de 256x256 afin de créer un set de 261 800 images non altérées. Puis ces images ont été altérées par quatre approches différentes :

— Filtre médian avec des noyaux 5 x 5
— Flou gaussien avec des noyaux 5 x 5 et une variance de $\sigma = 1.1$
— Bruit blanc gaussien additif de variance $\sigma = 1.1$(AWGN)
— Redimensionnement en utilisant une interpolation bilinéaire et un facteur de redimensionnement de 1.5

Ce sont finalement 333 200 images altérées qui ont été obtenues.

Pour la classification binaire ce sont 43 500 images non altérées ainsi que leur équivalent altérées qui ont été choisies (apprentissage par pair). De même pour le jeux de test avec 16 000 images de chaque et par pair. Il y a donc au total 87 000 images d'entraînement et 32 000 de test.
Pour la classification binaire, chaque problème a été traité séparément avec à chaque fois l'entraînement d'un nouveau réseau.

Les résultats du test ont été obtenus en une seule fois (pas de cross-validation), ce qui explicite l'absence d'un intervalle de confiance empirique. On peut penser que les auteurs ont considéré que 32 000 images est un échantillon suffisamment large pour se passer de cross validation. Pour la prédiction ils ont choisi de considérer le neurone de la couche de sortie avec l'activation la plus grande.

	Filtrage médian	Bruit gaussien	AWGN, $\sigma = 2$	Re-sampling
Précision	99,31%	99,32%	99,68%	99,40%

TABLE 2.1 – Classification binaire[1]

Les résultats sont plus que très bons, même très étonnants : 99% de bonne classification c'est énorme. On peut penser à quelques explications :
— l'apprentissage par pair qui améliore les résultats le plus souvent
— les images sont cropées dans des images plus grandes, il se peut donc que certaines contiennent très peu de variations à la base et donc l'ajout de variations se remarque bien
— les transformations sont assez grossières (AWGN avec une variance de 2 par exemple) ce qui rend le problème facile.

2.3.2 Classification multi-classe

Concernant l'approche multi-classe, le même protocole a été suivie mais avec cette fois pour l'entraînement 17 400 blocs et leurs homologues pour chaque classe et 6 400 pour le test. Ce qui fait encore 87 000 images d'entraînement et 32 000 de test. On peut se demander ici pourquoi encore une fois n'avoir pas utilisé l'intégralité des images à disposition. C'est sans doute car un nombre aussi conséquent d'images n'est pas nécessaire.

Le réseau a été entraîné sur 56 000 itérations avec un pas constant puis pour les 9000 dernières les poids des convolutions étaient fixés pour entraîné seulement la couche dense. C'est à dire que dans un premier temps on entraîne l'extraction et la construction de caractéristiques adaptées ainsi que le classifieur

puis seulement le classifieur dans un second temps. D'après les auteurs, cela a permis de gagner environ 1% de précision.

	Original	Filtre médian	Bruit gaussien	AWGN, $\sigma = 2$	Re-sampling
Original	**98.40%**	0.52%	0.29%	0.34%	0.44%
Filtre médian	0.23%	**98.27%**	1.24%	0.12%	0.12%
Bruit gaussien	0.00%	0.18%	**99.75%**	0.00%	0.06%
AWGN, $\sigma = 2$	0.03%	0.04%	0.14%	**99.77%**	0.00%
Re-sampling	0.27%	0.20%	0.15%	0.00%	**99.35%**

TABLE 2.2 – Matrice de confusion pour la précision de classification en multi-classes[1]

Les résultats sont ici aussi plus que excellents alors même que la base d'apprentissage ne semble pas si large que ça (à peine 100 000 images). Il n'y a toujours pas une classe qui ressort comme plus difficile à classer que les autres. Le réseau a bien su s'adapter aux différentes modifications proposées.

L'autre idée intéressante est que ce réseau a su faire de l'extraction de caractéristiques sans intervention humaine. L'apprentissage est supervisé mais la première couche qui extrait des caractéristiques n'est pas définie par l'expérimentateur. Cela veut dire que ce CNN peut aussi apprendre sur de nouvelles manipulations d'images que l'on ne connaît pas ou dont on sait pas encore quels sont les meilleurs descripteurs.

Chapitre 3

Implémentation

3.1 Génération des images

Les images proviennent de Flickr et sont en nuance de gris et 256x256. Ce sont des images entières et redimensionnées contrairement au papier où les images étaient croppées dans des images plus grandes. Ce cas est probablement légèrement moins favorable puisque potentiellement les images contiennent plus de variations (nous n'avons pas d'images de ciel bleu uniforme par exemple).

D'après le papier il n'y a pas une manipulation plus difficile à détecter que les autres et l'idée étant de tester des sensibilités, ce n'était pas le plus intéressant de se lancer dans le multi-classe. Nous avons donc choisit une manipulation et le choix s'est porté sur l'ajout d'un bruit blanc gaussien centré parce que c'est facilement implémentable. Nous avons tout d'abord généré des images avec un $\sigma = 2$ puis $\sigma = 0.5$ afin de se placer dans un cas moins favorable. Les 400 000 images ont été altérées ce qui nous fait finalement une base très large de 800 000 images.

Pour des raisons de performances et d'accès mémoire, ces images ont été placé dans une base LMDB. Tout d'abord en intégralité, puis seulement 100 000 quand nous nous sommes rendus compte qu'il n'est pas nécessaire d'avoir autant d'images pour l'entraînement.

3.2 Caffe

Le choix du framework pour les CNN s'est porté sur Caffe puisque c'était celui utilisé dans le papier. C'est un framework plutôt à destination des académiques développé par une équipe de Berkeley. Il est codé à la base en C++ mais possède aussi un interface Python nommée PyCaffe que nous avons utilisé. C'est un framework qui se situe entre le haut et le très haut niveau. Le bas niveau étant un langage qui interagit directement avec le GPU (type Théano) et le haut niveau un langage s'affranchissant de ces contraintes techniques pour

ne s'intéresser qu'au modèle. On peut aussi noter qu'il y a une interface Matlab également. Il permet aussi le calcul sur GPU, ce qui est primordial. Il n'aurait pas été réaliste de vouloir entraîner notre modèle sur des CPU pour des raisons de temps. Caffe propose aussi une intégration CUDA et CUDNN ce qui permet des performances sur GPU encore meilleures. Caffe peut gérer jusqu'à 4 GPU.
L'installation est un peu complexe à faire et demande d'y passer un peu de temps pour bien installer toutes les bonnes dépendances et avoir un Makefile correct. Nous l'avons installé sur un ordinateur portable classique (Nvidia 940mx), un serveur MAC (Nvidia 980) et un cluster de calcul hautes performances, celui de Lille 1 (deux Nvidia K80). Malheureusement Caffe n'est pas fait pour être installé sur un cluster. L'installation a été très complexe, la compilation a du se faire à la main. Même avec beaucoup d'efforts et d'expertise de la part de Cyrille Toulet, il n'a pas été possible de faire fonctionner Caffe sur le cluster. L'installation devait présenter un problème puisqu'il y avait un problème de connexion entre la base LMDB et Caffe.

On trouve de nombreux exemples et autres tutoriels sur le GitHub du projet pour s'initier à l'interface PyCaffe. Néanmoins c'est un framework un peu complexe et qui demande une bonne connaissance de Python. Nous avons également trouvé qu'il manque un peu de documentation sur les différentes couches existantes avec PyCaffe. Cette documentation existe pour des développement sans interface mais pas avec l'interface.

En pratique l'interface Python ou Matlab de Caffe permet de générer des fichier .prototxt pour le réseau d'entraînement, de test et le solveur. On définit deux réseaux, un de train et un de test car il est possible de demander une sortie autre que celle du réseau de train pour le réseau de test. On peut trouver cette situation lorsque que l'on souhaiterait remplacer le classifieur dense en sortie par un SVM [7]. Ces fichiers décrivent entièrement notre réseau. Celui du solveur est dépendant de ceux du réseau d'entraînement et de test. Ces fichiers sont en réalité des protocol buffers (lien entre Caffe et l'ordinateur pour communiquer).

Nous avons aussi utilisé au cours d'autres projets d'autres frameworks, nous avons donc pensé qu'il peut être intéressant de les comparer ici à Caffe.
Une expérience avait été tenté avec Theano qui est lui une framework très bas niveau de calcul GPU. Il en ressort que pour le développement de réseau Caffe est nettement plus judicieux car Theano est très complexe à utiliser.
Nous avons aussi pu utiliser Keras ailleurs. C'est un framework très haut niveau pour Python qui fait maintenant partie intégrante du développement de Tensorflow (Google) mais qui peut s'utiliser aussi avec Theano. Il demande donc forcément un moteur de calcul GPU (Tensorflow ou Theano) mais il permet très facilement de réutiliser des modèles pré-entraînés ou de construire et entraîner des modèles simples (comme celui que nous avions ici). Son utilisation pourrait être une piste.
Enfin pour terminer la liste des framework, nous citerons également Torch (Facebook) qui a connu un regain d'intérêt récemment avec la sortie d'une interface

en Python alors qu'il fallait précédemment utiliser un langage propre : le lua. Toujours dans les framework Python il y a également Lasagne (qui demande une sous-couche Théano) mais que nous n'avons jamais réellement testé.
Il existe également un certain nombre de framework Java tel que Deep4J.

3.3 Sensibilités

Nos résultats ont été obtenu avec des batch de 16 images, notre GPU ne possède pas suffisamment de mémoire pour faire passer des batchs de 32 images. Cela ne devrait normalement pas modifier les résultats. Afin d'être certain de converger, nous attendions 7000 itérations (donc 112 000 images pour l'entraînement). Un test de précision et de cross-entropie est effectué toutes les 28 itérations sur 100 batchs (1600 images) afin de suivre les performances. De plus la convergence étant atteinte, normalement, bien avant les 7000 itérations, cette mesure nous a aussi servi pour connaître la performance de notre classifieur.

Pour former notre base d'apprentissage nous avons tiré aléatoirement 70% du total d'image (560 000 images) et le reste pour le test (240 000). Le grand volume d'image nous garantit d'avoir des échantillons dans lesquels les classes sont équilibrées. Nous utilisions en réalité quel les 112 000 premières images de notre base d'entraînement. L'apprentissage ne s'est pas fait par paire puisque nous avons tiré uniformément dans les 800 000 images.

L'apprentissage prenait en moyenne 6h.

3.3.1 Convolution particulière

Une des premières choses que nous avons essayé est bien sûr de ne pas contraindre la première couche et donc de disposer d'un réseau tout à fait classique. La précision de classification ne dépasse alors pas les **50%**. Les modifications sont cachées dans l'image brute et il faut une première étape d'extraction pour qu'elles deviennent atteignables pour le réseau.

Par ailleurs la valeur -1, au centre du filtre semble importante aussi. Si on la remplace par un chiffre positif plus rien ne marche comme on peut s'y attendre (on a alors plus du tout un filtre dérivateur).

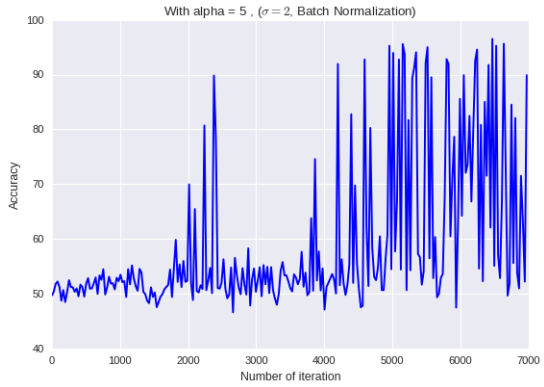

FIGURE 3.1 – Influence du α

3.3.2 Learning rate et solveur

Une première surprise est qu'en prenant un learning rate fixe de 10^{-6}comme préconisé dans le papier, le réseau ne dépasse pas les **50%** de bonnes classifications. La convergence ne se fait pas. En prenant un learning rate de 10^{-4} nous avons cette fois convergence et un taux de bonne classification de l'ordre de **95%** (sur plusieurs entraînements). Les pas de la descente de gradient doivent être trop petits dans le premier cas.

En changeant le solveur pour un solveur plus adaptatif (type Adam [3]), je ne constate pas d'amélioration significative des résultats (95% aussi), sachant que nous n'avons en plus qu'un estimateur empirique de l'erreur et qu'il faut considérer un intervalle de confiance. Ce résultat semble normal car l'objectif d'un tel solveur est la vitesse de convergence. Cependant, la convergence n'est pas plus rapide ce qui est surprenant au premiers abords. Au vu des résultats atteints (de l'ordre de 95%) on peut penser que le problème est déjà simple à la base et que donc un solveur adaptatif ne peut pas jouer pleinement son rôle de simplification. C'est pourquoi nous n'en avons pas testé d'autres.
A titre d'exemple, Adam signifie Adaptive Moment Estimation. Le principe est d'avoir un learning rate adapté à chaque paramètre de notre réseau qui suit aussi un décroissance exponentielle.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t \nabla \theta_t$$

Avec θ_{t+1}la mise à jour de notre paramètre,$\hat{v_t}$ un estimateur de la moyenne au carré du gradient au étapes précédentes, $\hat{m_t}$ un estimateur de la moyenne du gradient aux étapes précédentes, η le learning rate fixé au départ et enfin ϵ notre paramètre de « decay » (fixé le plus souvent à 10^{-8}).
Le propos n'étant pas ici de détailler les différents types de solveur existants, nous invitons le lecteur à visiter : An overview of gradient descent optimization algorithms [6].

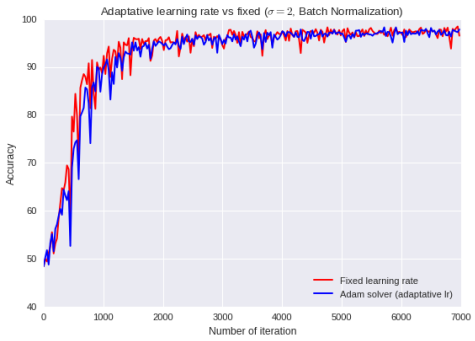

FIGURE 3.2 – Influence du solveur

3.3.3 Batch Normalization (BN)

Chaque « batch » de données (16 images dans notre cas), va être centré et réduit, avant le passage par la couche d'activation. Par cette technique on cherche à réduire deux problèmes. Tout d'abord le « covariance shift », c'est à dire les différences de distribution dans le dataset. Dans notre cas c'est l'utilisation de différents appareils photo ce qui est plutôt général comme problème. L'autre problème est ce que Sergey Ioffe appelait dans son papier « internal convariance shift » [5]. Grossièrement avec la backpropagation, la mise à jour d'une couche va influencer la distribution des poids de la couche précédente, ce qui n'a pas forcément de sens. On aimerait que chaque couche soit optimisée indépendamment, ainsi une erreur à niveau ne pénalise pas le tout. Avec la Batch Normalization avant de faire ce passage par la convolution, le batch est centré, réduit ce qui permet d'éviter ce problème.
Plus le nombre de couches augmente, plus « l'internal covariance shift » devient important. Ici nous avons seulement deux couches de convolution ce qui ne le rend pas particulièrement important, mais ajouté à la covariance shift d'ajouter des Batch Normalization prend tout à fait sens ici.

En pratique, l'ajout de BN a permis de passer de **95%** de bonne classification à **97%** en moyennant sur 3 runs. Notre variance empirique pour la précision étant inférieur à 1% (ce qui n'est pas choquant vu le grand nombre d'images de test), nous pouvons dire qu'il y a un réel intérêt à ajouter cette couche.

3.3.4 Variance du bruit

Pour tester les limites de notre réseau, nous avons re-généré un autre jeu d'image avec cette fois une variance de $\sigma = 0.5$ pour le bruit blanc gaussien ajouté. Cela n'a pas semblé déranger notre modèle puisque nous avons toujours des résultats de l'ordre de **97%**. Il pourrait être intéressant de voir jusqu'à quelle

valeur de σ la précision reste stable. Néanmoins chaque modèle prend approximativement 6h à entraîner donc nous avons préféré nous concentrer sur d'autres variations.

FIGURE 3.3 – Influence de la valeur de σ

3.3.5 Images de steganalyse

Nous avons également tenté de remplacer nos images en entrée par des images provenant du domaine de la steganalyse. Les images « cover » sont les images de base et les images « stégo » ont subit une substitution LSB à 0.1 bit par pixel (on a changé le bit de poids faible). Malheureusement notre réseau ne fonctionne pas du tout sur ces images là. Nous ne parvenons pas à dépasser les 50% de bonne classification, même en tentant plusieurs modifications de learning rate et autres paramètres du solveur.
Cela peut s'expliquer par le fait qu'un substitution LSB à 0.1 bit par pixel modifie seulement 5% des pixels donc la variance du bruit d'insertion est de 0.05. Précédemment notre variance la plus faible était de 0.5 soit dix fois plus grande. Les modifications doivent être trop fines pour être détectables.

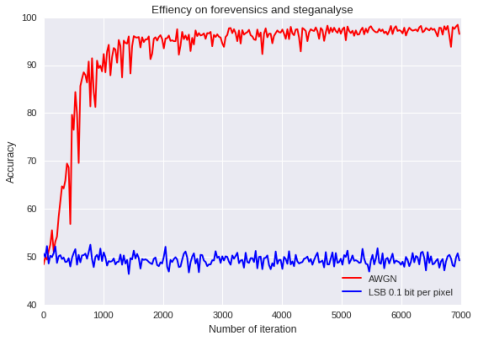

FIGURE 3.4 – Performance sur des images de steganalyse

3.3.6 Fonctions d'activation

3.3.6.1 Convolution particulière

Une couche de convolution est habituellement suivie d'une couche d'activation, le plus souvent non-linéaire. Hors-ici il n'y en a pas après la convolution particulaire. Les auteurs le justifie en disant que cette couche donne en sortie des features qui contiennent une information sur l'altération des images, ce qui pourrait être détruit par une non-linéarité.

En pratique, nous avons essayé de rajouter une couche de convolution classique en entrée et effectivement le réseau ne converge pas. Les résultats oscillent continuellement même après un très grand nombre d'itérations.

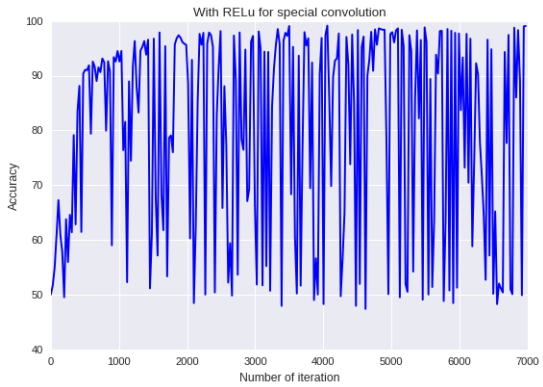

FIGURE 3.5 – Activation de la première couche

Cela semble logique car le but de la première couche n'est pas de commencer à faire de l'extraction de caractéristiques mais plutôt de fournir une image non

brute en entrée. L'intérêt de la mettre dans le réseau est que cette couche est également apprise et non réglée manuellement. Elle reste donc très différente des autres.

3.3.6.2 Autres convolutions

Dans une version mise à jour du papier, les auteurs proposent non seulement une nouvelle architecture (avec deux couches de convolution supplémentaire et de la Batch Normalization) mais aussi de changer les fonctions d'activations par une autre fonction non-linéaire : TanH. Nous avons donc testé également ce changement mais tout en gardant l'architecture de départ.
Les résultats ont été concluants puisque nous obtenons également autour de **97%** de bonne classification.

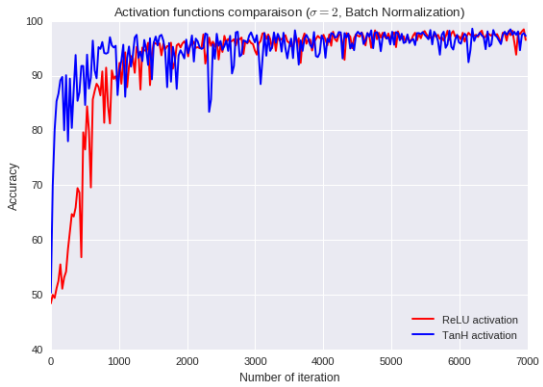

FIGURE 3.6 – Comparaison de tanH et ReLU

Sur deux runs, j'ai pu constaté qu'avec des fonctions d'activation tanH, la précision augmente plus rapidement qu'avec ReLU. Cependant ReLU varie moins autour de sa valeur finale et se stabilise plus vite.
L'échange entre tanH et ReLU semble donc, ici, pas apporter grand chose que ce soit en terme de vitesse de convergence ou précision atteinte.

Chapitre 4

Conclusions et perspectives

Nous avons réussi à reproduire des résultats très proches de ceux du papier dans le cas de la classification binaire pour des images manipulées avec un bruit blanc gaussien additif. Nous obtenons 97% de bonne classification quand ils en obtenaient 99%. Cette différence s'explique sans doute par la différence des images utilisées. Nos résultats semblent donc confirmer l'intérêt des CNN associés à un filtre dérivateur appris avec le réseau pour faire un détecteur universel de manipulation d'image.

Nous avons tout comme les auteurs, constaté l'importance de la première couche dérivatrice (activation et contrainte). Nous avons pu en plus proposer quelques évolutions à l'architecture réseau : ajout de Batch Normalization, changement du learning rate et de fonctions d'activation.
Il pourrait être aussi intéressant, en partant de ce réseau, de ré-implémenter toutes les évolutions que les auteurs proposent dans la version mise à jour de leur papier (pas encore publié et seulement en peer-review à l'heure actuelle) pour voir si elles apportent réellement un gain de performance ou si la première architecture basique qu'ils ont proposé est déjà suffisante.
L'ajout d'une couche de convolution supplémentaire aurait aussi pu être intéressante mais nous ne possédions pas l'expertise pour dimensionner les noyaux de convolutions et de pooling. Il aurait donc fallu procéder par GridSearch ce qui aurait pris beaucoup de temps puisque le réseau met environ 6h à s'entraîner, pour un intérêt limité.
Enfin un dernier point intéressant serait d'arriver à sauvegarder les poids des convolutions une fois le réseau entraîné et de remplacer la couche dense en sortie (après la couche de mise à plat) par un autre classifieur qu'un réseau de neurones. Typiquement ce sont des SVM qui sont utilisés dans ces couches et permettent de gagner un petit peu en précision.

Enfin concernant le côté technique, nous avons pu constaté que bien que les framework de deep learning se sont largement démocratisés ces dernières années, ils restent complexes à installer (surtout sur un cluster) et à utiliser. Un

CNN est plus difficile à implémenter qu'un SVM avec Scikit-Learn. Il y a cependant des initiatives dans ce sens avec notamment Keras qui cherche à devenir l'équivalent de Scikit-Learn pour le deep learning. Il ne faut donc pas négliger le côté technique avant de se lancer dans un projet deep learning.

Bibliographie

[1] Belhassen Bayar and Matthew C. Stamm. A deep learning approach to universal image manipulation detection using a new convolution layer.

[2] Colah. Colah's blog. http ://colah.github.io/posts/2014-07-Understanding-Convolutions/, 2014.

[3] Jimmy Ba Diederik P. Kingma. Adam : A method for stochastic optimization. *ICLR*.

[4] Bhiksha Raj Haohan Wang. On the origin of deep learning. *arXiv*.

[5] Sergey Ioffe and Christian Szegedy. Batch normalization : Accelerating deep network training by reducing internal covariate shift. *Arxiv*.

[6] Sebastian Ruder. An overview of gradient descent optimization algorithms an overview of gradient descent optimization algorithms. http ://sebastianruder.com/optimizing-gradient-descent, 2016.

[7] Yichuan Tang. Deep learning using linear support vector machines. *ICML workshop*.