

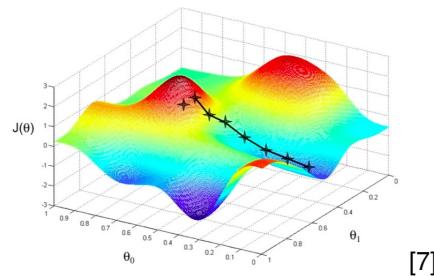
DESCENTE DE GRADIENT

ASPECTS MATHÉMATIQUES ET ALGORITHMIQUES

RAPPORT DE PROJET

Comment la descente de gradient intervient-elle dans la reconnaissance d'images et est-ce que cela peut être bénéfique pour dépolluer les océans ?

Gradient Descent



AUBRY Vincent

CATUSSE Gratient

CHANOINE Axel

GELINEAU Arthur

GUIARD Jean

MOUSSY Lucas

Professeur référent : BERTUOL Florian

Table des matières

I	Préambule	3
A	Remerciement	3
B	Abstract	3
C	Introduction	3
II	Qu'est-ce que la descente de gradient ?	4
A	Aspect mathématique	4
1	Contexte et premières définitions	4
2	Existence et unicité des solutions	4
3	Différentiabilité et conditions de stationnarité	6
4	Principe de l'algorithme	10
B	Implémentation de l'algorithme de descente de gradient en python	11
1	La descente de gradient classique	11
2	La descente avec aléatoire	11
3	La descente avec momentum	12
III	Reconnaissance d'images	13
A	Création de notre réseau	13
1	Création du modèle	13
2	Application	15
B	Participer à la dépollution des océans	15
1	Création d'une base de données	15
2	Comparaison entre notre modèle et TensorFlow	16
3	Bilan écologique	16
4	Conclusion	18
IV	Analyse réflexive du projet	19
A	Rôles	19
B	Logiciels	19
C	Organisation des temps de travail	19
1	Début de séance	19

2	Pendant la séance	19
3	Fin de séance	20
4	Entre les séances	20
V	Annexe	21
A	Learning rate	21
1	Optimiseur Adam [7]	21
B	Illustration de la descente de gradient	22
1	Utilisations de la bibliothèque manim	22
2	Illustration de la descente à une variable :	23
3	Illustration de la descente à deux variables :	23
C	Différentes méthodes de descente de gradient	25
1	Descente de gradient : limites de l'algorithme	25
2	Descente de gradient : Batch	25
3	La descente de gradient stochastique	25
D	Réseau de neurones	26
1	Perceptron monocouche et multicouche	26
2	Entraînement d'un réseau de neurones	30
3	Réseau neuronal convolutif	31
E	Reconnaissance d'images	33
1	Chiffres	33
2	Images diverses	34
3	Définition des modèles MLP et CNN	37
4	Présence de bouteilles dans les océans	46
5	Code de création de la base de données de bouteilles d'eau	52
F	Surapprentissage	55
G	Problème de conditionnement (normalisation)	55
H	Compte-rendus de réunions :	56
1	Réunion 1	56
2	Réunion 2	57
3	Réunion 3	58
4	Réunion 4	59
5	Attribution des rôles	59

I Préambule

A Remerciement

Nous remercions Florian Bertuol pour son implication dans le projet en tant que professeur référent mais aussi comme mentor et tuteur pendant ce projet.

Nous remercions Axel Carrier pour ses cours sur le machine learning. Nous le remercions aussi pour avoir répondu à nos questions que nous avions lors de la construction de ce projet, mais aussi du temps qu'il nous a accordé que ce soit pendant les cours mais aussi après les cours.

B Abstract

In our modern society, ocean pollution is a problem threatening the health of billions of people and endangering countless animal species. This situation is hard to change and one of the problems is that polluted areas are a challenge to find. Having some sort of algorithm that could automatically detect them would be a precious help ! In this context, implementing specialized picture recognition seems the key. In image recognition, as in many other optimization problems, the goal can be simplified to finding the minimum of a function, often called the “Loss function”. In this process, we can very quickly get lost. And we can wonder : can the problem of this search be formulated mathematically ? Can Gradient descent help us when working with numerical problems and the thousands of parameters that make up the so-called “neural networks” ? Is image recognition really a worthwhile path to decrease pollution on Earth ? This report will explore the building of a mathematical and algorithmic framework behind the search for function minimums with Gradient Descent, the performance of “neural networks” and “convolutional neural networks” to recognize pictures and its relevance in the quest to depollute the oceans. After a lot of research, both mathematical and algorithmic frames could be set down, the neural networks developed both manually and using existing libraries worked well on the ocean data. Finally, the environmental profit of these networks was not proven. Image recognition is very powerful, and with objectives guided by an environment expert or a biologist might do great things.

C Introduction

Imaginez, après de longues heures de marche, vous avez atteint le sommet du Mont Everest. Vous êtes rempli de joie et célébrez votre ascension. Mais sans y faire attention, un épais brouillard tombe et vous n'y voyez plus qu'à quelques mètres devant vous. Vous devez absolument rejoindre le camp de base alors vous adoptez la méthode suivante : vous regardez autour de vous et faites un pas dans la direction où la pente est la plus grande. Vous vous arrêtez et recommencez ; en minimisant ainsi votre altitude vous atteignez le bas de la vallée où se trouve le camp. Sans le savoir, vous venez d'utiliser un algorithme bien connu dans le monde de l'intelligence artificielle : la descente de gradient.

Au cœur de l'intelligence artificielle, la descente de gradient intrigue : comment est-elle mise en œuvre ? Pour quels usages ? Et avec quelles conséquences environnementales ?

Nous verrons donc, dans le cadre de ce rapport, les principes mathématiques de la descente de gradient, son implémentation dans l'intelligence artificielle et différents exemples concrets.

Pour ce faire, nous étudierons comment la descente de gradient intervient dans la reconnaissance d'images et si cela peut être bénéfique pour dépolluer les océans.

II Qu'est-ce que la descente de gradient ?

A Aspect mathématique

1 Contexte et premières définitions

L'optimisation est une branche des mathématiques qui vise à minimiser ou maximiser une fonction sur un ensemble. En représentant l'exemple précédent de notre randonneur qui veut atteindre le camp de base le plus proche, il suit le chemin de plus forte pente descendante, afin de minimiser son altitude. Il s'agit alors d'un problème d'optimisation.

Un problème d'optimisation consiste, étant donnée une fonction $f : S \rightarrow \mathbb{R}$, sous réserve d'existence, à trouver :

- Son minimum ν (resp. son maximum) dans (S) .
- Un point $x_0 \in S$ qui réalise ce minimum (resp. son maximum), i.e. $f(x_0) = \nu$.

Vocabulaire :

La fonction f , appelée **fonction objectif ou fonction de coût**, est la fonction dont on cherche un maximum ou un minimum.

Le réel ν est **la valeur optimale**.

Le réel x_0 est **une solution optimale**.

L'ensemble des solutions est noté S .

L'écriture mathématique du problème est : $\min_{x \in S} f(x)$ ou $\operatorname{argmin}_{x \in S} f(x)$.

Selon les propriétés de la fonction objectif, on distingue différents problèmes d'optimisation :

- Optimisation linéaire : $f(x) = \langle c, x \rangle$, où c est un vecteur fixé.
- Optimisation quadratique : $f(x) = \frac{1}{2} \langle Ax, x \rangle + \langle b, x \rangle$, où f est une fonction convexe quadratique et A est une matrice symétrique semi-définie positive.
- Optimisation convexe : f est une fonction convexe, qui garantit des propriétés intéressantes en optimisation.
- Optimisation différentielle : f est une fonction différentiable, permettant l'utilisation d'algorithmes basés sur les gradients.
- Optimisation non différentiable : f n'est pas différentiable en tout point, nécessitant des méthodes alternatives tels que les sous-gradiants.

2 Existence et unicité des solutions

Existence d'un optimum

Nous allons examiner les conditions garantissant l'existence d'une solution à un problème d'optimisation.

Théorème de Weierstrass : [5]

Soit $K \subset \mathbb{R}^d$ un ensemble **fermé** et **borné**, et soit $f : K \rightarrow \mathbb{R}$ une fonction **continue** sur K . Alors f atteint son minimum sur K , c'est-à-dire qu'il existe $x_* \in K$ tel que $f(x_*) = \inf_K f$.

Ce théorème est fondamental en optimisation car il garantit l'existence d'un minimum lorsque certaines conditions sont vérifiées.

Pour bien le comprendre, définissons d'abord les notions de compacité, de continuité et de borné.

Pour cela, nous devons introduire quelques notions de topologie. Nous travaillons sur des applications de l'espace vectoriel normé $(E, \|\cdot\|_E)$ vers l'espace vectoriel normé $(F, \|\cdot\|_F)$.

Définition : Fonction continue.

Soit (f, D_f) une fonction de E dans F . Soit $x_* \in D_f$. On dit que f est continue en x_* si : $\forall \epsilon > 0, \exists \delta > 0, \forall y \in D_f, \|y - x_*\|_E < \delta \Rightarrow \|f(y) - f(x_*)\|_F < \epsilon$.

Définition : Boule ouverte & boule fermée.

La **boule ouverte** de centre $x_0 \in E$ et de rayon $r > 0$ est le sous-ensemble de E défini par $B(x_0, r) := \{x \in E, \|x - x_0\| < r\}$.

La **boule fermée** de centre $x_0 \in E$ et de rayon $r > 0$ est le sous-ensemble de E défini par $\overline{B(x_0, r)} := \{x \in E, \|x - x_0\| \leq r\}$.

Définition : Ouverts & Fermés.

Un sous-ensemble $A \subset E$ est ouvert si $\forall x \in A, \exists r > 0 \mid B(x, r) \subset A$.

Un sous ensemble $A \subset E$ est fermé si son complémentaire est ouvert, i.e. si $\mathbb{R}^d \setminus A$ est ouvert.

Proposition :

$\forall x_0 \in E, \forall r > 0, B(x_0, r)$ est ouvert.

$\forall x_0 \in E, \forall r > 0, \overline{B(x_0, r)}$ est fermé.

De plus, le sous ensemble K d'un espace normé E est borné si : $\exists M \geq 0, \forall x \in K, \|x\| \leq M$

Dans la pratique, on étudie parfois des fonctions $f : \mathbb{R}^d \rightarrow \mathbb{R}$ définie sur tout l'espace \mathbb{R}^d . L'espace \mathbb{R}^d n'étant pas borné, nous ne pouvons donc pas utiliser le théorème de Weierstrass. On utilise donc la notion de coercivité.

Définition : Fonctions coercives.

Soit $f : \mathbb{R}^d \rightarrow \mathbb{R}$. On dit que f est coercive, si : $\forall M \in \mathbb{R}, \exists R > 0$ tel que $\forall x \in \mathbb{R}^d, \|x\| > R \Rightarrow f(x) > M$.

Cela signifie qu'en dehors d'une boule $B(0, R)$, la fonction f prend des valeurs arbitrairement grandes. On note parfois $\lim_{\|x\| \rightarrow \infty} f(x) = +\infty$ lorsque f est coercive.

L'intérêt des fonctions coercives est donné dans le théorème suivant.

Théorème : Existence de minimiseurs pour les fonctions continues coercives. [5]

Soit $f : \mathbb{R}^d \rightarrow \mathbb{R}$ une fonction continue et coercive. Alors f admet et atteint son minimum :

$\exists x_* \in \mathbb{R}^d \mid f(x_*) = \inf_{x \in \mathbb{R}^d} f$.

Unicité du minimum

Interessons nous aussi aux ensembles et fonctions convexes, qui admettent une unicité du minimum. Commençons par définir ce que sont les ensembles et les fonctions convexes.

Définition : Ensemble convexe.

Soit $K \subset \mathbb{R}^d$ un ensemble quelconque. On dit que K est convexe, si : $\forall x, y \in K, \forall 0 \leq t \leq 1, tx + (1-t)y \in K$.

Définition : Fonction convexe.

Soit $K \subset \mathbb{R}^d$ un ensemble convexe et soit $f : K \rightarrow \mathbb{R}$ une fonction définie sur K . On dit que f est convexe si $\forall x, y \in K, \forall t \in [0, 1], f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$.

On dit que f est strictement convexe, si $\forall x \neq y \in K, \forall t \in (0, 1), f(tx + (1-t)y) < tf(x) + (1-t)f(y)$.

L'intérêt de la convexité est donné dans le théorème suivant.

Théorème : Unicité du minimiseur pour les fonctions strictement convexes. [5]

Soit $K \subset \mathbb{R}^d$ un ensemble convexe et soit $f : K \rightarrow \mathbb{R}$ une fonction convexe sur K . Alors si f admet un minimiseur, celui-ci est unique.

3 Différentiabilité et conditions de stationnarité

Pour appliquer des méthodes d'optimisation comme la descente de gradient, la fonction objectif doit être différentiable, ce qui garantit l'existence du gradient et implique la continuité, condition clé du théorème de Weierstrass.

On se place dans le cadre général des espaces vectoriels normés $(E, \|\cdot\|_E)$ et $(F, \|\cdot\|_F)$ pour énoncer la définition de différentiabilité.

Dans la suite, nous écrirons $f : U \subset E \rightarrow F$ pour indiquer que U est un ouvert de E et que f est définie sur U , donc $U \subset D_f \subset E$.

Définition : Différentiabilité en un point.

Soit $f : U \subset E \rightarrow F$ et soit $x_0 \in U$. On dit que f est différentiable en x_0 , s'il existe une application linéaire continue $L(E, F)$ telle que $f(x_0 + h) = f(x_0) + L(h) + o(h)$.

Si c'est le cas, L est unique et est notée df_{x_0} . L'application df_{x_0} est appelé différentielle de f en x_0 .

Si f est différentiable en tout point de U , on dit que f est différentiable sur U .

Nous pouvons lier la notion de différentiabilité à la notion de dérivées partielles, elle même liée à la notion de dérivées directionnelles.

Définition : Dérivées directionnelles.

Soit $f : U \subset E \rightarrow F$, soit $x_0 \in U$ et soit $h \in E$, une direction quelconque de E . On s'intéresse à la fonction réelle, $g_{x_0, h} : \mathbb{R} \rightarrow F$ définie par : $g_{x_0, h}(t) := f(x_0 + th)$.

Si $g_{x_0, h}$ est dérivable en $t = 0$, on dit que f admet une dérivée directionnelle au point $x_0 \in U$, dans la direction $h \in E$.

On la note $\partial_h f(x_0) := \frac{\partial f}{\partial h}(x_0) := g'_{x_0, h}(0) = \lim_{t \rightarrow \infty} \frac{f(x_0 + th) - f(x_0)}{t} \in F$.

Dans le cas important où $E = \mathbb{R}^d$ et $h = e_i$ est le i-ème vecteur de la base canonique de \mathbb{R}^d , on parle plutôt de dérivées partielles. On écrit dans ce cas $\partial_i f(x_0) := \frac{\partial f}{\partial x_i}(x_0) := \lim_{t \rightarrow 0} \frac{f(x_1, \dots, x_{i-1}, x_i + t, x_{i+1}, \dots, x_d) - f(x_1, \dots, x_d)}{t}$.

Théorème : [5]

Si f est différentiable en x_0 , alors pour tout $h \in E$, on a $\frac{\partial f}{\partial h}(x_0) = df_{x_0}(h) \in F$.

De ce résultat, si on se place dans le cas où $E = \mathbb{R}^d$ et $F = \mathbb{R}^m$, on peut représenter l'application $df_{x_0} \in L(E, F)$ par une matrice de $M_{m \times d}(\mathbb{R})$ dans les bases canoniques. Celle-ci, appelée Jacobienne de f en x_0 , généralise la notion de dérivée aux espaces vectoriels normés de dimension finie supérieurs ou égales à 1.

Définition : Jacobienne.

La matrice de $M_{m \times d}(\mathbb{R})$ qui représente $df_{x_0}(h)$ dans les bases canoniques s'appelle matrice Jacobienne et est notée $J_f(x_0) \in M_{m \times d}(\mathbb{R})$.

Théorème : Identification de la Jacobienne. [5]

Si $f : U \subset \mathbb{R}^d \rightarrow \mathbb{R}^m$ est différentiable en $x_0 \in U$, alors les coefficients de la matrice Jacobienne $J_f(x_0)$ sont les dérivées partielles des f_i .

$$\text{Plus exactement, } J_f(x_0) = (\partial x_1 f, \dots, \partial x_d f) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \dots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} (x_0) \in M_{m \times d}(\mathbb{R})$$

La Jacobienne nous donne une information locale complète sur la variation de f , mais dans le cas scalaire, il est souvent plus pratique de travailler avec un objet plus simple : le gradient.

Définition : Gradient.

Soit $f : U \subset \mathbb{R}^d \rightarrow \mathbb{R}$ une fonction réelle différentiable en $x_0 \in U$. Le gradient de f en x_0 est le vecteur de \mathbb{R}^d , noté

$$\nabla f(x_0) \text{ dont les composantes dans la base canonique sont } \nabla f(x_0) := \begin{pmatrix} \partial_1 f \\ \partial_2 f \\ \vdots \\ \partial_d f \end{pmatrix} (x_0)$$

Remarque : Le gradient est la transposée de la Jacobienne. On peut donc écrire : $df_{x_0}(h) = J_f(x_0)h = \langle \nabla f(x_0), h \rangle$.

L'avantage de cette expression, c'est qu'elle montre que la notion de gradient est indépendante du choix de la base de \mathbb{R}^d .

On peut en retenir que le gradient de f en x_0 est le vecteur qui pointe vers la plus grande pente montante de f .

Cela vient de l'inégalité de Cauchy-Schwarz : $df_{x_0}(h) = \langle \nabla f(x_0), h \rangle \leq \|\nabla f(x_0)\|_{\mathbb{R}^d} \cdot \|h\|_{\mathbb{R}^d}$ avec égalité, si h est colinéaire à ∇f_{x_0} .

Théorème : Différentiabilité implique continuité. [5]

Si $f : U \subset E \rightarrow F$ est différentiable en $x_0 \in E$, alors f est continue en x_0 .

Démonstration [5] :

Soit (x_n) une suite qui converge vers x_0 .

Posons $h_n := x_n - x_0$, de sorte que $x_n = x_0 + h_n$.

La suite h_n tend vers 0 dans E . On a : $\|f(x_0 + h_n) - f(x_0)\|_F = \|df_{x_0}(h_n) + \|h_n\|_E \epsilon(h_n)\|_F \leq \|df_{x_0}\|_{op} \|h_n\|_E + \|h_n\|_E \epsilon(h_n)\|_F$.

Comme ϵ est continue en 0 avec $\epsilon(0_E) = 0_F$ et que $h_n \rightarrow 0$, le terme de droite converge vers 0 lorsque $n \rightarrow \infty$.

Ainsi, on a $f(x_n) \rightarrow f(x_0)$, comme souhaité.

Dans la pratique, nous utiliserons une notion plus forte encore pour le montrer, qui est la classe C^1 .

Définition : Classe C^1 .

Soit $f : U \subset E \rightarrow F$ une fonction de classe C^1 sur U et soit $U \ni x \mapsto df_x \in (L(E, F); \|\cdot\|_{op})$ sa différentielle. On dit que f est continûment différentiable ou de classe C^1 sur U si l'application $x \mapsto df_x$ est continue.

Théorème : Caractérisation des fonctions C^1 . [5]

Soit $f : U \subset \mathbb{R}^d \rightarrow \mathbb{R}^m$. La fonction f est de classe C^1 sur U ssi toutes les dérivées partielles $\frac{\partial f}{\partial x_i}$ existent et sont continues sur U .

Si ce théorème est vérifié, nous avons automatiquement l'application $x \mapsto J_f(x)$ ainsi que le gradient qui sont continus, d'après leurs définitions respectives.

De ce qui précède, nous avons l'existence d'un minimum, néanmoins, nous n'avons pas de moyen de le trouver. C'est pourquoi nous nous intéressons à la notion de classe C^2 .

Définition : Différentielles supérieures.

Soit $f : U \subset E \rightarrow F$ une fonction de classe C^1 sur U , de différentielle $df = U \rightarrow L(E, F)$. On dit que f est deux fois dérivable sur U si df est différentiable sur U . On dit que f est de classe C^2 sur U si df est de classe C^1 sur U .

Remarque : Par récurrence, on peut dire que f est de classe C^k sur U si df est de classe C^{k-1} sur U .

De la même manière que nous avons caractérisé les fonctions de classe C^1 (voir théorème), en appliquant ce résultat à la fonction df , on obtient la caractérisation suivante :

Théorème : Caractérisation des fonctions de classe C^2 . [5]

Soit $f : U \subset \mathbb{R}^d \rightarrow \mathbb{R}$. La fonction f est de classe C^2 sur U ssi toutes les dérivées partielles secondes existent et sont continues sur U .

Remarque : Nous pouvons généraliser ce résultat par récurrence puis dire que f est de classe C^∞ si elle est de classe $C^k, \forall k \in \mathbb{N}$.

Théorème : Théorème de Schwarz. [5]

Si $f : U \subset \mathbb{R}^d \rightarrow \mathbb{R}$ est de classe C^2 sur U alors : $\forall x \in U, \forall 1 \leq i, j \leq d, \frac{\partial^2 f}{\partial x_i \partial x_j}(x) = \frac{\partial^2 f}{\partial x_j \partial x_i}(x)$

Cette caractéristique importante nous sera très utile pour la notion suivante, qui est la matrice Hессienne. La Hessianne est essentielle pour analyser la convexité locale de f et permet de distinguer les minima, maxima et points selles.

Définition : Matrice Hessianne.

Soit $f : U \subset \mathbb{R}^d \rightarrow \mathbb{R}$ une fonction de classe C^2 . La Hessianne de f est la matrice de taille $d \times d$ contenant les dérivées

$$\text{croisées de } f. \text{ Explicitement, } H_f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_d \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_d \partial x_1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_d} & \frac{\partial^2 f}{\partial x_2 \partial x_d} & \cdots & \frac{\partial^2 f}{\partial x_d^2} \end{pmatrix}(x) = \left(\frac{\partial^2 f}{\partial x_i \partial x_j} \right)(x).$$

Le théorème de Schwarz assure que cette matrice est symétrique. On remarque aussi que cette matrice est la Jacobienne du gradient.

Définition : Matrices positives et négatives.

On note $S_d(\mathbb{R})$ l'ensemble des matrices symétriques de taille $d \times d$. Pour $A \in S_d(\mathbb{R})$, on note $\lambda_1(A) \leq \lambda_2(A) \leq \dots \leq \lambda_d(A)$ les valeurs propres de A rangées dans l'ordre croissant.

On dit que A est dégénérée si $\lambda_j(A) = 0$ pour un certain $1 \leq j \leq d$.

On dit que A est semi-définie positive si $\lambda_j(A) \geq 0, \forall 1 \leq j \leq d$.

On dit que A est définie positive si $\lambda_j(A) > 0, \forall 1 \leq j \leq d$.

Nous sommes prêts à trouver des critères pour déterminer les minima de fonctions.

Proposition : Les minimiseurs sont des points critiques.

Soit $f : \mathbb{R}^d \rightarrow \mathbb{R}$ une fonction de classe C^2 . Si x_* est un minimiseur local de f , alors $\nabla f(x_*) = 0$ et $H_f(x_*) \geq 0$.

Démonstration [5] :

Soit $\epsilon > 0$, tel que x_* est un minimiseur de f dans $B(x_*, \epsilon)$.

Soit $h \in \mathbb{R}^d \setminus \{0\}$ quelconque.

Pour tout $0 \leq t \leq \epsilon \|h\|^{-1}$, on a $\|th\| \leq |t| \|h\| \leq \epsilon$, et donc $th \in B(0, \epsilon)$.

Ainsi, $f(x_*) \leq f(x_* + th) = f(x_*) + t\langle \nabla f(x_*), h \rangle + \frac{1}{2}t^2 \langle h, H_f(x_*)h \rangle + t^2 h^2 \epsilon(th)$. avec $\epsilon(th) \rightarrow 0$ lorsque $t \rightarrow 0$.

En simplifiant par $f(x_*)$ et en divisant par t , on obtient $\langle \nabla f(x_*), h \rangle + \frac{1}{2}t \langle h, H_f(x_*)h \rangle + th^2 \epsilon(th) \geq 0$.

En prenant la limite $t \rightarrow 0$, on obtient $\langle \nabla f(x_*), h \rangle \geq 0$. Ceci devant être vrai pour tout $h \in \mathbb{R}^d$, on en déduit que $\nabla f(x_*) = 0$.

En reprenant l'inégalité précédente, on trouve $\frac{1}{2}t \langle h, H_f(x_*)h \rangle + th^2 \epsilon(th) \geq 0$. On divise à nouveau par t et on prend la limite lorsque $t \rightarrow 0$ pour obtenir cette fois que $\langle h, H_f(x_*)h \rangle \geq 0$.

Ceci étant vrai pour tout $h \in \mathbb{R}^d$, on conclut que $H_f(x_*)$ est une matrice symétrique positive.

Définition : Points critiques, points critiques non dégénérés.

Soit $f : U \subset \mathbb{R}^d \rightarrow \mathbb{R}$ une fonction de classe C^2 sur U . On dit que x_* est un point critique de f si $\nabla f(x_*) = 0$.

Un point critique est dit non dégénéré, si de plus $H_f(x_*)$ est non dégénérée ($\det H_f(x_*) \neq 0$).

Un point critique qui n'est ni un minimiseur local, ni un maximiseur local, est un point selle.

En combinant à la fois la proposition et la définition précédente, nous obtenons la proposition suivante :

Proposition :

Soit $f : \mathbb{R}^d \rightarrow \mathbb{R}$ une fonction de classe C^2 . Si $x_* \in \mathbb{R}^d$ tel que $\nabla f(x_*) = 0$ et $H_f(x_*) > 0$ alors x_* est un minimiseur local non dégénéré de f .

Si $H_f(x)$ est positive et le gradient nul ($\nabla f(x) = 0$, cela signifie que la fonction est « courbée vers le haut », garantissant un minimum local en x .

Proposition :

Si f est de classe $C^1(\mathbb{R}^d, \mathbb{R})$, alors f est convexe ssi $\forall x, y \in \mathbb{R}^d, f(x) \geq f(y) + \langle \nabla f(y), x - y \rangle$. Si f est de classe $C^2(\mathbb{R}^d, \mathbb{R})$, alors f est convexe ssi $H_f(x)$ est une matrice définie positive, pour tout $x \in \mathbb{R}^d$.

4 Principe de l'algorithme

L'idée de la méthode est de générer une suite (x_n) qui converge vers un minimiseur local de f . On construit donc la suite par récurrence avec x_0 un point initial quelconque puis on le définit par récurrence, comme tel

$$x_{n+1} = x_n - \alpha \nabla f(x_n).$$

Ici, α est un paramètre strictement positif appelé « pas » ou learning rate en anglais. C'est un paramètre important qui influence la convergence de la suite A.

Étudions maintenant la convergence de la suite (x_n) générée par la descente de gradient en utilisant une approche topologique. Grâce à la notion de boule ouverte, nous pouvons reformuler la convergence en termes plus abstraits.

Théorème : Caractérisation de la limite. [5]

Un point x_l est la limite de la suite (x_n) ssi pour tout ϵ , il existe $N \in \mathbb{N}$ tel que pour tout $n \geq N$, on a $x_n \in B(x_l, \epsilon)$.

Autrement dit, si x_l est la limite de la suite x_n (pour la norme $\|\cdot\|$), alors quelque soit le voisinage de x_l , tous les termes de la suite sont dans ce voisinage à partir d'un certain rang.

Pour que cette suite soit convergente et d'après la définition de l'algorithme de la descente de gradient, il suffit de respecter deux conditions :

- La suite (x_n) elle-même doit converger vers un point x^* , c'est à dire $x_n \rightarrow x^*$ lorsque $n \rightarrow \infty$.
- Le gradient doit tendre vers zéro : $\nabla f(x_n) \rightarrow 0$

Lorsque ces deux conditions sont réunies, on peut conclure que la limite x^* est un point stationnaire, autrement dit $\nabla f(x^*) = 0$.

Remarque : La vitesse de convergence du programme peut être étudié plus en détail à la référence suivante [5].

B Implémentation de l'algorithme de descente de gradient en python

Nous avons codé dans le cadre du projet plusieurs versions de la descente de gradient à commencer par l'algorithme de base :

1 La descente de gradient classique

```
1 import random
2 def descente_gradient(derivées, départ, epsilon, pas):
3     """Descente de gradient classique à n variables
4     IN : derives (list(function)) - les dérivées selon chaque variable de la fonction
5     IN : départ (list(float)) - le point de départ de la descente
6     IN : epsilon (float) - la valeur de la dérivée à partir de laquelle on arrête la descente (on n'atteint pas forcément le 0)
7     IN : pas (float) - la vitesse de convergence (ne doit pas être trop gros ou trop petit)"""
8     iterations = 0
9     terminer = False
10    while not terminer:
11        terminer = True
12        pentes = [derivées[i](*départ) for i in range(len(derives))]
13        for var in range(len(pentes)):
14            départ[var] -= pentes[var] * pas
15            if abs(pentes[var]) > epsilon:
16                terminer = False
17        iterations += 1
18        print(départ)
19    return départ, iterations
20 print(descente_gradient([lambda x, y: 2*x, lambda x, y: 2*y], # ex d'appel
21                         [random.randint(-100, 100), random.randint(-100, 100)],
22                         0.01,
23                         0.4))
```

Listing II.1 – DescenteClassique.py.

Cette fonction applique la descente de gradient à une fonction à n variables dont les dérivées partielles sont connues. Le programme n'a pas besoin de la formule explicite de la fonction car seule la différentielle est utilisée pour se rapprocher itération après itération d'un minimum.

- Le paramètre « `derives` » est une liste contenant les dérivées partielles selon chaque variable de la fonction (à noter que ces dérivées sont passées sous forme de fonctions ayant comme paramètres toutes les variables même si certaines ne sont pas utilisées par raison de simplicité du code)
- Le paramètre « `départ` » représente les coordonnées du point de départ de la descente.
- Le paramètre « `epsilon` » représente la condition d'arrêt de l'algorithme. Le programme s'arrête une fois que la valeur absolue de chaque dérivée partielle au point actuel est inférieure à ce dernier.
- Le paramètre « `pas` » est multiplié à la valeur de chaque dérivée afin de définir la vitesse de déplacement à chaque itération.

2 La descente avec aléatoire

Afin de régler le cas où l'algorithme s'arrête dans une zone où la dérivée est nulle mais qui n'est pas un minimum (en 0 de la fonction x^3 par exemple) on peut simplement ajouter une composante d'aléatoire qui déplace le point à chaque itération, ce qui

peut permettre de débloquer l'algorithme :

<https://github.com/Synchrones/DescenteGradient/blob/master/algos/DescenteRandom.py>.

Le code est identique à l'exception de l'ajout du déplacement aléatoire.

3 La descente avec momentum

```
1 import random
2 import numpy as np
3 def descente_gradient(derives, depart, epsilon, pas, facteur_momentum):
4     """Descente de gradient classique à n variables
5     IN : derives (list(function)) - les dérivées selon chaque variable de la fonction
6     IN : depart (list(float)) - le point de départ de la descente
7     IN : epsilon (float) - la valeur de la dérivée à partir de laquelle on arrête la descente (on n'atteint pas forcément le 0)
8     IN : pas (float) - la vitesse de convergence (ne doit pas être trop gros ou trop petit)
9     IN : facteur_momentum (float) - le facteur de prise de momentum"""
10    iterations = 0
11    terminer = False
12    nouvelles_coords = [0 for _ in range(len(derives))]
13    momentum = [0 for _ in range(len(derives))]
14    while not terminer:
15        terminer = True
16        pentes = [derives[i](*depart) for i in range(len(derives))]
17        for var in range(len(pentes)):
18            nouvelles_coords[var] = depart[var] - pentes[var] * pas + facteur_momentum * momentum[var]
19            momentum[var] = nouvelles_coords[var] - depart[var]
20            depart[var] = nouvelles_coords[var]
21            if abs(pentes[var]) > epsilon:
22                terminer = False
23        iterations += 1
24        print(depart)
25    return depart, iterations
26 print(descente_gradient([lambda x, y: 2*x, lambda x, y: 2*y], [random.randint(-100,100),random.randint(-100,100)],0.01,0.4,0.1))# ex d'appel
```

Listing II.2 – DescenteMomentum.py

Pour les cas où l'algorithme se retrouverait bloqué dans un minimum local, l'algorithme de descente avec momentum peut être utile. On réalise à chaque itération une combinaison linéaire du déplacement actuel ainsi que du précédent. Pour cela, la fonction possède un nouveau paramètre « facteur_momentum » qui représente à quel point le déplacement doit prendre du momentum (de la même manière qu'une balle descendant une pente).

Formule de la descente de gradient avec momentum pour l'itération t+1 :

$$\begin{aligned} m_{t+1} &= \alpha \nabla f(x) + \beta m_t \\ x_{t+1} &= x_t - m_{t+1} \end{aligned}$$

Avec α représentant le pas et β le facteur de momentum.

III Reconnaissance d'images

A Création de notre réseau

1 Création du modèle

Afin de nous familiariser avec le principe de réseau neuronal et d'avoir un modèle le plus personnalisable possible, nous avons besoin de comprendre quelques généralités sur ceux-ci.

Les perceptrons monocouche, les perceptrons multi-couche (MLP - Multi-Layer Perceptron) ainsi que les réseaux de neurones convolutifs (CNN - Convolutional Neural Networks) sont trois techniques d'apprentissage supervisé en machine learning, aussi appelées modèles. De plus, nous verrons qu'ils ont des architectures et des applications différentes.

Tout modèle fait correspondre à une donnée un label qu'il prédit. Nous pouvons nous intéresser aux différentes phases de la vie de notre modèle. Le modèle passe d'abord par une phase d'apprentissage supervisé, aussi appelée phase d'entraînement, durant laquelle il apprend à partir de données étiquetées. Une fois cette phase terminée, nous testons le modèle sur de nouvelles données non étiquetées pour évaluer sa capacité à généraliser. Cette phase est appelée phase de test. Une dernière phase pourrait être le maintien du modèle, mais elle ne sera pas détaillée. Il faut noter qu'en fonction du modèle choisi, ces phases sont différentes.

L'entraînement sert à obtenir les paramètres (les poids et les biais) optimaux de notre modèle, pour que celui-ci nous renvoie les meilleures solutions. Cette phase se fait en plusieurs étapes :

- Séparation des données
- Optimisation des paramètres
- Évaluation des performances

Intéressons-nous à une première phase de l'entraînement qui est commune à tous les modèles : la séparation des données.

Tout d'abord, quand nous avons notre base de données, il faut la séparer en sous-ensembles : les données d'entraînement, les données de validation et les données de test. Les données d'entraînement sont les données sur lesquelles est effectuée l'optimisation des paramètres. Les données de validation sont utilisées pendant l'entraînement pour calculer différentes métriques permettant de sélectionner les meilleurs hyperparamètres du modèle. Enfin, les données de l'ensemble de tests ne sont utilisées qu'à la fin afin d'estimer les performances du modèle sur un ensemble jamais vu par ce dernier.

Cependant, on pourrait se demander comment s'assurer qu'un jeu de données se généralise à un autre ? Prenons un exemple pour illustrer : si nous entraînons notre modèle à reconnaître des bouteilles dans l'océan, qui est donc sur un fond bleu et que nous présentons une bouteille sur une plage, le modèle aura beaucoup de mal à reconnaître la bouteille, car il n'aura jamais appris sur des données avec un fond de plage.

Il faut donc chercher à avoir dans notre base de données, ainsi que dans chaque sous-ensemble de celui-ci, un nombre similaire de données pour chaque classe et des données variées représentant la réalité, pas seulement une situation précise. De plus, il faut adapter les proportions des sous-ensembles en fonction des besoins.

Cette étape est très importante car elle déterminera les performances de notre modèle.

La phase de l'optimisation des paramètres diffère en fonction de notre modèle et sera présentée plus en détail en annexe. Cependant, elle s'appuie sur l'algorithme de la descente de gradient.

Nous avons donc mis en pratique ces connaissances à travers différents exemples : La reconnaissance de chiffres et la reconnaissance de bouteilles dans l'océan.

Structure du réseau

Remarque : le code définissant le réseau peut être retrouvé ici :

https://github.com/Synchrones/DescenteGradient/blob/master/algos/Reseau_objet.py

Par soucis de clarté, nous avons implémenté la structure de notre réseau neuronal sous la forme de classes python. Cela nous permet de voir clairement de quoi il est constitué et de définir des fonctions utiles sur ce dernier.

Nous avons deux classes pour cela : une classe « Reseau » et une classe « Neurone ».

- La classe « Reseau » possède deux attributs : « couches » qui représente les couches de neurones du réseau sous forme d'une liste de liste de neurones et « nb_entrees », un entier représentant le nombre d'entrées passées au réseau. Nous avons fait le choix de lier chaque neurone d'une couche donnée à chaque neurone de la couche suivante. Cela augmente drastiquement le nombre de poids sur un réseau avec de grandes couches mais à l'avantage d'être plus simple à implémenter. Ce choix peut avoir un impact sur la performance du réseau et ne permet pas de créer un réseau convolutif [8].

Cette classe possède également deux fonctions : « importer_poids() » et « exporter_poids() » qui permettent comme leurs noms l'indique de sauvegarder et charger les poids d'un réseau via un fichier json. Nous avons fait le choix d'arrondir la valeur des poids à 8 chiffres après la virgule par souci de taille de fichier.

- La classe « Neurone » représente un nœud du réseau, c'est-à-dire l'ensemble des poids des liaisons entre cette neurone et celles de la couche précédente ainsi qu'une fonction d'activation. La classe a également besoin de la dérivée de cette dernière qui est ensuite utilisée pendant l'entraînement du réseau. Pour une couche de n neurones, les neurones de la couche suivante possèdent n+1 poids car le poids d'indice 0 est associé au biais.

Exemple d'utilisation :

```
1 from Reseau_objet import *
2 reseau = Reseau(nb_entrees=2, couches=[[1, Relu, derive_Relu], (2, Relu, derive_Relu)])
3 # reseau.couches = [[Neurone1,1], [Neurone2,1, Neurone2,2]]
4 # Neurone1,1.poids = [poids0, poids1, poids2] car 2 entrées sur la couche précédente
5 neurone = Neurone(poids=[0.2, 0.3, 0.1], fonction_activation=Relu, derive_fonction_activation=derive_Relu)
```

Entraînement et utilisation du réseau

Remarque : le code de cette partie peut être retrouvé ici :

https://github.com/Synchrones/DescenteGradient/blob/master/algos/reconnaissance_chiffres_objet.py

Dans le cadre de l'entraînement de notre réseau neuronal, deux fonctions principales sont nécessaires : la passe avant et la passe arrière.

Nous avons décidé de coder ces fonctions à l'aide de boucles « for ». Il est fréquent et populaire de les implémenter sous forme de calculs matriciels pouvant être exécutés sur des processeurs graphiques, permettant de très bonnes performances

[14]. Nous n'avons cependant pas fait ce choix afin d'expliciter clairement les calculs réalisés pour mieux comprendre le fonctionnement des différents algorithmes.

Notre implémentation de la fonction de passe avant prend en paramètres un réseau et une entrée à lui passer sous forme d'une liste et renvoie l'ensemble des données nécessaires à la passe arrière (entrée du réseau et valeurs avant et après application de la fonction d'activation pour chaque neurone). Cette fonction a la possibilité d'être exécutée en parallèle sur différentes entrées afin d'accélérer le processus d'entraînement.

Notre fonction de passe arrière prend en paramètres un réseau, une liste représentant les sorties attendues du réseau, une liste des résultats donnés par l'appel de la fonction passe avant sur les entrées et un float représentant le facteur d'apprentissage. La fonction segmente la liste des sorties attendues couplées aux sorties du réseau afin d'exécuter l'algorithme de rétropropagation du gradient en parallèle.

2 Application

Les premières applications de notre réseau sont détaillées en annexe.

B Participer à la dépollution des océans

1 Création d'une base de données

Une base de données est un ensemble d'informations structurées et accessibles au moyen d'un logiciel [10]. Pour faire simple, et appliqué à notre projet, c'est un fichier qui contient toutes les images nécessaires à l'entraînement de notre réseau de neurones de manière accessible.

La création d'une base de données est un processus fastidieux et qui peut entraîner énormément de biais dans le fonctionnement des réseaux, le problème principal étant celui du surapprentissage. Il faut donc commencer par établir plusieurs critères sur nos réseaux, qui correspondent à nos besoins :

- Leur portée, à quel point l'on souhaite qu'ils puissent généraliser et sur quelles images veut-on qu'ils soient pertinents.
- Leur capacité, combien de paramètres au maximum peuvent-ils avoir, cela conditionnera notamment la résolution des images de la base de données.
- Le problème que l'on cherche à résoudre, ce que l'on cherche à reconnaître.

Pour nos réseaux, ces critères semblent assez simples à établir. Notre problème est le suivant : pour une image donnée de la surface des océans, nos réseaux doivent déterminer si un déchet est présent ou non. Nous allons rester très modestes et réaliser une base de données extrêmement simpliste : des images bleues représentant la surface des océans, de soixante pixels de côté, sur lequel une bouteille de plastique peut être présente, ou non (voir la base de données). Ces images seront ensuite enregistrées sur un fichier .csv.

Nos réseaux auront, par ces choix, une portée très limitée : confrontés à des images extérieures à la base de données, ils seront inefficaces, mais notre objectif n'est pas d'être généraliste, mais plutôt d'explorer ce qui est faisable avec des puissances de calcul raisonnables.

Même si la portée de nos réseaux est délibérément limitée, nous souhaitons que sur notre problème les réseaux généralisent au maximum, c'est pourquoi nous évitons le surapprentissage via de l'augmentation de données et une base de données importante, plus de 6000 images.

Maintenant que les caractéristiques de notre base de données sont établies, nous pouvons établir un algorithme qui permet de la créer.

2 Comparaison entre notre modèle et TensorFlow

TensorFlow est une librairie python qui permet de créer et d'entraîner des réseaux de neurones de manière optimisée. Il existe d'autres librairies similaires mais TensorFlow est une des principales compatibles avec notre équipement. Après plusieurs essais de réseaux à la recherche de paramètres optimaux, nous avons pu atteindre une précision de 87% sur la reconnaissance des bouteilles de notre base de données. Ce résultat a été obtenu avec un réseau possédant une couche de cent neurones utilisant la fonction d'activation ReLU suivie d'une couche de deux neurones utilisant la fonction Softmax pour les sorties. L'entraînement de ce réseau a duré une heure et, sur cette période, l'algorithme a pu parcourir deux epochs.

```
1 """
2 Test du réseau :
3 ...
4 label : 1      // sortie du réseau : [0.07325014259913828, 0.9267498574008617]
5 label : 1      // sortie du réseau : [1.7818333691105177e-06, 0.9999982181666309]
6 label : 0      // sortie du réseau : [0.9972853294865964, 0.0027146705134035555]
7
8 Précision : 0.873
9 Temps moyen d'exécution de la passe avant : 1.9383483996758095 sec
10 Temps moyen d'exécution de la passe arrière : 3.268905494763301 sec
11 Temps total de l'exécution : 988.9345817565918 sec
12 """
```

Listing III.1 – Fin de la deuxième partie de l'entraînement

À l'aide de TensorFlow, nous avons entraîné un modèle sur cinquante epochs. Cela a duré cinq minutes. Nous avons obtenu que le modèle est précis à 99,89 % sur des données qu'il n'a jamais vu. Ce pourcentage très élevé s'explique car notre problème est plutôt simple.

Comparé au réseau de neurones utilisant TensorFlow, notre réseau est bien plus lent (5 secondes par itération de l'entraînement comparé à quelques centièmes de secondes pour TensorFlow). Ceci peut être en partie expliqué par le matériel exécutant le programme : notre code, bien que parallélisé, est exécuté sur le processeur là où celui de TensorFlow est exécuté sur processeur graphique. La capacité de parallélisation de ces derniers est largement supérieure (12 exécutions en parallèle pour notre meilleur processeur contre plusieurs milliers pour un processeur graphique). Le code de TensorFlow est également mieux optimisé et plus efficace, utilisant des techniques comme l'optimiseur ADAM pour améliorer la vitesse de convergence du modèle.

3 Bilan écologique

Nous avons clairement observé que les deux modèles, qu'ils soient « faits maison » ou utilisant une bibliothèque déjà existante et optimisée « TensorFlow » parviennent à de bonnes performances sur les données « océaniques ».

Cependant, d'un point de vue écologique global, un des biais principaux à éviter est le GreenWashing .[9]

Le simple fait d'être capable de reconnaître des images de bouteilles dans l'océan est-il réellement intéressant ?

De plus, tous ces calculs, réalisés sur machine, ne sont pas gratuits pour la planète : chaque opération utilise une certaine

puissance, et ce, depuis le début de notre projet.

Bilan écologique des programmes

Pour mesurer l'impact écologique de notre code, nous avons utilisé le module Carbon tracker. Cette bibliothèque calcule la quantité d'électricité consommée (et donc de CO₂) durant l'entraînement d'un modèle. Elle sert donc à mesurer l'impact écologique des modèles d'intelligence artificielle. La mesure de référence est fixée à la valeur moyenne par défaut : 56.04 g de CO par kWh (pour la France en 2023). Nous avons donc implémenté Carbon tracker à un modèle de convolution codé en TensorFlow. Il est important de noter que ce type de modèle consomme plus d'énergie (voir les annexes sur la reconnaissance d'images). Par ailleurs, le code, la preuve de sa convergence et les difficultés que nous avons pu rencontrer pour l'entraîner sont disponibles en annexes, « Présences de bouteilles dans les océans ».

Le module nous renvoie la consommation pour une epoch :

```
Actual consumption for 1 epoch(s):
  Time: 0:04:47
  Energy: 0.004583014985 kWh
  CO2eq: 0.256825697683 g
  This is equivalent to:
  0.002389076258 km travelled by car.
```

FIGURE III.1 – Résultat du tracker de Carbone

Bilan écologique du projet

Après avoir réalisé le bilan écologique de l'exécution des programmes, il nous a semblé important de réaliser un bilan écologique plus global, sur l'ensemble du projet. En effet, tout au long des sept semaines du projet, tous les membres du groupe se sont activement servis d'au moins un ordinateur personnel chacun. En plus de la consommation à chaque exécution du programme, on pourrait donc rajouter la consommation de la création du programme et de tout le projet. C'est ce que nous allons essayer de faire, il y aura de nombreuses approximations dues à la complexité importante de ce calcul. L'objectif est d'avoir une idée de l'énergie qui a été consommée par ce projet.

Nous pouvons séparer la consommation énergétique du projet en 3 sources principales :

- La consommation directe des ordinateurs.
- La consommation des serveurs qui stockent les données en ligne.
- Les consommations extérieures aux ordinateurs.

Pour chacune de ces parties, nous chercherons à calculer les énergies consommées en kg équivalents CO₂ notées respectivement E_{ord} , E_{serv} , E_{ext} . On aura alors l'énergie totale consommée $E_{tot} = E_{ord} + E_{serv} + E_{ext}$.

- Calcul de E_{ord} .

Ce calcul est l'un des plus simples, car c'est ici où nous avons les données les plus précises, chacun des six membres du groupe a passé environ 30 h à utiliser activement son ordinateur personnel, qui consomme une puissance d'environ 70 W. On

a donc une première composante de E_{ord} qui est de $6 \times 30 \times 70 = 12,6 \text{ kWh}$ cette énergie provient du réseau électrique français sur lequel 1 kWh est équivalent à 0,06 kg de CO₂ [2] ce qui donne 0,756 kg de CO₂. Une deuxième composante majeure de la consommation due aux ordinateurs est leur fabrication, mais nous avons décidé de ne pas la compter car les ordinateurs n'ont pas été fabriqués spécialement pour le projet. Ainsi, nous obtenons $E_{ord} = 0,756 \text{ kg équivalent CO}_2$.

— Calcul de E_{serv} .

Ce calcul commence à être plus complexe, de très nombreuses données qui ont été utilisées durant le projet furent stockées sur des serveurs en ligne. La première étape est de comptabiliser la taille totale des fichiers liés au projet stockés en ligne, on obtient environ 735 Mo. Ensuite, ces données ont été stockées sur des serveurs européens pendant environ deux mois ce qui, après calcul, donne une centaine de grammes d'équivalent CO₂.

— Calcul de E_{ext} .

C'est dans ce calcul que l'on peut se perdre. En effet, si l'on comptait réellement l'impact écologique de tout ce qui a permis de réaliser le projet, il faudrait des pages et des pages de calcul. Nous allons donc nous concentrer sur les essentiels. Le transport : un membre de l'équipe a pris la voiture seul pour 15,2 km durant les sept séances, tandis que les autres ont utilisé les transports en commun pour 15 km. Ce qui donne $7 \times 3,31 + 5 \times 7 \times 1,72 = 83,37 \text{ kg équivalents CO}_2$.[1]

En rassemblant les trois grandeurs, on obtient donc : $E_{tot} = 0,756 + 0,100 + 83,37 = 84,226 \text{ kg équivalent CO}_2$. Cette quantité d'énergie n'est pas monumentale, mais reste conséquente ; en effet, notre projet reste à une échelle limitée, mais pour des projets concernant des réseaux plus importants, capables de tâches plus complexes, la consommation explose.

4 Conclusion

Ce projet avait pour but de dépolluer les océans en effectuant de la reconnaissance d'image à l'aide de programmes utilisant la descente de gradient. Nous avons dans ce contexte acquis de nombreuses compétences techniques et méthodologiques. Les recherches sur la descente de gradient nous ont apporté des connaissances en intelligence artificielle que nous avons appliquées à la création et l'utilisation de réseaux neuronaux. Nos programmes ont permis de reconnaître des images de bouteille de manière efficace, obtenant un taux de précision au-delà de 90%. Cependant il est bon de préciser que notre jeu d'images est simpliste et qu'il avait plutôt pour vocation de valider la faisabilité d'un projet similaire à plus grande échelle. L'utilité d'un tel projet est aussi questionnable et l'avis d'un spécialiste serait souhaitable afin de définir des objectifs permettant un réel impact positif sur l'environnement.

IV Analyse réflexive du projet

A Rôles

Pour ce premier projet de grande envergure réalisé en groupe, la question de l'organisation était centrale. Une première étape a été la répartition des rôles lors de chaque séance. À chaque séance, il y a eu une rotation des rôles permettant à chacun d'améliorer les capacités qu'ils impliquent (Leadership, communication, etc.).

B Logiciels

Une deuxième composante importante dans la réalisation du projet a été le choix des logiciels collaboratifs utilisés. En voici une liste exhaustive :

- Google doc a été utilisé pour un document partagé, pour prises de notes, brouillons, recherches personnelles, etc.
- Notion, outil collaboratif, nous a permis d'avoir un agenda partagé avec objectif des séances et répartition en rotation des rôles.
- Instagram, outil de communication, a été utile pour communiquer entre les membres du groupe et pour organiser la logistique des séances (salles, heure, etc.).
- Planète, outil de communication pour planifier les réunions et rencontres avec le professeur référent Florian Bertuol.
- Git, outil de synchronisation de fichiers, nous a permis de réaliser nos programmes au sein d'un dossier synchronisé entre les membres du groupe.
- Overleaf, outil de rédaction collaboratif, a rendu aisée la rédaction de la version finale du compte-rendu, où tout le monde voyait l'avancée des autres.

C Organisation des temps de travail

La dernière difficulté dans la réalisation d'un tel projet a été pour nous l'organisation des temps de travail, que nous avons découpé en quatre.

1 Début de séance

Durant ce temps, les membres du groupe se rassemblent dans la salle annoncée en amont de la séance sur un groupe Instagram dédié aux annonces.

Chacun sait le rôle qu'il doit jouer grâce au tableau de répartition des rôles sur Notion.

Chacun arrive avec son travail fait et un tour de table se lance où chacun fait part de ses avancées dans le projet à l'oral.

Nous définissons ensuite les objectifs de la séance, en intelligence collective grâce aux rôles de chacun.

2 Pendant la séance

Ici, les membres du groupe réalisent au maximum les objectifs donnés en début de séance, et effectuent la réunion avec le professeur référent si elle est prévue.

3 Fin de séance

Une fois la séance terminée, un tour de table (rapide) est effectué, chacun fait part de ses progrès durant la séance. Une discussion pour redéfinir la direction du projet est lancée. Nous y discutons des objectifs à long et court termes ainsi que de la répartition des devoirs pour la prochaine séance.

À l'issue de celle-ci, nous faisons un récapitulatif des devoirs sur le groupe annonce. Ils sont ensuite ajoutés au calendrier Notion, avec de nouveaux objectifs si besoin.

4 Entre les séances

C'est entre les séances que la majorité du travail est effectué car c'est là où le groupe a le plus de temps. La communication a lieu via un autre groupe Instagram intitulé « descente de Gratien » lorsqu'un travail de collaboration entre plusieurs membres est nécessaire.

Enfin, chacun doit prendre connaissance de son rôle pour la future séance et s'y préparer.

V Annexe

A Learning rate

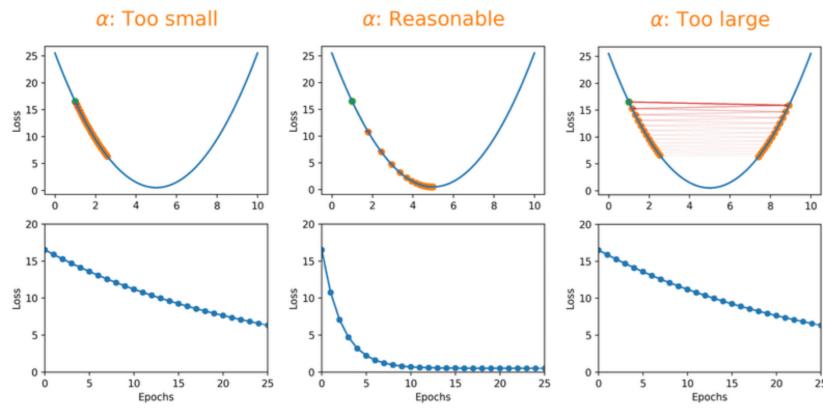


FIGURE V.1 – Illustration de différentes valeurs du «learning rate»

Le taux d'apprentissage, noté α , détermine la vitesse à laquelle le modèle converge. Lorsqu'il est trop petit, la descente de gradient est très lente (à gauche). S'il est bien choisi, le modèle converge rapidement vers un minimum de la fonction de perte (au milieu). En revanche, un taux trop élevé empêche une convergence rapide et peut même parfois diverger (à droite). Un bon réglage du paramètre est donc important pour un apprentissage efficace et rapide.

1 Optimiseur Adam [7]

Définition : L'optimiseur Adam (Adaptive Moment Estimation)

Représente une variante avancée de la descente de gradient. Il ajuste dynamiquement le taux d'apprentissage en utilisant une moyenne exponentielle des gradients passés et de leurs carrés, ce qui stabilise la convergence. Adam est particulièrement efficace pour les problèmes d'optimisation complexes avec de grandes quantités de données et des espaces de paramètres vastes.

Le fonctionnement d'Adam repose sur plusieurs étapes essentielles. Tout d'abord, à chaque itération, on commence par calculer le gradient de la fonction de perte par rapport aux paramètres du modèle. Ce gradient est noté g_t . Ensuite, pour mieux suivre l'évolution des gradients sur plusieurs itérations, on met à jour la première estimation du moment, qui est une moyenne mobile des gradients passés.

Cette mise à jour suit la formule : $m_t = \beta_1 \times m_{t-1} + (1 - \beta_1) \times g_t$.

où le paramètre β_1 (moment d'ordre 1) contrôle l'influence des gradients passés, avec une valeur par défaut de 0.9. Plus β_1 est proche de 1, plus l'apprentissage est lent.

À l'inverse, une valeur plus faible rend l'apprentissage plus réactif aux changements soudains. En parallèle, on met à jour une seconde estimation du moment, qui est cette fois une moyenne mobile des carrés des gradients passés, selon la formule :

$$v_t = \beta_2 \times v_{t-1} + (1 - \beta_2) \times g_t \odot g_t$$

avec le paramètre β_2 (moment d'ordre 2) qui contrôle l'influence des carrés des gradients passés, avec une valeur par défaut de 0.999. Plus β_2 est élevé, plus le suivi des gradients est stable mais lent. À l'inverse, une valeur plus basse augmente la réactivité tout en introduisant plus de bruit.

L'un des défis des premières itérations d'Adam est que ses estimations de moments peuvent être biaisées.

Pour compenser ce biais, on applique une correction qui normalise ces valeurs : $m^t = \frac{m_t}{1-\beta_1^t}$ & $v^t = \frac{v_t}{1-\beta_2^t}$

Ces valeurs corrigées permettent alors de mettre à jour les paramètres du modèle de manière plus précise et plus stable.

La mise à jour des paramètres suit la règle suivante : $\theta_t = \theta_{t-1} - \alpha \times \frac{m^t}{\sqrt{v^t + \epsilon}}$

Où α est le taux d'apprentissage, qui contrôle l'ampleur des mises à jour des poids. Par défaut, il est souvent fixé à 0.001. Si l'apprentissage est trop lent, on peut l'augmenter à 0.01, mais une valeur trop élevée risque de rendre l'entraînement instable. Le réel ϵ est un petit terme ajouté pour éviter toute division par zéro et sa valeur par défaut : 10^{-8} (ne change généralement pas). Si la perte oscille trop ou diverge, on peut le réduire (exemple : 0.0001).

Produit de Hadamard

Définition : Produit de Hadamard

Représente une opération qui s'applique à deux matrices de même taille ($n \times m$). C'est une multiplication élément par élément, ce qui signifie qu'on multiplie chaque élément d'une matrice avec l'élément correspondant de l'autre matrice.

Exemple : Pour $M = A \odot B$ ou $M = A \circ B$

$m_{1,1} = a_{1,1} \times b_{1,1}$ avec \times qui correspond au produit entre deux réels.

B Illustration de la descente de gradient

Afin d'illustrer l'algorithme de descente de gradient, nous utilisons la bibliothèque manim. Bien qu'elle soit principalement utilisée pour la réalisation de vidéos mathématiques, elle peut également produire des animations interactives en temps réel. Nous avons deux animations représentant graphiquement la descente de gradient, une pour les fonctions à une variable (2D) et une pour les fonctions à deux variables (3D).

1 Utilisations de la bibliothèque manim

La bibliothèque manim s'utilise via des commandes dans le terminal de la forme ci-dessous :

```
1 manim -paramètres fichier classe
```

Les classes héritant de la classe Scene sont interprétées par manim comme des animations et peuvent être exécutées via cette commande.

Paramètres importants :

- La qualité : définie la qualité de l'animation produite et donc le temps de rendu (-ql pour faible qualité, -qm pour moyenne, -qh pour haute)
- Le paramètre « départ » représente les coordonnées du point de départ de la descente.
- Le rendu en temps réel : nous utilisons manim pour avoir des animations interactives, le paramètre -p permet cela.
- Le moteur de rendu : afin d'avoir de l'interactivité nous utilisons le moteur de rendu opengl (-renderer=opengl). Ce moteur de rendu n'étant pas celui par défaut, il est peu documenté et apporte son lot de problèmes (cf : modification manim).

Note importante : Avant utilisation du programme d'illustration de la descente de gradient à 2 variables, il est nécessaire de suivre les instructions de l'Annexe 1 car des modifications ont été faites à la bibliothèque manim.

Annexe 1

Comme dit précédemment, le moteur de rendu opengl n'est pas celui par défaut et est donc peu utilisé. De ce fait, certaines fonctionnalités ne sont pas fonctionnelles, voire manquantes. C'est le cas pour les surfaces et notamment pour leur coloration. Nous avions besoin de pouvoir colorer des surfaces par coordonnées et avons donc modifié certains fichiers pour que cela nous soit possible.

Les fichiers à ajouter/remplacer dans le module manim se trouvent dans ce dossier :

DescenteGradient/tree/master/EditModule

Les dossiers « derive_x », « derive_y » et « gradient_z » sont à ajouter dans le dossier « manim\renderer\shaders ». Le fichier « opengl_mobject.py » est à remplacer dans le dossier « manim\mobject\opengl\ »

2 Illustration de la descente à une variable :

La classe DescenteGradient de IllustrationDescente.py, lien : [Code python de la Modélisation 2D](#)

est une représentation 2D de la descente de gradient. Voici la commande à entrer dans un terminal pour l'exécuter :

```
1 manim -qm -p --renderer=opengl IllustrationDescente.py DescenteGradient
```

L'animation va se lancer et dessiner une fonction ainsi qu'un point représentant le départ de la descente. Appuyer sur la barre d'espace permet alors de passer à l'itération suivante de la descente et la nouvelle coordonnée du point est affichée dans la console. Appuyer de nouveau sur la barre d'espace continuera de passer à l'itération suivante jusqu'à ce que la descente de gradient soit terminée (selon une valeur définie dans le code).

3 Illustration de la descente à deux variables :

La classe Descente3D de IllustrationDescente.py permet de visualiser la descente de gradient à 2 variables. Pour l'exécuter, voici la commande à entrer dans un terminal :

```
1 manim -qm -p --renderer=opengl IllustrationDescente.py Descente3D
```

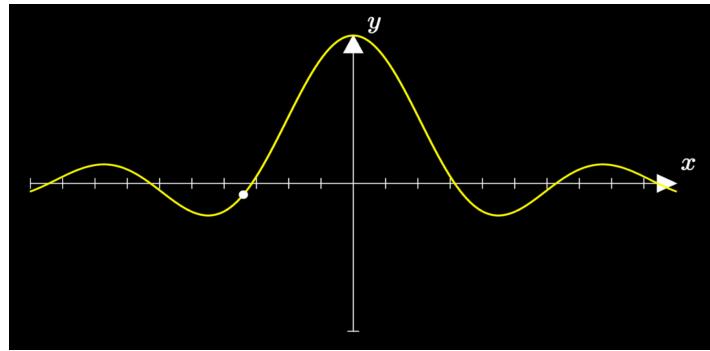


FIGURE V.2 – Exemple de lancement de la modélisation 2D

De la même manière que pour la descente 2D l'animation va se lancer et dessiner une surface représentant une fonction à deux variables ($\cos(x) \times \sin(x)$ par défaut) ainsi qu'un point représentant le départ de la descente. Appuyer sur espace permet également de passer à l'itération suivante de la descente et la touche tab permet de choisir le mode de coloration de la surface (gradient selon la hauteur, gradient selon la dérivée selon x et gradient selon la dérivée selon y). Il est également possible de se déplacer dans l'environnement 3D à l'aide de la souris (clic droit/gauche et glisser) ainsi que de zoomer à l'aide de la molette.

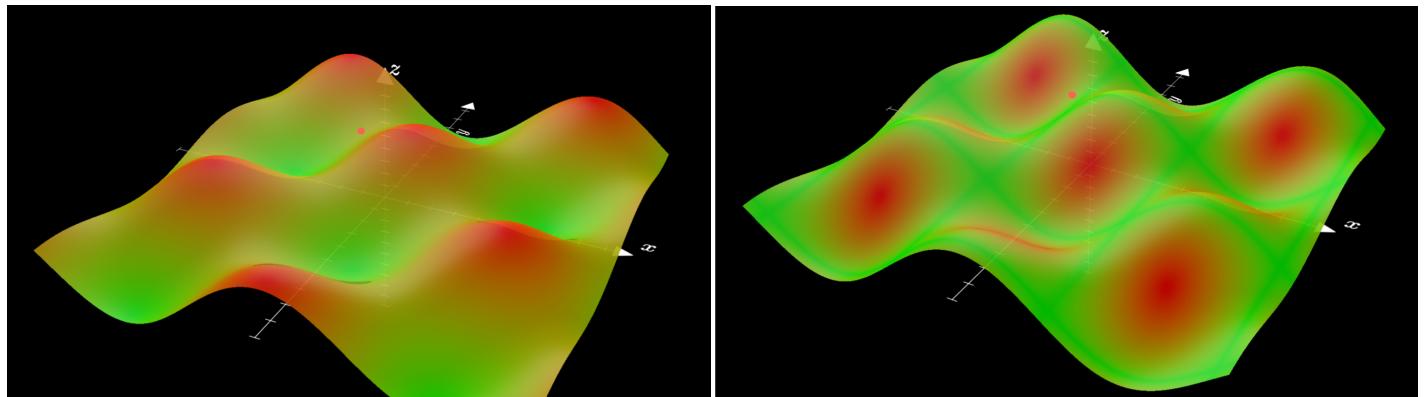


FIGURE V.3 – Gradient en fonction de la hauteur
(valeur de la fonction en (x, y))

FIGURE V.4 – Gradient en fonction de la dérivée par rapport à
 y

C Différentes méthodes de descente de gradient

1 Descente de gradient : limites de l'algorithme

L'un des principaux inconvénients de la descente de gradient est qu'elle ne permet de trouver que des minima locaux, sans garantir l'accès au minimum global. Une fois que l'algorithme a atteint un minimum local, il ne peut plus en sortir à moins que la taille du pas ne soit suffisamment grande pour franchir les barrières entourant ce minimum, souvent appelées « fossés ». En d'autres termes, si l'algorithme atteint une vallée sans issue, il y restera prisonnier à moins d'un ajustement approprié de la taille du pas. Le choix de la taille du pas (ou taux d'apprentissage) est donc crucial pour le bon fonctionnement de la descente de gradient. Si le pas est trop grand, l'algorithme risque de manquer des minima et d'osciller ou de diverger. En revanche, si le pas est trop petit, bien que la probabilité de convergence augmente, le nombre d'itérations nécessaires pour atteindre un minimum optimal devient très élevé. Ce problème est d'autant plus critique lorsque la fonction à minimiser possède un très grand nombre de variables (parfois des millions), ce qui alourdit considérablement le temps de calcul et les ressources informatiques nécessaires.

Un autre inconvénient majeur de la descente de gradient est qu'elle n'est efficace que pour les fonctions différentiables en tout point du domaine d'optimisation. Lorsqu'une fonction présente des discontinuités ou des points non dérivables, l'algorithme peut se retrouver bloqué ou incapable d'effectuer les mises à jour nécessaires. En effet, la formule de mise à jour repose sur le calcul du gradient, et si ce dernier n'est pas défini en un point donné, l'algorithme ne pourra pas progresser de manière conventionnelle. Pour pallier ces limites, plusieurs variantes de la descente de gradient ont été développées. Parmi elles, la descente de gradient stochastique, qui introduit un certain degré d'aléatoire pour explorer de nouvelles directions et éviter les pièges des minima locaux. Enfin, des techniques comme l'ajustement adaptatif du pas (par exemple, l'algorithme Adam) permettent d'améliorer la convergence en modulant dynamiquement la taille du pas en fonction du comportement du gradient. Ainsi, bien que la descente de gradient soit une méthode puissante et largement utilisée en optimisation, notamment en apprentissage automatique, elle nécessite une adaptation minutieuse pour surmonter ses limites et garantir une convergence efficace vers des solutions optimales.

2 Descente de gradient : Batch

3 La descente de gradient stochastique

Pour bien comprendre cette méthode, il est utile de clarifier ce que signifie le terme « stochastique ». Pour cela, nous définissons le mot stochastique. Stochastique est un synonyme d'aléatoire, en référence au hasard et s'oppose par définition au déterminisme. C'est un terme d'origine grecque qui signifie « basé sur la conjecture ». En français, il est couramment utilisé pour décrire des phénomènes aléatoires ou imprévisibles.

Lorsqu'on entraîne un modèle de machine learning sur un grand volume de données, le calcul exact du gradient de la fonction de perte peut être très long et coûteux en ressources. Une solution possible à ce problème est la descente de gradient stochastique. Le gradient stochastique est une estimation du vrai gradient, via un petit échantillon de données. La procédure de la descente de gradient stochastique est la suivante :

- Sélectionner un échantillon aléatoire de données.
- Estimer le gradient de la fonction de perte en utilisant cet échantillon.
- Mettre à jour les paramètres du modèle en fonction de ce gradient.

- Répéter ce processus jusqu'à convergence.

La succession de ces mises à jour permet une convergence plus rapide vers un optimum. Il est particulièrement utile lorsque les données sont redondantes, car il permet d'éviter des calculs inutiles. C'est pourquoi aujourd'hui les algorithmes de machine learning les plus performants utilisent la descente de gradient stochastique. Un autre avantage est que la descente de gradient stochastique est adaptée aux grands flux de données, ce qui signifie que l'on peut entraîner un modèle en ajoutant de nouvelles données sans devoir tout recommencer.

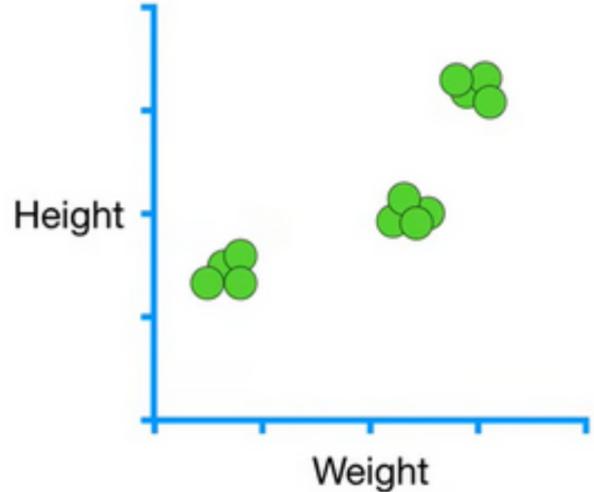


FIGURE V.5 – Poids en fonction de la taille [16]

Remarque : La définition stricte est qu'on ne doit utiliser qu'une donnée par pas. Cependant, il est commun d'utiliser deux ou trois (souvent beaucoup plus qu'une) données à la fois. En effet, le calcul de plus de données nous donne une meilleure estimation du vrai gradient plutôt qu'une seule. [16]

D Réseau de neurones

1 Perceptron monocouche et multicouche

Perceptron monocouche

Le terme monocouche signifie que ce type de perceptron ne possède qu'une unique couche de neurones. Un neurone dans un perceptron n'est rien d'autre qu'une opération mathématique : il prend plusieurs entrées (qu'elles soient des variables ou d'autres neurones), effectue un calcul sur celles-ci et produit une sortie. Dans le cas d'un perceptron monocouche, les entrées sont uniquement des variables car il n'y a qu'une unique couche. Le perceptron effectue principalement deux opérations sur ces entrées : une somme pondérée suivie de l'application d'une fonction d'activation.

- Calcul de la somme pondérée : Chaque entrée x_i est associée à un poids w_i et nous calculons la somme pondérée.
- Le neurone applique ensuite une fonction non linéaire appelée fonction d'activation.

Comme l'illustre l'image ci-dessus, il y a x_n entrées, respectivement toutes associées à un poids synaptique w_i . A cette somme, notée z , nous y ajoutons les biais, notés b . Enfin, le perceptron peut faire une prédiction, notée \hat{y} en appliquant la fonction d'activation, notée f .

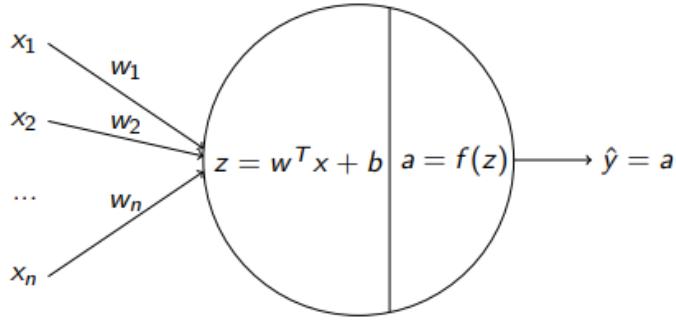


FIGURE V.6 – Perceptron monocouche [4]

Fonction d'activation

Définition : Fonction d'activation

Représente une fonction mathématique utilisée dans les réseaux de neurones pour introduire une non-linéarité dans le modèle, permettant ainsi au réseau d'apprendre et de modéliser des relations complexes dans les données. Ce sont des fonctions de seuillage. On peut ainsi le plus souvent les décomposer en 3 phases :

- Une partie non-activée (en dessous du seuil).
- Une phase de transition (à proximité du seuil).
- Une partie activée (au dessus du seuil).

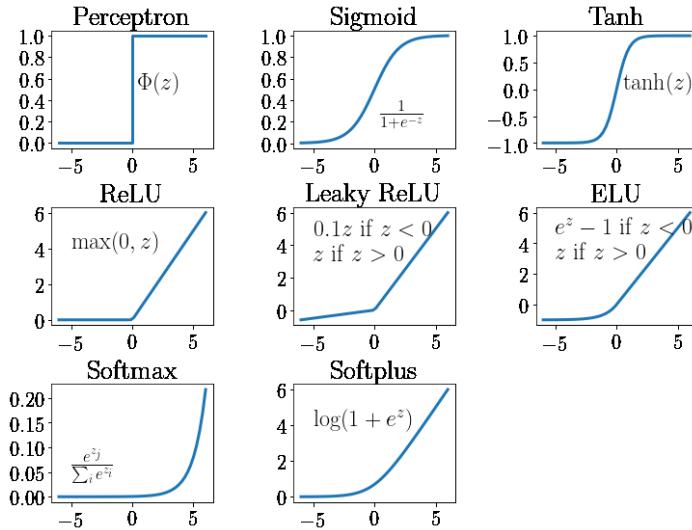


FIGURE V.7 – Différentes fonctions d'activations

Remarque : Les fonctions d'activation permettent à nos neurones de renvoyer une valeur. Leur choix dépend en autre de l'objectif du neurone.

Par convention [4] dans un réseau de neurones à n couches, nous utiliserons la fonction d'activation *ReLU* (ou une de ses dérivées) pour tous les neurones des couches $n - 1$.

En revanche, pour les neurones de la n -ième couche, un choix se pose [4] entre la fonction d'activation *Sigmoïde* et *Softmax*. Dans notre cas, pour notre réseau de neurones, nous partons sur du *Softmax*, utilisée plutôt dans la classification n-aire [4] (plus de deux classes) à la différence de *Sigmoïde* utilisée dans la classification binaire [4].

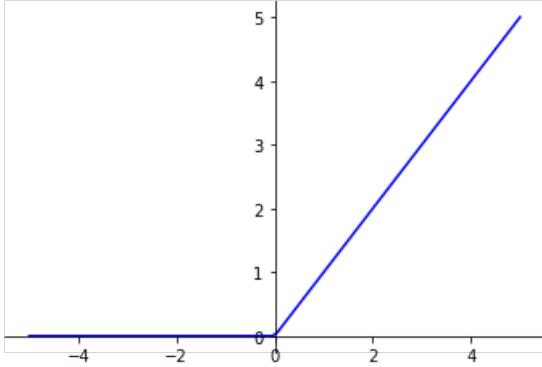


FIGURE V.8 – Fonction d'activation : *ReLU*

Caractéristique de *ReLU* :

$$\text{Équation : } f(x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$$

Renvoie des valeurs sur \mathbb{R}_+

$$\text{Dérivée faible : } f'(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$$

Remarque : Il est intéressant de prendre une fonction d'activation dont la dérivée n'est pas nulle afin de pouvoir appliquer l'algorithme de la descente de gradient

Remarque : Softmax renvoie un vecteur dont la somme de ses K composantes donne 1.

Elle suit l'expression suivante : $\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$ avec $j \in \{1, \dots, K\}$

Loss Function (fonction de perte ou fonction de coût)

Définition : Fonction de coût

Représente une fonction qui permet de mesurer l'erreur entre une valeur réelle connue donnée par l'utilisateur et une valeur renournée par le modèle.

Exemple : La présence d'un chat sur une image pouvant être représenté par un 1 et 0 sinon. Notre modèle, nous retourne alors une valeur présentée sous la même forme, avec la présence (1) ou non (0) d'un chat sur cette même image.

De la même manière que les fonctions d'activations, il existe un très grand nombre de fonctions de coût.

Il faut donc définir des critères pour justifier le choix d'une fonction particulière :

- Notre fonction doit être dérivable ou du moins présenter une dérivée discrète pour pouvoir appliquer la descente de gradient.
- Elle se doit d'être convexe, afin d'éviter la présence de minimum locaux.
- Notre fonction de coût compare les labels réels (valeurs renseignées par l'utilisateur) et les valeurs renvoyées par le modèle.
- Type de problème, exemple : régression, classification et etc.
- Le nombre de classes dans notre réseau de neurones est supérieur à 2

Dans le cas d'une régression, le choix de la fonction de coût se tourne vers l'Erreur Quadratique Moyenne qui s'exprime de la manière suivante : $J = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2$ pour m exemples, où $\hat{y}^{(i)}$ est la valeur prédite et $y^{(i)}$ est la valeur réelle pour le i -ème exemple.

Néanmoins pour notre cas, nous nous retrouvons en classification, ainsi de la même manière que pour la fonction d'activation avec *Sigmoïde* et *Softmax* un choix se pose [4] si nous sommes dans un modèle binaire ou non.

Dans le cas d'un modèle binaire, le choix de la fonction de coût se tourne vers l'Entropie Croisée Binaire (Log Loss) [4] qui s'exprime de la manière suivante : $J = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$ où $\hat{y}^{(i)}$ pour m exemples, où $\hat{y}^{(i)}$ est la valeur prédite et $y^{(i)}$ est la valeur réelle pour le i -ème exemple.

Remarque 1 : Du point de vue d'un récepteur, plus la source émet d'informations différentes, plus l'entropie (ou incertitude sur ce que la source émet) est grande.

Remarque 2 : Le log utilisé est le logarithme népérien.

En revanche, vu que notre réseau de neurones possède plus de 2 sorties, notre modèle n'est donc pas binaire. Ainsi, notre choix se retourne vers la fonction d'activation l'Entropie Croisée Multinomiale (cross-entropy) [4] qui s'exprime de la manière suivante : $J = -\frac{1}{m} \sum_{i=1}^m \sum_{j=1}^n y_j^{(i)} \log(\hat{y}_j^{(i)})$ pour m exemples, où $\hat{y}^{(i)}$ est la valeur prédictée et $y^{(i)}$ est la valeur réelle pour le i -ème exemple.

Nota bene :

La notion de convexité est maintenue dans le cas d'un seul neurone pour la fonction de coût. Néanmoins, dès que l'on est dans le cas d'un réseau de neurones, cette propriété est totalement perdue [4].

Justification du choix de Cross-Entropy en tant que fonction de coût :

On a $y_j^{(i)} \in \{0, 1\}$ représentant si oui (1) ou non (0) notre image fait partie de la catégorie recherchée.

Alors que $\hat{y}^{(i)} \in [0, 1]$ et dépend de la probabilité que cette image fasse partie de cette catégorie selon notre modèle.

Ainsi notre fonction de coût grâce au log nous renvoie une valeur comprise $[0, +\infty[$ grâce à la présence du $-\ln$ au début de son expression.

Remarque : $\hat{y}^{(i)}$ étant une valeur renvoyée par la fonction d'activation associée au neurone d'une classe de notre réseau, sa valeur peut être nulle. Néanmoins $\ln(0)$ n'étant pas définie, il est donc utile de sommer un $\epsilon \approx 10^{-5}$ [11] dans le \log .

Fonctionnement : Dans le cas de la classification d'image, notre fonction de coût nous renvoie une valeur qui doit diminuer à chaque itération de notre modèle afin de montrer la diminution du taux d'erreur de notre modèle.

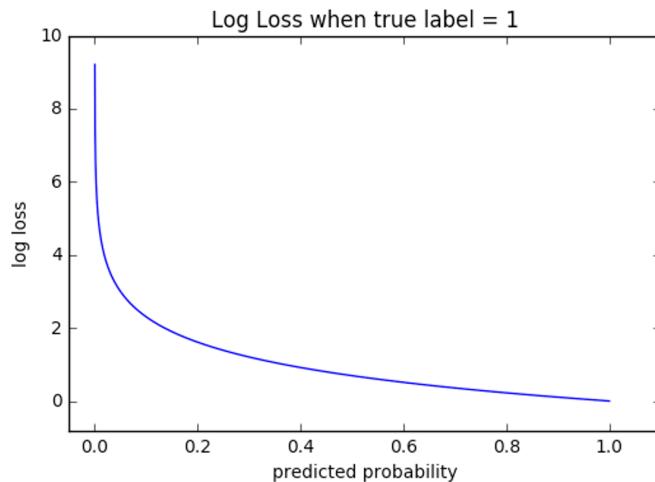


FIGURE V.9 – Valeurs renvoyées par la fonction de coût («Log Loss») à chaque itérations d'un modèle

Perceptron multicouche

L'association de plusieurs perceptrons monocouches à la suite donne un perceptron multicouche. Ces types de modèles sont utilisés pour la classification et la régression.

Un Multi Layer Perceptron est composé de trois types de couches :

- **La couche d'entrée** : reçoit les données brutes (exemple : images 32x32, vecteurs de caractéristiques...).
- **Les couches cachées** : effectuent des transformations sur les données grâce à des neurones et des fonctions d'activation.
- **La couche de sortie** : fournit le résultat final (ex : une probabilité pour chaque classe dans une classification).

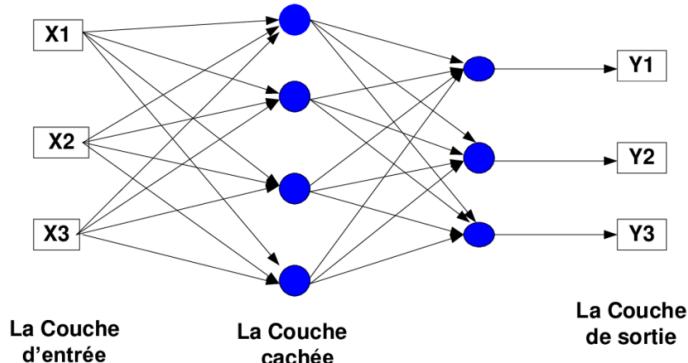


FIGURE V.10 – Réseau de neurones à 3 couches

Dans l'illustration ci-dessous, représentant ce concept, nous pouvons remarquer que chaque donnée qui sort d'un neurone (point bleu) est transférée à tous les autres neurones de la couche suivante. On dit que les couches sont entièrement connectées (fully connected).

2 Entraînement d'un réseau de neurones

Algorithme d'entraînement

L'entraînement d'un réseau de neurones consiste en la recherche des poids optimaux sur un ensemble d'exemples (entrées) connus. Comme définis précédemment, la fonction de coût nous permet de définir l'erreur du réseau pour une entrée donnée. Nous cherchons à obtenir une erreur la plus faible possible donc cela revient à appliquer l'algorithme de descente de gradient sur la fonction de coût dont les variables sont les poids.

Les réseaux de neurones peuvent rapidement devenir très complexes et les dérivées partielles de la fonction de coût par rapport à chaque poids ne sont pas facilement calculables, nous avons donc besoin d'un algorithme : la rétropropagation du gradient. Nous n'allons pas développer l'implémentation exacte de cet algorithme, il est juste nécessaire de savoir qu'il nous sert à obtenir les gradients selon chaque poids ce qui nous permet d'appliquer la descente de gradient.

Avec un pas bien choisi, le réseau converge vers un minimum de la fonction de coût correspondant à une possible solution au problème initial.

Descente de Gradient : mini-Batch

Vocabulaire :

Les epochs : Une epoch correspond à un passage complet de toutes les données d'entraînement à travers le réseau de neurones.

Un batch : Représente un groupe de nos données traitées en une seule fois pour calculer la mise à jour des poids du réseau. Lors de l'entraînement, les données ne sont pas envoyées une par une, mais en groupes (<> batches <>).

Les batches : Cela correspond au nombre d'échantillons traités en parallèle.

Validation split : Fraction de nos données d'entraînement pour faire la validation.

Remarque : Il est plus optimisé que le nombre de batch soit une puissance de deux car les GPU sont conçus pour gérer efficacement des blocs de mémoire alignés sur des puissances de deux. L'allocation de mémoire et les transferts de données entre la mémoire du GPU et ses coeurs de calcul sont plus efficaces.

Exemple de processus d'entraînement

Entraînement Pour entraîner notre modèle, on utilise 50 000 images.

Avec **validation_split = 0.2**, on a :

- Au total, 80% des images sont utilisées pour l'entraînement.
Cela représente 40 000 images utilisées pour ajuster les poids du réseau de neurones.
- Les 10 000 images restantes (20%) représentent les images qui servent à évaluer la performance de notre modèle.

Avec **batch_size = 64**, les 40 000 images d'entraînement sont divisées en petits groupes (batches) de 64 images.

Pour 40 000 images, on a donc $\frac{40000}{64} = 625$.

Il y a donc 625 lots (batches) de 64 images.

Au total, si on a 30 epochs, comme les poids du réseau de neurones sont mis à jour après chaque lot (batches), on a $30 \times 625 = 18750$ mises à jour.

Cela signifie que le réseau a vu passer au total sur les 30 epochs 18750 batches (lots de 64 images). Validation Après chaque epoch, c'est-à-dire après que le réseau ait fait passer une fois les 625 lots de 64 images (batches), le réseau évalue sa performance. Pour cela, il utilise les 10 000 images de validation (20% du lot d'entraînement de 50 000 images de départ). Le but est de voir comment le modèle réagit sur des images qu'il n'a jamais vues. On veut voir s'il arrive à s'adapter à ces images. On dit qu'il généralise.

Il est important de noter que les paramètres epochs, batch et validation split sont choisis de sorte à éviter au maximum le surapprentissage.

3 Réseau neuronal convolutif

Les réseaux de neurones convolutifs sont particulièrement adaptés au traitement des images, qui sont des tenseurs (matrices en 3 dimensions). Ils reprennent certains principes des perceptrons multicouches, tout en présentant une architecture différente.

Comme les perceptrons multicouches, les réseaux convolutifs sont composés de plusieurs couches empilées. Cependant, contrairement aux MLP (où toutes les couches sont généralement du même type, c'est-à-dire entièrement connectées), les réseaux convolutifs intègrent différents types de couches — telles que les couches de convolution, de pooling ou encore de normalisation — chacune ayant un rôle spécifique dans le traitement de l'information. Analysons-les :

Couches de convolutions :

Les couches de convolutions détiennent différents filtres de convolution, chacun correspondant à un neurone dans cette couche. Un filtre de convolution est un petit tenseur carré de taille impaire, le plus souvent de taille $3 \times 3 \times d$ ou $5 \times 5 \times d$, où d est un nombre égal au nombre de canaux de l'image d'entrée. Initialement, une image détient 3 canaux, représentant les trois couleurs principales (rouge, bleu et vert). Nous verrons que le nombre de canaux augmentera après le passage dans une couche de convolution.

Le filtre glisse sur tous les pixels de l'image d'entrée pour en obtenir une nouvelle image. À chaque position, un produit élément par élément est effectué entre les valeurs du filtre et celles de la région correspondante de l'image. Le résultat est la somme pondérée de cette région, ce qui permet d'extraire une information locale.

1	1	0	1
0	0	0	1
1	1	1	0
1	0	0	1
1	0	0	1

Image

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

Filtre 3×3
 $f = 3$

1	1
1	1

Réponse

1	1	0	1
0	0	0	1
1	1	1	0
1	0	0	1
1	0	0	1

Image

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

Filtre 3×3
 $f = 3$

1	1
1	1

Réponse

1	1	0	1
0	0	0	1
1	1	1	0
1	0	0	1
1	0	0	1

Image

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

Filtre 3×3
 $f = 3$

1	1
1	0

Réponse

FIGURE V.11 – Illustration d'un filtre convolutif [3].

Les poids du filtre sont des paramètres appris pendant l'entraînement du réseau. Un filtre correspond donc à un motif qu'il apprend à détecter (bord, texture, motif complexe, etc.)

Cependant, nous pouvons observer que les dimensions du tenseur diminuent, après le passage du filtre. C'est un problème car nous perdons de l'information. On résout donc celui-ci en utilisant du padding ou mirror padding [3].

Imaginons qu'une image d'entrée de taille $32 \times 32 \times 3$, soit donnée à une couche de convolution de 10 neurones. Chaque neurone est associé à un filtre $3 \times 3 \times 3$, par exemple, qui génère une carte de taille $32 \times 32 \times 1$. En empilant ces 10 cartes, on obtient un tenseur de sortie de taille $32 \times 32 \times 10$, qui sera l'entrée de la couche suivante.

Les couches de convolution sont souvent suivies soit d'une autre couche de convolution soit d'une couche de pooling

Couches de pooling :

Comme nous l'avons vu, les couches de convolution entraînent une augmentation de la profondeur des tenseurs d'une couche à une autre. Pour contrer cette croissance, les couches de pooling sont introduites à la suite d'une ou plusieurs couches de convolutions. Elles agissent uniquement sur la hauteur et la largeur des tenseurs, sans en modifier leur profondeur.

Plusieurs méthodes sont possibles pour diminuer les dimensions des tenseurs, comme le stride, le max-pooling ou l'average-pooling, cependant, dans la pratique, le max-pooling est de loin la plus utilisée. Ainsi, nous nous concentrerons uniquement sur cette technique. Pour davantage de détails sur les autres méthodes, nous renvoyons le lecteur à la bibliographie [3].

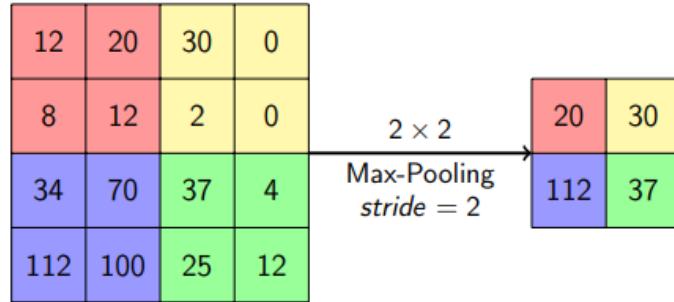


FIGURE V.12 – Max-pooling d'un tenseur [3].

Comme l'indique l'image ci-dessus, pour chaque canal du tenseur d'entrée, celui-ci est divisé en sous-parties. Dans le cas du max-pooling, seule la valeur maximale de chaque région est conservée.

Cette méthode est généralement plus efficace que de l'average-pooling. En effet, les filtres de convolution sont conçus pour détecter des motifs précis ; une valeur élevée signifie que le motif est fortement présent. Il est donc plus pertinent de retenir cette valeur maximale plutôt qu'une moyenne qui pourrait diluer cette information.

Le pooling permet donc de réduire la taille des tenseurs, ce qui diminue le nombre de paramètres à transmettre à la couche suivante, tout en préservant les caractéristiques essentielles de l'image.

Enfin, les réseaux convolutifs possèdent aussi des couches complètement connectées.

Maintenant, nous pouvons étudier l'architecture des réseaux convolutifs.

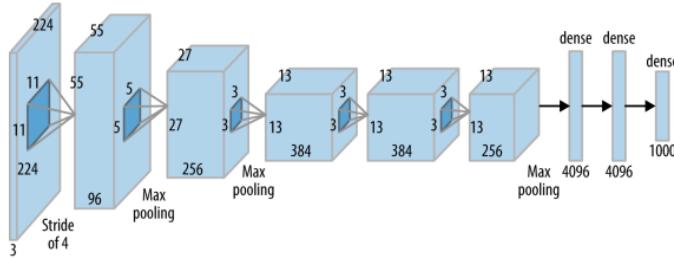


FIGURE V.13 – Structure simplifiée des réseaux convolutifs, illustré par l'exemple d'AlexNet [3].

Comme nous pouvons le voir, avec cette image ci-dessus, les réseaux convolutifs se présentent sous la forme d'une succession d'une ou plusieurs couches convolutives suivies d'une couche de pooling. Cette étape est répétée un nombre de fois assez important pour diminuer au maximum la taille du tenseur. Enfin, quand la taille est convenable, le tenseur est transmis à une ou plusieurs couches connectées à la suite, qui permettront l'identification et la prédiction de la donnée.

E Reconnaissance d'images

1 Chiffres

Après avoir testé notre modèle de réseau neuronal sur de simples régressions linéaires, nous avons tenté de l'appliquer à la reconnaissance de chiffres.

Nous avons pour cela une banque de données de 60000 images de taille 28 x 28 pixels en noir et blanc (base de données du MNIST [15]). Cela nous donne 784 entrées pour le réseau ainsi que 10 sorties (une pour chaque chiffre).

Après avoir testé différentes tailles de réseau et de facteurs d'apprentissage, nous avons atteint une précision de 90% sur la reconnaissance des chiffres. Ce résultat a été obtenu avec un réseau possédant une couche de 784 neurones utilisant la fonction d'activation ReLU suivie d'une couche de 10 neurones utilisant la softmax pour les sorties. Ce nombre élevé de neurones, bien que permettant une bonne représentation des données, peut être source de sur-apprentissage et un nombre de neurones bien plus faible s'est plus tard révélé comme suffisant.

Exemple de sorties du réseau :

```

1 label : 0
2 sortie :[0.99999727434567, 1.466499199386942e-30, 3.3001202705952095e-29, 5.1751263610308825e-23, 3.362926674116666e-27, 2.725654325223024e
           -07, 3.583870493298317e-16, 1.887090216942274e-33, 5.612611375939689e-19, 8.665046013717515e-30]
3
4 label : 7
5 sortie : [5.379913461140272e-20, 7.49042477451807e-17, 9.524447863881987e-07, 6.151502789864992e-18, 3.05280588464305e-20, 1.7680241728270652e
           -18, 1.0912478140662355e-28, 0.9999987988552603, 2.4869823276411794e-07, 1.7205067672722502e-12]
```

Nous pouvons remarquer que le réseau affiche une quasi certitude (0.99999...) de la valeur qu'il observe. Un résultat aussi clivant n'est pas souhaitable car dans certains cas douteux, le réseau sera certain de reconnaître un chiffre qui en est peut-être un autre.

Bien que le problème soit plus simple par la taille des entrées que la reconnaissance de bouteilles, cela nous a permis de prouver la réalisabilité de notre projet ainsi que d'avoir un ordre de grandeur pour les différents hyperparamètres du réseau.

2 Images diverses

Ce programme résout un problème de classification d'images à 10 classes (avions, voitures, oiseaux, chats, cerfs, chiens, grenouilles, chevaux, navires et camions).

Nous l'avons utilisé pour comparer les modèles de convolution à ceux des perceptrons multicouches.

Nous avons pu conclure que les réseaux de convolution sont plus efficaces, mais demandent beaucoup plus d'énergie.

Importation des modules

```

1 #importer les bibliothèques nécessaires
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 import tensorflow as tf
6 from tensorflow.keras.datasets import cifar10 #Base de données
7
8 from tensorflow import keras
9 from tensorflow.keras import layers
10
11 from tensorflow.keras.callbacks import EarlyStopping
12
13 import tensorflow as tf
14 from carbontracker.tracker import CarbonTracker
```

Remarque : On importe CarbonTracker pour évaluer le coût de l'entraînement du réseau de neurones. On importe TensorFlow, numpy et matplotlib pour le bon déroulé du programme

Chargement des données de tests

Exemple d'utilisation :

```
1 num_classes = 10
2 y_train_mlp = tf.keras.utils.to_categorical(y_train, num_classes)
3 y_test_mlp = tf.keras.utils.to_categorical(y_test, num_classes)
4
5 # Pareil
6 y_train_cnn = tf.keras.utils.to_categorical(y_train, num_classes)
7 y_test_cnn = tf.keras.utils.to_categorical(y_test, num_classes)
```

Ces deux lignes transforment nos étiquettes de classes, qui sont des entiers (de 0 à 10, chaque entier correspondant à une catégorie d'objet du dataset cifar10) en vecteurs one-hot encodés.

Exemple : $y_{train} = [5, 0, 3, 9, \dots]$ passe sous la forme :

$[0, 0, 0, 0, 0, 1, 0, 0, 0, 0] \leftarrow$ pour le chiffre 5

$[1, 0, 0, 0, 0, 0, 0, 0, 0, 0] \leftarrow$ pour le chiffre 0

$[0, 0, 0, 1, 0, 0, 0, 0, 0, 0] \leftarrow$ pour le chiffre 3

y_{train_mlp} contient donc une matrice de tous ses vecteurs one-hot encodés.

On fait cela car notre modèle va nous sortir un résultat de la forme $[0.1, 0.2, \dots, 0.9]$. De cette manière on pourra le comparer à un vecteur one-hot et réussir à calculer sa fonction de perte.

Affichage des 20 premières images

```
1 #Affiche les 20 premières images
2 CLASSES = np.array([
3     "avion",
4     "voiture",
5     "oiseau",
6     "chat",
7     "cerf",
8     "chien",
9     "grenouille",
10    "cheval",
11    "bateau",
12    "camion",
13])
14 # Afficher les 20 premières images
15 plt.figure(figsize=(12, 10)) # Augmenter la taille pour mieux voir les images
16 # le deuxième paramètre augmente la longitude entre 2 images
17 # le premier paramètre augmente la latitude entre 2 images
18 for i in range(20):
19     plt.subplot(5, 4, i + 1) # 4 lignes, 5 colonnes
20     plt.imshow(x_train[i])
21     plt.title(CLASSES[y_train[i][0]], fontsize=10) # Réduire la taille du titre si besoin
22     plt.axis('off')
23 plt.show()
```

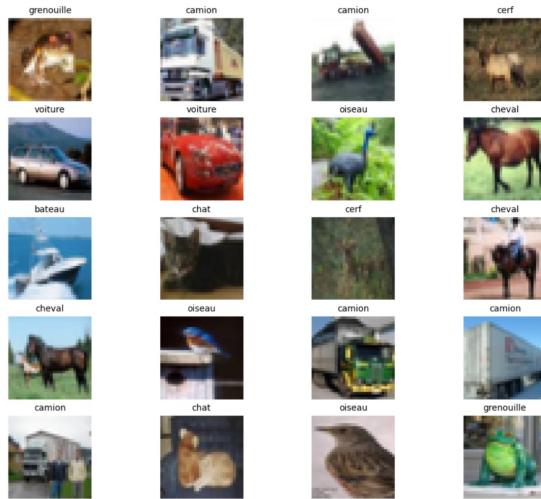


FIGURE V.14 – Résultat du programme

Affichage de l'équipartition des données

```

1 #Afficher la distribution des classes dans l'ensemble d'entraînement (équirépartition des données)
2
3 class_distribution = [0] * 10
4 for label in y_train:
5     class_distribution[label[0]] += 1
6
7 plt.figure(figsize=(10, 4))
8 plt.bar(CLASSES, class_distribution)
9 plt.title("Distribution des classes dans l'ensemble d'entraînement")
10 plt.xlabel("Classes")
11 plt.ylabel("Nombre d'images")
12 plt.xticks(rotation=45)
13 plt.show()

```

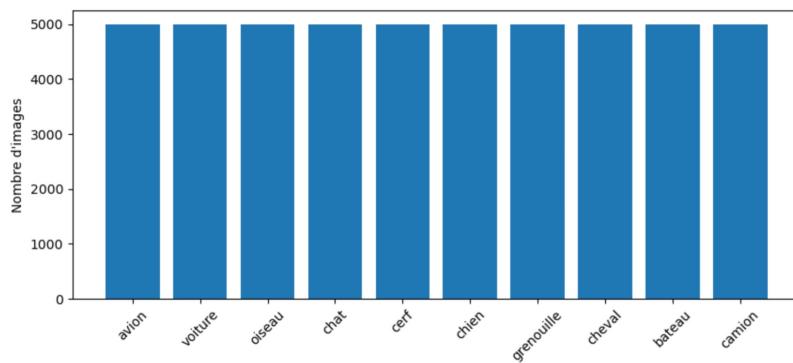


FIGURE V.15 – Distribution des classes dans l'ensemble d'entraînement

Normalisation des pixels

```

1 # Normaliser les pixels des images en mettant à l'échelle entre 0 et 1
2 x_train = x_train.astype('float32') / 255.0
3 x_test = x_test.astype('float32') / 255.0

```

Les images du dataset CIFAR10 sont des images en couleurs, où chaque pixel est un entier entre 0 et 255, « astype(float32) » convertit nos entiers en floats pour faire par la suite des calculs plus précis avec TensorFlow. On divise par 255 pour mettre à l'échelle entre 0 et 1.

Cela aide le réseau de neurones à apprendre plus efficacement, car :

- Les valeurs sont sur une même échelle cohérente (0.0 à 1.0).
- Cela évite des problèmes de gradients trop grands ou trop petits.

Affichage de premiers résultats

```

1 print("Forme de l'ensemble d'entraînement (images) :", x_train.shape)
2 print("Forme de l'ensemble de test (images) :", x_test.shape)
3 print("Forme de l'ensemble d'entraînement (étiquettes) :", y_train_mlp.shape)
4 print("Forme de l'ensemble de test (étiquettes) :", y_test_mlp.shape)

```

```

Forme de l'ensemble d'entraînement (images) : (50000, 32, 32, 3)
Forme de l'ensemble de test (images) : (10000, 32, 32, 3)
Forme de l'ensemble d'entraînement (étiquettes) : (50000, 10)
Forme de l'ensemble de test (étiquettes) : (10000, 10)

```

FIGURE V.16 – Réponse du programme.

3 Définition des modèles MLP et CNN

Modèles MLP :

```

1 #Définir le modèle MLP pour la classification d'images
2 model_mlp = keras.Sequential([
3     # Transforme une image 32x32 avec 3 canaux de couleurs (RVB) en un vecteur 1D.
4     layers.Flatten(input_shape=(32, 32, 3)),#3: RVB (#première couche ??)
5
6     # Ajouter une couche cachée de 128 neurones avec une fonction d'activation ReLU (rajoute non linearité)
7     layers.Dense(128, activation='relu'),
8
9
10    # Ajouter la couche de sortie avec 10 neurones (classes CIFAR-10) et une fonction d'activation softmax (très utilisé pour multiclass)
11    layers.Dense(10, activation='softmax') # 10 pour les 10 catégories qu'on souhaite prédire
12 ])
13
14 model_mlp = keras.Sequential([
15     layers.Flatten(input_shape=(32, 32, 3)), # Transforme une image 32x32 avec 3 canaux de couleurs (RVB) en un vecteur 1D
16     layers.Dense(128, activation='relu'), # Ajouter une couche cachée de 128 neurones avec une fonction d'activation ReLU (rajoute non
17         linearité)
18     layers.Dense(10, activation='softmax') # Ajouter la couche de sortie avec 10 neurones (pour les 10 catégories qu'on souhaite prédire) et
19         une fonction d'activation softmax
20 ])
21
22 model_mlp2 = keras.Sequential([
23     layers.Flatten(input_shape=(32, 32, 3)),
24     layers.Dense(256, activation='relu'), # Ajouter une couche cachée de 256 neurones avec une fonction d'activation ReLU (rajoute non
25         linearité)
26     layers.Dense(128, activation='relu'),
27     layers.Dense(10, activation='softmax')
28 ])

```

Dans ces lignes, nous définissons deux modèles MLP distincts pour la classification d'image.

Les deux modèles commencent par une couche Flatten, qui convertit chaque image d'entrée (32×32 pixels avec 3 canaux de couleur RVB) en un vecteur 1D. Cette transformation est nécessaire car les couches suivantes de type Dense attendent des entrées sous forme de vecteurs.

La première couche dense contient 128 neurones, entièrement connectés aux entrées. Elle utilise la fonction d'activation ReLU, qui introduit de la non-linéarité et permet au modèle d'apprendre des représentations plus complexes.

Enfin, la dernière couche est une couche de sortie Dense composée de 10 neurones, correspondant aux 10 classes de CIFAR-10. Chaque neurone produit une probabilité grâce à l'activation softmax, et la prédiction finale correspond à la classe ayant la probabilité la plus élevée.

La différence entre les deux modèles réside dans leur profondeur :

- Le premier modèle contient une seule couche cachée avec 128 neurones avant la couche de sortie.
- Le second modèle est plus complexe : il ajoute une première couche Dense de 256 neurones, suivie d'une deuxième couche de 128 neurones, avant la couche de sortie.

De cette façon, on observe que le second modèle a un meilleur taux de précision car il capture mieux les relations complexes.

Modèle CNN :

```
1 model_cnn = keras.Sequential()  
2 model_cnn.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))  
3 model_cnn.add(layers.MaxPooling2D(pool_size=(2, 2)))  
4 model_cnn.add(layers.Conv2D(64, (3, 3), activation='relu'))  
5 model_cnn.add(layers.MaxPooling2D(pool_size=(2, 2)))  
6 model_cnn.add(layers.Flatten())  
7 model_cnn.add(layers.Dense(128, activation='relu'))  
8 model_cnn.add(layers.Dropout(0.5))  
9 model_cnn.add(layers.Dense(10, activation='softmax'))
```

Le modèle est construit avec plusieurs types de couches qui permettent d'extraire progressivement des caractéristiques importantes des images avant de les classifier.

La première couche du réseau est une couche de convolution.

```
1 model_cnn.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
```

Cette couche applique 32 filtres de convolution de taille (3,3), ce qui permet d'extraire des motifs simples. On peut extraire par exemple les bords ou les textures de l'image. L'activation ReLU est utilisée pour ajouter de la non-linéarité. Ici, on apprend des représentations complexes.

Après cette opération, l'image conserve sa taille de 32×32 pixels, mais elle est maintenant représentée par 32 cartes de caractéristiques qui contiennent des informations sur les motifs détectés. Ensuite, une couche de pooling est appliquée.

```
1 (model_cnn.add(layers.MaxPooling2D(pool_size=(2, 2))))
```

Cette opération réduit la taille de l'image de moitié en ne conservant que les valeurs maximales dans chaque région 2×2 . Cela permet de réduire la quantité de données à traiter tout en conservant les informations importantes.

Une deuxième couche de convolution est ajoutée :

```
1 (model_cnn.add(layers.Conv2D(64, (3, 3), activation='relu')))
```

Ici, 64 nouveaux filtres sont appliqués pour détecter des caractéristiques plus complexes, comme des formes ou des motifs plus avancés.

Ensuite, une nouvelle couche de pooling est utilisée pour réduire encore la taille des données et ne conserver que les éléments essentiels.

```
1 model_cnn.add(layers.MaxPooling2D(pool_size=(2, 2)))
```

Cette opération réduit la taille de l'image de moitié et on ne conserve que les valeurs maximales dans chaque région 2×2 . Cela permet de réduire la quantité de données à traiter tout en conservant les informations importantes.

Une fois l'extraction des caractéristiques terminée, les données passent par une couche. Cette couche transforme la matrice de caractéristiques en un vecteur en une dimension, afin de pouvoir l'envoyer dans une couche dense classique.

```
1 (model_cnn.add(layers.Dense(128, activation='relu')))
```

Cette couche contient 128 neurones et utilise une activation ReLU. Elle permet de combiner les caractéristiques extraites par les couches précédentes pour aider à la classification.

```
1 model_cnn.add(layers.Dropout(0.5))
```

Une couche de Dropout est appliquée juste après. Cela empêche le réseau d'apprendre trop précisément les données d'entraînement (phénomène de sur-apprentissage) et améliore sa capacité à généraliser à de nouvelles images.

```
1 model_cnn.add(layers.Dense(10, activation='softmax'))
```

Elle contient 10 neurones, car il y a 10 classes dans CIFAR-10. L'activation softmax permet d'obtenir des probabilités associées à chaque classe, afin de prédire la catégorie de l'image.

Compilation des modèles et affiche le résumé

```
1 #COMPILEMENT DES MODLES
2 model_mlp.compile(optimizer='adam',
3                     loss='categorical_crossentropy',
4                     metrics=['accuracy'])
5
6 model_mlp2.compile(optimizer='adam',
7                     loss='categorical_crossentropy',
8                     metrics=['accuracy'])
9
10 model_cnn.compile(optimizer='adam',
11                     loss='categorical_crossentropy',
12                     metrics=['accuracy'])
```

On a utilisé l'optimiseur adam et la fonction de perte categorical crossentropy car dans notre cas il s'agit de problème de classification à 10 classes de sorties.

```
1 # AFFICHE LE RÉSUMÉ DES MODLES
2 model_mlp.summary()
3 model_mlp2.summary()
4 model_cnn.summary()
```

Layer (type)	Output Shape	Param #
flatten_2 (Flatten)	(None, 3072)	0
dense_4 (Dense)	(None, 128)	393,344
dense_5 (Dense)	(None, 10)	1,290

FIGURE V.17 – Modèle perceptron multicouche 1

Layer (type)	Output Shape	Param #
flatten_3 (Flatten)	(None, 3072)	0
dense_6 (Dense)	(None, 256)	786,688
dense_7 (Dense)	(None, 128)	32,896
dense_8 (Dense)	(None, 10)	1,290

FIGURE V.18 – Modèle perceptron multicouche 2

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
flatten_4 (Flatten)	(None, 2304)	0
dense_9 (Dense)	(None, 128)	295,040
dropout (Dropout)	(None, 128)	0
dense_10 (Dense)	(None, 10)	1,290

FIGURE V.19 – Modèle de convolution

« Output Shape » correspond à la forme (ou aux dimensions) des données en sortie de chaque couche du modèle. « Param » correspond à la somme de tous nos poids et de nos biais.

Implémentation de l'early stopping et des hyper paramètres

Pour rappel, l'early stopping nous permet d'éviter le surapprentissage.

```
1 early_stopping = EarlyStopping(monitor='val_loss', patience=8, restore_best_weights=True)
```

Le but est d'optimiser le nombre d'epochs en surveillant la perte de validation. Après 8 epochs sans amélioration, le modèle arrête sa période d'entraînement et récupère ensuite les meilleurs poids.

```
1 # Utilisation de la validation split (20% des données pour la validation)
2 epochs, batch_size, validation_split= 50 , 256 ,0.2
```

On peut se permettre un nombre élevé de batch size car notre entraînement se fait sur 40 000 images. Le nombre d'epochs est défini à 50 mais en pratique il sera plus petit dû à l'early stopping.

Définition des hyperparamètres et des trackers de carbone

```
1 # Création d'un tracker distinct pour chaque modèle
2 tracker_mlp = CarbonTracker(epochs=epochs, monitor_epochs=1, verbose=2, log_dir="logs_carbontracker/mlp/")
3 tracker_mlp2 = CarbonTracker(epochs=epochs, monitor_epochs=1, verbose=2, log_dir="logs_carbontracker/mlp2/")
4 tracker_cnn = CarbonTracker(epochs=epochs, monitor_epochs=1, verbose=2, log_dir="logs_carbontracker/cnn/")
```

Chaque modèle aura un tracker de carbone associé. On prévoit qu'ils relèvent des mesures sur au maximum 50 epochs.

Entraînement des 3 modèles

```
1 # Suivi de la consommation pour chaque modèle
2 tracker_mlp.epoch_start()
3 historique = model_mlp.fit(x_train, y_train_mlp, epochs=epochs, batch_size=batch_size, validation_split=validation_split, callbacks=[early_stopping])
4 tracker_mlp.epoch_end()

5
6 tracker_mlp2.epoch_start()
7 historique2 = model_mlp2.fit(x_train, y_train_mlp, epochs=epochs, batch_size=batch_size, validation_split=validation_split, callbacks=[early_stopping])
8 tracker_mlp2.epoch_end()

9
10 tracker_cnn.epoch_start()
11 history_cnn = model_cnn.fit(x_train, y_train_cnn, epochs=epochs, batch_size=batch_size, validation_split=0.2, shuffle=True, callbacks=[early_stopping])
12 tracker_cnn.epoch_end()
```

Chaque modèle est associé à un tracker de carbone.

Chaque modèle se stoppera en cas de début de surapprentissage.

On peut donc maintenant analyser quelques premiers résultats.

```
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to average carbon intensity 56.03859 gCO2/kWh
```

FIGURE V.20 – Test CarbonTracker

Code Carbon tracker est une bibliothèque qui calcule combien d'électricité (et donc de CO₂) a été utilisée pendant l'entraînement d'un modèle.

Elle sert donc à **mesurer l'impact écologique** des modèles d'intelligence artificielle. La mesure de référence est fixée à la **valeur moyenne par défaut** : 56.04 g de CO₂ par kWh (pour la France en 2023).

Le nombre 56 g CO₂/kWh signifie que pour chaque **kilowattheure (kWh)** d'électricité consommée, il est **émis environ** 56 g de **dioxyde de carbone (CO₂)** dans l'atmosphère.

Résultat du premier modèle (réseau perceptron multicouche à deux couches) :

Dû à l'early stopping, l'entraînement s'est arrêté au bout de 34 epochs.

```
Epoch 34/50
157/157 ━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.4850 - loss: 1.4511 - val_accuracy: 0.4395 - val_loss: 1.5940
```

FIGURE V.21 – Phase d'entraînement

Sur 50 epochs, la prédiction pour ce modèle par le module CodeCarbon était la suivante :

```
Predicted consumption for 50 epoch(s):
    Time: 0:27:59
    Energy: 0.021069611065 kWh
    CO2eq: 1.180711295919 g
    This is equivalent to:
    0.010983360892 km travelled by car
```

FIGURE V.22 – Prédiction pour 50 epochs

En réalité, la consommation pour une epoch est la suivante :

```
Actual consumption for 1 epoch(s):
    Time: 0:00:34
    Energy: 0.000421392221 kWh
    CO2eq: 0.023614225918 g
    This is equivalent to:
    0.000219667218 km travelled by car
CarbonTracker: Finished monitoring.
```

FIGURE V.23 – Prédiction pour 1 epoch

Donc en multipliant par 34 nos résultats, on a que le modèle a consommé :

- Energy : 0.014327335514 kWh
- CO₂eq : 0.802883681212 g

Ce qui correspond à environ 0.007468685411999999 km soit sept mètres parcourus par une voiture.

Résultat du deuxième modèle (réseau perceptron multicouche à trois couches) :

```
Epoch 1/50
157/157      3s 12ms/step - accuracy: 0.2293 - loss: 2.1233 - val_accuracy: 0.3376 - val_loss: 1.8646
Epoch 2/50
157/157      2s 10ms/step - accuracy: 0.3631 - loss: 1.7978 - val_accuracy: 0.3561 - val_loss: 1.8122
Epoch 3/50
157/157      2s 10ms/step - accuracy: 0.3878 - loss: 1.7100 - val_accuracy: 0.3926 - val_loss: 1.7239
Epoch 4/50
157/157      2s 10ms/step - accuracy: 0.4150 - loss: 1.6550 - val_accuracy: 0.3973 - val_loss: 1.6992
Epoch 5/50
157/157      2s 10ms/step - accuracy: 0.4263 - loss: 1.6035 - val_accuracy: 0.4245 - val_loss: 1.6107
Epoch 6/50
157/157      2s 11ms/step - accuracy: 0.4518 - loss: 1.5468 - val_accuracy: 0.4237 - val_loss: 1.6147
Epoch 7/50
157/157      2s 10ms/step - accuracy: 0.4644 - loss: 1.5147 - val_accuracy: 0.4394 - val_loss: 1.5870
Epoch 8/50
157/157      2s 10ms/step - accuracy: 0.4702 - loss: 1.4955 - val_accuracy: 0.4550 - val_loss: 1.5410
Epoch 9/50
157/157      2s 11ms/step - accuracy: 0.4764 - loss: 1.4717 - val_accuracy: 0.4485 - val_loss: 1.5526
Epoch 10/50
157/157      2s 10ms/step - accuracy: 0.4849 - loss: 1.4510 - val_accuracy: 0.4717 - val_loss: 1.5026
Epoch 11/50
157/157      2s 10ms/step - accuracy: 0.4849 - loss: 1.4510 - val_accuracy: 0.4717 - val_loss: 1.5026
Epoch 12/50
157/157      2s 10ms/step - accuracy: 0.4977 - loss: 1.4208 - val_accuracy: 0.4692 - val_loss: 1.5088
Epoch 13/50
157/157      2s 10ms/step - accuracy: 0.4966 - loss: 1.4120 - val_accuracy: 0.4672 - val_loss: 1.5036
Epoch 14/50
157/157      2s 10ms/step - accuracy: 0.5025 - loss: 1.4043 - val_accuracy: 0.4650 - val_loss: 1.5044
Epoch 15/50
157/157      2s 10ms/step - accuracy: 0.5014 - loss: 1.3909 - val_accuracy: 0.4528 - val_loss: 1.5630
157/157      2s 10ms/step - accuracy: 0.5080 - loss: 1.3715 - val_accuracy: 0.4526 - val_loss: 1.5544
```

FIGURE V.24 – Résultat de l'entraînement

Nous pouvons voir ici toutes les différentes epochs et leurs résultats. Dû à l'early stopping, le modèle s'est arrêté au bout de 15 epochs.

Sur 50 epochs, la prédiction pour ce modèle par le module codecarbon était la suivante :

```

Predicted consumption for 50 epoch(s):
    Time: 0:21:45
    Energy: 0.017116845315 kWh
    CO2eq: 0.959203876702 g
    This is equivalent to:
    0.008922826760 km travelled by car

```

FIGURE V.25 – Consomation 50 epochs

En réalité, la consommation pour une epoch est la suivante :

```

Actual consumption for 1 epoch(s):
    Time: 0:00:26
    Energy: 0.000342336906 kWh
    CO2eq: 0.019184077534 g
    This is equivalent to:
    0.000178456535 km travelled by car

```

FIGURE V.26 – Consomation 1 epoch

Donc en multipliant par 15, on obtient que l'entraînement a nécessité un coût énergétique de $0.00513505359 \text{ kWh}$, soit une production de $0.28776116301000004 \text{ g}$ de CO_2 ce qui représente $0.002676848025 \text{ km}$ (2 m).

Résultat du troisième modèle (réseau de convolution) :

FIGURE V.27 – Phase d'entraînement troisième modèle

Dû à l'early stopping, l'entraînement s'est arrêté au bout de 34 epochs. Sur 50 epochs, la prédiction pour ce modèle par le module CodeCarbon était la suivante :

```

Predicted consumption for 50 epoch(s):
    Time: 5:17:42
    Energy: 0.284197020023 kWh
    CO2eq: 15.926000284274 g
    This is equivalent to:
    0.148148839854 km travelled by car

```

FIGURE V.28 – Consomation 50 epochs troisième model

En réalité, la consommation pour une epoch est la suivante :

```
Actual consumption for 1 epoch(s):
Time: 0:06:21
Energy: 0.005683940400 kWh
CO2eq: 0.318520005685 g
This is equivalent to:
0.002962976797 km travelled by car
```

FIGURE V.29 – Consomation 1 epoch troisième model.png

Donc en multipliant par 34, on obtient que l'entraînement a nécessité un coût énergétique de $0.19325397360000002 \text{ kWh}$, soit une production de 10.82968019329 g de CO_2 ce qui représente $0.100741211098 \text{ km}$ (100 m).

En termes de bilan énergétique, le réseau convolutif consomme beaucoup plus. Cela s'explique par le fait qu'il fasse plus de calculs. On verra par la suite que c'est aussi ce modèle qui fournira les meilleurs résultats.

Au total, sur seulement un entraînement, c'est équivalent à parcourir environ 100 m en voiture.

Analyse des résultats :

Pour rappel, on entraîne notre modèle sur 40000 images à chaque epoch. On définit la perte d'entraînement qui mesure l'erreur entre les prédictions du modèle et les vraies classes. Notre objectif est donc de la diminuer au maximum.

Par ailleurs, on a mesuré la perte de validation. Cette fonction mesure l'erreur du modèle sur des images qu'il n'a jamais vues en entraînement. Elle est donc calculée sur les 10000 images de validation. Si ce taux remonte, cela signifie qu'il y a du surapprentissage.

La précision d'entraînement correspond aux bonnes réponses données par le modèle sur les 40000 images. Son augmentation est normale car le modèle apprend à classer les images.

La précision de validation correspond au taux de bonnes prédictions du modèle sur les 10000 images de validation. Ces images sont appelées données de validation. Elles ne servent pas à apprendre, mais à suivre la performance du modèle à chaque epoch pour éviter le surapprentissage.

On va maintenant analyser les résultats de nos trois modèles.

On évalue notre modèle sur les 10000 images de validation :

```
1 test_loss, test_accuracy = model_mlp.evaluate(x_test, y_test_mlp, verbose=2)
2 test_loss2, test_accuracy2 = model_mlp2.evaluate(x_test, y_test_mlp, verbose=2)
3 test_loss_cnn, test_accuracy_cnn = model_cnn.evaluate(x_test, y_test_cnn, verbose=2)
```

```
Précision sur les données de test : 46.20%
Précision sur les données de test : 47.86%
Précision sur les données de test : 71.08%
```

FIGURE V.30 – Précision de résultat

On observe donc que le modèle de convolution est le plus efficace.

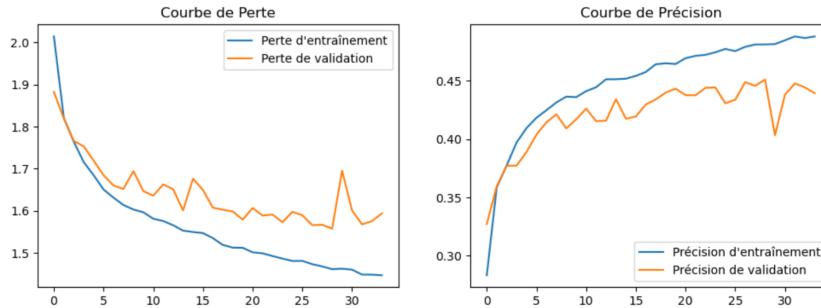


FIGURE V.31 – Évolution du modèle perceptron multicouche 1

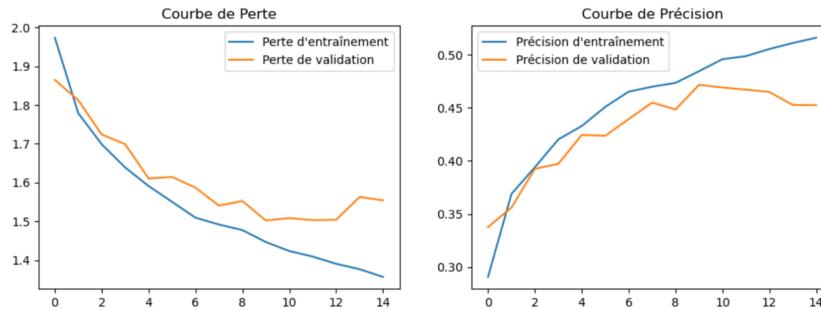


FIGURE V.32 – Évolution du modèle perceptron multicouche 2

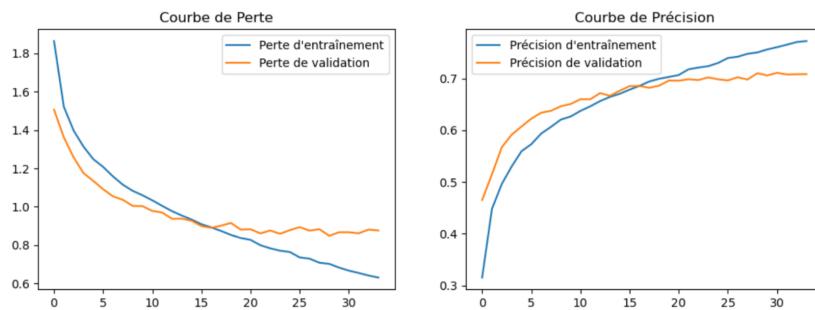


FIGURE V.33 – Évolution du modèle de convolution

On observe ici que nos modèles convergent bien sans trop de surapprentissage.

Avec les codes joints ci-dessous, on affiche les performances de nos modèles pour les n-premières images.

```
Pourcentage modèle 1: 46.0 ( 460 / 1000 )
Pourcentage modèle 2: 47.4 ( 474 / 1000 )
Pourcentage modèle 3: 71.0 ( 710 / 1000 )
```

FIGURE V.34 – Exemple pour mille images

Cela montre que sur les 1000 premières images de notre jeu de données de test, le modèle convolutif (modèle 3) est beaucoup plus efficace que les perceptrons multicouches. Voici un exemple plus visuel :



Prédiction modèle 1 = grenouille
 Prédiction modèle 2 = chat
 Prédiction modèle cnn = chat
 Étiquette réelle = chat



Prédiction modèle 1 = camion
 Prédiction modèle 2 = bateau
 Prédiction modèle cnn = voiture
 Étiquette réelle = bateau



Prédiction modèle 1 = avion
 Prédiction modèle 2 = bateau
 Prédiction modèle cnn = bateau
 Étiquette réelle = bateau



Prédiction modèle 1 = avion
 Prédiction modèle 2 = bateau
 Prédiction modèle cnn = avion
 Étiquette réelle = avion



Prédiction modèle 1 = cerf
 Prédiction modèle 2 = cerf
 Prédiction modèle cnn = grenouille
 Étiquette réelle = grenouille



Prédiction modèle 1 = grenouille
 Prédiction modèle 2 = grenouille
 Prédiction modèle cnn = grenouille
 Étiquette réelle = grenouille



Prédiction modèle 1 = chat
 Prédiction modèle 2 = chat
 Prédiction modèle cnn = voiture
 Étiquette réelle = voiture



Prédiction modèle 1 = grenouille
 Prédiction modèle 2 = grenouille
 Prédiction modèle cnn = oiseau
 Étiquette réelle = grenouille



Prédiction modèle 1 = cerf
 Prédiction modèle 2 = chien
 Prédiction modèle cnn = chat
 Étiquette réelle = chat



Prédiction modèle 1 = voiture
 Prédiction modèle 2 = voiture
 Prédiction modèle cnn = voiture
 Étiquette réelle = voiture



FIGURE V.35 – Affichage sur 10 premiers résultats

4 Présence de bouteilles dans les océans

Le but de ce code est de résoudre un problème de classification à 2 classes. On veut pouvoir dire, sur une partie d'océan, s'il y a une bouteille ou non.

Importation des modules

```
1 import tensorflow as tf
2 from tensorflow.keras import layers, models
3 import keras
```

Définition et compilation du modèle

```
1 # Définition du modèle CNN
2 model = models.Sequential([
3     # Première couche de convolution (32 filtres, taille 3x3)
4     layers.Conv2D(32, (3, 3), activation='relu', input_shape=(60, 60, 4)),
5     layers.MaxPooling2D(pool_size=(2, 2)), # Pooling pour réduire la taille
6
7     # Deuxième couche de convolution (64 filtres)
8     layers.Conv2D(64, (3, 3), activation='relu'),
9     layers.MaxPooling2D(pool_size=(2, 2)),
10
11     # Troisième couche de convolution (128 filtres)
```

```

12 layers.Conv2D(128, (3, 3), activation='relu'),
13 layers.MaxPooling2D(pool_size=(2, 2)),
14
15 # Passage en vecteur 1D
16 layers.Flatten(),
17
18 # Couche entièrement connectée (Dense) avec 32 neurones
19 layers.Dense(32, activation='relu'),
20
21 # Couche de sortie avec activation sigmode (binaire : bouteille ou non)
22 layers.Dense(1, activation='sigmoid')
23 ])
24
25 opt = keras.optimizers.Adam(learning_rate=0.00001)
26 model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
27
28
29 # Résumé du modèle
30 model.summary()

```

Le modèle est construit avec plusieurs types de couches qui permettent d'extraire progressivement des caractéristiques importantes des images avant de les classifier.

La première couche du réseau est une couche de convolution.

```
1 layers.Conv2D(32, (3, 3), activation='relu', input_shape=(60, 60, 4))
```

Cette couche applique 32 filtres de convolution de taille

$$(3, 3)$$

, ce qui permet d'extraire des motifs simples. On peut extraire par exemple les bords ou les textures de l'image. L'activation ReLU est utilisée pour ajouter de la non-linéarité. Ici, on apprend des représentations complexes.

Après cette opération, l'image conserve sa taille de 60×60 pixels, mais elle est maintenant représentée par 60 cartes de caractéristiques qui contiennent des informations sur les motifs détectés.

Ensuite, une couche de pooling est appliquée.

```
1 layers.MaxPooling2D(pool_size=(2, 2))
```

Cette opération réduit la taille de l'image de moitié en ne conservant que les valeurs maximales dans chaque région 2×2 . Cela permet de réduire la quantité de données à traiter tout en conservant les informations importantes.

Une deuxième couche de convolution est ajoutée :

```
1 layers.Conv2D(64, (3, 3), activation='relu')
```

Ici, 64 nouveaux filtres sont appliqués pour détecter des caractéristiques plus complexes, comme des formes ou des motifs plus avancés. Ensuite, une nouvelle couche de pooling est utilisée pour réduire encore la taille des données et ne conserver que les éléments essentiels.

```
1 model_cnn.add(layers.MaxPooling2D(pool_size=(2, 2)))
```

Cette opération réduit la taille de l'image de moitié et on ne conserve que les valeurs maximales dans chaque région 2×2 . Cela permet de réduire la quantité de données à traiter tout en conservant les informations importantes.

On répète encore ces opérations :

```
1 layers.Conv2D(128, (3, 3), activation='relu'),
2 layers.MaxPooling2D(pool_size=(2, 2)),
3
4 # Passage en vecteur 1D
5     layers.Flatten(),
```

Une fois l'extraction des caractéristiques terminée, les données passent par une couche Flatten. Cette couche transforme la matrice de caractéristiques en un vecteur 1D, afin de pouvoir l'envoyer dans une couche dense classique.

```
1 (model_cnn.add(layers.Dense(32, activation='relu')))
```

Cette couche contient 128 neurones et utilise une activation ReLU. Elle permet de combiner les caractéristiques extraites par les couches précédentes pour aider à la classification.

```
1 # Couche de sortie avec une fonction d'activation sigmode (binaire : bouteille ou non)
2     layers.Dense(1, activation='sigmoid')
3 ])
```

Elle contient un neurone, car il y a deux classes dans notre base de données . L'activation sigmoid permet d'obtenir une probabilités associées à chaque classe, afin de prédire la catégorie de l'image.

Par ailleurs, il est important de noter que nous avons dû modifier le learning rate de l'opérateur ADAM. Ce choix a été fait de sorte à pouvoir observer la convergence du modèle. Pour plus de détails sur l'impacte du choix du learning rate, on peut se référer à l'annexe.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 58, 58, 32)	1,184
max_pooling2d (MaxPooling2D)	(None, 29, 29, 32)	0
conv2d_1 (Conv2D)	(None, 27, 27, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 64)	0
conv2d_2 (Conv2D)	(None, 11, 11, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 128)	0
flatten (Flatten)	(None, 3200)	0
dense (Dense)	(None, 32)	102,432
dense_1 (Dense)	(None, 1)	33

FIGURE V.36 – Réseau

Code pour charger notre base de donnée dans le programme

```
1 import pandas as pd
2 import numpy as np
3
4 # Charger le fichier CSV
5 df = pd.read_csv("base_de_donneesV2.csv")
6
```

```

7 # Séparer les labels et les pixels
8 labels = df.iloc[:, 0].values # Colonne des labels
9 pixels_data = df.iloc[:, 1:] # Colonnes des pixels
10
11 def parse_pixel_column(pixel_str):
12     """Convertit une chaîne '[ 0 0 255 255]' en un tableau numpy [0, 0, 255, 255]"""
13     try:
14         # Supprimer les crochets et diviser en une liste de 4 entiers
15         pixel_values = list(map(int, pixel_str.strip("[]").split()))
16
17         # Vérifier que nous avons bien 4 valeurs (R, G, B, A)
18         if len(pixel_values) != 4:
19             raise ValueError(f"Pixel mal formé: {pixel_str}")
20
21         return np.array(pixel_values, dtype=np.uint8)
22
23     except Exception as e:
24         print(f"Erreur de conversion du pixel: {pixel_str} - {e}")
25         return np.array([0, 0, 0, 255], dtype=np.uint8) # Valeur par défaut (noir opaque)
26
27
28 pixels_array = np.array(pixels_data.applymap(parse_pixel_column).values.tolist())
29
30 print(f"Forme après conversion : {pixels_array.shape}") # Devrait être (1997, 3600, 4)
31
32 n_samples, n_pixels, n_channels = pixels_array.shape
33
34 # Vérifier que chaque pixel a bien 4 valeurs
35 if n_channels != 4:
36     raise ValueError(f"Erreur: Chaque pixel doit avoir 4 valeurs, mais shape={pixels_array.shape}")
37
38 # Vérifier que le nombre total de pixels correspond à une image carrée
39 image_size = int(np.sqrt(n_pixels))
40 if image_size * image_size != n_pixels:
41     raise ValueError("Les dimensions des images ne sont pas correctes !")
42
43 # Reshape pour obtenir (n_samples, 60, 60, 4)
44 images = pixels_array.reshape(-1, image_size, image_size, 4).astype("float32") / 255.0
45
46 print(f"Nouvelle forme des images : {images.shape}") # Devrait être (1997, 60, 60, 4)

```

Répartition de l'ensemble de nos données

```

1 # Diviser les données en ensembles d'entraînement et de test
2 from sklearn.model_selection import train_test_split
3 # D'abord, on divise en train + (validation + test)
4 X_train, X_temp, y_train, y_temp = train_test_split(images, labels, test_size=0.3, random_state=42)
5 # Ensuite, on divise (validation + test) en validation et test
6 X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
7
8 from carbontracker.tracker import CarbonTracker
9 from tensorflow.keras.callbacks import EarlyStopping
10 #optimisation du nombre d'epochs
11
12 print(f"Taille des ensembles : Train={len(X_train)}, Validation={len(X_val)}, Test={len(X_test)}")

```

La répartition de notre base de donnée est la suivante :

```
Taille des ensembles : Train=4200, Validation=900, Test=900
```

FIGURE V.37 – Tailles des ensembles

Donc on utilisera 4200 images pour l'entraînement, 900 images pour mesurer la progression de la précision de validation (jeu de données test) et nous testerons notre modèle sur 900 images (jeu de données validation).

Entraînement de notre modèle

```
1 epochs = 10
2
3 tracker_cnn = CarbonTracker(epochs=epochs, monitor_epochs=1, verbose=2, log_dir="logs_carbontracker/cnn/")
4
5
6 # Entraînement avec ensemble de validation
7 tracker_cnn.epoch_start()
8 historique = model.fit(X_train, y_train, epochs=epochs, batch_size=32, validation_data=(X_val, y_val))
9 tracker_cnn.epoch_end()
10 # Évaluation finale sur l'ensemble de test
11 loss, accuracy = model.evaluate(X_test, y_test)
12 print(f"Précision finale sur test : {accuracy * 100:.2f} %")
13
14
15 predictions = model.predict(X_val)
```

Nous avons entraîné notre modèle sur 50 epochs car il obtient très vite de bons résultats. Les résultats ont été directement donnés dans le rapport.

Analyse des résultats : Observation de la divergence d'un modèle

Pour rappel, on entraîne notre modèle sur 4200 images à chaque epochs. On définit la perte d'entraînement qui mesure l'erreur entre les prédictions du modèle et les vraies classes. Notre objectif est donc de la diminuer au maximum.

Par ailleurs, on a mesuré la perte de validation. Cette fonction mesure l'erreur du modèle sur des images qu'il n'a jamais vues en entraînement. Elle est donc calculée sur les 900 images de validation. Si ce taux remonte, cela signifie qu'il y a du surapprentissage.

La précision d'entraînement correspond aux bonnes réponses données par le modèle sur les 4200 images. Son augmentation est normale car le modèle apprend à classer les images.

La précision de validation correspond au taux de bonnes prédictions du modèle sur les 900 images de validation. Ces images sont appelées données de validation. Elles ne servent pas à apprendre, mais à suivre la performance du modèle à chaque epoch pour éviter le surapprentissage.

À l'aide du code suivant, on observe l'évolution des paramètres.

Les résultats suivant sont :

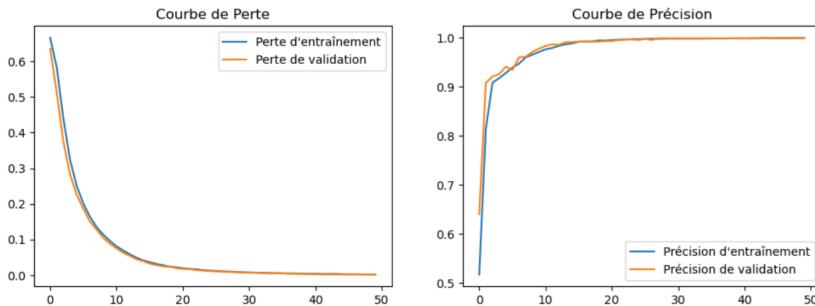


FIGURE V.38 – Résultat paramètre

Les courbes sont très proches, cela signifie que notre modèle est très performant. Ce résultat est dû au fait que notre problème à deux classes est extrêmement simple.

Nous avons entraîné notre modèle sur 50 epochs. Toutefois, nous avons aussi entraîné notre modèle sur 10 epochs. (le taux de précision est de 97,22 %.)

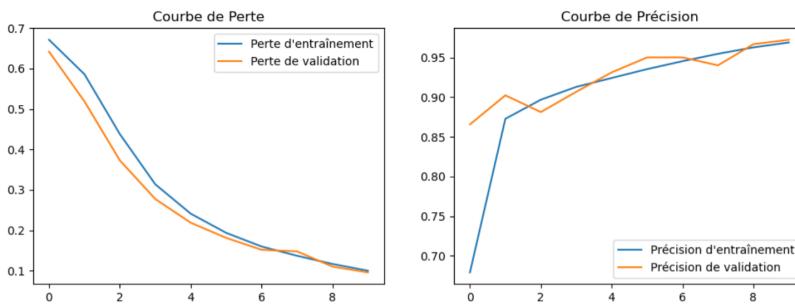


FIGURE V.39 – Différentes courbes

```
1 opt = keras.optimizers.Adam(learning_rate=0.1)
```

Cette fois si, notre modèle n'arrive pas à converger.

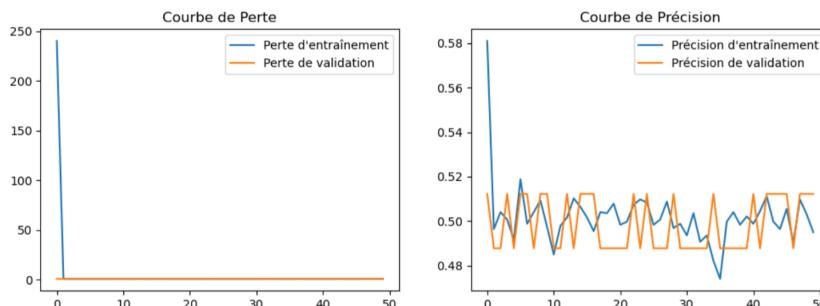


FIGURE V.40 – Résultat de l'entraînement 50 epochs

Nous avons donc défini les paramètres de notre modèle pour qu'il puisse converger tout en pouvant la visualiser.

Résultat en TensorFlow

À l'aide de TensorFlow, nous avons entraîné un modèle sur 50 epochs. Nous avons obtenu que le modèle est précis à 99,89% sur des données qu'il n'a jamais vu. Ce pourcentage très élevé s'explique car notre problème est plutôt simple.

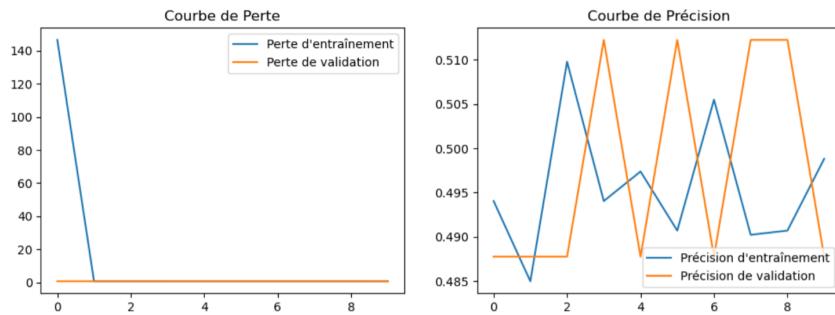


FIGURE V.41 – Résultat de l'entraînement 10 epochs

5 Code de création de la base de données de bouteilles d'eau

L'objectif de cette section est d'implémenter un programme qui permet de générer une base de données d'images répliquant la surface des océans. Cette base de données est un modèle très simpliste de réelles images : des carrés bleus de résolution 60×60 pixels sur lesquels des taches répliquant des reflets de l'eau, de l'écume, etc. sont ajoutées. De plus, sur la moitié des images de la base de données, des bouteilles en plastiques sont ajoutées, ayant subi une rotation et un zoom aléatoire.

Après avoir générée les images, elles sont enregistrées dans un fichier .csv (Comma Separated Values) où les valeurs en RGBA (Red Green Blue Alpha) de chaque pixel sont séparées par des virgules. Voir le code ci-dessous pour plus de détails sur la création de la base de données.

```

1 test_loss, test_accuracy = model_mlp.evaluate(x_test, y_test_mlp, verbose=2)
2 test_loss2, test_accuracy2 = model_mlp2.evaluate(x_test, y_test_mlp, verbose=2)
3 test_loss_cnn, test_accuracy_cnn = model_cnn.evaluate(x_test, y_test_cnn, verbose=2)
```

```

1 from PIL import Image
2 import numpy as np
3 import rembg
4 import tqdm
5 import matplotlib.pyplot as plt
6 import random
7 import csv
8 def bg_remove(infilename : str) -> Image.Image:
9     '''Enlève le fond d'une image, utilise la méthode rembg.remove()
10    IN : infilename (str) - le chemin vers l'image dont on cherche à enlever le fond
11    OUT : img (PIL.Image.Image) - l'image sans le fond'''
12    img = Image.open(infilename) # ouvre l'image img prends le type PIL.Image.Image
13    img.load()
14    img = rembg.remove(img) # enlève le fond
15    return img
16 def alea(image : Image.Image, resolution : int = 60) -> np.array:
17     '''Prend une image et la "randomise" ajoute un zoom et une rotation aléatoire, retourne une matrice numpy
18    IN : image (PIL.Image.Image) - l'image à "randomiser"
19    OUT : data (np.array) - l'image "randomisée"
20    img_bout = image # récupère l'image
21    img_bout.load()
22    # change la taille de manière aléatoire
23    img_bout = img_bout.resize((resolution, resolution))
24    size_factor = random.uniform(0.85, 2.5)
25    new_size = (int(resolution * size_factor), int(resolution * size_factor))
26    img = Image.new("RGBA", (new_size[0], new_size[1]), (255, 255, 255, 0))
27    img.paste(img_bout)
28    # fait une rotation sur l'image
```

```

29     img = img.rotate(random.randint(0, 360))
30     img = img.resize((resolution, resolution))
31     data = np.asarray(img, dtype="int32") # transforme en np.array
32     return data
33
34 def bg_bleu(data : np.array) -> None:
35     '''Permet d'ajouter un fond bleu uni sur une image avec un fond transparent
36     IN : data (np.array) - l'image à colorer'''
37     for i in range(len(data)):
38         for j in range(len(data[0])):
39             if data[i][j][3] == 0:
40                 data[i][j][2], data[i][j][3] = (255, 255)
41
42 def tache(data : np.array, longueur_max : int, hauteur : int) -> None:
43     '''Permet de potentiellement ajouter une tache de taille hauteur*longueur_max au maximum sur l'image.
44     IN : data (np.array) - l'image sur laquelle ajouter une tache
45     IN : longueur_max (int) - la longueur maximale de l'image
46     IN : hauteur (int) - la hauteur maximale de l'image'''
47     # definie la taille de la tache
48     taille = random.randint(10, longueur_max)
49     depart_x = random.randint(1, data.shape[1] - taille - 1)
50     depart_y = random.randint(1, data.shape[0] - hauteur - 1)
51     # couleur de la tache
52     random_blanck = random.randint(1, 150)
53     rand_bleu = (random_blanck, random_blanck, 255 - random.randint(1, 75), 255)
54     # écris la tache sur une position aléatoire si cette position ne se trouve pas sur du bleu
55     for i in range(hauteur):
56         taille_x = random.randint(5, taille)
57         for j in range(taille_x):
58             dx = random.randint(0, 2 * (i + 1))
59             if data[depart_y - i][depart_x + j - dx][0] == 0 and data[depart_y - i][depart_x + j - dx][1] == 0 and data[depart_y - i][depart_x + j - dx][2] == 255:
60                 data[depart_y - i][depart_x + j - dx] = rand_bleu
61
62 def faire_bleu(data : np.array) -> None:
63     '''Transforme une image blanche en une image bleue.
64     IN : data (np.array) - l'image blanche'''
65     for i in range(len(data)):
66         for j in range(len(data[0])):
67             data[i][j][0], data[i][j][1] = (0, 0)
68
69 def matrice_to_list(matrice : np.array, resolution : int = 60) -> list:
70     '''Transforme une matrice en liste. ;
71     IN : matrice (np.array) - la matrice à transformer
72     OUT : liste (list) - la matrice sous forme de liste'''
73     n, m = resolution, resolution
74     liste = []
75     for i in range(n):
76         for j in range(m):
77             liste.append(matrice[i][j])
78     return liste
79
80 if __name__ == '__main__':
81     bouteilles = [] # la liste qui va contenir toutes les données sous forme de matrices
82     nombre_bouteilles = 3000 # le nombre de bouteilles souhaité dans la base de données, divisible par 15
83     # boucle sur les 15 bouteilles téléchargées
84     for i in tqdm.tqdm(range(1, 16)):
85         datas = []
86         # enlève le fond des images
87         image = bg_remove(f'bouteille{i}.png')
88         # crée nombre de bouteilles // 15 bouteilles aléatoire à partir de la bouteille sont on vient d'enlever le fond
89         for _ in range(nombre_bouteilles // 15):
90             datas.append(alea(image))
91         # leur ajoute un fond bleu avec au plus 20 taches aléatoires

```

```

87     for i in range(nombre_bouteilles // 15):
88         bg_bleu(datas[i])
89         for _ in range(20):
90             tache(datas[i], random.randint(20, 50), random.randint(4, 20))
91         bouteilles.append(datas[i])
92
93 nombre_eau = 3000 # le nombre souhaité d'images sans bouteilles, habituellement le même que le nombre de bouteilles
94 # fait un carré bleu
95 # ajoute au plus 15 taches à ce carré
96 for i in tqdm.tqdm(range(nombre_eau)):
97     image = Image.open('carre_bleu.png')
98     image = image.convert('RGBA')
99     image = image.resize((60, 60))
100    data = np.asarray(image, dtype="int32")
101    faire_bleu(data)
102    for _ in range(15):
103        tache(data, random.randint(20, 50), random.randint(4, 20))
104    bouteilles.append(data)
105
106 # transforme la liste des données en liste de listes avec les label des données, pour écrire les données dans un csv
107 matrices_list = []
108 label = [1] * nombre_bouteilles + [0] * nombre_eau
109 for i, matrice in enumerate(bouteilles):
110     matrices_list.append([label[i]] + matrice_to_list(matrice))
111 print('matrice done')
112 noms = [[["label"] + [f"pixel{i}" for i in range(len(matrices_list[0]) - 1)]] # la ligne des noms des données
113 # écrit le fichier
114 with open('base_de_donnees.csv', "w", newline = '') as f:
115     writer = csv.writer(f)
116     writer.writerow(noms + matrices_list)
117
118 #pour l'affichage des données
119 """rows, cols = int(np.sqrt(len(bouteilles))), int(np.sqrt(len(bouteilles)))
120 fig, axes = plt.subplots(rows, cols, figsize=(10, 10))
121 for i, ax in enumerate(axes.flat):
122     ax.imshow(bouteilles[i])
123     ax.axis('off')
124 plt.show()"""

```

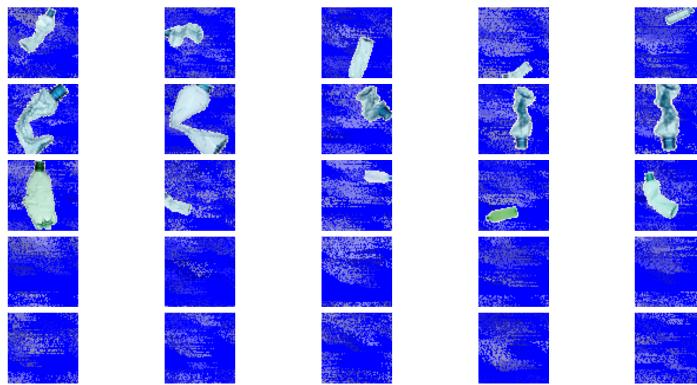


FIGURE V.42 – Aperçu de la base de donnée sur quelques images

```

1  label,pixel0,pixel1,pixel2,pixel3,pixel4,pixel5,pixel6,pixel7,pixel8,pixel9,pixel10,pixel11,pixel12,pixel13,....
2  1,[ 59  59 236 255],[ 69  69 243 255],[ 59  59 236 255],[ 59  59 236 255],[ 59  59 236 255],[ 69  69 243 255],...
3  1,[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],...
4  1,[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],[ 99  99 181 255],[ 0  0 255 255],[ 0  0 255 255],...
5  1,[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],...
6  1,[ 0  0 255 255],[ 0  0 255 255],[ 70  70 195 255],[ 0  0 255 255],[ 0  0 255 255],[ 0  0 255 255],...
7  ...
8  ...
9  ...

```

FIGURE V.43 – Aperçu du fichier qui contient les images

F Surapprentissage

On parle de surapprentissage (overfitting) lorsque le modèle a trop bien appris. Cela signifie qu'il devient trop spécifique à un jeu de données d'entraînement. Il en mémorise des caractéristiques propres à des images qu'il a déjà vues. De ce fait, il devient mauvais face à celles qu'il découvre. Par exemple, imaginons que le modèle essaye de reconnaître si une image représente un humain ou non, et que le modèle apprend à reconnaître un humain grâce à ses yeux (pendant l'entraînement). Si par la suite on lui donne un humain de dos ou avec des lunettes (pendant la phase de test) il ne pourra probablement pas le reconnaître.

Face à ce problème, plusieurs approches permettent d'améliorer la capacité de généralisation d'un modèle. Une méthode consiste à diminuer la capacité du réseau mais en pratique, cela est peu conseillé, cette méthode consiste à régulariser le modèle, c'est-à-dire à réduire légèrement la précision de l'entraînement afin d'améliorer sa capacité de généralisation. Cela peut notamment passer par une diminution du nombre de neurones ou de couches du réseau. Une autre méthode est de stopper l'entraînement plus tôt (et donc de diminuer le nombre d'epochs). On parle donc de « early stopping ». Une dernière méthode est de modifier le jeu d'images. En effet, y ajouter du bruit en zoomant, pixelisant ou encore tournant les images permet que l'apprentissage ne se fasse pas uniquement sur des images standardisées. Chacune des classes à détecter aura plus de représentants et dans des situations plus aléatoires. Cela va permettre au modèle de moins surapprendre et le rendre plus généralisable.

G Problème de conditionnement (normalisation)

Définition : Normalisation

Limiter l'intervalle d'étude dans lequel nos variables d'entrée dans notre réseau de neurones peuvent varier. Permettant ainsi à notre modèle de converger plus facilement et de manière plus stable.

Risque : La fonction de coût risque d'ignorer les variables avec un ordre de grandeur très faible vis-à-vis des autres. Notre modèle peut être biaisé, pour cause, toutes les variables ne sont pas considérées de la même manière.

Exemple : Chaque pixel d'une image représente une variable. Un pixel peut varier entre 0 et 255. Appliquer une normalisation permet de limiter l'intervalle de variation de chaque pixel entre 0 et 1. Permettant ainsi à chaque pixel d'avoir un ordre de grandeur semblable. Pour cause, dans le cas où un pixel a comme valeur 255, celle-ci aura beaucoup plus de poids pour notre fonction de coût qu'un autre pixel ayant comme valeur 1. **Méthode de normalisation :**

Formule : MinMax

$$X_{\text{norm}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

Avec :

- La variable $X_{\text{norm}} \in [0, 1]$
- La variable X représente la valeur possible
- La variable X_{\min} représente la valeur minimale possible pour X
- La variable X_{\max} représente la valeur maximale possible X

Remarque : L'intervalle de variation de nos variables devient donc $[0, 1]$ pour toutes.

H Compte-rendus de réunions :

1 Réunion 1

Noms : CATUSSE Gratien, MOUSSY Lucas, CHANOINE Axel, GELINEAU Arthur, AUBRY Vincent, GUIARD Jean
Absent(s) : ø

Projet 15 : Descente de gradient : aspects mathématiques et algorithmiques

COMPTE RENDU DE REUNION N°1

Date : 13 / 02 / 2025

Professeur référent : BERTUOL Florian

Ordre du jour : Communication et validation de la direction du projet.

Dans un premier temps, nous avons communiqué à Monsieur Bertuol une problématique correspondant à notre projet : Qu'est-ce que la descente de gradient et comment intervient-elle dans la reconnaissance d'images ?

Notre problématique a été validée, néanmoins certains points sont à revoir. En effet, il faudrait faire apparaître plus nettement l'aspect écologique de notre problématique. Pour cela, M. Bertuol nous a suggéré d'évaluer la complexité de nos programmes, afin d'en limiter les coûts énergétiques.

De plus, il nous a conseillé d'entrer en détail dans les aspects mathématiques et algorithmiques durant les prochaines séances. Il est donc important que nous explorions encore le sujet pour en délimiter les limites. De cette façon, nous éviterons de nous perdre dans ce sujet très vaste.

Dans une seconde partie, nous avons fait part à M. Bertuol de notre manière de travailler en intelligence collective. Nous avons partagé notre équipe en deux groupes de trois, les uns travaillant sur la compréhension de la descente de gradient, les autres explorant le champ des applications possibles. En particulier, nous nous sommes orientés vers les réseaux de neurones pour faire de la reconnaissance d'images.

À l'issue de la réunion, le groupe travaillant sur l'aspect mathématique devra essayer de mettre en place ses propres bibliothèques, tandis que le second groupe devra, lui, continuer à exploiter les programmes existants pour en tirer le meilleur, tout en mettant en évidence la descente de gradient dans les algorithmes.

Monsieur Bertuol nous a aussi rappelé l'importance de citer toutes nos sources (livres sur le sujet, vulgarisateurs sur Youtube, sites internet, etc.) ainsi que de vérifier leur fiabilité.

2 Réunion 2

Noms : CATUSSE Gratien, MOUSSY Lucas, CHANOINE Axel, GELINEAU Arthur, AUBRY Vincent, GUIARD Jean
Absent(s) : ø

Projet 15 : Descente de gradient : aspects mathématiques et algorithmiques

COMPTE RENDU DE REUNION N°2

Date : 06 / 03 / 2025

Professeur référent : BERTUOL Florian

Ordre du jour : Ajouter la notion de transition écologique et sociale au projet

Au cours de cette réunion, nous avons d'abord réfléchi à « Comment pouvions nous efficacement ajouter la notion de TES dans le cadre de la reconnaissance d'image ? ».

Nous avons d'abord eu comme idée la problématique suivante : Comment la descente de gradient intervient-elle dans la reconnaissance d'images et est ce que cela peut être bénéfique pour déterminer les zones à fortes densité de déchets dans l'océan ?

Monsieur Bertuol nous a demandé de réfléchir à directement intégrer un exemple d'enjeu TES dans la problématique. Notre choix s'est donc tourné, à la suite de cette réunion, à la problématique suivante : Comment la descente de gradient intervient-elle dans la reconnaissance d'images et est ce que cela peut être bénéfique pour dépolluer les océans ?

Nous avons également réfléchi à la possibilité d'optimiser nos différents algorithmes, afin de limiter le plus possible une consommation énergétique inutile.

Il nous a également demandé de plus expliciter la partie mathématique de la descente de gradient et de ne pas se limiter qu'à son application algorithmique.

plus, nous nous sommes fixés comme objectif la rédaction d'une première version de notre compte rendu. Avec comme deadline : la semaine prochaine.

Il est également utile de justifier nos différents choix (exemple : choix de fonction de coût) et d'expliciter les autres possibilités en annexe.

Pour la rédaction du compte rendu, on doit avant tout garder en tête que l'objectif est de se mettre dans la position de quelqu'un qui explique la descente de gradient à quelqu'un d'autre qui n'y connaît rien du sujet.

Pour la rédaction du compte rendu, le choix s'est tourné vers le logiciel web Overleaf.

3 Réunion 3

Noms : CATUSSE Gratien, MOUSSY Lucas, CHANOINE Axel, GELINEAU Arthur, AUBRY Vincent, GUIARD Jean
Absent(s) : ø

Projet 15 : Descente de gradient : aspects mathématiques et algorithmiques

COMPTE RENDU DE REUNION N°3

Date : 20/03/2025

Professeur référent : BERTUOL Florian

Ordre du jour : Retour sur une première version du compte-rendu.

Lors de ce rendez-vous, nous avons pris conscience des nombreux points forts de notre compte-rendu, mais aussi des améliorations à apporter.

Outre les erreurs classiques ou coquilles d'écriture comme les guillemets anglais, fautes de frappe ou mauvaise présentation des codes, il a été rapporté que certaines parties n'étaient pas finalisées.

Dans la partie mathématique, les points cruciaux étaient présents, mais il y avait un problème de structure ; la différence entre les lemmes, théorèmes et propositions n'était pas claire.

La partie 4, étant en cours de construction, n'apparaissait pas dans le compte-rendu, mais a été sujet de discussion. Les enjeux TES, au cœur de cette partie, ont également pu être clarifiés grâce à M. Bertuol qui d'un point de vue de contenu, nous a dit d'utiliser plus de données chiffrées. Dans un projet tel que le nôtre, qui fait force de la puissance informatique, la consommation, notamment électrique, de cette dernière doit nous préoccuper. Dans cet objectif M. Bertuol nous a fourni des bibliothèques pour évaluer la consommation énergétique de nos algorithmes.

Ces conseils nous ont bien aiguillés pour poursuivre notre compte-rendu, sans avoir radicalement changé notre direction.

4 Réunion 4

Noms : CATUSSE Gratien, MOUSSY Lucas, CHANOINE Axel, GELINEAU Arthur, AUBRY Vincent, GUIARD Jean
Absent(s) : ø

Projet 15 : Descente de gradient : aspects mathématiques et algorithmiques

COMPTE RENDU DE REUNION N°4

Date : 03/04/2025

Professeur référent : BERTUOL Florian

Ordre du jour : Retour sur une deuxième version du compte-rendu.

Lors de cette quatrième et dernière réunion M. Bertuol nous a dit qu'il était très satisfait de notre résultat pour cette version.

Dans l'aspect mathématique le choix des couleurs pour séparer les parties était très judicieux, les erreurs ont été globalement corrigées mais quelques démonstrations de preuves sont encore à faire. Certaines parties qui n'avaient pas été faites auparavant ont été écrites et nous avons pu en discuter, tout semblait correct.

Des anglicismes ont été rapportés et des précisions doivent encore être effectuées sur certaines expressions.

M. Bertuol nous a également fait part de la nécessité de justifier certains des choix faits par le groupe notamment sur les fonctions utilisées dans nos programmes.

Un autre point était aussi l'ordre de présentation entre par exemple batch après les batchs et non l'inverse.

Après avoir discuté de ces détails techniques du rapport, nous avons fait part à M. Bertuol de quelques interrogations que nous avions au sujet des annexes du rapport. Mais les hypothèses que nous avions faites sur nos interrogations se sont avérées vraies, donc nous n'avons pas eu besoin de faire de modifications.

5 Attribution des rôles

rôles	Kick off	Séance 1	Séance 2	Séance 3	Séance 4	Séance 5	Séance 6	Séance 7
facilitateur	Lucas	Gratien	Arthur	Vincent	Jean	Axel	Lucas	Gratien
maître du temps	Axel	Lucas	Gratien	Arthur	Vincent	Jean	Axel	Lucas
distributeur de parole	Jean	Axel	Lucas	Gratien	Arthur	Vincent	Jean	Axel
pousse-décision	Vincent	Jean	Axel	Lucas	Gratien	Arthur	Vincent	Jean
scribe	Arthur	Vincent	Jean	Axel	Lucas	Gratien	Arthur	Vincent
Le coach	Gratien	Arthur	Vincent	Jean	Axel	Lucas	Gratien	Arthur

FIGURE V.44 – Tableau de la rotation des rôles réalisé sur l'outil collaboratif Notion.

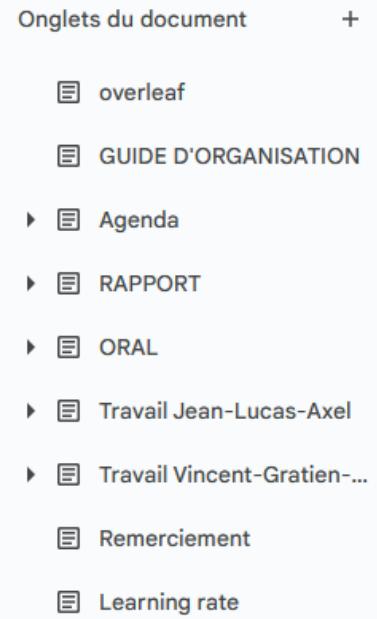


FIGURE V.45 – Aperçu des onglets et donc de l'organisation du Google doc.



FIGURE V.46 – Aperçu des modifications apportées sur le dossier synchronisé avec Git.

Bibliographie

- [1] Ademe. Calculer les émissions carbone de vos trajets, 2009. Consulté le 5 avril 2025.
- [2] Frédéric Bordage. Combien de co2 dans un 1 kwh d'électricité ?, 2009. Consulté le 5 avril 2025.
- [3] Axel Carlier. Introduction aux réseaux de neurones convolutifs. Diaporama de cours, 2025.
- [4] Axel Carlier. Introduction à l'apprentissage et aux réseaux de neurones. Diaporama de cours, 2025. Disponible à <https://acarlier.fr/laprepa/1-NN.pdf>.
- [5] David Gontier. Calcul différentiel et optimisation, 2025. Disponible à https://www.ceremade.dauphine.fr/~gontier/Publications/CalculDiff_Optimisation.pdf.
- [6] Google. Tensorflow api documentation, 2025. Disponible à l'url suivant : https://www.tensorflow.org/api_docs.
- [7] Oles H ospodarskyy, Vasyl Martsenyuk, Natalia Kukharska, Andriy H ospodarskyy, and Sofiia Sverstiuk. Understanding the adam optimization algorithm in machine learning. 2024.
- [8] D Kelleher John. *Deep Learning*. Cambridge Mass. : The MIT Press, 2019.
- [9] Teulières Laure. Mécanismes du greenwashing. COURS 1er année, 2023-2024.
- [10] Le Robert. Base - définition, 2024. Consulté le 5 avril 2025.
- [11] Machine Learnia. Formation deep learning. https://www.youtube.com/playlist?list=PL0_f4PEV1fKoanjvTJbIbd9V5d9Pzp8Rw, 2021. Accédé le : 2025-04-03.
- [12] Moinier Stéphane. Calcul differentiel 3, extrema de fonctions et integrales curvilignes. COURS 2ème année, 2023-2024.
- [13] Moinier Stéphane. Calcul differentiel 2 : Fonctions vectorielles de plusieurs variables. COURS 2ème année, 2024-2025.
- [14] Daphné Wallach. *Le deep learning pour le traitement d'images : classification, détection et segmentation avec Python et TensorFlow*. ENI, Paris, France, 2024.
- [15] Wikipédia. Mnist database. https://en.wikipedia.org/wiki/MNIST_database, 2025. Accédé le : 2025-04-04.
- [16] StatQuest with Josh Starmer. Stochastic gradient descent, clearly explained!!! <https://www.youtube.com/watch?v=vMh0zPT0tLI>, 2019. Accédé le : 2025-04-03.
- [17] Yoshua, Bengio. Codecarbon documentation, 2025. Disponible à l'url suivant : <https://mlco2.github.io/codecarbon/>.