Le génie pour l'industrie

Réseaux de neurones et systèmes flous SYS843

Rapport 2 : Étude expérimentale

Titre du projet :

Comparaison de réseaux à convolution pour la reconnaissance automatique des panneaux dans le cadre de véhicules autonomes.

Rémy CARPENTIER

École de Technologie Supérieure

Montréal, CANADA

22 avril 2017

Table des matières

Table des figures

Table des tableaux

1. Mise en situation

1.1. Domaine d'application

L'erreur humaine représente 90 % des accidents de la route. Malgré l'amélioration des voitures en termes de sécurité et les nombreuses campagnes de sensibilisations, la route fait encore plus de 2000 décès par an et 13 000 blessés [1]. Si l'on créer une voiture qui se déplace sans actions humaines, on pourra sauver de nombreuses vies sur les routes. On ouvrira la porte à de nombreuses idées futuristes comme se faire déposer par sa voiture au travail puis la laisser aller se garer seule dans parking loin du centre-ville en toute sécurité.

Depuis les années 1970, des chercheurs et des scientifiques se sont donné un objectif : créer une voiture capable de rouler dans le trafic sans aucune aide humaine. Depuis les années 1980, plusieurs prototypes ont vu le jour. L'autonomie, la vitesse et l'environnement ne font que de se rapprocher des situations réelles. Au début, les véhicules se déplaçaient en suivant une ligne au sol puis sur une route sécurisée et sans trafic. On a ensuite complexifié l'environnement pour créer des véhicules de type tout-terrain autonomes. Dans les années 1990, on commence à rajouter des fonctionnements autonomes à des voitures classiques, comme le dépassement et le stationnement dans des situations réelles. Aujourd'hui, l'ensemble des constructeurs automobiles ainsi que d'autres entreprises liées à l'informatique comme Google, continuent d'améliorer leurs prototypes.

Comment remplacer l'homme pour une tâche si complexe qu'est la conduite ? Lorsque l'on conduit, on fait appel à trois grands axes, la vision, l'analyse et le mouvement. Trois axes simples pour le cerveau humain, mais compliquer à reproduire à l'aide d'un ordinateur, notamment dû au fait de la multitude des situations sur la route. L'ordinateur doit être capable de prendre en compte tous les obstacles qui l'entourent et la trajectoire de la route.

Après avoir vu, compris et analysé la scène, l'ordinateur doit savoir comment conduire la voiture en respectant le Code de la route ou d'autres règles comme suivre un itinéraire.

La majorité des prototypes se concentre sur la trajectoire de la voiture et sur son interaction avec les autres véhicules, piétons, etc. Dans notre étude, nous allons étudier un aspect moins mis en avant, les panneaux de signalisation (« Traffic Signs Recognition », TSR). Ils jouent un rôle essentiel pour la sécurité. Ils informent les automobilistes des directions à prendre, de différents dangers de la route ou encore des conditions de circulation à respecter. Un véhicule autonome doit voir ses informations, les reconnaitre et les comprendre. Sans quoi, il serait dangereux pour ses passagers et pour les autres utilisateurs de la route.

1.2. Problématique

Même si dans le futur, les voitures seront surement interconnectées, géolocalisées, et autres, nous n'aurons surement plus besoin de panneaux, de volants et autres idées de sciences fiction. On peut déjà le remarquer avec l'utilisation des GPS, qui rendent les panneaux de directions obsolètes. Mais il n'est pas encore question de les supprimer étant donné que nous sommes dans une phase de transition. Il faut penser à créer des voitures autonomes s'intégrant facilement à l'environnement routier actuel.

Nous savons qu'il existe déjà de tels procédés sur des voitures haut de gamme. Une caméra filme la route et repère les panneaux pour ensuite l'afficher au conducteur. C'est fonctionnel, mais simplement indicatif pour le conducteur. Ce que nous cherchons à faire est de reproduire ce procédé avec différents réseaux de neurones et de comparer leurs performances. Nous avons déjà des idées des performances de chacun, étant donné qu'il existe des tests de performance, benchmarks, sur ce type de problème.

Nous nous sommes interrogées sur le point suivant : Comment varie les performances et le temps de calcul en fonction de la complexité d'un réseau de neurones pour identifier rapidement et de manière sûre le type de panneau de nous avons face à nous ?

1.3. Objectifs du projet

Nous avons un problème de type classification multiclasses, 43 panneaux différents. Il existe de nombreux algorithmes pour résoudre ce type de problème. Nous avions initialement choisi de prendre trois réseaux de neurones différents : un « Multi Layer perceptron » (MLP), un « Support Vector Machine » (SVM) et un réseau à convolution (CNN). L'objectif suite à l'étude de l'art était de comparer les trois réseaux indépendants. Durant la réalisation du projet, et grâce à l'aide du professeur, on a réduit ce nombre à deux réseaux à convolution avec comme classificateur en sortie soit un MLP, soit un SVM.

Le choix de ces trois modèles repose sur les raisons suivantes : le CNN est un excellent extracteur de caractéristiques pour les images et les MLP et SVM sont tous les deux de bons classificateurs pour les images. Le MLP est un modèle simple vu en classe, simple, mais ancien, ayant de bonnes performances [2]. Le SVM est aussi un modèle simple offrant un bon rapport complexité/résultats [3] [4] [5]. Le CNN est plus complexe, mais beaucoup plus performant. On retrouve le CNN en tête de plusieurs benchmarks comme le TSR, le MNIST et bien d'autres [6] [7] [8].

Le but de notre projet est réalisé ces réseaux en python, de les optimiser le plus possible afin de comparer leurs performances. Les performances évaluées seront le temps, la complexité et la précision sur la même base de données : des images présegmentées de panneaux issus de la base de données GTSRB [6] utilisée pour faire des benchmarks. Nous ne

cherchons pas à obtenir de meilleurs résultats que ceux obtenus dans les articles, mais de bien comprendre le fonctionnement de ses réseaux de neurones et de les programmé nous-mêmes.

Figure 1 : Exemples d'images de la base de données GTSRB

1.4. Structure du document

Le présent document sera structuré de la manière suivante, nous allons vous présenter une vue d'ensemble de notre cas d'étude, c'est-à-dire l'ensemble du processus d'acquisition et de reconnaissance du panneau. Le but étant de bien comprendre où se situe notre travail dans la globalité de cette thématique. Nous allons rapidement rappeler le fonctionnement des réseaux utilisés. Étant donné que nous souhaitons faire une comparaison entre deux types d'algorithmes pour bien comprendre leurs différences fondamentales. Il nous a semblé normal d'avoir exactement les mêmes données d'entrées et les mêmes évaluations pour ne pas biaiser les résultats de cette étude.

Ensuite, nous verrons en détail le cœur du travail réalisé durant ce projet. Dans cette partie, nous verrons les données utilisées, leurs provenances et les modifications que nous y avons apportées. Nous verrons ensuite le changement de vision que nous avons opéré au cours du projet et enfin, l'architecture des réseaux comparer ainsi que tous leurs paramètres. Pour finir cette partie, nous vous présenterons les critères de performances évalués et les différents tests que nous avons appliqués aux réseaux.

Enfin, les résultats obtenus à travers des différents tests seront présentés et nous discuterons ensuite sur les implications qu'entrainent ses résultats. Nos interprétations nous permettront de répondre à notre problématique initiale et d'ouvrir la voie à des travaux futurs pour améliorer ce projet.

2. Sommaire des techniques étudiées

2.1. Présentation du système global

Pour une meilleure compréhension de notre travail, nous allons vous présenter le système global dans lequel il s'inclut. Comme nous l'avons dit dans l'introduction, le thème étudié ici est la détection et la reconnaissance des panneaux de circulation de façon autonome. La structure de ce système peut se représenter sous la forme de la figure ci-dessous. En rouge, la partie que nous avons réalisée dans ce projet.

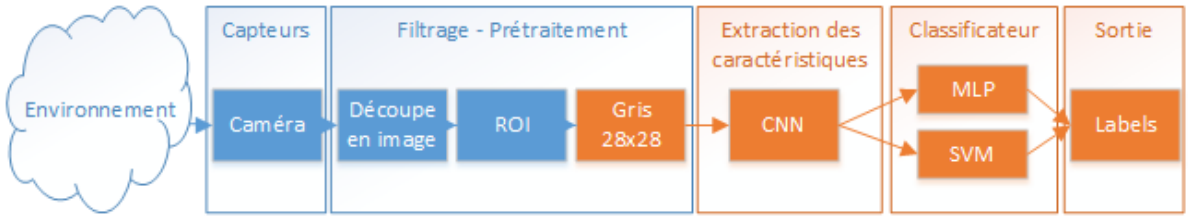


Figure 2 : Schéma du système global

Pour détailler ce schéma, nous avons en entrée, à gauche, l'environnement. Dans notre cas, cela représente concrètement tout ce qu'un automobiliste peut voir : la route, les autres véhicules, le paysage alentour et bien sûr les différents panneaux routiers. À partir de ce moment, il faut acquérir de l'information, capturer cet environnement sous forme informatique. Pour cela, on utilise une caméra fixée à l'avant du véhicule, qui va filmer ce que voit le conducteur.

Suite à cela, on va échantillonner la vidéo en un certain nombre d'images par seconde. Plus ce nombre est grand et plus notre système sera fluide pour le conducteur, mais demandera des ressources importantes. Une fois que ce découpage a été fait, il fait mettre en place un algorithme capable de détecter les panneaux dans l'image, capable de détecter les régions intéressantes (« Regions Of Interest », ROI). Il existe différentes méthodes pour ce

genre de problème, on peut vous faire remarquer que la première partie du benchmark [6] est consacrée à cette tâche. Les données d'entrées sont des images de ce que voit un automobiliste et en sortie, on doit obtenir une image centrée uniquement sur le ou les panneaux présents dans l'image de départ. Cette étape est cruciale puisque si l'algorithme ne cible pas bien les panneaux, la reconnaissance des ceux-ci sera alors très aléatoire. Enfin, on vient faire du prétraitement sur le ROI que nous avons obtenu. Cela permet d'avoir le même type de données mises dans le réseau. Dans notre étude, nous avons décidé de mettre les ROI en niveau de gris pour n'avoir qu'une valeur par pixel, contrairement au RGB. Nous avons aussi réduit la taille des ROI à 28x28 pixels pour éviter d'avoir de trop grandes images gourmandes en ressources.

La partie suivante est l'extraction des données. Dans cette partie, il faut extraire des images prétraitées les caractéristiques qui vont nous permettre ensuite de les comparer. Nous avons réalisé cette étape avec un réseau de neurones de type convolution, CNN.

Avec tous ces filtres obtenus, on va les catégoriser pour pouvoir reconnaitre de quel type de panneau il s'agit. Pour cela, on a utilisé deux réseaux de neurones différents. Premièrement, un MLP entièrement connecté et en deuxième, un SVM linéaire. Ces deux types de réseaux ont permis d'assigner une classe à chaque image. À partir de ce moment, il est facile de continuer le processus en affichant, par exemple, les panneaux que la voiture a vus au conducteur, de réguler la vitesse en fonction des panneaux détectés, etc.

2.2. Rappels des différents réseaux de neurones utilisés

Pour ce projet, nous avons utilisé trois types de réseaux : un CNN, un MLP et un SVM. Le premier sert à l'extraction des filtres et les deux derniers servent à la classification de ses

filtres selon les 43 classes prises en compte dans ce projet. Dans cette partie, nous allons voir en détail le fonctionnement de chacun de ses réseaux.

2.2.1. Réseau à convolution (CNN)

Les réseaux « Convolution Neural Network », ou CNN, utilisent le même fonctionnement neuronal que la vision animale et par conséquent, la vision humaine. Les premières études sur la vision chez les animaux débutent dans les années 1950 et en 1990, Yann LeCun informatise les résultats. Il met en place un algorithme semblable à la vision humaine, le premier réseau à convolution. Dans les années 2000, l'augmentation de la puissance de calcul informatique, notamment des GPU, « graphic processing unit », permet de réduire significativement les temps de calcul. Autre point important qui a favorisé le développement des CNN est le très grand nombre de données disponibles grâce aux big data. Ces deux facteurs ont fait des réseaux CNN, un outil très utilisé et très performant.

Pour avoir une idée générale du fonctionnement, l'humain va voir en premier des formes. Selon les contours et les caractéristiques de cette forme (couleurs, dimensions…), il est capable de retrouver dans sa mémoire à quoi cette forme ressemble et associe le nom que l'objet qui lui ressemble le plus. Le CNN se focalise sur la partie extraction des caractéristiques et c'est aux classificateurs de retrouver le groupe auquel appartient l'objet.

Le CNN est composé de trois grandes étapes : la convolution, le « pooling » et la normalisation. Première étape avant de faire la convolution, on doit définir des filtres simples, des carrées composés de quelques pixels noirs ou blanc ou en couleur selon les données en entrée. Vient ensuite l'étape de convolution [9] qui consiste à balayer notre image avec ces filtres. Pour chaque pixel balayé, on note sa correspondance avec le filtre. On obtient ainsi

une nouvelle image filtrée. On répète la convolution pour tous les filtres que nous avons. Ci-dessous les différents filtres extraits par le CNN.

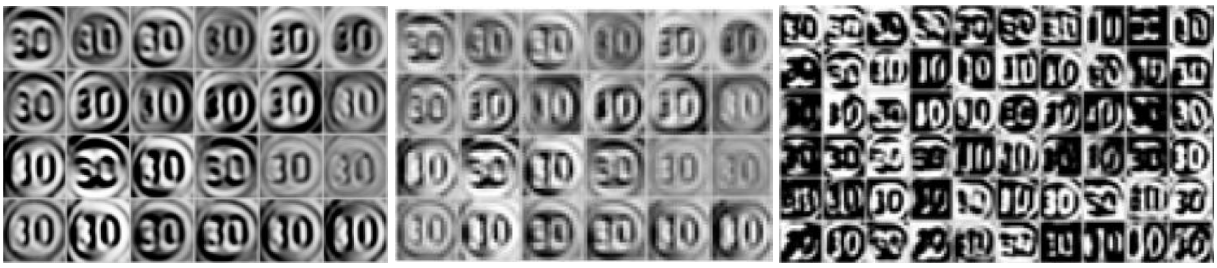


Figure 3 : Exemple de features à différentes profondeurs du CNN [8].

L'étape suivante est le « pooling », littéralement la mise en commun. Cela consiste à prendre une fenêtre de 2x2 pixels, généralement [7] [8], de l'image filtrée et de prendre la valeur maximale dans cette fenêtre et de la gardé en mémoire dans une nouvelle image. On vient balayer l'image filtrée [9]avec cette fenêtre. On peut choisir de faire du recouvrement ou non, c'est un hyper paramètre à fixer. Cette opération nous permet d'obtenir une image plus petite, mais conservant le maximum d'information de l'image initiale. On peut reproduire plusieurs fois ces deux étapes conjointement ou indépendamment.

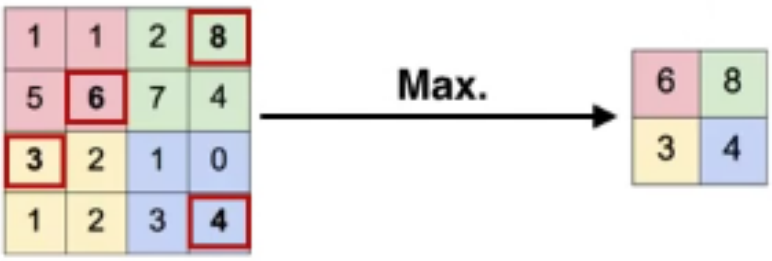


Figure 4 : Exemple de « max-pooling »

On peut aussi utiliser d'autres fonctions comme la normalisation des valeurs obtenues suite à la convolution. On peut appliquer une fonction de la forme « Rectified Linear Units », ReLU [9], qui fait comme une sorte de filtre pour toutes les valeurs négative en les fixant à zéro. Les valeurs positives restent inchangées. Pour terminer, on doit avoir obtenu un

ensemble de — neurones se résumant à un scalaire chacun. À partir de cette couche, on applique un MLP [7] [8] entièrement connecté pour classer nos images selon les différents types de panneaux.

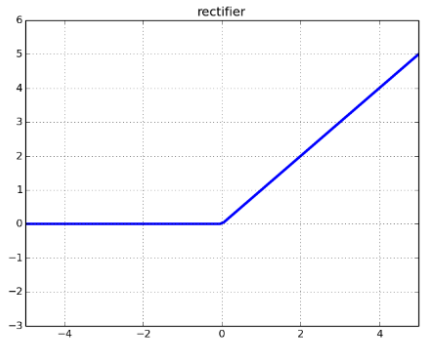


Figure 5 : Fonction d'activation ReLU

On peut répéter ces étapes autant de fois que l'on souhaite. On peut aussi mettre en parallèle des étapes. Cette organisation des étapes est appelée l'architecture du réseau. Elles peuvent être plus ou moins complexes selon le résultat recherché et les ressources mises à disposition. En exemple ci-dessous l'architecture de réseau CNN de Google, GoogLeNet. Pour la suite ce projet, nous avons choisi l'architecture LeNet-5 pour ça simplicité et VGG16 pour son aspect linéaire. Nous reviendrons sur ces réseaux dans la partie 4.

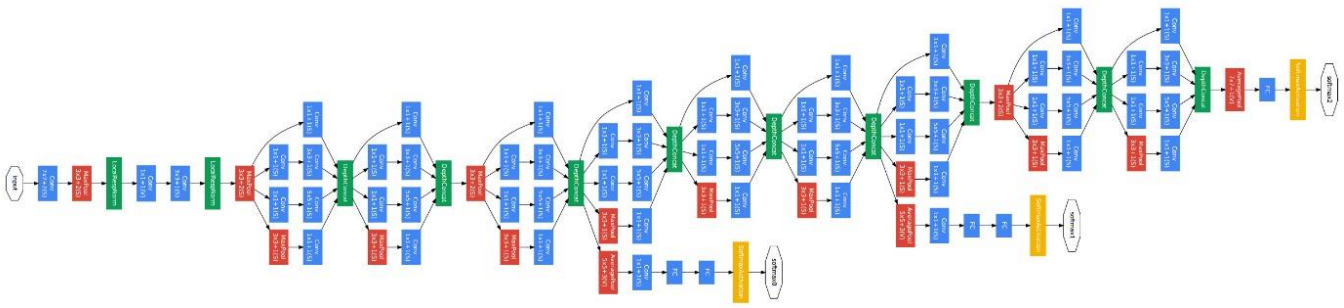


Figure 6 : Architecture GoogLeNet [10]

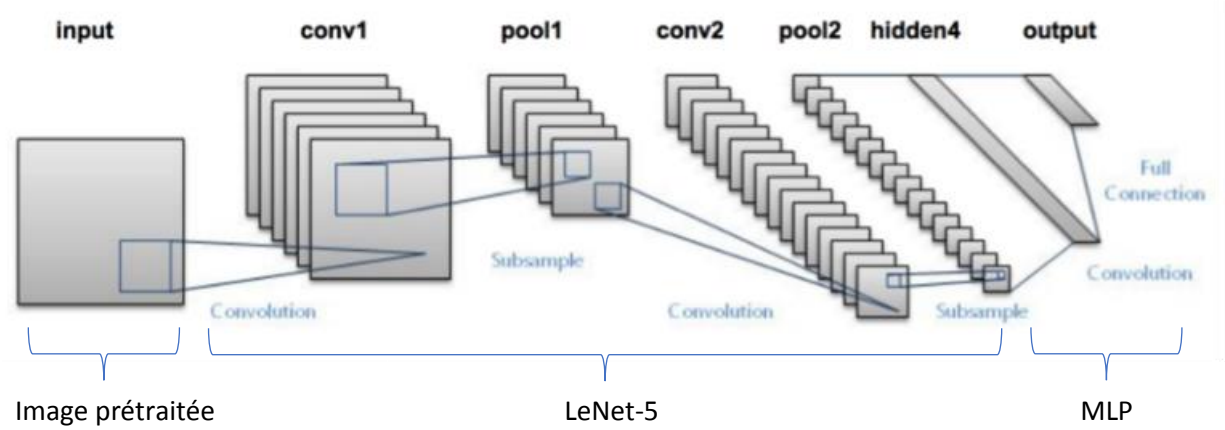


Figure 7 : Architecture LeNet-5 [8]

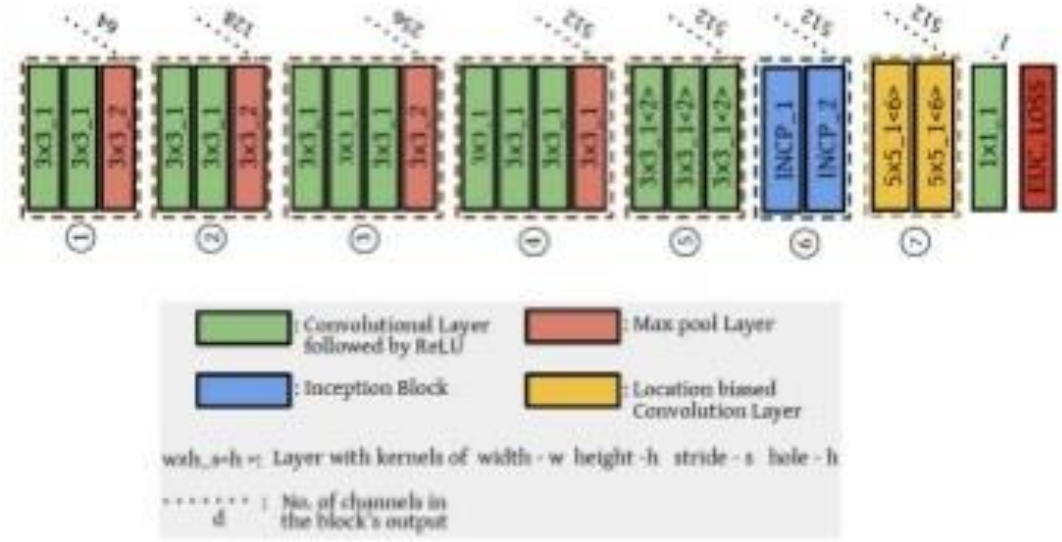


Figure 8 : Architecture VGG16 [11]

2.2.2. Réseau Multi Layer Perceptron (MLP)

Le réseau de neurones « Multi Layer Perceptron », ou MLP, est comme son nom l'indique, composé de plusieurs couches de perceptron. Un perceptron est le premier neurone informatique, inventé en 1957 par Franck Rosenblatt à Cornell, USA. Le MLP mis au point par David Rumelhart en 1986 est le premier réseau de neurones artificiel. L'architecture de ce type de réseau est uniquement composée de perceptrons disposés en plusieurs couches de profondeur, au minimum deux, comme on peut le voir sur la figure suivante.

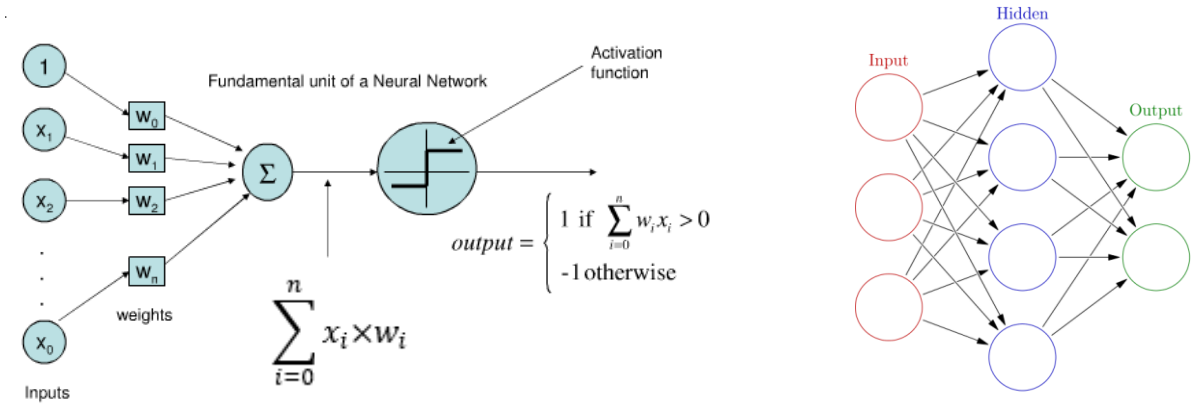
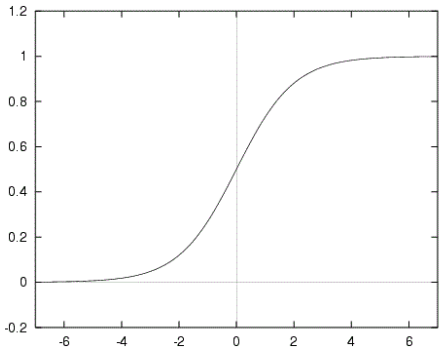


Figure 9 : Schéma d'un perceptron — Exemple d'un MLP

Chaque rond dans la figure de droite représente un perceptron, figure de gauche [12]. Un perceptron peut se décomposer en deux parties. La première qui va sommer tous les poids multipliés par la valeur du neurone précédent. Deuxièmement, une fonction d'activation dont va dépendre la valeur en sortie du perceptron. Cette fonction peut être un seuil ou bien continue.

Dans la suite de ce projet, nous avons utilisé un MLP avec 784 neurones en entrée, soit la valeur de chaque pixel de l'image 28x28. La deuxième couche est composée de 500 neurones et la dernière correspond au nombre de classes du problème, ici 43 classes différentes. La fonction d'activation utilisée pour la couche cachée est une sigmoïde, comme le montre la figure suivante.



$$f(x) = \frac{1}{1+e^{-\lambda x}}$$

Figure 10 : Fonction d'activation sigmoïde [13]

La fonction d'activation utilisée pour la dernière couche est appelée « softmax ». Elle permet de calculer la probabilité d'appartenance à chaque classe pour une image. On sélectionne ensuite le plus gros score comme étant la classe à laquelle appartient notre image.

2.2.3. Réseau Support Vector Machine (SVM)

Les réseaux « Support Vector Machine », ou SVM, sont un type de réseau de neurones classificateur à noyau avec un apprentissage supervisé pour la classification et la régression. Ce modèle est a été inventé par V. Vapnik et A. Chervonenkis en 1963. Le but premier de ce type de réseau est de trouver des hyperplans permettant de trouver une séparation entre les données du problème. On cherche ensuite à agrandir au maximum la marge entre les deux classes, comme on peut le voir sur la figure ci-dessous.

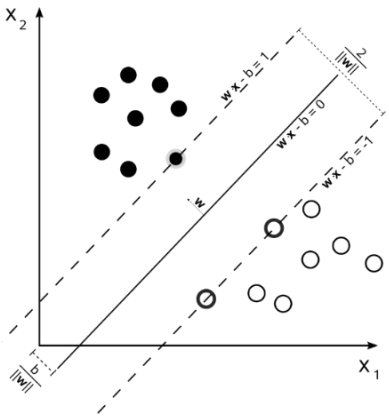


Figure 11 : Schéma du plan et de la marge obtenue avec un SVM

Chaque classe est représentée par un ensemble de gaussiennes en n dimensions avec un centre et une dispersion. On a plusieurs types de noyaux se basant sur différentes équations mathématiques : linéaire, non linéaire, polynomiale, RBF (« Radial Basis Function ») et sigmoïde. Pour notre projet, nous avons comparé les noyaux linéaire, polynomial et RBF.

$$K(x, x^T) = x^T . x \qquad K(x, x^T) = \langle \Phi(x^T), \Phi(x) \rangle$$

$$K(x, x^T) = (\gamma x^T . x + r)^d, \gamma > 0 \qquad K(x, x^T) = e^{\left(-\gamma \|x - x^T\|^2\right)}, \gamma > 0$$

$$K(x, x^T) = tanh(\gamma x^T . x + r)$$

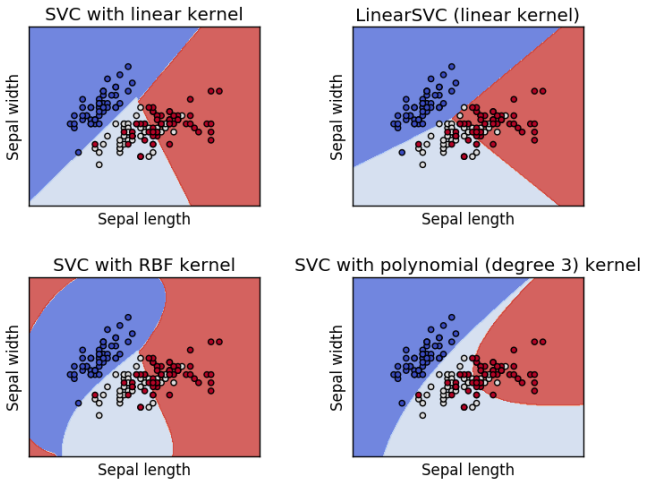


Figure 12 : Différents types de noyaux pour le SVM

Initialement, un SVM classique ne permet d'attribuer que deux classes. Pour rendre un SVM multiclasse, on compare plusieurs SVM binaires [3]. Pour cela, on a plusieurs approches différentes, « one vs all », « one vs one » ou « directed acyclic graph SVM » (DAGSVM). Le premier consiste à comparer une classe avec toutes les autres. On met la classe testée positive et toutes les autres négatives. Un élément appartient à notre classe seulement si la sortie est positive. On refait cette boucle le nombre de fois qu'on a des classes. La deuxième méthode demande plus de calcul, pour n classes, on doit faire $n(n-1)/2$ itérations. Dans cette méthode, on vient comparer une entrée avec toutes les paires de classes possibles. À chaque test, la classe la plus proche de l'entrée est incrémentée de 1. Au final, la classe retenue est celle qui a obtenu le plus grand score. La dernière méthode, DAGSVM, reprend la comparaison du modèle « one vs one », mais diffère dans la sélection de la classe, empêchant les égalités entre les classes.

3. Méthodologie expérimentale

3.1. Base de données

3.1.1. Présentation du GTSRB

Les données que nous avons utilisées pour notre étude proviennent toutes de l'institut de calcul neuronal de Bochum en Allemagne, « Institut für Neuroinformatik » [6]. Cette base de données est coupée en deux parties, une où les photos sont présegmentés, voir l'image ci-dessous. Et la deuxième partie est composée d'images semblables à la vue d'un conducteur. Il faut donc trouver le panneau dans l'image et ensuite le classer. Les premières données ont été utilisées lors de la compétition IJCNN en 2011 à San Jose en Californie. L'objectif était de comparer et de trouver l'algorithme neuronal le plus performant pour un problème ayant de nombreuses classes, 43 panneaux différents. La deuxième partie de la base de données a aussi été utilisée pour une compétition ILCNN en 2013 à Dallas au Texas qui consistait à utiliser les réseaux de neurones pour extraire les panneaux de l'environnement.

Figure 13 : Présentation des différentes classes de panneaux

La base de données regroupe plus de 50 000 images, 39 209 pour l'entrainement et 12 631 pour le test. Les images sont en couleurs RGB et ont une dimension comprise entre 20x20 à 200x200 pixels. À noter aussi que celles les images de l'entrainement sont labélisées et que celles du test ne le sont pas. Deuxième point, le nombre d'images par classe n'est pas égal et peut varier avec un coefficient de 10. Exemple, le panneau de limitation à 20 km/h est présent 210 fois alors que la limite à 50 km/h est présente 2250 fois. Ci-dessous un graphique montrant la répartition des photos selon les classes. Pour information, l'image ci-dessus a les classes dans le bon ordre, c'est-à-dire que la première image correspond au premier label, label 0 pour la programmation.

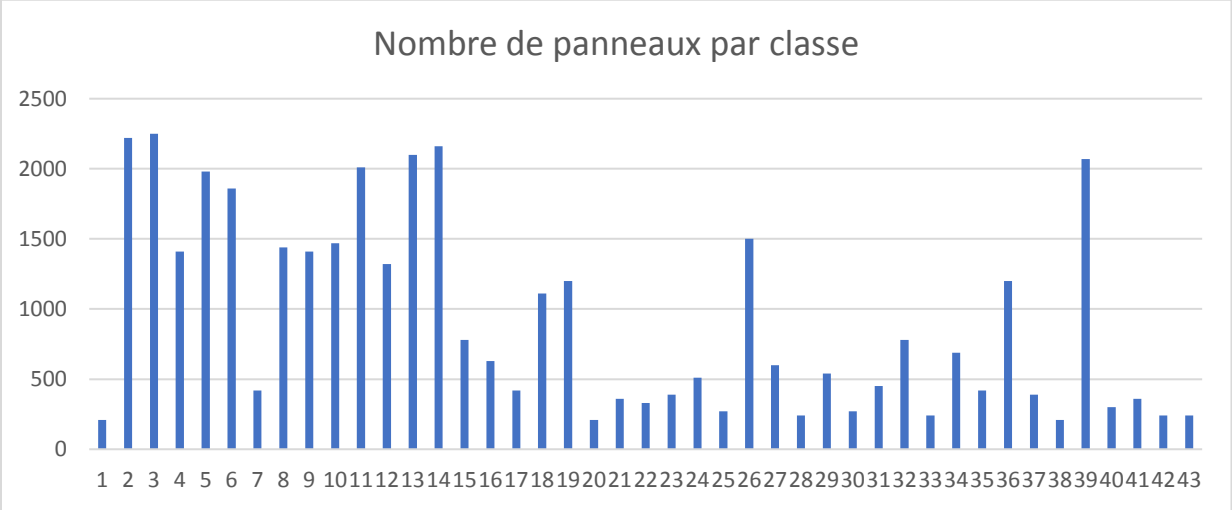


Figure 14 : Distribution des images en fonction des classes

3.1.2. Augmentation du nombre d'images

Lors de nos premiers résultats de nos réseaux SVM, nous avions des scores assez faibles de l'ordre de 50 % de reconnaissance. Nous avons aussi obtenu la matrice de confusion ci-dessous. La matrice de confusion est un résultat très utile pour analyser la performance d'un réseau. Elle est sous forme d'un tableau en deux dimensions avec en abscisse les prédictions du réseau et en ordonnée les vrais labels et on remplit le tableau en fonction des résultats

obtenus. Si un réseau est performant à 100 %, on doit obtenir une ligne diagonale, c'est-à-dire que les prévisions sont exactement les mêmes que les vrais labels. La matrice ci-dessous est celle obtenue avec un SVM linéaire, gamma égal à 0,1 et C=1.

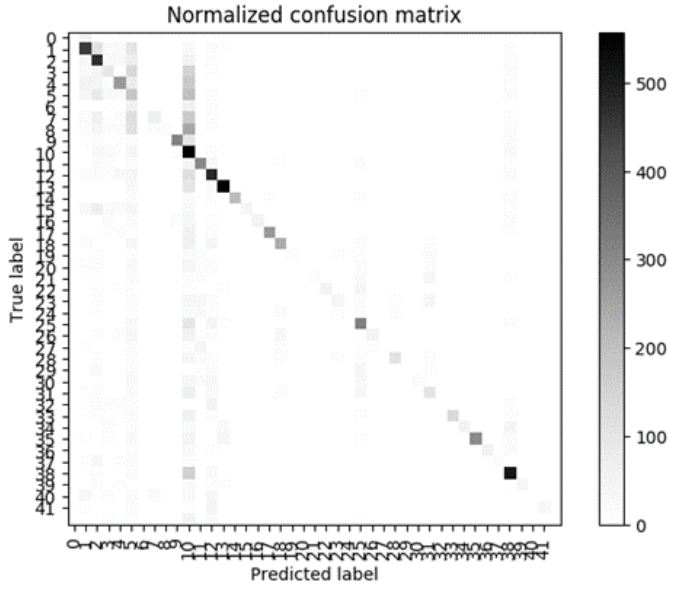


Figure 15 : Matrice de confusion d'un SVM

On voit très clairement sur l'image qu'il y a des traits verticaux. Ces traits verticaux correspondent aux classes les plus peuplées, classes 2, 3, 5, 10, 26 et 39. Nous avons donc pensé que le SVM n'arrivait pas à reconnaitre les classes faiblement peuplées et que les classes les plus peuplées écrasaient les plus petites.

Nous avons donc réalisé un code qui permet d'augmenter le nombre d'images. Pour faire cela, le programme sélectionne une image, lui applique au hasard, un effet miroir vertical ou horizontal, les deux ou bien ne la modifie pas. Ensuite, toujours au hasard, l'image subit une rotation variant entre -15 ° à + 15 °. Ces opérations ont lieu jusqu'à ce que toutes les classes aient le même nombre d'images soit 2250. En faisant cela, on a plus que doublé notre base de données de départ en passant de 39 209 images à 96 750 images.

Nous avons étudié la combinatoire de ce programme pour savoir s'il n'y a pas trop d'images en doubles créées. Ci-dessous des exemples d'images transformées. On a une chance sur 4 pour l'effet miroir et une chance sur 31 (nombres entiers entre -15 et +15) pour la rotation ; soit une chance sur une chance sur 124. Pour repeupler la classe l'a moins dense, il faut créer 2040 images à partir de 210 images.

$$P_{doublon} = \frac{Nb\ d'images\ créés}{Nb\ de\ flips \times Nb\ de\ rotations \times Nb\ d'images\ d'origine} = 7{,}83\ \%$$

Étant donné que le nombre de doublons dépend du nombre d'images par classes, nous avons fait une moyenne des probabilités obtenues. Nous avons une probabilité moyenne de 2,84 % de créer des doublons dans la base de données, soit 2 750 images sur 96 750.

Figure 16 : Exemples d'images supplémentaires créées

Nous avons relancé le SVM avec les anciennes et les nouvelles images. Nous avons obtenu des scores plus faibles et des temps de calcul beaucoup plus longs. C'est pourquoi nous avons mis de côté cette idée et n'avons pas cherché à diminuer le nombre de doublons.

3.2. Réalisation des réseaux étudiés

3.2.1. Première approche

Suite à l'étude de la littérature, nous avions envie de comparer trois réseaux. Nous avons trouvé trois articles [2] [4] [8], utilisant respectivement un MLP, un SVM et un CNN pour classifier les panneaux issus de notre base de données. Un point qui nous a paru

problématique pour réaliser notre comparaison : chaque article utilise un prétraitement différent.

Pour le MLP, les auteurs ont utilisé un algorithme de reconnaissance de forme, circulaire, carré ou triangle. Pour ensuite appliquer le MLP aux nouvelles catégories préclassées. De plus, les auteurs enregistrent les maximums des valeurs R, G et B de l'ensemble de l'image. Ensuite, ils convertissent les images originelles en images grises de dimension 30x30 pixels pour en extraire la valeur maximale par ligne et par colonne. Ils obtiennent donc 63 nombres par images qui passent ensuite dans leurs MLP.

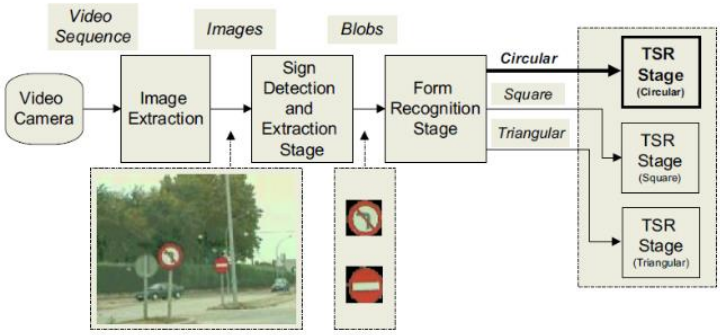


Figure 17 : Vu d'ensemble du premier article

Deuxièmement, les auteurs de l'article se basant sur un SVM utilisent la décomposition suivante : couleur dominante, puis forme du panneau, pour enfin appliquer le SVM à l'intérieur du panneau. Ils n'appliquent pas de traitement sur les images directement, mais ces deux décompositions successives reposent sur deux autres SVM.

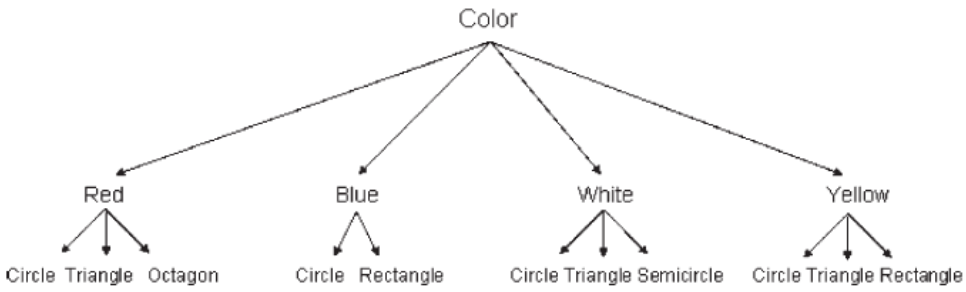


Figure 18 : Vue d'ensemble du second article

Le dernier article a lui aussi un prétraitement. Les auteurs ont comparé justement différents types d'opérations sur les images avant de les mettre dans le CNN pour voir ensuite la différence dans les résultats. Ils convertissent l'image en 32x32 pixels et passent en YUV pour les couleurs. Ci-dessous, les différents prétraitements appliqués dans l'article.

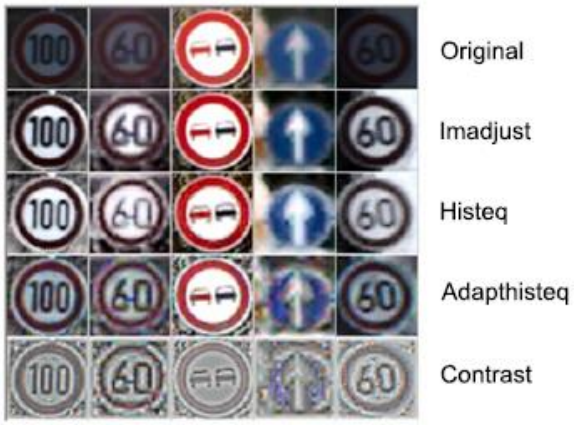


Figure 19 : Vue d'ensemble du troisième article

Étant donné que nous voulions comparer les trois réseaux, il nous faut la même base de comparaison et donc le même prétraitement. Sans cela, les résultats obtenus ne seront pas pleinement exploitables et seront biaisés par le prétraitement plus au moins favorable, plus ou moins optimisé pour un réseau plutôt qu'un autre. Voici ci-dessous la première architecture de nos réseaux. Le but était de n'avoir que le réseau à changer dans le code. Le prétraitement appliquer ici est de simplement passé l'image en nuances de gris et de les redimensionner en 28x28 pixels. Ces choix sont tout à fait arbitraires et pourront faire l'objet de futures comparaisons. Nous avons donc réalisé trois codes python relativement similaires. Seule la partie du réseau diffère comme vous pouvez le voir en annexe.

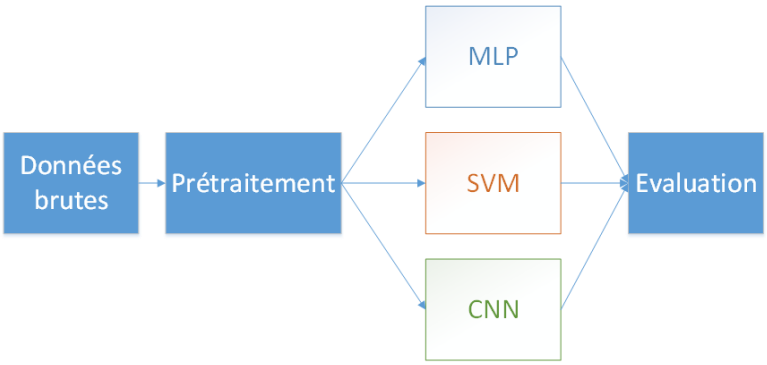


Figure 20 : Première architecture du système

3.2.2. Deuxième approche

Avec les résultats que nous avons obtenus et avec les conseils de Eric Granger durant la présentation du projet, nous avons modifié l'architecture de notre étude. En effet, les réseaux MLP et SVM ont des résultats peu élevés puisqu'ils ne sont pas faits pour extraire les caractéristiques d'une image. La principale modification que nous avons apportée suite à la présentation est de mettre un CNN commun pour ensuite comparer les performances entre le MLP et le SVM comme le montre l'image ci-dessous.

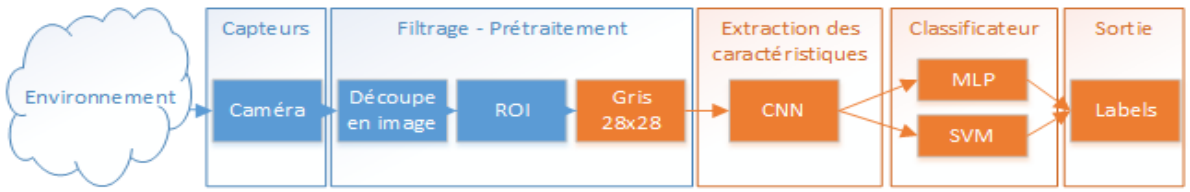


Figure 21 : Deuxième architecture du système

3.2.3. Précision sur la programmation

Pour l'ensemble du projet, nous avons réalisé les réseaux en langage Python 2.7 sous Ubuntu 16.04 avec l'environnement de développement intégré PyCharm sur un ordinateur avec un processeur Core i5-3230 et 8GB de mémoire. Nous avons utilisé différentes bibliothèques spécialisées dans les réseaux de neurones. Pour le MLP, nous avons utilisé TensorFlow. Pour le SVM, nous avons utilisé Scikit Learn et pour le CNN, nous avons utilisé deux bibliothèques, TensorFlow et Keras. Nous avons aussi utilisé la bibliothèque OpenCV

pour lire les images et réaliser l'ensemble des opérations pour agrandir la base de données. L'ensemble des codes est présent en annexe de ce document.

Figure 22 : Bibliothèques utilisées pour la programmation

Pour nous accompagner dans nos toutes premières lignes de codes en Python, nous nous sommes appuyées sur de nombreux exemples et aides en lignes. Les principales sources sont les suivantes, pour le MLP [14] [15], pour le SVM [16] et pour le CNN [17]. Notre mode de fonctionnement a été de trouver des codes semblables à nos attentes et nous les avons adaptés pour qu'ils fonctionnent avec nos données.

Nous avons réussi à faire trois programmes distincts ayant le même prétraitement et la même analyse des résultats, c'est-à-dire la première approche. Malheureusement, nous n'avons pas réussi à mettre un SVM après le CNN comme souhaité dans la deuxième méthode. Pour pallier à ce manque, nous contourner le problème et nous avons décidé de comparer le SVM et le MLP avant le CNN. C'est-à-dire, de ne mettre que le SVM ou le MLP pour faire toute la classification. Nous avons comparé deux architectures de CNN et fait varier plusieurs variables comme nous le ferons dans la partie suivante.

Le deuxième problème que nous avons eu est que les bibliothèques ne fonctionnent pas de la même façon. TensorFlow et Keras utilisent 80 % des quatre cœurs de notre PC alors que Scikit Learn n'utilise qu'un seul cœur à 100 %. Un travail futur serait de faire la programmation en utilisant la même bibliothèque pour les trois réseaux pour avoir des résultats plus comparables. De même, nous n'avons pas été capables d'installer CUDA et

CAFFE, des bibliothèques qui permettent de faire les calculs sur le GPU et ainsi de réduire les temps de calcul. Dernier point, pour le réseau VGG16, nous avons simplifié son architecture, comme le montre la figure ci-dessous pour pouvoir le faire tourner sur notre ordinateur.

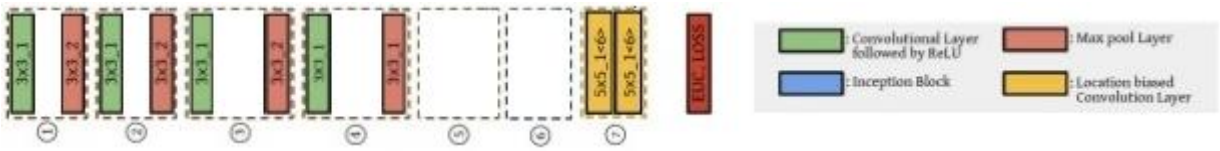


Figure 23 : Architecture modifiée du VGG16

3.3. Critères d'évaluation des performances

3.3.1. Entrainement des réseaux

Dans la suite de projet, quand nous parlons de temps, il s'agit du temps nécessaire à la réalisation complète du code. Chaque code se compose selon le même ordre : importation des paquets nécessaires, lecture et prétraitement de toutes les images disponibles, découpage des images en deux groupes (entrainement et test), entrainement du réseau avec les images dédiées, évaluation de la performance du réseau avec les images de test, construction et sauvegarde de la matrice de confusion et du ROC. Pour éviter d'avoir trop de répétition dans le code, nous avons mis uniquement les parties exclusives en annexes.

Pour revenir à l'entrainement des réseaux ; étant donné que nous n'avons pas été capables d'enregistrer les poids pour chacun d'eux, nous les avons comparées sur le temps d'exécution total. C'est-à-dire le temps de l'entrainement, de l'évaluation et de la sauvegarde des résultats.

Pour ce qui est des données, le code lie les 39 209 images de la base de données qui sont mises dans un seul dossier. Il les convertit en noir et blanc et vient stocker la valeur de chaque pixel en une seule ligne, comme le montre la figure ci-dessous.

Figure 24 : Stockage des données en ligne

Le label du panneau est présent dans son nom de fichier et le code vient le stocker dans un vecteur. Ensuite, on transforme ce vecteur en une matrice ayant un nombre de colonnes égales au nombre de classe du problème. Au lieu d'avoir le numéro du label, on a uniquement un 1 dans la colonne correspondant au label de l'image. Par exemple, si une image correspond au label 10, alors il y aura un 1 dans la 10^{e} colonne et des zéros dans toutes les autres cases (attention de bien commencer à zéro pour les deux notations). À partir de ce moment, on a deux matrices : une de dimension 39 209 x 784 rassemblant la valeur de chaque pixel de chaque image, et l'autre de 39 209 x 43 avec les labels, qui seront les mêmes pour tous les réseaux.

Dernière étape, on vient couper en deux ces deux matrices selon un ratio de 66 %, 33 %. Les deux premiers tiers des données seront consacrés à l'entrainement du réseau. Le dernier tiers servira à l'évaluation du réseau. L'évaluation a lieu comme suit, on compare la prédiction que fait le réseau sur les données de test avec leurs véritables labels. Grâce à cela, on peut connaitre la précision du réseau, c'est-à-dire le pourcentage d'images correctement labélisées. On peut aussi obtenir la matrice de confusion étant donné que l'on connait les prédictions du réseau et les vrais labels. De même, avec ses données, on peut avoir le ROC pour chaque classe.

3.3.2. Différents critères de performance et test d'évaluation

Pour comparer les performances de nos réseaux, nous avons défini plusieurs outils dans la partie précédente. À savoir que nous avons comme critères : le temps d'exécution, la précision du réseau, la matrice de confusion et le ROC (« Receiver operating characteristic »). Dernier critère important, nous avons divisé la précision des réseaux par le temps de calcul pour donner ce que nous avons appelé la performance du réseau. Cette valeur permet de rapidement comparer si un réseau est meilleur qu'un autre. La précision s'exprime en pourcentage multiplié par cent, variant entre 0 et 100. Le temps est en minute et varie entre 0 et 440. Il n'est pas question ici de normaliser ou de pondérer les deux variables.

$$Performance = \frac{Précision\ (\%) \times 100}{Temps\ (minutes)}$$

Le ROC permet de montrer graphiquement la performance d'un classifieur binaire. Plus la courbe est proche des côtés gauche et haut et plus le classificateur est performant. À l'inverse, si le résultat est proche de la diagonale alors la performance est basse. Les ROC présents dans la partie résultat regroupent l'ensemble des classes sur un seul graphique. Il faut bien comprendre que le ROC est pour comparer un classificateur binaire, ayant que deux classes, ce qui n'est pas notre cas. Nous avons donc représenté les vrais positifs et les faux positifs pour chaque label.

Nous avons aussi 4 types de réseaux différents : le MLP, le SVM, le CNN LeNet et le CNN VGG16 simplifié. Pour le premier, nous avons fait varier le nombre de neurones dans la couche caché pour voir comment cela influe sur les performances. Pour le SVM, nous avons

fait varier le noyau parmi les suivants : linéaire, polynomiale et RBF. Nous avons fait varier le coefficient d'apprentissage, le nombre d'époques et la taille du lot pour les deux CNN.

Nous avons aussi mis en place un test pour comparer les réseaux sur 43 images représentant les panneaux dans des conditions optimales (sans déformation et sans environnement autour). Nous avons testé ces panneaux avec le réseau le plus performant après l'étude des différentes variables évoquée précédemment. Le résultat est que le réseau n'a reconnu correctement que 26 panneaux sur les 43 présentés. Cela nous a paru trop distant des images utilisées pour l'entrainement des réseaux d'où le résultat très mauvais. Nous nous contenterons de comparer les critères de temps et de précision pour la suite de ce projet.

Figure 25 Panneaux pour le test et matrice de confusion obtenue avec LeNet5

4. Résultats des simulations

4.1. Présentation des résultats

Dans cette partie, nous allons vous présenter tous les résultats obtenus. Nous discuterons d'eux dans la partie suivante. Voici l'ordre dans lequel nous allons vous présenter les résultats et les différentes variables évaluées à chaque fois.

- MLP
  - Neurones dans la couche cachée : 2000, 1500, 1000, 500, 250, 125, 75, 43, 25, 10, 1
- SVM
  - Noyau : linéaire, polynomial et RBF
- Comparaison entre SVM et MLP
  - Base de données simple et étendue
- CNN LeNet5
  - Taux d'apprentissage : 0,1, 0,01
  - Batch : 10, 100, 1000
  - Epoch : 10, 100
  - Comparaison LeNet5 et VGG16 simplifiée
  - Epoch : 10, 100

Soit un total de 30 entrainements différents, 12 pour le MLP, 4 pour le SVM, 12 pour le CNN LeNet5 et 2 pour VGG16.

Les premiers résultats proviennent d'un MLP ayant 784 neurones sur la première couche et 43 sur la dernière. Cela correspond au nombre de pixels par image en entrée et au nombre de classes que nous avons. Nous avons donc fait varier le nombre de neurones dans la couche caché et noté les résultats obtenus. Ci-dessous, le tableau regroupe tous les résultats obtenus et nous avons mis les matrices de confusions et les ROC pour 1, 43, 500 et 2000 neurones.

À noter que nous avons réalisé un code ayant une deuxième couche cachée. Mais les résultats n'étaient vraiment pas élevés et le fait de varier le nombre de neurones dans ces

deux couches cachées n'a pas fait évoluer les résultats. C'est pourquoi nous ne vous présenterons pas cela dans ce rapport.

Nb neurones cachés	Temps	Précision	Performance
2000	6'0"	89,41 %	14,90
1500	7'12"	89,84 %	12,57
1000	4'12"	89,81 %	21,49
500	2'24"	89,71 %	37,54
250	1'36"	88,86 %	55,89
125	1'6"	89,63 %	81,48
75	0'48"	88,79 %	110,99
43	0'42"	88,38 %	124,48
25	0'42"	87,77 %	135,03
10	0'36"	86,90 %	140,16
1	0'36"	56,45 %	91,05

Tableau 1 : Comparaison des performances selon les neurones cachés du MLP

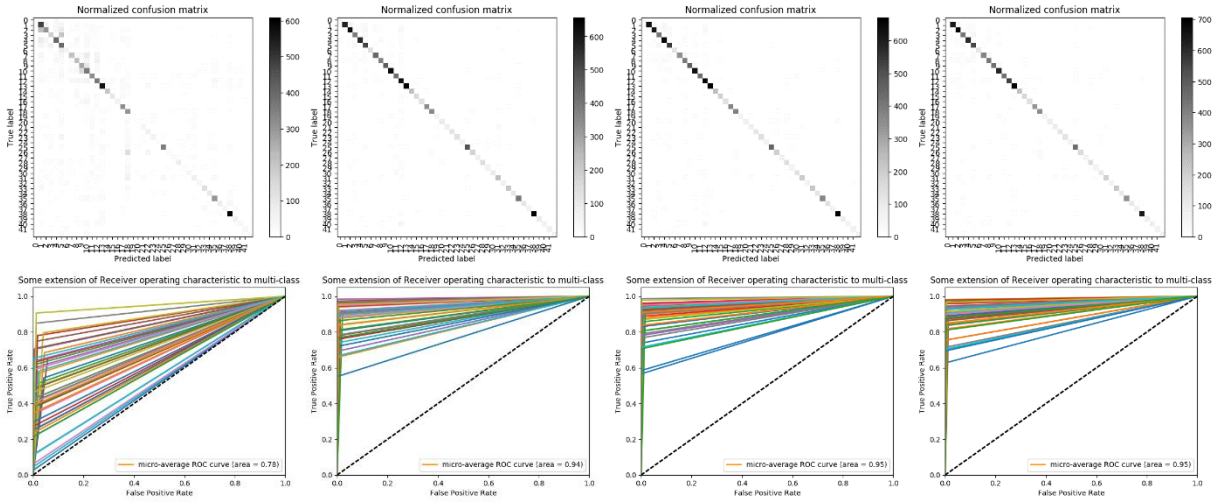


Figure 26 : Matrices de confusion et ROC pour le MLP (1 – 43 – 500 – 2000 neurones cachés)

Nous passons ensuite aux SVM. Pour les différents noyaux, nous avons pris un gamma égal à 0,1 et un C à 1. Le gamma est une variable dans les formules du RBF et du noyau polynomial que l'on définit. Le C correspond au nombre de points pris en compte pour tracer

l'hyperplan séparateur. Comme pour le MLP, nous avons rassemblé les résultats dans un tableau suivi par les matrices de confusions et les ROC des trois tests ici.

Noyau	Temps	Précision	Performance
Linéaire	5'36"	93,40 %	16,56
Polynomiale	6'36"	79,81 %	12,02
RBF	22'49"	83,96 %	3,68

Tableau 2 : Comparaison des performances selon le noyau du SVM

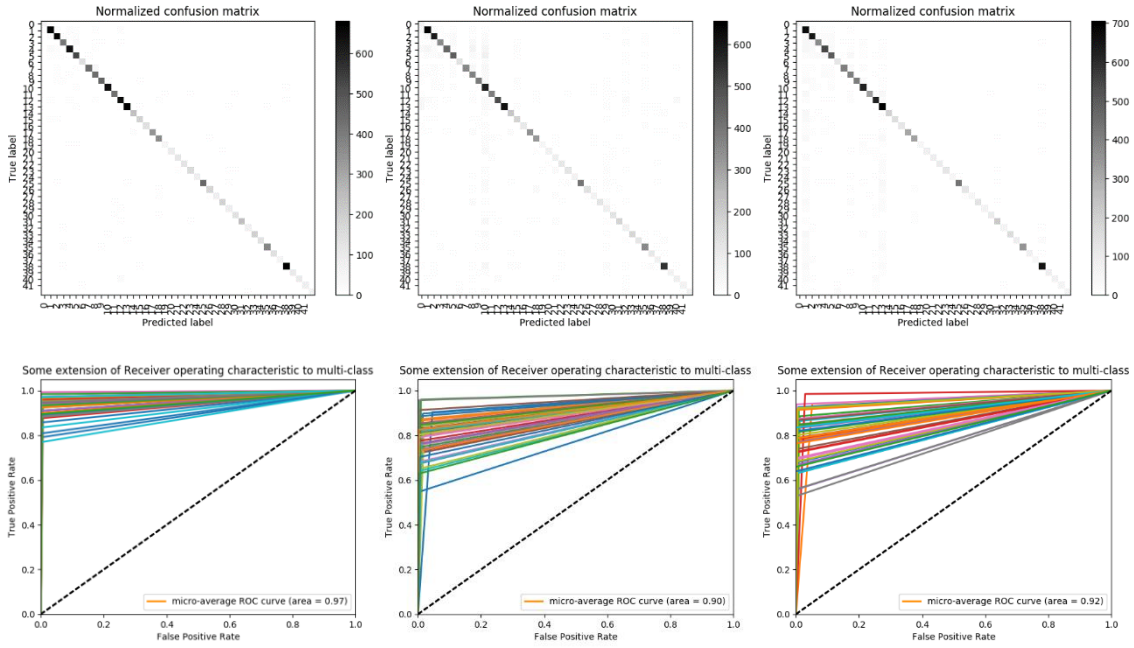


Figure 27 : Matrices de confusion et ROC des SVM linéaire, polynomiale et RBF

Après avoir évalué le MLP et le SVM, on a sélectionné les deux plus performants. C'est-à-dire le MLP 784-500-43 et le SVM linéaire. On les a entrainés à nouveau avec la base de données étendue de 96 750 images, obtenues à partir des 39 209 images originales. On peut voir ci-dessous le tableau regroupant les résultats ainsi que la matrice de confusion et le ROC du SVM linéaire entrainé avec la base de données étendue.

Réseau	Données	Temps	Précision	Performance
SVM Linéaire	Simple	5'36"	93,40 %	16,56
SVM Linéaire	Étendue	37'54"	80,94 %	2,14
MLP 500	Simple	2'24"	89,71 %	37,54
MLP 500	Étendue	5'12"	79,42 %	15,16

Tableau 3 : Comparaison entre SVM et MLP avec les deux bases de données

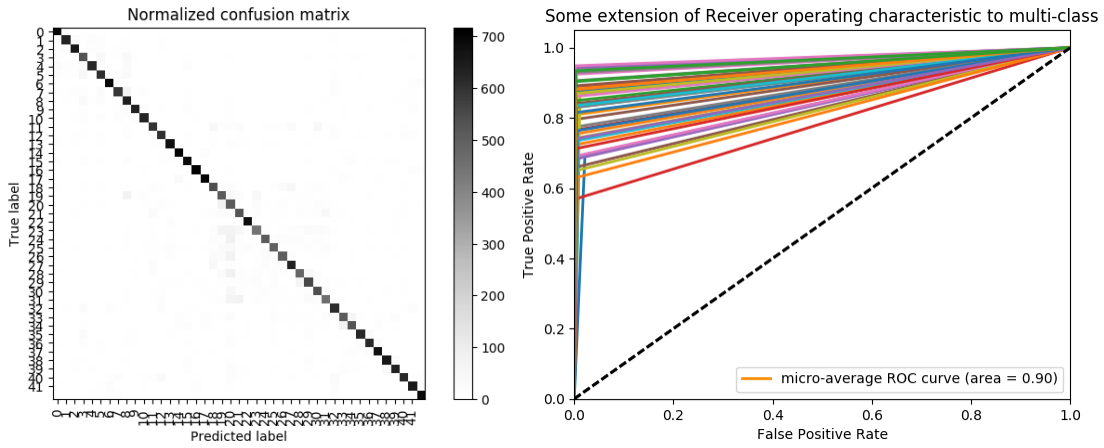


Figure 28 : Matrice de confusion et ROC du SVM linéaire avec la base de données étendue

Nous passons ensuite au premier CNN, LeNet5. Pour celui-ci nous avons fait varier 3 paramètres : le taux d'apprentissage, le nombre d'époques et la taille du lot. À savoir que nous avons utilisé la descente du gradient (Stochastic gradient descent, SGD) comme optimisateur. Le taux d'apprentissage est le paramètre associé au SGD, plus il est haut et plus le réseau est sensible à la dernière modification et à l'inverse, pour un taux d'apprentissage faible, le réseau réagira lentement. Le nombre d'époques et la taille du lot sont des paramètres utilisés durant l'entrainement du réseau. Le lot est le nombre d'images, batch en anglais, qui va traverser le réseau avant qu'on enregistre les poids. On répète cette opération jusqu'à avoir fait passer toutes les images disponibles pour l'entrainement. Cela représente une époque. On peut donc choisir de refaire cette opération pour que le réseau apprenne à bien calibrer ses poids.

Les deux graphiques suivants illustrent l'évolution de l'erreur à chaque époque.

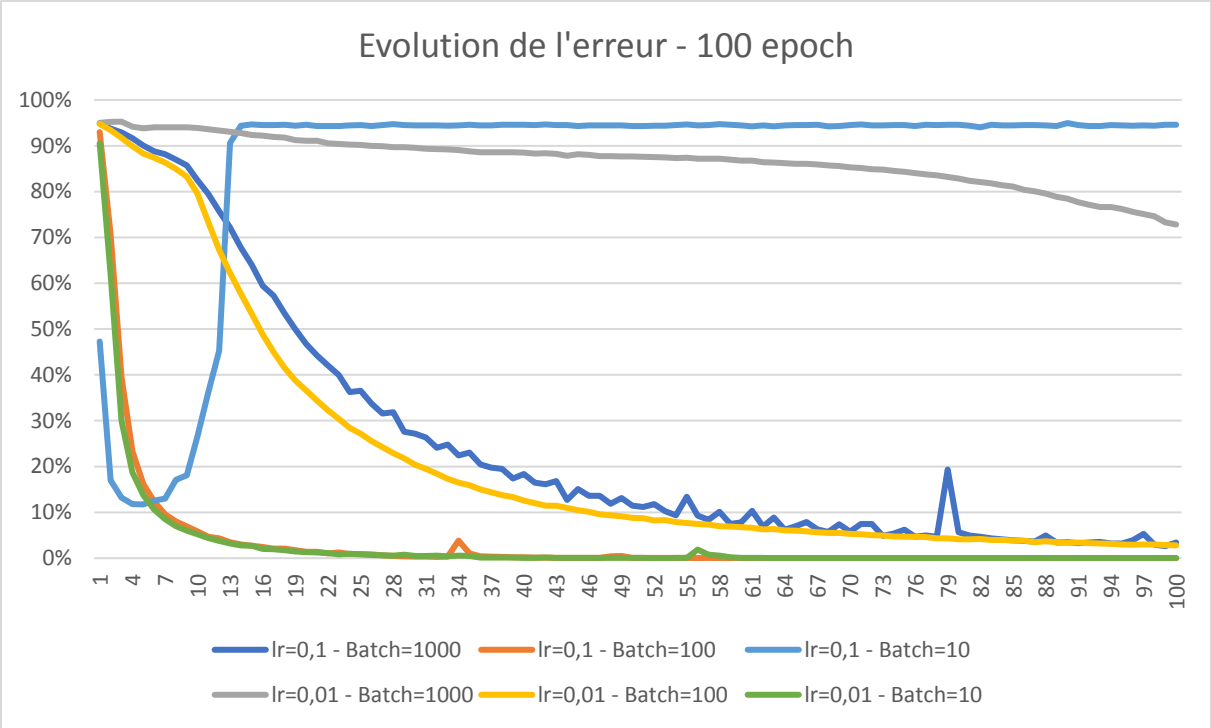


Figure 29 : Evolution de l'erreur du CNN LeNet5 pour 100 époques

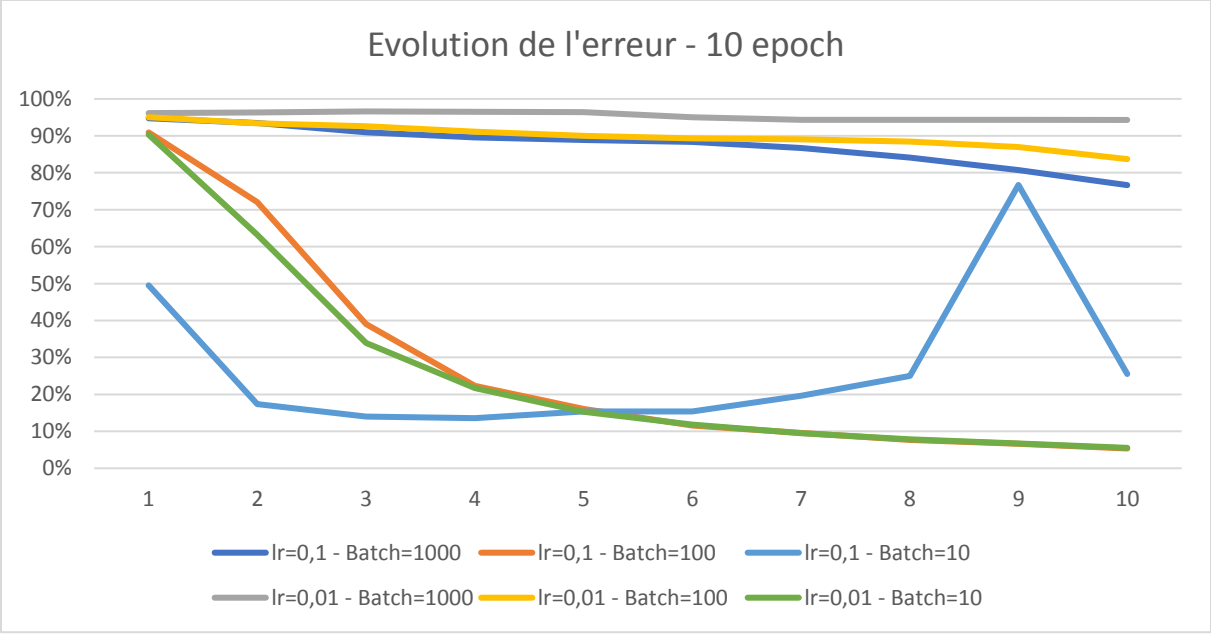


Figure 30 : Evolution de l'erreur du CNN LeNet5 pour 10 époques

Nous avons regroupé les résultats de chacun des douze tests dans le tableau suivant. Les résultats obtenus dépendent de 3 facteurs différents d'où la complexité de représentation des résultats.

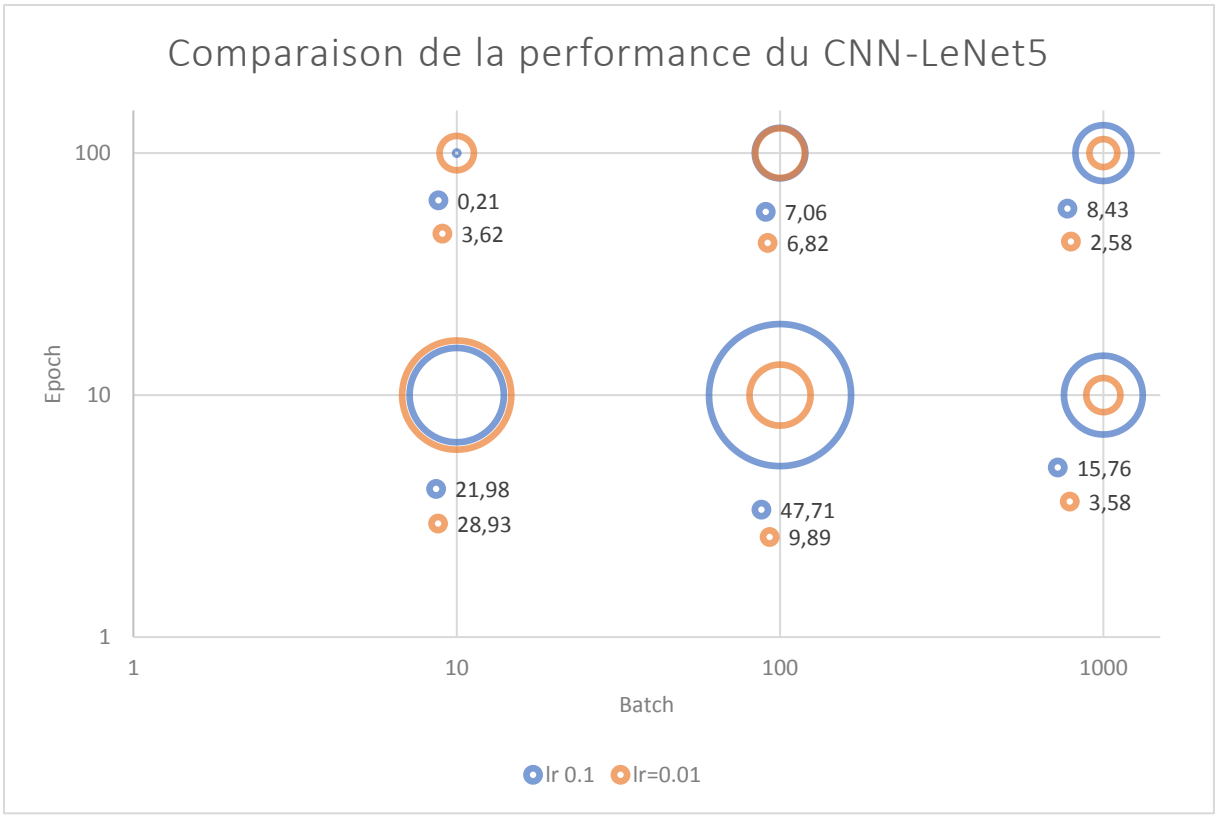


Figure 31 : Comparaison des performances obtenues avec le réseau CNN LeNet5

Epoch	100	5,8% 27'48" Perf : 0,21	97,2% 26'48" Perf : 3,62	97,5% 13'48" Perf : 7,06	94,2% 13'48" Perf : 6,82	92% 10'54" Perf : 8,43	27,8% 10'48" Perf : 2,58
	10	74,3 % 3'24" Perf : 21,98	92,8 % 3'12" Perf : 28,93	93,5 % 1'36" Perf : 47,71	19,1 % 1'54" Perf : 9,89	25,4 % 1'36" Perf : 15,76	5,7 % 1'36" Perf : 3,58
lr=0,1		10		100		1000	
lr=0,01		Batch					

Tableau 4 : Comparaison des performances obtenues avec le réseau CNN LeNet5

Enfin, dernière comparaison, celle entre le réseau LeNet5 et le VGG16 simplifié. Nous n'avons réalisé que deux tests sur le VGG16 étant donné ses temps de calcul très importants (plus de 7 h pour 100 époques) et sa faible précision. Ci-dessous, le tableau des résultats suivi de la matrice de confusion et le ROC pour 10 époques et un lot de 100.

Réseau	lr	batch	epochs	time (m)	accuracy	Perf
VGG16	0,1	100	100	440,62	1,47	3,34 E-03
VGG16	0,1	100	10	46,78	5,04	0,11
LeNet5	0,1	100	100	13,8	97,47	7,06
LeNet5	0,1	100	10	1,96	93,52	47,71

Tableau 5 : Comparaison des performances entre les CNN LeNet5 et VGG16

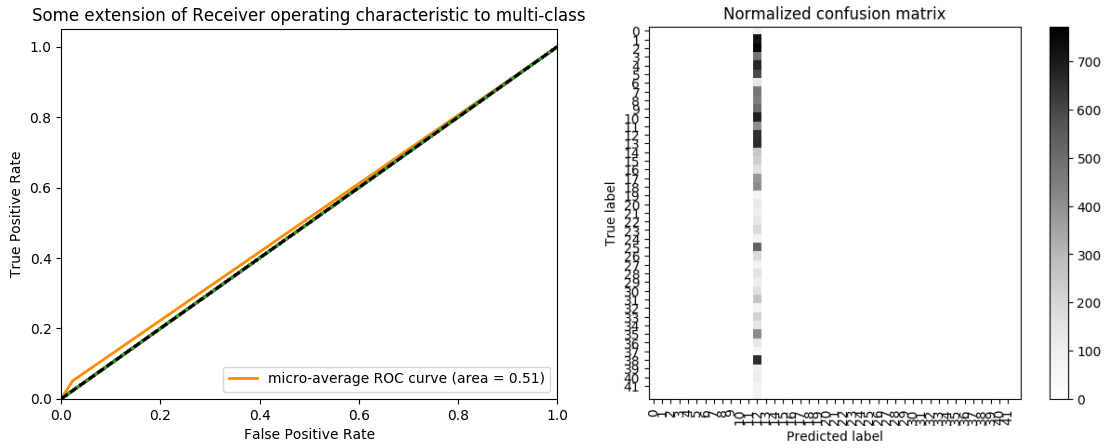


Figure 32 : Matrice de confusion et ROC obtenus avec le CNN VGG16 (Batch=100 ; Epoch=10)

4.2. Interprétation des résultats

Dans cette partie, nous allons revenir sur les résultats que nous avons obtenus, l'interpréter et essayer de les expliquer.

Tout d'abord, pour le MLP et le nombre de neurones dans la couche cachée, les résultats montrent que plus on augmente le nombre de neurones plus le réseau a besoin de temps. Cette augmentation n'est pas linéaire et est même des fois inversées par exemple pour 1500 neurones, on a besoin de 7 minutes alors qu'avec 500 neurones de plus, soit 2000, on n'a besoin que de 6 minutes seulement. Il y a aussi une limite basse dans le temps de calcul qui se résume à la lecture des données et à la création de la matrice de confusion et du ROC. Ce qui est à noter aussi est que le fait d'augmenter le nombre de neurones ne fait pas augmenter la précision du MLP. Elle stagne juste en dessous de 90 % qu'on ait 2000 neurones

ou bien 125. Pour les valeurs en dessous de 125, on voit que la précision vient à décroitre très rapidement. Cela est dût au fait que le réduit de trop l'information présente dans les 784 neurones précédents. Aux vues des résultats et de la littérature [2], nous avons décidé de garder une réduction d'un tiers du nombre de neurones entre la couche d'entrée et la couche cachée, soit 500 neurones dans la couche cachée dans notre cas. Les matrices de confusion sont intéressantes dans le sens où, du fait du nombre différent de photos par classe, on obtient une diagonale plutôt marquée, mais pas équilibrer. Elle reprend la densité des classes.

Passons aux différents noyaux pour le SVM. On observe que le noyau le plus simple, le linéaire, obtient de meilleurs résultats et dans un temps moindre que les deux autres noyaux. La principale différence entre le linéaire et la polynomiale et le RBF est que ces deux derniers ne sont pas linéaires. Ce qu'on peut observer sur les matrices de confusions est que le SVM linéaire présente moins de lignes verticales ce qui montre bien qu'il est plus performant que les deux autres. C'est donc lui qui sera retenu pour être comparé au MLP 784-500-43 pour savoir qui est le meilleur classificateur.

Vient ensuite le test entre le meilleur MLP et le meilleur SVM sur les deux bases de données. On connait déjà les performances du SVM et du MLP sur la base de données simple. Les résultats sont assez proches, le MLP a pourtant une légère avance étant donné qu'il est deux fois plus rapide que le SVM pour seulement 4 % de moins sur la précision. Quand on regarde leurs performances, le MLP est à 37,5. Il est presque deux fois plus performant que le SVM qui n'est qu'à 16,5. Les résultats des tests avec la base de données étendue ne sont pas très satisfaisants dans le sens où l'on perd en précision et qu'on augmente les temps. Le SVM voit son temps multiplier par sept et perd en même temps 13 % de précision en passant à 80 % de bonne reconnaissance. Le MLP n'échappe pas non plus à cela. On perd 10 % de

précision tout en doublant le temps de calcul. Le fait que l'on augmente les temps de calcul en plutôt compréhensible vu que l'algorithme à plus de données a traité. Ce qui est moins logique est que l'on perd de la précision. Notre avis sur ce point est que premièrement, à cause de l'effet miroir, certains panneaux deviennent confondus. C'est le cas pour les obligations de tourner à droite et à gauche. De même pour les limites de vitesse où le 2 est confondu avec le 5 lorsqu'il est inversé. Deuxièmement, les rotations font apparaitre des triangles noirs sur les contours de l'image. Le réseau a donc moins de pixels, moins de données pour les reconnaitre. Pire encore, il peut assimiler les triangles noirs à une certaine classe et donc mal classer les images. Le seul point positif à l'augmentation du nombre d'images est que l'on a bien une diagonale ayant le même ordre de grandeur dans la matrice de confusion. On a bien répondu à notre hypothèse au début, en voyant une diagonale inégale. Toutefois, la précision du réseau est beaucoup plus préoccupante que la répartition des données dans les classes. D'autant plus que dans la réalité, il n'y a pas un nombre égal de chaque type de panneau.

On passe ensuite au CNN LeNet5. Le premier graphique concernant l'évolution de l'erreur en fonction des époques est très intéressant. Il nous permet de voir que toutes les courbes tendent à diminuer plus moins rapidement, sauf pour celle ayant un taux d'apprentissage à 0,1 et un lot de 10. Nous reviendrons sur ce cas après nos explications. Pour le taux d'apprentissage de 0,01, on voit que plus le lot est petit et plus l'apprentissage est rapide, moins il y a d'erreurs (courbes grise, jaune et verte). On retrouve la même relation sur les deux premières courbes pour le taux d'apprentissage de 0,1. Mais ces deux dernières sont déjà plus rapides. Cela s'explique par le fait que si augmente l'importance du dernier lot, c'est-à-dire un taux d'apprentissage élevé, il faut aussi diminuer la taille du lot. Malgré le fait qu'en augmentant le taux d'apprentissage, on la vitesse d'apprentissage, on voit qu'il y a des

perturbations plus importantes. C'est tout à fait logique puisque plus on augmente le taux d'apprentissage plus le réseau apprend uniquement sur le dernier lot vu. Si ce lot est mal classé alors le réseau va changer beaucoup de poids. Alors qu'avec un taux d'apprentissage plus bas, il apprend moins vite, mais est beaucoup plus stable. Pour revenir au cas de la courbe bleue claire, lr=0,1 - Batch=10, nous pensons qu'il y a eu comme une perte de contrôle, le réseau n'arrive pas à converger vers une solution puisqu'il change fortement à chaque petit lot. C'est comme si l'on demandait de faire un paramétrage personnalisé pour chaque image. Forcément, cela ne va pas reconnaitre les caractéristiques importantes.

On retrouve exactement le même schéma sur le deuxième graphique, basé sur 10 époques seulement. Les courbes orange et verte (lr=0,1 — Batch=100 et lr=0,01 — Batch=10) sont confondues, tout comme la bleue foncée et la jaune (lr=0,1 — Batch=1000 et lr=0,01 — Batch=100). De la même façon, la courbe bleu clair n'arrive pas à converger.

Au niveau des performances, on obtient de très bons résultats, jusqu'à 97,5 % de bonnes reconnaissances pour le CNN lr=0,1 — Batch=100-epoch=100. La meilleure performance est obtenue grâce à un score de 93,5 % de reconnaissances exactes en seulement 1 minute et 36 secondes, lectures des données et enregistrement des résultats.

Pour finir cette analyse des résultats, nous devons dire que le VGG16 n'a pas apporté de bons résultats. De plus, il est très demandeur en ressources et en temps de calcul. Nous pensons que, peut-être, pour ce type de problème assez simple (données présegmentées, bien labélisées et assez distinctes) une architecture simple suffit pour correctement le résoudre. On peut employer une méthode bulldozer, forcément cela marchera, mais cela nécessite beaucoup de ressources. C'est peut-être le fait d'avoir simplifié très abruptement le VGG16 qui a fait qu'il n'est plus efficace.

5. Conclusion

5.1. Recommandations pour la suite du projet

Pour la suite de ce projet, nous pensons qu'il faudrait en tout premier lieu apporter plus de formalisme sur la réalisation informatique des réseaux. Il faudrait passer encore plus de temps pour réussir à installer toutes les bibliothèques nécessaires, notamment CUDA, pour pouvoir faire les calculs sur le GPU. Il serait même intéressant de refaire l'entrainement avec des réseaux plus profonds sur des ordinateurs plus performants pour vraiment voir si la précision s'améliore considérablement. Nous ne pensons pas à remettre en cause la base de données. Elle est suffisamment conséquente et l'on a vu que sa répartition dans les classes n'était pas importante outre mesure.

On pourrait aussi passer plus de temps à tester les différents paramètres et variables de chaque réseau pour les régler de façon très précise à notre base de données. De la même manière, réussir à faire un test capable de comparer la performance des réseaux de façon plus formelle. Nous n'avons pas pu calculer la complexité des algorithmes étant donné qu'avec l'utilisation des bibliothèques, nous avons utilisé des fonctions qui réalisent d'autres codes. L'exemple type est la fonction fit pour le SVM qui à lui seul fait tout l'entrainement du réseau.

Comme on l'a vu dans la partie précédente sur les résultats, on n'obtient qu'au maximum 97,5 % de reconnaissance exacte. Étant donné le contexte, les véhicules autonomes, cela n'est pas très sécuritaire vu le nombre de panneaux présent sur les routes. Parmi tous ces panneaux, 2,5 % d'entre eux seraient mal reconnu par notre réseau de neurones. Cela pose une nouvelle question, est-ce que les réseaux de neurones et leurs parts d'autonomie sont assez performants pour pouvoir assurer la sécurité des passagers ? On parle ici dans le cas d'une voiture complètement autonome, ne nécessitant aucune action humaine

pour être conduite. Que se passerait-il si la voiture comprend une limitation à 120 km/h au lieu de 50 en ville ? Cela pose de réels problèmes de sécurité. Ils devront être corrigés en utilisant d'autres aspects comme la détection de présence d'autres véhicules, l'analyse de leur vitesse, etc. étant donné que les réseaux de neurones ne garantissent pas une précision de 100 % à tous les coups.

Il est de noter que pour ce genre d'application, la précision du réseau est bien plus importante que sa rapidité d'entrainement. En fait, il suffit d'entrainer le réseau une première fois et ensuite on ne modifie plus les poids. Avec cette optique, le meilleur réseau que nous avons obtenu de 97,5 % a été obtenu avec le CNN LeNet5 et les paramètres suivants : lr=0,1 — Batch=100-epoch=100.

En plus, le score que nous avons obtenu est obtenu avec une base de données très soignée. Toutes les images sont parfaitement bien cadrées sur les panneaux, même si certaines photos sont sur ou sous exposées, qu'il y a parfois des objets qui cachent partiellement les panneaux ou encore des images très floues. Il ne faut pas oublier que notre projet nécessite la réalisation de la première grosse partie qui est la reconnaissance de la présence d'un panneau dans les vidéos prises par la voiture. Toute notre étude dépend de la qualité des données obtenues après cette première étape. On aura beau avoir le réseau le plus performant qu'il existe, entrainé le mieux possible, si les images que nous devrons analyser ne sont pas bien cadrées, alors les résultats seront très probablement médiocres.

5.2. Conclusion principale

Pour revenir sur l'ensemble du projet réaliser durant ces quatre derniers mois, nous avons réalisé plusieurs réseaux de neurones. Grâce à l'enseignement donné durant le cours, nous avons compris les différences intrinsèques de chacun des réseaux réutilisés dans ce projet. Nous avons essayé autant que possible de rester focaliser sur la problématique du départ qui était de comparer différents réseaux de neurones dans la classification de panneaux routiers. Nous avons dû faire face à une importante difficulté qui est la programmation en Python. Ce projet est nos premiers codes dans un langage de programmation à proprement parler, autre que le VBA.

Même si pour cette application particulière, la reconnaissance des panneaux routiers, les résultats obtenus ne sont pas forcément très réjouissants, nous avons pu apprécier la puissance de ces algorithmes. Ils sont capables d'extraire les features des images et de les comprendre, de les garder en mémoire et de classer les images en fonction de la présence de tel ou tel features en fonction de leur appartenance à telle ou telle catégorie. Ces algorithmes nous seront utiles pour d'autres applications et sont déjà en cours d'application dans le cadre d'un travail pour le club étudiant de l'ÉTS qui est la détection de cibles dans des images du sol prises depuis un drone.

A. Références

[1] «Sécurité routière au Canada,» Transports Canada, 07 02 2017. [En ligne]. Available: https://www.tc.gc.ca/fra/securiteautomobile/tp-tp15145-1201.htm. [Accès le 12 4 2017].

[2] R. G.-P. R. R.-Z. M. U.-M. M. &. L.-F. F. Vicen-Bueno, «Multilayer perceptrons applied to traffic sign recognition tasks,» *International Work-Conference on Artificial Neural Networks,* pp. 865-872, June 2005.

[3] C.-W. H. a. C.-J. Lin, «A comparison of methods for multiclass support vector machines,» *Neural network, Vol. 13, No. 2,* pp. 415-425, March 2002.

[4] S. L.-A. P. G.-J. H. G.-M. a. F. L.-F. Saturnino Maldonado-Bascón, «Road-sign detection and recognition based on support vector machines,» *IEEE transactions on intelligent transportation systems, 8(2),* pp. 264-278, 2007.

[5] M. W. H. &. F. H. Shi, «Support vector machines for traffic signs recognition,» *Neural Networks,* pp. 3820-3827, 2008.

[6] Neuroinformatik, Institut für, «The German Traffic Sign Recognition Benchmark,» 20 July 2015. [En ligne]. Available: http://benchmark.ini.rub.de/?section=gtsrb&subsection=news. [Accès le 28 Février 2017].

[7] P. &. L. Y. Sermanet, «Traffic sign recognition with multi-scale convolutional networks,» *Neural Networks (IJCNN), The 2011 International Joint Conference on,* pp. 2809-2813, 2011.

[8] D. M. U. M. J. &. S. J. CireşAn, «Multi-column deep neural network for traffic sign classification,» *Neural Networks, 32,* pp. 333-338, 2012.

[9] B. Rohrer, «How Convolutional Neural Networks work,» 18 août 2016. [En ligne]. Available: https://www.youtube.com/watch?v=FmpDIaiMIeA&t=300s. [Accès le 28 Février 2017].

[10] E. V. Alliance, «"Large-Scale Deep Learning for Building Intelligent Computer Systems," a Keynote Presentation from Google,» 16 05 2016. [En ligne]. Available: https://www.slideshare.net/embeddedvision/largescale-deep-learning-for-building-intelligent-computer-systems-a-keynote-presentation-from-google. [Accès le 18 04 2017].

[11] X. Giro, «DeepFix: a fully convolutional neural network for predicting human fixations (UPC Reading Group),» 27 10 2015. [En ligne]. Available:

https://www.slideshare.net/xavigiro/deepfix-a-fully-convolutional-neural-network-for-predicting-human-fixations. [Accès le 18 04 2017].

[12] «Diagram of a perceptron,» 27 Février 2017. [En ligne]. Available: http://tex.stackexchange.com/questions/104334/tikz-diagram-of-a-perceptron. [Accès le 28 Février 2017].

[13] D. M. Humphrys. [En ligne]. Available: http://computing.dcu.ie/%7Ehumphrys/Notes/Neural/Bitmaps/sigmoid.gif. [Accès le 14 04 2017].

[14] «MNIST For ML Beginners,» TensorFlow, 20 12 2016. [En ligne]. Available: https://www.tensorflow.org/versions/r0.11/tutorials/mnist/beginners/. [Accès le 18 4 2017].

[15] davidshen84, «Implement MLP in tensorflow,» Stack Overflow, 29 1 2016. [En ligne]. Available: http://stackoverflow.com/questions/35078027/implement-mlp-in-tensorflow#36386287. [Accès le 18 4 2017].

[16] «Support Vector Machines,» Scikit Learn, 2016. [En ligne]. Available: http://scikit-learn.org/stable/modules/svm.html. [Accès le 18 4 2017].

[17] A. Rosebrock, «LeNet – Convolutional Neural Network in Python,» pyimagesearch, 1 8 2016. [En ligne]. Available: http://www.pyimagesearch.com/2016/08/01/lenet-convolutional-neural-network-in-python/. [Accès le 18 4 2017].

B. Annexes : Programmes réalisés

1. Code python pour augmenter le nombre de données

```
from __future__ import division
from matplotlib import pyplot as plt
from matplotlib import colors
import time, re, cv2, numpy as np
from os import listdir
from os.path import isfile, join
from random import randint

debut = time.time()

for i in range (0,43):
    i= "%02d" % (i)
    mypath = '/media/remy/TOURO/Images/000' + i
    onlyfiles = [f for f in listdir(mypath) if isfile(join(mypath, f))]
    taille=len(onlyfiles)
    #print taille
    dim = 2250-taille
    j=0
    while dim != 0 :
        j=(randint(0, len(onlyfiles)-1))
        img = cv2.imread(join(mypath, onlyfiles[j]))
        image=img
        #flip the image
        k=(randint(0,3))
        if k==0:
            img = img
        elif k==1:
            img =cv2.flip( img, 0 )
        elif k == 2:
            img =cv2.flip( img, 1 )
        elif k == 3:
            img =cv2.flip( img, -1 )
        #cv2.imshow("Original", img)
        #cv2.waitKey(0)

        # rotate the image by k degrees
        k=(randint(-15,15))
        (h, w) = img.shape[:2]
        center = (w / 2, h / 2)
        M = cv2.getRotationMatrix2D(center, k, 1.0)
        img = cv2.warpAffine(img, M, (w, h))
        numero = "%05d" % (dim)
        name=mypath+'/'+onlyfiles[j][:11]+'_rev_'+str(numero)
        cv2.imwrite(name+'.ppm', img)
        # cv2.imshow("Original", img)
        # cv2.waitKey(0)

        '''''plt.subplot(121), plt.imshow(image, cmap='gray')
        plt.title('Original'), plt.xticks([]), plt.yticks([])
        plt.subplot(122), plt.imshow(img, cmap='gray')
        plt.title(name), plt.xticks([]), plt.yticks([])
        plt.show()'''

        print i, dim

        dim-=1
```

2. Code pour la lecture et le prétraitement des données

```
mylist1 = []
today = datetime.date.today ()
mylist1.append (today)
debut = time.time ()

################################################################################
mypath = '/home/remy/SYS843/data'
#mypath = '/home/remy/SYS843/data_full'
onlyfiles = [f for f in listdir(mypath) if isfile(join(mypath, f))]
print len(onlyfiles)

dim = len(onlyfiles)
#dim = 1000

data = np.empty([dim, 784], dtype=np.int16)
label = np.empty(dim, dtype=np.int16)

# Rentre les datas dans la bonne forme
for a in range(0, dim):
    image = cv2.imread(join(mypath, onlyfiles[a]), 0)
    name = onlyfiles[a]
    labels = re.search('000(.+?)_', name)
    if labels:
        found = labels.group(1)
    label[a] = found

    image = cv2.resize(image, (28, 28))
    # height, width = image.shape[:2]
    # plt.subplot(121), plt.imshow(image, cmap='gray')
    # plt.title('Original'), plt.xticks([]), plt.yticks([])
    # plt.show()
    # A = np.asarray(image)
    # A=A[np.newaxis,np.newaxis,:,:]
    # trainData[0,:,:,:]=A[0,0,:,:]

    for b in range(0, 28):
        for c in range(0, 28):
            data[a, b + c * 28] = image[b, c]
            c += 1
        b += 1
    adv = a / dim * 100
    if (a) % 1000 == 0:
        print ("[INFO] Reading data: ") + str(a) + " /" + str(dim)
    a += 1
# print label

dataset = (data, label)
data = dataset[0].reshape((dataset[0].shape[0], 28, 28))
data = data[:, np.newaxis, :, :]

(trainData, testData, trainLabels, testLabels) = train_test_split(data / 255.0, dataset[1]
.astype("int"),test_size=0.33)

testlabel=testLabels
trainLabels = np_utils.to_categorical(trainLabels, 43)
testLabels = np_utils.to_categorical(testLabels, 43)

fin = (time.time() - debut) / 60
print("[TIME] Reading data: {:.2f}".format(fin)), " min"
################################################################################
```

3. Code pour le MLP 784-500-43

```
# ---------------------------------------- Create model ----------------------
#input layer
x0 = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 43])

#hidden layer
hidden_node=10
W1 = tf.Variable(tf.random_normal([784, hidden_node]))
b1 = tf.Variable(tf.random_normal([hidden_node]))
x1=tf.sigmoid(tf.matmul(x0, W1) + b1)

#output layer
W2 = tf.Variable (tf.random_normal ([784, 43]))  # essai deuxieme couche
b2 = tf.Variable (tf.random_normal ([43]))
y = tf.nn.softmax (tf.matmul (x0, W2) + b2)  # attention indice

# ---------------------------------------- Training --------------------------
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
train_step = tf.train.GradientDescentOptimizer(0.1).minimize(cross_entropy)
# initialise toutes les variables
init = tf.initialize_all_variables()
# definit la session
sess = tf.Session()
sess.run(init)

# ---------------------------------------- Evaluation ------------------------
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print(sess.run(accuracy, feed_dict={x0: testData, y_: testLabels}))

batch=100 #batchsize
epoch=200
for i in range(epoch):
    idx = np.arange(0, len(trainData))  # get all possible indexes
    np.random.shuffle(idx)  # shuffle indexes
    idx = idx[0:batch]  # use only `num` random indexes
    for j in idx:
        batch_xs = [trainData[j]]  # get list of `num` random samples
        batch_xs = np.asarray(batch_xs)  # get back numpy array
        batch_ys = [trainLabels[j]]  # get list of `num` random samples
        batch_ys = np.asarray(batch_ys)  # get back numpy array

    sess.run(train_step, feed_dict={x0: batch_xs, y_: batch_ys})
    print("[EPOCH "+str(i+1)+" /"+str(epoch)+")] {:.2f}%".format((sess.run(accuracy, feed_
dict={x0: testData, y_: testLabels})) * 100))

prediction=np.empty(len(testLabels), dtype=np.int16)
prediction=(sess.run(tf.argmax(y,1), feed_dict={x0: testData, y_: testLabels}))
# ----------------------------------------------------------------------------
```

4. Code pour le SVM linéaire

```
1.  ########################################################## TRAINING ######################
2.  print "training..."
3.  cc=1.
4.  Gamma=0.1
5.
6.  clf = svm.SVC(C=cc, gamma=Gamma, kernel='linear')
7.
8.  print "Training..."
9.  clf.fit(X_train, y_train)
10. fin1 = (time.time() - debut)/60
11. print("[TIME] Training: {:.2f}".format(fin1)), " min"
12.
13. ########################################################## TEST ######################
14. #print(clf.predict(X_test))
15. accuracy=0
16. #rempli la matrice de confusion
17. prediction=np.empty(len(y_test), dtype=np.int16)
18. for i in range(0, len(y_test)) :
19.     # classify the digit
20.     prediction[i] = clf.predict(X_test[np.newaxis, i])
21.     if prediction[i] == y_test[i] :
22.         accuracy=accuracy+1
23. fin = (time.time() - debut) / 60-fin1
24. print("[TIME] Testing: {:.2f}".format(fin)), " min"
25. ##########################################################################################
```

5. Code pour le CNN LeNet5

```
class LeNet :
    @staticmethod
    def build(width, height, depth, classes, weightsPath=None):
        # initialize the model
        model = Sequential()

        model.add(Convolution2D(20, (5, 5), padding="same", input_shape=(depth, height, width)))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(strides=(2, 2), pool_size=(2, 2), data_format="channels_first"))

        model.add(Convolution2D(50, (5, 5), padding="same"))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), data_format="channels_first"))

        model.add(Flatten())
        model.add(Dense(2000, activation="relu"))

        model.add(Dense(classes))
        model.add(Activation("softmax"))

        if weightsPath is not None:
            model.load_weights(weightsPath)
        return model

# permet d'enregistrer et de charger les poids d'un modele
ap = argparse.ArgumentParser()
ap.add_argument("-s", "--save-model", type=int, default=-1,help="(optional) whether or not model should be saved to disk")
ap.add_argument("-l", "--load-model", type=int, default=-1,help="(optional) whether or not pre-trained model should be loaded")
ap.add_argument("-w", "--weights", type=str, help="(optional) path to weights file")
args = vars(ap.parse_args())

# ------------------------------------------------------------------------------
# initialize optimizer and model
lr=0.1
batch=100
epoch=10
#-------------------------------------------------------------------------------

print("[INFO] compiling model...")
opt = SGD(lr=lr)
model = LeNet.build(width=28, height=28, depth=1, classes=43,weightsPath=args["weights"] if args["load_model"] > 0 else None)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])


# only train and evaluate the model if we *are not* loading a pre-existing model
if args["load_model"] < 0:
    print("[INFO] training...")
    model.fit(trainData, trainLabels, batch_size=batch, epochs=epoch, verbose=1)

# show the accuracy on the testing set
print("[INFO] evaluating...")
(loss, accuracy) = model.evaluate(testData, testLabels, batch_size=128, verbose=1)
print("[INFO] accuracy: {:.2f}%".format(accuracy * 100))

# check to see if the model should be saved to file
if args["save_model"] > 0:
    print("[INFO] dumping weights to file...")
    model.save_weights(args["weights"], overwrite=True)
```

6. Code pour le CNN VGG16 simplifié

```
1.  class VGG :
2.      @staticmethod
3.      def build(width, height, depth, classes, weightsPath=None):
4.          # initialize the model
5.          model = Sequential()
6.          model.add(Convolution2D(64, (3, 3), activation='relu', padding="same", input_shape=(depth, he
    ight, width)))
7.          #model.add(Convolution2D(64, (3, 3), activation='relu', padding="same"))
8.          model.add(MaxPooling2D((2, 2), strides=(2, 2), data_format="channels_first"))
9.
10.         model.add(Convolution2D(128, (3, 3), activation='relu', padding="same"))
11.         #model.add(Convolution2D(128, (3, 3), activation='relu', padding="same"))
12.         model.add(MaxPooling2D((2, 2), strides=(2, 2), data_format="channels_first"))
13.
14.         model.add(Convolution2D(256, (3, 3), activation='relu', padding="same"))
15.         #model.add(Convolution2D(256, (3, 3), activation='relu', padding="same"))
16.         #model.add(Convolution2D(256, (3, 3), activation='relu', padding="same"))
17.         model.add(MaxPooling2D((2, 2), strides=(2, 2), data_format="channels_first"))
18.
19.         model.add(Convolution2D(512, (3, 3), activation='relu', padding="same"))
20.         #model.add(Convolution2D(512, (3, 3), activation='relu', padding="same"))
21.         #model.add(Convolution2D(512, (3, 3), activation='relu', padding="same"))
22.         model.add(MaxPooling2D((2, 2), strides=(2, 2), data_format="channels_first"))
23.
24.         model.add(Convolution2D(512, (3, 3), activation='relu', padding="same"))
25.         #model.add(Convolution2D(512, (3, 3), activation='relu', padding="same"))
26.         #model.add(Convolution2D(512, (3, 3), activation='relu', padding="same"))
27.         model.add(ZeroPadding2D([1,1]))
28.         model.add(MaxPooling2D(pool_size=(3, 3), strides=(3, 3), data_format="channels_first"))
29.
30.         model.add(Flatten())
31.         model.add(Dense(4096, activation='relu'))
32.         #model.add(Dropout(0.5))
33.         model.add(Dense(4096, activation='relu'))
34.         #model.add(Dropout(0.5))
35.         model.add(Dense(classes, activation='softmax'))
36.
37.         # Si on a deja entrainer le reseau
38.         # if a weights path is supplied (inicating that the model was
39.         # pre-trained), then load the weights
40.         if weightsPath is not None:
41.             model.load_weights(weightsPath)
42.
43.         # return the constructed network architecture
44.         return model
45.
46.
47. # permet d'enregistrer et de charger les poids d'un modele
48. ap = argparse.ArgumentParser()
49. ap.add_argument("-s", "--save-model", type=int, default=-
    1,help="(optional) whether or not model should be saved to disk")
50. ap.add_argument("-l", "--load-model", type=int, default=-1,help="(optional) whether or not pre-
    trained model should be loaded")
51. ap.add_argument("-w", "--weights", type=str, help="(optional) path to weights file")
52. args = vars(ap.parse_args())
53.
54.
55. # -----------------------------------------------------------------------------------
56. # initialize the optimizer and model
57. lr=0.1
58. batch=100
59. epoch=10
```

```
#-------------------------------------------------------------------------------------

print("[INFO] compiling model...")
opt = SGD(lr=lr)
model = VGG.build(width=28, height=28, depth=1, classes=43,weightsPath=args["weights"] if args["load_
model"] > 0 else None)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])

# only train and evaluate the model if we *are not* loading a
# pre-existing model
if args["load_model"] < 0:
    print("[INFO] training...")
    model.fit(trainData, trainLabels, batch_size=batch, epochs=epoch, verbose=1)

# show the accuracy on the testing set
print("[INFO] evaluating...")
(loss, accuracy) = model.evaluate(testData, testLabels, batch_size=128, verbose=1)
print("[INFO] accuracy: {:.2f}%".format(accuracy * 100))

# check to see if the model should be saved to file
if args["save_model"] > 0:
    print("[INFO] dumping weights to file...")
    model.save_weights(args["weights"], overwrite=True)
```

7. Code pour enregistrer la matrice de confusion

```
#rempli la matrice de confusion
print ("[INFO] Confusion matrix : Wait a moment...")
prediction=np.empty (len (testLabels), dtype=np.int16)
for i in range (0, len (testLabels)) :
    # classify the digit
    proba = model.predict(testData[np.newaxis, i])
    prediction[i] = proba.argmax(axis=1)
cm = metrics.confusion_matrix(testlabel,prediction)


def plot_confusion_matrix(cm, classes,normalize=False,title='Confusion matrix',cmap=plt.cm
      .Greys):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks,  rotation=90)
    plt.yticks(tick_marks)

    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    #print("Normalized confusion matrix")
    #print(cm)

    thresh = cm.max() / 2.

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Compute confusion matrix
#np.set_printoptions(precision=2)
class_names = [i for i in range(0,42)]
#print class_names
# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cm, classes=class_names, normalize=True,title='Normalized confusion
      matrix')
plt.savefig('/home/remy/SYS843/L_'+str(lr)+'_'+str(batch)+'_'+str(epoch)+'_Matrice.png', b
      box_inches='tight')
```

8. Code pour enregistrer le ROC et afficher les résultats

```
1.  ################################### ROC ##################################
2.  # Compute ROC curve and ROC area for each class
3.  fpr = dict()
4.  tpr = dict()
5.  roc_auc = dict()
6.
7.  testlabel = np_utils.to_categorical(testlabel, 43)
8.  prediction = np_utils.to_categorical(prediction, 43)
9.
10. for i in range(0,43):
11.     fpr[i], tpr[i], _ = roc_curve(testlabel[:, i], prediction[:, i])
12.     roc_auc[i] = auc(fpr[i], tpr[i])
13.
14.
15. # Compute micro-average ROC curve and ROC area
16. fpr["micro"], tpr["micro"], _ = roc_curve(testlabel.ravel(), prediction.ravel())
17. roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])
18.
19.
20. lw=2
21.
22. plt.figure()
23. plt.plot(fpr["micro"], tpr["micro"],label='micro-
        average ROC curve (area = {0:0.2f})'''.format(roc_auc["micro"]),
24.          color='darkorange', linewidth=2)
25. plt.plot([0, 1], [0, 1], 'k--', lw=lw)
26. plt.xlim([0.0, 1.0])
27. plt.ylim([0.0, 1.05])
28. plt.xlabel('False Positive Rate')
29. plt.ylabel('True Positive Rate')
30. plt.title('Some extension of Receiver operating characteristic to multi-class')
31. plt.legend(loc="lower right")
32. plt.savefig('/home/remy/SYS843/L_'+str(lr)+'_'+str(batch)+'_'+str(epoch)+'_ROC.png', bbox_
        inches='tight')
33.
34. for i in range(0,43):
35.     plt.plot(fpr[i], tpr[i],  lw=lw)#,label='ROC {0} (area = {1:0.2f})'''.format(i, roc_au
        c[i]))
36. plt.plot([0, 1], [0, 1], 'k--', lw=lw)
37. plt.xlim([0.0, 1.0])
38. plt.ylim([0.0, 1.05])
39. plt.xlabel('False Positive Rate')
40. plt.ylabel('True Positive Rate')
41. plt.title('Some extension of Receiver operating characteristic to multi-class')
42. plt.legend(loc="lower right")
43. plt.savefig('/home/remy/SYS843/L_'+str(lr)+'_'+str(batch)+'_'+str(epoch)+'_ROCfull.png', b
        box_inches='tight')
44.
45.
46. ################################### Resultats ##################################
47.
48. print("[RESULTAT] accuracy: {:.2f}%".format(accuracy * 100))
49. print"[RESULTAT] lr=", lr
50. print"[RESULTAT] batch=", batch
51. print "[RESULTAT] epochs=", epoch
52. print mylist1[0]
53. fin = (time.time() - debut)/60
54. print("[RESULTAT] temps: {:.2f}".format(fin)), " min"
```