Rapport de TIPE
Réseaux de neurones convolutifs pour le guidage automatique d'un véhicule autonome

Mohamed KOUHOU

Juin 2019

Résumé

Loosely inspired by biological brain, artificial neural networks (ANNs) are computational tools that are used for prediction. They are optimized by machine learning methods in order to be able to approximate a function that maps between inputs and outputs. ANNs are commonly employed in artificial intelligence especially for image recognition, classification, and data processing. Our goal is to « train » an autonomous vehicle by approximating the function that maps between the images received from the road and the appropriate steering angle and speed. Since convolutional neural networks (CNNs) are better suited for image processing, a CNN model will be adopted.

1 Perceptron multicouche

1.1 Introduction

Définition

- Un **réseau de neurones artificiels** est un système dont l'idée est inspirée du fonctionnement des neurones biologiques. Optimisé par les méthodes d'**apprentissage automatique** (*machine learning*), il est employé pour prédire des résultats à partir de données fournies en entrée. Les réseaux de neurones artificiels trouvent plusieurs applications notamment dans le domaine de l'intelligence artificielle, telles que le traitement d'image, la traduction, le contôle de qualité, la classification, ... etc [5]
- Un **Perceptron multicouche** (MLP : ***M**ulti**L**ayer **P**erceptron*) est un type de réseau de neurones artificiels. Il est constitué d'un ensemble de **nœuds**, appelés aussi **neurones**, organisés en **couches**, celles-ci étant reliées par des connexions pondérées (**poids**)

Il y a trois type de couches :

— **Couche d'entrée** (*Input Layer*) : elle reçoit les valeurs d'entrée ;

— **Couches cachées** (*Hidden Layers*) : il peut y en avoir une ou plusieurs selon la profondeur du réseau. Les valeurs de chaque couche cachée sont calculées à partir des valeurs de la couche qui lui précède ;

— **Couche de sortie** (*Ouput Layer*) : elle donne le résultat final issu de toutes les autres couches.

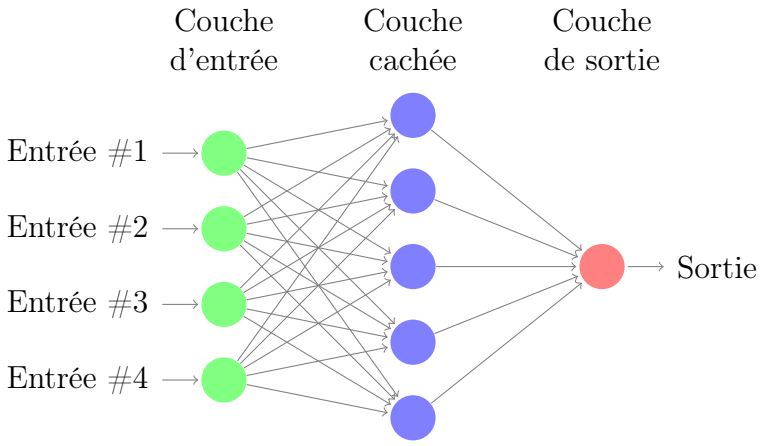


FIGURE 1 – Architecture d'un perceptron multicouche avec une couche cachée

ATTENTION ! Les neurones d'une même couche ne sont pas reliés entre eux.

— Les neurones d'entrée sont destinés à recevoir des valeurs numériques (comme, par exemple, les pixels d'une image)

— Cette information numérique se propage vers le reste du réseau à travers les connexions qui portent chacune un poids (scalaire) traduisant la contribution du neurone.

L'**entraînement** (ou l'**apprentissage**) d'un MLP se fait en deux phases : la phase du calcul des prédictions (*feedforward*), et la phase de la mise à jour des paramètres par la rétropropagation (*backpropagation*).

1.2 Phase de la propagation directe : le *Feedforward*

C'est la phase pendant laquelle on calcule les prédictions. Les poids des connexions étant initialisés, *a priori*, arbitrairement, le signal se propage dans le sens direct (de l'entrée vers la sortie) et donne alors un résultat arbitraire.

Considérons un perceptron multicouche dont la couche d'entrée est constituée de n neurones. On note :

$x_i^{(k)}$: la valeur du $i^{\text{ème}}$ neurone de la $k^{\text{ème}}$ couche
$w_{ji}^{(k)}$: le poids de la connexion entre le $i^{\text{ème}}$ neurone de la $(k-1)^{\text{ème}}$ couche et le $j^{\text{ème}}$ neurone de la $k^{\text{ème}}$ couche

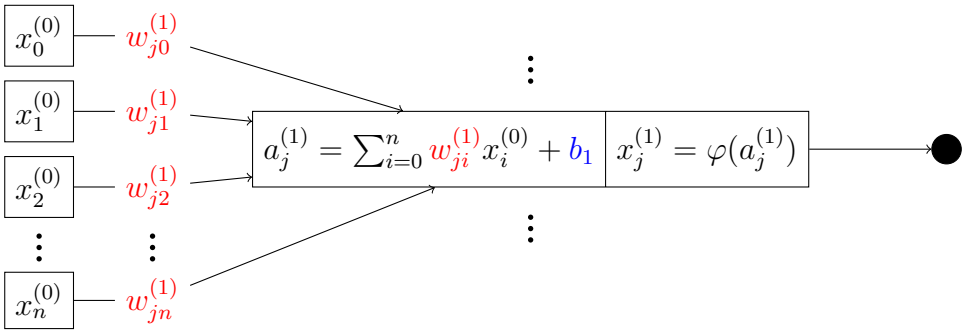
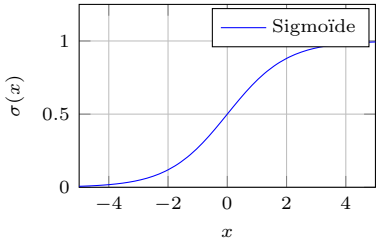
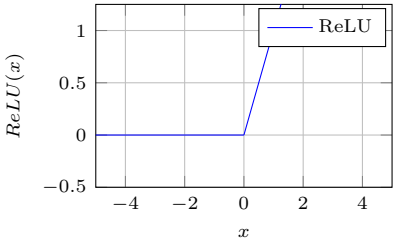


FIGURE 2 – Calcul de la valeur du $j^{\text{ème}}$ neurone de la $1^{\text{ère}}$ couche cachée.

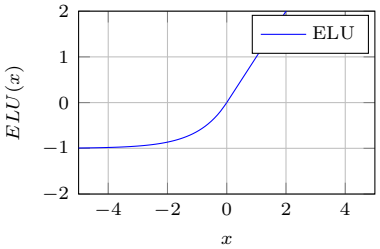
Dans la figure 2, on constate que chaque neurone de de la première couche cachée est une combinaison linéaire des neurones d'entrée (avec comme cœfficients les poids des liaisons), à laquelle on a ajouté un terme de **biais** b_1 et appliqué une **fonction d'activation** φ pour introduire la non-linéarité.



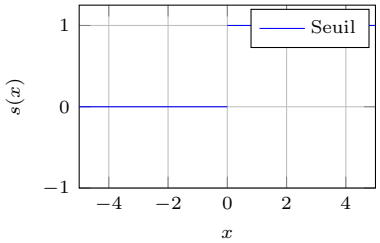
(a) La fonction sigmoïde $\sigma(x) = \frac{1}{1+e^{-x}}$



(b) ReLU (Rectified Linear Unit) $ReLU\,(x) = \max(0, x)$



(c) ELU (Exponential Linear Unit) $ELU(x) = \begin{cases} \alpha(e^x - 1) & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$



(d) La fonction seuil $s(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$

FIGURE 3 – Exemples de fonctions d'activation

Les valeurs de la couche suivante sont déterminées récursivement de la même façon en considérant la première couche cachée comme la nouvelle couche d'entrée.

On peut représenter la $k^{\text{ème}}$ couche d'un réseau par un vecteur $X^{(k)}$ calculé (sauf pour la couche d'entrée) de la manière suivante :

$$X^{(k)} = \varphi\left(W^{(k)}\, X^{(k-1)} + B_k\right)$$

Avec :

— $W^{(k)} = \begin{pmatrix} w_{1,1}^{(k)} & w_{1,2}^{(k)} & \cdots & w_{1,n}^{(k)} \\ w_{2,1}^{(k)} & w_{2,2}^{(k)} & \cdots & w_{2,n}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{l,1}^{(k)} & w_{l,2}^{(k)} & \cdots & w_{l,n}^{(k)} \end{pmatrix}$: la matrice des poids des connexions entre la couche $X^{(k-1)}$ et la couche $X^{(k)}$

— $X^{(k-1)} = \begin{pmatrix} x_1^{(k-1)} \\ x_2^{(k-1)} \\ \vdots \\ x_n^{(k-1)} \end{pmatrix}$: le vecteur des neurones de la $(k-1)^{\text{ème}}$ couche

— $B_k = \begin{pmatrix} b_k \\ b_k \\ \vdots \\ b_k \end{pmatrix}$: le vecteur des biais de la couche $X^{(k)}$

— φ : une fonction d'activation

n étant le nombre de neurones de la couche $X^{(k-1)}$, et l le nombre de neurones de la couche $X^{(k)}$

Le calcul initial des prédictions donne un résultat arbitraire. Nous avons donc besoin de modifier les paramètres du réseau (poids et biais) de manière à obtenir les sorties souhaitées.

1.3 Phase de la modification des paramètres : la rétropropagation *(Back-propagation)*

L'optimisation d'un réseau de neurones artificiels revient à lui apprendre la relation correcte qui relie les entrées aux sorties. Très souvent, cela se fait en se servant d'un ensemble d'exemples annotés dit l'**ensemble d'apprentissage** (apprentissage supervisé). Chaque exemple de cet ensemble est un couple constitué d'une entrée et de la sortie cible (celle que l'on souhaite avoir).
Dans la phase de la rétropropagation, l'erreur (l'écart calculé entre la sortie réelle et la sortie cible de chaque entrée) est propagée en arrière (de la sortie vers l'entrée) pour estimer le taux par lequel on doit corriger les paramètres.

1.3.1 La fonction de perte

Définition

■ Une **fonction de perte** (*Loss function*) est une fonction dont l'utilité est d'évaluer la pertinence d'un modèle. Elle donne une idée sur l'écart entre la sortie réelle et la sortie désirée.

On considère un perceptron multicouche avec une entrée de taille n, une sortie de taille p, et de profondeur (nombre total de couches) d ($n, p, d \in \mathbb{N}^*$ avec $d \geq 3$).

On note $W = (W^{(k)})_{2 \leq k \leq d}$ et $B = (B_k)_{2 \leq k \leq d}$ respectivement les poids et les biais du réseau.

Soit $\mathbb{E}$ l'ensemble d'apprentissage. On a :

$$\mathbb{E} = \{(X, \hat{Y}_X) \in E \times S \quad \text{tel que, si l'entrée est X, on veut que la sortie soit } \hat{Y}_X\}$$

Avec :
$E \subset \mathbb{R}^n$: une partie finie de $\mathbb{R}^n$ contenant les entrées de l'ensemble d'apprentissage ;
$S \subset \mathbb{R}^p$: une partie finie de $\mathbb{R}^p$ contenant les sorties désirées de l'ensemble d'apprentissage.

Pour tout $X \in E$, soit Y_X la sortie réelle (véritable) donnée par le réseau (c'est-à-dire $Y_X = X^{(d)}$). On choisit comme fonction de perte l'**erreur quadratique moyenne MSE** (*Mean Squared Error*) définie par :

$$J(Y_X) = \frac{1}{2N} \sum_{\hat{Y}_X \in S} \left\| \hat{Y}_X - Y_X \right\|^2$$

($N = Card(\mathbb{E})$)

On constate que la fonction de perte dépend de la sortie qui, à son tour, dépend des paramètres du réseau (poids et biais).

Autrement, si l'on assimile l'ensemble du réseau à une fonction f qui relie les entrées aux sorties, et si on note f^* la fonction qui devrait les lier, on aura $f(X, W, B) = Y_X$ et $f^*(X) = \hat{Y}_X$

La fonction de perte devient :

$$J(W, B) = \frac{1}{2N} \sum_{X \in E} \|f^*(X) - f(X, W, B)\|^2$$

Objectif

Pour entraîner le réseau à faire de bonnes prédictions, on doit faire tendre J vers 0. Et ce en ajustant W et B de manière à avoir $f \approx f^*$.

1.3.2 L'algorithme de la rétropropagation

Définition

- Le ***Gradient Descent*** est un algorithme destiné à minimiser la fonction de perte en calculant la contribution de chaque paramètre à cette dernière puis en la retranchant (voir la suite).
- La **rétropropagation du gradient** est une méthode pour calculer le gradient de l'erreur, de la dernière couche vers la première.

On rappelle que pour tout $k \in \{2, \cdots, d\}$:

$$X^{(k)} = \varphi\left(W^{(k)} X^{(k-1)} + B_k\right)$$

Posons $a^{(k)} = W^{(k)} X^{(k-1)} + B_k$ (C'est-à-dire $a_j^{(k)} = \sum_i w_{ji}^{(k)} x_i^{(k-1)} + b_k$) donc $X^{(k)} = \varphi\left(a^{(k)}\right)$

J est la moyenne de l'erreur sur tout l'ensemble d'apprentissage. Pour un exemple individu d'entraînement $(X, \hat{Y}_X) \in \mathbb{E}$, on pose $J_X = \frac{1}{2} \left\| \hat{Y}_X - Y_X \right\|^2$ l'erreur sur un exemple. Du coup on a $J = \frac{1}{N} \sum_{X \in E} J_X$.

■ L'erreur à la sortie :

Pour $k \in \{2, \cdots, d\}$, on introduit $\delta_j^{(k)}$, l'erreur sur le $j^{\text{ème}}$ neurone de la $k^{\text{ème}}$ couche, définie par :

$$\delta_j^{(k)} = \frac{\partial J_X}{\partial a_j^{(k)}} \tag{1}$$

Ainsi, l'erreur sur les neurones de la sortie est

$$\delta_j^{(d)} = \frac{\partial J_X}{\partial a_j^{(d)}}. \tag{2}$$

D'après la **règle de la chaîne**, on a :

$$\delta_j^{(d)} = \sum_{i=1}^{p} \frac{\partial J_X}{\partial x_i^{(d)}} \frac{\partial x_i^{(d)}}{\partial a_j^{(d)}} \tag{3}$$

Or $x_i^{(d)}$ ne dépend que de $a_j^{(d)}$ lorsque $i = j$, alors

$$\delta_j^{(d)} = \frac{\partial J_X}{\partial x_j^{(d)}} \frac{\partial x_j^{(d)}}{\partial a_j^{(d)}} \tag{4}$$

On rappelle que $x_j^{(d)} = \varphi\left(a_j^{(d)}\right)$. Ainsi :

$$\delta_j^{(d)} = \frac{\partial J_X}{\partial x_j^{(d)}} \varphi'\left(a_j^{(d)}\right) \tag{5}$$

On introduit le produit matriciel de ***Hadamard*** $\odot$ tel que, pour $A, B \in \mathcal{M}_{n,p}(\mathbb{R})$, pour $(i,j) \in \{1, \cdots, n\} \times \{1, \cdots, p\}$ $\quad [A \odot B]_{i,j} = [A]_{i,j} \times [B]_{i,j}$.
On peut donc écrire la relation (5) vectoriellement (pour toute la couche de sortie) de la façon suivante :

$$\delta^{(d)} = \nabla_d J_X \odot \varphi'\left(a^{(d)}\right) \tag{6}$$

Avec $\quad \nabla_d J_X = \begin{pmatrix} \frac{\partial J_X}{\partial x_1^{(d)}} \\ \frac{\partial J_X}{\partial x_2^{(d)}} \\ \frac{\partial J_X}{\partial x_3^{(d)}} \\ \vdots \end{pmatrix}$

■ Calcul de $\delta^{(k-1)}$ en fonction de $\delta^{(k)}$:

Toujours avec la règle de la chaîne,

$$\begin{aligned} \delta_j^{(k-1)} &= \frac{\partial J_X}{\partial a_j^{(k-1)}} \\ &= \sum_i \frac{\partial J_X}{\partial a_i^{(k)}} \frac{\partial a_i^{(k)}}{\partial a_j^{(k-1)}} \qquad (7) \\ &= \sum_i \frac{\partial a_i^{(k)}}{\partial a_j^{(k-1)}} \delta_i^{(k)} \qquad (8) \end{aligned}$$

On a $a_i^{(k)} = \sum_j w_{ij}^{(k)} x_j^{(k-1)} + b_k = \sum_j w_{ij}^{(k)} \varphi\left(a_j^{(k-1)}\right) + b_k$

Si l'on dérive par rapport à $a_j^{(k-1)}$, on obtient :

$$\frac{\partial a_i^{(k)}}{\partial a_j^{(k-1)}} = w_{ij}^{(k)} \varphi'\left(a_j^{(k-1)}\right) \tag{9}$$

En remplaçant cette quantité dans (8), on aura :

$$\delta_j^{(k-1)} = \sum_i w_{ij}^{(k)} \delta_i^{(k)} \varphi'\left(a_j^{(k-1)}\right) \tag{10}$$

Pour l'ensemble de la $(k-1)^{\text{ème}}$ couche

$$\delta^{(k-1)} = \left({}^t\left(w^{(k)}\right) \delta^{(k)} \right) \odot \varphi'\left(a^{(k-1)}\right) \tag{11}$$

■ **Taux de variation de la fonction de perte par rapport aux poids $\frac{\partial J_X}{\partial w_{ji}^{(k)}}$ et par rapport aux biais $\frac{\partial J_X}{\partial b_k}$:**

Toujours en utilisant la règle de la chaîne et en procédant de la même manière que dans les équations précédentes, on aboutit aux relations suivantes :

$$\frac{\partial J_X}{\partial b_k} = \delta_j^{(k)} \tag{12}$$

$$\frac{\partial J_X}{\partial w_{ji}^{(k)}} = x_i^{(k-1)} \delta_j^{(k)} \tag{13}$$

[4]

■ **Récapitulation : algorithme de la rétropropagation**

1. Fournir un vecteur $X^{(1)}$ en entrée ;
2. Pour chaque $k = 2, 3, \cdots, d$, calculer $a^{(k)} = W^{(k)} X^{(k-1)} + B_k$ et $X^{(k)} = \varphi\left(a^{(k)}\right)$;
3. Calculer le gradient de l'erreur sur la sortie : $\delta^{(d)} = \nabla_d J_X \odot \varphi'\left(a^{(d)}\right)$;
4. Pour chaque $k = d-1, d-2, \cdots, 2$, propager l'erreur de la sortie vers l'entrée en calculant $\delta^{(k)} = \left({}^t\left(w^{(k+1)}\right) \delta^{(k+1)} \right) \odot \varphi'\left(a^{(k)}\right)$;
5. Calculer les gradients sur les paramètres : $\frac{\partial J_X}{\partial b_k} = \delta_j^{(k)}$ et $\frac{\partial J_X}{\partial w_{ji}^{(k)}} = x_i^{(k-1)} \delta_j^{(k)}$

■ **La mise à jour des paramètres (l'apprentissage) avec le *Gradient Descent***

Après avoir calculé le gradient, on l'exploite pour la modification des paramètres :

$$b_k \leftarrow b_k - \eta \frac{\partial J_X}{\partial b_k}$$

$$w_{ji}^{(k)} \leftarrow w_{ji}^{(k)} - \eta \frac{\partial J_X}{\partial w_{ji}^{(k)}}$$

Avec $\eta \in [0, 1]$ le **taux d'apprentissage**. Il représente la proportion par laquelle on modifie les paramètres à chaque itération. [3]

1.4 Théorème d'approximation universelle

Soient :

- $m \in \mathbb{N}^*$
- $X = (x_1, \cdots, x_m) \subseteq \mathbb{R}^m$: un compact de $\mathbb{R}^m$
- $\mathcal{C}(X)$: l'ensemble des fonctions continues sur X

Théorème

Soit φ une fonction d'activation.
$\forall f \in \mathcal{C}(X), \exists n \in \mathbb{N}, \quad a_{ij}, b_i, w_i \in \mathbb{R}, \quad i \in \{1, \cdots, n\}, \quad j \in \{1, \cdots, m\}$ tels que :

$$(A_n f)(X) := \sum_{i=1}^{n} w_i \, \varphi \left(\sum_{j=1}^{m} a_{ij} x_j + b_i \right)$$

et $\forall \varepsilon > 0$:

$$\| f - A_n f \| < \varepsilon$$

Le **théorème d'approximation unviverselle** affirme que l'on peut approcher une fonction continue sur un compact de $\mathbb{R}^m$ avec un perceptron multicouche standard à une couche cachée avec un nombre fini de neurones et une fonction d'activation arbitraire. [2]

Dans notre étude, on s'intéressera au traitement des images qui sont souvent représentées par des matrices ou même des tenseurs dont les cœfficients sont les pixels. Or, le perceptron multicouche standard ne prend en entrée que des vecteurs. Et si l'on tente de vectoriser une image pour la faire passer dans un MLP, elle risque de perdre ses caractéristiques spatiales. De plus, vu que les neurones des couches adjacentes sont entièrement connectés, on aura des matrices de poids de tailles énormes.

2 Réseau de neurones convolutif

Lorsqu'il s'agit du traitement d'images complexes et de grandes dimensions, le perceptron multicouche standard semble être un mauvais choix.

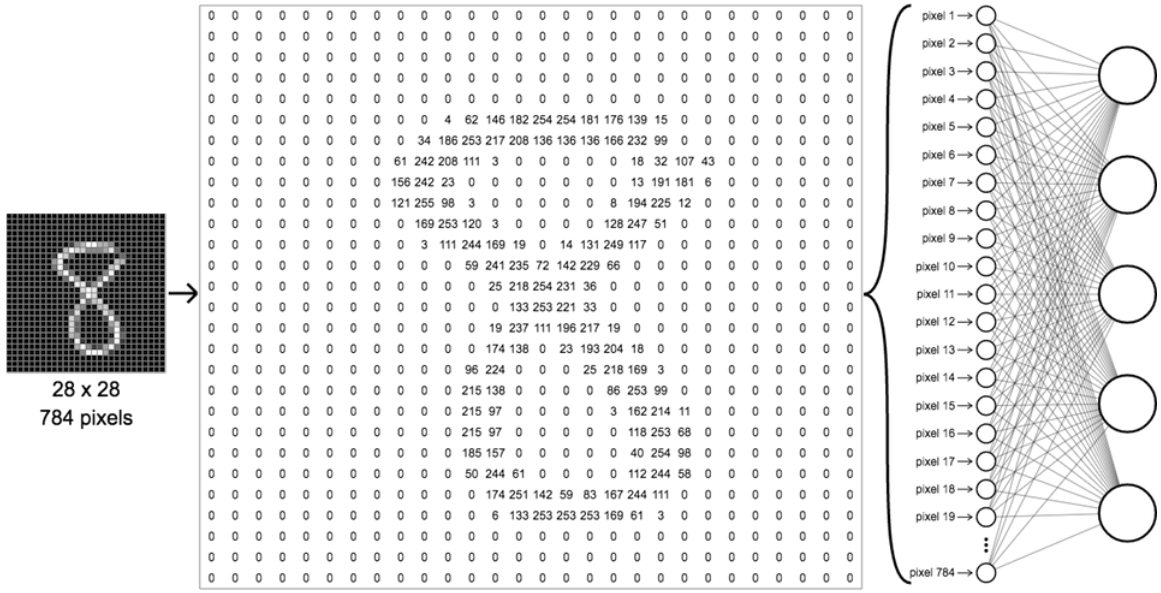


FIGURE 4 – Illustration des inconvénients du MLP traditionnel.

Les **réseaux de neurnes convolutifs** (**ConvNets** : ***Con**volutional Neural **Net**works*) sont une variante de réseaux de neurones artificiels qui sont essentiellement dédiés à faire des prédictions à partir d'images complexes, vu qu'ils les soumettent à un pré-traitement pour en extraire les caractéristiques avant d'arriver à la phase de la prise de décision.

À la différence du perceptron multicouche traditionnel, un ConvNet est constitué de couches de différents type et qui ne sont pas toutes entièrement connectées entre elles, mais sur de petites portions "mobiles" appelées **filtres**.

2.1 Préliminaires

Définition

Soient $m, n, p \in \mathbb{N}^*$.
On pose $I = \{1, \cdots, m\}, \quad J = \{1, \cdots, n\}, \quad K = \{1, \cdots, p\}$.
Soit $(t_{i,j,k})_{i \in I, j \in J, k \in K}$ une famille de $\mathbb{R}$.
On appelle un **tenseur** à cœfficients réels de taille $m \times n \times p$ l'application :

$$\begin{aligned} T : I \times J \times K &\to \mathbb{R} \\ (i, j, k) &\mapsto t_{i,j,k} \end{aligned}$$

On note $\mathbb{R}^{m \times n \times p}$ l'ensemble de tels tenseurs au lieu de $\mathbb{R}^{I \times J \times K}$

Un tenseur est une notion plus générale qu'une matrice.

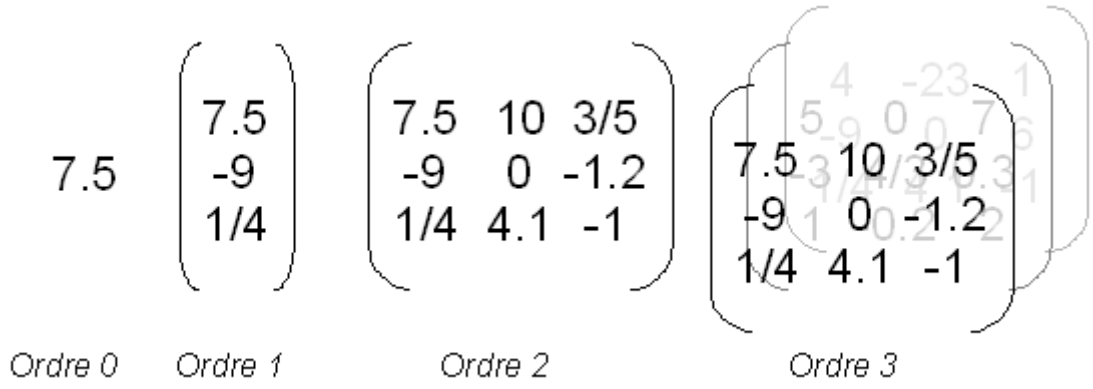


FIGURE 5 – Formes du tenseur

On peut modéliser une image en format **RGB** (*Red-Green-Blue*) par un tenseur de $\mathbb{R}^{n\times p\times 3}$, c'est-à-dire un tenseur constitué de 3 matrices, chacune de taille $n \times p$ et représente un canal **R** ou **G** ou **B**. En effet, chaque pixel d'une image est une combinaison de ces trois couleurs.

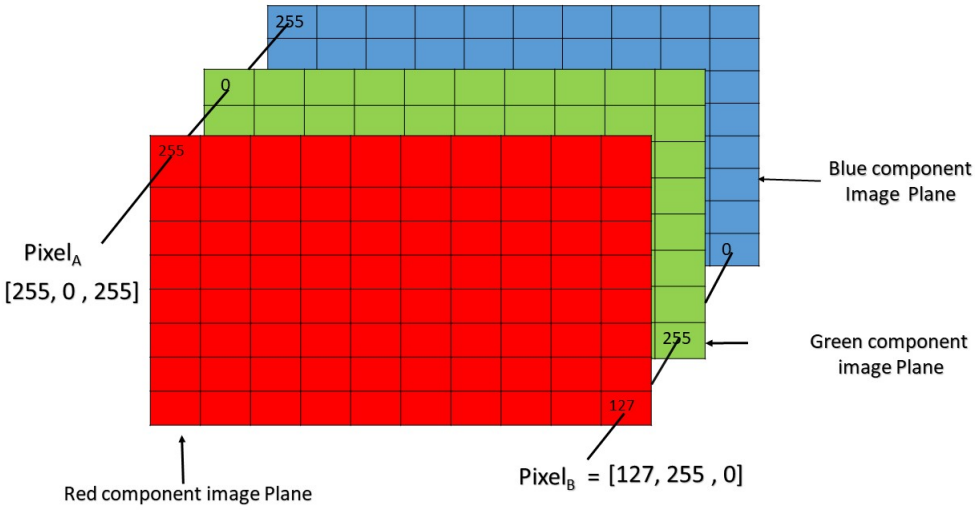


FIGURE 6 – Représentation d'une image RGB

2.2 Architecture du réseau

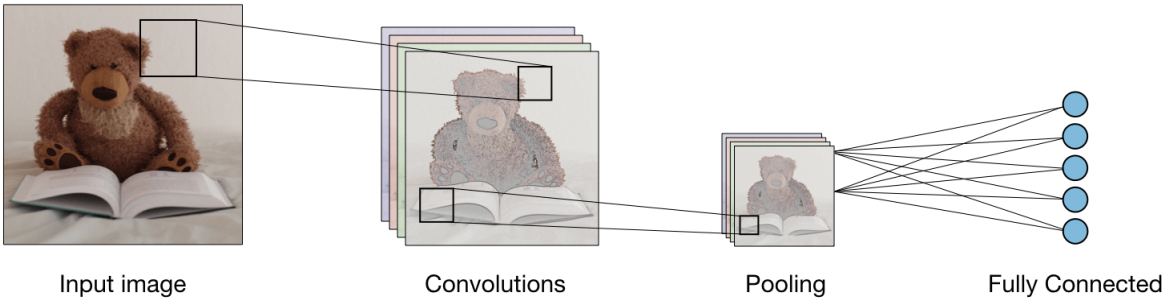


FIGURE 7 – Architecture globale d'un ConvNet

Les ConvNets sont semblables aux réseaux de neurones traditionnel en ce qu'ils sont composés de neurones qui ont des poids et des biais modifiables. Chaque neurone reçoit une entrée, effectue un produit scalaire et applique une fonction d'activation. Ils ont aussi une fonction de perte à partir de laquelle on calcule le gradient pour ajuster les paramètres. Du coup, la plupart des points abordés dans le perceptron multicouche sont encore valables pour un ConvNet. Nous ne nous intéresserons donc qu'aux traits qui font la particularité de ce dernier.

2.2.1 La couche de convolution

- La **couche de convolution** se situe souvent au début du réseau.
- Elle utilise un filtre, qui est une matrice (ou un tenseur) de taille petite dont les cœfficients sont les poids, pour réaliser l'opération de convolution.

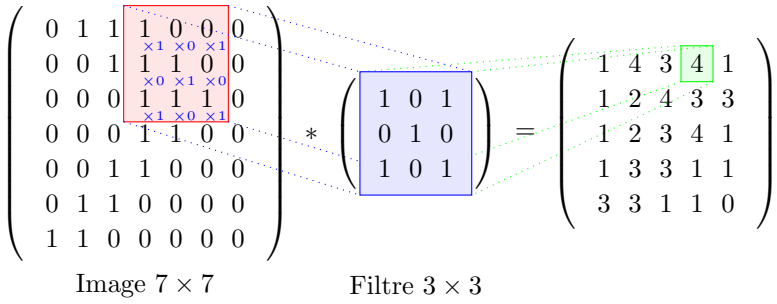


FIGURE 8 – Exemple de convolution sur une image 2D de taille 7×7 avec un filtre de taille 3×3

Le filtre, étant de taille plus petite que celle de l'image, se déplace sur cette dernière en occupant des zones appelées des **champs récepteurs**. Sur chaque champ récepteur, on effectue la somme des cœfficients de l'image pondérée par ceux du filtre.

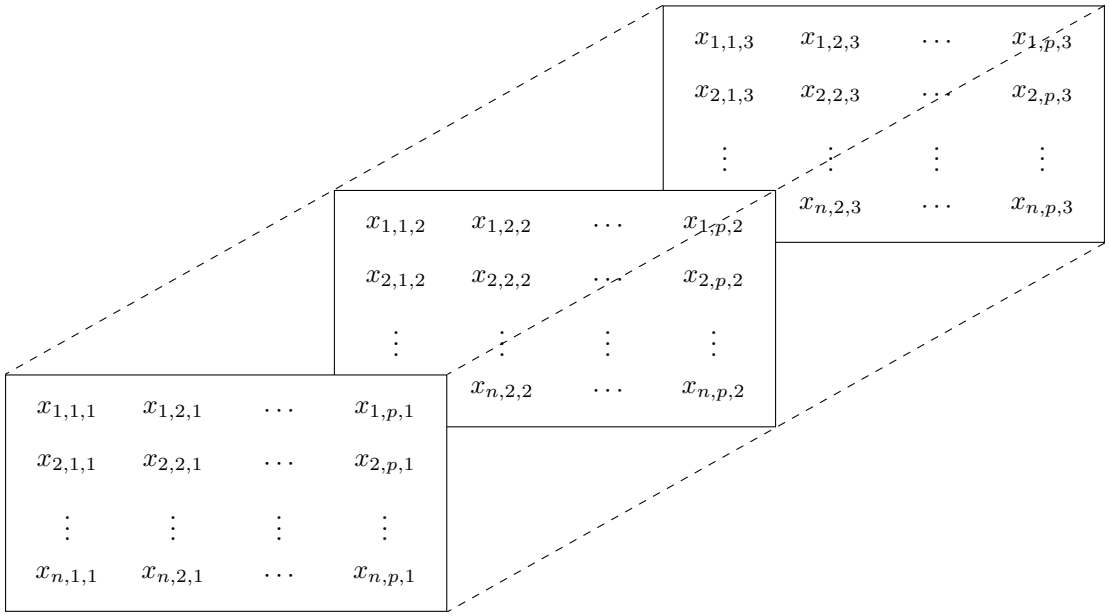
À l'issue de cette opération, on obtient une nouvelle matrice appelée *feature map* ou *activation map*.

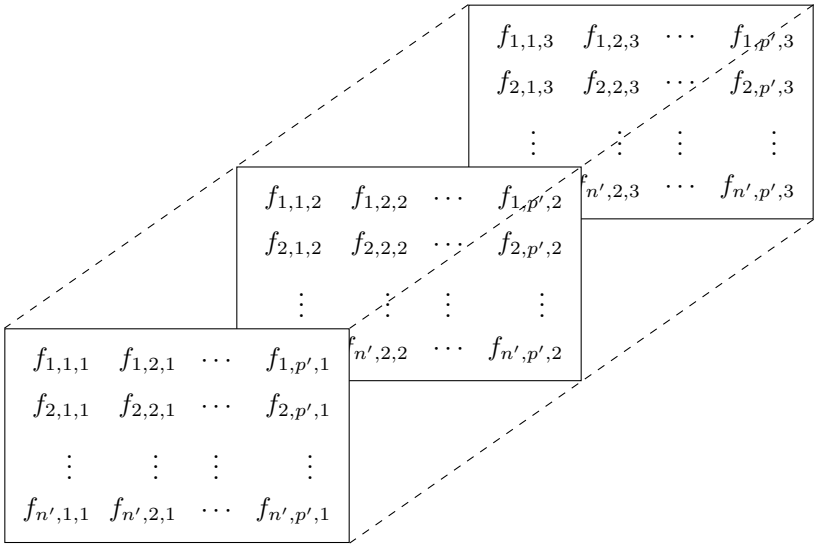
Chaque filtre est entraîné pour la détection de caractéristiques spécifiques. Ainsi, les valeurs du *feature map* sont d'autant plus grandes que ces caractéristiques sont présentes dans l'image d'entrée.

Un terme de biais est souvent ajouté au *feature map*. Mais pour des raisons de simplicité on n'en tient pas compte.

■ Une formulation mathématique d'une convolution à 3D :

Soient $n, p, n', p' \in \mathbb{N}^*$ avec $n' < n$ et $p' < p$.
On considère l'image de taille $n \times p \times 3$ et le filtre de taille $n' \times p' \times 3$ suivants :


$x_{1,1,3}$ $x_{1,2,3}$ $\cdots$ $x_{1,p,3}$
$x_{2,1,3}$ $x_{2,2,3}$ $\cdots$ $x_{2,p,3}$
$x_{n,2,3}$ $\cdots$ $x_{n,p,3}$
$x_{1,1,2}$ $x_{1,2,2}$ $\cdots$ $x_{1,p,2}$
$x_{2,1,2}$ $x_{2,2,2}$ $\cdots$ $x_{2,p,2}$
$x_{n,2,2}$ $\cdots$ $x_{n,p,2}$
$x_{1,1,1}$ $x_{1,2,1}$ $\cdots$ $x_{1,p,1}$
$x_{2,1,1}$ $x_{2,2,1}$ $\cdots$ $x_{2,p,1}$
$x_{n,1,1}$ $x_{n,2,1}$ $\cdots$ $x_{n,p,1}$


$f_{1,1,3}$ $f_{1,2,3}$ $\cdots$ $f_{1,p',3}$
$f_{2,1,3}$ $f_{2,2,3}$ $\cdots$ $f_{2,p',3}$
$f_{n',2,3}$ $\cdots$ $f_{n',p',3}$
$f_{1,1,2}$ $f_{1,2,2}$ $\cdots$ $f_{1,p',2}$
$f_{2,1,2}$ $f_{2,2,2}$ $\cdots$ $f_{2,p',2}$
$f_{n',2,2}$ $\cdots$ $f_{n',p',2}$
$f_{1,1,1}$ $f_{1,2,1}$ $\cdots$ $f_{1,p',1}$
$f_{2,1,1}$ $f_{2,2,1}$ $\cdots$ $f_{2,p',1}$
$f_{n',1,1}$ $f_{n',2,1}$ $\cdots$ $f_{n',p',1}$

À l'issue de l'opération de convolution, on aura une matrice 3D de taille $(n-n'+1)\times(p-p'+1)\times 3$ avec comme éléments :

$$y_{i,j,k} = \sum_{d=1}^{n}\sum_{h=1}^{p}\sum_{l=1}^{3} x_{d,h,l}\, f_{i-d,j-h,k-l}$$

[6]

2.2.2 La couche de Pooling

- La couche de ***Pooling*** effectue une opération de sous-échantillonnage de l'image après une convolution.
- On y trouve encore un filtre ou une fenêtre qui parcourt l'image, sauf que ce filtre n'a pas de paramètres.
- Il y a deux types de pooling :
  - Max-pooling qui sélectionne la valeur maximale sur le champ récepteur ;
  - Average-pooling qui calcule la valeur moyenne du champ récepteur.

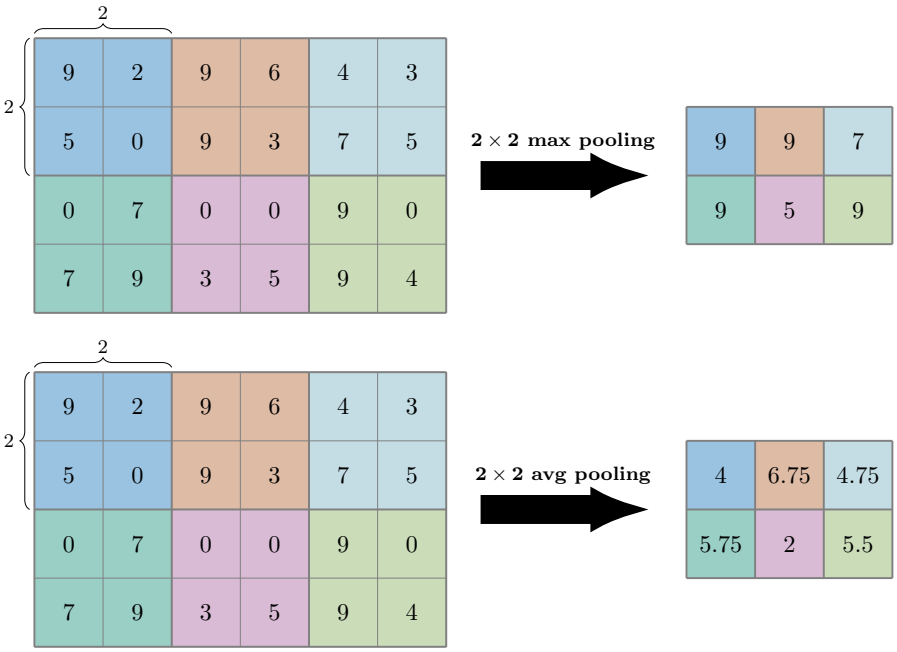


FIGURE 9 – Exemple de max-pooling et d'average-pooling

2.2.3 La couche *Fully Connected*

La couche ***fully connected*** (entièrement connectée) se situe à la fin du réseau. Elle est connectée à l'intégralité de l'image après avoir aplati (transformé de matrice en vecteur) cette dernière. Il s'agit d'un simple perceptron qui effectuera la prédiction après avoir fait passer l'image par une succession de transformations (convolution, pooling, ...)
Tout ce qui a été vu concernant le MLP reste valable pour la couche fully connected.

Lorsqu'il s'agit du traitement d'images, les ConvNets s'avèrent être un moyen puissant d'apprentissage supervisé, vu que leur structure adaptée à l'image facilite l'extraction de ses caractéristiques en conservant son aspect général, ce qui favorise la prise des bonnes décisions à la sortie.

3 Un modèle de réseau de neurones convolutif pour la conduite automatique d'un véhicule autonome

Objectif

On envisage l'entraînement (l'apprentissage) d'un véhicule autonome à conduire lui-même sur une voie vide en n'utilisant que des caméras comme moyens de "voir" sa route.

■ **Idée :**

— Adopter un modèle de réseau de neurone convolutif qui prenne en entrée les images reçues de la route et les transforme en une décision (**angle de braquage**, **vitesse**, ...)

— Optimiser ce modèle pour qu'il puisse prendre les bonnes décisions, et ce en lui fournissant une base de données servant d'exemples d'apprentissage.

3.1 Modèle adopté

Construire un modèle de ConvNet revient à choisir les **couches** à utiliser, leur **nombre**, leur **agencement**, ainsi que leurs **hyper-paramètres**[1]. Puisqu'il n'y a pas de nombres magiques ou de modèles universels, cette tâche est réservée aux experts de ce domaine.

Dans ma manipulation, j'ai adopté le modèle de ConvNet conçu par NVIDIA pour son projet de voiture autonome [1] (voir figure 10), et je l'ai entraîné en utilisant des données que j'ai générées avec le simulateur de voiture autonome de *Udacity* (***Udacity Self-driving Car Simulator***).

J'aurais pu implémenter ce modèle en Python en définissant manuellement chaque couche. Mais Python dispose déjà de plusieurs outils d'apprentissage automatique (Tensorflow, Keras, Scikit-learn,...) qui permettent la construction des réseaux de neurones avec des fonctions prédéfinies, ce qui améliore la performance et la rapidité des programmes. De ce fait, je me suis contenté de télécharger le code de ce modèle à partir de Github pour m'en servir dans ma manipulation. Cependant, une implémentation manuelle des différentes couches du ConvNet en Python sera donnée en annexe.

1. Les hyper-paramètres du réseau sont les paramètres que l'on doit choisir avant de commencer l'apprentissage. Ils restent fixes tout au long du processus de l'entraînement et ne sont pas modifiables par celui-ci, à la différence des poids et des biais. Exemples : Taux d'apprentissage, taille des filtres, le *stride* (nombre de pixels par lesquels les filtres se déplacent après chaque opération),...

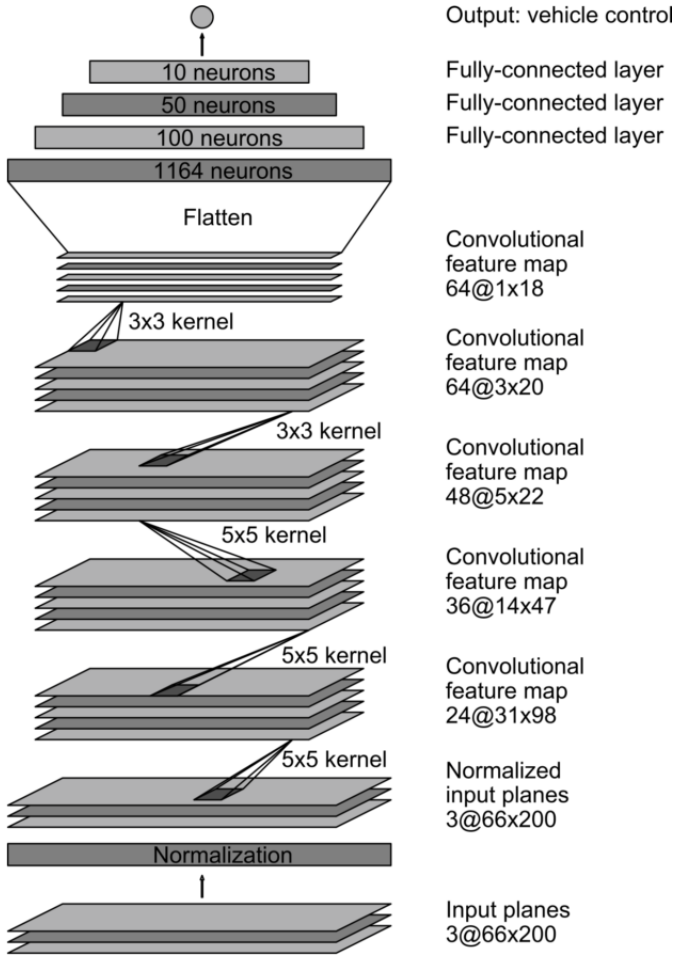


FIGURE 10 – Architecture du ConvNet conçu par NVIDIA. Il compte environ 250 000 paramètres à entraîner.

3.2 Explication de la démarche

a) **Télécharger et installer le simulateur de voiture autonome de *Udacity*** : C'est un simulateur qui a été mis en disposition du grand public par *Udacity*, et que l'on peut cloner à partir de son dépôt sur Github en utilisant la commande suivante :

```
git clone https://github.com/udacity/self-driving-car-sim.git
```

Le simulateur dispose de deux modes : le mode d'entraînement (***TRAINING MODE***) dans laquel on conduit le véhicule manuellement pour générer les données d'apprentissage, et le mode autonome (***AUTONOMOUS MODE***) dans lequel on teste la performance du véhicule après l'avoir entraîné. Il dispose aussi de deux scènes différentes.

FIGURE 11 – L'écran d'accueil du simulateur de voiture autonome de *Udacity*

b) **Télécharger le modèle de ConvNet :**
L'implémentation du modèle de *NVIDIA* se trouve dans le projet Github que l'on peut cloner avec la commande suivante :

```
git clone https://github.com/naokishibuya/car-behavioral-cloning
```

Ce projet contient aussi les environnements à installer et les fichiers qui assurent l'interaction du simulateur avec Python.

c) **Créer un environnenment de travail :**

```
conda env create -f environments.yml
```

Cette commande va installer toutes les bibliothèques nécessaires pour le travail.

d) **Générer les données d'entraînement avec le *TRAINING MODE* du simulateur :**

FIGURE 12 – Conduite manuelle du véhicule

C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0	0.09792101	0	0	22.02592
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	-0.1819504		0	0	21.66091
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0		0	0	21.45187
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0		0	0	21.23293
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0	0.1636876	0	0	21.02131
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	-0.3721615		0	0	20.59172
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	-0.03718242		0	0	20.52837
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0		0	0	20.30733
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0		0	0	19.77275
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0	0.3352679	0	0	20.0201
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0	0.01183733	0	0	20.27391
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0		0	0	19.64971
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0	0.3607632	0	0	19.7622
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	-0.4418601	0.8026233	0	0	20.70411
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	-0.1189289	0.4796921	0	0	21.14188
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0	0.1416959	0	0	21.32686
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0		0	0	20.85255
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0		0	0	20.59092
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0.3417885		0	0	20.57492
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0.6553874		0	0	19.8385
C:\Users\lenovo\Desktop\Kouhou\IMG\center_20	C:\Users\lenovo\Desktop\Kouhou\IMG\left_2	C:\Users\lenovo\Desktop\Kouhou\IMG\right	0.2121361		0	0	19.49987

FIGURE 13 – Extrait du fichier *driving_log.csv* dans lequel le simulateur enregistre les données générées

Dans le mode d'entraînement, on appuie sur *Record* et on commence à conduire le véhicule avec le clavier ou avec une manette de jeu.

Alors que le véhicule avance, le simulateur enregistre les images de la route capturées par trois caméras attachées au véhicule (caméra centrale, caméra de droite et caméra gauche), ainsi que l'angle de braquage, la vitesse et l'accélération du véhicule au même instant. Chaque exemple d'entraînement est, donc, de la forme :

Entrée			Sortie souhaitée		
Image centre	Image gauche	Image droite	Braquage	Accélération	Vitesse

À la fin de l'entraînement, le modèle doit apprendre la relation qui relie les entrées aux sorties souhaitées pour la généraliser pour d'autres situations.

e) **Installer l'environnement de développement de Python, créer un environnement virtuel et l'activer :**
Cet environnement virtuel va recevoir Tensorflow qui va servir de *backend* pour Keras. Pour ce faire, exécuter successivement les commandes suivantes :

```
pip3 install -U pip virtualenv

virtualenv --system-site-packages -p python3 ./venv

.\venv \Scripts\activate
```

f) **Phase de l'entraînement :**
Après avoir préparé l'environnement de travail, on commence l'entraînement du réseau en exécutant la commande suivante :

```
python model.py
```

- Le fichier `model.py` contient le modèle de ConvNet adopté. Lorsqu'on l'exécute, il importe le fichier `driving_log.csv` contenant les données d'apprentissage préalablement générées, puis l'entraînement commence.
- Lors de l'apprentissage, le réseau peut itérer sur tout l'ensemble des données plusieurs fois. Chaque cycle est appelé ***epoch***. À la fin de chaque *epoch*, si la performance est meilleure que celle de l'*epoch* précédent, un fichier `model<epoch>.h5` est généré.
- Dans ma manipulation, cela a pris environ 20 heures pour que le modèle effectue 3 *epoch* sur 10.

```
1
2 WARNING:tensorflow:From C:\Users\lenovo\Desktop\Kouhou\How_to_simulate_a_self_driving_car-master\
      venv\lib\site-packages\tensorflow\python\ops\math_ops.py:3066: to_int32 (from tensorflow.python.
      ops.math_ops) is deprecated and will be removed in a future version.
3 Instructions for updating:
4 Use tf.cast instead.
5 Epoch 1/10
6 2019-04-01 21:32:28.388798: I tensorflow/core/platform/cpu_feature_guard.cc:141]
7 Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
8 20000/20000 [==============================] - 7714s 386ms/step - loss: 0.0302 - val_loss: 0.0202
9 Epoch 2/10
10 20000/20000 [==============================] - 7747s 387ms/step - loss: 0.0240 - val_loss: 0.0201
11 Epoch 3/10
12 20000/20000 [==============================] - 49501s 2s/step - loss: 0.0219 - val_loss: 0.0178
13 Epoch 4/10
14   515/20000 [==============================] - ETA: 2:15:17 - loss: 0.0214
```

FIGURE 14 – Processus d'entraînement

3.3 Résultats

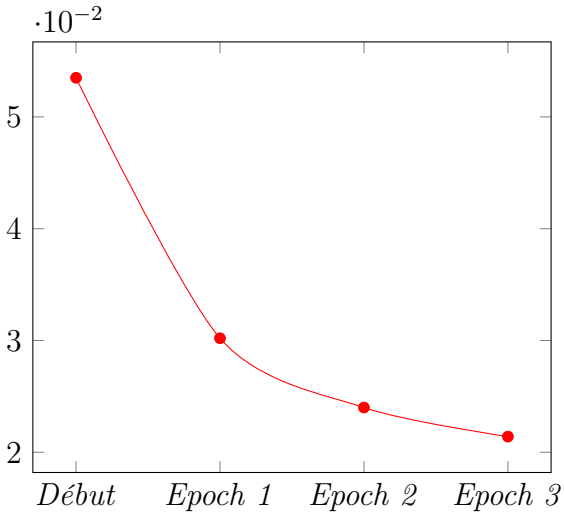


FIGURE 15 – Variation de la fonction de perte après chaque *epoch*

Après avoir entraîné le modèle jusqu'à un nombre d'*epoch* jugé suffisant, on met le simulateur en mode autonome pour tester le modèle entraîné.
On exécute la commande suivante :

```
python drive.py model-003.h5
```

Maintenant que l'on a fourni au véhicule le modèle optimisé, il devrait "savoir" quel angle de braquage, quelle vitesse et quelle accélération associer à chaque image qu'il "voit".

■ Résultats observés :

★ Le véhicule roule **automatiquement** sur la route dans laquelle on l'a entraîné.
★ Mais lorsqu'on change de scène, il ne parvient plus à suivre la voie.
★ Cela peut être dû au problème du **sur-apprentissage**.

Références

[1] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv :1604.07316*, 2016.

[2] B. C. Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, page 11, 2001.

[3] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[4] M. A. Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA :, 2015.

[5] Wikipedia. Réseau de neurones artificiels — Wikipedia, the free encyclopedia. `http://fr.wikipedia.org/w/index.php?title=R\%C3\%A9seau\%20de\%20neurones\%20artificiels&oldid=160285134`, 2019.

[6] J. Wu. Introduction to convolutional neural networks. *National Key Lab for Novel Software Technology. Nanjing University. China*, 2017.

Annexe : codes Python

■ Perceptron multicouche

1. Initialisation du réseau

```
# Initialisation du reseau
import numpy as np
from random import random

def init_reseau(*args):
    ''' la fonction prend en argument la structure du reseau.
    Exemple : init_reseau(2,4,3,1) initialise un reseau a 4
    couches, une couche d'entree a 2 neurones, une couche
    cachee a 4 neurones, une autre couche cachee a 3 neurone,
    et une couche de sortie a 1 neurone. Le nombre d'arguments
    correspond au nombre de couches  '''
     couche=[]
     for i in args:
         couche.append(i)
     assert len(couche)>=3, "Le reseau doit avoir au moins 3
    couches"
     poids=[]
     biais=[]
     for i in range(1, len(couche)):
         W=np.array([[random() for j in range(couche[i-1])]for
    k in range(couche[i])])
         poids.append(W)
         B=np.array([[random()] for k in range(couche[i])])
         biais.append(B)
     return poids, biais
```

2. Propagation directe

```
from math import exp
 #si on prend la sigmoide comme fonction d'activation
def sigmoid(a):
  n,p=np.shape(a)
  b=np.zeros([n,p])
  for i in range(n):
    b[i][0]=1/(1+exp(-a[i][0]))
  return b

def feedforward(x, poids, biais):  #x : vecteur des entrees,
    cela doit etre un array dont le nombre d'elements est egal
    au nombre des neurones d'entree.

     for w,b in zip(poids, biais):
         x=sigmoid(np.dot(w,x)+b)
     return x
```

3. Rétropropagation

```python
def backprop(x,yc,poids,biais):
    #x:entree
    #yc:sortie cible
    X, A=feedforward(x,poids,biais)
    y_s=X[-1]          #Couche de sortie
    a_s=A[-1]

    dB=[0]*len(poids)
    dW=[0]*len(poids)
    delta=[0]*len(poids)

    #Insertion de l'erreur de la couche de sortie
    #dJ/dy_s=(yc-y_s)
    #On peut montrer que sigmoid'(x)=sigmoid(x)*(1-sigmoid(x))

    s=a_s.shape
    delta[-1]=(yc-y_s)*sigmoid(a_s)*(np.ones(s)-sigmoid(a_s))

    #Retropropagation de l'erreur

    for i in range(len(poids)-2, -1 ,-1):
        s=A[i].shape
        delta[i]=np.dot(poids[i+1].T, delta[i+1])*sigmoid(A[i
    ])*(np.ones(s)-sigmoid(A[i]))
        #Calcul des gradients sur les parametres
        dB[i]=delta[i]
        dW[i]=np.array([[X[i-1][j]*delta[i][k] for k in range(
    len(delta[i]))] for j in range(len(X[i-1]))])

    return dB, dW
```

4. **Modification des paramètres**

```python
def update(E,poids,biais,eta=0.001,epochs=50):
    #E:ensemble d'apprentissage
    #eta:taux d'apprentissage
    n_poids=poids
    n_biais=biais
    for d in range(epochs):
        for ex in E:
            x,yc=ex
            dB,dW=backprop(x,yc,n_poids,n_biais)
            n_poids=[w-eta*dw for w, dw in zip(n_poids,dW)]
            n_biais=[b-eta*db for b, db in zip(n_biais,dB)]

    return n_poids, n_biais
```

■ ConvNet

1. **Convolution**

```python
import numpy as np

def convolution(img, flt, biais): #img:image , flt:filtre
```

```
    #Dimensions de l'image et du filtre
    mf, nf, pf=flt.shape
    mi, ni, pi=img.shape

    feature_map=np.zeros((mf, ni-nf+1, pi-pf+1))

    #Convolution de chaque filtre sur l'image
    for f in range(mf):
        y_f=map_y=0             #Position initiale du filtre sur l'image et sur le feature map
        #Deplacement du filtre verticalement
        while y_f+pf <= pi :
            x_f=x_map=0
            #Deplacement du filtre horizontalement
            while x_f+nf <= ni :
                #Effectuer l'operation de convolution
                feature_map[f, map_y, x_map]=np.sum(flt[f]*img[:,y_f:y_f+pf,x_f:x_f+nf])+biais[f]
                x_f+=1
                x_map+=1
            y_f+=1
            map_y+=1

    return feature_map
```

2. **Pooling**

```
def maxpool(img, fenetre):
    mi, ni, pi=img.shape
    nf, pf=fenetre.shape

    #Initialisation de la sortie
    sortie=np.zeros((mi, ni-nf+1, pi-pf+1))

    #Deplacement de la fenetre sur chaque canal de l'image
    for c in range(mi):
        y_f=y_sortie=0
        #Deplacement vertical
        while y_f+pf <= pi :
            x_f=x_sortie=0
            #Deplacement horizontal
            while x_f+nf <= ni :
                #Prendre la valeur maximale sur chaque fenetre
                sortie[c, y_sortie, x_sortie]=np.max(img[c, y_f:y_f+pf, x_f:x_f+nf])
                x_f+=1
                x_sortie+=1
            y_f+=1
            y_sortie+=1

    return sortie
```