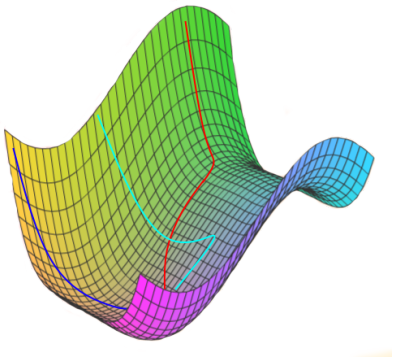
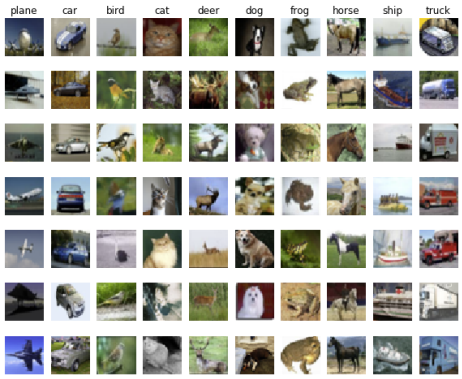
Lycée Blaise Pascal Clermont-Ferrand

Dossier TIPE

Big data, intelligence artificielle et reconnaissance d'espèces marines





Maxime Cautrès
Victor Bellot

Professeurs :
M. Germa
M. Mertens

Octobre 2017 — Juillet 2019

Sommaire

I Solutions technologiques **2**

1 Courte étude de l'existant **2**

2 Qu'est ce que l'intelligence artificielle ? **2**
2.1 Un peu d'histoire . . . 2
2.2 Les grandes idées . . . 2

II Le Projet **3**

3 Les notions : **3**
3.1 Les fonctions F^* et f . . . 3
3.2 La fonction erreur E . . . 3
3.3 L'apprentissage . . . 4
3.4 Les différents types d'algorithmes . . . 4
3.5 Les bases de données . . . 4

III Premier algorithme : **4**

4 Deep Neural Networks : **4**

5 Evolution de l'algorithme **5**
5.1 Découverte de la technologie . . . 5
5.2 Familiarisation avec la rétro-propagation . . . 5
5.2.1 Les notations . . . 5
5.2.2 Les équations mathématiques . . . 5
5.2.3 Les mini-batchs . . . 6
5.2.4 L'algorithme d'apprentissage : . . . 7
5.3 Algorithme final . . . 7
5.3.1 Fonctions d'activations . . . 7
5.3.2 Les optimiseurs . . . 7
5.3.3 Le Drop out et principe de sur-apprentissage . . . 8

6 Résultat **8**

IV Second algorithme **10**

7 Convolutional Neural Networks : **10**

8 Évolution **12**
8.1 Prolongement et adaptation de la rétro-propagation . . . 12
8.1.1 Les notations . . . 12
8.1.2 Les équations mathématiques . . . 12
8.1.3 L'algorithme d'apprentissage . . . 13
8.2 Algorithme final et Batch-Normalization . . . 13

9 Résultats **15**

V Conclusion Générale **16**

VI Preuves **19**

10 Démonstration des équations de la rétro-propagation DNN **19**

11 Démonstration des équations de la rétro-propagation CNN **19**

12 Démonstration des équations de rétro-propagation Batch-normalisation **21**

VII Algorithmes : **22**

13 Algorithme interface et sauvegarde : **22**

14 Algorithme d'exécution du réseau de neurones **23**

15 Algorithme de rétro-propagation **25**

Abstract

In this document, you will find our temporary solution to the problem of the image recognition, using Deep Neural Networks and a version of the Convolutional Neural Network we developed. We explain both techniques and the back-propagation learning algorithms. We show our way of working, discovering new technologies to improve our results on MNIST and CIFAR datasets. Most of the work was done to make algorithms faster, and to find solutions to the problem of over-fitting which appears with data sets like CIFAR.

Introduction

Thème :

Pour cette première année de classe préparatoire, nous avons eu le plaisir dans le cadre du TIPE de travailler sur un Projet dont le Thème était Océan. C'est un sujet vaste qui ouvrait à de nombreuses problématiques allant des questions environnementales à l'optimisation des routes commerciales, en passant par des problématiques de transports. Étant passionnés d'informatique et de mathématiques, nous avons voulu aborder ce sujet à travers la reconnaissance d'images.

Problématique :

Il y a de nos jours de plus en plus d'entreprises telle que Google, Tesla, et bien d'autres qui pratiquent la reconnaissance d'images dans de nombreux domaines,s pour simplifier les recherches sur le net, permettre à des voitures de devenir autonome ou encore faire de la reconnaissance faciale. Lorsque l'on fait des photos sous-marines, il est très difficile pour quelqu'un ne possédant pas de connaissances sur le sujet de savoir ce qu'il photographie ; pour répondre à ce problème, nous avons souhaité utiliser la reconnaissance d'images pour permettre l'identification d'une espèce marine seulement grâce à une photographie. Nous allons donc chercher une technologie nous permettant de répondre à la problématique suivante : **Comment utiliser la grande quantité d'images que l'on peut se procurer aujourd'hui pour permettre l'identification de spécimen marin ?**

Première partie

Solutions technologiques

1 Courte étude de l'existant

Pour répondre à cette problématique, nous avons regardé les différentes technologies existantes. **Une première, très naïve, consiste à comparer notre photo à une base de données constituée d'autres photos pour ensuite choisir celle qui ressemble le plus à la notre, et ainsi en déduire l'espèce**. Cette solution possède les désavantages d'**être extrêmement lente** car pour toute image, elle devra la comparer à une base de données colossale pour que celle-ci soit représentative. De plus, d'autre difficultés liées à la **composi-**

tion de l'image apparaissent, ce qui limite grandement cette technique au point de la rendre inutilisable. Une autre approche consiste à utiliser **l'intelligence artificielle**, elle possède de nombreux avantages puisqu'elle **s'exécute rapidement** une fois entrainée, et n'est pas ou **peu sensible aux différents problèmes de composition**. Nous avons donc décidé d'utiliser cette technologie. Nous étions déjà en partie familiarisés avec celle-ci à la suite d'un projet déjà en cours. L'intelligence artificielle n'est pas forcement la technologie la plus aisée à mettre en place mais c'est celle qui garantira le meilleur ratio résultat/temps d'exécution une fois celle-ci entrainée.

2 Qu'est ce que l'intelligence artificielle ?

2.1 Un peu d'histoire

À la fin du $20^{ème}$ siècle, le créateur de l'ordinateur et le crackeur de la machine de cryptographie Énigma, **Alan Turing**, pensait déjà qu'un jour les ordinateurs, alors appelés Machine de Turing, arriveraient à battre l'homme dans de nombreux domaines. De son vivant, son génie lui a permis de faire des prévisions sur l'évolution des technologies qui se sont finalement avérées justes. Dans un célèbre article de celui-ci , **"Computing machinery and intelligence"**, Alan Turing postule qu'un cerveau humain aurait une capacité mémoire d'environ 1 Go, ce qui à l'époque est colossale. Il en conclue qu'il faudrait créer un algorithme d'au minimum 1 Go qui ne se répète pas pour obtenir une complexité équivalente, ce qui serait le résultat de décennies de recherches et encore, sans être certain que la complexité obtenue soit suffisante. **Il postule qu'il est donc impossible pour l'homme et ses capacités intellectuelles de recréer un cerveau humain adulte. Cependant, il explique qu'il devrait être possible de recréer le cerveau d'un nourrisson, un cerveau "partiellement fini".** Celui-ci est comme vierge, il possède la même structure mais n'a aucune connaissance, aucune logique et donc une complexité bien plus faible car la structure est très simple. L'intelligence artificielle naît de ce postula. Alan Turing n'aura malheureusement pas la chance de voir l'évolution de son travail, puisqu'il meurt prématurément le 7 juin 1954.

2.2 Les grandes idées

Une grande majorité des intelligences artificielles qui ont été créées jusqu'à présent s'inspirent grandement de l'incroyable et perfectionné résultat de l'évolution, notre cerveau. Sa structure est pourtant très simple à comprendre. Le cerveau est un ensemble de neurones plus ou moins connectés entre eux, qui vont modifier un signal électrique porteur d'une l'information. Ce fonctionnement permet de faire émerger des raisonnements extrêmement complexes. **Cette idée d'émergence est la solution au problème de complexité.** En mathématiques et informatique, l'émergence qualifie la capacité d'un phénomène très simple à petite échelle à faire apparaitre des comportements extrêmement complexes à grande échelle. On peut par exemple penser au jeu de la vie de Conway qui à partir de lois simples, permet de faire des choses extraordinaire. La complexité qui émerge de ce jeu est telle qu'il a été démontré qu'il est Turing complet, c'est à dire qu'il est possible d'y simuler tout ce qu'on souhaite, même le jeu lui-même. **Pour créer une intelligence artificielle, il faut recréer un cerveau artificiel** ; il sera composé de neurones, reliés entre eux par des synapses. Il y aura une entrée d'un côté, et de l'autre une sortie. C'est grâce à cette structure que l'on obtiendra la complexité requise. Mais comment passer de cette structure simple à un algorithme ? Pour cela, il faut s'intéresser aux relations entre les neurones, à l'intérieur de ce réseau. **Ici, à chaque synapse est associé un nombre, appelé poids. À chaque neurone est associé un nombre, appelé biais. Les poids et les biais du réseau de neurones sont regroupés dans l'ensemble**

des paramètres du réseau appelé $\mathcal{P}$. Maintenant que l'on a exposé ce qu'est un réseau de neurones, nous pouvons nous intéresser à son fonctionnement.

Deuxième partie
Le Projet

3 Les notions :

3.1 Les fonctions F^* et f

Maintenant, on sait qu'une intelligence artificielle est un cerveau artificiel, qu'il possède une entrée et on attend de lui qu'il nous ressorte une information, que l'on appellera sortie. De cette description, on peut faire une analogie avec un modèle plutôt familier que l'on retrouve en mathématique, l'application. **On va donc représenter le cerveau comme une fonction** qui va de l'ensemble des entrées à l'ensemble des sorties [5] et [1]. **L'entrée sera un vecteur (feature) appartenant à l'ensemble des features noté $\mathcal{F}$, et la sortie (label) sera un vecteur de l'ensemble des labels noté $\mathcal{L}$.** Pour chaque entrée, on attend une sortie précise qui soit en accord avec l'entrée. **On peut donc créer les couples (features, labels) qui sont des points que la fonction doit vérifier**. Ces couples à vérifier peuvent être fournis de différentes manières. Soit avec une **base de données** qui contient les couples (feature, label), cette option est celle que nous avons choisi pour notre reconnaissance d'images ; soit l'algorithme devra de lui même les construire. On ne développera pas les détails de la construction qui ne sont pas utiles dans la réalisation de ce projet, et qui sont très complexes. Nous appellerons dès à présent $F^* : \mathcal{F} \mapsto \mathcal{L}$ **la fonction qui vérifie tout les couples (feature, label)** contenus dans la base de données. **Le but de notre cerveau est d'associer à un feature le bon label**. Ce problème ressemble fortement à celui qu'a résolue Lagrange grâce à ses polynômes interpolateurs. Sauf qu'ici, notre fonction est dans un espaces de bien plus grande dimension et la quantité de points à vérifier est astronomique. C'est pour cela que dans la majorité des cas, **il n'est pas possible d'avoir une résolution algébrique des inconnues de notre cerveau, qui sont les poids et les biais.** Nous allons donc définir la fonction $f : \mathcal{L} \mapsto \mathcal{L}$ qui exécute notre réseau de neurones ; elle associe un label à un feature. **Les images de cette fonction (labels) vont évoluer au cours d'un processus d'apprentissage** qui sera développé plus tard. Il faut cependant savoir qu'**au début de celui-ci, les poids et biais sont définis aléatoirement**. Le but est, au cours de l'entrainement, d'arriver à **modifier les paramètres de telle sorte que f approxime au mieux F^*.**

Nous pouvons donner ici un exemple de problème où il existe une résolution algébrique : la régression affine. Prenons un neurone ; en entrée est présentée l'abscisse d'un point, le poids est un coefficient multiplicatif qui s'appliquera sur l'entrée, le biais est une valeur que l'on ajoute simplement. La fonction f est donc de la forme $f = ax + b = y$. Si l'on veut que notre nuage de point soit le mieux approximé par ce réseau de neurones très simple, il faut calculer a et b grâce à la technique des moindres carrés. L'approximation ne sera pas parfaite car une fonction affine est trop simpliste. Mais on sait trouver les valeurs de a et b qui permettent de minimiser l'approximation.

3.2 La fonction erreur E

Il nous faut maintenant définir une nouvelle fonction permettant de **mesurer les performances de f**. Comme dit précédemment, les paramètres sont initialement posés aléatoirement. Les images de f seront donc elles aussi aléatoires. Pour mesurer l'erreur entre f et F^* pour un même feature, nous allons nous intéresser à **la distances E qu'il y a entre le label Y^* qui aurait dû être prédit et le label prédit y par le réseau de neurones**. Pour cela il faut faire un simple calcul de norme avec le produit scalaire canonique de $\mathbb{R}^{dimension\ de\ la\ sortie}$:

$$E = (Y^* - y)^2 \tag{1}$$

Ici, (1) est vectorisée, c'est-à-dire qu'elle calcule la distance entre un certain nombre de $Y*$ et les y associées. Cela permet de mieux représenter l'erreur du réseau de neurones sans pour autant avoir à recalculer l'erreur sur toutes les données à chaque fois. Cette fonction erreur ne dépend donc pas des features mais seulement des paramètres. Cette fonction aura donc comme ensemble de départ les paramètres de notre réseau de neurones et comme ensemble de sortie une seule valeur, dans $\mathbb{R}^+$. D'où $E : \mathcal{P} \mapsto \mathbb{R}^+$. **Pour augmenter les performances de notre cerveau, il nous faudra donc chercher à minimiser l'erreur entre F^* et f, c'est-à-dire, trouver le minimum de E.**

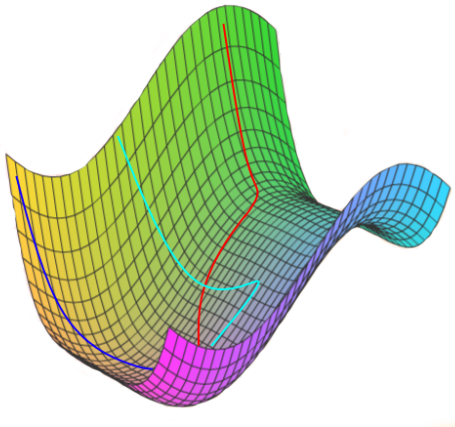


FIGURE 1 – Représentation d'une fonction erreur quelconque (Surface) avec des chemins la minimisant (Courbes)

3.3 L'apprentissage

Maintenant que nous possédons un moyen de calculer l'erreur de notre réseau de neurones, il faut l'utiliser pour modifier nos paramètres. Ces modifications vont se dérouler durant le processus nommé apprentissage. **Le réseau de neurones va être testé et ses paramètres seront modifiés plusieurs milliers de fois jusqu'à obtenir une approximation optimale**. Lors de chacun de ces tests, l'erreur sera calculée sur un certain nombre de couples (feature, label), et sera grâce à des formules mathématiques propagée sur les paramètres, ce qui permettra de savoir dans quel sens et avec quelle intensité chaque paramètre devra être modifié. Il sera donc possible de modifier les paramètres et recommencer ce processus autant de fois qu'il sera nécessaire.

3.4 Les différents types d'algorithmes

Pour la reconnaissance d'images, deux technologies sont accessibles. **La première, appelée DNN (pour Deep Neural Network)** sert de base pour **une seconde appelé**

CNN (pour Convolutional Neural Network). La première utilise un simple **réseau de neurones profond** pour permettre l'analyse d'une image, alors que la seconde plus complète utilise **une analyse préliminaire** pour dégager les informations importantes. Nous avons donc décidé de travailler sur ces deux technologies. Ainsi nous pouvons maintenant commencer à aborder notre travail.

3.5 Les bases de données

Pour ce projet, nous allons majoritairement utiliser **deux bases de données**. Une première nommé **MNIST** [4] mise en ligne par Yann LeCun, contenant **60 000 couples d'images de chiffres écris à la main** (features) avec leur valeur (labels). Cette base de données possède 6000 images pour chaque label ce qui nous donne une belle diversité d'images nécessaire pour l'apprentissage. Cette base de données est **un passage incontournable à valider** avant de s'attaquer à d'autres bien plus complexes. La seconde base de données qui sera utilisée est **extraite CIFAR100** [3], mise en ligne par l'université de Toronto ; elle est composée de 60 000 images classifiées dans 100 catégories. Seule une partie des catégories comporte des images en rapport avec l'océan et les espèces marines. Nous isolerons donc **une quinzaine de catégories** afin de créer une nouvelle base de données que l'on nommera dès à présent **CIFAR15**. Il sera bien plus ardu d'avoir une bonne approximation de F^* que pour MNIST car elle possède un nombre plus faible d'images : 600 images par label et un nombre plus important de labels. De plus, les données internes à chaque catégorie sont bien plus variées ce qui complexifie encore la tâche.

Troisième partie
Premier algorithme :

4 Deep Neural Networks :

Nous avons donc commencer par utiliser un DNN. Il est la **clef de voute de l'intelligence artificielle** car il occupe une place majeur dans une grande partie des algorithmes. Généralement il s'occupe de la dernière partie du **traitement des données** et fournit ainsi les résultats à l'utilisateur. Il est donc incontournable. Cette technologie utilise **seulement un réseau de neurones**. Il y a plusieurs couches constituées de nombreux neurones. On peut ainsi schématiser un réseau de neurones de la manière suivante :

Pour simplifier la compréhension, nous allons nous intéresser à ce qu'il se passe pour un neurone isolé. Voici un schéma plus explicite :

On y voit ainsi un neurone avec 3 synapses. Cette configuration aura donc 3 poids notés w_1, w_2, w_3 respectif à 3 valeurs d'entrées x_1, x_2, x_3 et un biais noté b. On appelle a la valeur du neurone. La relation mathématique reliant a aux autres variables est la suivante :

$$a = \sigma(x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + b) \tag{2}$$

On remarque ici l'apparition d'une fonction σ interne à (2), il s'agit ici d'une fonction d'activation, elle a pour but d'apporter de la non linéarité au réseau de neurones. Nous détaillerons ceci dans un court instant.
Cette formule permettant de calculer a est unique pour tout le réseau, il faut juste l'adapter

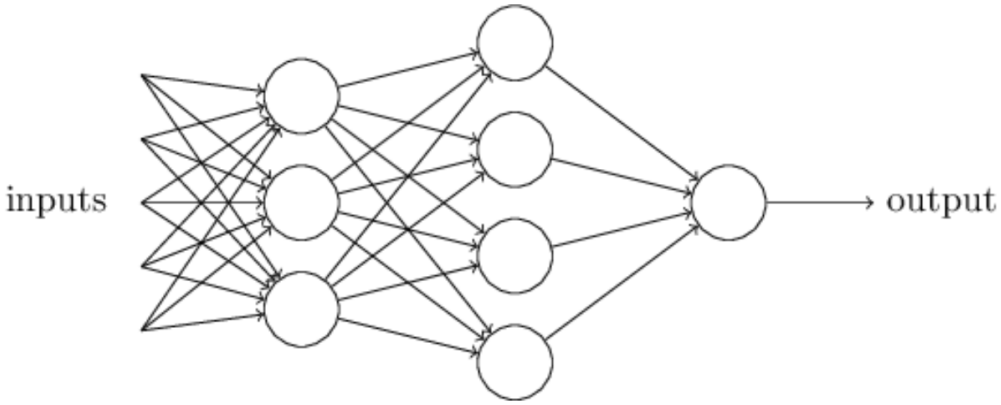


FIGURE 2 – Schéma d'un réseau de neurones

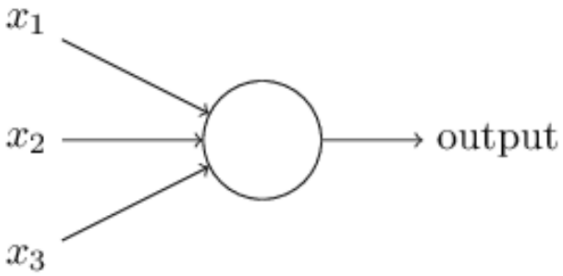


FIGURE 3 – Schéma d'un perceptron (neurone) isolé

en fonction de la taille de chaque couche. Sa variante généralisée est la suivante :

$$a_j^l = \sigma(\sum_k w_{jk}^l \cdot a_k^{l-1} + b_j^l)$$

Ceci étant peut lisible, nous utiliserons une notation :

$$a^l = \sigma(w^l \cdot a^{l-1} + b^l) \tag{3}$$

Voici donc (3) **l'unique relation mathématique nécessaire** pour l'exécution d'un réseau de neurones lorsqu'il s'agit d'un DNN. Il faut maintenant revenir sur **la fonction d'activation**. Elle peut être de plusieurs types, les plus communes étant les fonctions RELU, tangente hyperbolique, sigmoïde et softmax (détaillée plus tard). Elles ont toutes en commun le fait d'être **non linéaire**. En effet, le but de notre réseau de neurones est de **créer de la complexité**, or si on enlève les fonctions d'activations, les relations mathématiques internes au réseau ce **résumeront en une seule et unique combinaison linéaire** des valeurs d'entrées, ce qui n'est clairement pas souhaitable car cela réduirait toute la complexité de notre algorithme. Ainsi ces fonctions d'activations sont des éléments clefs du bon fonctionnement du DNN. Il nous faut donc maintenant définir les relations mathématiques utiles à l'algorithme d'apprentissage.

5 Evolution de l'algorithme

5.1 Découverte de la technologie

Maintenant que nous possédons le cerveau artificiel, il faut l'entrainer. Nous avons souhaité commencer avec un **algorithme de descente de gradient**. La technologie est accessible et

sa mise en place aisée. Pour savoir comment il faut modifier chaque paramètre, on va chercher leur **influence individuelle sur l'erreur**. Pour cela, on calcule l'erreur avec une première version de nos paramètres, et on va ensuite modifier un poids en lui ajoutant une petite valeur ϵ. Puis on calcule à nouveau l'erreur et on la compare à l'ancienne, si celle-ci est supérieure, c'est que la modification est néfaste, et donc qu'il faut effectuer la modification inverse. Au contraire si cette modification diminue l'erreur, on peut la considérer comme bénéfique, et donc la conserver. Lorsque l'on effectue ce processus sur tout les paramètres du réseau, poids et biais, on réduit de manière importante l'erreur de notre notre réseau de neurones. **Au fur et à mesure des itérations, les poids et les biais définis initialement de manière aléatoire vont évoluer jusqu'à tendre vers des valeurs minimisant l'erreur**. Dans cet algorithme, il faut donc choisir la valeur de ϵ. Nous avons choisi $\epsilon = 10^{-5}$, une valeur assez importante pour pouvoir rapidement converger vers une solution, mais pas trop élevée, ce qui aurait pu empêcher l'algorithme de converger. On aperçoit déjà ici **de nombreuses limites** à notre algorithme. Le fait que nous ayons à choisir un pas fixe ralentit l'entrainement au début et l'empêche d'obtenir des résultats très précis sur la fin. De plus, le pas de modification ϵ est le même pour tout les paramètres, donc l'entrainement n'est pas forcement optimal. L'algorithme met aussi **un temps très important** pour atteindre des résultats cohérents car il doit faire un nombre très important de modifications sur chaque paramètre avant de converger. Or chaque modification de paramètre implique deux calculs d'erreur qui nécessitent d'exécuter l'entièreté du réseau. **Sur la base de donnée MNIST, nos résultats était 60% de réussite, ce qui est faible pour cette base de données**. Nous sommes donc partis en quête d'un nouvel outil, plus efficace, pour nous permettre d'obtenir de meilleures performances.

5.2 Familiarisation avec la rétro-propagation

5.2.1 Les notations

Cette solution, nous l'avons trouvée après de nombreuses recherches sur le Net : il s'agit de l'**algorithme de rétro-propagation** [6]. Il va limiter un maximum les calculs en cherchant à minimiser le nombre de fois où l'on ré-exécute le réseau de neurones. Ici, on va utiliser les mathématiques, et grâce à des relations **on va propager l'erreur sur tous les neurones, les poids et les biais**. Cela nous permettra d'avoir à calculer une seule fois l'erreur pour ensuite modifier les paramètres.

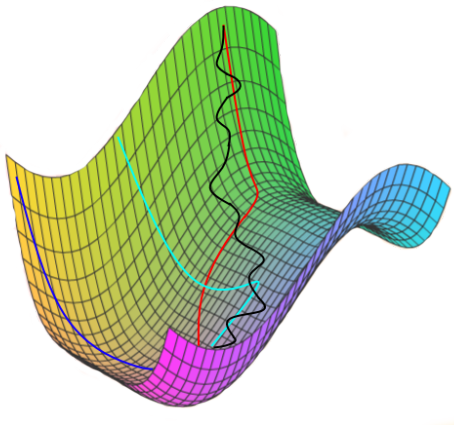


FIGURE 4 – **En rouge :** Descente de gradients conventionnelle très directe mais lente. **En noir :** Exemple de rétro-propagation, moins directe mais bien plus rapide.

Pour arriver à cela, il faut faire des mathématiques. Le but de cette partie est de **trouver des relations permettant de faire reculer l'erreur dans les neurones** et ainsi obtenir pour

chacun des paramètres leur influence sur l'erreur. Il va donc falloir définir plusieurs relations. Commençons par introduire les notations de nos neurones, poids ,biais et autres paramètres :

- La valeur d'un neurone est notée a_n^l pour le n-ième neurone de la l-ième couche.
- Un poids est notée $w_{n,m}^l$ pour le poids du n-ième neurone de la l-ième couche qui s'applique sur le neurone m de la $(l-1)$-ième couche.
- Un biais est notée b_n^l pour le biais du n-ième neurone de la l-ième couche.
- On défini z de la manière suivante :

$$z_n^l = \sum_k a_k^{l-1} \cdot w_{n,k}^l + b_n^l \tag{4}$$

- On a donc $a_n^l = \sigma(z_n^l)$ où σ est une fonction d'activation.

5.2.2 Les équations mathématiques

Maintenant que nous avons présenté les notations, il est possible de travailler sur les équations. Il faut d'abord obtenir **l'influence de la sortie sur l'erreur :**
Pour cela rappelons notre fonction E :

$$E = (Y^* - y)^2 \Leftrightarrow E = < (Y^* - y) | (Y^* - y) > \Leftrightarrow E = \sum_k (Y_k^* - a_k^L)^2$$

Grâce à cette écriture on peut, en utilisant les **dérivées partielles**, calculer l'**influence de chaque paramètre** et valeur de notre réseau sur l'erreur. Cela permet donc d'en déduire les formules suivantes (les preuves sont détaillées dans l'annexe) :

$$\mathrm{d}a_k^L = a_k^L - 2Y_k^* \tag{5}$$
$$\mathrm{d}z^l = \sigma'(z^l) \circ \mathrm{d}a^l \tag{6}$$
$$\mathrm{d}a^{l-1} = T_{w^l} \cdot \mathrm{d}z^l \tag{7}$$
$$\mathrm{d}w^l = \mathrm{d}z^l \cdot T_{a^{l-1}} \tag{8}$$
$$\mathrm{d}b^l = \mathrm{d}z \tag{9}$$

Maintenant que nous possédons l'influence des poids et des biais sur l'erreur, il est possible de les modifier pour ainsi réduire l'erreur. Tout **les paramètres seront donc redéfinis de la façon suivante** :

$$w_{n,m}^l - \alpha \cdot \mathrm{d}w_{m,n}^l \mapsto w_{n,m}^l$$
$$b_n^l - \alpha \cdot \mathrm{d}b_n^l \mapsto b_n^l$$

Ici, α est le learning rate, valeur arbitraire que nous définissons nous même au début de l'entrainement. Elle est de l'ordre de 10^{-3}. Grâce à ces relations, **on possède le strict minimum pour procéder à un entrainement.**

5.2.3 Les mini-batchs

Il y a cependant **quelques modifications à apporter** au fonctionnement de notre algorithme ; pour accélérer son exécution, on peut introduire le concept de mini-batchs. **Un mini-batch est un groupement de features** traités en parallèle grâce à la vectorisation. Cela va nous permettre de faire tout les calculs en une seule fois en considérant le vecteur de features du mini-batch. Cela va ajouter une dimension à toutes nos données, et par des choix astucieux dans les arguments des fonctions numpy (librairie python de calculs matriciels), nous

allons pouvoir conserver les mêmes expressions mathématiques pour l'exécution et la rétro-propagation avec ou sans mini-batchs. Il nous faut préalablement définir de nouvelles notations prenant compte des mini-batchs :

$$a_n^{l,i}, z_n^{l,i}, b_n^{l,i}, w_{n,m}^{l,i}, \mathrm{d}a_n^{l,i}, \mathrm{d}z_n^{l,i}, \mathrm{d}b_n^{l,i}, \mathrm{d}w_{n,m}^{l,i}$$

Nous pouvons donc maintenant définir deux nouvelles notations :

— Les gradients pour les poids :

$$gw_{n,m}^{l} =< \mathrm{d}w_{n,m}^{l,i} >_i$$

— Les gradients pour les biais :

$$gb_{n}^{l} =< \mathrm{d}b_{n}^{l,i} >_i$$

Les paramètres seront donc redéfinis de la manière suivante :

$$w_{n,m}^{l} - \alpha \cdot gw_{m,n}^{l} \mapsto w_{n,m}^{l} \tag{10}$$

$$b_{n}^{l} - \alpha \cdot gb_{n}^{l} \mapsto b_{n}^{l} \tag{11}$$

5.2.4 L'algorithme d'apprentissage :

Maintenant que nous possédons les équations, il ne nous reste plus qu'à créer un protocole d'apprentissage.

— Il faut **créer les mini-batchs** : pour cela il reste une dernière étape de manipulation des données. Nous avons notre base de données sous formes d'une très grande liste de couple (feature, label). Il nous faut la séparer en **deux listes que l'on nommera Train et Test**, puisqu'il nous faut des données pour permettre au réseau de neurones d'apprendre puis des données pour tester celui-ci. Ces tests doivent s'effectuer sur **des données qui lui sont inconnues** pour pouvoir mesurer ses performances dans une utilisation réelle : les images de chiffres ou de poissons que l'on voudra reconnaître ne peuvent pas être dans la base de données lors de l'entrainement. Nous séparons donc en mettant 85% des données dans la liste Train et les 15% restant dans la liste Test.

— Maintenant que l'on possède une liste de données pour l'entrainement, il faut la séparer en mini-batchs de taille n. Pour cela, on **mélange notre liste Train puis l'on sépare la liste en plusieurs mini-batchs.** La taille des mini-batchs ne divisant pas forcément celle de la base de données, il y aura dans la majorité des cas un mini-batch avec moins de données que les autres. Nous décidons de ne pas le considérer pour l'entrainement.

— Nous pouvons donc commencer un entrainement. Pour chaque mini-batch, **l'algorithme va exécuter le réseau de neurones et sauvegarder les matrices de valeurs** a **et** z.

— Il va ensuite **calculer** E.

— Il va **rétro-propager l'erreur** à travers l'entièreté du réseau de neurones grâce aux 4 équations pour **obtenir** $\mathrm{d}a, \mathrm{d}z, \mathrm{d}w$ **et** $\mathrm{d}b$.

— Il va **calculer les gradients** gw **et** gb.

— Et il finira par **mettre à jour** w **et** b

Ce processus est répété autant de fois qu'il y a de mini-batchs. Une fois tout les mini-batchs utilisés, il reste à regarder les **performances de l'algorithme sur la liste Test**. On calcule donc son pourcentage de succès, c'est-à-dire le nombre de bonnes prédictions sur le nombre d'éléments de Test. **On appelle époque l'ensemble de ce processus d'apprentissage sur un unique set de mini-batchs**. Pour améliorer les performances, il faut faire plusieurs époques. Un entrainement complet est donc l'exécution de plusieurs époques (avec à chaque itération des mini-batchs différents grâce au mélange au début de l'époque) sur un même cerveau. Ceci est donc la base de la rétro-propagation. Il existe encore cependant des façons d'améliorer cette technologie.

5.3 Algorithme final

5.3.1 Fonctions d'activations

Maintenant que nous savons précisément ce qu'est une rétro-propagation et comment l'intégrer à notre réseau de neurones, nous pouvons nous intéresser à d'autre **outils qui permettent d'accélérer la convergence** lors de l'entrainement. Nous allons tout d'abord présenter les différentes fonctions d'activation que nous avons souhaitées utiliser et les petites modifications que celles-ci peuvent engendrer.

- La Sigmoïde. Il s'agit de la fonction d'activation **la plus connue**, elle est de la forme $\sigma(z) = \frac{1}{1+e^{-z}}$ de dérivée $\sigma'(z) = \sigma(z)\cdot(1-\sigma(z))$. Elle fournie des valeurs entre 0 et 1 et est non linéaire. Elle possède donc les caractéristiques d'une fonction d'activation mais elle possède **un principal défaut, l'utilisation d'une exponentielle**. En informatique, l'opérateur exponentiel est très lent, donc l'utiliser des millions de fois lors de chaque entrainement va engendrer **une perte de temps conséquente**, c'est pour cette raison que nous avons rapidement **décidé de ne plus l'utiliser**.
- La tangente hyperbolique de dérivée $\tanh'(z) = 1 - \tanh^2(z)$. Cette fonction possède des **caractéristiques similaires à la sigmoïde**, elle est croissante, avec son image comprise entre -1 et 1. Elle est aussi une bonne fonction d'activation, mais elle possède les **mêmes défauts temporels que la sigmoïde**, donc nous avons décidé de ne pas l'utiliser.
- La fonction RELU de la forme $\sigma(z) = z^+ = max(z, 0)$. Sa dérivée est de la forme $\sigma'(z) = \frac{max(0,z)}{|z|} = \frac{RELU(z)}{|z|}$. Cette fonction va dans $\mathbb{R}^+$ et est non linéaire. Elle possède **l'avantage d'être très légère à calculer**, et ce même pour sa dérivée. Elle est donc **idéale** pour la grande quantité de calculs où celle-ci va être utilisée. Par conséquent **elle sera utilisée dans la grande majorité des cas**.
- La fonction Softmax. Celle-ci est **très spécialisée**, elle s'utilise seulement sur la **dernière couche**. Elle est particulière dans le fait qu'elle nécessite de connaître toutes les valeurs de la dernière couche. Elle s'exprime ainsi :

$$\sigma(z_n^L) = \frac{e^{z_n^L}}{\sum_k e^{z_k^L}}$$

Cette fonction d'activation map les valeurs de sortie entre 0 et 1 **de manière exponentielle**. Cela nous permet de pouvoir plus facilement comparer nos résultats avec les labels car cette fonction **dilate les écarts** entre les valeurs basses et élevées du vecteur de sortie (aspect exponentiel). De plus **la somme des composantes du vecteur sortie sera 1, cela pourra nous permettre d'approximer la sortie à un vecteur de probabilité.** Cette fonction d'activation nous oblige ainsi à modifier notre définition de $\mathrm{d}a^L$ et de dz^L. La définition de $\mathrm{d}a^L$ n'est plus nécessaire et l'on trouve directement $\mathrm{d}z^L$ grâce à la relation suivante :

$$\mathrm{d}z_k^L = \frac{1 - Y_k^*}{1 - a_k^L} - \frac{Y_k^*}{a_k^L}$$

Ici la division et l'addition sont des opérateurs termes à termes.

5.3.2 Les optimiseurs

Il faut maintenant introduire les optimiseurs [9]. Dans notre algorithme précédemment présenté, **seuls les gradients calculés lors d'un seul mini-batch sont utilisés pour modifier les paramètres**. Cependant, il existe des **manières plus optimales de procéder** pour modifier les paramètres. Le principales sont les suivantes : **RMSprop, Momentum et Adadelta**. Nous allons expliquer en détail ces 3 optimiseurs :

— RMSprop pour Root Mean Squared prop. Cet optimiseur utilise de nouvelles variables supplémentaires que l'on nomme p, q, gp et qgp avec p notre matrice des paramètre, qg la matrice des gradients des paramètres, et qgp une sauvegarde de la dernière matrice de gradients calculée, et l'on à les relations mathématiques suivantes :

$$q^l = \gamma \cdot qgp^l + (1-\gamma)(gp^l)^2$$

$$p^l - \alpha \frac{gp^l}{\sqrt{q^l + \epsilon}} \mapsto p^l$$

$$qgp^l = q^l$$

RMSprop va influer les gradients calculés en fonction de ceux précédemment calculés, cela va apporter un **petite inertie** grâce au paramètre γ. ici $\epsilon = 10^{-6}$ Après de nombreux tests, nous nous sommes rendu compte que cet optimiseur est bien moins efficaces que les deux suivants.

— Le Momentum est un optimiseur qui va apporter **beaucoup d'importance aux modifications passées**. Il va utiliser, comme le RMSprop, une sauvegarde, mais celle-ci ne permettra pas seulement d'augmenter plus ou moins la norme de gp, elle **influera même sur direction** de gp en mettant une certaine inertie dans la direction des modifications. Donc si les paramètres sont modifiés dans la bonne direction, cela va confirmer la direction. Il est possible qu'un petit nombre de gradients demandent des modifications inverses, or grâce à cet optimiseur, ils seront négligés grâce à l'inertie s'il s'avère que cette modification de direction était seulement passagère, mais elle seront prises en compte si la modification de trajectoire devient plus importante. Le paramètre nous permettant de gérer l'inertie est β. Intéressons-nous maintenant aux équations :

$$q^l = \gamma \cdot qgp^l + (1-\gamma)gp^l$$

$$p^l - \alpha \cdot q^l \mapsto p^l$$

$$qgp^l = q^l$$

— L'Adadelta, il possède les avantages du momentum et la caractéristique d'augmenter grandement la norme du vecteur gradient quand celui-ci confirme la direction des gradients des mini-batchs précédents. Cela permet **encore d'accélérer l'apprentissage** car il se déplacera plus vite. Son expression mathématique est complexe, elle ne sera donc pas évoquée ici.

Nous allons utiliser alternativement les deux derniers optimiseurs au cours du projet, voici des liens [8] et [7] pour des animations illustrant visuellement les différences de performances.

5.3.3 Le Drop out et principe de sur-apprentissage

Une dernière amélioration que l'on peut intégrer est appelée **Drop Out**. Cela va permettre de **réduire la différence entre l'erreur obtenue sur les Tests et la seconde sur les Trains**. Effectivement, notre réseau de neurones a été entrainé sur un certain nombre de données dont il a connaissance, mais nous souhaitons savoir ses performances sur des données qui lui sont inconnues. On juge la capacité d'un réseau de neurones à extraire la sémantique des images qu'il a analysées. **S'il possède un pourcentage de réussite très important sur les Train mais bien plus faible sur les Test**, nous pouvons conclure qu'il a **appris par cœur les données** d'entrainement. Il ne sait donc pas extraire la sémantique des images et sera donc **moins performant quand il ne connaîtra pas les images** sur lesquels il sera testé. **Ce phénomène nommé sur-apprentissage** est très présent pour les bases de données avec

peu d'images pour chaque label et plus généralement peu d'images tout labels confondus. Pour limiter le sur-apprentissage, on peut utiliser le **drop out** : cela consiste à **handicaper l'algorithme en le privant de certain de ses neurones durant l'entrainement**, ce qui a pour conséquences de l'obliger à s'adapter à des situations jusqu'alors inconnues. Plus précisément, le drop out va permettre de **dédoubler un certain nombre de calculs internes** au cerveau. Par exemple, si un seul neurone fait tout le travail et qu'il se retrouve éteint, l'algorithme va en adapter d'autres pour le remplacer, cela permet ainsi quand on le rallumera d'avoir **multiplié le nombre de neurones permettant d'obtenir le résultat**. Or, ce sont les probabilités qui nous le disent, dédoubler les tests permet grandement de limiter les erreurs. À la fin de l'entrainement, **on lui redonne pleine capacité sur des images inconnues et on voit une diminution de l'écart de performance entre Tests et Trains**. Cette diminution confirme l'importance d'effectuer plusieurs fois le même test. Pour désactiver un certain nombre de neurones, on va générer une matrice de même format que $a^{:L}$ contenant des 1 et l'on va mettre dans chaque layer l des valeurs à 0 avec une probabilité de Dor^l (Drop out rate). Ensuite nous multiplierons à un moment précis chaque valeurs par son coefficient associé dans cette matrice, ce qui aura comme conséquence de désactiver le neurone si la valeur est 0 ou l'activer si la valeur vaut 1. Cependant, cette technologie possède le désavantage de beaucoup **réduire les performances du réseau sur les trains**, l'apprentissage étant plus long et les résultats moins bons. Cette technologie est aussi **délicate à mettre en place** car mal intégrée à l'apprentissage, elle peut le rendre impossible. Il nous faudra donc trouver une nouvelle optimisation pour rehausser les performances. Pour exemple, les FIGURE 5 ET 6 montrent des courbes d'évolutions de l'erreur sans drop out et avec un drop out mal placé.

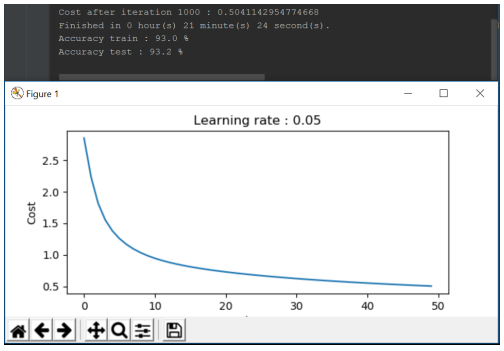


FIGURE 5 – Évolution de l'erreur durant l'apprentissage sans Drop Out

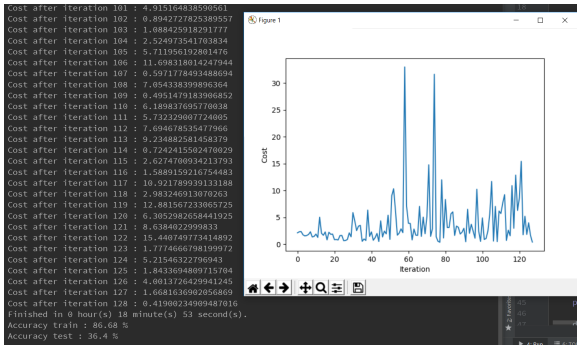


FIGURE 6 – Évolution de l'erreur durant l'apprentissage avec Drop Out mal implémenté

6 Résultat

Maintenant que nous avons un algorithme, nous pouvons parler de **ses performances**. Tout d'abord la base de données **MNIST**, composée de 60000 couples, a été séparée en deux : 50000 pour l'apprentissage et 10000 pour les tests. Ici, les images sont au format 28×28 en nuances de gris. Cela donne 784 neurones sur la couche d'entrée, un par pixel. Puisqu'il y a 10 labels différents, le réseau aura besoin de 10 neurones sur la couche de sortie. Pour augmenter la complexité, nous avons choisi, après de nombreux tests, une couche intermédiaire entre la couche d'entrée et de sortie de 64 neurones, ce qui optimisait les performances par rapport au temps. Cela amène à une **configuration avec** 50816 **poids et** 74 **biais** pour un nombre total de 50890 paramètres à entrainer. **Sans optimiseurs, sans drop out** et avec les fonctions

d'activations, nous avons atteint les 94% **de réussite sur les Trains et** 89% **sur les Tests**. Lorsque l'on rajoute les **optimiseurs, on peut atteindre les** 100% **de réussite sur les Trains et** 94.5% **sur les Test**. En utilisant le **drop out**, nos performances sur les **Trains ont réduit à** 99.1% **mais celle sur les Tests on augmenter avec** 96.1%.

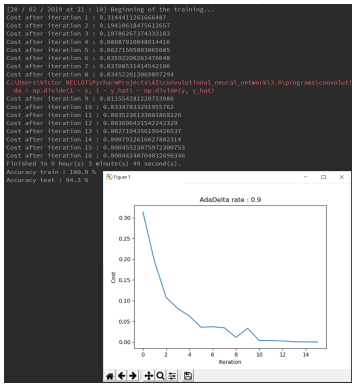


FIGURE 7 – Résultat du meilleur entrainement sans drop out sur MNIST (100% et 94.5%)

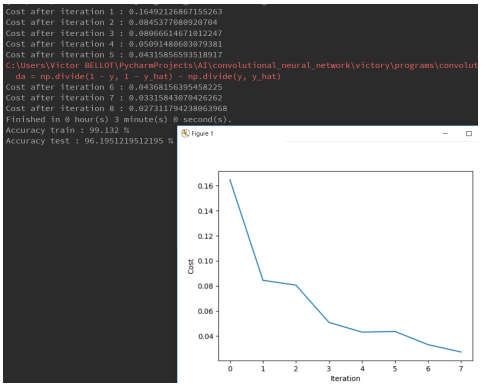


FIGURE 8 – Résultat du meilleur entrainement avec drop out sur MNIST (99.1% et 96.1%)

Nous nous sommes intéressés à **l'influence du drop out sur l'entrainement**, les courbes obtenues ci-dessous représentent l'évolution des performances avec différents taux de drop out à plusieurs endroits du réseau.

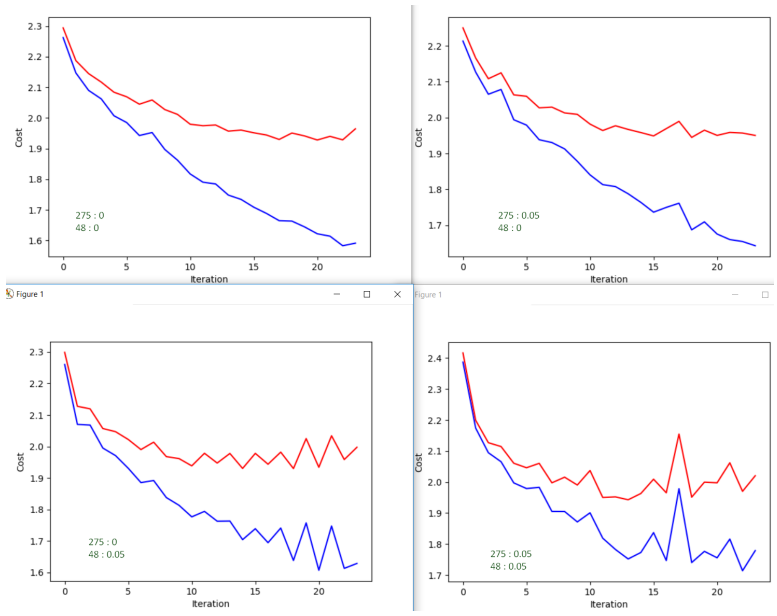


FIGURE 9 – Courbes de performance en fonction des taux de drop out

Les courbes présentées sur les images sont les **courbes d'évolution de l'erreur sur les Trains (bleue) et les Tests (rouge)** au cours des époques d'entrainements. On voit clairement, grâce aux courbes qui se séparent de plus en plus au fils des itérations, que notre algorithme a **un sur-apprentissage important**. On voit aussi **des différences au niveau du lissage de la courbe** en fonction du **taux de drop out. Plus celui-ci est important, plus les courbes sont chaotiques**, cela est dû au fait que le drop out déstabilise l'algorithme en désactivant de façon aléatoire un certain nombre de neurones. On voit aussi qu'il y a **peu ou pas de variation des taux de sur-apprentissage** en fonction du drop out, cela montre

donc la faible influence du drop out sur l'écart. Une technologie permet de rendre le drop out bien plus efficace, elle sera intégrée plus tard. Mais une **très faible variation au niveau de l'erreur va se ressentir grandement au niveau des pourcentages de réussites**. C'est pour cela que nous avons tout de même pu avoir une amélioration de 2% sur les Tests.

Nous souhaitons maintenant utiliser **CIFAR15** comme base de données pour notre réseau de neurones. Celle-ci comporte des **images bien plus lourdes en** $32 \times 32 \times 3$ **RGB**. Si l'on veut conserver une configuration similaire à une couche intermédiaire, cela nous porte à 3072 neurones pour l'entrée, 64 pour la couche intermédiaire et 15 pour la finale. Cependant, d'après nos tests, la complexité permise par cette configuration n'est pas assez importante pour cette base de données , nous avons donc décidé après d'autres tests de rajouter une seconde couche intermédiaire située directement après l'entrée avec 784 neurones. Cela nous fait 2459584 **poids et** 863 **biais**, ce qui porte à 2460447 le nombre de paramètres **à entrainer**. Cette énorme quantité de paramètres à entrainer rend notre **algorithme** presque **inutile** puisqu'il lui faut u**n temps très important pour commencer à converger et ensuite tendre vers des résultats acceptables**. Nous avons donc rapidement **oublié cet algorithme pour répondre à nos besoins** et avons cherché **une nouvelle technologie plus appropriée à la reconnaissance d'images : les réseaux de neurones de convolution.**

Quatrième partie
Second algorithme

7 Convolutional Neural Networks :

Les **réseaux de neurones de convolution** [10] sont l'évolution logique des réseaux de neurones profonds que l'on utilisait jusqu'alors. Cette technologie, que l'on nommera **CNN**, utilise un **DNN classique**, mais ce, **parée d'une première étape de modifications des images**, qui sera elle aussi entrainable. Elle aura pour but de condenser les informations contenues dans l'image initiale dans une image de bien plus petite dimension. Pour cela il nous faut introduire le **principe de filtrage par convolution**. Le filtrage par convolution est une technologie utilisée pour faire **ressortir certaines caractéristiques d'une image**. Elle s'appuie sur le principe de **produit de convolution entre une image et un masque**, l'image et le masque étant sous forme matricielle. Le fonctionnement est le suivant : on prend une matrice k de taille $n \times m$. On fait le produit membre à membre des éléments de k avec un bloc de format $n \times m$ extrait de l'image et de coordonnées de l'angle supérieur gauche (x, y). Une fois le produit d'Hadamard effectué, il faut sommer toutes les valeurs de la matrice obtenue. Cette somme sera assignée à la coordonnée (x, y) de la nouvelle image. Cette opération va ensuite être effectuée sur l'ensemble des pixels de coordonnées (x, y) de l'image d'entrée, ce qui donne une nouvelle image correspondant à l'image initiale passée par le filtre. Il existe des **matrices précises qui ont des rôles déterminés**. Par exemple :

— La matrice $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ permet par un produit de convolution de **faire ressortir les contours des objets** d'une image.

— La matrice $\frac{1}{16} \cdot \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$ **applique un flou** sur l'image d'entrée.

Ces deux matrices sont deux exemples issus de la diversité des filtres existant. **On peut ce-**

pendant se demander comment intégrer ces filtres à un réseau de neurones en sachant qu'il est impossible de savoir quels filtres seront utiles à notre algorithme. Il faudra donc trouver une manière de les déterminer. Un CNN, pour Convolutional Neural Network, va être en deux parties. Une première qui utilisera les convolutions et une seconde qui utilisera un réseau de neurones. **La seconde partie est similaire en tout point à notre premier algorithme.** Il faut donc s'intéresser à la partie **convolutions** : Elle est composée de **plusieurs couches de filtres** ; nous allons considérer une de ces couches pour introduire les notations, le fonctionnement, et les formules. On voit qu'il y a **deux étapes** pour chaque

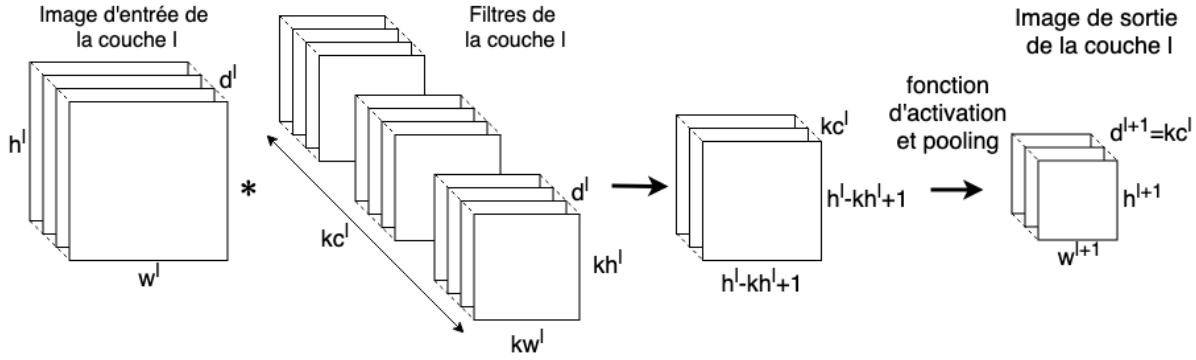


FIGURE 10 – Lien entre deux couches du réseau de convolution.

couche de convolution. Une première consiste en l'**application du produit de convolution**, et une seconde appelée pooling sert à **réduire la dimension de l'image**.

— Commençons par la convolution. Il faut déjà comprendre **comment appliquer un filtre appelé kernal sur une image**. Utilisons un schéma pour appuyer l'explication :

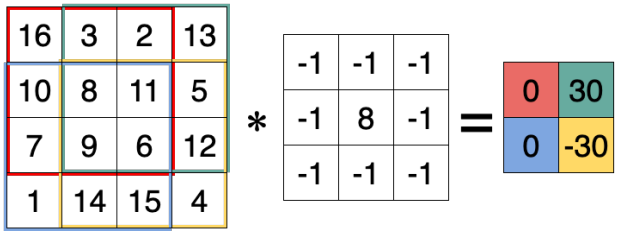


FIGURE 11 – Exemple de calcul d'un produit de convolution.

$$0 = -16 - 3 - 2 - 10 - 11 - 9 - 6 - 7 + 8 \times 8 \tag{12}$$
$$0 = -1 - 6 - 7 - 8 - 10 - 11 - 14 - 15 + 9 \times 8 \tag{13}$$
$$30 = -2 - 3 - 5 - 6 - 8 - 9 - 12 - 13 + 8 \times 11 \tag{14}$$
$$-30 = -4 - 5 - 8 - 9 - 11 - 12 - 14 - 15 + 8 \times 6 \tag{15}$$

Dans la grande majorité des cas, les images sont sur 3 dimensions en non deux, les filtres auront donc 3 dimensions. Il faut donc trouver une écriture mathématique du produit de convolution en 3 dimensions, mais il nous faut avant définir les notations. Soit $A^l_{w,h,d}$ le pixel de coordonnées (w, h, d) de l'image d'entrée A^l de format (W^l, H^l, D^l), soit $K^l_{w,h,d}$ la valeur de coordonnées (w, h, d) du filtre K^l de format $(kw^l, kh^l, kd^l = D^l)$. Soit $Z^l_{w,h,d}$ le pixel de coordonnées (w, h, d) de l'image de sortie Z^l de format $(W^l - kw^l + 1, H^l - kh^l + 1, 1)$. On a donc l'égalité suivante :

$$\forall (w,h,d) \in [\![0, W^l - kw^l]\!] \times [\![0, H^l - kh^l]\!] \times \{0\},$$
$$Z^l_{w,h,0} = \sum_{\substack{aw \in [\![0, kw^l - 1]\!] \\ ah \in [\![0, kh^l - 1]\!] \\ ad \in [\![0, kd^l - 1]\!]}} K^l_{aw,ah,ad} \cdot A^l_{w+aw,h+ah,ad}$$

Or pour **augmenter le pouvoir d'analyse des filtres**, on va appliquer un nombre kc^l de filtres qui dépend de la couche. Il faut donc rajouter un indice à certaines nos notations :

$$K^{l,n}_{w,h,d}, Z^l_{w,h,d}$$

On a donc le format de Z^l qui vaut $(W^l - kw^l + 1, H^l - kh^l + 1, kc^l)$, d'où la nouvelle équation :

$$\forall (w,h,d) \in [\![0, W^l - kw^l]\!] \times [\![0, H^l - kh^l]\!] \times [\![0, kc^l - 1]\!],$$
$$Z^l_{w,h,d} = \sum_{\substack{aw \in [\![0, kw^l - 1]\!] \\ ah \in [\![0, kh^l - 1]\!] \\ ad \in [\![0, kd^l - 1]\!]}} K^{l,d}_{aw,ah,ad} \cdot A^l_{w+aw,h+ah,ad} \tag{16}$$

— Il faut donc maintenant **justifier de la nécessité de la seconde partie d'une couche de convolution**. Elle est composée d'une **fonction d'activation et d'un pooling**. La **fonction d'activation** s'occupe toujours d'apporter de **la non linéarité** pour rendre les calculs finaux irréductibles, ce qui permet de conserver une complexité élevée. Il reste donc à détailler le fonctionnement du pooling. Il faut rappeler que **l'intérêt d'un réseau de convolution est d'avoir des images plus légères à l'entrée du DNN pour limiter le nombre de poids et de biais, il faut donc réduire le nombre de pixels de notre image à chaque étape.** C'est ici qu'intervient le pooling : il va s'appliquer de **différentes manières**, comme le montre l'image qui suit : On voit ici

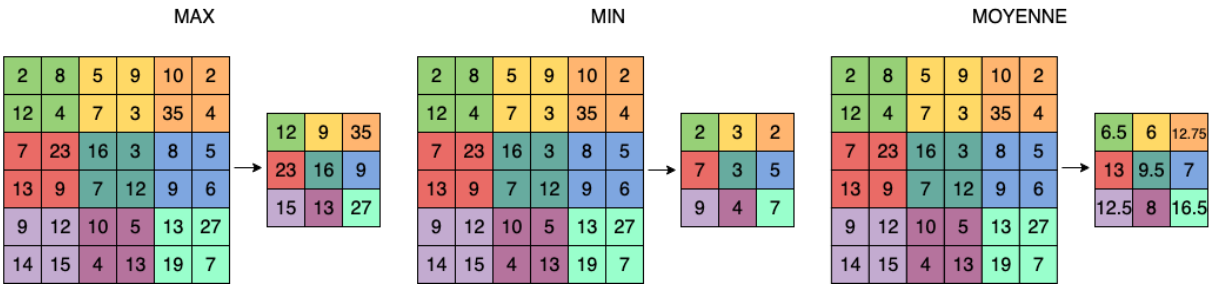


FIGURE 12 – Illustrations selon 3 fonctions de pooling

une logique apparaître : premièrement, après avoir fait passer notre image Z^l dans une fonction d'activation, on obtient une nouvelle image Z'^l de même format (W'^l, H'^l, D'^l). Il va falloir **quadriller l'image en rectangle** de format (pw^l, ph^l) pour leur **associer leur moyenne, leur maximum ou leur minimum** que l'on notera Φ. L'ensemble des valeurs associées à l'image Z'^l est A^{l+1} Établissons les relations mathématiques :

$$\forall (w,h,d) \in [\![0, \left\lfloor \frac{W'^l}{pw^l} \right\rfloor - 1]\!] \times [\![0, \left\lfloor \frac{H'^l}{ph^l} \right\rfloor - 1]\!] \times [\![0, D'^l - 1]\!],$$

$$A_{w,h,d}^{l+1} = \Phi\left((Z_{x,y,d}^{ll}) \begin{cases} x \in [\![w \cdot pw^l, (w+1) \cdot pw^l - 1]\!] \\ y \in [\![h \cdot ph^l, (h+1) \cdot ph^l - 1]\!] \end{cases} \right) \tag{17}$$

Nous avons les trois équations, il ne reste plus qu'à **bien ordonner** cet enchainement sous forme de plusieurs couches puis de faire passer la dernière image dans un réseau de neurones classique pour obtenir notre CNN complet. Voici un **schéma global** regroupant toutes les étapes d'un CNN :

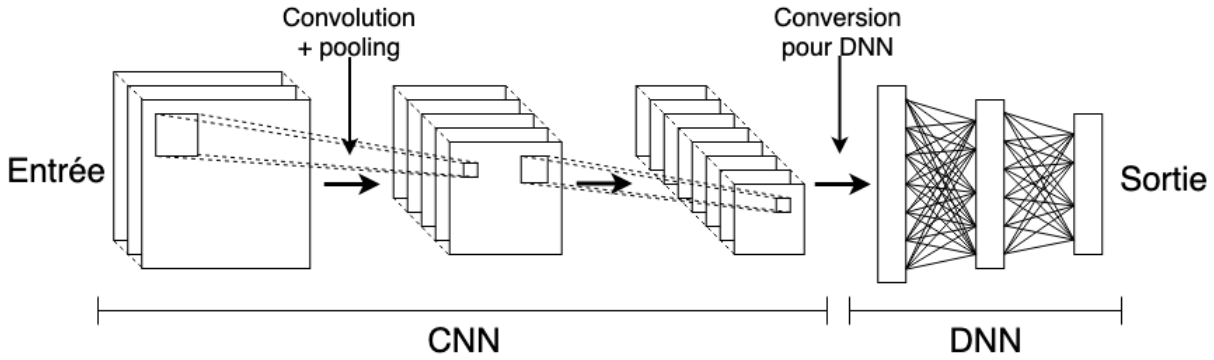


FIGURE 13 – Schéma du cerveau artificiel complet

Cependant nous avons choisi après de nombreuses heures de travail et de nombreux échecs lors de notre première version avec cette architecture, de l'**adapter et de modifier le concept de pooling** pour rendre l'intégration plus aisée. Elle sera **théoriquement moins efficace** qu'un pooling classique mais sera **réellement plus efficace** que notre implémentation du pooling classique grâce aux temps de calculs plus limités. Nous allons poser l'**hypothèse suivante** : **"Le concept de pooling est grossièrement approximable par une simple sélection arbitraire d'un pixel dans chaque carreau de notre image avant pooling"**. Cette hypothèse réduit **légèrement la complexité de notre algorithme mais grandement celle du code**, il ne nous est donc plus utile de programmer les **fonctions de pooling qui s'avèrent très lourdes et compliquées à manipuler lors de la rétro-propagation**. Nous remplaçons donc le pooling par le Stride. Le **Stride** est qualifié avec un format (sw, sh). Si l'on applique simplement notre hypothèse, nous devrions calculer des valeurs qui nous seraient inutiles, or on sait à l'avance quelles valeurs vont nous être utiles, nous pouvons donc encore réduire le nombre de calculs à effectuer en effectuant le strict minimum des opérations nécessaires. Le **Stride ici représente la répartition des valeurs que l'on va calculer dans l'image initiale**. Par exemple, si l'on choisi un Stride de format $(3, 3)$ avec des kernel de format $(2, 2)$ ou un stride de $(2, 2)$ avec des kernel de $(3, 3)$, nous obtenons les schémas suivants :

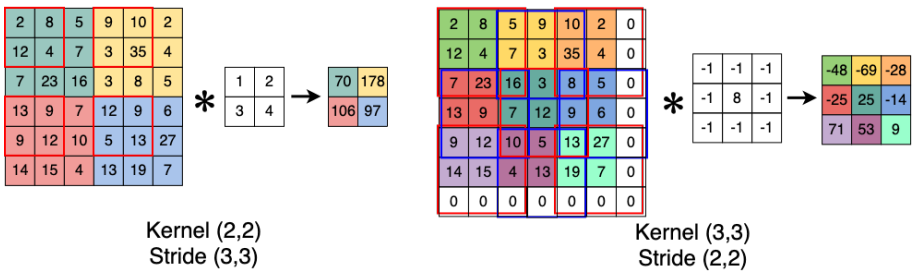


FIGURE 14 – Schéma pour deux formats de kernel et de strides

Ici sur le **deuxième schéma**, on voit qu'une colonne et qu'une ligne de zéro ont été ajoutées

sur les bordures de la matrice. En effet dans cet exemple, **pour appliquer le kernal, il faut adapter la méthode** : il y a deux solutions qui sont le **padding et l'overlapping**. Le **padding** consiste en **l'intégration de l'image initiale dans une plus grosse remplie de 0 de telle sorte que son format soit le plus petit plus grand applicable à la convolution**. **L'overlapping** quant à lui **modifie les emplacements d'application des kernel** en partie hors limites, pour leur donner les derniers emplacements viables.

Il est important de rappeler que dans tous les réseaux de neurones, les paramètres sont posés aléatoirement. **Ici, les poids et les biais du DNN sont toujours des paramètres mais ils sont à présent accompagnés par les valeurs des composantes (poids) des nombreux kernel** qui composent notre CNN. Il faudra donc **compléter les équations de rétro-propagation** pour les adapter au CNN et ainsi permettre d'entraîner nos poids.

8 Évolution

8.1 Prolongement et adaptation de la rétro-propagation

Pour entrainer notre algorithme, nous allons toujours utiliser **une base de données**. Les **idées générales** régissant l'algorithme d'apprentissage seront donc **en partie identiques**. Comme vu sur les schémas précédents, Figure 13, la structure de l'algorithme ce découpe en deux parties dont la seconde est celle d'un DNN. Lors de la **rétro-propagation**, on fait reculer l'erreur dans le réseau donc la **première partie de la rétro-propagation** est **déjà maitrisée**. Il n'y a plus qu'à **établir les relations de rétro-propagation dans le CNN**.

8.1.1 Les notations

Définissons de nouvelles notations pour les influences des paramètres et valeurs du CNN sur l'erreur :

- $\mathrm{d}A^l$ correspond à la matrice des influences de chaque valeur de A^l. Elle se détaille de la manière suivante : $\mathrm{d}A^{l,n}_{w,h,d}$ si l'on considère qu'il s'agit de la matrice du n-ième élément du mini-batch.
- $\mathrm{d}Z^l$ correspond à la matrice des influences de chaque valeur de Z^l. Elle se détaille de la manière suivante : $\mathrm{d}Z^{l,n}_{w,h,d}$ si l'on considère encore une fois la matrice du n-ième élément du mini-batch.
- $\mathrm{d}K^l$ correspond à la matrice des influences de chaque valeurs de K^l. Elle se détaille de la manière suivante : $\mathrm{d}K^{l,i,n}_{w,h,d}$ si l'on considère encore une fois la matrice du n-ième élément du mini-batch. Ici, i est l'indice du i-ème kernal de la couche.

8.1.2 Les équations mathématiques

Il est donc possible de trouver des équations mathématiques pour la rétro-propagation. Commençons par traiter **la jonction entre les deux formes d'algorithmes**. La dernière image en sortie du CNN est mise sous le format d'un vecteur avant d'être mise en entrée du DNN. Lorsque l'on aura grâce à la rétro-propagation obtenu **les influences** $\mathrm{d}a^0$**, nous devrons les reformater de sorte à obtenir** $\mathrm{d}A^L$**.** Ensuite, grâce à des **dérivées partielles, on rétro-propage l'erreur sur les paramètres et valeurs du CNN**. Les équations obtenues par dérivation (les démonstrations sont disponibles dans l'annexe) sont les suivantes :

$$\mathrm{d}Z^l = \sigma'(Z^l) \cdot \mathrm{d}A^{l+1} \tag{18}$$

$$\mathrm{d}A^{l,n}_{w,h,d} = \sum_{k=0}^{kc^l-1} \sum_{x=\lfloor\frac{w-kw^l}{sw^l}\rfloor+1}^{\lfloor\frac{w}{sw^l}\rfloor} \sum_{y=\lfloor\frac{h-kh^l}{sl}\rfloor+1}^{\lfloor\frac{h}{sh^l}\rfloor} (\mathrm{d}Z^{l,n}_{x,y,k} \cdot W^{l,k}_{w-x\cdot sw^l,h-y\cdot sh^l,d}) \tag{19}$$

$$\mathrm{d}K^{l,c,n}_{w,h,d} = \sum_{x=0}^{\lceil\frac{W^l}{sw^l}\rceil} \sum_{y=0}^{\lceil\frac{H^l}{sh^l}\rceil} \mathrm{d}Z^{l,n}_{x,y,c} \cdot A^{l,n}_{x\cdot sw^l+w,y\cdot sh^l+h,d} \tag{20}$$

Ces quelles formules nous permettent ainsi de rétro-propager l'erreur dans tout l'algorithme et ainsi modifier les paramètres qui sont les poids, les biais et le contenu des kernel. Mais, il faut **d'abord obtenir les gradients** de chaque paramètre en faisant la moyenne des influences selon les mini-batchs. La formule de moyenne est la suivantes, avec $n =$ nombre de batchs :

$$gp^l = \frac{1}{n} \sum_{k=0}^{n-1} \mathrm{d}p^{l,n} \tag{21}$$

La formules de modification des paramètres est la suivante :

$$p^l - \gamma \cdot gp^l \mapsto p^l \tag{22}$$

Il reste à établir le processus d'apprentissage.

8.1.3 L'algorithme d'apprentissage

La rétro-propagation va se dérouler en plusieurs étape :
- Calcul de l'erreur en sortie du réseau de neurones.
- Rétro-propagation de l'erreur sur l'ensemble des valeurs et paramètres du DNN.
- Conversion de format de l'erreur du DNN au format de l'erreur de la dernière couche du CNN.
- Pour chaque couche de CNN, on rétro-propage l'erreur en appliquant dans l'ordre la formule (18) pour obtenir $\mathrm{d}Z^l$. Ces influences, grâce aux formules (19) et (20), nous permettent d'obtenir $\mathrm{d}A^l$ et $\mathrm{d}K^l$.
- Calcul des gradients en effectuant la moyenne des influences selon les mini-batchs.
- Modification des paramètres en fonction de ces gradients.

8.2 Algorithme final et Batch-Normalization

Maintenant que la structure de l'algorithme et que l'apprentissage ont été établis, il nous faut essayer d'**optimiser nos performances**. Avec l'algorithme nu précédemment présenté, nous arrivions à atteindre les **100% de réussite sur les Trains de MNIST mais seulement 70% sur le trains**, ce qui n'est **pas qualifiable de bon résultat**. Sur la base de données **CIFAR10**, une variante plus légère et facile de CIFAR15, les performances était de **70% sur les Trains** mais un problème que l'on pensait en partie avoir réglé est réapparue, le **sur-apprentissage**. Nous obtenions des courbes similaires à la suivante :

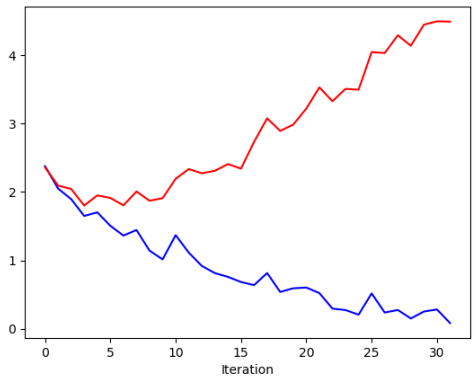


FIGURE 15 – Courbe d'illustration du sur-apprentissage

Ici, l'on voit bien que les **performances** de l'algorithme sur les **Trains** ne cessent de **s'améliorer** mais à partir d'un certains nombres d'époques, les **performances sur les Tests** arrêtent de s'améliorer non pas pour tendre vers une asymptote horizontale **mais pour diverger**. Cela **engendre un algorithme** qui est **inutile** car il n'arrive pas à extraire une sémantique des informations qui lui sont fournies : **sa complexité est utilisée pour apprendre par cœur les données d'entrainement plutôt que de chercher une méthode d'analyse qui pourrait s'adapter à un plus grand nombres d'images**, ce qui est handicapant car il doit être en capacité de reconnaître des objets dans des positions qu'il risque de ne jamais avoir vu, dans des contextes qu'il n'a jamais vu. La dernière partie du projet se consacre donc à **régler au maximum le problème du sur-apprentissage**. Pour cela, nous allons utiliser une technologie appelée batch-normalisation.

La batch-normalization

Ici, il faut réussir à **limiter le sur-apprentissage**. Sur des bases de données comme **MNIST**, où l'on reconnait des **chiffres écrits à la main**, il y a une **faible diversité d'images** pour chaque label. Alors que sur une base de données telle que **CIFAR**, où **deux images correspondant au même label peuvent être très éloignées**.

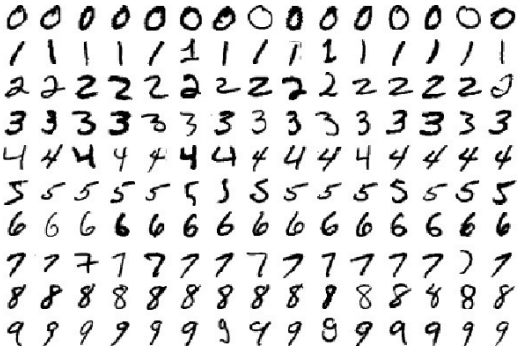


FIGURE 16 – Images issuent de MNIST

FIGURE 17 – Images issuent de CIFAR 10

Il nous faut donc réussir à "uniformiser" nos données. C'est ce que propose la **Batch-normalisation** [2]. Elle ce base sur l'hypothèse que **l'algorithme aurait plus de facilité à reconnaître un peu plus d'images si celle-ci avait une répartition statistique équivalente**. Cette technique se concentre grandement sur **l'utilisation des moyennes et des variances** de nos images. D'un point de vue technique, l'implémentation est cependant plus complexe. Si l'on reprend le schéma global d'un CNN :

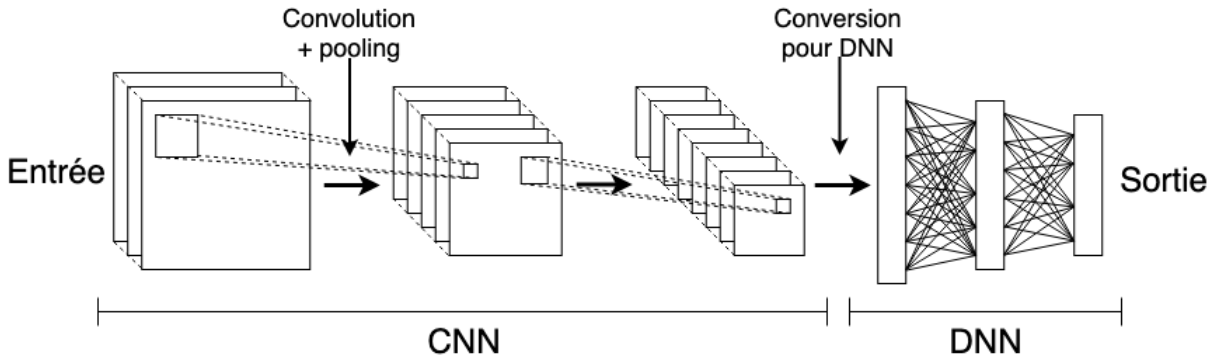


FIGURE 18 – Schéma du cerveau artificiel complet

Il nous faut **rajouter une étape** à l'entrée de chacune des couches. L'étape de normalisation. Cette étape va permettre **d'uniformiser l'entrée en ajustant sa moyenne et sa variance**. Elle va tout d'abord **centrer et réduire** statistiquement les entrées pour ensuite **choisir un nouvel écart type et une nouvelle moyenne**. Ces nouvelles répartitions statistiques que l'on va attribuer à chaque couche sont représentées par deux valeurs ba^l et bg^l, qui sont respectivement les moyennes et les écart-type de chaque couche. Les équations permettant des les appliquer sont les suivantes :

$$< n^l > = \frac{1}{Card(n^l)} \cdot \sum_{k \in n^l} k \tag{23}$$

$$z_k^l = \frac{n_k^l - < n^l >}{\sqrt{\sum_{k \in n^l} (k - < n^l >)^2 + \epsilon}} \cdot bg^l + ba^l \tag{24}$$

Cette étape se situe plus précisément entre le calcul principal de chaque couche et la fonction d'activation qui le suit. On appelle donc n^l l'image de a^l par notre calcul principal, et on a z^l l'image de n^l par la normalisation. Il faut donc établir les équations de rétro-propagation sur cette étape ; ici, les paramètres sont ba^l et bg^l. **Il faut donc 3 nouvelles équations de rétro-propagation :**

$$\mathrm{d}n_k^l = \frac{bg^l}{sqrt \sum_{k \in n^l} (k - < n^l >)^2 + \epsilon} \cdot \mathrm{d}z_k^l \tag{25}$$

$$\mathrm{d}bg^l = \sum_k \frac{n_k^l - < n^l >}{\sqrt{\sum_{k \in n^l} (k - < n^l >)^2 + \epsilon}} \cdot \mathrm{d}z_k^l \tag{26}$$

$$\mathrm{d}ba^l = \sum_k \mathrm{d}z_k^l \tag{27}$$

Elles sont obtenues de la même manière, en appliquant les **dérivées partielles** sur chaque expression selon le paramètre voulu. (Démonstration dans l'annexe)

Maintenant que l'architecture est complète, nous pouvons procéder aux tests et chercher à comprendre les performances de l'algorithme.

9 Résultats

Nous avons **testé de nombreuses configurations** pour le CNN et le DNN en **faisant varier** :

— Pour le CNN :

- Le nombre de couches
- Le nombre de kernel par couche
- Le format des kernel sur chaque couche
- Le format des strides sur chaque couche

— Pour le DNN :
- Le nombre de couches
- Le nombre de neurones par couche

— Les hyper-paramètres :
- les fonctions d'activation
- Ceux des optimiseurs
- Ceux des modifications des paramètres
- L'aléatoire initial

Cela nous a amené après de très **nombreux tests** à une configuration que nous avons considérée **suffisamment performante** pour répondre à notre problème.

Il s'agit de la configuration suivante :

— Pour le CNN :
- 2 couches
- Respectivement 6 et 12
- Respectivement (4,4) et (3,3)
- Respectivement (3,3) et (2,2)

— Pour le DNN :
- 3 couches
- Respectivement 275, 48 et 15 de l'entrée à la sortie

— Pour les hyper-paramètres :
- RELU sauf pour la sortie en Softmax
- Drop out 0.2 sur DNN, Momentum pour l'optimiseur, 0.9 pour le paramètre du momentum
- Utilisation de taux qui varie sur tout la durée de l'entrainement de manière sinusoïdale (de sortes que l'ensemble de l'entrainement corresponde à $[0, \pi]$) ; la taille des mini-batchs est de 128.

Avec cette configuration, **les résultats sur CIFAR15 ont évolué jusqu'à atteindre les 42% de réussite(Tests) en seulement 4 minutes d'entrainement**. Maintenant, si l'on prend en compte (en plus de la sortie de l'algorithme) le second résultat le plus probable, on se rend compte que l'algorithme à des performances atteignant les 70% de réussite. **Cela est très convenable. En sachant que de nombreuses technologies plus performantes n'ont pas été intégrées à l'algorithme car nous ne les comprenions pas, nous pourrions espérer dans un futur proche atteindre les 60 % de réussite en nous penchant plus sérieusement sur ces méthodes.** La batch-normalisation nous a donc permis d'augmenter de 15% les performances de l'algorithme sur CIFAR15. Elle nous a en partie permis de **contourner le problème de diversification**, causé entre autre par la faible quantité de données disponibles pour l'entrainement et de la grande diversité de celle-ci, qui ne permet pas de convenir convenablement l'entièreté de chaque catégorie.

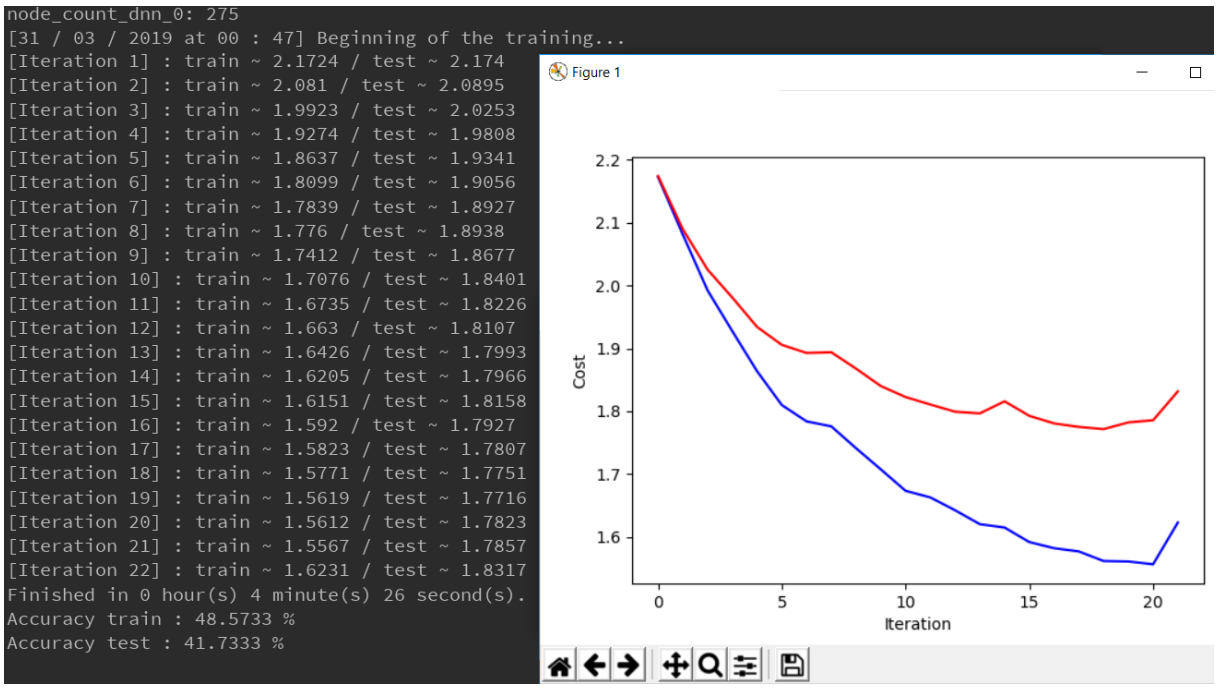


FIGURE 19 – Performance durant l'entrainement sur CIFAR15

Maintenant que nous avons un **algorithme "performant"**, nous avons voulu nous attaquer à un nouveaux problème. Sur la base de donnée **CIFAR100**, deux types de labels sont disponibles. Le premier, **le plus précis**, classe précisément chaque image dans une des **100 sous catégories** de la base de données. **Le second, plus généraliste** regroupe les **100 sous catégories sous 20 catégories plus larges**.

Superclass	Classes
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

FIGURE 20 – Ensemble des catégories et sous catégories de CIFAR100

Essayer de faire de **la reconnaissance** sur ces 20 catégories s'est avéré **très intéressant**, puisque malgré **la grande quantité de données, la diversité de celles-ci est énorme, et il sera donc difficiles d'obtenir de bons résultats**.

Les résultats obtenus après avoir trouvé les bons réglages **paraissent médiocres**, 12%

de réussite. **Cependant nous avons été agréablement surpris en constatant que, oui les résultats n'étaient pas parfaits, mais l'algorithme avait réussi à obtenir une compréhension sémantique de la base données.** Par exemple, quand on lui montrait une **image d'homme**, **il renvoyait à l'ordre près : people, medium-sized mammals et large omnivores and herbivores**, alors qu'il était **seulement entrainé à reconnaître people**. Cela témoigne qu'il arrive à **obtenir un comportement pour lequel il n'a pas été entrainé**. Cela a été un **moment clef** de notre projet : pour la première fois, un **comportement inattendu a émergé**, qui de plus était correct. Cela montre la puissance de cette technologie.

Nous avons terminé l'exploration de cette branche de l'intelligence artificielle sur cette réussite. Dans le futur, **nous allons nous concentrer sur une nouvelle branche qui permet de faire de la prise de décision**.

Cinquième partie

Conclusion Générale

À notre problématique, nous avons trouvé une solution technique. Cet algorithme de reconnaissance d'images fait appel à de **nombreuses thématiques technologiques de notre temps** : l'accès à une grande quantité d'images avec le **bigdata**, la capacité de calculs avec les **super ordinateurs**, et pour finir l'**intelligence artificielle**. Durant ce projet, nous avons appris et en partie compris le fonctionnement d'une intelligence artificielle fonctionnant avec un apprentissage supervisé. Nous avons réussi à obtenir des résultats sur différentes bases de données très diverses, de difficultés variées. Nous avons fait évoluer nos algorithmes pour qu'ils correspondent aux besoins que nous avions. Il est vrai, nous aurions pu avec plus de temps nous intéresser à d'autres types d'algorithmes de deep learning permettant de faciliter la reconnaissance d'images, nous aurions pu nous intéresser à d'autres optimiseurs, et un grand nombre de facettes de la reconnaissance d'images reste à explorer. **Ce projet nous a permis de nous donner des bases solides nous permettant de nous atteler à de nouveaux projets, et nous allons dès à présent concentrer notre travail sur les algorithmes de prises de décisions, avec comme objectif de faire apprendre à nager à un modèle 3D**.

Références

[1] Sanderson Grant. Neural netwotk. 2017. Youtube, 3Blue1Brown.

[2] Sergey Ioffe and Christian Szegedy. Batch normalization : Accelerating deep network training by reducing internal covariate shift. *Le beau journal*, Mars 2015.

[3] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar's dataset, year=2009, note=https ://www.cs.toronto.edu/ kriz/cifar.html.

[4] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits. 1998. http ://yann.lecun.com/exdb/mnist/.

[5] Lê Nguyên Hoang. Intelligence artificielle. 2017. Youtube, Science4All.

[6] Michael Nielsen A. *"Neural Network and Deep Learning"*. Determination Press, 2015.

[7] Alec Radford. Contours of a loss surface and time evolution of different optimization algorithms. Janvier 2016. http ://cs231n.github.io/assets/nn3/opt2.gif.

[8] Alec Radford. A visualization of a saddle point in the optimization landscape, where the curvature along different dimension has different signs. Janvier 2016. http ://cs231n.github.io/assets/nn3/opt1.gif.

[9] Sebastian Ruder. An overview of gradient descent optimization algorithms. Janvier 2016. http ://ruder.io/optimizing-gradient-descent/.

[10] Zhang Zhifei. Derivation of backpropagation in convolutional neural network (cnn). Octobre 2016.

Sixième partie
Preuves

10 Démonstration des équations de la rétro-propagation DNN

Démontrons par récurrence la rétro-propagation, c'est-à-dire qu'il est possible de définir l'influence de chacune des valeurs définies par chaque notation sur la fonction E :
Rappelons l'expression de la fonction erreur :

$$E = (Y^* - y)^2 \Leftrightarrow E = < (Y^* - y) | (Y^* - y) > \Leftrightarrow E = \sum_k (Y_k^* - a_k^L)^2$$

Initialisation :

On peut en déduire l'influence de la dernière couche sur la sortie en utilisant une dérivée partielle.

$$\mathrm{d}a_k^L = \frac{\partial E}{\partial a_k^L} = a_k^L - 2Y_k^*$$

À partir de maintenant, nous allons trouver des formules mathématiques permettant de répandre de manière récursive l'influence de l'erreur.

Hérédité :

Supposons qu'il existe l tel que l'on connaisse da^l, et essayons de calculer da^{l-1}. Nous pouvons tout d'abord calculer l'influence de z sur l'erreur :

$$\mathrm{d}z_k^l = \frac{\partial E}{\partial z_k^l} = \frac{\partial E}{\partial a_k^l} \cdot \frac{\partial a_k^l}{\partial z_k^l} = \sigma'(z_k^l)\mathrm{d}a_k^l$$

Il nous faut maintenant avoir l'influence de a^{l-1} sur E :

$$\mathrm{d}a_k^{l-1} = \frac{\partial E}{\partial a_k^{l-1}} = \frac{\partial E}{\partial z^l} \cdot \frac{\partial z^l}{\partial a_k^{l-1}} = \sum_i w_{i,k}^l \mathrm{d}z_i^l$$

Il nous reste donc à calculer l'influence des poids et biais sur l'erreur :

— Pour les poids :

$$\mathrm{d}w_{i,k}^l = \frac{\partial E}{\partial w_{n,k}^l} = \frac{\partial E}{\partial z_k^l} \cdot \frac{\partial z_k^l}{\partial w_{n,k}^l} = a_i^{l-1}\mathrm{d}z_k^l$$

— Pour les biais :

$$\mathrm{d}b_k^l = \frac{\partial E}{\partial b_k^l} = \frac{\partial E}{\partial z_k^l} \cdot \frac{\partial z_k^l}{\partial b_k^l} = \mathrm{d}z_k^l$$

Conclusion :

D'après le principe de récurrence, on peut en déduire que les influences sont en effet rétro-propageables sur l'ensemble du réseau de neurones.

11 Démonstration des équations de la rétro-propagation CNN

Démontrons par récurrence la rétro-propagation dans le CNN. La propriété P_l est la suivante : Si l'on connait l'influence de la couche l sur l'erreur alors il est possible de connaître l'influence de la couche $l-1$.

Initialisation :
On connait l'influence de la sortie de la dernière couche du CNN grâce à la rétro-propagation sur le DNN. On connait donc P_L.

Hérédité :
Supposons qu'il existe $l \in [\![2, L]\!]$ tel que P_l soit vraie. Montrons que la propriété est vraie au rang $l-1$

— La fonction d'activation :

$$\mathrm{d}Z^{l-1,n}_{w,h,d} = \frac{\partial E}{\partial Z^{l-1,n}_{w,h,d}} \tag{28}$$

$$\mathrm{d}Z^{l-1,n}_{w,h,d} = \frac{\partial E}{\partial A^{l,n}_{w,h,d}} \cdot \frac{\partial A^{l,n}_{w,h,d}}{\partial Z^{l-1,n}_{w,h,d}} \tag{29}$$

$$\mathrm{d}Z^{l-1,n}_{w,h,d} = \sigma'(Z^{l-1,n}_{w,h,d}) \cdot \mathrm{d}A^{l,n}_{w,h,d} \tag{30}$$

On peut ici vectoriser l'expression pour la rendre plus lisible :

$$\mathrm{d}Z^{l-1} = \sigma'(Z^{l-1}) \cdot \mathrm{d}A^{l} \tag{31}$$

— Le produit de convolution :

$$\mathrm{d}A^{l-1,n}_{w,h,d} = \frac{\partial E}{\partial A^{l-1,n}_{w,h,d}} \tag{32}$$

$$\mathrm{d}A^{l-1,n}_{w,h,d} = \frac{\partial E}{\partial Z^{l-1,n}_{w,h,d}} \cdot \frac{\partial Z^{l-1,n}_{w,h,d}}{\partial A^{l-1,n}_{w,h,d}} \tag{33}$$

$$\mathrm{d}A^{l-1,n}_{w,h,d} = \frac{\partial Z^{l-1,n}_{w,h,d}}{\partial A^{l-1,n}_{w,h,d}} \cdot \mathrm{d}Z^{l-1,n}_{w,h,d} \tag{34}$$

Ce formalise ici ne marche pas car les valeurs de $A^{l-1,n}$ s'applique sur de nombreuses valeurs de $\mathrm{d}Z^{l-1,n}_{w,h,d}$, mais pour une seule application de la valeur de $A^{l-1,n}$, cela fonctionne. Il faut maintenant réussir à obtenir l'expression de $\frac{\partial Z^{l-1,n}_{w,h,d}}{\partial A^{l-1,n}_{w,h,d}}$. Tout d'abord, chaque valeur $A^{l-1,n}_{w,h,d}$ est impliquée dans un certain nombre de fois le même calcul, nous allons donc trouver son expression pour un de ces calculs, puis il faudra sommer pour les prendre tous en compte. Ce simple calcul est un produit entre un poids qui s'applique sur la valeur (dont on cherche l'influence) et l'influence de la valeur qui a été calculée avec ce poids. Il nous faut ensuite sommer en faisant varier le poids, cela s'effectue en sommant sur les différents kernel de la couche et sommer sur les différentes influences $\mathrm{d}Z^{l-1,n}$ qui

utilise $A^{l-1,n}_{w,h,d}$ pour être calculées. Cela nous donne la formule suivante :

$$\mathrm{d}A^{l-1,n}_{w,h,d} = \frac{\partial Z^{l-1,n}_{w,h,d}}{\partial A^{l-1,n}_{w,h,d}} \cdot \mathrm{d}Z^{l-1,n}_{w,h,d} \tag{35}$$

$$\mathrm{d}A^{l-1,n}_{w,h,d} = \sum_{k=0}^{kc^{l-1}-1} \sum_{x=\lfloor \frac{w-kw^{l-1}}{sw^{l-1}} \rfloor +1}^{\lfloor \frac{w}{sw^{l-1}} \rfloor} \sum_{y=\lfloor \frac{h-kh^{l-1}}{sl^{l-1}} \rfloor +1}^{\lfloor \frac{h}{sh^{l-1}} \rfloor} (\mathrm{d}Z^{l-1,n}_{x,y,k} \cdot W^{l-1,k}_{w-x\cdot sw^{l-1},h-y\cdot sh^{l-1},d}) \tag{36}$$

— Les poids des kernel :

$$\mathrm{d}K^{l-1,c,n}_{w,h,d} = \frac{\partial E}{\partial K^{l-1,c,n}_{w,h,d}} \tag{37}$$

$$\mathrm{d}K^{l-1,c,n}_{w,h,d} = \frac{\partial E}{\partial Z^{l-1,n}_{w,h,d}} \cdot \frac{\partial Z^{l-1,n}_{w,h,d}}{\partial K^{l-1,c,n}_{w,h,d}} \tag{38}$$

$$\mathrm{d}K^{l-1,c,n}_{w,h,d} = \frac{\partial Z^{l-1,n}_{w,h,d}}{\partial K^{l-1,c,n}_{w,h,d}} \cdot \mathrm{d}Z^{l-1,n}_{w,h,d} \tag{39}$$

Ce formalise ici ne marche pas car les poids s'applique sur de nombreuses valeurs de $\mathrm{d}Z^{l-1,n}_{w,h,d}$, mais pour une seule application de celui-ci, cela fonctionne. Pour obtenir l'expression de $\frac{\partial Z^{l-1,n}_{w,h,d}}{\partial K^{l-1,c,n}_{w,h,d}}$ il faut procéder de la même manière que précédemment. Pour un calcul, l'expression sera le produit d'une des influences de $\mathrm{d}Z^{l-1,n}$ par la valeur de $A^{l-1,n}$ qui correspondant. Il faut ensuite sommer tout ça pour couvrir toute les applications du poids. Pour cela on somme sur les emplacement du kernal durant le produit de convolution. Cela nous donne la formule suivante :

$$\mathrm{d}K^{l-1,c,n}_{w,h,d} = \frac{\partial Z^{l-1,n}_{w,h,d}}{\partial K^{l-1,c,n}_{w,h,d}} \cdot \mathrm{d}Z^{l-1,n}_{w,h,d} \tag{40}$$

$$\mathrm{d}K^{l-1,c,n}_{w,h,d} = \sum_{x=0}^{\lceil \frac{W^{l-1}}{sw^{l-1}} \rceil} \sum_{y=0}^{\lceil \frac{H^{l-1}}{sh^{l-1}} \rceil} \mathrm{d}Z^{l-1,n}_{x,y,c} \cdot A^{l-1,n}_{x\cdot sw^{l-1}+w,y\cdot sh^{l-1}+h,d} \tag{41}$$

Conclusion :

Par le principe de récurrence, on peut conclure que l'erreur est bien rétro-propageable sur l'ensemble du CNN. Cela nous permet donc d'obtenir une rétro-propagation complète sur l'ensemble du réseau de neurones.

12 Démonstration des équations de rétro-propagation Batch-normalisation

Démontrons la propriété suivantes, il est possible de rétro-propager l'erreur sur les zones de normalisation.

On connait les valeurs des influences $\mathrm{d}z_k^l$. En dérivant selon n^l l'expression suivante $\frac{n_k^l - <n^l>}{\sqrt{\sum_{k \in n^l}(k - <n^l>)^2 + \epsilon}} \cdot bg^l + ba^l$ on obtient l'expression $\frac{bg^l}{\sqrt{\sum_{k \in n^l}(k - <n^l>)^2 + \epsilon}}$. En sachant que l'on a l'équivalence suivante :

$$\mathrm{d}n_k^l = \frac{\partial E}{\partial n_k^l} \Leftrightarrow \mathrm{d}n_k^l = \frac{\partial E}{\partial z_k^l} \cdot \frac{\partial z_k^l}{\partial n_k^l} = \frac{\partial z_k^l}{\partial n_k^l} \cdot \mathrm{d}z_k^l \tag{42}$$

On peut ensuite remplacer $\frac{\partial z_k^l}{\partial n_k^l}$ par l'expression obtenu précédemment :

$$\mathrm{d}n_k^l = \frac{bg^l}{\sqrt{\sum_{k \in n^l}(k - < n^l >)^2 + \epsilon}} \cdot \mathrm{d}z_k^l \tag{43}$$

En vectorisant les expressions, on arrive à l'expression suivante :

$$\mathrm{d}n^l = \frac{bg^l}{\sqrt{\sum_{k \in n^l}(k - < n^l >)^2 + \epsilon}} \cdot \mathrm{d}z^l \tag{44}$$

On a ensuite, de manière assez aisée, l'influence du paramètre de moyenne. Lorsque l'on dérive $\frac{n_k^l - <n^l>}{\sqrt{\sum_{k \in n^l}(k - <n^l>)^2 + \epsilon}} \cdot bg^l + ba^l$ selon ba^l, le résultat vaut 1. Si l'on effectue la même dérivation sur $\frac{n^l - <n^l>}{\sqrt{\sum_{k \in n^l}(k - <n^l>)^2 + \epsilon}} \cdot bg^l + ba^l$, on obtient un vecteur du même format que z^l remplit de 1. Or on a le résultat suivant :

$$\mathrm{d}ba^l = \frac{\partial E}{\partial ba^l} \tag{45}$$

$$\mathrm{d}ba^l = \frac{\partial E}{\partial z^l} \cdot \frac{\partial z^l}{\partial ba^l} \tag{46}$$

$$\mathrm{d}ba^l = \frac{\partial z^l}{\partial ba^l} \cdot \mathrm{d}z^l \tag{47}$$

On peut remplacer $\frac{\partial z^l}{\partial ba^l}$ par un vecteur remplit de 1, ce qui nous donne donc le produit scalaire qui somme les composante de z^l :

$$\mathrm{d}ba^l = \frac{\partial z^l}{\partial ba^l} \cdot \mathrm{d}z^l \tag{48}$$

$$\mathrm{d}ba^l = \sum_k \mathrm{d}z_k^l \tag{49}$$

il nous reste à calculer l'influence du paramètre gérant l'écart type. Lorsque l'on dérive $\frac{n^l - <n^l>}{\sqrt{\sum_{k \in n^l}(k - <n^l>)^2 + \epsilon}} \cdot bg^l + ba^l$ selon bg^l, on obtient le vecteur dont les composantes sont $\forall k$, $\frac{n^l - <n^l>}{\sqrt{\sum_{k \in n^l}(k - <n^l>)^2 + \epsilon}} \cdot bg^l + ba^l$. De même que pour les influences précédentes :

$$\mathrm{d}bg^l = \frac{\partial E}{\partial bg^l} \tag{50}$$

$$\mathrm{d}bg^l = \frac{\partial E}{\partial z^l} \cdot \frac{\partial z^l}{\partial bg^l} \tag{51}$$

$$\mathrm{d}bg^l = \frac{\partial z^l}{\partial bg^l} \cdot \mathrm{d}z^l \tag{52}$$

Ici on a de même un produit scalaire, qui se développe et nous donne :

$$\mathrm{d}bg^l = \frac{\partial z^l}{\partial bg^l} \cdot \mathrm{d}z^l \tag{53}$$

$$\mathrm{d}bg^l = \sum_k \frac{n_k^l - < n^l >}{\sqrt{\sum_{k \in n^l} (k - < n^l >)^2 + \epsilon}} \cdot \mathrm{d}z_k^l \tag{54}$$

Septième partie

Algorithmes :

13 Algorithme interface et sauvegarde :

```
""" Main program -- manage UI and convolutional neural network """

import pickle
import numpy as np
import matplotlib.pyplot as plt
import convolutional_neural_network as cnn

" Set hyper-parameters "

save_name = ''
save_type = 'test'
data_set_name = 'cifar_100_coarse'
count = 20
inputs_dimensions = (32, 32, 3)

cnn_topology = {'Lc': 2,
                'kc1': 6, 'kc2': 12,
                'kd1': (4, 4), 'kd2': (3, 3),
                'sc1': (3, 3), 'sc2': (2, 2),
                'afc1': 'relu', 'afc2': 'relu',
                }
dnn_topology = {'Ld': 2,
                'nc1': 48, 'nc2': count,
                'afd1': 'relu', 'afd2': 'softmax',
                'dor0': 0.2, 'dor1': 0.2
                }

epoch_count = 16
mini_batch_size = 256
optimizer = 'momentum'
alpha = '0.6 + 0.4 * math.sin(math.pi * t)'
beta = '0.95 - 0.05 * math.sin(math.pi * t)'
gamma = '0.9'
rho = '0.9'
lambda2C = 0.01
lambda2D = 0.1
training_count = 50000
testing_count = 5000

data = pickle.load(open(data_set_name, 'rb'))
training = (data['train_x'][:, :, :, :training_count], data['train_y'][:, :
    training_count])
testing = (data['test_x'][:, :, :, :training_count], data['test_y'][:, :
    training_count])
labels = data['labels']

" Get parameters "

if save_name == '':

    arg = (cnn_topology, dnn_topology, inputs_dimensions)
    parameters = cnn.initialize_parameters(*arg)
```

```
51
52      print('node_count_dnn_0:', parameters['nc0'])
53
54      arg = (parameters, epoch_count, mini_batch_size, optimizer, alpha, beta,
            gamma, rho, lambda2C, lambda2D, training, testing, save_type)
55      parameters, name, best_cost = cnn.train(*arg)
56
57      hyper_parameters = {'cnn_topology': cnn_topology, 'dnn_topology':
            dnn_topology,
58                          'epoch_count': epoch_count, 'mini_batch_size':
                               mini_batch_size,
59                          'optimizer': optimizer, 'alpha': alpha, 'beta': beta
                               ,
60                          'gamma': gamma, 'rho': rho, 'lambda2C': lambda2C, '
                               lambda2D': lambda2D,
61                          'best_cost': best_cost, 'training_count':
                               training_count,
62                          'testing_count': testing_count, 'save_type':
                               save_type}
63      save = {'parameters': parameters, 'hyper_parameters': hyper_parameters}
64      pickle.dump(save, open(data_set_name+'_'+name, 'wb'))
65
66  else:
67
68      save = pickle.load(open(save_name, 'rb'))
69      parameters = save['parameters']
70      print(save['hyper_parameters'])
71
72      test_x, test_y = testing
73      permutation = list(np.random.permutation(np.arange(testing_count, dtype=
            np.int16)))
74      shuffled_x = test_x[:, :, :, permutation]
75      shuffled_y = test_y[:, permutation]
76
77      for i in range(testing_count):
78          img = shuffled_x[:, :, :, i].reshape(inputs_dimensions + (1, ))
79          goal = list(shuffled_y[:, i]).index(1)
80          prediction = cnn.predict(parameters, img)
81
82          result = {labels[k]: round(float(prediction[k]) * 100, 2) for k in
                range(count)}
83
84          print(sorted(result.items(), key=lambda z: z[1], reverse=True))
85          print(labels[goal])
86
87          plt.close()
88          plt.imshow(np.squeeze(img))
89          plt.show()
```

14 Algorithme d'exécution du réseau de neurones

```
import time
import math
import numpy as np
from copy import deepcopy
import matplotlib.pyplot as plt

def forward(parameters, x, masks=None):
    """
    Apply a forward propagation in order to compute cache

    Take :
    parameters -- dictionary containing all the information about the whole
       network
    x -- features (w, h, d, n)
    masks -- dropout's mask

    Return :
    cache -- dictionary of results
    """

    n = x.shape[3]
    cache = {'xc0': x}

    g = parameters['gc0']
    b = parameters['bc0']
    xh, a, v, m = normalize(x, g, b)
    cache['xhc0'] = xh
    cache['ac0'] = a
    cache['vc0'] = v
    cache['mc0'] = m

    for l in range(1, parameters['Lc'] + 1):
        w = parameters['wc' + str(l)]
        rc = parameters['rc' + str(l)]
        x = convolve(a, w, rc)
        cache['xc' + str(l)] = x

        g = parameters['gc' + str(l)]
        b = parameters['bc' + str(l)]
        xh, z, v, m = normalize(x, g, b)
        cache['xhc' + str(l)] = xh
        cache['zc' + str(l)] = z
        cache['vc' + str(l)] = v
        cache['mc' + str(l)] = m

        af = parameters['afc' + str(l)]
        if af == 'relu':
            a = relu(z)
        elif af == 'tanh':
            a = tanh(z)
        elif af == 'sigmoid':
            a = sigmoid(z)
        cache['ac' + str(l)] = a

    a = a.reshape(-1, n)
    cache['ad0'] = a

    if masks is not None:
```

```
58          a *= masks[0]
59
60      for l in range(1, parameters['Ld'] + 1):
61          w = parameters['wd' + str(l)]
62          x = np.dot(w, a)
63          cache['xd' + str(l)] = x
64
65          g = parameters['gd' + str(l)]
66          b = parameters['bd' + str(l)]
67          xh, z, v, m = normalize(x, g, b)
68          cache['xhd' + str(l)] = xh
69          cache['zd' + str(l)] = z
70          cache['vd' + str(l)] = v
71          cache['md' + str(l)] = m
72
73          af = parameters['afd' + str(l)]
74          if af == 'relu':
75              a = relu(z)
76          elif af == 'softmax':
77              a = softmax(z)
78          elif af == 'tanh':
79              a = tanh(z)
80          elif af == 'sigmoid':
81              a = sigmoid(z)
82
83          if masks is not None and masks[l] is not None:
84              a *= masks[l]
85
86          cache['ad' + str(l)] = a
87
88      return cache
89
90  def convolve(A, W, rc):
91      """
92      Apply weights and biases on A
93
94      Take :
95      A -- numpy matrix, non linear values of the previous layer (w_A, h_A, d,
            n)
96      W -- numpy matrix, weights to apply (count, w_W, h_W, d, 1)
97      rc -- tuple, range of values of w and h
98
99      Return :
100     Z -- numpy matrix, linear values of the current layer (w_Z, h_Z, count,
            n)
101     """
102
103     lx, ly = rc
104     tx, ty = len(lx), len(ly)
105     count, w_W, h_W, _, _ = W.shape
106     Z = np.zeros((tx, ty, count, A.shape[3]))
107
108     for k in range(count):
109         for x in range(tx):
110             for y in range(ty):
111                 w, h = lx[x], ly[y]
112                 Z[x, y, k] += np.sum(A[w:w+w_W, h:h+h_W] * W[k], axis=(0, 1,
                        2))
113
114     return Z
```

15 Algorithme de rétro-propagation

```
1
2  import time
3  import math
4  import numpy as np
5  from copy import deepcopy
6  import matplotlib.pyplot as plt
7
8  def backward(parameters, y, cache, lambda2C=0, lambda2D=0):
9      """
10     Apply a backward propagation in order to compute gradients
11
12     Take :
13     parameters -- dictionary containing all the information about the whole
           network
14     y -- labels (v, n)
15     cache -- dictionary of results
16     lambda2C -- L2 regularization rate for cnn
17     lambda2D -- L2 regularization rate for dnn
18
19     Return :
20     gradients -- partial derivative of each parameters with respect to cost
21     """
22
23     gradients = {}
24     n = y.shape[1]
25
26     y_hat = cache['ad' + str(parameters['Ld'])]
27     da = np.divide(1 - y, 1 - y_hat) - np.divide(y, y_hat)
28     dz = None
29
30     for l in reversed(range(1, parameters['Ld'] + 1)):
31         z = cache['zd' + str(l)]
32         af = parameters['afd' + str(l)]
33
34         if af == 'relu':
35             dz = da * relu_prime(z)
36         elif af == 'softmax':
37             dz = y_hat - y
38         elif af == 'tanh':
39             dz = da * tanh_prime(z)
40         elif af == 'sigmoid':
41             dz = da * sigmoid_prime(z)
42
43         g = parameters['gd' + str(l)]
44         x_p = cache['xd' + str(l)]
45         xh_p = cache['xhd' + str(l)]
46         v = cache['vd' + str(l)]
47         m = cache['md' + str(l)]
48         dz, dg, db = normalize_prime(dz, g, x_p, xh_p, v, m)
49         gradients['dgd' + str(l)] = dg
50         gradients['dbd' + str(l)] = db
51
52         a_p = cache['ad' + str(l - 1)]
53         w = parameters['wd' + str(l)]
54         gradients['dwd' + str(l)] = (1 / n) * (np.dot(dz, a_p.T) + lambda2D
               * np.power(w, 2))
55
56         da = np.dot(w.T, dz)
```

```
57
58      da = da.reshape(cache['ac' + str(parameters['Lc'])].shape)
59
60      for l in reversed(range(1, parameters['Lc'] + 1)):
61          z = cache['zc' + str(l)]
62          af = parameters['afc' + str(l)]
63
64          if af == 'relu':
65              dz = da * relu_prime(z)
66          elif af == 'tanh':
67              dz = da * tanh_prime(z)
68          elif af == 'sigmoid':
69              dz = da * sigmoid_prime(z)
70
71          g = parameters['gc' + str(l)]
72          x_p = cache['xc' + str(l)]
73          xh_p = cache['xhc' + str(l)]
74          v = cache['vc' + str(l)]
75          m = cache['mc' + str(l)]
76          dx, dg, db = normalize_prime(dz, g, x_p, xh_p, v, m)
77          gradients['dgc' + str(l)] = dg
78          gradients['dbc' + str(l)] = db
79
80          a_p = cache['ac' + str(l - 1)]
81          w = parameters['wc' + str(l)]
82          rc = parameters['rc' + str(l)]
83          da, dw = deconvolve(dx, w, a_p, rc)
84
85          gradients['dwc' + str(l)] = (1 / n) * (dw + lambda2C * np.power(w,
                2))
86
87      g = parameters['gc0']
88      x_p = cache['xc0']
89      xh_p = cache['xhc0']
90      v = cache['vc0']
91      m = cache['mc0']
92      dx, dg, db = normalize_prime(da, g, x_p, xh_p, v, m)
93      gradients['dgc0'] = dg
94      gradients['dbc0'] = db
95
96      gradients['dxc0'] = dx
97
98      return gradients
99
100 def deconvolve(dZ, W, A_p, rc):
101     """
102     Compute gradients
103
104     Take :
105     dZ -- numpy matrix, gradients of the next layer (w_dZ, h_dZ, count, n)
106     W -- numpy matrix, weights (count, w_W, h_W, d, 1)
107     A_p -- previous A (w_dA, h_dA, d, n)
108     rc -- range of values of w and h
109
110     Return :
111     dA -- numpy matrix, gradients of the current layer (w_dA, h_dA, d, n)
112     dW -- numpy matrix, weights gradients (count, w_W, h_W, d)
113     db -- numpy matrix, biases gradients (1, 1, count, 1)
114     """
115
```

```
    lx, ly = rc
    w_dZ, h_dZ, count, n = dZ.shape
    _, w_W, h_W, _, _ = W.shape

    dA = np.zeros_like(A_p)
    dW = np.zeros_like(W)

    for k in range(count):
        for x in range(w_dZ):
            for y in range(h_dZ):
                w, h, dZ_n = lx[x], ly[y], dZ[x, y, k].reshape(1, 1, 1, n)
                dA[w:w+w_W, h:h+h_W] += W[k] * dZ_n
                dW[k] += np.sum(A_p[w:w+w_W, h:h+h_W] * dZ_n, axis=3,
                    keepdims=True)

    return dA, dW
```