Table des matières

Liste des tableaux

Liste des figures

1. Introduction

Depuis quelques années, les systèmes d'intelligence artificielle ont vu une accélération de leur développement. Les avancées rapides dans ce milieu ont créé une peur chez les populations(Boyd, Wilson, 2017). Et si ces systèmes pouvaient être utilisés pour protéger les personnes ?

Premièrement, nous allons étudier le fonctionnement d'un réseau de neurones artificiels, comment entrainer un système de *machine learning* et les bonnes pratiques à utiliser.

Deuxièmement, nous allons comprendre comment utiliser la librairie de *machine learning* TensorFlow. Pour cela, nous commencerons par analyser la structure de la librairie, pour enfin pouvoir mettre en pratique les concepts appris lors du premier chapitre avec un cas simple.

Enfin, grâce aux deux chapitres, théorique et pratique, nous pourrons entrainer notre propre réseau de neurones et l'appliquer à un cas réel.

Le prototype qui sera créé utilisera des réseaux de neurones artificiels appliqués aux images de caméras de surveillance tout en gardant l'anonymat des personnes. Ainsi, ces systèmes de *machine learning* sont utilisés pour protéger les personnes. Aussi, les entités responsables possédant des caméras ou autres manières de récolter des informations non exploitables à cause des problèmes d'anonymat, pourraient enfin les exploiter et en obtenir des statistiques étant donné que les images ne seraient pas stockées mais seulement analysées.

2. Réseaux de neurones

2.1 Machine learning

Le *machine learning,* apprentissage automatique ou apprentissage statistique en français, est une technologie d'intelligence artificielle basée sur le fait que la machine peut apprendre toute seule en se basant sur des modèles statistiques permettant d'effectuer des analyses prédictives.(Bastien L, 2018).

Il existe plusieurs modes d'apprentissage, comme par exemple : l'apprentissage supervisé, l'apprentissage non-supervisé et l'apprentissage par renforcement.(Chaouche, 2018).

Dans le cas de l'apprentissage supervisé, nous devons fournir des données à la machine et nous devons lui donner la réponse attendue, aidant la machine à faire les bonnes prédictions(Chaouche, 2018).

Ensuite, dans le cas de l'apprentissage non-supervisé, nous n'indiquons pas à la machine ce que nous voulons comme sortie, la machine va tout simplement regrouper les données par similitude(Chaouche, 2018).

Et dans le cas de l'apprentissage par renforcement, la machine va apprendre par récompense. C'est-à-dire, lorsque celle-ci améliore sa performance, elle reçoit une récompense et ceci va donc augmenter les chances que ce comportement se reproduise(Chaouche, 2018).

Dans le cadre de ce travail de bachelor, c'est l'apprentissage supervisé qui sera expliqué et analysé dans le but de créer un modèle statistique qui nous permette de classifier correctement des données.

Le concept d'apprentissage automatique existe depuis les années 1950 mais, aujourd'hui les systèmes d'apprentissage automatique sont devenus plus complexes, la puissance de calcul a augmenté manière exponentielle (Moore, 1965) et avec Internet, la quantité de données à disposition est astronomique(Schultz, 2017). Ce sont quelques raisons qui ont fait en sorte que ces systèmes soient en vogue depuis quelques années(Hodjat et al., 2015).

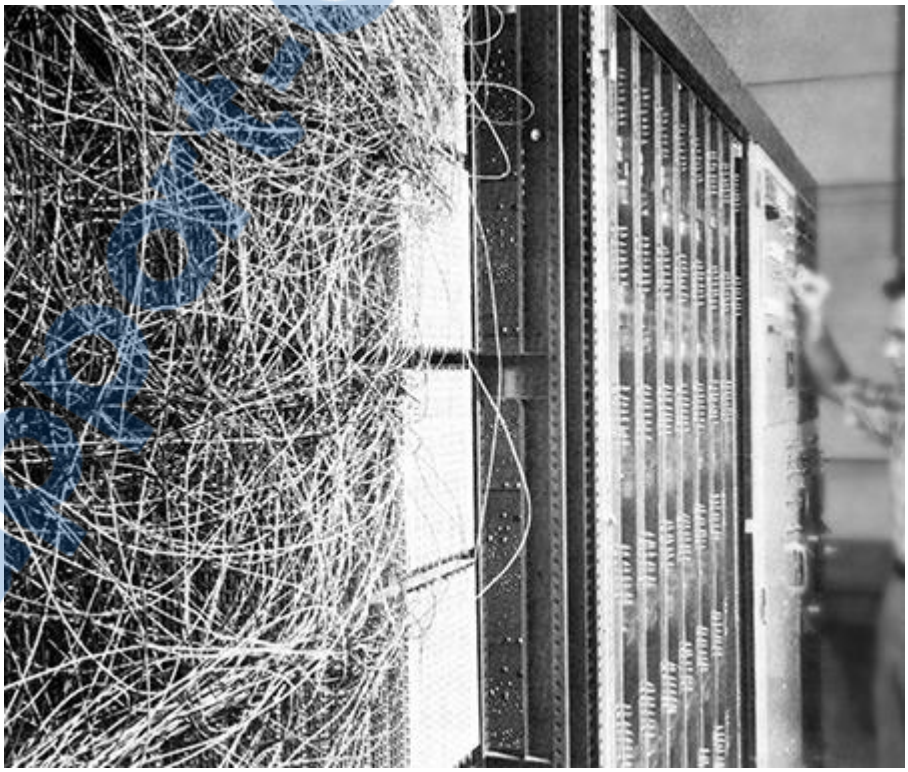
Afin de bien comprendre le fonctionnement de ceux-ci, nous devons d'abord comprendre l'un des premiers concepts de neurones artificiels : le perceptron.

2.2 Perceptron

Le perceptron est un algorithme d'apprentissage automatique supervisé de classification binaire. Il s'agit du type de réseau de neurones artificiels le plus simple. Les perceptrons, aussi communément appelés neurones artificiels, ont été créés par Frank Rosenblatt en 1958(Brain, Rosenblatt, 1958). Son créateur l'a comparé à des neurones capables de répondre à des stimuli externes d'une manière qui imite les vrais neurones biologiques dans sa publication « The perceptron : a probabilistic model for information storage and organization in the brain » (Brain, Rosenblatt, 1958)

A l'époque, une machine nommée Mark 1 a été spécialement conçue pour implémenter l'algorithme. La machine était connectée à une caméra de 20 pixels par 20 pixels qui transformait l'image en un tableau de 400 pixels et son objectif était donc d'effectuer de la reconnaissance d'image(L, 2018). Lors de la création de la machine, les chercheurs la voyaient comme une avancée extraordinaire dans le domaine de l'intelligence artificielle. Cependant, les limites techniques ne leur auront pas permis d'exploiter tout le potentiel de cette technologie.

Figure 1 : Mark 1, le premier perceptron du monde

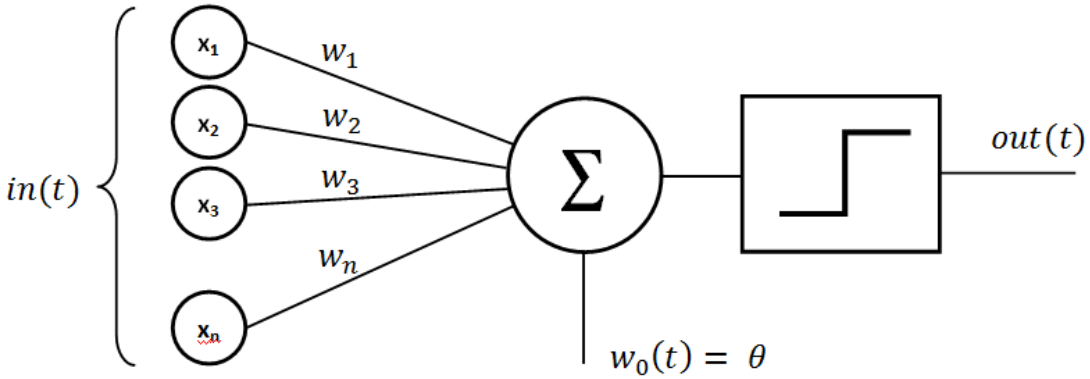


(Tadeusiewicz, 2018)

2.2.1 Structure

La structure d'un perceptron est très simple (voir Figure 2) :

Figure 2 : Représentation graphique d'un perceptron



(Rajarajan, 2015)

Comme nous pouvons le constater, sa structure est composée de 6 parties (Janecek, 2007):

- Les entrées (x1, x2, x3, …, xn)
- Les poids (w1, w2, w3, …, wn)
- Une fonction de somme (∑)
- Un biais (w0)
- Une fonction d'activation
- Une sortie

Les entrées sont les données que l'on fournit au perceptron afin qu'il calcule un résultat. Ces données doivent être des mesures en lien avec le cas étudié. Dans le cadre de ce travail de bachelor, les mesures sont des pixels. Dans le cas de Iris de Fischer(Wikipedia, 2018), un jeu de données très connu et qui est utilisé dans l'introduction à l'apprentissage automatique, l'objectif est de classifier des fleurs dans plusieurs catégories selon la taille des pétales.

Étant donné que les données utilisées peuvent varier, on peut avoir certains problèmes de confiance avec les résultats obtenus. En effet, si nous imaginons qu'une entrée peut avoir des valeurs qui varient entre 0 et 1 et qu'une deuxième entrée peut avoir des valeurs qui varient entre 0 et 3000, alors la deuxième entrée va avoir un impact beaucoup plus important dans le résultat calculé.

Afin de remédier à cela, il est impératif de normaliser les données en entrée(Jordan., 2018). Nous verrons plus tard que la normalisation peut avoir un impact sur l'apprentissage.

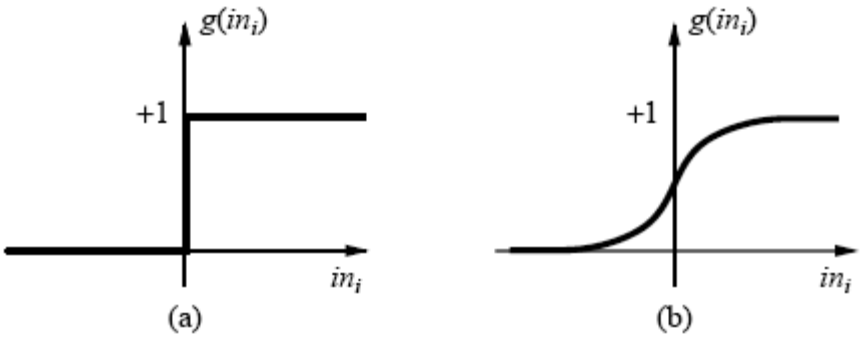
Les poids sont des valeurs que l'on donne à chaque connexion entre les entrées et la fonction de somme, et ils sont modifiés à chaque itération d'apprentissage (2.2.2 Phase d'apprentissage).

La fonction de somme pondérée va faire la somme de la multiplication de chaque entrée par son poids correspondant et va ajouter le biais, qui est une valeur constante. Avant de pouvoir utiliser le résultat, la somme est passée dans une fonction d'activation(Janecek, 2007).

Il existe deux fonctions souvent utilisées (Voir Figure 3) :

- Fonction d'activation à seuil
- Fonction sigmoïde

Figure 3 : Graphiques de fonctions d'activation



(Ali et al., 2006)

La fonction d'activation à seuil (à gauche) est une fonction qui pour $x < 0$ alors $y = 0$. Pour $x >= 0$, alors $y = 1$. Nous pouvons l'utiliser pour des cas où il est nécessaire d'avoir une réponse binaire : oui ou non(Sharma, 2017).

La fonction d'activation sigmoïde (à droite) pour une valeur x quelconque, y reste entre 0 et 1. Ce cas est très utile lorsque nous avons besoin d'avoir un résultat continu(Sharma, 2017).

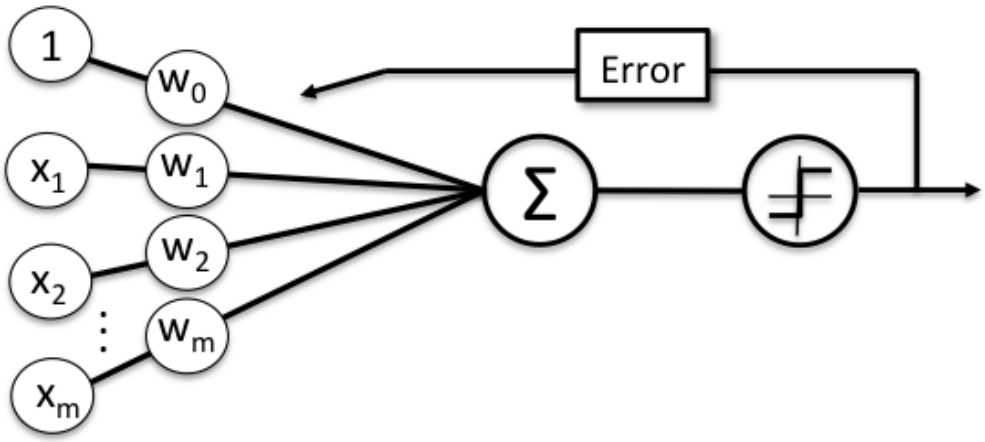
Après la fonction d'activation, nous pouvons consulter le résultat donné par le perceptron. Le problème est que celui-ci n'est pas encore entraîné et il est donc nécessaire de trouver un moyen de lui indiquer comment s'adapter afin de donner la bonne réponse attendue.

2.2.2 Phase d'apprentissage

La phase d'apprentissage peut être faite de plusieurs façons, comme vu dans le chapitre « 2.1 Machine learning ». Ici sera seulement explicité le mode d'apprentissage supervisé.

Cette technique consiste à envoyer des données au perceptron et à analyser le résultat. Ensuite, on indique au perceptron quel était le résultat attendu. Par exemple, si on obtient la valeur 0.3 mais que la valeur attendue était de 1, alors l'algorithme d'apprentissage du perceptron détecte qu'il a fait une erreur de $1 - 0.3 = 0.7$. Cette valeur peut être vue comme une fonction de coût et le perceptron doit s'adapter afin de réduire le coût au minimum possible. Pour cela, nous utilisons un algorithme appelé rétropropagation qui a pour but de changer les poids de chaque connexion(Rojas, 1996).

Figure 4 : Représentation du fonctionnement de l'algorithme de rétropropagation



(USBDATA, [sans date])

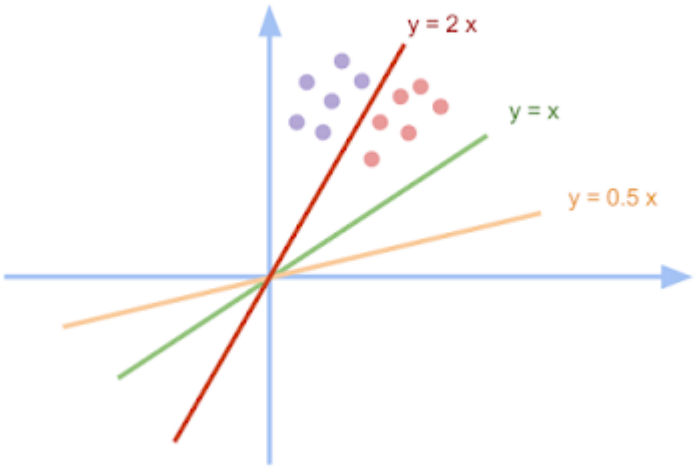
Ce processus d'apprentissage peut être répété plusieurs fois jusqu'à ce que le perceptron nous donne le bon résultat. C'est à ce moment que le biais devient très important.

Si toutes les entrées sont à 0, alors peu importe la valeur des poids donnés à chaque connexion, la valeur en sortie sera toujours 0. Dans ce cas, nous pouvons ajouter le biais qui nous permet de modifier la sortie afin que l'apprentissage se fasse correctement(Logo, 2013).

2.2.2.1 Le biais

Afin de mieux comprendre le rôle des biais, prenons l'exemple d'une fonction linéaire $y = A * x$, pour essayer de séparer deux ensembles de données (bleu et rouge) (Figure 5).

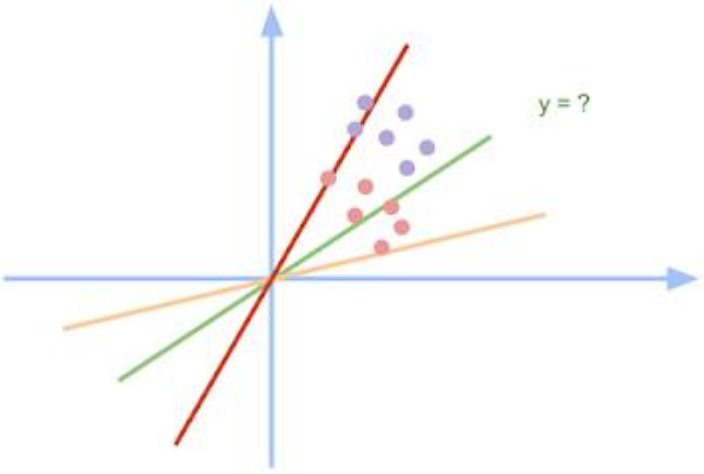
Figure 5 : Séparation d'un ensemble de données à l'aide d'une fonction linéaire



(Myo, 2016)

Comme nous pouvons le constater, la fonction $y = 2x$ sépare parfaitement les deux ensembles. Cependant, la limitation de ces fonctions se présente lorsque les ensembles de données sont moins faciles à séparer (Figure 6).

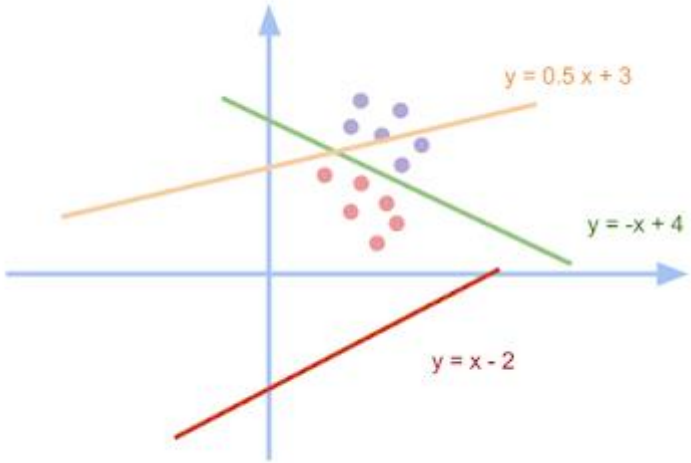
Figure 6 : Séparation impossible à l'aide de fonction linéaire



(Myo, 2016)

Nous pouvons voir qu'aucune fonction ne permet de séparer les ensembles correctement. De cette façon, nous introduisons le biais (Figure 7).

Figure 7 : Séparation de deux ensembles de données à l'aide d'une fonction linéaire et un biais



(Myo, 2016)

Ici, la fonction utilisée est $y = A * x + B$ où $+B$ correspond au biais. Grâce à ce biais, nous avons pu trouver la bonne fonction linéaire $y = -x + 4$ qui sépare correctement les deux ensembles.

2.2.2.2 Surentraînement

La phase d'entraînement peut s'avérer délicate. En effet, il est facile de surentraîner un perceptron, ce qui veut dire que celui-ci a mémorisé les données fournies et n'est pas assez générale pour être appliqué à d'autres données. Pour éviter ce problème, celui-ci doit passer une phase de tests(Geitgey, 2014).

Pour cela, nous devons avoir un jeu de données que l'on peut diviser en deux parties. Cette division peut se faire de plusieurs façons. Un exemple, c'est de prendre 80% des données comme entraînement et 20% de test(Shah, 2017).

La phase de tests sert à introduire de nouvelles données dans le réseau de neurones artificiels qui n'ont pas été utilisées dans la phase d'apprentissage(Ivanov, 2017), et avoir un pourcentage de réponses justes.

Si le pourcentage n'est pas satisfaisant, nous pouvons modifier la façon dont les données sont divisées. Par exemple, 70% pour la phase d'apprentissage et 30% pour la phase de test(Ivanov, 2017).

En revanche, pour savoir si le pourcentage est suffisant, il n'y a pas de règle universelle préconisée car cela dépend du domaine en question(Saitta, 2010). Il y a certains problèmes, comme la prédiction des valeurs des actions d'une entreprise, où il n'est pas possible de prendre en compte toutes les variables influençant les résultats. A l'inverse, dans le domaine de la reconnaissance d'image, les données que nous avons à traiter son uniquement des pixels. Par conséquent, il n'existe pas de variables externes influençant ce résultat(Saitta, 2010).

D'autres techniques qui influencent le surentrainement seront abordées dans le chapitre dédiés aux réseaux de neurones à convolution « 2.4.5 Apprentissage d'un réseau de neurones à convolution ».

2.2.2.3 Le sous-apprentissage

Un réseau de neurones artificiels peut mémoriser les données mais il peut aussi ne pas apprendre suffisamment pour être généralisable.

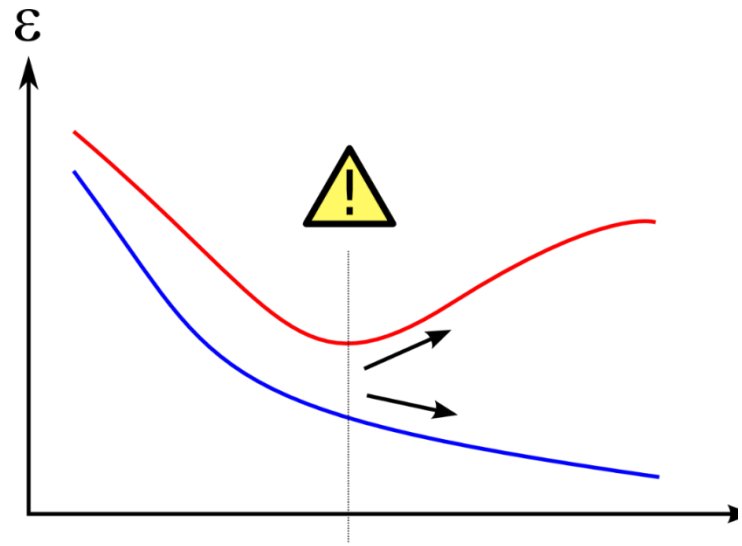
Afin d'éviter le sous-apprentissage, nous devons mélanger toutes les données avant la phase d'entraînement (Ivanov, 2017). En effet, si nous voulons classifier des chats et des chiens mais que tous les chats se trouvent au début de l'ensemble de données et que les chiens se trouvent à la fin, alors le réseau saura classifier des chats, mais il n'aura jamais reçu un chien en entrée. Sa précision sera alors mauvaise.

Aussi, il est important d'avoir un nombre équilibré de données pour chaque classe à prédire(Ivanov, 2017). Si nous possédons 20 fois plus d'images de chats que de chiens, alors il est possible que le réseau soit optimisé pour la classification de chats mais qu'il soit mauvais pour la classification de chiens.

2.2.2.4 Le meilleur modèle statistique

Le défi des *data scientists* est de pouvoir trouver le juste milieu entre le surentrainement et le sous-apprentissage(Benzaki, 2017).

Figure 8 : Représentation graphique du meilleur modèle statistique



(Benzaki, 2017)

A la Figure 8, la ligne bleue représente la fonction de coût pendant la phase d'apprentissage et la ligne rouge représente la fonction de coût pendant la phase de test. Le panneau de *warning* représente le meilleur modèle statistique. A gauche de celui-ci le modèle est en sous-apprentissage, à droite de celui-ci il est en surentrainement(Benzaki, 2017).

Pour atteindre le meilleur modèle statistique, nous pouvons créer un ensemble de validation pour appliquer une technique de *early stopping*(Early Stopping, 2018).

Un ensemble de validation est semblable à un ensemble de test mais qui est testé après un certain nombre d'itérations d'apprentissage, défini en fonction de la taille du jeu de données. Cet ensemble de validation est testé et l'apprentissage continue tant que la fonction de coût de l'ensemble de validation continue de réduire. Une fois que la fonction de coût commence à augmenter, nous arrêtons l'apprentissage(Early Stopping, 2018). Pour finir, nous testons le réseau de neurones artificiels avec l'ensemble de test pour s'assurer de la bonne généralisation de celui-ci.

Les règles d'adaptation à suivre pour l'ensemble de validation sont les mêmes que pour celui d'apprentissage et test – Voir « 2.2.2.2 Surentraînement ». Par exemple, 70% d'entrainement, 15% de validation et 15% de test.

Il existe plusieurs techniques pour vérifier si le réseau est en phase de surentrainement et que l'augmentation de la fonction de coût dans la phase de validation n'est pas qu'une variation. En effet, il se peut parfois que la fonction de coût augmente pour quelques itérations alors que le réseau de neurones artificiels n'est pas surentrainé. Toutefois, ces

techniques sortent du cadre de mon étude et ne seront pas abordées dans le cadre de ce travail de bachelor.

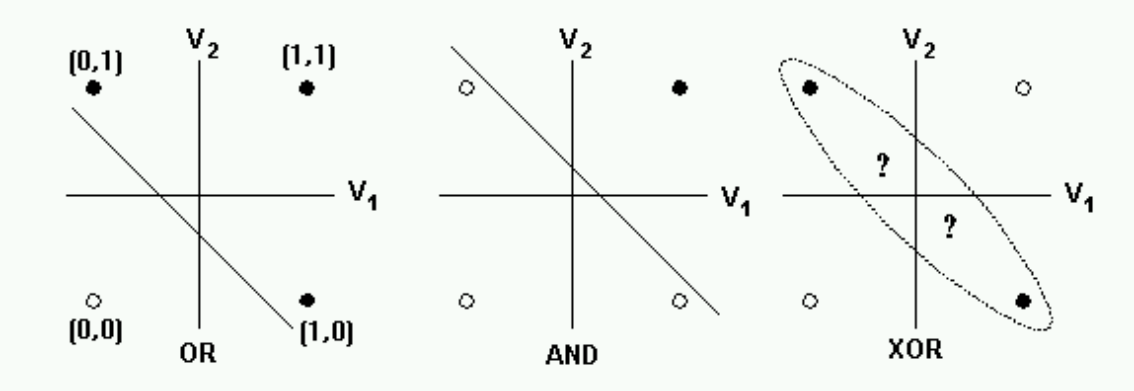
2.2.3 Le problème XOR

Si le perceptron peut prendre tout type de données en entrée et donner un résultat en sortie, alors pourquoi avons-nous besoin d'avoir des réseaux multicouches ?

La réponse à ce problème est que le perceptron ne peut résoudre que des problèmes linéaires et donc simples(Minsky, Papert, 1969). Malheureusement, souvent les problèmes traités par les algorithmes d'apprentissage automatique sont plus complexes.

Par exemple, dans un graphique si on représente les ports logiques OR et AND, il est très facile de tracer une ligne et séparer les résultats TRUE et FALSE. En revanche, avec le port logique XOR, cela devient impossible (voir Figure 9).

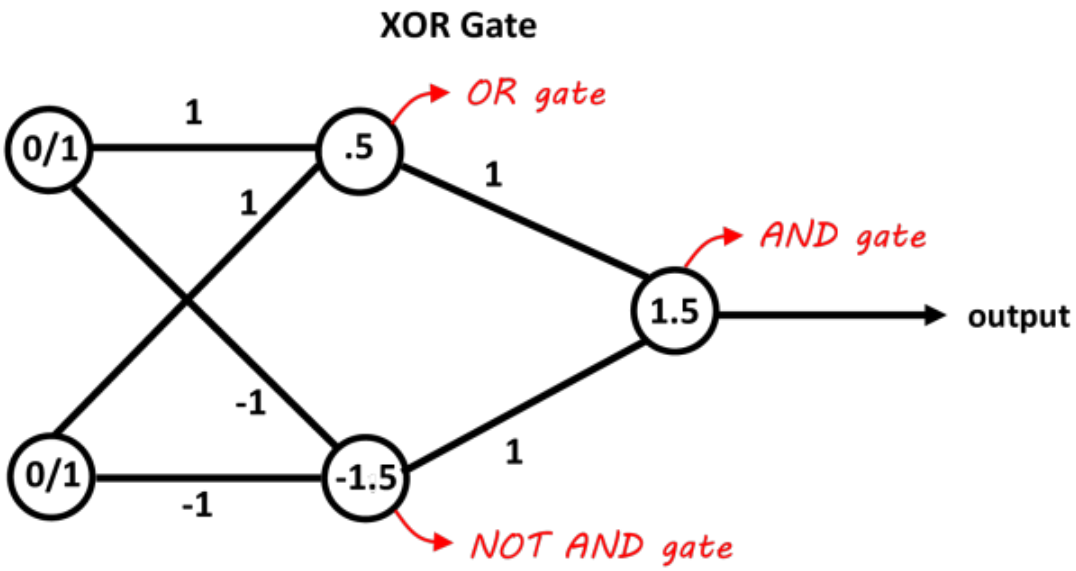
Figure 9 : Comment découper les résultats des ports logiques avec une seule ligne



(Beeman, 2001)

Afin d'implémenter une solution au problème XOR en utilisant des perceptrons, il faut s'inspirer de l'électronique. En effet, avec un port logique OR et NAND, en utilisant les résultats de deux ports et en les passant dans un port AND on obtient un port logique XOR(Electronics, 2013). Il suffit d'effectuer le même traitement avec les perceptrons (voir Figure 10).

Figure 10 : Implémentation d'un port logique XOR à l'aide de perceptrons



(Neel, 2015)

Cette solution a été proposée par Minsky et Papert en 1969 et c'est ainsi que les réseaux de neurones multicouches sont nés.

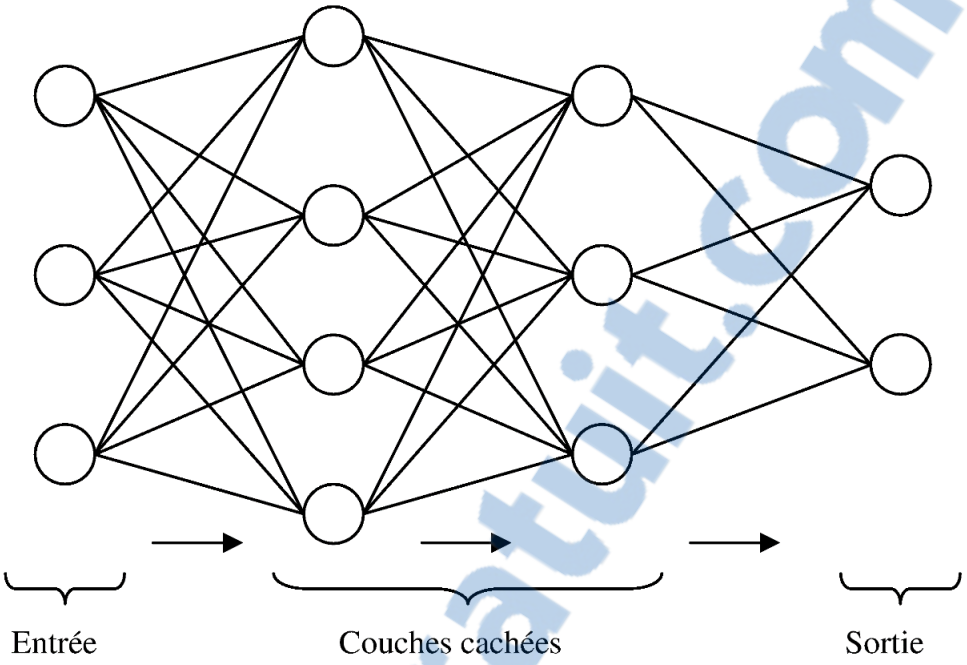
2.3 Les réseaux de neurones multicouches

Le principe du fonctionnement d'un perceptron et d'un réseau de neurones multicouches est essentiellement le même(Mestan, 2008). Nous avons toujours des entrées et des sorties mais nous avons des couches intermédiaires appelées couches cachées (voir Figure 11).

Chaque élément d'entrée est connecté à chaque élément de la couche suivante, et ainsi de suite jusqu'à la couche de sortie. Comme dans le perceptron, chaque connexion contient un poids qui lui est attribué.

Pour finir, chaque neurone contient aussi une fonction de somme, une fonction d'activation et un biais.

Figure 11 : Représentation graphique d'un réseau de neurones artificiels



(Delaunay, 2016)

2.3.1 L'architecture d'un réseau de neurones

Dans le cadre de ce travail de bachelor, l'architecture fait référence au nombre de couches et de neurones utilisés dans chaque couche.

Certains éléments d'un réseau de neurones sont faciles à définir. En effet, le nombre d'entrées dépend uniquement des données que vous pouvez fournir au réseau. Par exemple, si vous avez une image de 28 pixels par 28 pixels, vous avez besoin de 784 entrées car chaque pixel est considéré comme une valeur à analyser.

Le nombre de sorties reste facile à déterminer également, et celui-ci dépend du type de réponses que nous cherchons. Si nous avons des images et que l'on veut savoir si c'est un chat ou chien qui se trouve sur l'image, alors il suffit d'avoir deux éléments de sortie. Chaque neurone de sortie représente une classe. Si nous prenons un autre cas où nous essayons de classifier des chiffres, alors on va avoir 10 sorties : une sortie par chiffre.

L'élément structurel qui est le plus complexe ce sont les couches cachées. Il y a encore des débats concernant le bon nombre de couches et de nœuds par couche à utiliser.

2.3.2 Les couches cachées

Le théorème d'approximation universelle nous indique que tous les problèmes non-linéaires peuvent être résolus à l'aide d'un réseau de neurones ayant une seule couche

cachée avec un nombre fini de neurones(Heaton, 2017). Cependant, ce théorème n'explique pas comment le réseau pourrait apprendre tout type de problème. Ce théorème a été démontré par George Cybenko en 1989 (Cybenko, 1989) en utilisant des fonctions sigmoïdes et, en 1991, Kurt Hornik (HORNIK, 1991) a prouvé que c'est l'architecture elle-même des réseaux de neurones artificiels qui permet d'approximer toutes les fonctions(Heaton, 2017).

Or, depuis 1991, notre compréhension du fonctionnement des réseaux de neurones artificiels a changé. En utilisant une seule couche, nous pouvons donc approximer toute fonction, mais elle ne sera pas généralisable, d'où la nécessité d'utiliser plusieurs couches cachées. En 2006, Geoffrey Hinton a créé le *Deep Learning* (Hinton et al., 2006). Il a démontré qu'en utilisant plusieurs couches cachées, la précision des réseaux s'améliore dans les domaines de la reconnaissance d'image, de la reconnaissance vocale et de l'analyse de texte(Kate Allen, 2015).

Mais, comment définir l'architecture d'un réseau étant donné la multitude de configurations possibles de nombre de couches cachées et de neurones artificiels par couche à utiliser ?

2.3.2.1 Le nombre de couches à utiliser

Le nombre de couches cachées à utiliser dépend du problème à traiter. En effet, si le jeu de données à traiter est simple, deux couches cachées ou moins, sont souvent suffisantes(Heaton, 2017). Cependant, lorsque le problème et les données à traiter sont plus complexes, comme la reconnaissance d'image, il peut être utile d'utiliser des couches additionnelles(Heaton, 2017).

Pour décider si nous devons utiliser une ou deux couches cachées, nous pouvons utiliser une technique qui consiste à commencer avec une seule couche cachée et de l'adapter (Voir 2.3.2.2 Le nombre de nœuds à utiliser par couche) jusqu'à trouver une architecture satisfaisante. Si une seule couche cachée ne produit pas des résultats satisfaisants, alors nous devons passer à deux couches cachées.

Définir ce qu'est un problème simple n'est pas facile. Cependant, si le problème à traiter est lié à du traitement de texte ou à de la reconnaissance d'image, par exemple, il est recommandé de commencer avec plus de 2 couches(Heaton, 2017). Rien n'indique que tous les autres types de problèmes peuvent être résolus avec 2 couches cachées ou moins. Ces recommandations nous permettent juste d'avoir un point de départ pour tester plusieurs architectures.

2.3.2.2 Le nombre de nœuds à utiliser par couche

Choisir le bon nombre de nœuds à utiliser par couche est difficile et il s'agit d'un des éléments qui font que le réseau peut apprendre correctement ou non(Heaton, 2017). Si nous utilisons peu de neurones, alors il se peut que nous ayons des problèmes de sous-apprentissage(Heaton, 2017). A l'inverse, trop de neurones utilisés et nous pouvons avoir des problèmes de surentrainement(Heaton, 2017).

Il est donc nécessaire de trouver un juste milieu en ce qui concerne le nombre de nœuds des couches cachées. Jeff Heaton(Heaton, 2017) nous donne quelques conseils qui nous permettent d'avoir une base sur laquelle commencer pour ensuite adapter le réseau jusqu'à trouver une bonne configuration pour une seule couche cachée:

Le nombre de neurones d'une couche cachée doit rester entre le nombre de neurones de la couche d'entrée et du nombre de neurones de la couche de sortie(Heaton, 2017).

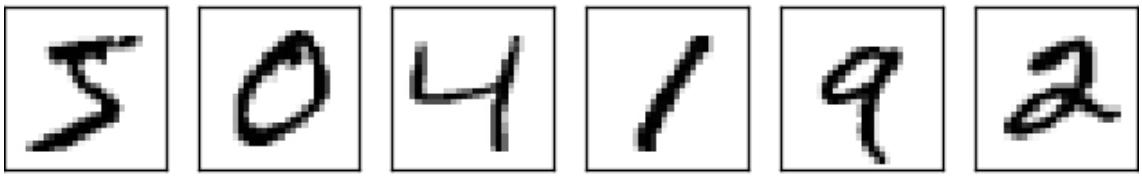
Finalement, afin de trouver la meilleure configuration, nous devons faire plusieurs tests car il n'y a pas de règle précise qui nous indique comment créer l'architecture d'un réseau à partir d'un problème donné(Heaton, 2017).

2.3.3 Application à la reconnaissance d'image

Lorsque nous voulons faire de la reconnaissance d'image avec un réseau de neurones, la procédure la plus facile est de transformer l'image en noir et blanc. Ensuite, nous utilisons chaque valeur d'intensité de chaque pixel qui correspond à une valeur, qui peut varier entre 0 et 255. Ces valeurs doivent être normalisées (2.2.1 Structure) et passées en entrée du réseau.

Pour exemple, nous allons utiliser le jeu de données MNIST, qui contient des chiffres écrits à la main(Nielsen, 2017). Les images sont déjà en noir et blanc et correspondent à la Figure 12 :

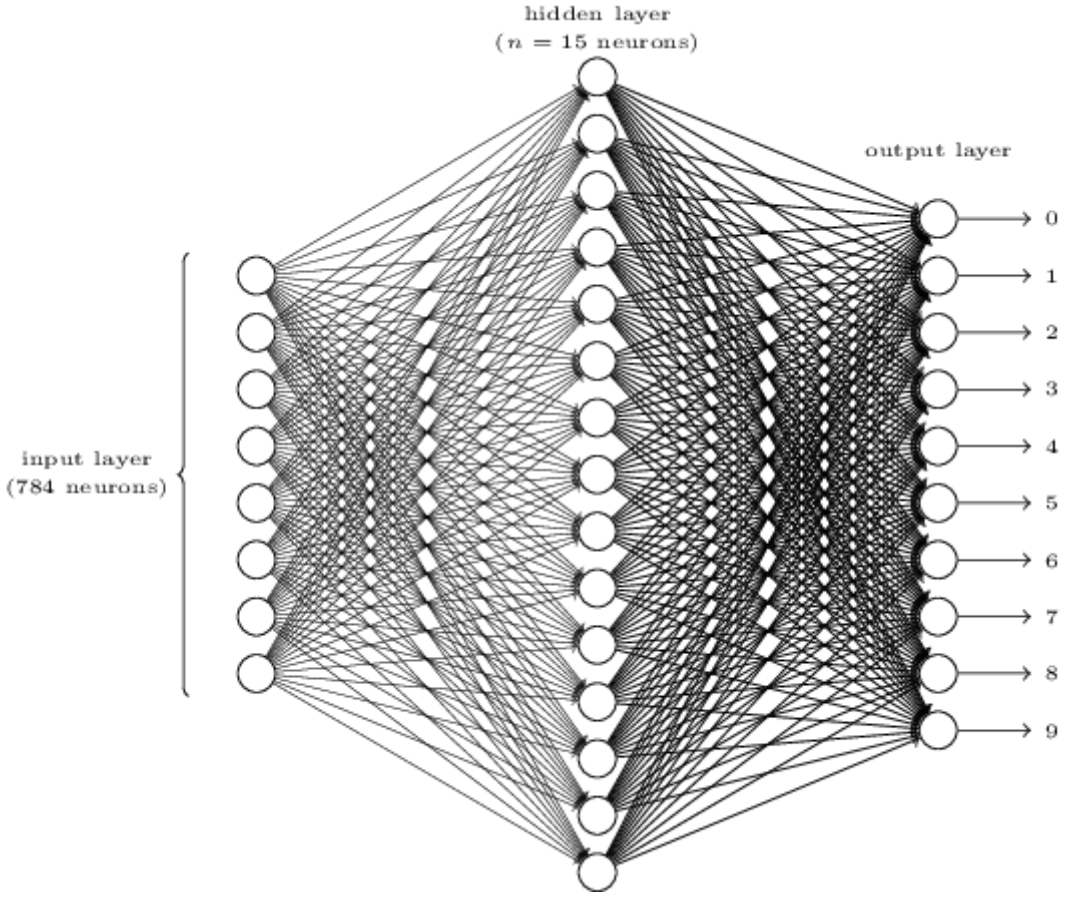
Figure 12 : Chiffres MNIST



(Ebert-Uphoff, Gil, 2015)

Chaque élément contient 28 pixels par 28 pixels ce qui correspond à 784 pixels et correspond aussi au nombre d'entrées dans le réseau d'exemple. Le nombre de sorties est donc 10, un pour chaque chiffre (voir Figure 13).

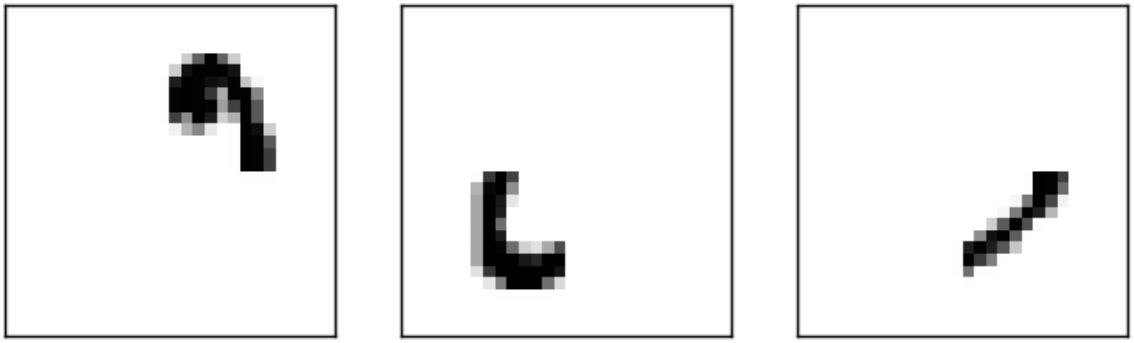
Figure 13 : Exemple de réseau de neurones pour reconnaissance d'image



(Ebert-Uphoff, Gil, 2015)

Afin d'essayer d'avoir une compréhension de ce que chaque neurone caché peut représenter, nous pouvons imaginer que chaque chiffre est décomposé en plusieurs parties (voir Figure 14) et que le neurone qui correspond à une partie aura une valeur élevée.

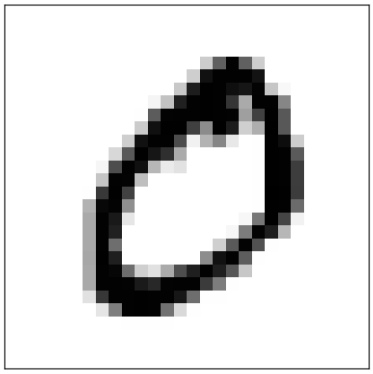
Figure 14 : Décomposition du chiffre 0



(Ebert-Uphoff, Gil, 2015)

Ces valeurs élevées auront pour effet d'activer le neurone de sortie qui correspond au chiffre 0.

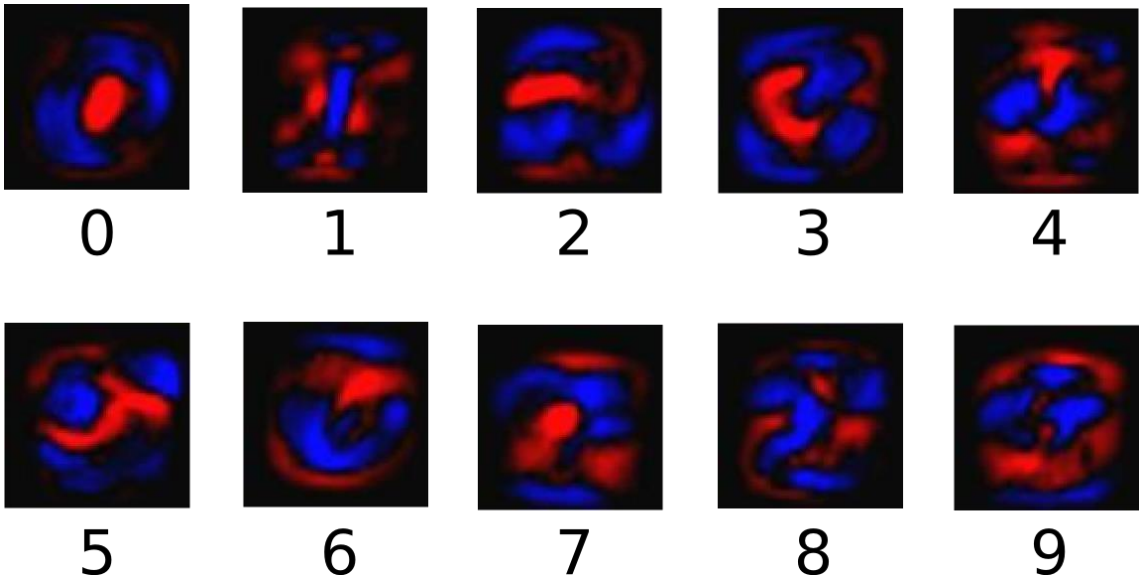
Figure 15 : Chiffre 0 complet



(Ebert-Uphoff, Gil, 2015)

Pour comprendre comment chaque neurone peut correspondre à une partie spécifique d'un chiffre, nous devons nous rappeler que chaque connexion possède un poids, et que chaque poids de la couche d'entrée est lié à un pixel. Ensuite, si on veut reconnaître une certaine partie de l'image, il suffit que les poids correspondants soient positifs et tous les autres négatifs(Google, 2017) (voir Figure 16).

Figure 16 : Poids représentant chaque chiffre



(TensorFlow, 2017)

Cet exemple a été tiré d'un tutoriel fait par Google pour les débutants sur TensorFlow. Les régions bleues représentent les poids positifs, et les rouges les négatifs.

Nous pouvons voir que la pondération de certains chiffres, comme le 0, le 1 et le 3 est facilement compréhensible par un être humain. Cependant, d'autres chiffres comme le 4, le 5 et 8 deviennent moins reconnaissables. Il est important de noter que malgré cela, la machine arrive tout de même à classifier la plupart des chiffres (selon Google, avec ces poids on peut atteindre 92% de précision) même si nous, les humains, ne comprenons pas toujours ce qui se passe de manière interne pour cette reconnaissance.

Ceci pose un problème fondamental dans l'utilisation des réseaux de neurones puisque dans certains cas, comme dans la médecine, il peut être nécessaire de pouvoir expliquer pourquoi la machine a fait une telle prédiction(Charpentier, 2018). Or, dans les réseaux de neurones artificiels, nous ne comprenons pas toujours pourquoi certaines prédictions sont faites.

2.3.3.1 Les difficultés d'un réseau de neurones pour la reconnaissance d'image

2.3.3.1.1 Le nombre de poids :

Étant donné que chaque entrée doit être connectée à tous les neurones de la couche suivante, le nombre de poids peut très rapidement devenir considérable.

Par exemple, avec 784 entrées, 15 neurones cachés et 10 sorties nous devons garder en mémoire 11910 poids, sans compter les biais :

$$784 * 15 + 15 * 10 = 11910$$

En revanche, avec une image de 224 pixels par 224 pixels avec 3 canaux (rouge, vert et bleu), ce nombre devient très grand. Imaginons un réseau avec 150 528 entrées (224 * 224 * 3) liées à une couche cachée de 1000 neurones. Ensuite nous devons multiplier ce nombre de poids par le nombre de sorties. Imaginons un réseau avec 10 sorties :

$$150\,528 * 1000 * 10 = 1\,505\,280\,000$$

Sans compter les biais, le nombre de poids s'élève à 1.5 milliards, et cela pour une seule couche cachée ! En comparaison, en 2012, AlexNet (Krizhevsky et al., 2017) un réseau de neurones à convolution (Voir 2.4 Réseau de neurones à convolution) a gagné la compétition *Large Scale Visual Recognition Challenge (ILSVRC)*, une compétition qui existe depuis 2010 mettant à l'épreuve plusieurs algorithmes de reconnaissance d'images(ImageNet, 2017), et le nombre de poids total du réseau était de 60 millions. Encore plus impressionnant, en 2014, Google avec son réseau GoogLeNet a réussi à gagner la compétition en utilisant un réseau composé de 22 couches mais qui contient seulement 4 millions de poids au total(Szegedy et al., 2015). Comme nous pouvons le

voir, le lien entre le nombre de couches et le nombre de neurones à utiliser par couche pour résoudre un problème, est subtil.

2.3.3.1.2 Reconnaissance d'images plus complexes

Nous avons constaté qu'un réseau de neurones arrive à bien classifier des chiffres écrits à la main. Ceci est impressionnant, mais cela reste tout de même un contexte très simple et bien défini.

En effet, si les chiffres sont tournés ou déplacés, le réseau n'arrive plus à les reconnaître. Si nous essayons de faire de la reconnaissance d'image sur les visages, cela devient très difficile car un réseau devrait pouvoir détecter un visage dans plusieurs positions.

La solution serait de pouvoir détecter individuellement un œil, une bouche, un sourcil, etc. D'après cette idée est né le CNN (*Convolutional Neural Network*) – Réseau de neurones à convolution.

2.4 Réseau de neurones à convolution

Les réseaux de neurones à convolution ont été créés dans les années 80(ujjwalkarn, 2016). L'idée qui a mené à sa création est le fait d'essayer de faire ressortir certaines parties de l'image qui pourraient être intéressantes et les analyser, indépendamment de leur position dans l'image.

Un réseau de neurones à convolution est essentiellement composé de 4 parties :

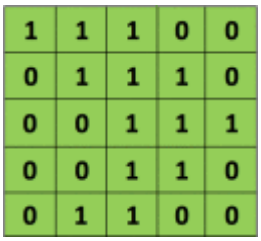
- Convolution
- Non-linéarité (ReLu)
- *Pooling*
- Classification

2.4.1 Convolution

La convolution, d'un point de vue simpliste, est le fait d'appliquer un filtre mathématique à une image. D'un point de vue plus technique, il s'agit de faire glisser une matrice par-dessus une image, et pour chaque pixel, utiliser la somme de la multiplication de ce pixel par la valeur de la matrice. Cette technique nous permet de trouver des parties de l'image qui pourraient nous être intéressantes. Des démonstrations de cette technique seront expliquées plus loin dans le chapitre « 2.4.1.1 Effets d'application de filtres ».

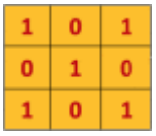
Prenons la Figure 17 comme exemple d'image et la Figure 18 comme exemple de matrice :

Figure 17 : Exemple simpliste des valeurs des pixels d'une image 5x5



(ujjwalkarn, 2016)

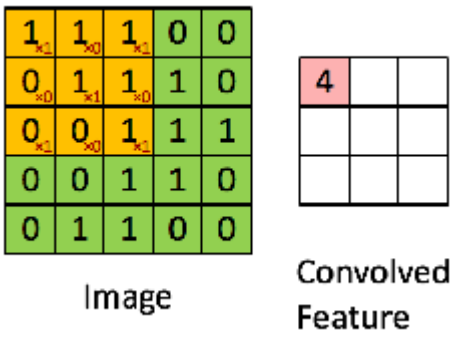
Figure 18 : Exemple de valeurs d'une matrice utilisée comme filtre



(ujjwalkarn, 2016)

Dans le cas de la Figure 17, les valeurs sont binaires. Dans un cas réel, les valeurs devraient varier entre 0 et 255. Dans la Figure 18, les valeurs sont représentées par des 1 et 0. Dans un cas réel, ces valeurs sont continues et peuvent être positives ou négatives.

Figure 19 : Démonstration de convolution



(ujjwalkarn, 2016)

Dans la Figure 19, nous pouvons voir que chaque valeur des pixels de l'image est multipliée par chaque valeur correspondante du filtre (Figure 18). Cette valeur va générer un nouveau pixel qui fera partie d'une nouvelle image(ujjwalkarn, 2016).

Le filtre doit se déplacer d'une case à chaque itération jusqu'à ce que la première ligne soit finie. Lorsque nous avons fini la première ligne, le filtre « descend » d'une case et la même procédure se répète pour chaque ligne et colonne(ujjwalkarn, 2016).

Figure 20 : Résultat attendu après une phase de convolution

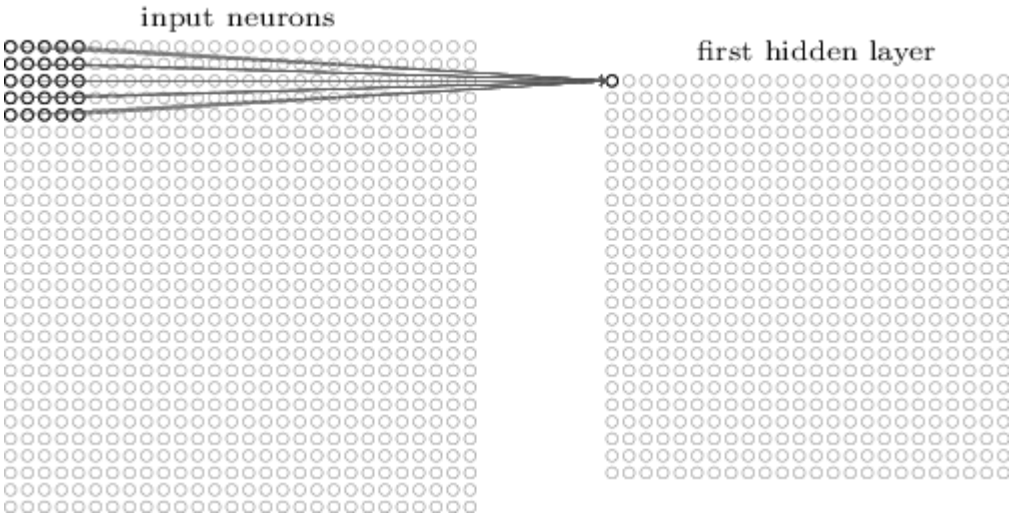
4	3	4
2	4	3
2	3	4

(ujjwalkarn, 2016)

Dans la Figure 20, nous pouvons voir le résultat obtenu après une phase de convolution. Un réseau de neurones à convolution contient de multiples filtres et ces filtres sont appliqués sur l'image d'origine. Après la première étape nous avons donc autant de nouvelles images que de filtres(ujjwalkarn, 2016).

La phase de convolution peut aussi être vue comme des couches de neurones cachées où chaque neurone n'est connecté qu'à quelques neurones de la couche suivante (Figure 21).

Figure 21 : Représentation d'une couche de convolution en utilisant des neurones artificiels



(Nielsen, 2015)

2.4.1.1 Effets d'application de filtres

Pour expliquer comment un filtre peut trouver des parties intéressantes dans les images, prenons la Figure 18 comme exemple. Les valeurs qui sont à 1 forment un « X ». Pendant la phase de convolution, lorsque ce filtre sera appliqué à une forme qui correspond exactement à un « X », alors la valeur obtenue sera plus élevée. Moins la forme de l'image correspond à la forme du filtre, plus la valeur obtenue sera basse. Nous pouvons donc réduire la taille de l'image et en faire ressortir les éléments intéressants.

Afin de démontrer les effets d'une phase de convolution, la figure suivante, qui est une image de synthèse du bâtiment B de la Haute École de Gestion de Genève, sera modifiée avec plusieurs filtres mathématiques.

Figure 22 : Campus Batelle – Bâtiment B Haute École de Gestion de Genève

(Genève, 2014)

Toutes les valeurs des filtres utilisés, ont été tirés de l'article Convolutions écrit par Utkarsh Sinha (Sinha, 2010) et l'outil utilisé pour effectuer les convolutions a été mis à disposition par Victor Powell (Victor Powell, 2015).

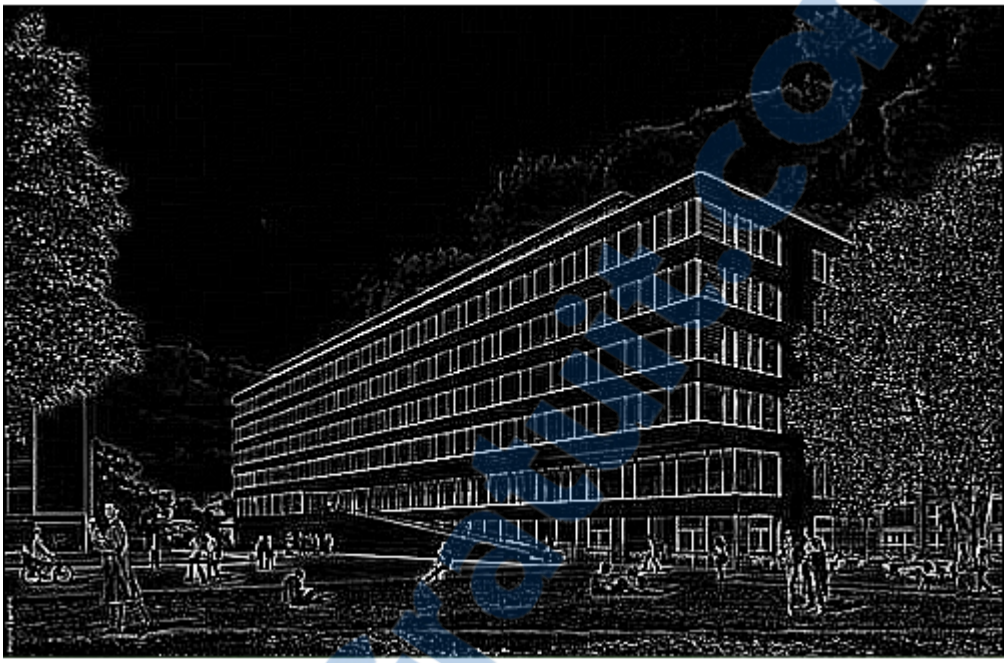
Par exemple, si nous voulons trouver des arêtes, nous pouvons utiliser les valeurs du Tableau 1.

Tableau 1 : Filtre de convolution pour détection de bordures

-1	-1	-1
-1	8	-1
-1	-1	-1

Et le résultat est le suivant :

Figure 23 : Effet de l'application d'un filtre pour détection de bordures



(Genève, 2014)

Comme vu au chapitre « 2.4.1 Convolution », la convolution réduit la taille de l'image. Cependant, vu que l'image utilisée avait une taille de 1200 par 780 pixels et qu'un filtre de 3x3 réduit l'image de seulement 2 pixels en largeur et 2 pixels en hauteur, cette différence n'est pas perceptible pour un humain. Voir « 2.4.3 Pooling » pour les techniques concernant la réduction de la taille des images.

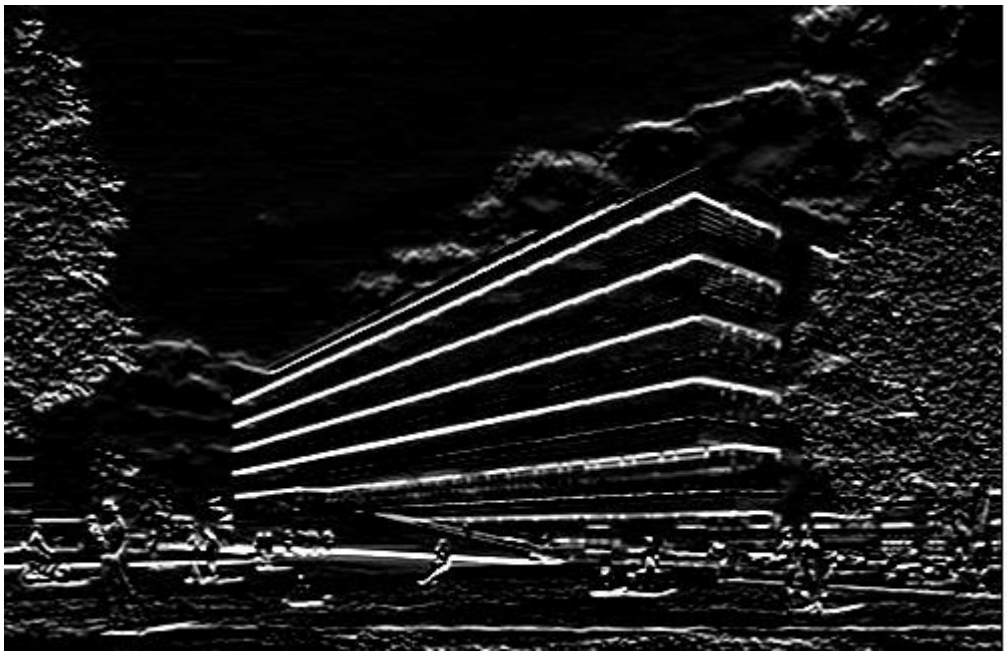
Pour détecter des lignes horizontales, nous pouvons utiliser le filtre suivant :

Tableau 2 : Filtre de convolution pour détection de lignes horizontales

-1	-2	-1
0	0	0
1	2	1

Le résultat obtenu est le suivant :

Figure 24 : Application de filtre pour détection de lignes horizontales



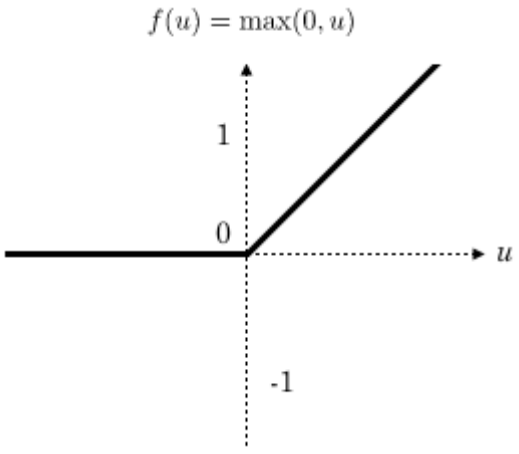
(Genève, 2014)

Les filtres sont aussi adaptés à chaque itération d'apprentissage car les valeurs des filtres mathématiques utilisés sont des poids comme dans les réseaux de neurones multicouches(ujjwalkarn, 2016) – Voir Figure 21.

2.4.2 ReLu – Unité linéaire rectifiée

ReLu est une fonction qui doit être appliquée à chaque pixel d'une image après convolution, et remplace chaque valeur négative par un 0. Si cette fonction n'est pas appliquée, la fonction créée sera linéaire et le problème XOR persiste puisque dans la couche de convolution, aucune fonction d'activation n'est appliquée.

Figure 25 : Représentation graphique de ReLu



(Pauly et al., 2017)

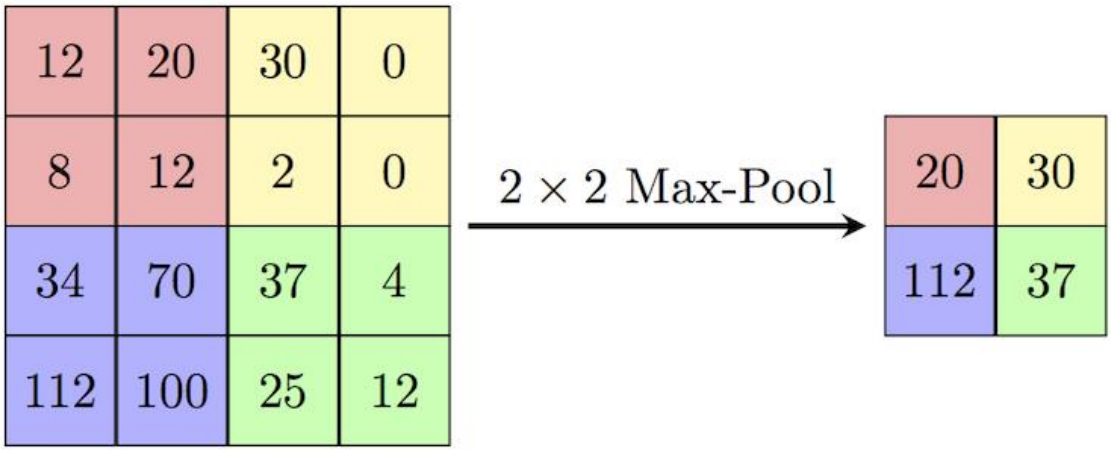
ReLu est très utilisée dans les réseaux de neurones à convolution car il s'agit d'une fonction rapide à calculer : $f(x) = max(0, x)$. Sa performance est donc meilleure que d'autres fonctions où des opérations coûteuses doivent être effectuées, comme la division dans le cas de la fonction sigmoïde, et le résultat final d'un réseau reste similaire(Nair, Hinton, 2010).

2.4.3 Pooling

Pooling est le dernier outil utilisé par les réseaux de neurones à convolution. Celui-ci est utilisé afin de réduire la taille d'une couche tout en s'assurant que les éléments importants d'une image sont gardés.

La méthode utilisée consiste à imaginer une fenêtre de 2 ou 3 pixels qui glisse au-dessus d'une image, comme pour la convolution. Mais, cette fois-ci, nous faisons des pas de 2 pour une fenêtre de taille 2, et des pas de 3 pour 3 pixels. La taille de la fenêtre est appelée « kernel size » et les pas s'appellent « strides » Pour chaque étape, nous prenons la valeur la plus haute parmi celles présentes dans la fenêtre et cette valeur constitue un nouveau pixel dans une nouvelle image. Ceci s'appelle Max Pooling. Il existe d'autres méthodes mais Max Pooling est celle qui montre les meilleurs résultats(ujjwalkarn, 2016).

Figure 26 : Démonstration du fonctionnement de Pooling



(ComputerScienceWiki, 2018)

Après une phase de Pooling, nous pouvons constater que la taille de l'image fait maintenant un quart de celle d'origine.

2.4.4 Architecture

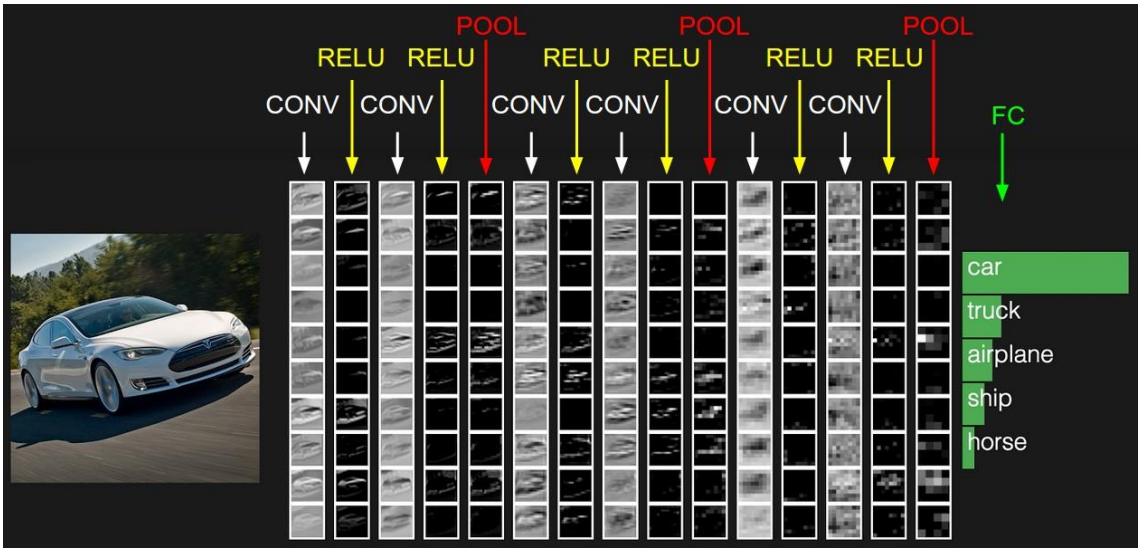
Un réseau de neurones à convolution peut avoir plusieurs étapes de convolution, ReLu et Pooling.

Une règle à respecter est que la fonction de ReLu doit obligatoirement être appliquée après une étape de convolution afin d'avoir une réponse non-linéaire, mais le Pooling n'est pas obligatoire(Deshpande, 2016).

Après être passé par toutes les étapes de convolution, ReLu et Pooling, nous pouvons passer à la classification des images.

La dernière phase consiste à envoyer tous les pixels dans un réseau de neurones multicouches. Étant donné que nous avons pu récupérer les parties les plus importantes d'une image que nous avons condensée, la phase de classification sera beaucoup plus performante qu'en utilisant un réseau de neurones artificiels sans convolution.

Figure 27 : Étapes possibles d'un réseau de neurones à convolution



(Karpathy, 2018)

2.4.5 Apprentissage d'un réseau de neurones à convolution

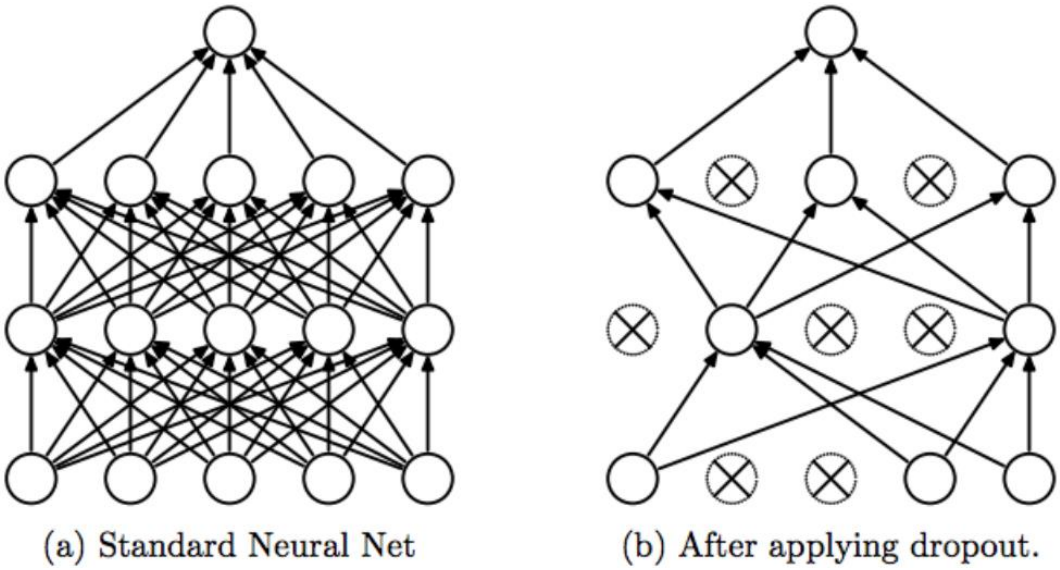
La phase d'apprentissage, de test et validation du réseau de neurones à convolution reste très similaire à celui du réseau de neurones multicouches, et les filtres sont adaptés à chaque itération également.

Cependant, d'autres techniques de prévention de surentrainement ou sous-apprentissage peuvent être appliquées.

Une technique est le *DropOut*. L'idée de cette technique est de désactiver des neurones de manière aléatoire dans notre réseau afin que celui-ci soit redondant et qu'il puisse trouver de nouveaux moyens de résoudre un même problème(Deshpande, 2016). Cette technique n'est utilisée que pendant la phase d'apprentissage. Il faut que la probabilité qu'un neurone ne soit pas désactivé soit entre 0.5 et 0.8 pour obtenir les meilleurs

résultats(Srivastava et al., 2014). Ceci peut se faire par un tirage aléatoire. Par exemple, en jetant une pièce pour décider de la désactivation d'un neurone, on obtiendrait une probabilité de 0.5. Le nombre de neurones est donc réduit mais de façon aléatoire à chaque itération d'apprentissage. Cette technique fonctionne bien dans la généralisation d'un réseau car comme vu dans le chapitre « 2.3.2.2 - Le nombre de nœuds à utiliser par couche », moins nous avons de neurones, plus le réseau se généralise.

Figure 28 : Un réseau de neurones lors de l'application de la technique de DropOut



(Budhiraja, 2016)

La technique de DropOut peut être utilisée dans la phase d'apprentissage d'un réseau de neurones multicouches ou après une phase de Pooling. Il est tout de même recommandé de le faire dans la phase d'apprentissage du réseau de neurones multicouches car il y a plus de connexions entre les neurones et l'impact du DropOut est plus élevé(Deshpande, 2016).

Une technique qui nous permet d'éviter le sous-apprentissage est d'utiliser un modèle statistique pré-entrainé(Deshpande, 2016). Cela consiste à prendre un réseau déjà entrainé sur un ensemble de données semblables mais plus grand que celui que nous avons à disposition. Ce réseau a aura donc appris à détecter des formes et des objets dans notre ensemble de données et il suffit d'adapter la dernière couche du réseau de neurones à convolution afin que celui-ci classifie correctement les éléments qui nous intéressent. Pendant la phase d'apprentissage, tous les poids restent gelés, c'est-à-dire, ils ne sont pas adaptés à chaque itération. Seuls les poids de la dernière couche (sortie) s'adaptent(Deshpande, 2016).

La dernière technique consiste à augmenter son jeu de données afin d'éviter le sous-apprentissage, ou le surentrainement. En effet, si le réseau de neurones artificiels est entrainé avec très peu de données, il peut être sous-entrainé car il n'aura pas reconnu assez d'images différentes pour pouvoir donner des réponses satisfaisantes. De plus, il est possible d'entrainer un réseau plusieurs fois avec le même ensemble de données. Dans ce cas, le réseau aura reconnu la même image trop de fois et il ne sera pas généralisé. Pour remédier à cela, il est possible d'effectuer des transformations d'image pour avoir un ensemble de données plus varié. Par exemple, selon le problème il se peut que nous puissions tourner l'image de 90º vers la gauche ou la droite. D'autres modifications peuvent aussi être effectuées, comme le changement des couleurs des images ou en inversant le sens de l'image vers la gauche ou la droite.(Deshpande, 2016). Certaines transformations peuvent être imperceptibles pour un être humain, mais pour un ordinateur elles sont assez importantes pour nous permettre de généraliser un réseau de neurones artificiels.

3. TensorFlow

TensorFlow est une librairie open source d'apprentissage automatique créée par Google Brain Team, une équipe de l'organisation de recherche de Google(Unruh, 2017). Celle-ci a été publiée sous licence open source en 2015 afin que les experts du domaine puissent apporter de nouveaux éléments et améliorer le code existant.(Famien, 2017).

En 2017, la librairie a atteint la version 1.0 et est devenue stable(Famien, 2017). Au mois de novembre 2017, il était possible de constater le développement rapide de la librairie avec déjà plus de 21 000 *commits*(Unruh, 2017), synonyme de beaucoup d'activité.

3.1 Pourquoi TensorFlow ?

La raison pour laquelle TensorFlow a été la librairie choisie pour la réalisation de ce travail de bachelor, est que cette librairie est la plus utilisée en ce moment dans le monde du *machine learning*(Tableau 3). En octobre 2017, Rachel Allen et Michael Li ont analysé les activités de 23 frameworks et librairies sur GitHub, StackOverflow et le nombre de recherches Google afin de déterminer lesquels étaient les plus populaires.

Voici quelques résultats extraits de leur analyse :

Tableau 3 : Extrait de données concernant les librairies et frameworks d'apprentissage automatique les plus utilisés

Nom	**Forks**	**Étoiles**	**Tags**	**Questions**	**Recherches**
TensorFlow	34355	69781	16462	17641	98500
Keras	7067	19504	4098	4529	30800
Caffe	12371	20155	2191	2636	46300

(Allen, Li, 2017)

Forks et **Étoiles** correspondent à GitHub, **Tags** et **Questions** correspondent à StackOverflow et **Recherches** correspond à Google.

Comme nous pouvons le constater, TensorFlow domine ses concurrents les plus proches. En termes d'activité sur GitHub, il existe environ trois fois plus de forks et étoiles, ce qui laisse entendre que la communauté est bien plus active sur TensorFlow. Aussi, il existe au moins quatre fois plus de questions sur StackOverflow, donc il sera plus facile de trouver une réponse aux difficultés auxquelles les autres développeurs ont

déjà été confrontés. Pour finir, étant donné que la communauté est plus active, il est naturel que cette librairie soit plus présente dans les recherches Google.

3.2 Structure

TensorFlow est une librairie codée en C++ afin d'obtenir une performance élevée lors des phases d'apprentissage de nouveaux modèles statistiques mais elle propose quelques APIs dans d'autres langages, comme Python, R, Java et C#(Unruh, 2017).

Il est important de noter que Google propose plusieurs APIs, mais qu'elle recommande que l'apprentissage soit fait avec le langage Python et qu'ensuite la mise en production se fasse sur d'autres langages, comme Java par exemple(TensorFlow, 2018b).

TensorFlow est cross-platform, c'est-à-dire, l'exécution peut être lancée sur plusieurs types de machines et processeurs, notamment les CPU (Central Processing Unit), les GPU (Graphic Processing Unit) et les TPU (Tensor Processing Unit)(Unruh, 2017).

Les TPU sont des processeurs optimisés pour l'exécution de traitements mathématiques sur des tensors. Les tensors seront expliqués plus loin dans ce chapitre - 3.4.1 Tensors. Ces processeurs ne sont pas disponibles pour le grand public mais il est possible de les utiliser à travers la plateforme Google Cloud(Jean Dean, 2017).

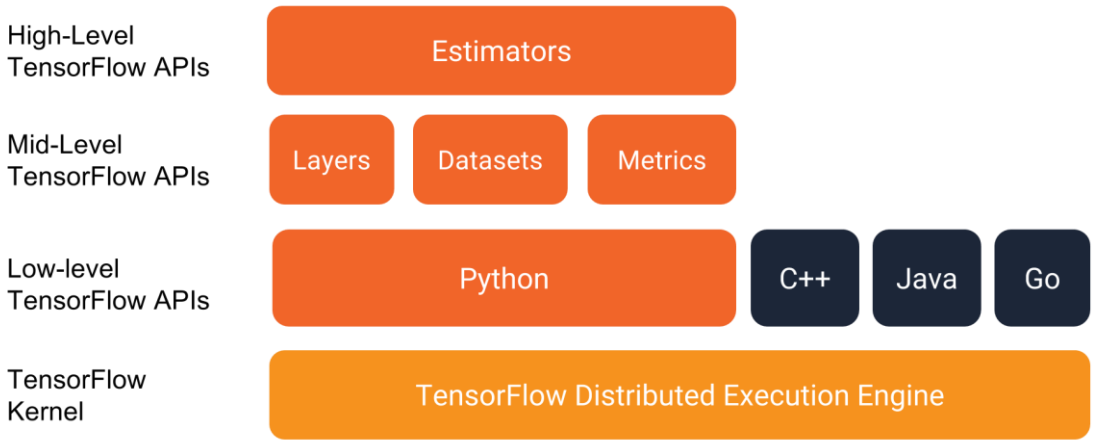
Nous pouvons voir à la Figure 29 que le développement est complètement indépendant de la plateforme d'exécution car TensorFlow possède une couche « TensorFlow Distributed Execution Engine » qui permet de distribuer les processus nécessaires(Unruh, 2017).

Nous pouvons aussi voir que la librairie possède plusieurs niveaux d'API. Les API de bas niveau sont supportés dans plusieurs langages, comme Python, C++, Java et Go. Cependant, seul Python dispose des API Layers, Dataset, Metrics et Estimator (TensorFlow, 2018c).

L'API Layers permet de paramétrer des couches de neurones artificiels au moyen d'une seule instruction. Datasets contient des fonctions qui permettent d'effectuer des transformations d'importantes quantités de données. Metrics contient des fonctions qui permettent de calculer des métriques par rapport au réseau de neurones artificiels utilisé, comme la précision.(TensorFlow, 2018c).

L'API Estimators comporte des modèles statistiques déjà structurés et prêts à être utilisés. Google nous recommande d'utiliser cette API autant que possible car les bonnes pratiques sont déjà codées de façon à ce qu'il y ait moins de risques d'erreur et que la performance du réseau soit plus élevée(TensorFlow, 2018c).

Figure 29 : Structure des couches de la librairie TensorFlow



(TensorFlow, 2018c)

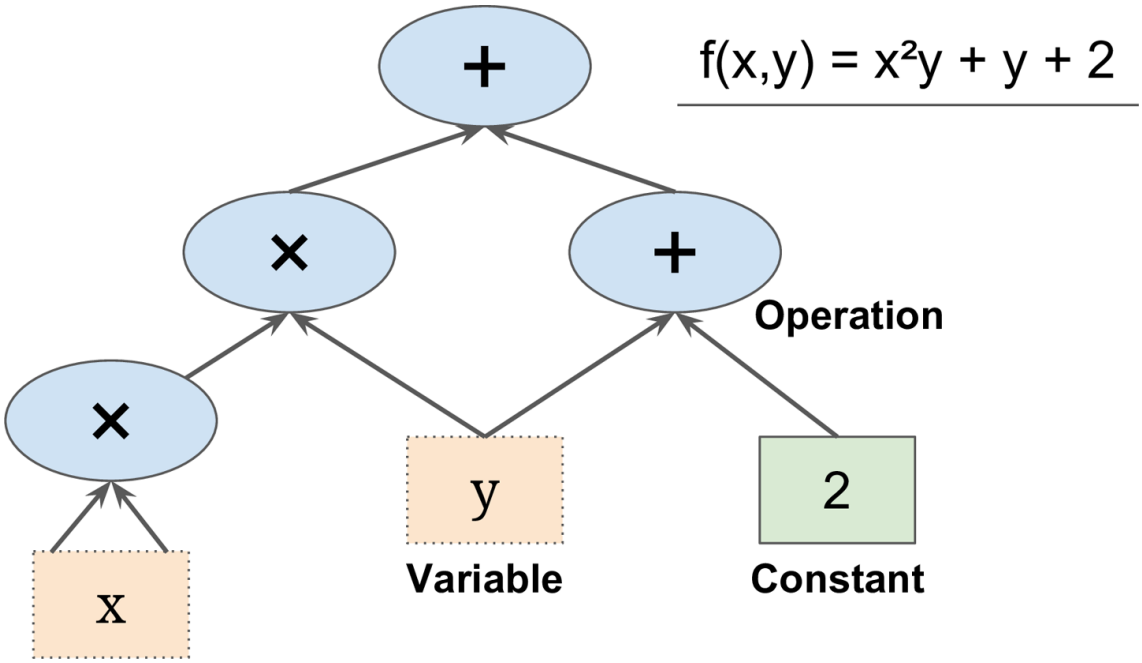
3.3 Exécution

TensorFlow propose plusieurs modes d'exécution différents :(Unruh, 2017). Dans le cadre de ce travail de bachelor, seul le graphe de flux de données sera analysé.

3.3.1 Graphe de flux de données

Un graphe de flux de données schématise un flux d'informations d'un processus ou système(LucidChart, 2017). Dans le cas de TensorFlow, ce graphe est représenté sous forme d'arbre où chaque feuille est une valeur et chaque nœud est une opération (Voir Figure 30). Lorsque nous construisons un graphe de flux de données, l'exécution ne se fait pas à chaque instruction. D'abord, le programme va lire chaque ligne du code et crée un workflow de données et d'opérations, et seulement après avoir tout interprété, alors l'exécution peut se lancer (Voir Figure 30).

Figure 30 : Démonstration d'un graphe de flux de données



(Géron, 2018)

Tout le paramétrage du réseau est fait en Python où nous avons toutes les API d'apprentissage à notre disposition mais l'exécution du code est faite en C++.

3.4 En pratique

3.4.1 Tensors

Tensor est un objet mathématique très complexe. Un tensor peut être tout simplement des tableaux de valeurs, des fonctions et peut aussi décrire des propriétés d'un système physique réel(Warren Davis, 2015).

Étant donné que l'objectif de ce travail bachelor est d'appliquer un réseau de neurones à convolution à un problème donné, nous partirons du principe qu'un Tensor est un tableau de valeurs à plusieurs dimensions.(SHARMA, 2017b). Un tensor peut donc être un scalaire, un vecteur ou une matrice.

Le nom « TensorFlow » indique donc que nous allons créer des flux d'objets mathématiques.

3.4.2 Les bases

Pour commencer à l'utiliser nous devons importer la librairie dans notre code. Les exemples suivants seront faits en Python car est le langage recommandé par Google(TensorFlow, 2018b).

```
1    import tensorflow as tf
```

La ligne ci-dessus est devenue une convention syntaxique pour aider à la compréhension et au partage de code entre les développeurs (Willems, 2017). De cette manière, à chaque fois que nous interagissons avec la librairie TensorFlow, nous devons commencer par « tf.<fonction> ».

Ensuite, nous pouvons déclarer une constante et l'afficher à l'écran :

```
import tensorflow as tf
hello = tf.constant('Hello world!')
print(hello)
```

Avec ce code, l'affichage que nous obtenons est le suivant :

```
Tensor("Const:0", shape=(), dtype=string)
```

Cet affichage montre la structure de la variable. Pour afficher son contenu, nous devons déclarer explicitement une session et lancer son exécution afin d'obtenir la valeur de la variable.

```
import tensorflow as tf
hello = tf.constant('Hello world!')
sess = tf.Session()
print(sess.run(hello))
```

Avec ce code, nous obtenons bien l'affichage attendu.

3.4.3 Les opérations

Les opérations s'exécutent facilement avec TensorFlow. Par exemple, pour une addition de vecteurs :

```
import tensorflow as tf

x1 = tf.constant([1, 2, 3])
x2 = tf.constant([5, 6, 10])
y = tf.add(x1, x2)

sess = tf.Session()
print(sess.run(y))
```

Output :

```
[ 6  8 13]
```

3.4.4 Les variables

Avec TensorFlow, il est possible de déclarer des constantes et deux types de variables. Le premier type est la « variable » et la deuxième est le « placeholder ».

La constante a déjà été démontrée dans « 3.4.2 - Les bases ». La variable dans TensorFlow est comme une variable typique en programmation, nous pouvons la déclarer et l'initialiser. Les variables sont utilisées pour stocker les valeurs des poids et des biais dans un réseau de neurones artificiels comme vu dans le chapitre « 2.2.2 Phase d'apprentissage », par exemple(Mahesh, 2017).

Pour déclarer une variable, qui est un vecteur dans ce cas précis, nous pouvons le faire de la manière suivante :

```
3    variable = tf.Variable([1.8, 1, 5], tf.float32)
```

Le premier paramètre correspond aux valeurs et le deuxième paramètre correspond au type de données qui pourront être contenus dans la variable.

Ensuite, nous devons initialiser la variable, qui est un vecteur, avec les valeurs spécifiées lors de sa déclaration. En effet, à la ligne 3, seule la structure est déclarée mais la variable ne contient aucune valeur.

```
6    init = tf.global_variables_initializer()
```

Pour finir, nous devons lancer init grâce à une session et ce n'est qu'après cela que la variable est prête à être utilisée :

```
1    import tensorflow as tf
2
3    variable = tf.Variable([1.8, 1, 5], tf.float32)
4
5    sess = tf.Session()
6    init = tf.global_variables_initializer()
7
8    sess.run(init)
9    print(sess.run(variable))
```

Comme dans un réseau de neurones artificiels nous pouvons avoir beaucoup de poids et biais à gérer, TensorFlow possède une fonction qui nous aide à déclarer plus facilement les valeurs de départ d'une variable :

```
import tensorflow as tf

variableZeros = tf.Variable(tf.zeros([3, 4]), tf.float32)
variableRandom = tf.Variable(tf.random_normal([3, 4]), tf.float32)

init = tf.global_variables_initializer()

sess = tf.Session()
sess.run(init)
print("Variable zeros : \n" + str(sess.run(variableZeros)))
print("Variable random : \n" + str(sess.run(variableRandom)))
```

En appelant tf.zeros ou tf.random_normal, il suffit de donner la structure à passer à la variable, ici 3 lignes par 4 colonnes, et tout se fait automatiquement. La fonction tf.random_normal génère des valeurs aléatoires en suivant une distribution normale avec un écart-type de 1. Les valeurs créées ont plus de probabilité de se retrouver entre -1 et 1.

```
Variable zeros :
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
Variable random :
[[-0.73030573 -0.02573803 -1.0941644  -0.4485787 ]
 [-0.92387676  1.1931279   0.58468735 -0.5280248 ]
 [ 0.9204512   0.79933965  1.5282028  -0.03545823]]
```

L'autre type de variable est le placeholder. Celle-ci est utilisée pour définir les entrées et sorties d'un réseau de neurones.

3.4.5 Un réseau de neurones à convolution

Dans ce chapitre, nous allons voir comment nous pouvons déclarer une architecture de réseau de neurones à convolution, lancer l'entraînement et tester le réseau de neurones artificiels.

Pour la démonstration de la création d'un réseau de neurones à convolution simple avec TensorFlow, l'entrainement et les tests se feront sur l'ensemble de données MNIST. Cet ensemble contient 55 000 images d'entrainement et 10 000 images de test(Google, 2017).

TensorFlow nous fournit déjà un moyen de charger toutes les images déjà préparées pour une analyse. Pour cela, il faut les importer :

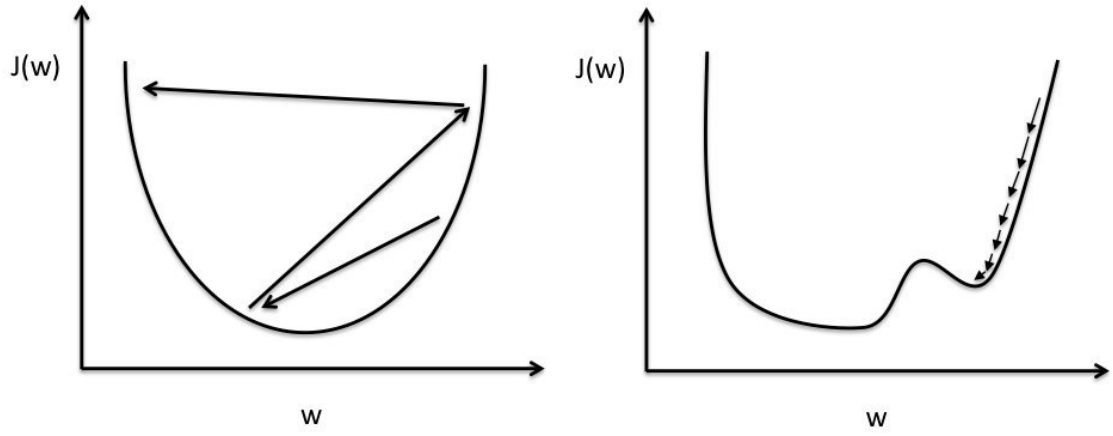
```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Le paramètre one_hot indique que nous allons obtenir des vecteurs qui représentent le chiffre correspondant pour chaque image avec toutes les valeurs à 0, sauf pour le chiffre correct. Par exemple, si l'image contient le chiffre 2, alors le vecteur de « labels » pour cette image correspond à : [0,0,1,0,0,0,0,0,0,0] parce que chaque indice du tableau correspond au chiffre que nous essayons de reconnaitre. Ensuite, nous devons définir les hyperparamètres du réseau de neurones artificiels. Les hyperparamètres sont des paramètres qui définissent l'architecture du réseau de neurones artificiels(Brownlee, 2017b).

```
learning_rate = 0.0001
nb_inputs = 784
nb_outputs = 10
batch_size = 100
```

Learning_rate est un paramètre qui permet d'indiquer à quelle vitesse les poids du réseau seront modifiés. Si cette valeur est trop petite, il faut énormément de données pour que le réseau apprenne correctement. A l'inverse, si la valeur est trop élevée, le réseau apprend plus vite mais il ne sera peut-être pas assez précis car les adaptations des poids seront trop grandes pour qu'elles se rapprochent de la bonne valeur(Surmenok, 2017).

Figure 31 : Fonction de coût dans un réseau de neurones artificiels



(Isaac, Harikumar, 2016)

Pour chaque itération d'apprentissage nous calculons une fonction de coût - Figure 31. Cette fonction de coût représente la différence entre le résultat obtenu et le résultat attendu. Lors de l'apprentissage, nous essayons de trouver les poids qui nous donnent la valeur la plus faible dans notre fonction de coût. La fonction doit déterminer comment les poids seront adaptés. Le learning_rate définit à quelle vitesse les poids changeront. A gauche, nous pouvons voir que les changements sont trop importants pour atteindre

un résultat satisfaisant. Cependant, à droite, le learning_rate fait qu'il est nécessaire d'avoir plus d'itérations d'apprentissage, mais les changements sont assez petits pour atteindre un coût minimal(Surmenok, 2017).

Pour cette expérimentation, nous aurons 784 entrées car chaque image fait 28 x 28 pixels et 10 sorties car cela correspond au nombre de chiffres de nous essayons de reconnaitre.

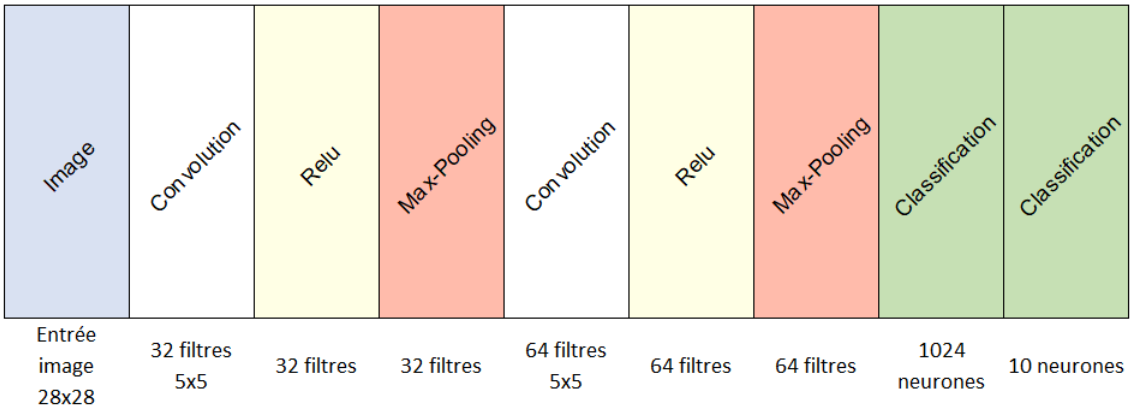
Le batch_size indique combien d'images seront traitées à la fois. Plus cette valeur est élevée, plus l'entrainement se fera vite car TensorFlow pourra charger plus d'images en une seule opération(Martin Görner, 2017). Sa seule limitation est la mémoire de l'ordinateur à disposition.

Ensuite, nous définissons les points d'entrée et de sortie de notre réseau. Ici, « None » indique à TensorFlow que la taille est variable. Cette taille correspond à la taille du batch en entrée.

```
entrees = tf.placeholder(tf.float32, [None, nb_inputs])
sorties = tf.placeholder(tf.float32, [None, nb_outputs])
```

Avant de commencer à déclarer la structure du réseau de neurones à convolution dans TensorFlow, voici un modèle simple qui nous aidera à mieux comprendre ce que nous souhaitons faire :

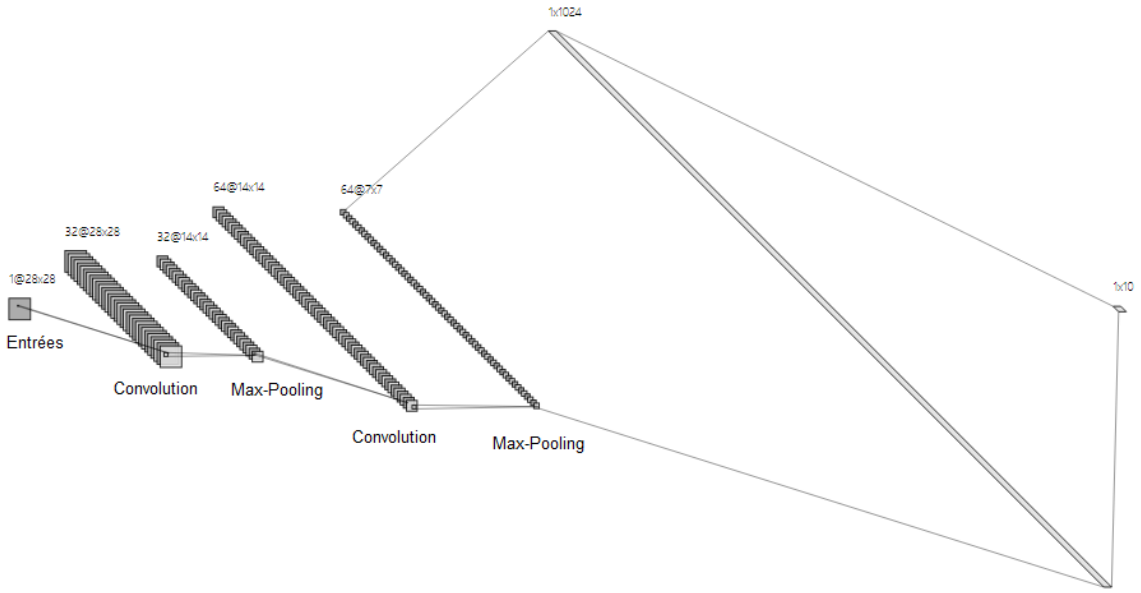
Figure 32 : Schéma simple de l'architecture d'un réseau de neurones à convolution utilisé dans l'exemple TensorFlow



Fait à l'aide de l'outil : Microsoft Office Excel

L'objectif de ce chapitre est d'apprendre à paramétrer un réseau à l'aide de la librairie TensorFlow. L'architecture ainsi que les hyperparamètres utilisés pour l'expérimentation sont basés sur ceux proposés dans le tutoriel MNIST fait par Google(Google, 2017).

Figure 33 : Architecture d'un réseau de neurones à convolution utilisé dans l'exemple TensorFlow



Fait à l'aide de l'outil : alexlenail.me

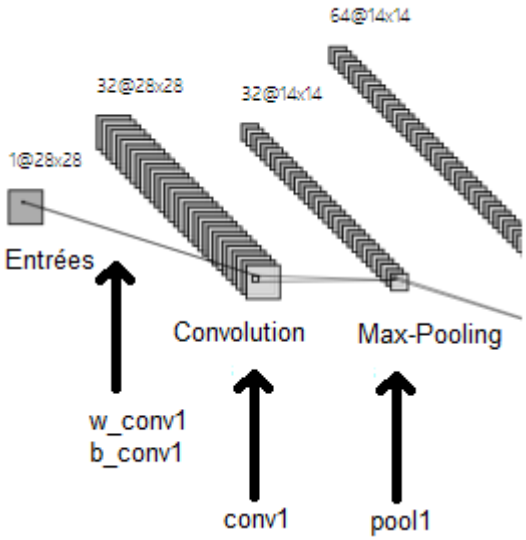
La Figure 33 sera utilisée pour mieux expliquer comment paramétrer cette architecture avec TensorFlow. Nous pouvons voir qu'après les couches de convolution, la taille des images ne change pas et cela est contraire à ce qui a été vu dans le chapitre « 2.4.1 Convolution ». En effet, lors des phases de convolution, un padding avec des zéros a été ajouté pour garder la même taille de l'image dans le but de faciliter le calcul du nombre de neurones utilisés dans les couches suivantes.

Premièrement, nous devons modifier la forme des images. En effet, la fonction utilisée pour charger les images nous retourne un vecteur. Or, notre réseau utilise des matrices. Avec la fonction reshape, nous pouvons modifier la forme d'un tenseur en lui indiquant la taille du nouveau tenseur dans le deuxième paramètre. La syntaxe à respecter pour indiquer la forme de l'image est [batch, hauteur de l'image, largeur de l'image, canaux]. La valeur -1 est une notation pour indiquer que la taille sera variable et dans MNIST un seul canal est utilisé : le noir et blanc.

```
15      image = tf.reshape(entrees, [-1, 28, 28, 1])
```

Ensuite, nous devons déclarer notre première étape de convolution et pooling :

Figure 34 : Première étape de convolution



Fait à l'aide de l'outil : alexlenail.me

Pour commencer, nous devons déclarer les poids et les biais des connexions entre les entrées et la première couche de convolution.

```
w_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
b_conv1 = tf.Variable(tf.constant(0.1), [32])
```

Dans le tenseur déclaré dans la fonction truncated_normal, les valeurs 5 et 5 indiquent la taille de chaque filtre à appliquer. Le 1 correspond au nombre de canaux, ici un seul car l'image utilisée est en noir et blanc, et 32 correspond au nombre de filtres mathématiques individuels. Les valeurs choisies pour l'écart-type (stddev) et les biais ont été tirées du tutoriel Google(Google, 2017).

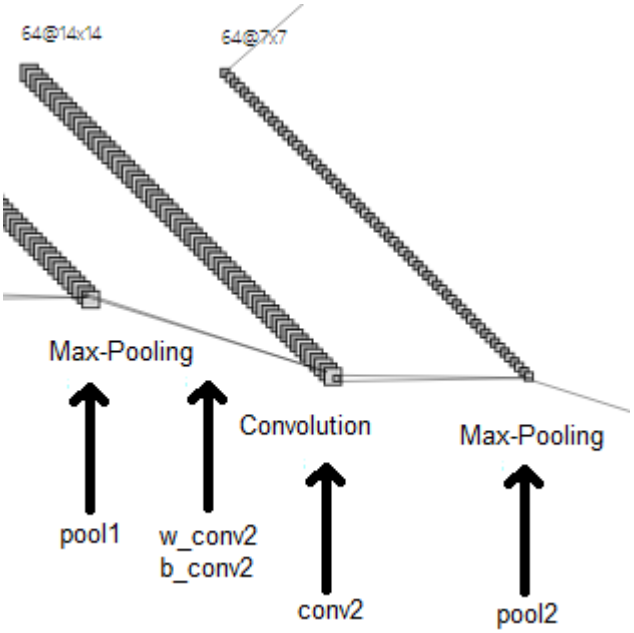
```
conv1 = tf.nn.relu(tf.nn.conv2d(image, w_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1)
pool1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

Pour créer une couche de convolution, nous devons appeler la fonction conv2d. Le paramètre strides correspond au nombre de cases auxquelles le filtre doit être déplacé à chaque étape avec la notation [batch, hauteur, largeur, canal]. Ici nous indiquons 1 case à l'horizontal et 1 case à la verticale. Nous indiquons aussi que tous les canaux et chaque batch doivent être traités un par un. Le paramètre padding indique qu'aucun pixel n'est perdu pendant la phase de convolution, car TensorFlow ajoute des 0 dans les bords des images dans le but de garder la même taille. En effet, nous pourrions choisir de perdre les pixels des bords.

La fonction ReLu permet d'appliquer la fonction d'activation à la couche de convolution définie précédemment.

Pour réduire la taille de chaque image, nous utilisons une couche de Max-Pooling. Le paramètre ksize correspond à la taille de la fenêtre utilisée pour la phase de pooling. La fenêtre est de taille 2 x 2. En ce qui concerne les strides, cette fois-ci nous utilisons des pas de 2 à l'horizontale et à la verticale.

Figure 35 : Deuxième étape de convolution



Fait à l'aide de l'outil : alexlenail.me

La même procédure doit être appliquée à la deuxième couche de convolution. Seulement, il est nécessaire que les données en entrée correspondent à la sortie de la première couche.

```
w_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
b_conv2 = tf.Variable(tf.constant(0.1), [64])

conv2 = tf.nn.relu(tf.nn.conv2d(pool1, w_conv2, strides=[1, 1, 1, 1], padding='SAME') + b_conv2)
pool2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

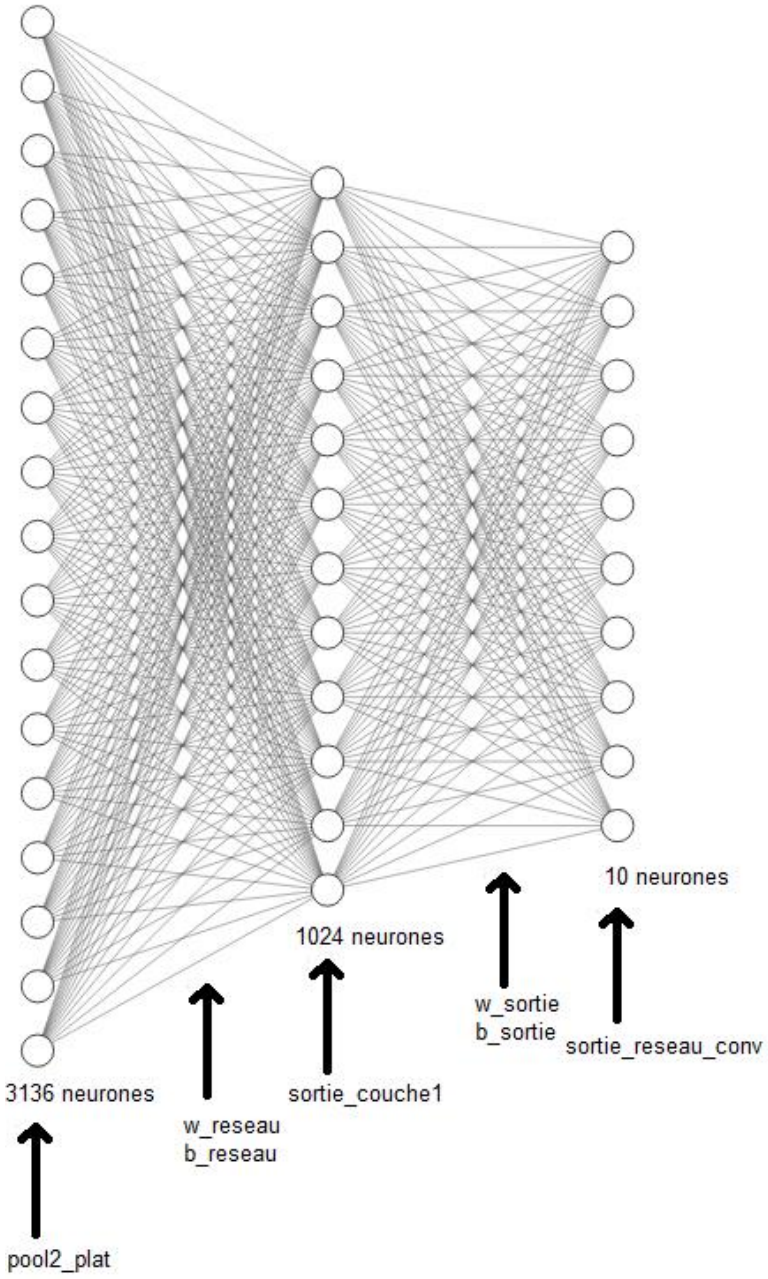
Il est aussi important de noter que nous avons utilisé 64 filtres au lieu de 32.

Pour finir, nous devons aplatir notre résultat et l'utiliser dans un réseau de neurones artificiels multicouches. La valeur « -1 » correspond à la taille du batch et elle indique que la taille est variable. La deuxième valeur 7 * 7 * 64 correspond à la taille du vecteur. Nous devons multiplier 7 et 7 car deux couches de Max-Pooling ont été appliquées et la taille de l'image originale (28 et 28) a été divisée par deux, deux fois. 64 correspond au nombre de filtres de la dernière couche de convolution.

```
pool2_plat = tf.reshape(pool2, [-1, 7 * 7 * 64])
```

Nous avons maintenant un vecteur de taille $7 * 7 * 64 = 3136$. Ce vecteur sera utilisé en tant qu'entrée dans un réseau de neurones artificiels multicouches pour effectuer de la classification. La Figure 36 représente les couches de classification vues dans les figures 32 et 33, et le nombre de neurones représentés dans chaque couche cachée ne correspond pas au nombre de neurones réels utilisés.

Figure 36 : Architecture du réseau de neurones artificiels utilisé dans l'exemple TensorFlow



Fait à l'aide de l'outil : alexlenail.me

Comme dans une couche de convolution, nous devons commencer par déclarer les poids et les biais à utiliser :

```
w_reseau = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1))
b_reseau = tf.Variable(tf.constant(0.1), [1024])
```

Pour définir comment calculer la sortie de la couche cachée, nous indiquons qu'il faut calculer le produit matriciel des entrées et ses poids correspondants, et additionner les biais. Dans cette couche, nous appliquons la fonction d'activation ReLu, comme dans les couches de convolution. Cependant, nous devons déclarer la multiplication matricielle pour cette couche car la librairie nous l'impose ici, alors que cela est appliqué implicitement dans les couches de convolution.

```
sortie_couche1 = tf.nn.relu(tf.matmul(pool2_plat, w_reseau) + b_reseau)
```

Ensuite, pour éviter que le réseau surentraine, nous appliquons du dropout avec une probabilité de 80% (0.8) ; valeur recommandée par Google.

```
sortie_couche1_drop = tf.nn.dropout(sortie_couche1, keep_prob)
```

Pour finir, la même procédure est appliquée pour les poids, biais et couche de sortie, mais ni dropout ni fonction d'activation ne sont utilisés :

```
w_sortie = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1))
b_sortie = tf.Variable(tf.constant(0.1), [10])

sortie_reseau_conv = tf.matmul(sortie_couche1_drop, w_sortie) + b_sortie
```

La prochaine étape est de calculer la fonction de coût qui permet de savoir si le réseau de neurones artificiels a prédit le bon résultat. Ceci peut être calculé à l'aide de la fonction softmax_cross_entropy_with_logits qui applique une fonction d'activation softmax. Celle-ci nous permet d'avoir résultat où chaque valeur est entre 0 et 1, et où la somme de toutes les valeurs est égale à 1. Grâce à cette fonction, nous obtenons une probabilité pour chaque chiffre à prédire où « *labels* » correspond à la réponse attendue et « *logits* » à la réponse retournée par le réseau de neurones artificiels.

```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=sorties, logits=sortie_reseau_conv))
```

Pour finir la construction du graphe de flux de données en ce qui concerne l'entraînement, nous devons indiquer comment réduire la fonction de coût. Le fonctionnement interne de cette fonction n'est pas analysé dans le cadre de ce travail de

bachelor. Ici nous utilisons AdamOptimizer car il s'agit de la fonction qui affiche les meilleurs résultats(Brownlee, 2017a).

```
optimiser = tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
```

Concernant la phase de test, nous devons déterminer quels indices du vecteur de sortie du réseau de neurones artificiels contiennent la probabilité la plus élevée avec la fonction argmax. Pour chaque résultat, nous allons vérifier si l'indice est le même que celui de la réponse attendue. Si oui, alors la prédiction est correcte. Ensuite, nous pouvons utiliser le nombre de bonnes prédictions pour calculer la précision du réseau.

```
correct_prediction = tf.equal(tf.argmax(sortie_reseau_conv, 1), tf.argmax(sorties, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Nous pouvons désormais passer à l'entraînement. La ligne 40 récupère les prochaines « batch_size » images de l'ensemble des données. La ligne 41 demande à TensorFlow de calculer la fonction de coût et d'optimiser le réseau pour les images récupérées à la ligne d'au-dessus.

```
39  for i in range(total_batch):
40      images_entrainement, labels_entrainement = mnist.train.next_batch(batch_size=batch_size)
41      sess.run([optimiser, cross_entropy], feed_dict={entrees: images_entrainement, sorties: labels_entrainement})
```

Une fois que la boucle s'est terminée, nous pouvons calculer la précision de notre réseau :

```
43  print("Entrainement fini!")
44  print("Performance : ", sess.run(accuracy, feed_dict={entrees: mnist.test.images, sorties: mnist.test.labels}))
```

Et le résultat obtenu est :

```
Entrainement fini!
Performance :  0.949
```

Ce qui montre que le réseau de neurones artificiels a bien été entraîné et que sa précision est de 94.9% !

3.4.6 TensorBoard

TensorFlow utilise une application qui nous permet de visualiser le graphe de flux de données. Pour l'utiliser nous pouvons faire :

```python
import tensorflow as tf

x1 = tf.constant(1)
x2 = tf.constant(2)
y = tf.multiply(x1, x2)
x3 = tf.constant(5)
y = tf.multiply(y, x3)
sess = tf.Session()
writer = tf.summary.FileWriter("output", sess.graph)
print(sess.run(y))
writer.close()
```

Nous avons besoin d'utiliser un FileWriter car TensorBoard se base sur des fichiers de log afin de créer les graphiques.

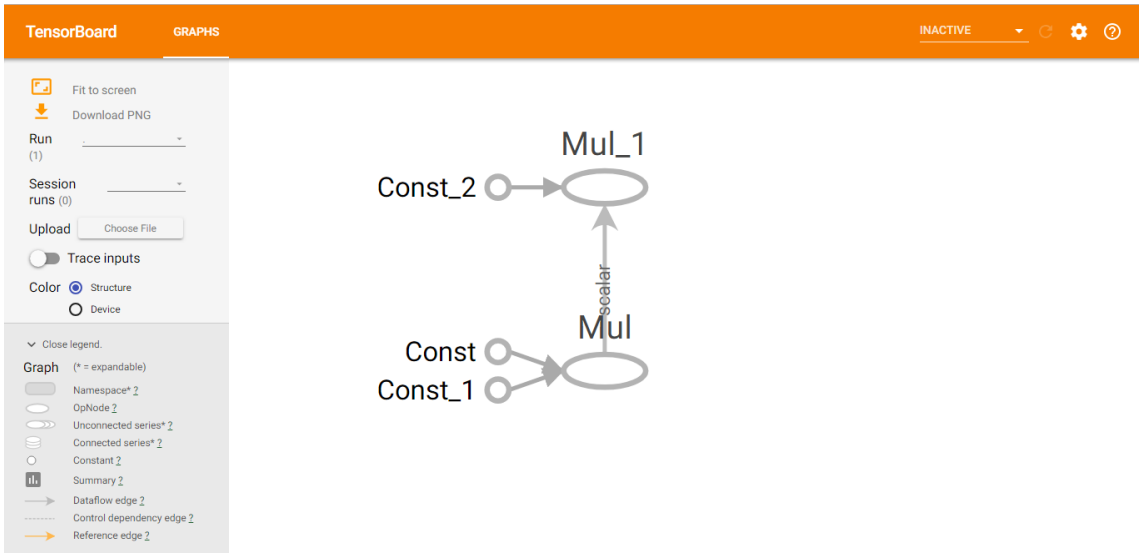
Une fois que le script a été exécuté, nous devons utiliser la commande suivante :

```
$ tensorboard --logdir=chemin/vers/logs
```

Après avoir exécuté la commande, un serveur TensorBoard se lance :

```
TensorBoard 1.8.0 at http://DESKTOP-6OJML8D:6006 (Press CTRL+C to quit)
```

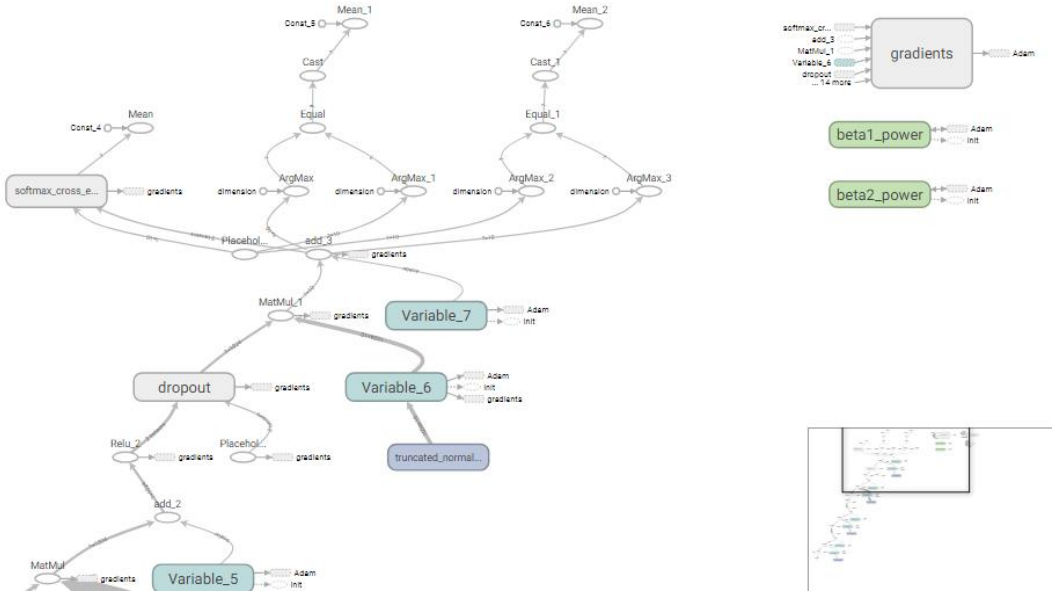
Figure 37 : Graphe de flux de données simple sur TensorBoard



Fait à l'aide de l'outil : TensorBoard

Et ceci est l'affichage qui correspond au graphe de flux de données. Cependant, ce graphe peut devenir très complexe, comme dans l'exemple du réseau de neurones à convolution :

Figure 38 : Graphe de flux de données complexe sur TensorBoard



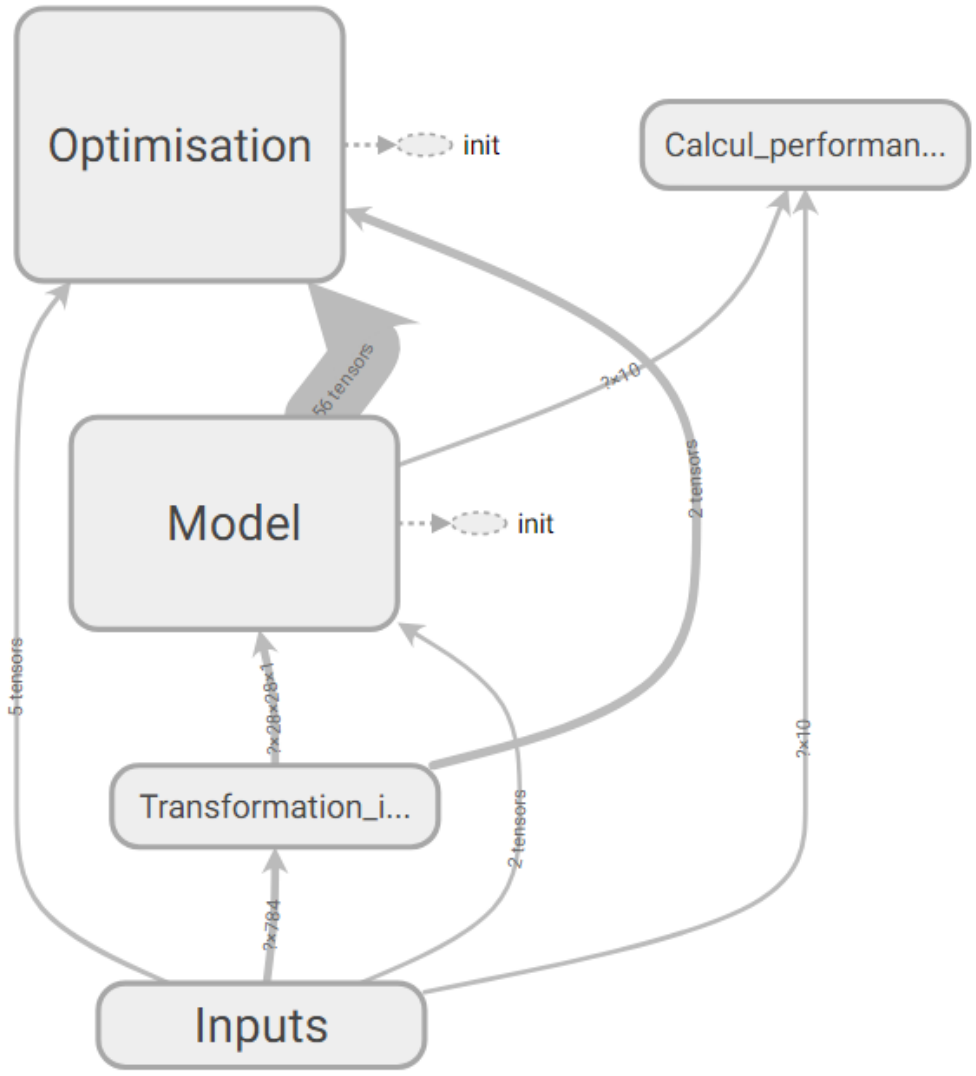
Fait à l'aide de l'outil : TensorBoard

De plus, nous ne pouvons pas savoir à quoi correspondent les variables. Afin de remédier à cela, TensorFlow propose des « scopes » et des « names » dans le but d'améliorer la lisibilité des graphiques. Les scopes nous permettent de séparer des parties spécifiques de notre graphe et les names nous permettent de donner des noms plus parlants aux variables.

```
11  with tf.name_scope("Inputs"):
12
13      entrees = tf.placeholder(tf.float32, [None, nb_inputs], name="Entrees")
14      sorties = tf.placeholder(tf.float32, [None, nb_outputs], name="Sorties")
15      keep_prob = tf.placeholder(tf.float32, name="Keep_prob")
```

Il est possible d'imbriquer plusieurs « scopes ». Une fois que tout le réseau est structuré, nous obtenons le résultat suivant :

Figure 39 : Graphe de flux de données structuré avec TensorFlow



Fait à l'aide de l'outil : TensorBoard

En plus de cela, TensorBoard nous permet de générer des graphiques avec des indicateurs qui pourraient nous être intéressants, comme par exemple, la précision du réseau de neurones artificiels.

```
70      tf.summary.scalar("accuracy", accuracy)
```

Lorsqu'un élément nous intéresse, il suffit d'écrire la ligne ci-dessus pour qu'un graphique soit généré.

Il est aussi possible de créer des histogrammes :

```
47              tf.summary.histogram('Poids_premiere_couche', w_reseau)
```

Ensuite, la ligne suivante doit être ajoutée pour que le système regroupe toutes les données en une seule variable :

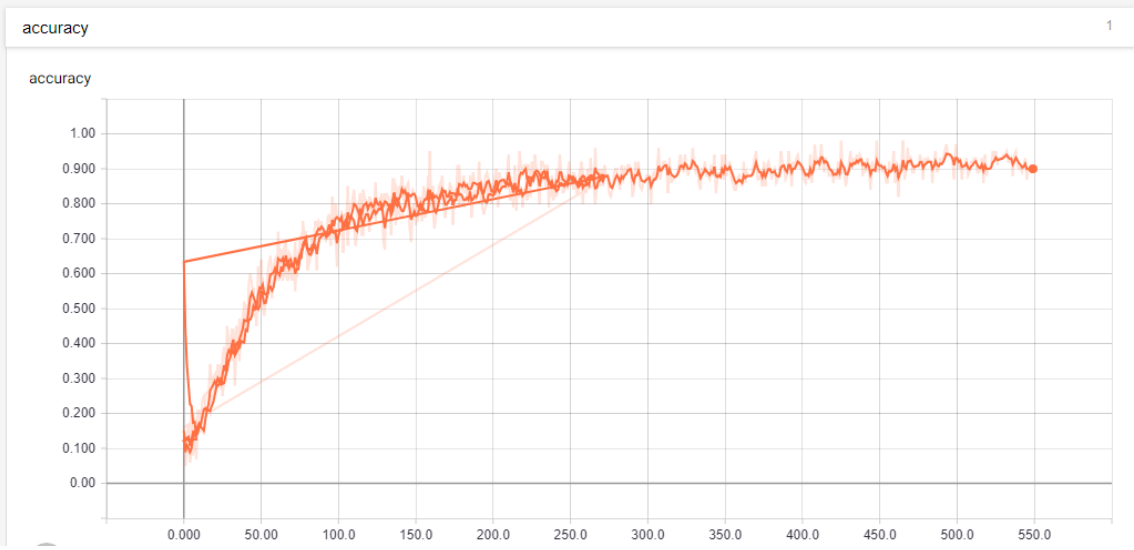
```
77      summaryMerged = tf.summary.merge_all()
```

Pour écrire les données dans le log, nous pouvons l'ajouter au writer qui se charge de structurer les données correctement :

```
86          _,_, summary = sess.run([optimiser, cross_entropy, summaryMerged],
87          writer.add_summary(summary, i)
```

Et voici un exemple de graphique que TensorBoard peut générer :

Figure 40 : Graphique de performance sur TensorBoard



Fait à l'aide de l'outil : TensorBoard

L'axe des abscisses représente le nombre de batches exécutés et l'axe des ordonnées représente la performance du réseau en pourcentage.

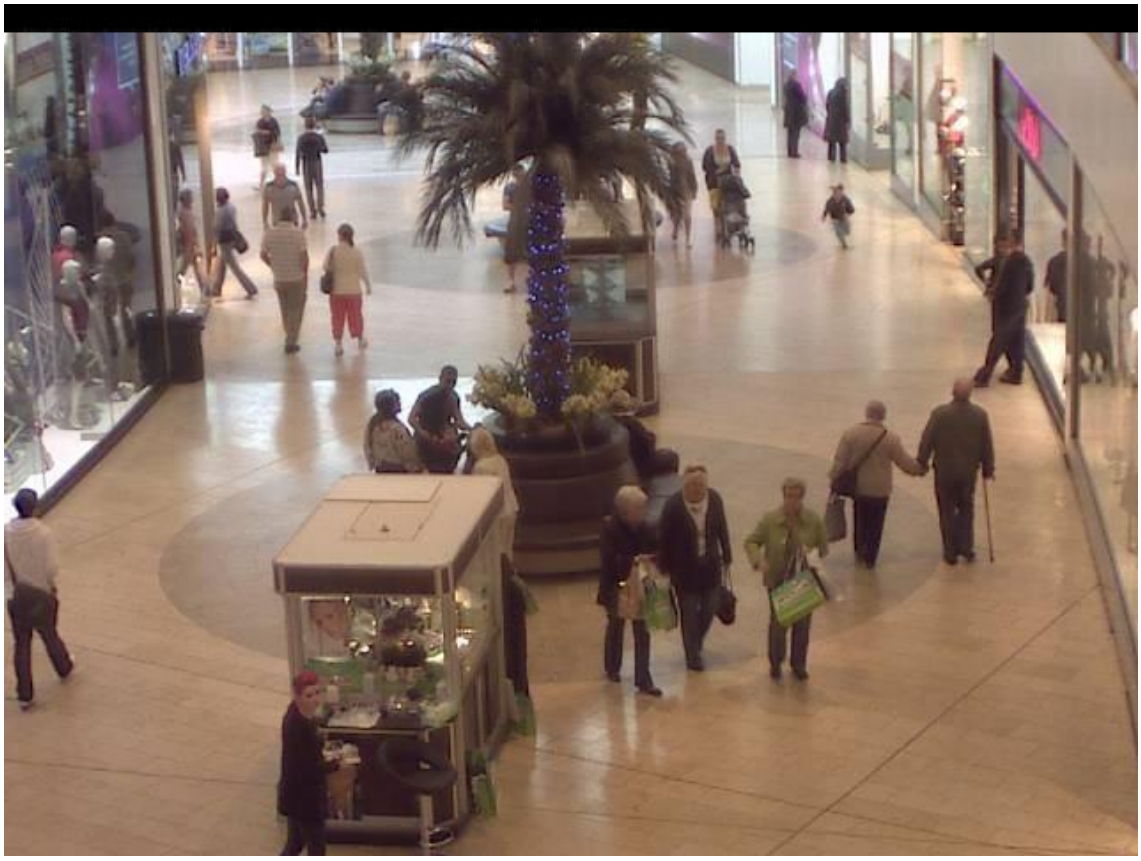
4. Implémentation

Concernant l'implémentation, le problème à traiter est de pouvoir compter des personnes. Pour cela, il est nécessaire d'avoir un jeu de données avec lequel le réseau de neurones à convolution sera entraîné. Il y a plusieurs alternatives : faire son propre jeu de données ou trouver un jeu de données disponible sur Internet qui puisse être utilisé librement.

Puisque compter des personnes sur des images est une tâche chronophage, la meilleure solution est de trouver un jeu de données disponible.

Le jeu de données utilisé dans cette phase d'implémentation contient des images d'une caméra de surveillance dans un centre-commercial(Loy, 2014) et peut être utilisé librement à des fins de recherche. Elle contient 2000 images et elle compte plus de 60 000 personnes. Pour qu'une personne soit comptabilisée, sa tête doit être visible.

Figure 41 : Exemple d'image dans le jeu données utilisé pour l'implémentation



(Loy, 2014)

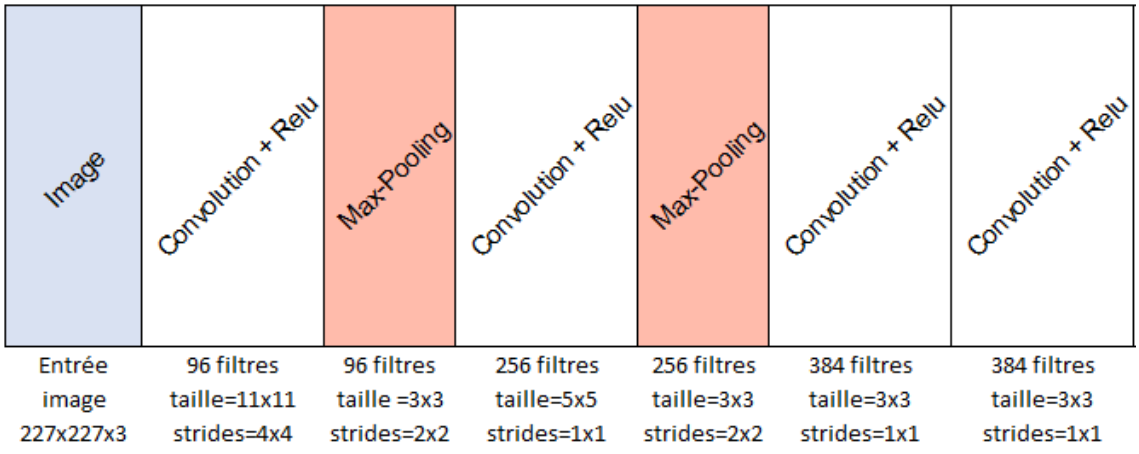
4.1 Architecture pour compter des personnes

En regardant les images obtenues et le nombre de paramètres à prendre en compte dans un réseau de neurones à convolution, j'ai décidé d'utiliser une architecture connue

pour résoudre le problème de compter des personnes. Je trouve que, pour un ordinateur, reconnaître des personnes est une tâche complexe, car chaque personne est habillée de manière différente, avec des couleurs différentes, possède une structure physique unique, etc. L'architecture choisie est celle d'AlexNet. Comme abordé dans le chapitre « 2.3.3.1 Les difficultés d'un réseau de neurones pour la reconnaissance d'image », cette architecture a gagné la compétition *Large Scale Visual Recognition Challenge (ILSVRC)* qui met à l'épreuve plusieurs algorithmes de reconnaissance d'image. De plus, elle est plus simple à paramétrer dans TensorFlow, comparée à d'autres architectures plus récentes comme LeNet qui contient 22 couches.

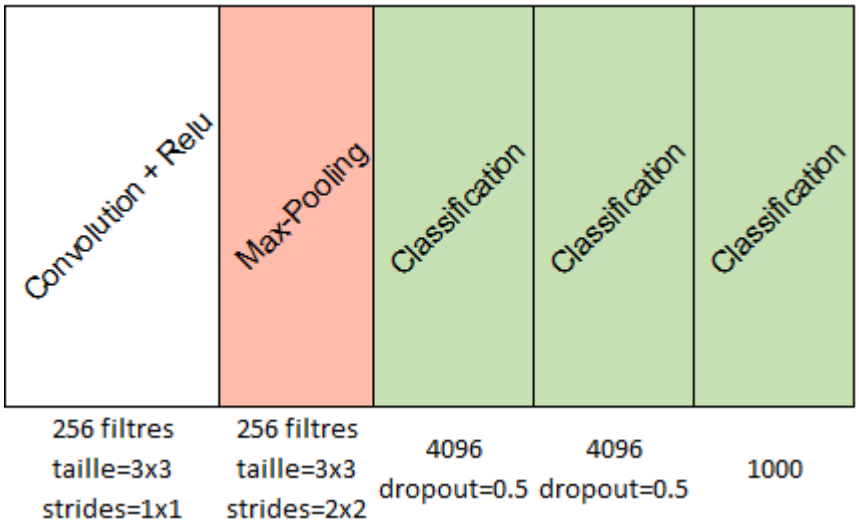
Voici un schéma de son architecture (Gao, 2017):

Figure 42 : Architecture AlexNet – Partie 1



Fait à l'aide de l'outil : Microsoft Office Excel

Figure 43 : Architecture AlexNet – Partie 2



Fait à l'aide de l'outil : Microsoft Office Excel

Pour pouvoir l'utiliser, un changement doit être effectué à cette architecture. En effet, AlexNet était utilisé pour faire de la classification. C'est-à-dire que le réseau pourra dire si pour une image donnée, il y a une probabilité que ce soit un chien, un chat ou une voiture, par exemple. En revanche, dans le cas à traiter dans ce chapitre, le résultat obtenu ne doit pas être une probabilité mais il doit être une valeur continue.

Pour cela, il suffit de remplacer la dernière couche qui contient 1000 neurones, par une couche avec un seul neurone sans qu'aucune fonction d'activation ne soit appliquée. Tous les exemples faits jusqu'ici font référence à de la classification car la plupart de la littérature, exemples et même tutoriels sont codés pour de la classification. Mais, les mêmes principes sont appliqués à de la régression(MathWorks, 2018).

Un autre changement à devoir être apporté au réseau de neurones artificiels est la fonction de coût utilisée. La fonction est l'erreur quadratique moyenne. C'est-à-dire, nous calculons la soustraction de la réponse attendue à la prédiction au carré. Ainsi, l'erreur est toujours positive(MathWorks, 2018). La fonction qui doit réduire ce coût est toujours AdamOptimizer, comme dans de la classification.

4.2 Traitement des données

Pour commencer, les images ont dû être séparées en trois dossiers différents : train, validation et test. J'ai décidé d'utiliser 1600 images pour entrainer le réseau, 200 pour le valider et 200 pour le tester, comme recommandé dans le chapitre « 2.2.2.4 Le meilleur modèle statistique ». Les labels et les valeurs qui représentent le bon nombre de personnes sur une image, sont enregistrés dans trois fichiers séparés : train.csv, validation.csv et test.csv. Comme vu au chapitre « 2.2.2.3 Le sous-apprentissage », les données doivent être mélangées pour éviter le sous-apprentissage. Le jeu de données était déjà mélangé, cette étape pouvait être ignorée. J'ai simplement dû m'assurer que l'ordre du nom des images était le même que celui des labels correspondant dans les fichiers.

4.2.1 Chargement des images

Pour charger les images, nous allons d'abord créer une fonction qui prend en paramètre un chemin relatif :

```
def createBatch(relative_path):
```

Ce chemin relatif est utilisé pour charger tous les noms de fichiers d'un répertoire, selon le type de fichier :

```
filename_dataset = tf.data.Dataset.list_files(relative_path, shuffle=False)
```

Le paramètre shuffle définit si les noms de fichiers doivent être mélangés. Dans notre cas, le paramètre doit être à false pour garder une correspondance entre les images et les labels.

Pour définir le type de fichier, cela doit être fait dans le chemin relatif :

```
train_image_path = 'train/*.jpg'
```

Avec « *.jpg » seuls les noms de fichiers au format JPEG sont chargés.

Après avoir chargé les noms de fichiers, nous les transformons en tenseurs contenant des images :

```
image_dataset = filename_dataset.map(lambda x: tf.image.decode_jpeg(tf.read_file(x)))
```

La fonction map permet d'appliquer n'importe quelle fonction à chaque élément d'un tenseur. Cette ligne va lire chaque fichier par rapport à son nom et le décoder au format JPEG.

Puisque nous avons 2000 images à charger, cela va consommer trop de ressources si nous essayons de tout charger en même temps. Nous allons donc créer un batch qui nous permet de charger quelques images à la fois :

```
dataset_batch = tf.data.Dataset.batch(image_dataset, batch_size)
```

Pour parcourir ce batch, nous devons déclarer un itérateur :

```
iterator_initializer = dataset_batch.make_initializable_iterator()
```

La fonction make_initializable_iterator() nous permet de déclarer plusieurs itérateurs pour parcourir plusieurs fois les mêmes données.

Avant que les images puissent être utilisées dans le réseau de neurones artificiels, nous allons effectuer quelques modifications pour qu'elles aient le bon format et pour augmenter le nombre d'images à disposition. Le fait d'augmenter le nombre d'images à disposition réduit la probabilité qu'un réseau soit surentrainé (Voir « 2.4.5 Apprentissage d'un réseau de neurones à convolution »).

```
images_resized = tf.image.resize_images(iterator_initializer.get_next(), [image_size, image_size])
```

Ici, à chaque fois qu'un nouveau batch devra être retourné, les images seront transformées pour qu'elles fassent toutes la même taille : 227 x 277.

```
flipped = tf.map_fn(lambda frame: tf.image.random_flip_left_right(frame), images_resized)
```

Ensuite, ces images sont passées dans une fonction qui les inverse horizontalement. La fonction inverse les images de manière aléatoire avec une probabilité de 50% pour chaque image.

```
brightness = tf.map_fn(lambda frame: tf.image.random_brightness(frame, max_delta=0.3), flipped)
```

La prochaine étape est de modifier la luminosité des pixels. Le paramètre max_delta indique par quel pourcentage sa luminosité sera changée. La valeur 0.3 indique que la luminosité peut augmenter ou diminuer au maximum de 30%, mais cette valeur est aléatoire. J'ai décidé d'utiliser cette valeur car dans un article écrit par Aggarwal concernant les modifications d'image, nous pouvons voir qu'une modification de 50% a été faite et l'image est devenue trop foncée(Aggarwal, 2018). Puisque dans le jeu de données il n'y a pas beaucoup de luminosité, je ne voulais pas que la luminosité soit réduite au point que les personnes ne soient plus reconnaissables.

Avant que les images soient prêtes, nous devons les normaliser (Chapitre 2.2.1 Structure) à l'aide de la fonction suivante et nous pouvons retourner ce résultat :

```
batch = tf.map_fn(lambda frame: tf.image.per_image_standardization(frame), brightness)
return iterator_initializer, batch
```

La fonction per_image_standardization normalise les pixels d'une image.

La variable iterator_initializer permettra de recommencer au début des données et batch permettra de récupérer le prochain batch de données.

Cette fonction peut donc être appelée pour chaque type d'images (entrainement, validation et test) dans notre jeu de données :

```
train_batch_initializer, train_batch = createBatch(relative_path2images + train_image_path)
validation_batch_initializer, validation_batch = createBatch(relative_path2images + validation_image_path)
test_batch_initializer, test_batch = createBatch(relative_path2images + test_image_path)
```

4.2.2 Chargement des labels

Pour charger les labels, nous pouvons utiliser la librairie numpy, une librairie qui est installée avec TensorFlow.

```
train_labels = genfromtxt(relative_path2labels + train_labels_filename)
```

Pour cela, nous devons appeler la fonction genfromtxt avec le chemin relatif vers le fichier .csv. Puisque nous avons un seul label par ligne, la fonction nous retourne automatiquement un vecteur avec chaque valeur.

Pour les labels, j'ai décidé de ne pas utiliser de batch car les labels sont moins gourmands en ressources que les images. Pour m'assurer que j'envoie les bons labels au réseau de neurones à convolution, j'ai fait une gestion d'indice. C'est-à-dire que je m'assure que les valeurs envoyées au réseau de neurones artificiels correspondent aux indices. Pour l'exemple, imaginons un batch de taille 10 : Lors de la première itération d'apprentissage, j'envoie les indices 0 à 9, lors de la deuxième itération, 10 à 19, et ainsi de suite.

4.3 Entrainement

Pendant la phase d'entrainement, certaines modifications sont faites aux images. Pour cette raison, il est possible de continuer à entrainer un réseau sur un même ensemble d'images tant que l'erreur de l'ensemble de données de validation continue de descendre. Chaque cycle complet d'une phase d'entrainement s'appelle « Epoch » et l'ensemble de validation est testé à chaque fin d'Epoch(SHARMA, 2017a). Pour s'assurer que le réseau est bien entrainé, il faut tout de même avoir un ensemble de test qui n'est testé qu'une fois que tous les cycles d'entrainement ont été finis.

Les paramètres utilisés pour l'entrainement sont les suivants et ont été tirés des recommandations pour cette architecture (Gao, 2017):

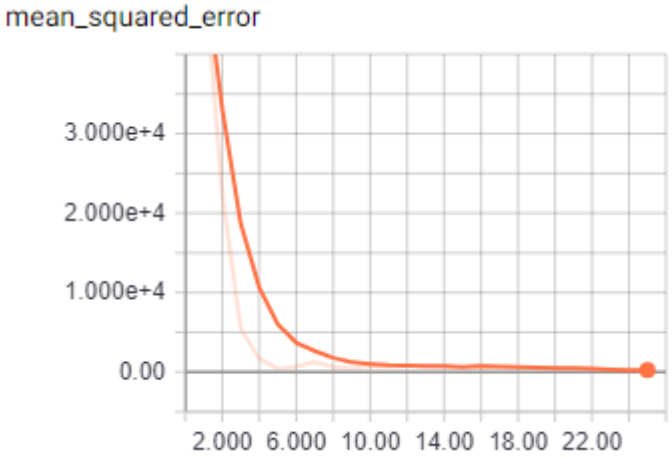
- Taille de chaque batch : 8
- Learning rate : 0.0001
- Epochs : 25
- Probabilité de DropOut : 0.5

Le paramétrage du réseau à l'aide de TensorFlow ne sera pas abordé dans ce chapitre. Le principe est le même que celui vu dans le chapitre « 3.4.5 Un réseau de neurones à convolution ».

4.4 Résultats

Nous pouvons voir grâce à la Figure 44, que le réseau est bien entrainé. J'ai décidé de faire seulement 25 epochs car à partir de là, l'erreur de l'ensemble de validation avait tendance à augmenter. Pour définir le bon nombre d'epochs à utiliser, j'ai tout d'abord lancé un entrainement avec 100 epochs. Cela étant, j'ai analysé à partir de quel moment le réseau commençait à surentrainer (Voir « 2.2.2.3 Le sous-apprentissage »).

Figure 44 : Erreur de l'ensemble de validation



Fait à l'aide de l'outil : TensorBoard

L'ensemble de test a calculé une erreur quadratique moyenne de 338. Ce résultat n'est donc pas optimal mais c'est le meilleur que je puisse obtenir avec cette architecture et ce paramétrage.

4.5 Production

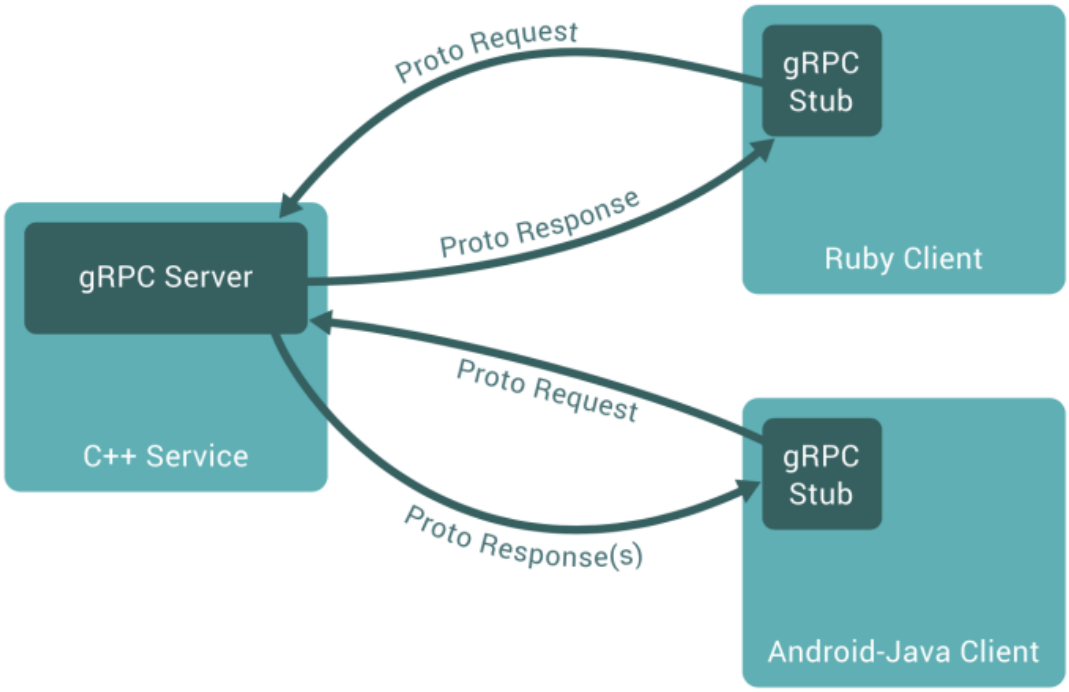
Lorsqu'un réseau de neurones est bien entraîné, il est intéressant de pouvoir le mettre en production afin que d'autres personnes puissent effectuer des prédictions sans devoir l'entrainer de nouveau.

Pour la mise en production, il faut générer un fichier qui contient les valeurs de chaque poids du réseau de neurones artificiels, le charger à l'aide d'un serveur et pouvoir le requêter pour obtenir de nouvelles prédictions.

4.5.1 gRPC

Le serveur de production TensorFlow utilise gRPC, une version adaptée de RPC, acronyme de Remote Procedure Call, qui est un protocole réseau permettant d'appeler des procédures sur un serveur distant. Cette technique d'appel de procédures rend ces appels indépendants de la technologie utilisée(Santos Lourenço, 2017).

Figure 45 : Diagramme d'appels de procédures distantes avec gRPC



(Santos Lourenço, 2017)

De plus, Google a développé son propre protocole de sérialisation : Protocol Buffer. Selon Santos Lourenço, cette technique de sérialisation est plus rapide, plus compacte et plus facile à utiliser que d'autres types de sérialisation, comme le XML(Santos Lourenço, 2017).

Pour utiliser un modèle statistique en production, nous devons utiliser gRPC. Si un client doit utiliser un modèle, il doit juste connaître l'adresse de son serveur, le nom du modèle et le nom de la procédure à appeler(Petlund, 2018). Les détails techniques du protocole gRPC et de la technique de sérialisation Protocol Buffer, sortent du cadre de mon étude. Cependant, une compréhension du concept de ce protocole est essentielle à la compréhension des rubriques suivantes.

4.5.2 Enregistrer un modèle statistique

Tout d'abord, nous devons définir les entrées et les sorties de notre modèle. Dans ce cas précis, les entrées et sorties font référence à ce que l'utilisateur va envoyer au modèle et ce que le modèle va retourner à l'utilisateur, respectivement.

```
entry_tensor = sess.graph.get_tensor_by_name('input_image:0')
out_tensor = sess.graph.get_tensor_by_name('regression:0')
```

Le String input_image :0 correspond à <nom_du_tenseur :numero>. TensorFlow va organiser les tenseurs par nom, et si un nom est répété, la librairie lui assigne un numéro pour pouvoir distinguer ce tenseur de manière unique. Le nom du tenseur est défini ainsi :

```
entrees = tf.placeholder(tf.float32, [None, image_size, image_size, channels], name="input_image")
```

Ensuite, nous devons enregistrer les informations des tenseurs pour les utiliser comme points d'entrée et sortie du modèle statistique :

```
model_input = build_tensor_info(entry_tensor)
model_output = build_tensor_info(out_tensor)
```

Ces informations seront utilisées pour créer une signature de modèle :

```
signature_definition = signature_def_utils.build_signature_def(
    inputs={'input': model_input},
    outputs={'regression': model_output},
    method_name=signature_constants.PREDICT_METHOD_NAME)
```

Cette signature est très importante car lorsque le client voudra faire une prédiction, d'où la constante PREDICT_METHOD_NAME, il devra spécifier que c'est la méthode 'input' qu'il veut appeler et ce sera un objet 'regression' qui lui sera retourné.

Puis, nous devons spécifier dans quel dossier le modèle statistique sera enregistré.

```
builder = saved_model_builder.SavedModelBuilder('./models/alexnet_counting/1')
```

Nous définissons les variables et signatures à enregistrer :

```
builder.add_meta_graph_and_variables(
    sess, [tag_constants.SERVING],
    signature_def_map={
        signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY:
            signature_definition
    })
```

La signature correspond à la procédure à appeler que le client devra spécifier lors d'une requête.

Et pour finir, notre modèle est enregistré !

```
builder.save()
```

Cette opération crée un dossier qui contient :

- Un fichier saved_model.pb – Format Protocol Buffer

- Un dossier variables
    - variables.data-00000-of-00001
    - variables.index

4.5.3 Serveur de production

Pour mettre le serveur en production, une machine virtuelle Linux Ubuntu 04.18 a été utilisée.

Dans la machine Linux, il faut installer python 2.7 et ensuite il est nécessaire d'installer le package tensorflow-serving-api à l'aide de la commande suivante (TensorFlow, 2018a):

- pip install tensorflow-serving-api

Ensuite, il faut installer tensorflow-model-server avec la commande :

- sudo apt-get install tensorflow-model-server

Une fois l'installation terminée, nous pouvons exécuter la commande :

- tensorflow_model_server –port :9000 –model_name=model
  --model_base_path= /path/to/model

Le paramètre –model_name peut recevoir n'importe quel string. Il est important de choisir un nom pertinent car lorsque le client envoie une requête au serveur, il doit spécifier à quel modèle il fait référence.

Après avoir exécuté cette commande, les messages suivants devraient s'afficher :

```
2018-06-29 12:32:04.356552: I external/org_tensorflow/tensorflow/cc/saved_mo
del/loader.cc:196] Running LegacyInitOp on SavedModel bundle.
2018-06-29 12:32:04.360497: I external/org_tensorflow/tensorflow/cc/saved_mo
del/loader.cc:291] SavedModel load for tags { serve }; Status: success. Took
 4661957 microseconds.
2018-06-29 12:32:04.367879: I tensorflow_serving/core/loader_harness.cc:86]
Successfully loaded servable version {name: model version: 1}
2018-06-29 12:32:04.439197: I tensorflow_serving/model_servers/main.cc:323]
Running ModelServer at 0.0.0.0:9000 ...
```

Le modèle a donc été bien chargé et nous pouvons créer un client qui va requêter le serveur.

4.5.4 Création d'un client

TensorFlow nous fournit des méthodes à utiliser pour effectuer toutes les requêtes nécessaires. Premièrement, nous devons déclarer où le serveur se trouve :

```
channel = implementations.insecure_channel('localhost', int(9000))
stub = prediction_service_pb2.beta_create_PredictionService_stub(channel)
```

Deuxièmement, nous devons lui spécifier qu'il s'agit d'une requête de prédiction :

```
request = predict_pb2.PredictRequest()
request.model_spec.name = 'model'
request.model_spec.signature_name = 'serving_default'
```

L'attribut spec_name doit contenir le nom du modèle déclaré dans la commande qui lance le serveur de production. L'attribut signature_name contient le nom déclaré dans la définition de signature lorsque le modèle statistique a été enregistré.

Nous préparons les données à envoyer :

```
request.inputs['input'].CopyFrom(
    tf.contrib.util.make_tensor_proto(image_to_predict,
                                      shape=[1, image_size, image_size, channels],
                                      dtype=tf.float32)
)
```

Dans l'attribut inputs, le string déclaré (« input ») fait référence au tenseur d'entrée du modèle statistique. La fonction make_tensor_proto va sérialiser l'image qui lui est passée en paramètre au format Protocol Buffer. Il faut s'assurer que la forme et le type de données sont correctement encodées afin que le serveur puisse les interpréter et l'utiliser dans le modèle. L'image envoyée doit être normalisée et être de la même taille que celle utilisée dans l'entrainement du réseau.

Pour finir, la requête peut être envoyée :

```
result = stub.Predict(request, 5.0)
```

Le deuxième paramètre correspond au timeout de la requête HTTP, et ici ce timeout est défini à 5 secondes.

En affichant le résultat retourné par le serveur, nous pouvons voir un objet qui contient une clé « regression » qui a été définie lors de l'enregistrement du modèle statistique et la prédiction pour l'image envoyée.

```
outputs {
  key: "regression"
  value {
    dtype: DT_FLOAT
    tensor_shape {
      dim {
        size: 1
      }
      dim {
        size: 1
      }
    }
    float_val: 26.4608097076
  }
}
```

Voici comment afficher le résultat de la prédiction :

```
print(result.ouputs['regression'].float_val[0], 5.0)
```

En faisait une boucle, nous pouvons afficher une prédiction pour chaque image. Voici un extrait des résultats obtenus :

```
26.4608097076
26.4608097076
26.4608097076
98.1630249023
26.4608097076
26.4608097076
68.4119033813
85.1203842163
36.9298019409
26.4608097076
26.4608097076
26.4608097076
26.4608097076
26.4608097076
26.4608097076
26.4608097076
42.2345733643
26.4608097076
```

Nous pouvons constater que la valeur 26.4608097076 est retournée presque à chaque requête. Il y a donc un problème avec ce modèle car il n'a pas appris à compter, mais il a trouvé une valeur moyenne qui donne une erreur minimale dans cet ensemble de données. Après ces résultats, j'ai formulé deux hypothèses :

- L'architecture n'est pas adaptée
- Le réseau est en train de surentraîner

La deuxième hypothèse a été écartée car le modèle a été enregistré avant que l'erreur de l'ensemble de validation commence à augmenter. Pour cette raison, le modèle enregistré est le meilleur que je puisse entrainer avec cet ensemble de données et cette architecture. Puisqu'il est chronophage de créer mon propre ensemble de données, je dois me concentrer sur l'architecture. De plus, selon ce qui a été étudié au chapitre « 2.3.2.2 Le nombre de nœuds à utiliser par couche », lorsqu'un réseau possède trop de neurones, celui-ci a tendance à surentrainer. Pour cette raison, j'ai décidé d'utiliser une architecture plus simple.

4.6 Nouvelle architecture de réseaux de neurones à convolution

Puisque je devais trouver une nouvelle architecture adaptée au problème que je devais traiter, j'ai décidé de commencer par une architecture avec peu de couches et d'y ajouter des couches jusqu'à ce que le résultat soit meilleur.

Pour commencer, j'ai décidé d'utiliser la même architecture que dans le chapitre « 3.4.5 Un réseau de neurones à convolution ». Les seules différences sont qu'un seul neurone est utilisé dans la couche de sortie et les images d'entrée font 128 par 128 pixels.

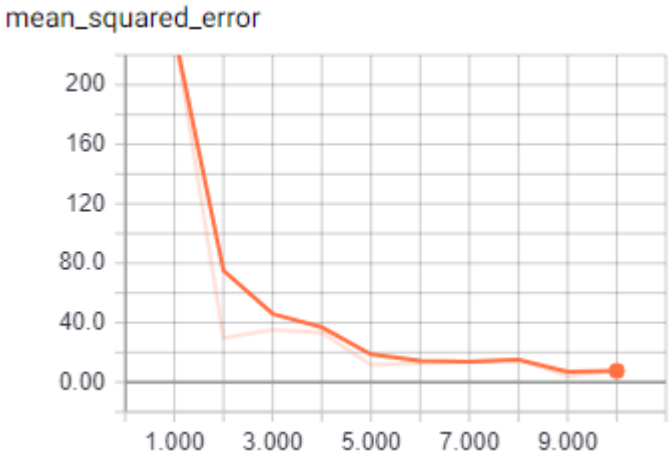
4.6.1 Entrainement

Comme dans l'architecture précédente, j'ai tout d'abord lancé un entrainement avec 100 epochs avant de décider où arrêter l'entrainement. Voici les hyperparamètres utilisés pour l'entrainement de cette nouvelle architecture :

- Taille de chaque batch : 8
- Learning rate : 0.0001
- Epochs : 10
- Probabilité de DropOut : 0.5

Voici le résultat de l'ensemble de validation :

Figure 46 : Erreur de l'ensemble de validation dans la nouvelle architecture



L'ensemble de test a calculé une erreur quadratique moyenne de 13. Ceci montre que cette architecture est bien plus adaptée au problème à traiter puisque l'architecture précédente avait une erreur quadratique moyenne de 338 !

En testant le modèle en production avec l'ensemble de test, voici les résultats retournés par le serveur qui confirment que cette fois le réseau est bien entrainé :

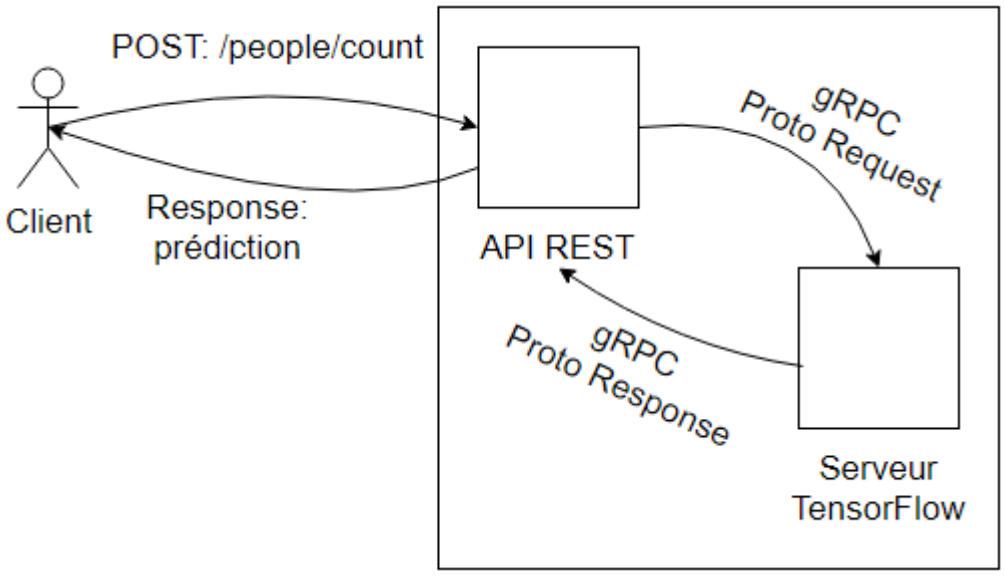
```
41.5674858093
32.4325294495
26.7615909576
30.4221172333
29.9375648499
42.3184585571
39.0631980896
48.011051178
```

Nous avons désormais un modèle en production capable de prédire combien de personnes se trouvent sur une image. Cependant, pour qu'un client puisse requêter ce serveur de production, il doit utiliser le client gRPC fournit avec la librairie TensorFlow. Or, un client ne devrait pas être dépendant d'une librairie. Pour cette raison, nous allons créer un service REST qui simplifie les requêtes à notre serveur de production.

4.6.2 Architecture avec une API REST

Tout d'abord, voici comment l'architecture de notre service de prédiction sera structuré :

Figure 47 : Architecture et interactions entre le client, le service REST et le serveur TensorFlow



Fait à l'aide de l'outil : draw.io

Le client sera donc indépendant de la technologie utilisée pour la prédiction du nombre de personnes dans une image. De plus, la responsabilité du pré-traitement des images appartient au service REST.

Pour l'implémentation de ce service, nous allons utiliser Flask, un « micro Framework » en Python permettant de créer des service REST en quelques lignes. Pour cela, nous devons l'installer :

- pip install Flask

La version utilisée dans cette démonstration est la 1.0.2.

Une fois le framework, installé, celui-ci peut être importé dans un script Python :

```
from flask import Flask
from flask import request
from flask import jsonify
```

L'import Flask nous permet de créer une nouvelle application, request sera utile pour obtenir l'image envoyée dans la requête POST et jsonify sérialise des objets Python au format JSON.

Voici comment instancier une nouvelle application :

```
my_app = Flask(__name__)
```

Ensuite, nous créons la fonction qui sera appelée lors d'une requête de la part du client :

```
@my_app.route('/people/count', methods=['POST'])
def prediction():
    try:
        image_to_predict = get_image(request)
        predict_request, stub = prepare_request(image_to_predict)
        result = stub.Predict(predict_request, 5.0)# 5 seconds
        return jsonify(prediction = str(result.outputs['regression'].float_val[0]))
    except Exception as err:
        return str(err)
```

L'annotation @my_app.route permet de déclarer la route que le client doit appeler et le verbe de la requête HTTP.

Cette fonction récupère l'image envoyée dans la requête HTTP, instancie une nouvelle requête gRPC, exécute la requête de prédiction au serveur TensorFlow et retourne le résultat au format JSON au client.

```
def get_image(request):
    image_file = request.files.get('image','')
    image_decoded = tf.image.decode_jpeg(image_file.read())
    image_resized = tf.image.resize_images(image_decoded, [128, 128])
    image_to_return = tf.image.per_image_standardization(image_resized)
    sess = tf.Session()
    return sess.run(image_to_return)
```

La première ligne de cette fonction récupère l'image envoyée dans la requête. Les lignes suivantes font le pré-traitement sur cette image pour l'envoyer au serveur TensorFlow. Chaque ligne a été expliquée dans le chapitre « 4.2.1 Chargement des images ».

```
def prepare_request(image_to_predict):
    # create connection
    channel = implementations.insecure_channel('localhost', int(9000))
    stub = prediction_service_pb2.beta_create_PredictionService_stub(channel)

    # initialize a request
    request_to_predict = predict_pb2.PredictRequest()
    request_to_predict.model_spec.name = 'model'
    request_to_predict.model_spec.signature_name = 'serving_default'

    request_to_predict.inputs['input'].CopyFrom(
    tf.contrib.util.make_tensor_proto(image_to_predict, shape=[1, 128, 128, 3], dtype=tf.float32)
    )
    return request_to_predict, stub
```

Pour comprendre le fonctionnement de chaque ligne de la fonction de préparation de la requête gRPC, veuillez consulter le chapitre « 4.5.4 Création d'un client ».

Une fois ces fonctions déclarées, le serveur REST peut être lancé :

```
if __name__ == '__main__':
    my_app.run(debug=True, host='0.0.0.0')
```

La condition ici présente s'assure que ce script est celui lancé par l'utilisateur. C'est-à-dire, s'il est importé par un autre fichier, la condition ne sera pas satisfaite et le serveur n'est pas lancé.

Pour finir, nous lançons l'application. Le paramètre debug permet d'afficher des messages d'erreur en cas de problèmes avec le serveur. Celui-ci doit être « False » en cas réel de production pour éviter d'exposer d'éventuels problèmes de sécurité. Le paramètre host n'est pas obligatoire. Cependant, ici nous devons l'indiquer de cette manière pour que l'application soit visible par des clients externes à la machine sur laquelle le serveur est exécuté.

Pour lancer le serveur, il faut exécuter la commande suivante :

```
flavio@flavio:~/Bureau/flask_tensorflow_api$ python app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 296-929-623
```

Le fichier app.py correspond au nom du script créé pour l'application Flask. Celui-ci peut avoir un autre nom. Si le message « Running on … » s'affiche, le serveur peut recevoir des requêtes http pour effectuer une prédiction auprès du serveur TensorFlow.

Pour requêter ce service, il faut spécifier les attributs suivants dans les headers :

- Content-Type : multipart/form-data
- Accept: application/json

L'image doit être envoyée dans un attribut « image ». Et le verbe utilisé doit être POST à l'URL : http://localhost:5000/people/count.

Une fois la requête envoyée, la réponse du serveur est la suivante :

```
{
  "prediction": "36.4452819824"
}
```

Le client est donc indépendant de la technologie, ne sait pas quelle librairie est utilisée et ne doit pas prétraiter son image.

4.6.3 Fiabilité du modèle statistique

À l'aide du service REST créé dans la rubrique précédente, le réseau a été testé avec l'image suivante Figure 48, qui contient 30 personnes, et celui-ci a prédit 30.17 personnes, même ayant des personnes assises où seulement une petite partie des personnes est visible (cercle rouge).

Figure 48 : Image testée pour confirmer la précision du modèle statistique

(Loy, 2014)

Mais qu'arrive-t-il lorsque de nouvelles images, qui n'ont aucun lien avec ce centre commercial, sont utilisées dans ce réseau ?

Figure 49 : Image de test de fiabilité du modèle statistique ne faisant pas partie du jeu de données utilisé

(Zawya, 2017)

Dans cette image j'ai compté 39 personnes. Les reflets et les mannequins ne sont pas comptés pour respecter le comptage utilisé dans le jeu de données d'entrainement. Le modèle statistique a calculé qu'il y a environ 79 personnes dans cette image. Nous

pouvons donc conclure que ce réseau est bien entrainé mais seulement d'un seul point de vue.

4.6.3.1 Améliorer la précision du modèle statistique

Pour améliorer la précision du modèle statistique utilisé, il aurait fallu utiliser un jeu de données plus grand, avec des points de vue variés. En effet, comme vu au chapitre « 2.2.2.3 Le sous-apprentissage », pour qu'un réseau de neurones artificiels puisse être généralisable pour tous les cas, il est nécessaire que celui-ci soit entrainé sur un panel d'exemples variés. Aussi, il serait intéressant de savoir si en ajoutant des points de vue différents, les résultats des deux architectures changeraient.

Pour finir, si l'objectif d'une entreprise est de compter les personnes présentes dans une pièce d'un seul point de vue, les résultats sont assez satisfaisants pour utiliser ce système en production.

5. Conclusion

En conclusion, les réseaux de neurones artificiels sont un système avec plein de potentiel. Ils peuvent reconnaître des objets sur une image, les compter et comprendre le langage naturel. Cependant, cette technologie doit encore se développer car aujourd'hui il n'est pas encore possible de définir une architecture à partir d'un problème donné. Ceci impose le problème de ne jamais savoir si une autre architecture pourrait donner de meilleurs résultats. De plus, pour les débutants en *machine learning,* comme moi, la multitude de paramètres possibles dans un tel système est intimidant.

Ensuite, la librairie TensorFlow est une librairie performante avec une large communauté qui la soutient. Elle est difficile à prendre en main dû aux sessions qui doivent être déclarées explicitement. Mais, les niveaux d'abstraction fournis par la librairie la rendent puissante. De plus, les autres outils qu'elle intègre, comme TensorBoard et le serveur de production, aident les développeurs à travailler dans un seul et unique écosystème.

Pour finir, le prototype réalisé m'a permis de consolider les connaissances apprises lors des deux premiers chapitres. La réalisation du prototype et de ce travail de bachelor a été une expérience enrichissante, pour moi, au point où elle m'a aidé à mieux définir mon parcours académique. En effet, j'ai décidé de suivre un Master en Sciences de l'Information dans lequel les concepts de *deep learning* sont abordés.

Bibliographie

AGGARWAL, Keshav, 2018. Tensorflow Image: Augmentation on GPU. In : *Towards Data Science* [en ligne]. 17 avril 2018. [Consulté le 29 juin 2018]. Disponible à l'adresse : https://towardsdatascience.com/tensorflow-image-augmentation-on-gpu-bf0eaac4c967.

ALI, Raja Hashim, SAFRI, Murtaza, BHATTI, Iqbal Talaat et SAID, Wajiah, 2006. LipSync - lip synchronization with speech. In : [en ligne]. mai 2006. [Consulté le 30 juin 2018]. Disponible à l'adresse : https://www.researchgate.net/publication/306038981_LipSync_-_lip_synchronization_with_speech?_sg=SAGQXxnZNULRdqZIDVNLWdjV9DMQ251kz0j3vuQQuu_7dbgueLUdyL8cX8yuJ1O0axHk_VHgqw.

ALLEN, Rachel et LI, Michael, 2017. Ranking Popular Deep Learning Libraries for Data Science. In : [en ligne]. 1 août 2017. [Consulté le 4 mai 2018]. Disponible à l'adresse : https://www.kdnuggets.com/2017/10/ranking-popular-deep-learning-libraries-data-science.html.

BASTIEN L, 2018. Machine Learning et Big Data : définition de l'apprentissage automatique. In : *LeBigData.fr* [en ligne]. 16 avril 2018. [Consulté le 19 avril 2018]. Disponible à l'adresse : https://www.lebigdata.fr/machine-learning-et-big-data.

BEEMAN, Dave, 2001. Multi-layer perceptrons and back propagation. In : [en ligne]. 1 novembre 2001. [Consulté le 30 juin 2018]. Disponible à l'adresse : http://ecee.colorado.edu/~ecen4831/lectures/NNet3.html.

BENZAKI, Younes, 2017. Overfitting et Underfitting : Quand vos algorithmes de Machine Learning dérapent ! In : *Mr. Mint : Apprendre le Machine Learning de A à Z* [en ligne]. 11 juillet 2017. [Consulté le 20 mai 2018]. Disponible à l'adresse : https://mrmint.fr/overfitting-et-underfitting-quand-vos-algorithmes-de-machine-learning-derapent.

BOYD, Matthew et WILSON, Nick, 2017. Rapid Developments in Artificial Intelligence. In : . novembre 2017. p. 8.

BRAIN, In The et ROSENBLATT, F., 1958. *The Perceptron: A Probabilistic Model for Information Storage and Organization*. S.l. : s.n.

BROWNLEE, Jason, 2017a. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. In : *Machine Learning Mastery* [en ligne]. 3 juillet 2017. [Consulté le 19 juin 2018]. Disponible à l'adresse : https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.

BROWNLEE, Jason, 2017b. What is the Difference Between a Parameter and a Hyperparameter? In : *Machine Learning Mastery* [en ligne]. 26 juillet 2017. [Consulté le 22 mai 2018]. Disponible à l'adresse : https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/.

BUDHIRAJA, Amar, 2016. Learning Less to Learn Better — Dropout in (Deep) Machine learning. In : *Medium* [en ligne]. 15 décembre 2016. [Consulté le 30 juin 2018]. Disponible à l'adresse : https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5.

CHAOUCHE, Yannis, 2018. Identifiez les différents types de problèmes de machine learning. In : *OpenClassrooms* [en ligne]. 14 mai 2018. [Consulté le 20 mai 2018]. Disponible à l'adresse : https://openclassrooms.com/courses/initiez-vous-au-machine-learning/identifiez-les-differents-types-de-problemes-de-machine-learning.

CHARPENTIER, Jacqueline, 2018. Les questions éthiques soulevées par l'intelligence artificielle dans la médecine. In : [en ligne]. 14 mars 2018. [Consulté le 1 juillet 2018]. Disponible à l'adresse : https://actualite.housseniawriting.com/science/intelligence-artificielle/2018/03/15/les-questions-ethiques-soulevees-par-lintelligence-artificielle-dans-la-medecine/26138/.

COMPUTERSCIENCEWIKI, 2018. Max-pooling / Pooling - Computer Science Wiki. In : [en ligne]. 27 février 2018. [Consulté le 30 juin 2018]. Disponible à l'adresse : https://computersciencewiki.org/index.php/Max-pooling_/_Pooling.

CYBENKO, G, 1989. Approximation by superpositions of a sigmoidal function. In : . 1989. p. 12.

DELAUNAY, David, 2016. Mathématiques Sup et spé. In : [en ligne]. 2016. [Consulté le 30 juin 2018]. Disponible à l'adresse : http://mp.cpgedupuydelome.fr.

DESHPANDE, Adit, 2016. A Beginner's Guide To Understanding Convolutional Neural Networks Part 2. In : [en ligne]. 17 2016. [Consulté le 21 mai 2018]. Disponible à l'adresse : https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/.

EARLY STOPPING, 2018. *Early stopping* [en ligne]. S.l. : s.n. [Consulté le 20 mai 2018]. Disponible à l'adresse : https://en.wikipedia.org/w/index.php?title=Early_stopping&oldid=841537378.

EBERT-UPHOFF, Imme et GIL, Yolanda, 2015. Exploring Synergies between Machine Learning and Knowledge Representation to Capture Scientific Knowledge. In : . octobre 2015. p. 9.

ELECTRONICS, 2013. Exclusive-OR Gate Tutorial with Ex-OR Gate Truth Table. In : *Basic Electronics Tutorials* [en ligne]. 21 août 2013. [Consulté le 18 avril 2018]. Disponible à l'adresse : https://www.electronics-tutorials.ws/logic/logic_7.html.

FAMIEN, Olivier, 2017. TensorFlow, la bibliothèque d'apprentissage automatique de Google est disponible en version 1.0, avec plus performance et compatible avec Java et Go. In : *Developpez.com* [en ligne]. 16 février 2017. [Consulté le 4 mai 2018]. Disponible à l'adresse : http://www.developpez.com/actu/118059/TensorFlow-la-bibliotheque-d-apprentissage-automatique-de-Google-est-disponible-en-version-1-0-avec-plus-performance-et-compatible-avec-Java-et-Go/.

GAO, Hao, 2017. A Walk-through of AlexNet. In : *Hao Gao* [en ligne]. 7 août 2017. [Consulté le 28 juin 2018]. Disponible à l'adresse : https://medium.com/@smallfishbigsea/a-walk-through-of-alexnet-6cbd137a5637.

GEITGEY, Adam, 2014. Machine Learning is Fun! In : *Adam Geitgey* [en ligne]. 5 mai 2014. [Consulté le 3 avril 2018]. Disponible à l'adresse : https://medium.com/@ageitgey/machine-learning-is-fun-80ea3ec3c471.

GENÈVE, 2014. Bâtiment des lits 2 | Construction - Chantiers et travaux - Etat de Genève. In : [en ligne]. 2014. [Consulté le 30 juin 2018]. Disponible à l'adresse : https://www.ge.ch/construction/chantiers-travaux/battelle.asp.

GÉRON, Aurélien, 2018. Up and Running with TensorFlow. In : [en ligne]. 2018. [Consulté le 30 juin 2018]. Disponible à l'adresse : https://www.safaribooksonline.com/library/view/hands-on-machine-learning/9781491962282/ch09.html.

GOOGLE, 2017. MNIST For ML Beginners. In : *TensorFlow* [en ligne]. 15 juin 2017. [Consulté le 18 avril 2018]. Disponible à l'adresse : https://www.tensorflow.org/versions/r1.1/get_started/mnist/beginners.

HEATON, Jeff, 2017. The Number of Hidden Layers. In : *Heaton Research* [en ligne]. 1 juin 2017. [Consulté le 21 mai 2018]. Disponible à l'adresse : https://www.heatonresearch.com/2017/06/01/hidden-layers.html.

HINTON, Geoffrey E., OSINDERO, Simon et TEH, Yee-Whye, 2006. A Fast Learning Algorithm for Deep Belief Nets. In : *Neural Computation*. juillet 2006. Vol. 18, n° 7, p. 1527-1554. DOI 10.1162/neco.2006.18.7.1527.

HODJAT, Author: Babak, HODJAT, Sentient Technologies Babak et TECHNOLOGIES, Sentient, 2015. The AI Resurgence: Why Now? In : *WIRED* [en ligne]. mars 2015. [Consulté le 1 juillet 2018]. Disponible à l'adresse : https://www.wired.com/insights/2015/03/ai-resurgence-now/.

HORNIK, KURT, 1991. Approximation Capabilities of Muitilayer Feedforward Networks. In : . 1991. p. 7.

IMAGENET, 2017. ImageNet Large Scale Visual Recognition Competition (ILSVRC). In : [en ligne]. 2017. [Consulté le 21 mai 2018]. Disponible à l'adresse : http://www.image-net.org/challenges/LSVRC/.

ISAAC, Jackson et HARIKUMAR, Sandhya, 2016. Logistic regression within DBMS. In : . S.l. : s.n. 1 décembre 2016. p. 661-666.

IVANOV, Slav, 2017. 37 Reasons why your Neural Network is not working. In : *Slav* [en ligne]. 25 juillet 2017. [Consulté le 20 mai 2018]. Disponible à l'adresse : https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607.

JANECEK, 2007. 2007 02 01b Janecek Perceptron | Computational Neuroscience | Artificial Neural Network. In : *Scribd* [en ligne]. 1 février 2007. [Consulté le 17 avril 2018]. Disponible à l'adresse : https://www.scribd.com/document/326923737/2007-02-01b-Janecek-Perceptron.

JEAN DEAN, 2017. Build and train machine learning models on our new Google Cloud TPUs. In : *Google* [en ligne]. 17 mai 2017. [Consulté le 4 mai 2018]. Disponible à l'adresse : https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning/.

JORDAN, Jeremy, 2018. Normalizing your data (specifically, input and batch normalization). In : *Jeremy's Blog* [en ligne]. 27 janvier 2018. [Consulté le 17 avril 2018]. Disponible à l'adresse : https://www.jeremyjordan.me/batch-normalization/.

KARPATHY, 2018. CS231n Convolutional Neural Networks for Visual Recognition. In : [en ligne]. 12 juin 2018. [Consulté le 30 juin 2018]. Disponible à l'adresse : http://cs231n.github.io/convolutional-networks/.

KATE ALLEN, 2015. How a Toronto professor's research revolutionized artificial intelligence. In : [en ligne]. 17 avril 2015. [Consulté le 21 mai 2018]. Disponible à l'adresse : https://www.thestar.com/news/world/2015/04/17/how-a-toronto-professors-research-revolutionized-artificial-intelligence.html.

KRIZHEVSKY, Alex, SUTSKEVER, Ilya et HINTON, Geoffrey E., 2017. ImageNet classification with deep convolutional neural networks. In : *Communications of the ACM*. 24 mai 2017. Vol. 60, n° 6, p. 84-90. DOI 10.1145/3065386.

L, +Bastien, 2018. Perceptron – Tout savoir sur le plus vieil algorithme de Machine Learning. In : *LeBigData.fr* [en ligne]. 29 mars 2018. [Consulté le 17 avril 2018]. Disponible à l'adresse : https://www.lebigdata.fr/perceptron-machine-learning.

LOGO, 2013. Why do We Need a Bias Neuron? In : [en ligne]. 16 juillet 2013. [Consulté le 18 avril 2018]. Disponible à l'adresse : http://www.chioka.in/why-do-we-need-a-bias-neuron/.

LOY, Chen Change, 2014. Mall Dataset - Crowd Counting Dataset. In : [en ligne]. avril 2014. [Consulté le 28 juin 2018]. Disponible à l'adresse : http://personal.ie.cuhk.edu.hk/~ccloy/downloads_mall_dataset.html.

LUCIDCHART, 2017. Qu'est-ce qu'un diagramme de flux de données ? In : *Lucidchart* [en ligne]. 8 février 2017. [Consulté le 22 mai 2018]. Disponible à l'adresse : https://www.lucidchart.com/pages/fr/quest-ce-quun-diagramme-de-flux-de-donn%C3%A9es%C2%A0.

MAHESH, Nidhin, 2017. Understanding a TensorFlow program in simple steps. In : *Towards Data Science* [en ligne]. 26 septembre 2017. [Consulté le 5 mai 2018]. Disponible à l'adresse : https://towardsdatascience.com/understanding-fundamentals-of-tensorflow-program-and-why-it-is-necessary-94cf5b60e255.

MARTIN GÖRNER, 2017. *Tensorflow and deep learning - without a PhD by Martin Görner* [en ligne]. 26 décembre 2017. [Consulté le 6 mai 2018]. Disponible à l'adresse : https://www.youtube.com/watch?v=zAFzWimsIn8.

MATHWORKS, 2018. Train Convolutional Neural Network for Regression - MATLAB & Simulink - MathWorks Switzerland. In : [en ligne]. 2018. [Consulté le 2 juillet 2018]. Disponible à l'adresse : https://ch.mathworks.com/help/nnet/examples/train-a-convolutional-neural-network-for-regression.html.

MESTAN, Alp, 2008. Introduction aux Réseaux de Neurones Artificiels Feed Forward. In : *Developpez.com* [en ligne]. 1 janvier 2008. [Consulté le 18 avril 2018]. Disponible à l'adresse : http://alp.developpez.com/tutoriels/intelligence-artificielle/reseaux-de-neurones/.

MINSKY, Mavin et PAPERT, Seymor A., 1969. *Perceptrons: An Introduction to Computational Geometry, Expanded Edition*. S.l. : s.n. ISBN 978-0-262-13043-1.

MOORE, G.E., 1965. Cramming More Components Onto Integrated Circuits. In : *Proceedings of the IEEE*. 19 avril 1965. Vol. 86, n° 1, p. 82-85. DOI 10.1109/JPROC.1998.658762.

MYO, 2016. Make Your Own Neural Network: Bias Nodes in Neural Networks. In : *Make Your Own Neural Network* [en ligne]. 28 juin 2016. [Consulté le 30 juin 2018]. Disponible à l'adresse : http://makeyourownneuralnetwork.blogspot.com/2016/06/bias-nodes-in-neural-networks.html.

NAIR, Vinod et HINTON, Geoffrey E, 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In : . 21 juin 2010. p. 8.

NEEL, 2015. Training Neural Networks with Genetic Algorithms. In : *One Life* [en ligne]. 3 mars 2015. [Consulté le 30 juin 2018]. Disponible à l'adresse : https://blog.abhranil.net/2015/03/03/training-neural-networks-with-genetic-algorithms/.

NIELSEN, Michael A., 2015. Neural Networks and Deep Learning. In : [en ligne]. 2015. [Consulté le 30 juin 2018]. Disponible à l'adresse : http://neuralnetworksanddeeplearning.com.

NIELSEN, Michael A., 2017. Neural Networks and Deep Learning. In : [en ligne]. 2 décembre 2017. [Consulté le 18 avril 2018]. Disponible à l'adresse : http://neuralnetworksanddeeplearning.com.

PAULY, Leo, PEEL, Harriet, LUO, Shan, HOGG, David et FUENTES, Raul, 2017. Deeper Networks for Pavement Crack Detection. In : . S.l. : s.n. 3 juillet 2017.

PETLUND, Stian Lind, 2018. TensorFlow Serving 101 pt. 2. In : *epigramAI* [en ligne]. 28 janvier 2018. [Consulté le 29 juin 2018]. Disponible à l'adresse : https://medium.com/epigramai/tensorflow-serving-101-pt-2-682eaf7469e7.

RAJARAJAN, Jagadeesh, 2015. What is « Multilayer perceptrons using backpropagation algorithm », in simple words? - Quora. In : [en ligne]. 23 février 2015. [Consulté le 30 juin 2018]. Disponible à l'adresse : https://www.quora.com/What-is-Multilayer-perceptrons-using-backpropagation-algorithm-in-simple-words.

ROJAS, R, 1996. *Perceptron Learning* [en ligne]. 1996. S.l. : s.n. [Consulté le 18 avril 2018]. Disponible à l'adresse : https://page.mi.fu-berlin.de/rojas/neural/chapter/K4.pdf.

SAITTA, Sandro, 2010. What is a good classification accuracy in data mining? | Data Mining Blog - www.dataminingblog.com. In : [en ligne]. 11 avril 2010. [Consulté le 20 mai 2018]. Disponible à l'adresse : http://www.dataminingblog.com/what-is-a-good-classification-accuracy-in-data-mining/.

SANTOS LOURENÇO, Alexandre, 2017. grpc – Technology explained. In : [en ligne]. 2 mai 2017. [Consulté le 29 juin 2018]. Disponible à l'adresse : https://alexandreesl.com/tag/grpc/.

SCHULTZ, Jeff, 2017. How Much Data is Created on the Internet Each Day? | Micro Focus Blog. In : [en ligne]. 10 octobre 2017. [Consulté le 1 juillet 2018]. Disponible à l'adresse : https://blog.microfocus.com/how-much-data-is-created-on-the-internet-each-day/#.

SHAH, Tarang, 2017. About Train, Validation and Test Sets in Machine Learning. In : *Towards Data Science* [en ligne]. 6 décembre 2017. [Consulté le 18 avril 2018]. Disponible à l'adresse : https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7.

SHARMA, Avinash, 2017. Understanding Activation Functions in Neural Networks. In : *Medium* [en ligne]. 30 mars 2017. [Consulté le 18 avril 2018]. Disponible à l'adresse : https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.

SHARMA, SAGAR, 2017a. Epoch vs Batch Size vs Iterations. In : *Towards Data Science* [en ligne]. 23 septembre 2017. [Consulté le 2 juillet 2018]. Disponible à l'adresse : https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9.

SHARMA, SAGAR, 2017b. What the Hell is "Tensor" in "TensorFlow"? In : *Hacker Noon* [en ligne]. 9 septembre 2017. [Consulté le 5 mai 2018]. Disponible à l'adresse : https://hackernoon.com/what-the-hell-is-tensor-in-tensorflow-e40dbf0253ee.

SINHA, Utkarsh, 2010. Convolutions: Image convolution examples - AI Shack. In : [en ligne]. 2010. [Consulté le 21 mai 2018]. Disponible à l'adresse : http://aishack.in/tutorials/image-convolution-examples/.

SRIVASTAVA, Nitish, HINTON, Geoffrey, KRIZHEVSKY, Alex, SUTSKEVER, Ilya et SALAKHUTDINOV, Ruslan, 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In : . 2014. p. 30.

SURMENOK, Pavel, 2017. Estimating an Optimal Learning Rate For a Deep Neural Network. In : *Towards Data Science* [en ligne]. 13 novembre 2017. [Consulté le 6 mai 2018]. Disponible à l'adresse : https://towardsdatascience.com/estimating-optimal-learning-rate-for-a-deep-neural-network-ce32f2556ce0.

SZEGEDY, Christian, WEI LIU, YANGQING JIA, SERMANET, Pierre, REED, Scott, ANGUELOV, Dragomir, ERHAN, Dumitru, VANHOUCKE, Vincent et RABINOVICH, Andrew, 2015. Going deeper with convolutions. In : [en ligne]. S.l. : IEEE. juin 2015. p. 1-9. [Consulté le 21 mai 2018]. Disponible à l'adresse : http://ieeexplore.ieee.org/document/7298594/.

TADEUSIEWICZ, RYSZARD, 2018. Pierwszy elektroniczny model mózgu. In : *naTemat.pl* [en ligne]. 2018. [Consulté le 30 juin 2018]. Disponible à l'adresse : http://ryszardtadeusiewicz.natemat.pl/129195,pierwszy-dzialajacy-techniczny-model-mozgu.

TENSORFLOW, 2017. MNIST For ML Beginners. In : *TensorFlow* [en ligne]. 26 avril 2017. [Consulté le 30 juin 2018]. Disponible à l'adresse : https://www.tensorflow.org/versions/r1.0/get_started/mnist/beginners.

TENSORFLOW, 2018a. Installation. In : *TensorFlow* [en ligne]. 25 mai 2018. [Consulté le 29 juin 2018]. Disponible à l'adresse : https://www.tensorflow.org/serving/setup.

TENSORFLOW, 2018b. Installing TensorFlow for Java. In : *TensorFlow* [en ligne]. 28 avril 2018. [Consulté le 4 mai 2018]. Disponible à l'adresse : https://www.tensorflow.org/install/install_java.

TENSORFLOW, 2018c. Premade Estimators. In : *TensorFlow* [en ligne]. 28 avril 2018. [Consulté le 22 mai 2018]. Disponible à l'adresse : https://www.tensorflow.org/get_started/premade_estimators.

UJJWALKARN, 2016. An Intuitive Explanation of Convolutional Neural Networks. In : *the data science blog* [en ligne]. 10 août 2016. [Consulté le 18 avril 2018]. Disponible à l'adresse : https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.

UNRUH, Unruh, 2017. What is the TensorFlow machine intelligence platform? In : *Opensource.com* [en ligne]. 9 novembre 2017. [Consulté le 4 mai 2018]. Disponible à l'adresse : https://opensource.com/article/17/11/intro-tensorflow.

USBDATA, [sans date]. Rosenblatt Perceptron 1589 | USBDATA. In : [en ligne]. [Consulté le 30 juin 2018]. Disponible à l'adresse : http://www.usbdata.co/rosenblatt-perceptron.html.

VICTOR POWELL, 2015. Image Kernels explained visually. In : *Explained Visually* [en ligne]. 24 avril 2015. [Consulté le 21 mai 2018]. Disponible à l'adresse : http://setosa.io/ev/image-kernels/.

WARREN DAVIS, 2015. What is a tensor? In : [en ligne]. 2015. [Consulté le 22 mai 2018]. Disponible à l'adresse : http://www.physlink.com/education/askexperts/ae168.cfm.

WIKIPEDIA, 2018. *Iris de Fisher* [en ligne]. S.l. : s.n. [Consulté le 17 avril 2018]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Iris_de_Fisher&oldid=145827119.

WILLEMS, Karlijn, 2017. TensorFlow Tutorial For Beginners. In : *DataCamp Community* [en ligne]. 13 juillet 2017. [Consulté le 5 mai 2018]. Disponible à l'adresse : https://www.datacamp.com/community/tutorials/tensorflow-tutorial.

ZAWYA, 2017. Azadea to open 12 new stores at Mall of Egypt. In : [en ligne]. 30 novembre 2017. [Consulté le 30 juin 2018]. Disponible à l'adresse : https://www.zawya.com/uae/en/story/Azadea_to_open_12_new_stores_at_Mall_of_Egypt-SNG_104614840/.

Annexe 1 : Document de vision

Introduction

Objectifs du document

Ce document a pour objectif de donner une meilleure vision de l'ensemble des sujets à traiter lors de la réalisation de ce même travail de bachelor.

Ce document de vision est une adaptation d'un document qui vise des produits plutôt commerciaux. Je décris donc ici la vision de mon travail de bachelor et le prototype que je souhaite créer d'une manière académique et non d'une manière qui a pour objectif de créer un produit lucratif.

Portée

Le sujet de ce travail de bachelor porte sur l'application spécifique des réseaux de neurones aux caméras de surveillance tout en gardant l'anonymat des personnes. L'idée serait de pouvoir appliquer des techniques récentes de *machine learning* à un cas utile et pertinent.

Définitions, acronymes et abréviations

Machine Learning : Anglicisme – « Apprentissage automatique des machines », est un programme qui, à l'aide d'une énorme quantité de données utilisée en tant qu'ensemble d'apprentissage, peut réaliser des opérations complexes et apporter des solutions à des problèmes compliqués.[1]

Références

Vue générale du document

Ce document sera divisé en 5 parties :

- Positionnement
- Description des intervenants et des utilisateurs
- Vue d'ensemble du produit
- Caractéristiques essentielles du produit
- Tolérances de qualité non fonctionnelles

[1] https://digitalinsiders.feelandclic.com/construire/definition-quest-machine-learning

Positionnement

Opportunité commerciale

Les entités responsables possédant des caméras ou autres manières de récolter des informations non exploitables à cause des problèmes d'anonymat, pourraient enfin les exploiter et en obtenir des statistiques étant donné que les images ne seraient pas stockées mais seulement analysées.

Position du problème

Le problème	Informations non exploitables afin de garantir l'anonymat des personnes
Affecte	Entreprises, gouvernements…
L'impact du problème est	Les entités affectées ne peuvent pas développer de nouvelles solutions dû aux freins imposés par la législation
Une solution satisfaisante serait	Rendre la plupart des informations exploitables

Position du produit

Pour	Entreprises, gouvernements…
Qui	-
Le <nom produit>	-
Qui	-
A la différence de	-
Notre produit	-

Description des intervenants et des utilisateurs

Les intervenants

Nom	Description	Rôle
M. Philippe Dugerdil	Se charge de l'encadrement du Travail de Bachelor	Directeur de mémoire

Profil des intervenants

Représentant	M. Philippe Dugerdil
Description	Chargé de suivre l'avancement du travail de bachelor.
Type	Professeur HES et Responsable de la Recherche
Responsabilités[2]	• Assurer la pertinence du sujet, des objectifs proposés ainsi que la faisabilité du travail de bachelor • Accepter le sujet • Définir la période de début et de fin ainsi que la date de soutenance • Signer la convention de travail de bachelor • Suivre l'état d'avancement du travail de l'étudiant • Conseiller l'étudiant dans le cadre de la rédaction du mémoire et l'assister en cas de difficulté • Évaluer le mémoire • Participer à l'évaluation de la soutenance orale

[2] https://www.hesge.ch/heg/sites/default/files/formation-base/IG/plan-modulaire/plans-print-2018/fiches-modulaires-print-2018/656.pdf

Vue d’ensemble du produit

Perspective du produit

Le mémoire sera constitué essentiellement de deux parties :

- Théorique
- Prototype

La partie théorique aura pour objectif d’analyser les solutions et méthodes déjà existantes. Les méthodes qui devront être analysées sont les suivantes :´

- Réseau de neurones multicouches
- Réseau de neurones à convolution (CNN)
- D’autres méthodes de machine learning qui puissent être intéressantes

Cette partie sera la base de la partie de prototype. En effet, le prototype sera créé selon les recherches et les résultats trouvés.

Le prototype devra être capable de passer par une phase d’apprentissage qui lui donnera le moyen de prédire certaines statistiques liées aux images qui lui sont données. C’est-à-dire, le système reçoit des milliers d’images avec un résultat attendu. Au bout d’un certain nombre d’itérations, celui-ci doit pouvoir prédire, avec un pourcentage de performance élevé, les statistiques attendues.

Caractéristiques essentielles du produit

- Apprentissage sur un ensemble d'images
- Prédiction sur la base d'une image

Packaging, labelling, copyright

Selon le plan modulaire, le mémoire du travail de bachelor appartient à la Haute École de Gestion de Genève étant donné que je n'utiliserai pas d'informations collectées en entreprise et n'ai pas de clause de confidentialité. L'école a donc le droit de diffuser ce travail en entier ou en partie.[3]

[3] https://www.hesge.ch/heg/sites/default/files/formation-base/IG/plan-modulaire/plans-print-2018/fiches-modulaires-print-2018/656.pdf