



INSTITUT SUPÉRIEUR D'INFORMATIQUE DE MODÉLISATION ET DE LEURS APPLICATIONS



**Institut Supérieur
d'Informatique, de
Modélisation et de
leurs Applications**

Campus des Cézeaux
1 rue de la Chébarde
TSA 60125
CS 60026
63178 Aubière CEDEX

**Laboratoire d' Informatique
de Modélisation et d'
Optimisation des
Systèmes**

Campus des Cézeaux
1 rue de la Chébarde
TSA 60125
CS 60026
63178 Aubière CEDEX

Rapport Stage
Calcul et Modélisation Scientifiques

Méthodes de deep learning pour la classification et l'annotation de documents.

Kergann Le Cornec

Tuteur Limos : **Vincent Barra**
Tuteur Limos : **Jian-Jin LI**

Date : **4 Septembre 2017**

Remerciements

Ce projet m'a permis de pousser mes connaissances sur l'apprentissage profond ainsi qu'en imagerie. Je tiens à remercier monsieur Vincent Barra mon tuteur au Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes pour son accompagnement tout au long du stage ainsi que ses conseils pertinents.

Je remercie également Madame Jian-Jin LI, ma tutrice Isima, qui m'a aidé sur le rapport de stage ainsi qu'Alexandre Fabre pour l'aide et le modèle procuré pour la partie Reconnaissance Optique de Caractères.

Enfin, je remercie madame Murielle Mouzat pour ses nombreux conseils qui ont permis de mener à bien ce rapport de stage.

Résumé

Apprentissage Profond pour la classification et l'annotation de documents

Ce rapport présente des méthodes et algorithmes **d'apprentissage profond** (ou deep-learning) pour la classification et **l'annotation** de documents. Depuis qu'un tel algorithme a gagné au concours de classification d'images "Large Scale Visual Recognition Challenge" en 2012, les algorithmes n'ont pas arrêté de s'améliorer et le deep-learning s'installe maintenant dans notre vie de tous les jours. Que ce soit sur les réseaux sociaux, les moteurs de recherche, la bourse ou la météo, les algorithmes de deep-learning sont présents.

Nous avons travaillé en **Python**, avec le framework **Tensorflow** très utile pour le deep-learning. Le stage a pu se décomposer en quatre parties. Il fallait dans un premier temps s'imprégner des méthodes de deep learning, puis utiliser le "**Transfer learning**" sur le réseau AlexNet pour la classification de nos documents et arriver à plus de 75% de bonnes réponses. Par la suite nous avons essayé d'utiliser la **reconnaissance optique de caractère** (OCR) pour transformer nos images de document en fichiers textes. Cependant, le réseau choisit n'apprend pas bien et les résultats ne sont pas convaincants. Enfin nous avons utilisé des méthodes de **résumé automatique** pour annoter nos documents et arriver à des résultats plus que convenable.

Mots Clés : Apprentissage profond, Python, Tensorflow, Réseaux profonds, Transfer learning , OCR, annotation résumé ,automatique

Abstract

Deep Learning Networks for document Classification and Annotation

This paper presents **deep learning** methods and algorithms for image classification and annotation.

Since a deep learning algorithm won the "Large Scale Visual Recognition Challenge" in 2012, these types of algorithm did not stop improving. Nowadays, deep learning is all around us, in social networks, search engines, stock market and even weather.

We worked with the free language **Python** and the framework **Tensorflow**, which is a very powerful framework for **deep learning networks**.

We first learned more about deep learning methodes how to work with **Tensorflow**.

Then we fined tuned the AlexNet model to have a working image **classification** model that works about 75% of the time on documents over 16 differents classes.

After that we transformed our images to text using Optical Character Recognition **OCR** methodes. However, the choosen model doesn't learn properly and the results are not convincing.

To finally use text **summarization** to perform **document annotation** and to reach some decent results.

Key Words : Deep learning, Tensorflow, Python, Deep learning network, classification, OCR, summarization, document annotation

Table des matières

1	Apprentissage Profond	2
1.1	Machine Learning	2
1.2	Réseaux Neuronaux	3
1.3	Deep learning	6
1.3.1	Réseau neuronal de convolution	7
1.3.2	Explication des différentes couches	8
1.4	Entraînement du réseau	14
1.4.1	Rétro-propagation de l'erreur	15
1.4.2	Explication sur un exemple	16
2	Classification de Documents	18
2.1	Choix Des Outils de Travail	18
2.1.1	Python	18
2.1.2	Tensorflow	18
2.1.3	Base de Données	19
2.2	Transfer Learning	19
2.2.1	Explication	19
2.2.2	Extraction automatique de caractéristiques	20
2.2.3	Fine tuning	20
2.2.4	Bilan	20
2.3	Implémentation	21
2.4	Résultat	23
2.4.1	Extraction automatique de caractéristiques sur le modèle AlexNet	23
2.4.2	Fine tuning sur le modèle AlexNet	25
2.4.3	Extraction automatique de caractéristiques sur le modèle Inception	26
3	Reconnaissance Optique de Caractères	28
3.1	Détection de lignes	28
3.1.1	Binarisation	28
3.1.2	Segmentation	29
3.2	Détection de mots	31
3.2.1	Histogramme	31
3.2.2	The Iterative Self-Organizing Data Analysis Technique" (ISODATA)	32
3.3	Apprentissage de mots	33
3.3.1	Recurrent Neural Networks	33
3.3.2	Long Short-Term Memory	37
3.3.2.1	Présentation	37
3.3.2.2	Exemple pas à pas	38
3.3.3	Apprentissage	40
3.3.3.1	Générer des données d'apprentissage	40

3.3.3.2	Modèle CRNN	41
3.3.3.3	Resultats	43
4	Annotation d'un document	44
4.1	Modèle d'Attention	44
4.2	Résumé automatique d'un document	46
4.3	Le modèle "Sequence to Sequence"	47
4.3.1	Problèmes	48
4.3.2	Solution : Pointer-Generator Networks	48
4.3.3	Datasets	50
4.4	Word Embedding	51
4.5	Résultats	53
4.5.1	Exemple	53
4.5.2	Problèmes	54
Annexes		vi
Annexe 1		vi
1	Code source pour la création du réseau AlexNet	vi
2	Data	ix
3	Transfer Learning sur AlexNet	x
Annexe 2		xv
1	Modèle séquence to séquence	xv

Table des figures

1.1	Exemple simple de machine learning	2
1.2	Comment les ordinateurs comprennent les images	3
1.3	Fonctionnement d'un neurone	4
1.4	graphé de la fonction d'Heaviside	4
1.5	graphé de la fonction sigmoïde	5
1.6	Deep Learning plus puissant si nous avons beaucoup de données	7
1.7	Réseau convolutif inspirée du réseau LeNet	8
1.8	Convolution d'un filtre 5*5 sur une image	10
1.9	Image de départ	11
1.10	Filtre détectant les courbes	11
1.11	Multiplication d'une courbe de l'image par le filtre	12
1.12	Multiplication d'un endroit sans courbes de l'image par le filtre détectant des courbes	12
1.13	Principe d'un filtre de pooling	13
1.14	CNN LeNet	14
1.15	Retro-Propagation	16
2.1	Nombre de contributions	19
2.2	Bilan Transfer Learning	21
2.3	Architecture AlexNet	22
2.4	Précision d'entraînement sur le modèle AlexNet	24
2.5	Précision de test sur le modèle AlexNet	24
2.6	Précision d'entraînement sur le modèle AlexNet	25
2.7	Précision de test sur le modèle AlexNet	25
2.8	Précision d'entraînement sur le modèle Inception	26
2.9	Précision de test sur le modèle Inception	27
3.1	Dérivé de du filtre gaussien selon y	29
3.2	Segmentation : application de la dérivé du filtre de gaussien	30
3.3	Segmentation : détection de ligne	30
3.4	Détection des lignes sur un de nos documents	31
3.5	Histogramme des espaces	32
3.6	Détection des mots dans les lignes de nos documents	33
3.7	RNN plié	34
3.8	RNN déplié	34
3.9	Simple fonction	35
3.10	Calcule de l'erreur	35
3.11	Calcule du gradient	36
3.12	Fonctions plus complexes	37
3.13	L'idée derrière le LSTM	38

3.14	Les portes	38
3.15	Principe de la "forget gate"	39
3.16	Principe de "l'input gate"	39
3.17	Mise à jour du nouvel état	40
3.18	Principe de l'output gate	40
3.19	Génération d'images de mots	41
3.20	Génération d'image abîmée	41
3.21	L'erreur commise en fonction du temps	43
4.1	Model d'attention	44
4.2	A l'intérieur d'un model d'attention	45
4.3	Nous prenons en compte le vecteur du contexte	45
4.4	Calcule des poids avec une couche SoftMax	46
4.5	Moyenne pondérée pour calculer la sortie	46
4.6	Modèle "Seq to Seq avec attention	47
4.7	Réseau pointeur-générateur	49
4.8	Représentation des mots par des vecteurs	51
4.9	Matrice d'occurrence	52
4.10	Apprentissage de la matrice d'occurrence afin de représenter les mots	52
4.11	L'erreur (Sans unité) en fonction du temps (en heure)	53

Introduction

Durant les dernières décennies, nous avons observé une explosion dans le domaine de l'intelligence artificielle. Nous avons pu constater la naissance de véhicules sans conducteur, ou encore des programmes de reconnaissance de parole ou d'image de plus en plus performants. L'apprentissage automatique (sous toutes ses différentes formes) nous entoure. Nous l'utilisons sans nous en rendre compte dans notre vie de tous les jours, que ce soit avec une simple recherche web ou une connexion aux réseaux sociaux. Cela dit, la discipline est en fait relativement ancienne, elle voit ses origines dans les années 1950. Si le terme est aujourd'hui à la mode, c'est que les récents progrès technologiques, liés au Big Data, accélèrent le développement d'outils permettant l'industrialisation de la Data Science.

Dans les années 2000, nous constatons une grande amélioration des algorithmes d'apprentissage profond. Cela a donné naissance à toutes sortes d'algorithmes de reconnaissance et de classification performants. Le "deep learning" ou l'apprentissage profond, fait partie d'une famille de méthodes d'apprentissage automatique fondée sur l'apprentissage de modèles de données.

Durant ce stage, effectué au Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes (LIMOS) durant la période d'avril à août, nous nous sommes imprégnés des méthodes d'apprentissage profond et de l'implémentation en Python avec l'aide du framework Tensorflow. Nous avons classifié et annoté des documents de 16 classes différentes. Nous voulons automatiquement classer et résumer un ensemble de documents, nous allons donc présenter comment nous sommes passé d'une simple photo de document à l'annotation final.

Tout d'abord, nous commencerons par expliquer brièvement en quoi consiste l'apprentissage profond, en passant par l'explication succincte du fonctionnement des réseaux de neurones et du machine learning en général. Pour ensuite, découvrir comment nous avons classifié nos documents, en présentant le fine-tuning et le modèle AlexNet. Par la suite, le troisième chapitre présente les méthodes de reconnaissance optique de caractères utilisées, pour la transformation de nos images de documents en fichiers textes. Enfin, nous allons présenter comment avons résumé automatiquement ces textes pour pouvoir annoter nos documents.

1.1 Machine Learning

Nous allons rester sur des problèmes simples pour l'explication. Cependant, la raison pour laquelle le machine learning existe est qu'en réalité, les problèmes sont beaucoup plus compliqués. Ici, nous allons prendre en compte un problème à deux dimensions, mais les problèmes que traite le machine learning (par exemple le traitement d'images) sont souvent à des milliers de dimensions et utilise des fonctions complexes.

Dans l'exemple de la figure 1.1, des employés ont donné leur satisfaction sur une échelle de 0 à 100. Nous allons tenter de mettre ces chiffres en relation avec leur salaire.

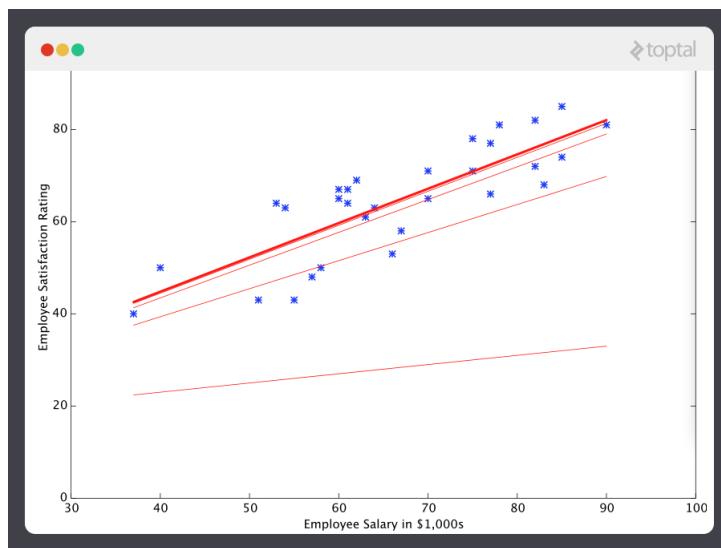


FIGURE 1.1 – Exemple simple de machine learning
[1]

Premièrement, nous pouvons noter que nos données (les points bleus) sont un peu bruités. Cependant, nous pouvons quand même remarquer que généralement, dans cette entreprise, plus le salaire est haut plus le niveau de satisfaction est haut. Le but est de créer un "programme" qui, si l'on donne en entrée un salaire ou un niveau de satisfaction, nous sort respectivement un niveau de satisfaction ou un salaire. L'ordinateur va apprendre sur des données, il va créer un modèle et va essayer de bien "prédir", de "deviner" les prochaines données. Sur la figure 1.1 nous voyons les différentes droites que l'ordinateur va construire au fur et à mesure que nous rajoutons des données. La première droite (celle du bas) est la droite d'initialisation avec ici des coefficients aléatoires. L'ordinateur va ensuite chercher les coefficients a et b (de la droite $ax+b$) qui vont permettre une bonne représentation du problème. Il va calculer l'erreur qu'il a commise puis ajuster les coefficients pour donner de meilleurs résultats. C'est dans le choix

du calcul de l'erreur que réside la partie "difficile" de l'algorithme, c'est le cœur du machine learning. Il y a beaucoup de possibilités, mais la plus répandue est le calcul de l'erreur avec la méthode des moindres carrés.

Nous remarquons bien que plus nous avons de données, plus la représentation va être précise et va nous permettre une meilleure prédition par la suite. Avec cette droite, si nous avons un salaire donné, l'ordinateur va prendre le point de la droite correspondant à l'abscisse et va lire sur l'ordonnée le niveau de satisfaction correspondant.

Pour nous, cette représentation (cette droite) va être insuffisante. En effet, ici, l'ordinateur doit seulement trouver deux coefficients pour obtenir la bonne droite, pour la reconnaissance d'image, c'est un peu différent.



FIGURE 1.2 – Comment les ordinateurs comprennent les images
[13]

L'ordinateur comprend une image en niveaux de gris comme une matrice de chiffres correspondant à l'intensité de gris du pixel en question (entre 0 et 255). 0 représente le pixel noir et 255 le pixel blanc.

Nous avons donc beaucoup plus d'entrées. Par exemple pour une image 400*400, nous avons déjà 160000 données et trois fois plus si nous travaillons en RGB (une matrice pour l'intensité du pixel rouge, une autre pour le vert et le bleu). C'est ici qu'interviennent les réseaux de neurones.

1.2 Réseaux Neuronaux

Les réseaux de neurones sont des systèmes qui imitent les neurones de notre cerveau. À force d'apprentissage, le système s'imprègne des conclusions à donner face à une situation nouvelle. Après avoir connu une période de doute (années 1990), les réseaux de neurones sont de nouveau très utilisés et nous allons dans ce rapport les utiliser pour la classification d'images. La reconnaissance d'image consiste à prendre une image en entrée du programme et deviner ce que représente l'image. Dans notre cas, nous avons un document en entrée, et nous devons déterminer s'il s'agit d'une lettre, d'un contrat, d'une facture ... Pour les humains, la reconnaissance est l'une des premières compétences que l'on apprend enfant et qui nous vient naturellement étant adulte. Nous allons reconnaître rapidement des modèles, les généraliser et les adapter à différents exemples, les réseaux neuronaux se basent sur cet a priori.

Définitions

- Un Neurone

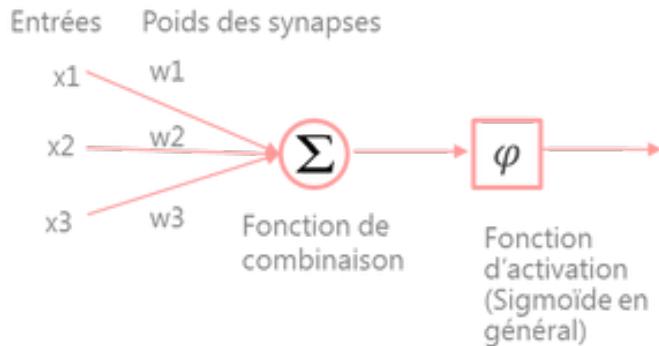


FIGURE 1.3 – Fonctionnement d'un neurone

Le neurone va prendre le rôle de notre droite dans la partie précédente. Il va nous permettre de prédire une réponse à un problème donné, grâce à l'étude des entrées. L'idée de base est que nous allons associer à chaque entrée un poids, selon son importance. Nous pouvons voir sur la figure 1.3 que chacune des n variables d'entrée x vont se voir attribuer un poids w lors de la phase d'apprentissage. Ce poids peut être vu comme la force de prédiction de la variable pour le problème $z = (w_1x_1, w_2x_2, \dots, w_nx_n)$

Notre vecteur z va ensuite passer dans une fonction dite "de combinaison", laquelle prend n variables en entrée, et n'en produit qu'une en sortie. Nous pouvons par exemple citer : la combinaison linéaire : $\sum_{i=1}^n z_i = z_1 + z_2 + \dots + z_n$ et la combinaison non-linéaire (notamment de type Gaussienne) : $y = f(z) = e^{-|z-m(z)|^2/2\sigma^2}$. Avec $m(z)$ le vecteur moyen du jeu de données et σ^2 la variance de notre distribution.

Une fois passé par notre fonction de combinaison, nous nous retrouvons avec une unique valeur, laquelle va parfois être soumise à une fonction "seuil". Il existe plusieurs, nous pouvons citer la fonction Heaviside, définie comme ceci : $H(x) = 1$ si $x \geq 0$ et 0 sinon. Son graphe est représenté figure 1.4.

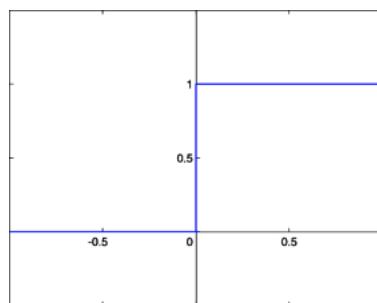


FIGURE 1.4 – Graphe de la fonction Heaviside

Nous pouvons encore citer la fonction sigmoïde qui présente l'avantage d'être dérivable (ce qui peut être utile par la suite) ainsi que de donner des valeurs intermédiaires (des réels compris entre 0 et 1) par opposition à la fonction de Heaviside qui elle renvoie soit 0 soit 1.

Elle est définie par $S(x) = \frac{1}{1+e^{-x}}$ et son graphe est représenté figure 1.5.

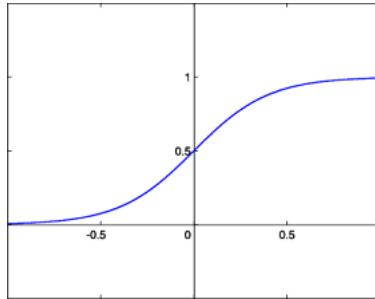


FIGURE 1.5 – Graphe de la fonction sigmoïde

La fonction sigmoïde présente l'avantage d'être dérivable (ce qui va être utile par la suite) ainsi que de donner des valeurs intermédiaires (des réels compris entre 0 et 1) par opposition à la fonction de Heaviside qui elle renvoie soit 0 soit 1.

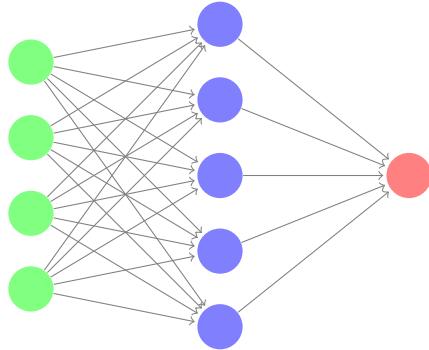
L'ordinateur peut faire varier le seuil et le poids (de la même manière que les coefficients a et b de l'exemple initial) pour trouver un bon modèle. Nous obtenons ainsi un perceptron, c'est-à-dire un réseau de neurones muni d'un seul neurone. Cependant, un neurone ne va pas suffire pour représenter des fonctions complexes (telles que celles de la reconnaissance d'image). C'est pourquoi nous avons besoin d'un réseau de neurones.

- Le réseau de neurones

Un réseau de neurones est composé de plusieurs neurones empilés. En associant des neurones comme cela, nous pouvons fabriquer des fonctions aussi complexes que l'on veut. L'idée est de stimuler beaucoup de neurones interconnectés qui vont communiquer dans un réseau, pour apprendre, reconnaître des motifs et prendre des décisions comme un humain. Un réseau peut avoir une dizaine à des milliers de neurones connectés, formant des couches. Les neurones auquel nous envoyons les données sont appelés neurone d'entrées. Pour la classification d'image, les données d'entrées vont être des parties importantes de l'image, par exemple, à un neurone nous pouvons associer l'information "de la présence de texte en haut", ou "une absence de texte en bas"... Le réseau va essayer d'apprendre avec les caractéristiques d'entrées pour reconnaître une image de ce type la prochaine fois. Il va modifier les poids des arêtes et les fonctions d'activation pour créer un réseau correspondant à notre modèle. L'entraînement du réseau consiste à trouver des poids synaptiques w_{ij} tels que la couche de sortie permette de classer avec précision les images d'un ensemble d'entraînement. On espère naturellement que le réseau de neurones présentera des capacités de généralisation sur des exemples qu'il n'a jamais rencontrés. Par exemple, si, pour une lettre il est important d'avoir du texte en haut à gauche (nom et coordonnées) il y aura un poids important sur la synapse sortant du neurone qui régit cette entrée.

Puis, une fois que le réseau a appris (sur un nombre important de données), nous allons tester l'image sur le réseau, il va regarder les parties importantes de l'image (haut, gauche, etc...). Puis les informations vont se propager dans le réseau de neurones (la majeure partie des réseaux ont entre chaque couche des connexions entre chaque neurone). Finalement, cela va arriver au neurone de sortie qui va nous donner un chiffre représentant la réponse attendue.

Entrées Neurones Sortie



L'erreur est souvent calculée avec la méthode des moindres carrés $E = (1/2) * (t - y)^2$, avec t la valeur que l'on veut atteindre et y la solution que l'on a obtenue. Nous allons minimiser l'erreur avec l'algorithme de descente de gradient. Les valeurs optimales de chaque poids sont celles atteintes dans le minimum global de la fonction qui donne les erreurs en fonction des entrées. Pendant la phase d'entraînement, les poids sont remis à jour, petit à petit, à chaque image d'entraînement.

La limite de cette méthode est qu'il faut que l'humain fasse le travail de sélectionner les bonnes caractéristiques de l'image. Ce travail est fait en amont du programme. C'est ici qu'intervient l'apprentissage profond (deep learning).

1.3 Deep learning

L'apprentissage profond (en anglais deep learning) consiste à donner l'image brute en entrée et grâce à divers traitements, le réseau va découvrir lui-même les caractéristiques importantes pour la classification. En 2012, grâce à la victoire d'un algorithme de deep learning au concours Large Scale Visual Recognition Challenge, il est revenu à la mode. Pourquoi ? Ce succès s'explique par le progrès des algorithmes, l'augmentation de la puissance de calcul et la sortie de bases de données immenses d'images labellisées par Imagenet. Comme nous montre la figure 1.6, le deep learning obtient des résultats bien meilleurs que les algorithmes plus anciens, dès lors que les jeux de données sont de tailles suffisamment importantes.

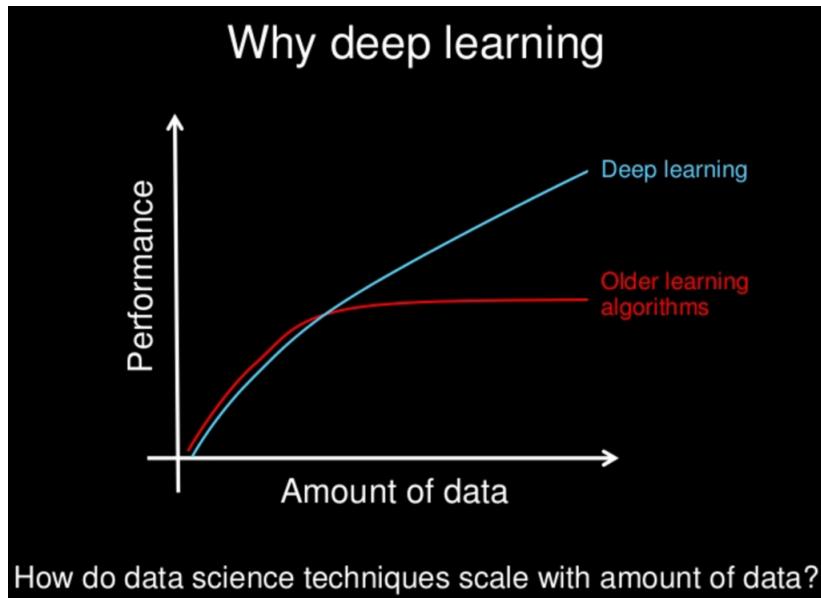


FIGURE 1.6 – Deep Learning plus puissant si nous avons beaucoup de données [4]

Pour l'instant il faut indiquer au réseau de neurones sur quelles parties de l'image il doit apprendre. Nous devons lui donner en entrée les caractéristiques importantes de l'image. Cependant, avec le deep learning et plus précisément les réseaux de convolution, le réseau va pouvoir trouver (et apprendre) les parties importantes de l'image (les caractéristiques importante pour la reconnaissance d'une lettre par exemple)

En effet, si nous donnons l'image brute au réseau de convolution, il va apprendre et trouver les caractéristiques essentielles pour la classification de l'image. Nous avons besoin d'un nombre important d'images pour apprendre précisément et d'une puissance de calcul assez élevée. Nous pouvons donc, comme dans les réseaux de neurones, empiler les couches de convolutions pour avoir un meilleur apprentissage des caractérisantes importantes de l'image. Cependant, il ne faut pas le faire n'importe comment car plus nous mettons de couches de neurones entre les entrées et la sortie, plus nous avons besoin de calculs..

Les algorithmes de deep learning donnent aujourd'hui les meilleures solutions à beaucoup de problèmes, dans beaucoup de domaines (reconnaissance d'image, reconnaissance vocale ...)

1.3.1 Réseau neuronal de convolution

En apprentissage automatique, un réseau de neurones de convolution (ou réseau de neurones à convolution, ou CNN) est un type de réseau de neurones artificiels reposant sur l'apprentissage profond. Il va appliquer des filtres sur l'image brute afin de trouver des caractéristiques importantes (courbes, coins, ...) pouvant être utile pour la classification. Les réseaux de neurones convolutifs sont l'outil de choix du Data Scientist pour la classification d'images. Ce sont des algorithmes phares du deep learning, objets d'intenses recherches dont les résultats sont aujourd'hui impressionnantes. Les réseaux de neurones convolutifs sont à ce jour les modèles les plus performants pour classer des images.

Nous allons d'abord fixer l'architecture du réseau, c'est-à-dire le nombre de couches, leurs tailles, leurs fonctions d'activations. L'entraînement consiste alors à optimiser les coefficients des poids du réseau et des filtres pour minimiser l'erreur de classification en sortie. Cet entraînement peut prendre plusieurs semaines pour les meilleurs CNN, avec de nombreux GPU travaillant sur des centaines de milliers d'images annotées. Des équipes de recherche se spécialisent dans

l'amélioration des CNN. Elles publient leurs innovations techniques, ainsi que le détail des réseaux entraînés sur des bases de données de référence. Le challenge ImageNet (ILSVRC) fournit par exemple 1.2 million d'images classées en 1000 catégories.

1.3.2 Explication des différentes couches

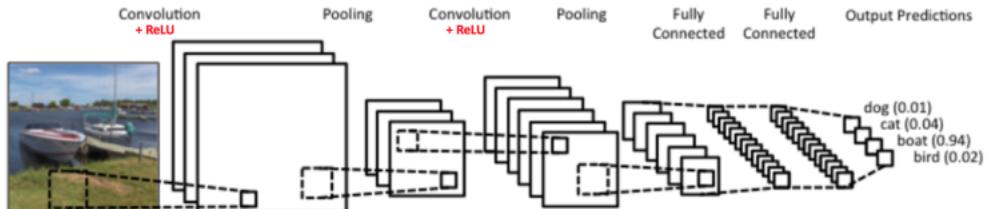


FIGURE 1.7 – Réseau convolutif inspirée du réseau LeNet
[15]

La figure 1.7 est une représentation d'un réseau inspiré du réseau LeNet qui est l'un des premiers CNN servant à la reconnaissance de chiffres manuscrits. Il est ici utilisé pour classifier des images en quatre catégories : chien, chat, oiseau et bateau. La sortie du réseau est un vecteur de taille quatre représentant une distribution de probabilité. Par exemple, sur la figure 1.5, on obtient une haute probabilité pour le bateau et trois faibles probabilités pour les autres.

Nous voulons que le réseau différencie les images en trouvant les caractéristiques uniques du chien, du chat, de l'oiseau et du bateau. Il doit trouver les différences entre un oiseau et un bateau par exemple. Quand nous regardons une photo d'un oiseau nous pouvons trouver des caractéristiques remarquables (par exemple la présence d'ailes). En pratique, l'ordinateur va classifier les images en cherchant des caractéristiques de bas niveau comme les bords et les courbures dans l'image. Puis en utilisant ces caractéristiques en passant par les couches de convolution, il va trouver des concepts plus abstraits.

Nous remarquons quatre opérations majeures dans le réseau : la convolution, la Relu, le pooling et la classification. Ces opérations sont la base de tous les Cnns et nous allons donc les détailler.

- La Convolution

La première couche d'un réseau convolutif (CNN) est toujours la convolution. La partie convolutive du réseau fonctionne comme un extracteur de caractéristiques des images. Une image est passée à travers une succession de filtres, créant de nouvelles images appelées cartes de convolutions. Un filtre convolutif peut être apparenté à un noyau en imagerie, c'est une petite matrice utilisée pour détecter les bords dans l'image, pour lui appliquer un flou,... Nous allons donc appliquer une convolution entre un noyau et l'image. La différence est qu'ici, les noyaux changent au fur et à mesure de l'apprentissage.

Par exemple nous pouvons effectuer la convolution entre ces deux matrices.

Une partie de notre image (en noir et blanc pour l'exemple) :

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

et un filtre :

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}.$$

Pour un pixel donné, par exemple

Le filtre calcule successivement la convolution, pour chacun des pixels de l'image. Pour chaque pixel, que nous appellerons « pixel initial », il multiplie la valeur de ce pixel et de chacun des 8 pixels qui l'entourent par la valeur correspondante dans le noyau. Il additionne ensuite l'ensemble des résultats et le pixel initial prend alors la valeur du résultat final.

Par exemple si nous appliquons ce filtre sur le pixel (2,2) ("initial") de la matrice de départ. Nous aurons $1 * 1 + 1 * 0 + 1 * 1 + 0 * 1 + 1 * 1 + 10 * 0 + 0 * 1 + 0 * 1 = 4$, que nous allons inscrire sur la matrice final.

Nous allons appliquer ce filtre à tous les pixels de l'image où nous pouvons l'appliquer. Nous ne pouvons pas l'appliquer sur les bords de l'image, le filtre prend un voisinage de $3 * 3$, nous avons donc besoin de huit voisins. Après application de ce filtre, nous aurons :

$$\begin{pmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{pmatrix}.$$

Puis nous allons répéter ce processus sur chaque partie de l'image. Nous allons ensuite bouger le filtre d'un pixel à droite puis quand nous avons atteint le bord de l'image d'un pixel vers le bas.

La matrice résultat s'appelle la carte des caractéristiques.

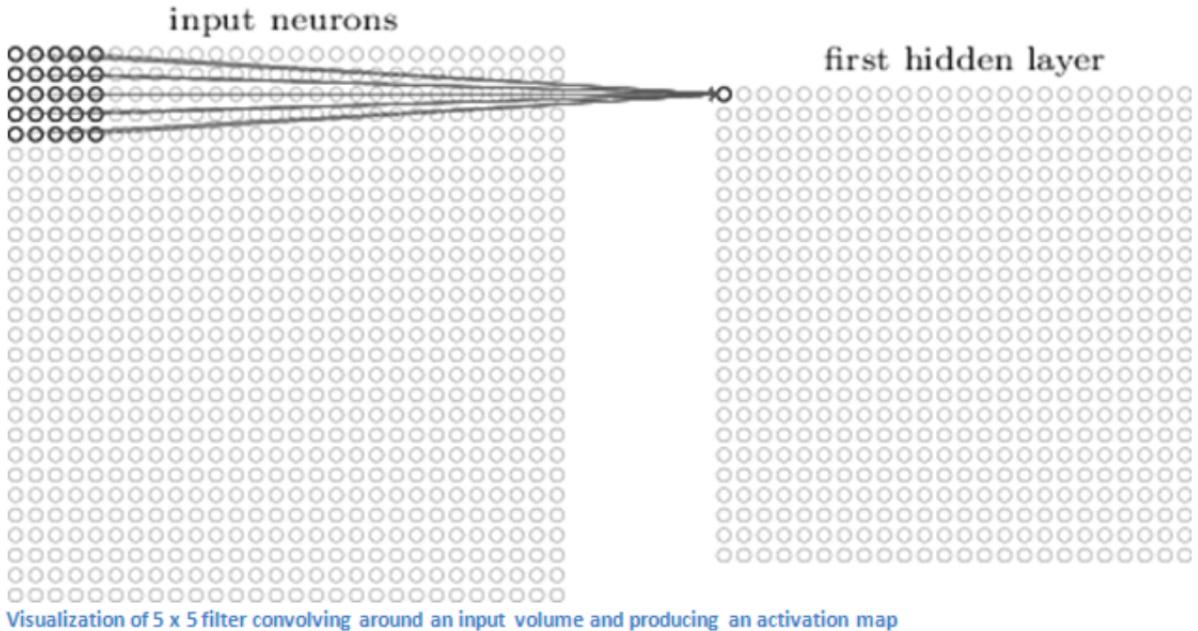


FIGURE 1.8 – Convolution d'un filtre 5*5 sur une image
[3]

Il y a bien entendu un panel de différents filtres que l'on peut appliquer à l'image.

- Filtre créateur de flou

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

- Filtre détecteur de bords

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

- Filtre augmentant le contraste

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -5 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Dans la pratique, le CNN apprend les valeurs de ces filtres tout seul pendant l'entraînement (même si nous devons spécifier quelques éléments, le nombre de filtres, la taille ...). Plus on applique de filtres sur l'image, plus on a de caractéristiques extraites et meilleur sera notre réseau, cependant on doit trouver un équilibre entre la puissance de calcul et la performance.

Voyons ce que la convolution représente à haut niveau.

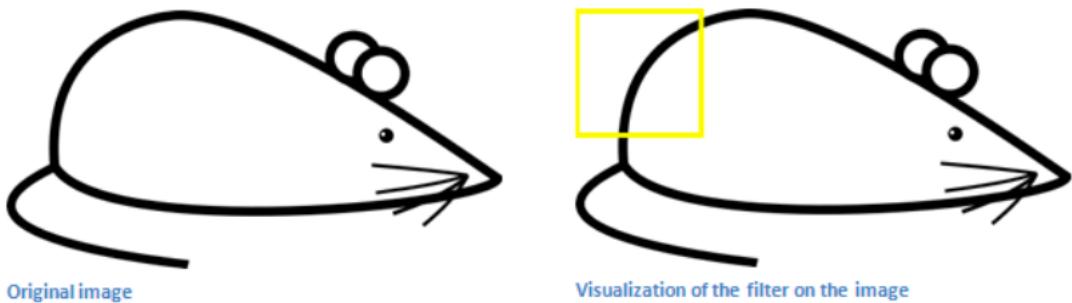


FIGURE 1.9 – Image de départ
[10]

Nous allons essayer de détecter les courbes sur l'image de la figure 1.9

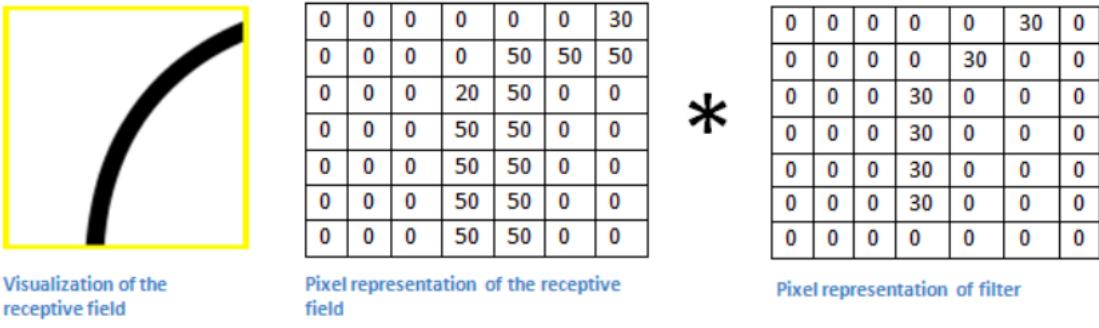
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



FIGURE 1.10 – Filtre détectant les courbes
[10]

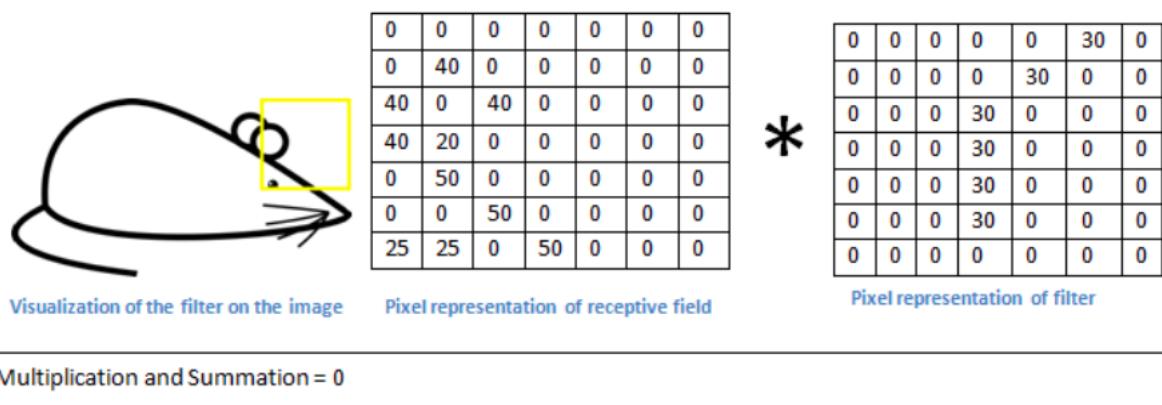
Nous avons sur la figure 1.10 u filtre qui a appris à détecter une courbe dans l'image.



$$\text{Multiplication and Summation} = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 \text{ (A large number!)}$$

FIGURE 1.11 – Multiplication d'une courbe de l'image par le filtre [10]

Nous voyons sur la figure 1.11 que la multiplication d'une courbe de l'image par le filtre donne une valeur élevée. Cela indique qu'à cet endroit de l'image, il y a une sûrement une courbe importante pour la classification.



$$\text{Multiplication and Summation} = 0$$

FIGURE 1.12 – Multiplication d'un endroit sans courbes de l'image par le filtre détectant des courbes [10]

la partie de l'image considérée 1.12 ne comporte pas de courbe, la valeur renvoyée va être plus petite (ici 0), car elle ne va pas être importante pour classifier.

Fréquemment, après les filtres de convolution, nous appliquons une couche de ReLU.

- Rectified Linear Unit (ReLU)

Après application des filtres de convolution, comme nous pouvons appliquer des filtres avec des poids négatifs, nous pouvons avoir des pixels négatifs. Pour représenter la réalité, nous remplaçons les pixels négatif par 0 : $\text{Output} = \text{Max}(0, \text{Input})$.

En général, l'application d'une couche ReLU après chaque convolution donne de meilleur résultats

- Le Pooling

Le pooling (spatial pooling) va permettre de réduire les dimensions de la carte de caractéristiques tout en gardant les informations importantes. La carte sera ainsi plus facile à utiliser.

La méthode de pooling la plus populaire est le max pooling dont nous allons voir un exemple. Nous allons prendre une fenêtre (2×2 , 3×3 , ...) et nous allons lui appliquer une opération comme ici le maximum ou la moyenne, la somme ...

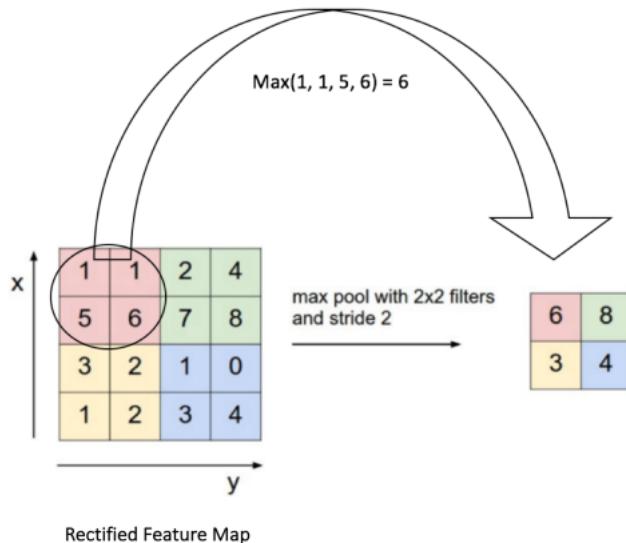


FIGURE 1.13 – Principe d'un filtre de pooling
[15]

Dans l'exemple figure 1.13 nous passons d'une image 4×4 à une image 2×2 . Nous allons progressivement, après chaque convolution réduire la taille de la représentation.

L'intuition se cachant derrière cette couche est qu'une fois que l'on a trouvé une caractéristique importante dans l'image (emplacement où il y a une valeur d'activation élevée), son emplacement dans l'image n'est pas très important. Il est plus important de garder son emplacement relatif à d'autres caractéristiques

Les dimensions de la carte des caractéristiques importantes vont être radicalement réduites, cela va permettre de réduire le nombre de paramètres pour éviter le phénomène de sur-apprentissage avoir apprentissage plus rapide.

Le pooling permet également d'introduire des invariances. Par exemple, l'invariance par translation : si nous translatons notre image, les prédictions serons toujours les mêmes. Prenons un exemple simple, un réseau détectant les têtes de chiens. La carte des caractéristiques (juste avant la classification) sera les probabilités de la présence d'un chat pour chaque pixel. Supposons que l'activation d'un pixel écrase toutes les autres. Peu importe la translation que nous faisons, si la très forte activation reste dans la carte des caractéristiques le chat sera toujours détecté. Par la suite nous pouvons généraliser, la localisation exacte des caractéristiques n'est pas importante comme le maxpool prendra la région avec la plus grande valeur d'activation.

- Local Response Normalization (LRN)

En neurobiologie, il y a un concept appelé l'inhibition latérale. Quand un neurone est excité, il a la capacité d'atténuer ses neurones voisins. Le but pour nous, est d'augmenter

le contraste de l'image et obtenir des maximums locaux. Les LRN permettent de diminuer des réponses quand tout le voisinage est grand et d'amplifier les pixels forts quand leurs voisinages sont petits. Cela va permettre de créer un meilleur contraste dans la carte d'activation.

Pour chaque pixel i, nous appliquons cette transformation : $xi = \frac{xi}{(k + (\alpha \sum_j x_j^2))^\beta}$

Avec k , α et β constantes (durant l'apprentissage ils sont bien entendu modifiés).

- La Classification

La dernière couche, appelé en anglais fully connected, est une des plus importantes. Elle consiste en un réseau de neurones classique, avec en entrées les sorties données par les convolutions (relus et poolings) précédentes. Nous appelons cela des couches fully connected (fc) car tous les neurones sont connectés entre eux.

Cette couche va calculer le produit entre les différents poids et les différentes entrées pour avoir au final les probabilités des différentes classes.

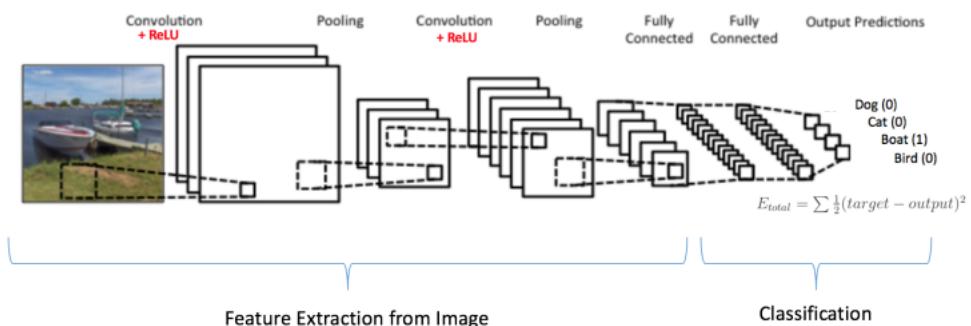


FIGURE 1.14 – CNN LeNet
[15]

Puis à chaque image il va calculer l'erreur totale dans l'image, ensuite il changera les poids des filtres avec des algorithmes. Les poids vont être ajustés selon leurs importances.

- Le Dropout

Dans les réseaux de neurones, les couches "FC" prennent une partie conséquente de la mémoire du CNN. Si tous les neurones travaillent au même moment, il est possible qu'un problème de mémoire arrive. Cela veut dire qu'il y a trop de connexions dans notre réseau et que l'apprentissage ne se passe pas correctement. Cela va ralentir les traitements dans le réseau. Pour empêcher cela d'arriver, nous utilisons la méthode du dropout, qui va "éteindre" des neurones aléatoirement (souvent un sur deux) dans les couches FC. Ainsi avec moins de neurones, le réseau va apprendre plus rapidement.

1.4 Entrainement du réseau

Le but de cet entraînement est de permettre au réseau de neurones "d'apprendre" à partir des exemples. On va pour cela utiliser l'algorithme de rétro-propagation de l'erreur.

1.4.1 Rétro-propagation de l'erreur

La rétro-propagation de l'erreur est une manière d'entraîner un réseau de neurones.

Comme nous avons vu dans le chapitre un, à chaque arrête du réseau de neurones est associé un poids. Le but de cet algorithme est d'optimiser les poids des arêtes, pour que le réseau remarque par lui-même les parties importantes de l'image. Les poids dans le réseau de neurones sont aux préalable initialisés avec des valeurs aléatoires, ou, si l'on fait du transfer learning, des poids déjà entraînés sur un autre réseau (cf. chapitre 4). Comme nous connaissons les résultats que nous devrions avoir nous pouvons calculer l'erreur à chaque itération et corriger selon le résultat.

C'est une méthode d'entraînement supervisée, cela veut dire que le réseau va apprendre sur des données labellisées. L'algorithme va en quelque sorte apprendre de ses erreurs et corriger, petit à petit, les poids du réseau de neurones à chaque itération pour avoir un plus faible pourcentage d'erreur à chaque fois. Le processus est répété jusqu'à ce que l'erreur atteigne un seuil fixé par l'utilisateur.

L'algorithme est une descente de gradient sur la fonction d'erreur. Nous comparons la prédiction avec le résultat que nous voulions avoir, puis nous allons calculer le gradient de l'erreur en prenant en compte les poids du réseau. Cela va nous donner la direction à prendre (comment changer les poids) pour que l'erreur devienne plus petite.

- Exemple de fonction d'erreur

Soit \vec{y} et \vec{y}' deux vecteurs, un choix commun de la fonction erreur est le carré de la distance euclidienne

$$E(y, y') = \frac{1}{2} \|y - y'\|^2$$

Cette fonction peut se généraliser sur n exemples comme cela : $E = \frac{1}{2n} \sum_x \|(y(x) - y'(x))\|^2$

Nous pouvons modéliser cela sous forme mathématique. l'algorithme se présente comme ceci : soit un échantillon \vec{x} que nous mettons à l'entrée du réseau de neurones et soit \vec{t} la sortie recherchée pour cet échantillon. nous allons propager le signal en avant dans les couches du réseau de neurones : $x_k^{(n-1)} \mapsto x_j^{(n)}$.

Comme nous l'avons vu dans le chapitre I, la propagation vers l'avant se calcule à l'aide de la fonction de seuil g , de la fonction de combinaison h (souvent un produit scalaire entre les poids et les entrées du neurone) et des poids synaptiques \vec{w}_{jk} entre le neurone $x_k^{(n-1)}$ et le neurone $x_j^{(n)}$ (comme nous l'avons vu dans le chapitre I).

\vec{w}_{jk} indique un poids de k vers j .

Nous pouvons modéliser la propagation comme cela :

$$x_j^{(n)} = g^{(n)}(h_j^{(n)}) = g^{(n)}\left(\sum_k w_{jk}^{(n)} x_k^{(n-1)}\right)$$

Lorsque la propagation vers l'avant est terminée, nous obtenons à la sortie le résultat \vec{y} .

Nous allons ici calculer l'erreur entre la sortie donnée par le réseau \vec{y} et le vecteur \vec{t} désiré à la sortie pour cet échantillon. Pour chaque neurone i dans la couche de sortie, nous calculons :

$$e_i^{sortie} = g'(h_i^{sortie})[t_i - y_i]$$

(g' est la dérivée de g)

Puis nous allons propager l'erreur vers l'arrière

$$e_i^{(n)} \mapsto e_j^{(n-1)}$$

:

$$e_j^{(n-1)} = g'^{(n-1)}(h_j^{(n-1)}) \sum_i w_{ij} e_i^{(n)}$$

note :

$$e_j^{(n)} = \sum_i [t_i - y_i] \frac{\partial y_i}{\partial h_j^{(n)}}$$

En définissant la suite des $e_i^{(n)}$ comme nous l'avons fait, on peut facilement obtenir la dérivée de l'erreur par rapport aux poids synaptiques d'un neurone à distance $n - l$ de la sortie. On met à jour les poids dans toutes les couches

$$w_{ij}^{(n)} = w_{ij}^{(n)} + \lambda e_i^{(n)} x_j^{(n-1)}$$

où λ représente le taux d'apprentissage (de faible magnitude et inférieur à 1.0)

Une fois que l'algorithme est terminé, et que le réseau a appris sur plusieurs données, on peut considérer que le réseau a appris et est prêt à travailler sur des nouvelles données.

1.4.2 Explication sur un exemple

Pour l'explication, on considère le réseau simple figure 1.15 :

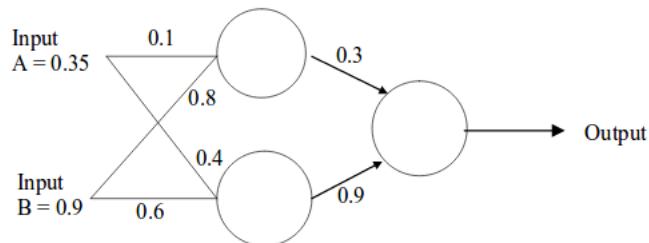


FIGURE 1.15 – Retro-Propagation

Nous prenons comme fonction d'activation la fonction sigmoid $g(x) = 1/(1 + e^{-x})$ pour tout réel x et un calcul d'erreur $\delta = out(1 - out)(Target - out)$

Avec out la sortie donnée par le réseau et $Target$ la valeur que nous voulons atteindre.

Le $out(1 - out)$ est nécessaire car nous utilisons la fonction d'activation Sigmoid. Sinon nous pourrions simplement calculer $(Target - out)$.

Propagation avant

Pour le neurone du haut : $InputH = (0.35 * 0.1) + (0.9 * 0.8) = 0.755$

Puis $Out = g(Input) = 1/(1 + e^{-0.755}) = 0.68$.

Pour celui du milieu : $InputM = (0.9 * 0.6) + (0.35 * 0.4) = 0.68$ et $Out = 0.6637$.

Pour le dernier neurone : $InputD = (0.3 * 0.68) + (0.9 * 0.6637) = 0.80133$ et $Out = 0.69$.

Propagation arrière

L'erreur de notre résultat final $\delta = (t - o)(1 - o)o = (0.5 - 0.69)(1 - 0.69)0.69 = -0.0406$
 Calcul des nouveaux poids avec la formule : $W_{new}AB = WAB + (ErrorB * OutputA)$

$$w1+ = w1 + (\delta * input) = 0.3 + (-0.0406 * 0.68) = 0.272392$$

$$w2+ = w2 + (\delta * input) = 0.9 + (-0.0406 * 0.6637) = 0.87305$$

L'erreur pour les couches du milieu :

$$\delta1 = \delta * w1 = -0.0406 * 0.272392 * (1 - o)o = -2.406 \times 10^{-3}$$

$$\delta2 = \delta * w2 = -0.0406 * 0.87305 * (1 - o)o = -7.916 \times 10^{-3}$$

Calcul des nouveaux poids :

$$w3+ = 0.1 + (-2.406 \times 10^{-3} * 0.35) = 0.09916. w4+ = 0.8 + (-2.406 \times 10^{-3} * 0.9) = 0.7978.$$

$$w5+ = 0.4 + (-7.916 \times 10^{-3} * 0.35) = 0.3972. w6+ = 0.6 + (-7.916 \times 10^{-3} * 0.9) = 0.5928.$$

L'erreur était de 0.19, maintenant elle est de 0.18205, elle a bien été réduite.

Classification de Documents

Nous allons, dans ce chapitre, créer un réseau profond nous permettant de classifier des documents sur un panel de 16 classes différentes.

Nous avons utilisé la banque de données de document labellisés "Tobacco" qui est composé de 300000 documents annotés sur 16 classes différentes, à savoir : 'letter', 'form', 'email', 'handwritten', 'advertisement', 'scientific report', 'scientific publication', 'specification', 'file folder', 'new article', 'budget', 'invoice', 'presentation', 'questionnaire', 'resume' et des 'memos'

2.1 Choix Des Outils de Travail

2.1.1 Python

Python est un langage de programmation objet, interprété et multi-plateformes. Il favorise la programmation impérative structurée, fonctionnelle et orientée objet. Nous avons choisi de travailler sur Python, car c'est un langage libre possédant une abondance de librairies d'imagerie et de machine-learning. Ainsi qu'une bonne documentation et une communauté très active. C'est un langage de haut niveau, facile à prendre en main et adapté pour obtenir des résultats rapidement.

2.1.2 Tensorflow

Tensorflow a été créée par l'organisation de recherche de Google : "Brain's Team". Le code source a été ouvert le 9 novembre 2015 et est en développement continu depuis (la toute dernière version est sortie le 30 juin 2017). Ce framework est écrit en python et C++. Nous avons choisi de travailler sur Tensorflow pour plusieurs raisons :

- On remarque sur la figure 2.1 que Tensorflow a la plus grande communauté et la plus active du moment

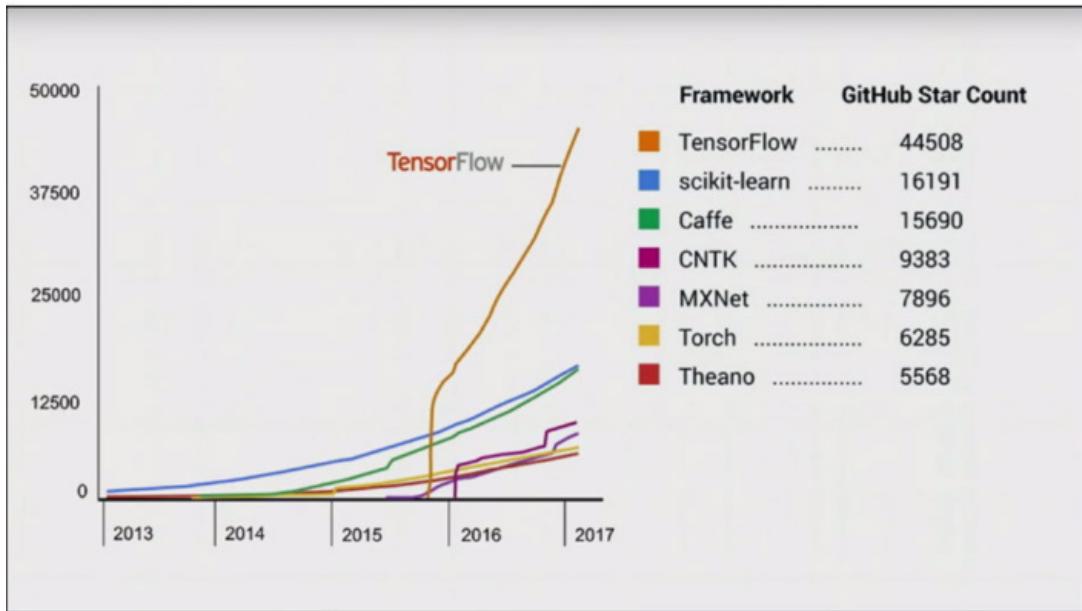


FIGURE 2.1 – Nombre de contributions
[10]

- La documentation est donc riche et les problèmes corrigés rapidement.
- Tensorflow est adapté à toutes les plates-formes, depuis les ordinateurs multi-GPUs avec l'utilisation de CUDA aux portables muni de simple CPU.
- L'outil de visualisation tensorboard est très pratique pour voir si notre réseau est correct.

2.1.3 Base de Données

Nous avons tout au long du projet utilisé deux bases de données. La base de données Hierarchical Data Format (HDF) et la base de données Lightning Memory-Mapped Database (LMDB), chacune ayant ses avantages et ses inconvénients. Si nous travaillons avec beaucoup de données, normalement la base de données la plus adaptée est la LMDB. Les deux bases de données sont très utilisées en machine learning, car elles sont très pratiques. De notre côté nous avons plus utilisé la LMDB.

2.2 Transfer Learning

2.2.1 Explication

Comme nous l'avons vu, le deep learning nécessite beaucoup d'images et de temps, de l'ordre de plusieurs semaines de calculs pour des réseaux compliqués avec des machines possédant plusieurs GPUs. Il est pourtant possible de créer des modèles performants avec peu d'images et peu de capacités de calcul par des techniques de transfer learning. Ces techniques sont ainsi nommées, car elles exploitent la connaissance acquise sur un problème de classification générale pour l'appliquer de nouveau à un problème particulier. Il existe deux stratégies de transfer learning : l'extraction automatique de caractéristiques ou affiner le réseau.

2.2.2 Extraction automatique de caractéristiques

L'extraction automatique de caractéristiques exploite uniquement la partie convolutive d'un réseau pré-entraîné. Elle l'utilise comme extracteur de caractéristiques des images.

À noter que cela fonctionne d'autant mieux si l'on travaille sur un problème proche du problème initial, sur lequel le réseau a été entraîné. Lorsque le problème est très similaire, nous pouvons utiliser des poids déjà entraînés pour nous permettre de gagner du temps. Dans cette stratégie nous allons enlever la dernière couche du modèle déjà entraîné, nous fixons les poids des couches restantes et nous faisons un simple réseau de neurones (machine learning) à partir des résultats des couches du réseau pré-entraîné. Chaque image de notre ensemble d'entraînement est ainsi transformée en un vecteur de caractéristiques, qui est utilisé pour entraîner un nouveau classificateur.

Cette méthode présente de nombreux intérêts pratiques. Tout d'abord, l'image est transformée en un vecteur de petites dimensions, qui extrait des caractéristiques pertinentes, cela réduit la dimension du problème. De plus, nous pouvons utiliser ce que nous voulons comme classificateur final. Enfin, l'extraction de caractéristiques permet d'apprendre beaucoup plus rapidement car tout les poids des couches de convolution sont gelés.

2.2.3 Fine tuning

Dans cette stratégie, nous affinons le réseau déjà entraîné sur les nouvelles données en continuant la rétro-propagation du gradient. On peut soit le faire sur tout le réseau (dans ce cas, si les poids trouvés auparavant ne correspondaient pas, le temps gagné est très petit), soit fixer quelques poids ou même quelques couches. L'intérêt est double : on utilise une architecture optimisée (faite par des spécialistes) et l'on profite des capacités d'extraction de caractéristiques apprises sur un jeu de données de qualité. Le fine tuning sur des images, consiste en quelque sorte à prendre un système déjà entraîné sur une tâche de classification pour le raffiner sur une tâche similaire.

Le réseau pré-entraîné possède déjà des coefficients optimisés avec soin sur un grand jeu de données. Il s'agit de les modifier faiblement à chaque itération, pour s'adapter en douceur au nouveau problème, sans écraser agressivement la connaissance déjà acquise.

Pour l'entraînement, il est possible de geler les couches initiales du réseau de neurones et d'adapter seulement les couches finales pour le nouveau problème de classification. Geler toutes les couches convolutives correspondent à la première méthode présentée, avec comme classificateur final un réseau de neurones.

2.2.4 Bilan

L'idée du transfer learning peut se résumer via la figure 2.2.

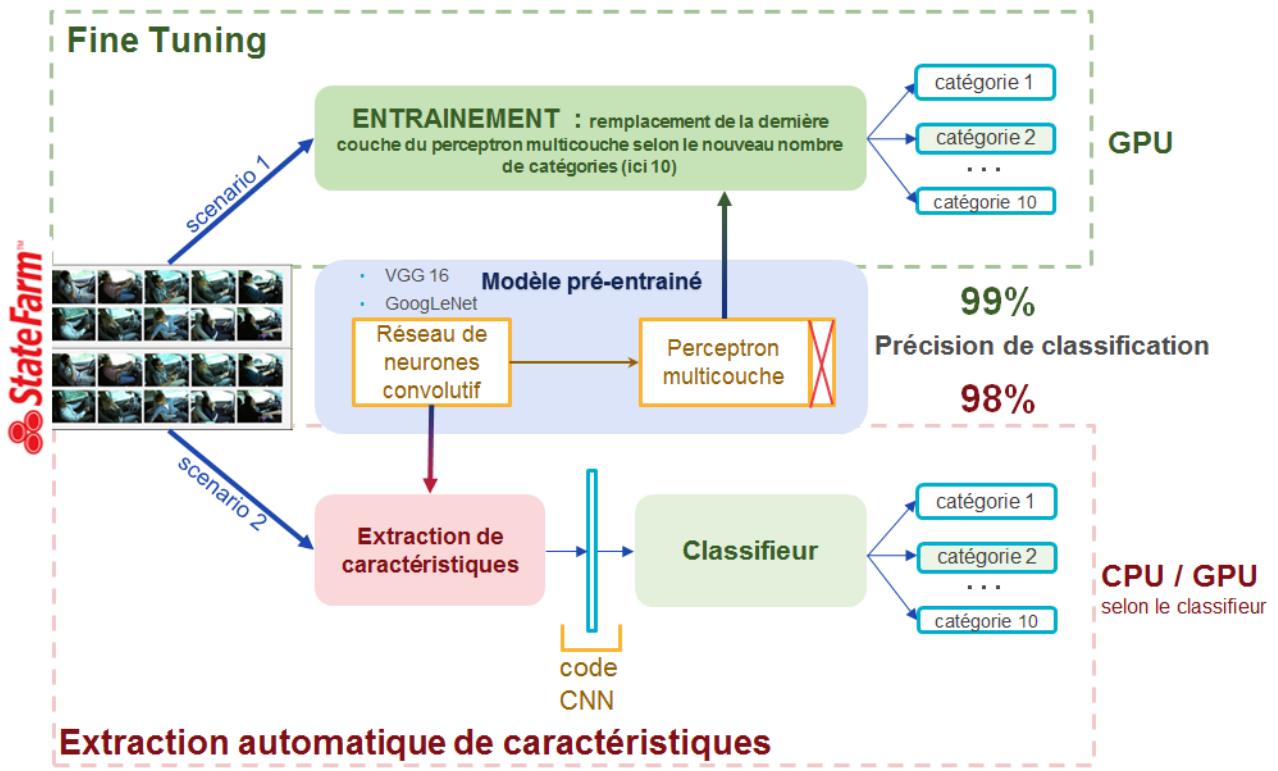


FIGURE 2.2 – Bilan Transfer Learning
[7]

Ce problème est à dix classes (le but est de remarquer si le conducteur est attentif, ou au téléphone, ou en train de boire ...). Il faut donc changer le classificateur avec dix classes. Dans la figure, le perceptron multicouche est, ce que nous avons appelé le réseau de neurones. C'est ici la partie qui va classifier les images en s'aidant des caractéristiques importantes trouvées. Dans leurs exemples, les auteurs ont obtenus 99% de bonnes réponses avec le fine tuning et 98% avec l'extraction de caractéristiques.

2.3 Implémentation

Nous avons choisi d'appliquer les méthodes de transfer learning et de fine tuning sur le modèle AlexNet en tensorflow. Alexnet est un réseaux CNN qui était codé en CUDA à la base. Comme notre tensorflow GPU utilise CUDA, le modèle est approprié. Il a gagné la compétition d'Imagenet, le "Large Scale Visual Recognition Challenge" en 2012, en classifiant des images sur 1000 classes différentes. Il a été crée par le groupe SuperVision, constitué de Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever et est de nos jours encore très utilisé. Son architecture est présentée 2.3

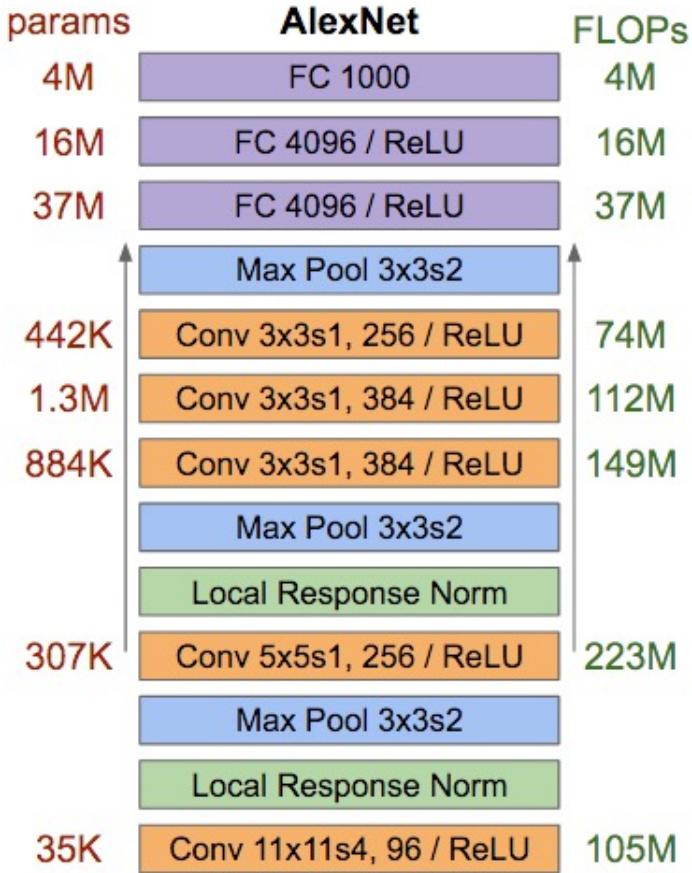


FIGURE 2.3 – Architecture AlexNet
[12]

Ci-dessous, nous avons codé le modèle Alexnet en Tensorflow. L'intégralité est en annexe.

```

1 # First layer: Conv (w ReLu) -> Pool -> Lrn
2 conv1 = conv(self.X, 11, 11, 96, 4, 4, padding = 'VALID', name = 'conv1')
3 norm1 = lrn(pool1, 2, 2e-05, 0.75, name = 'norm1')
4 pool1 = max_pool(conv1, 3, 3, 2, 2, padding = 'VALID', name = 'pool1')
5
6
7 # Second layer: Conv (w ReLu) -> Pool -> Lrn with 2 groups
8 conv2 = conv(norm1, 5, 5, 256, 1, 1, groups = 2, name = 'conv2')
9 norm2 = lrn(pool2, 2, 2e-05, 0.75, name = 'norm2')
10 pool2 = max_pool(conv2, 3, 3, 2, 2, padding = 'VALID', name = 'pool2')
11
12
13 # Third layer: Conv (w ReLu)
14 conv3 = conv(norm2, 3, 3, 384, 1, 1, name = 'conv3')
15
16 # Fourth layer: Conv (w ReLu) splitted into two groups
17 conv4 = conv(conv3, 3, 3, 384, 1, 1, groups = 2, name = 'conv4')
18
19 # Fifth layer: Conv (w ReLu) -> Pool splitted into two groups
20 conv5 = conv(conv4, 3, 3, 256, 1, 1, groups = 2, name = 'conv5')
21 pool5 = max_pool(conv5, 3, 3, 2, 2, padding = 'VALID', name = 'pool5')
22

```

```

23 # sixth layer: Flatten -> FC (w ReLu) -> Dropout
24 flattened = tf.reshape(pool5, [-1, 6*6*256])
25 fc6 = fc(flattened, 6*6*256, 4096, name='fc6')
26 dropout6 = dropout(fc6, self.KEEP_PROB)
27
28 # seventh layer: FC (w ReLu) -> Dropout
29 fc7 = fc(dropout6, 4096, 4096, name = 'fc7')
30 dropout7 = dropout(fc7, self.KEEP_PROB)
31
32 # eighth layer: FC
33 self.fc8 = fc(dropout7, 4096, self.NUM_CLASSES, relu = False, name='fc8')

```

Il est important de remarquer que les fonctions conv, fc, lrn , max pool sont des fonctions que nous avons créées mais qui utilisent des fonctions de base de Tensorflow.

La fonction conv est créée avec la méthode de Tensorflow "conv2d", permettant de créer une couche de convolution

```
1 tf.nn.conv2d(i, k, strides = [1, stride_y, stride_x, 1], padding = padding)
```

le Maxpooling est créée avec la fonction max pool

```

1 tf.nn.max_pool(x, ksize=[1, filter_height, filter_width, 1],
2                 strides = [1, stride_y, stride_x, 1],
3                 padding = padding, name = name)

```

Le couche local response normalization est créée avec la fonction local response normalization

```

1 tf.nn.local_response_normalization(x, depth_radius = radius, alpha = alpha,
2                                     beta = beta, bias = bias, name = name)

```

La couche fully connected est créée avec une multiplication de matrices entre les poids et les entrées, et en ajoutant des biais.

```

1 # Create tf variables for the weights and biases
2 weights = tf.get_variable('weights', shape=[num_in, num_out], trainable=True)
3 biases = tf.get_variable('biases', [num_out], trainable=True)
4
5 # Matrix multiply weights and inputs and add bias
6 act = tf.nn.xw_plus_b(x, weights, biases, name=scope.name)
7
8 if relu == True:
9     # Apply ReLu non linearity
10    relu = tf.nn.relu(act)

```

Le dropout est créée avec la fonction dropout.

```
1 return tf.nn.dropout(x, keep_prob)
```

2.4 Résultat

2.4.1 Extraction automatique de caractéristiques sur le modèle Alex-Net

Ici nous avons gelé les poids du réseau ALEXNET, nous avons seulement appris sur deux couches FC (FC7 et FC8) en changeant les dimensions de la dernière couche du modèle ALEXNET pour une sortie de 16 classes. Après plusieurs jours d'entraînement sur une machine GPU (TITAN

X PASCAL), le modèle commence à stagner et les résultats sont donnée sur les figures 2.4 et 2.5.

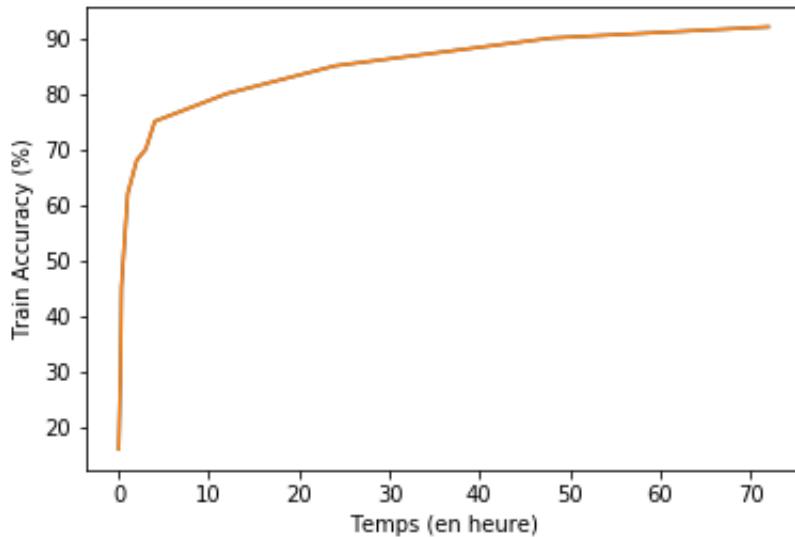


FIGURE 2.4 – Précision d’entraînement sur le modèle AlexNet

Nous pouvons remarquer que le transfert learning a été très utile car, seulement après 30 minute, une heure nous étions déjà à 65% de précision dans l’entraînement. La précision étant calculée comme le pourcentage de bonne réponse, comme nous avons 16 classes, le résultat est déjà satisfaisant. En attendant quelques jours supplémentaires l’apprentissage va stagner vers les 87%. Ce qui est très satisfaisant étant donné que même l’humain n’est pas capable de reconnaître certaines d’entre elles.

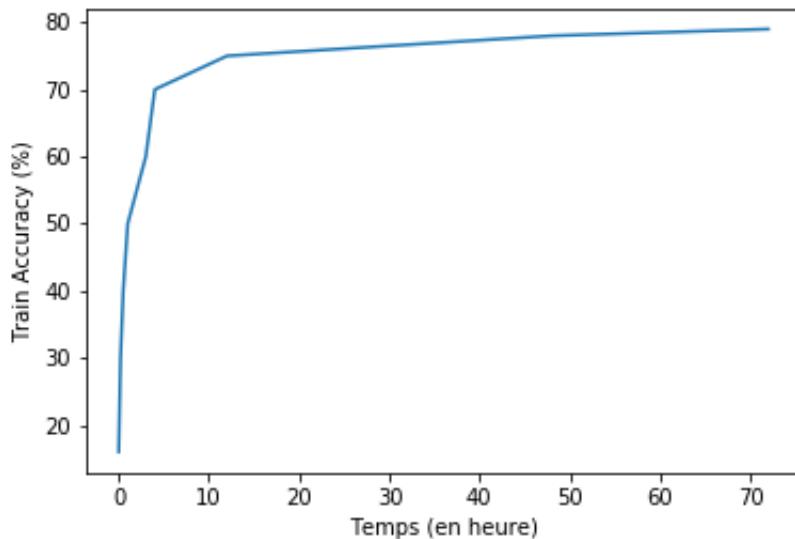


FIGURE 2.5 – Précision de test sur le modèle AlexNet

Pour les tests, nous remarquons la même chose, nous arrivons très rapidement à une précision correcte grâce au transfer learning, qui atteint 74% après quelques jours.

2.4.2 Fine tuning sur le modèle AlexNet

Ici nous avons initialisé les poids avec les poids déjà entraînés du réseau ALEXNET. Cependant, nous apprenons sur toutes les couches du réseau, sans oublier de changer les dimensions de la dernière couche FC pour une sortie de 16 classes. Après plusieurs jours d'entraînement le modèle commence à stagner et les résultats sont donnés sur les figures 2.6 et 2.7.

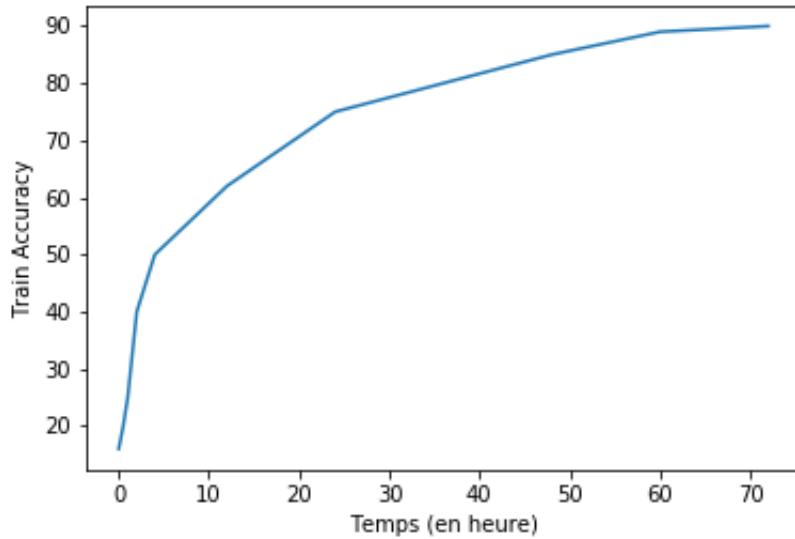


FIGURE 2.6 – Précision d’entraînement sur le modèle AlexNet

Nous pouvons remarquer que l’entraînement est plus long et le modèle met plus longtemps à stagner. C’était prévisible car le finetuning est plus long que le Transfer learning. Nous pouvons voir qu’après trois heures nous sommes seulement à 50%.

Cependant, il donne finalement de meilleurs résultats, il se rapproche des 90% de bonne réponse dans l’entraînement.

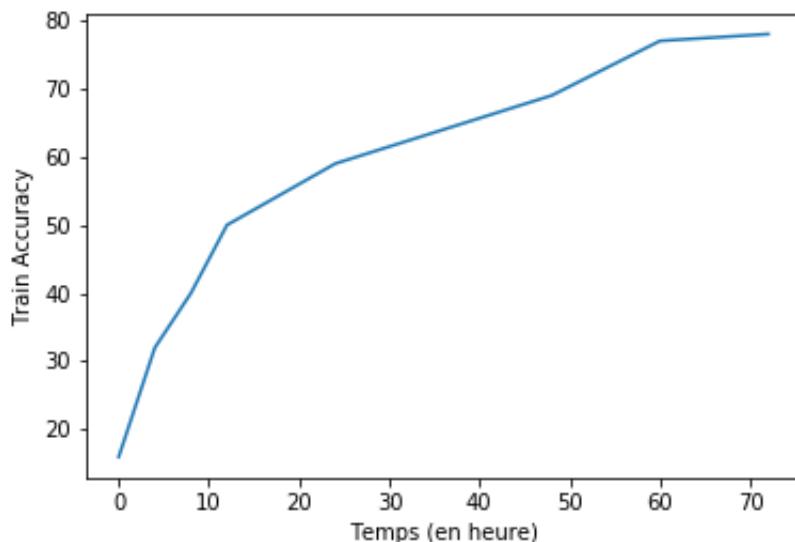


FIGURE 2.7 – Précision de test sur le modèle AlexNet

Pour les tests, nous remarquons à peu près la même chose et après quelques jours la précision final va être autour de 77%. Au final, le Fine Tuning donne de meilleur résultats.

2.4.3 Extraction automatique de caractéristiques sur le modèle Inception

Nous avons aussi testé nos documents sur le modèle Inception v.3. Cependant, comme il est plus compliqué et donne de moins bon résultats, nous n'allons pas le développer ici. Ici nous avons extrait toutes les caractéristiques de toutes les images que nous avions, sur le réseau Inception (en enlevant les couches FC finale), ce qui nous a pris une journée (12 h) en GPU.

Sur les figures 2.8 et 2.9 les courbes représentent l'apprentissage de nos nouvelles données (nos vecteurs de caractéristiques) sur les dernières couches FC du modèle Inception

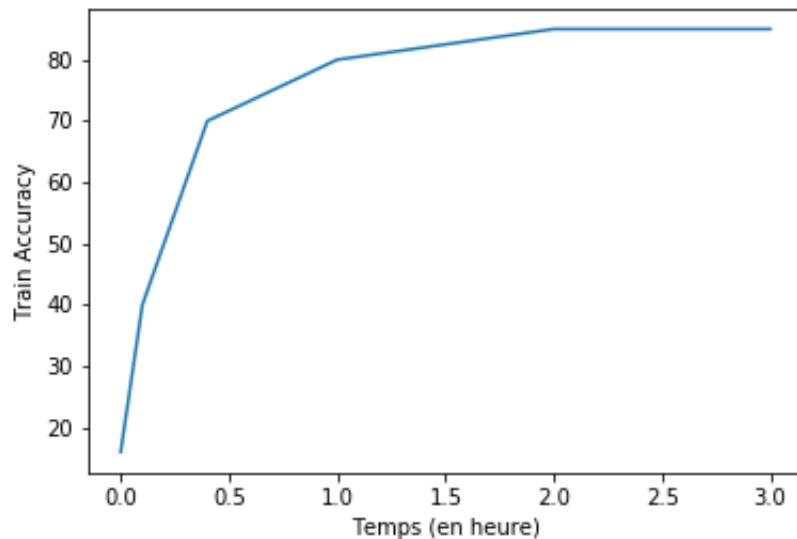


FIGURE 2.8 – Précision d'entraînement sur le modèle Inception

Nous remarquons que l'entraînement se fait plus rapidement et donne à peu près les mêmes résultats, ce qui est très positif.

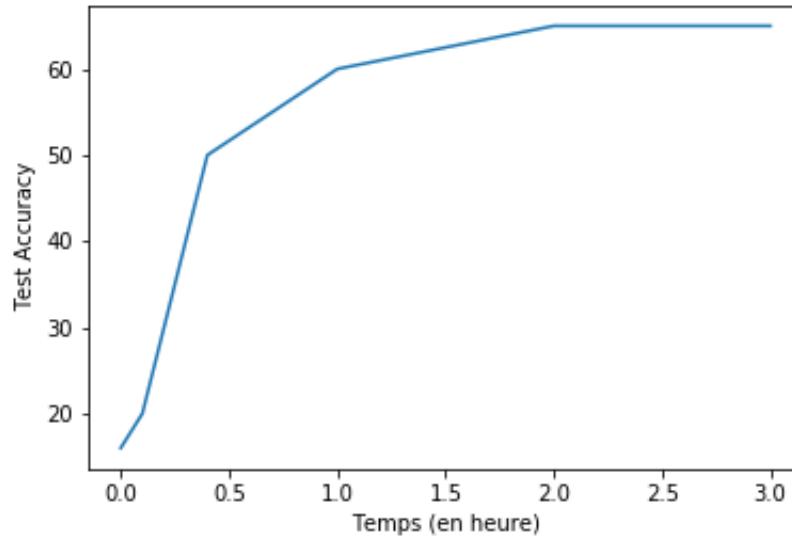


FIGURE 2.9 – Précision de test sur le modèle Inception

Cependant, la précision de test stagne en arrivant vers 70%. Ce qui est moins que sur le réseau AlexNet.

Nous avons donc choisi de classifier nos documents avec le modèle AlexNet.

Reconnaissance Optique de Caractères

La Reconnaissance Optique de Caractères (OCR) consiste à transformer une image d'un texte, en texte réel (fichier .txt). Nous allons reconnaître les mots de l'image pour ensuite pouvoir les utiliser pour annoter notre image. Avant de détecter les mots, il va falloir détecter où se trouve l'information sur notre image, où se trouve les lignes. Puis nous allons détecter les mots dans les lignes pour enfin les reconnaître.

3.1 Détection de lignes

Pour détecter les lignes, nous avons utilisé une librairie appelée kraken. C'est une librairie dérivée de la librairie très utilisée Ocropus. Elle a été créée pour simplifier beaucoup de problèmes de la librairie Ocropus.

3.1.1 Binarisation

Avant de détecter les lignes, nous devons binariser notre image, afin de la simplifier. Nous voulons simplement l'emplacement des lignes dans le document donc nous pouvons simplifier l'image sans perdre l'information importante. Tout d'abord l'image va être transformée en niveau de gris. Nous nous retrouvons avec une matrice de pixel allant de zéro pour un pixel noir à 255 (ou un) pour un pixel blanc. Ensuite, la binarisation va transformer chaque pixel en un ou en zéro. Pour nous le 1 va être un pixel d'une lettre dans un mot par exemple, donc de l'information importante. Et le 0 va être l'absence d'information, un pixel blanc. Pour ce faire nous allons transformer tous les pixels en dessous d'un certain seuil en 0 et tous les pixels au dessus en un.

C'est la binarisation globale :

$$image(x, y) = \begin{cases} 0 & \text{si } image(x, y) \leq seuilGlobal \\ 1 & \text{sinon} \end{cases} \quad (3.1)$$

Nous pouvons autrement appliquer une binarisation locale avec un seuil choisi sur une partie de l'image :

$$image(x, y) = \begin{cases} 0 & \text{si } image(x, y) < seuilLocal(x, y) \\ 1 & \text{sinon} \end{cases} \quad (3.2)$$

de l'image,

Kraken utilise une binarisation locale, le seuil va donc varier dans les différentes parties de l'image. Ceci est utile car nous travaillons avec des photos ou des scans de documents, pour lesquels l'Intensité lumineuse peut varier entre les parties de la page.

3.1.2 Segmentation

Après avoir binarisé le document, nous pouvons appliquer la segmentation du document afin de trouver les lignes dans le document.

Nous allons tout d'abord estimer la taille du texte en trouvant sur l'image binarisée les informations connecté (les lettres dans les mots). Puis nous allons calculer la médiane de toutes les tailles (autant de tailles en dessous qu'au-dessus). Cela va permettre de retirer les blocs trop grand ou trop petit de l'image qui ne peuvent pas correspondre à des lettres.

Maintenant que nous connaissons la taille d'une ligne, nous pouvons détecter toutes les lignes du document.

Nous allons appliquer la dérivée d'un filtre gaussien sur la largeur l trouvés afin de trouver leurs limites pour trouver le bloc correspondant à une ligne.

Filtre gaussien sur un pixel (x,y) qui va appliqué un flou sur l'image :

$$h(x, y) = \frac{1}{2\pi\sigma^2} \frac{-x^2 + y^2}{2\sigma^2} \quad (3.3)$$

σ est la déviation standard de la distribution gaussienne.

Puis la dérivée de ce filtre va nous permettre de distinguer des formes et de distinguer les lignes :

$$\frac{\partial}{\partial l} h(x, y) = -\frac{y * e^{-\frac{(x^2+y^2)}{2\sigma^2}}}{2\Pi\sigma^4} \quad (3.4)$$

Ici l est la largeur des bloc restants (nous allons donc dérivé selon y).

La figure 3.1 représente la dérivé du filtre gaussien selon y .

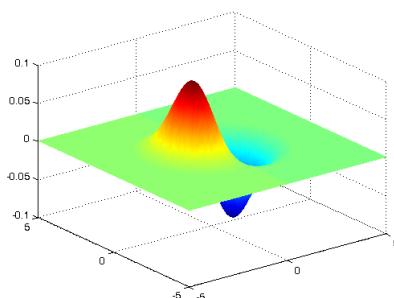


FIGURE 3.1 – Dérivé de du filtre gaussien selon y
[6]

Les bords de l'image sont localisés au maximums locaux des gradients. Nous allons donc appliquer une convolution avec la dérivée du filtre gaussien pour trouver les gradients. Puis comme la direction du gradient est toujours perpendiculaire à la direction des bords, nous pouvons facilement retrouver tous les bords de notre image.

Sur la figure 3.2 de la segmentation, la partie blanche représente le haut de la ligne et la partie noire le bas.



FIGURE 3.2 – Segmentation : application de la dérivé du filtre de gaussien [6]

Puis, nous pouvons détecter les lignes en fusionnant les hauts et les bas trouvés.



FIGURE 3.3 – Segmentation : détection de ligne [6]

Nous pouvons remarquer sur la figure 3.3 que les lignes trouvées ne sont pas parfaites, ce ne sont pas tous des rectangles comme nous le voudrions.

Cependant, lorsque nous dessinons sur l'image, nous allons le faire avec des rectangles.

Structural proteins of human respiratory coronavirus OC43

Brenda G. Hogue and David A. Brian

Department of Microbiology, The University of Tennessee, Knoxville, TN 37996-0845, U.S.A.

(Accepted for publication 28 February 1986)

Summary

The human respiratory coronavirus OC43 was grown on a human rectal tumor cell line and was isotopically labeled with amino acids, glucosamine, and orthophosphate to analyze virion structural proteins. Four major protein species were resolved by electrophoresis and many of their properties were deduced from digestion studies using proteolytic enzymes. The four proteins are: (1) A 190 kDa protein, the presumed peplomeric protein, that was glycosylated and proteolytically cleavable by trypsin into subunits of 110 and 90 kDa. The subunits each represent a different amino acid sequence on the basis of peptide mapping; (2) a 130 kDa protein that was glycosylated and behaved as a disulfide-linked dimer of 65 kDa molecules. It is the apparent virion hemagglutinin on the basis of digestion studies with trypsin bromelain and pronase; (3) a 55 kDa nucleocapsid protein that was phosphorylated; (4) a 26 kDa matrix protein that was glycosylated. The 190, 130, 55 and 26 kDa species can therefore be designated P, H, N and M, respectively. They exist in molar ratios of 4:1:33:33, and are calculated to be present at the rate of 88, 22, 726, and 726 molecules per virion, respectively.

human respiratory coronavirus OC43, structural proteins

Introduction

Coronaviruses are estimated to cause approximately 20% of upper respiratory disease in humans (Larsen et al., 1980; McIntosh et al., 1970). Human respiratory coronaviruses are found as either one of two antigenic types: those related to human coronavirus OC43 (viruses that could originally be grown only in tracheal organ culture), and those related to human coronavirus 229E (viruses that could be

FIGURE 3.4 – Détection des lignes sur un de nos documents

Nous observons sur la figure 3.4 que la segmentation se passe correctement et que sur ce document, toutes les lignes sont trouvées. Sur d'autres documents il peut y avoir des problèmes de rotation ou de qualité instaurant une perte d'information et des oubliés de lignes.

3.2 Détection de mots

3.2.1 Histogramme

Nous partons du principe que les espaces entre les mots sont plus grands que les espaces entre les lignes. Donc, dans l'image binarisée, il y aura plus de pixels blancs (égal à un) entre deux mots (grand ensemble de pixels noirs) qu'entre deux lettres (petit ensemble de pixels noirs). Les tailles des espaces (que ce soit entre les lettres ou les mots) peuvent varier légèrement au

cours du document, si nous faisons l'histogramme du nombre d'espaces selon leurs tailles, nous aurons normalement deux distributions gaussiennes. La première pour les petits espaces et la deuxième pour les grands.

Sur la figure 3.5, nous avons l'histogramme de l'image test de la figure 3.4

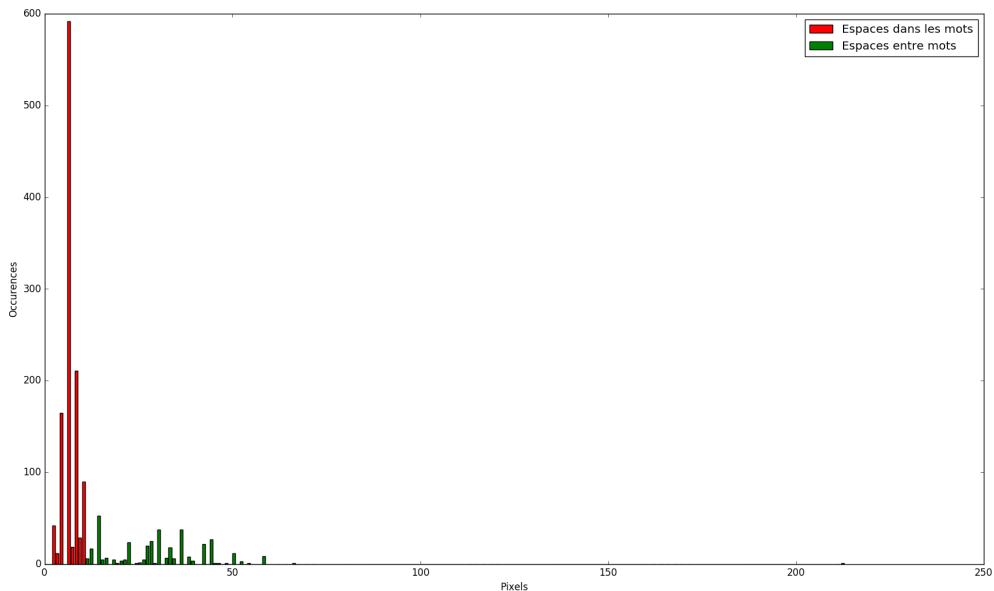


FIGURE 3.5 – Histogramme des espaces

L'histogramme nous indique la séparation entre les espaces entre les lettres et les mots, nous remarquons que le point optimal pour la séparation se trouve entre les deux gaussiennes. Après avoir testé des méthodes non concluantes de clustering pour trouver ce point optimal, nous avons décidé d'utiliser la méthode ISODATA.

3.2.2 The Iterative Self-Organizing Data Analysis Technique" (ISODATA)

Nous allons prendre le milieu des deux points max des gaussiennes trouvées. Ce sont en fait les tailles qui apparaissent le plus de fois pour les espaces entre les mots et les espaces entre les lettres. Puis, si ça ne suffit pas nous allons réitérer avec comme nouveau point l'endroit trouvé. Formellement, la méthode d'ISODATA cherche tous les points $t = \frac{l+m}{2}$ où l est la moyenne des tailles des espaces inférieurs ou égal à t et m la moyenne des tailles des espaces supérieurs à t . Cette équation peut nous donner plusieurs solutions mais en choisissant le premier, la plus petite, le point optimal est trouvé sur presque tous les types de documents.

Une fois ce point optimal trouvé, nous découpons chaque ligne en mots aux points où les espaces sont de taille supérieurs à t .

Structural proteins of human respiratory coronavirus OC43

Brenda G. Hogue and David A. Brian

Department of Microbiology, The University of Tennessee, Knoxville, TN 37996-0845 U.S.A.

(Accepted for publication 28 February 1986)

Summary

The human respiratory coronavirus OC43 was grown on a human rectal tumor cell line and was isotopically labeled with amino acids, glucosamine, and orthophosphate to analyze virion structural proteins. Four major protein species were resolved by electrophoresis and many of their properties were deduced from digestion studies using proteolytic enzymes. The four proteins are: (1) A 190 kDa protein, the presumed hemagglutinin, that was glycosylated and proteolytically cleavable by trypsin into subunits of 110 and 90 kDa. The subunits each represent a different amino acid sequence on the basis of peptide mapping; (2) 130 kDa protein that was glycosylated and behaved as a disulfide-linked dimer of 65 kDa molecules. It is the apparent virion hemagglutinin on the basis of digestion studies with trypsin, bromelain and pronase; (3) a 55 kDa nucleocapsid protein that was phosphorylated; (4) a 26 kDa matrix protein that was glycosylated. The 190, 130, 55 and 26 kDa species can therefore be designated P, H, N and M respectively. They exist in molar ratios of 4 : 1 : 33 : 33 and are calculated to be present at the rate of 88, 22, 726, and 726 molecules per virion, respectively.

human respiratory coronavirus OC43, structural proteins

Introduction

Coronaviruses are estimated to cause approximately 20% of upper respiratory disease in humans (Larsen et al., 1980; McIntosh et al., 1970). Human respiratory coronaviruses are found as either one of two antigenic types: those related to human coronavirus OC43 (viruses that could originally be grown only in tracheal organ culture), and those related to human coronavirus 229E (viruses that could be

0168-1702/86/003.50 © 1986 Elsevier Science Publishers B.V. (Biomedical Division)

FIGURE 3.6 – Détection des mots dans les lignes de nos documents

Nous remarquons, sur la figure 3.6 que le point optimal a bien été trouvé et que sur ce document, tous les mots sont trouvés. Sur d'autres documents il peut y avoir des problèmes, dans ce cas il faut recommencer l'algorithme en prenant comme nouveau m le point t que nous avons trouvé. Cependant, il faut faire attention à ne pas réitérer trop de fois, sinon nous pouvons séparer à l'intérieur des mots. Après bien des tests, nous avons remarqué que sur nos documents, un espace entre les mots est supérieur à 1.5 fois la médiane de tous les espaces du document. Si nous dépassons cette borne, nous arrêtons de réitérer l'algorithme.

3.3 Apprentissage de mots

3.3.1 Recurrent Neural Networks

Présentation

Quand nous lisons, notre compréhension de chaque mot est basée sur la compréhension des mots précédents et du contexte créé par ces mots. Nous avons une mémoire qui nous permet de

comprendre le texte. Les réseaux de neurones traditionnels peuvent seulement faire cela, ils ne peuvent seulement traiter des informations à un temps donné et donc ne pourrons pas utiliser un contexte (des informations données précédemment dans le texte) pour trouver le bon mot.

Ces dernières années, les réseau de neurones récurrents (RNN) sont devenus l'approche favorite pour les tâches linguistiques. C'est un type de réseau qui permet de calculer séquentiellement et d'avoir une "mémoire" dans le temps, ce qui est essentiel quand nous travaillons avec des mots. En effet il y a une logique de langage et dans la langue française par exemple, il y a rarement un x après un z. Le réseau récurrent est capable de remarquer cela et ne va jamais reconnaître un x s'il a détecté un z avant.

Un RNN comporte des boucles qui permettent à l'information d'être mémorisée. Il y a des connections à l'intérieur d'une même "couche", cela va créer un état interne qui va nous permettre de garder des informations en mémoire.

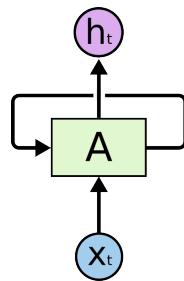


FIGURE 3.7 – RNN plié
[5]

Sur la figure 3.7, un réseau de neurone (représenté par le A) traite l'entrée x_t et transmet une valeur de sortie h_t . La boucle nous indique qu'avant la sortie finale, l'information de sortie est passé par les différentes étapes du RNN.

Nous pouvons identifier ce réseau comme plusieurs fois le même réseau et chacun donne sa sortie (le mot ou la lettre trouvée) à son voisin (Schéma figure 3.8).

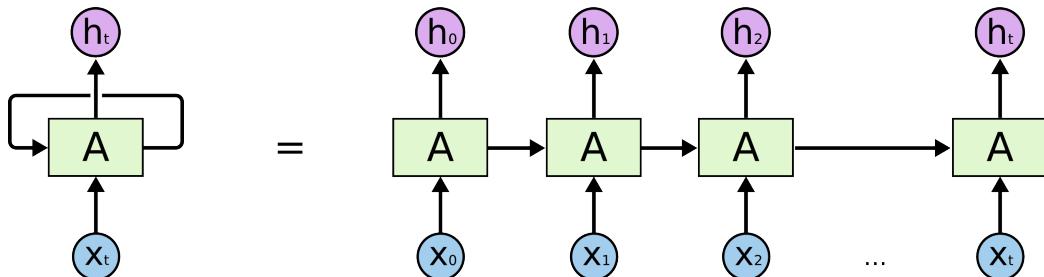


FIGURE 3.8 – RNN déplié
[5]

Dans ces RNNs de base, la fonction qui va être utilisée va être assez simple, comme une tangente hyperbolique par exemple.

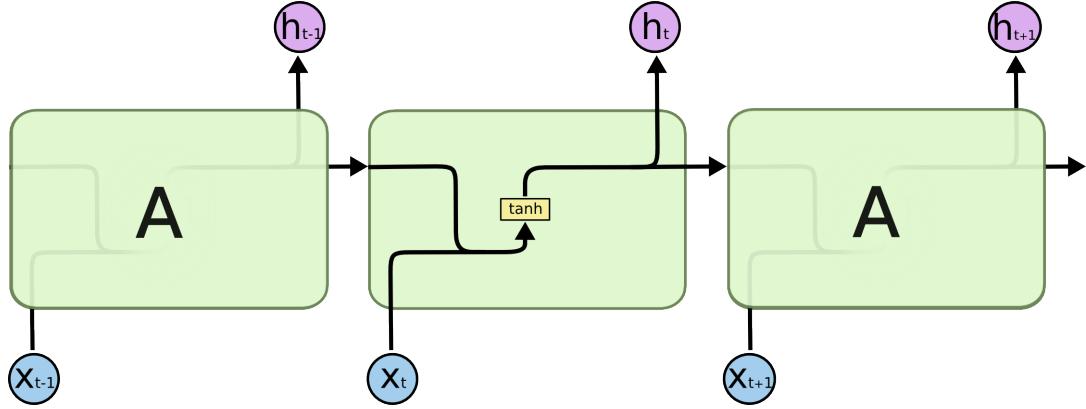


FIGURE 3.9 – Simple fonction
[5]

Dans ces dernières années, les RNNs ont eu beaucoup de succès sur une multitude de problèmes. Reconnaissance vocale, traduction, modélisation d'un langage, annotation d'image, Tous utilisent des RNNs.

Problème lors de la Rétro-propagation dans le temps

Nous allons ici expliquer brièvement la rétro-propagation dans le temps et remarquer un problème dès que la phrase devient longue.

Par la suite, x est toujours l'entrée, y la sortie et s_t ce que l'on va fournir au prochain RNN. U, V, W sont les poids du réseau que l'on va changer au fur et à mesure. Ils sont localisés dans sur la figure 3.10

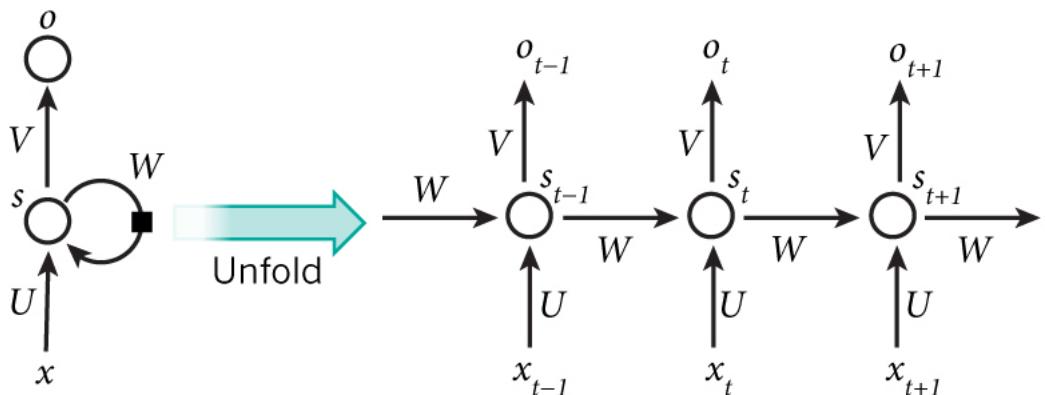


FIGURE 3.10 – Calcule de l'erreur
[11]

Les équations du RNN peuvent se résumer à cela :

On définit l'état des cellules cachées d'un RNN simple, par l'équation : $s_t = \tan(U * x_t + W * s_{t-1})$. Ou x_t est l'entrée du réseau au temps t , qui peut être un vecteur de chiffres représentant un mot par exemple.

On définit également la sortie du réseau $o_t = \text{softmax}(V * s_t)$ car nous voulons que la sortie soit un vecteur de probabilité des différentes classes possibles.

Afin d'effectuer la descente de gradient, il faut dériver l'erreur totale par rapport à tous les poids. C'est la somme des erreurs à chaque temps t :

$$\frac{\partial E}{\partial P} = \sum_t \frac{\partial E_t}{\partial P}$$

Prenons l'exemple de la figure 3.11 en calculant l'erreur sur le poids W et pour $t = 3$:

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$

Pour $t=3$ on calcule avec la règle de la chaîne :

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial o_3} \frac{\partial o_3}{\partial s_3} \frac{\partial s_3}{\partial W}$$

Pour la backpropagation simple nous avons besoin du calcul du gradient de l'erreur selon les paramètres, pour pouvoir trouver dans quelle direction les changer pour minimiser l'erreur. Cependant, maintenant le gradient se calcule avec tous les gradients depuis le début ($t=0$) or $\frac{\partial s_3}{\partial W}$ n'est pas une constante, car $s_3 = \tan(U * xt + W * s_2)$ et s_2 dépend aussi de W

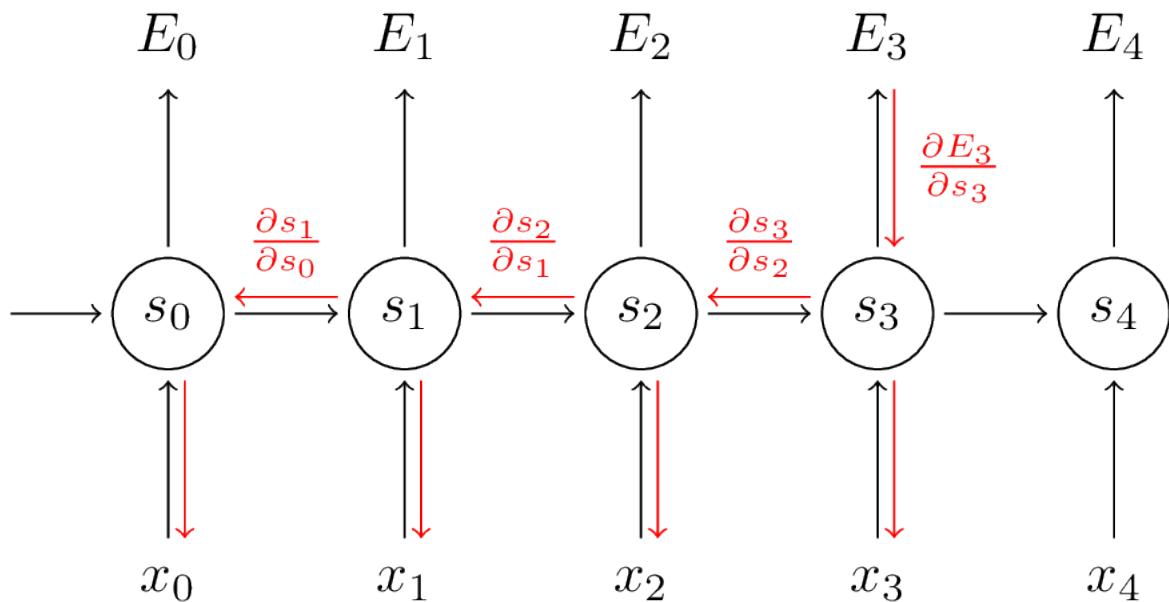


FIGURE 3.11 – Calcule du gradient

Puis $s_2 = \tan(U * xt + W * s_1)$ et s_1 dépend aussi de W. Nous aurons donc les termes

$$\frac{\partial s_3}{\partial W} + \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial W} + \frac{\partial s_3}{\partial s_1} \frac{\partial s_1}{\partial W} + \dots$$

Nous allons donc arriver à

$$\frac{\partial E_t}{\partial W} = \sum_{k=0}^t \frac{\partial E_t}{\partial o_t} \frac{\partial o_t}{\partial s_t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial W}$$

C'est ici qu'intervient le problème de la disparition du gradient. En effet, le terme $\frac{\partial s_t}{\partial s_k}$ se calcule avec la règle de la chaîne.

Pour $k = 2$ par exemple nous avons : $\frac{\partial s_2}{\partial s_0} = \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$. Donc plus le t devient grand, plus nous multiplions les dérivées de s . Or ces termes sont calculés avec des tangentes hyperboliques ou des sigmoïdes donc la dérivée sera inférieure à 1. Nous multiplions des termes plus petits que 1 ensemble donc plus t devient grand, plus le gradient devient proche de 0.

Donc quand nous avons des phrases trop longues, des textes, des livres ... Le gradient va devenir petit et le réseau va oublier les termes importants du début de la phrase. Il va attribuer des poids de plus en plus petits au début de la phrase. Ces réseaux utilisent donc seulement les informations récentes pour effectuer la tâche demandée. Par exemple, si nous voulons prédire le prochain mot d'une phrase courte comme "les nuages sont dans le ciel". Le mot "ciel" est assez facile à trouver avec seulement une mémoire à cours terme, les cinq derniers mots par exemple, nous n'avons pas besoin d'un contexte plus grand.

Cependant, dès que les phrases deviennent plus longues, nous avons besoin de plus de contexte. Si nous avons un article à résumer (comme dans le chapitre suivant) et que nous avons besoin de la première phrase pour comprendre le contexte de la dernière phrase.

3.3.2 Long Short-Term Memory

3.3.2.1 Présentation

Les réseaux "Long Short Term Memory" (appelé généralement LSTM) ont été inventés par Hochreiter Schmidhuber en 1997 et sont énormément utilisés ces dernières années pour répondre à une multitude de problèmes. Ils font partie des RNN, mais ils ont été conçus pour être capable d'apprendre des dépendances "à long terme". En effet, ils ont été créés pour ne pas perdre de l'information importante située au début d'un texte au fur et à mesure de l'avancement de l'apprentissage.

Nous avons vu que les RNN de base avaient des fonctions assez simples pour trouver les informations importantes (comme une fonction tanh dans leurs modules de répétition).

Nous pouvons remarquer figure 3.12 que, dans un LSTM, la fonction va être beaucoup plus compliquée.

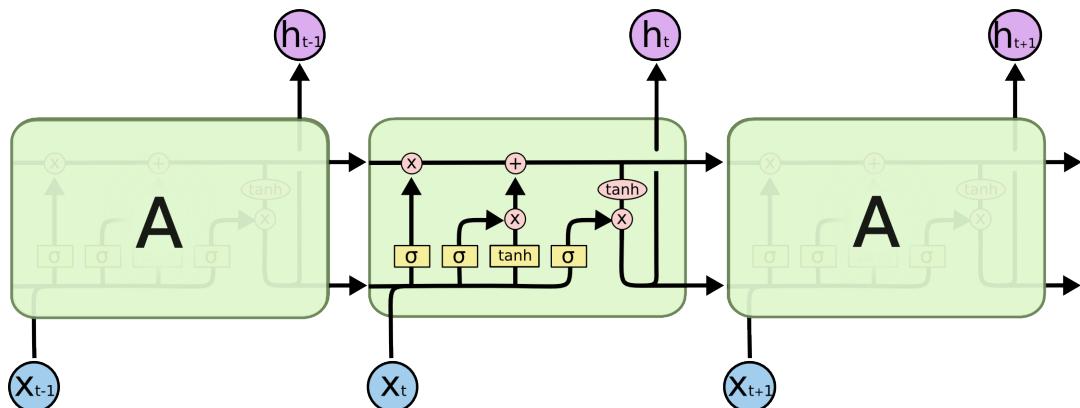


FIGURE 3.12 – Fonctions plus complexes
[5]

Sur la figure 3.12, chaque ligne porte un vecteur de chiffre (ici les lettres vont être transformées en chiffre) d'une sortie d'un noeud à l'entrée d'un autre noeud. Les cercles roses représentent des opérations classiques (comme une addition de vecteur par exemple). Et les boîtes jaunes

sont des réseaux de neurones classiques avec une fonction d'activation. Les lignes qui fusionnent représentent une concaténation de deux vecteurs, tandis que deux lignes bifurquantes montrent que le vecteur a été copié pour être utilisé en plusieurs endroits

L'importante information dans le LSTM est l'état de la cellule. Dans la figure 3.13, l'état va passer d'un temps $t - 1$ à t en subissant seulement quelques opérations mineurs. Il se peut que l'information passe sans que l'état de la cellule change.

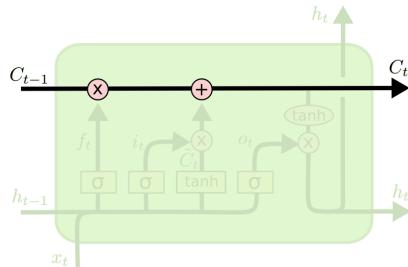


FIGURE 3.13 – L'idée derrière le LSTM
[5]

L'important dans les LSTM est que les cellules ont la possibilité d'enlever ou de rajouter de l'information à l'aide de portes ("gates"). Ce qui va permettre une mémoire à long terme, effacer les informations inutiles (mettre des poids proche de zéro) et au contraire mettre des poids plus grand quand l'information est importante, même si elle se trouve au début de la phrase. Comme cela, elle aura encore un gros impact même à la fin de la phrase.

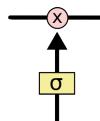


FIGURE 3.14 – Les portes
[5]

La porte de la figure 3.14 peut par exemple laisser passer de l'information ou la bloquer. Elle est composée d'un réseau de neurone avec la fonction d'activation sigmoïde, ainsi qu'une multiplication de vecteurs. La sortie de cette couche est un chiffre entre zero et un représentant l'importance de l'information précédente, l'information qui faut ou non laisser passer. Zéro veut dire ne rien laisser passer, l'information précédente n'as aucune importance, tandis que un laisse l'information précédente influencer totalement l'état de la cellule.

Une cellule LSTM est composée de trois de ces portes, la *forget gate*, l'*input gate* et l'*output gate*. L'idée est de prendre l'état précédent, d'effacer (d'oublier) les informations inutiles, de sélectionner d'autres informations importantes et de le donner au prochain LSTM.

3.3.2.2 Exemple pas à pas

- Forget Gate (FG)

La première étape dans notre LSTM est de bien sélectionner les informations inutiles de notre cellule, les informations que l'on ne va pas prendre. Cette décision est faite par la FG . Nous voyons sur la figure 3.15, que cette porte est simplement une sigmoïde qui va prendre en entrée x_t (l'information au temps t) et h_{t-1} (l'information traitée par un LSTM au temps t-1). Puis si l'information est importante, la sortie va être proche de un

et nous allons tout garder, au contraire si l'information est proche de zero, c'est qu'elle est inutile pour la suite. Pour notre exemple de langage et d'essayer de prédire le mot suivant basé sur les mots précédents, la cellule peut par exemple contenir le genre d'un sujet ce qui est important pour utiliser par la suite le bon pronom. Cependant, quand nous voyons un autre sujet, nous voulons oublier cet ancien sujet pour accorder la suite avec ce nouveau sujet.

Cette porte va nous permettre d'oublier ce genre (de le vider de la cellule)

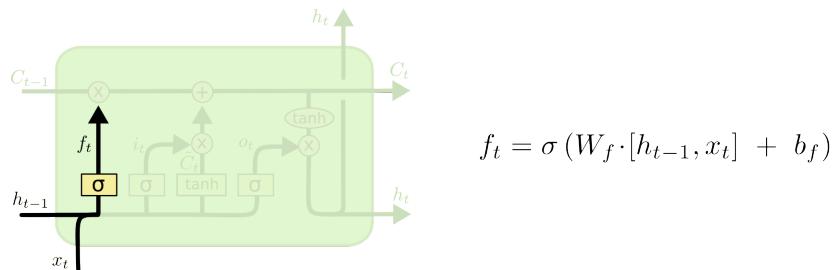


FIGURE 3.15 – Principe de la "forget gate"
[5]

- Input Gate

La prochaine étape est de décider quelles nouvelles informations nous allons stocker dans la cellule. Cela se fait en deux parties, premièrement, sur la figure 3.16, le LSTM utilise une "input gate" qui est une sigmoïde qui va sélectionner les valeurs à mettre à jour et les stocker dans une variable i_t . Puis une tangente hyperbolique va créer un vecteur de nouveaux candidats \tilde{C}_t qui pourraient être ajoutés dans la cellule. Pour notre exemple, nous voudrions ajouter le genre du nouveau sujet dans la nouvelle cellule.

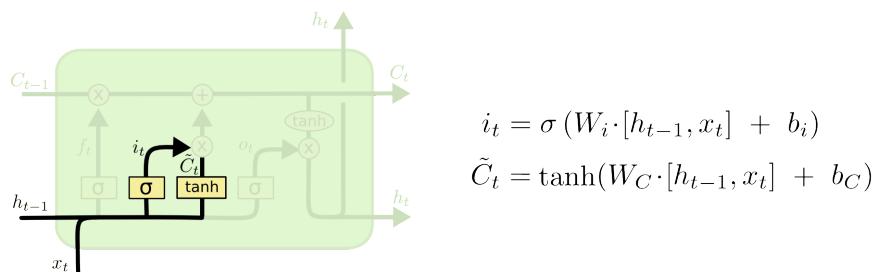


FIGURE 3.16 – Principe de "l'input gate"
[5]

Il faut maintenant, comme montré sur la figure 3.17, mettre à jour la cellule C_{t-1} dans la nouvelle cellule C_t . Les portes précédentes ont sélectionné quelles informations garder et quelles informations oublier, il faut maintenant le faire.

Nous allons multiplier l'état de la cellule C_{t-1} par f_t , pour oublier ce que nous devons oublier et on additionne $i_t * \tilde{C}_t$, les nouveaux candidats avec le poids attribué au vecteur (pour savoir s'ils sont importants ou non)

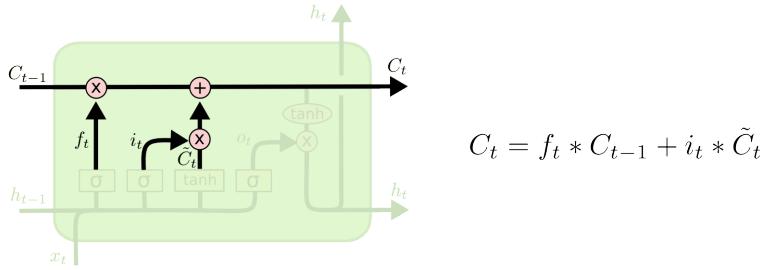


FIGURE 3.17 – Mise à jour du nouvel état
[5]

- Output Gate

Nous avons maintenant besoin de décider quoi envoyer au réseau suivant. On remarque, sur la figure 3.18, que la sortie va être basée sur l'état de la cellule, mais elle va être filtrée par une sigmoïde qui va choisir les parties importantes. Puis nous allons le multiplier avec la sortie pour avoir les parties importantes pour la suite.

Pour notre exemple, si la cellule reconnaît un sujet..elle pourrait transmettre des informations pertinentes comme le fait qu'il soit singulier ou pluriel à la cellule voisine" ?

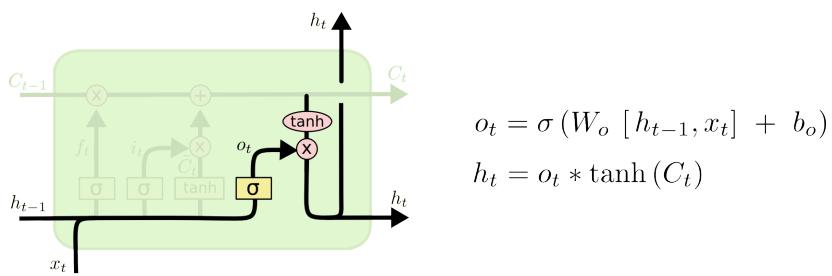


FIGURE 3.18 – Principe de l'output gate
[5]

Au début de cette partie, nous avons dit que beaucoup de problèmes linguistiques étaient résolus à l'aide de RNNs. En fait, ils sont souvent résolus par des LSTMs qui marchent bien mieux dans la plupart des cas.

3.3.3 Apprentissage

3.3.3.1 Générer des données d'apprentissage

Nous voulons générer les mots d'un dictionnaire anglais afin d'entraîner un réseau capable de reconnaître ceux-ci et de travailler sur des documents en anglais. Nous avons récupéré un dictionnaire anglais avec 58110 mots.

Pour la génération, nous avons utilisé plusieurs polices de caractères, car nos documents n'ont pas toujours la même police.

Nous remarquons figure 3.19 que les images générées sont parfaites. Or dans nos documents les images (photos, scans) sont loin d'être parfaites. Sur les images contenues dans la figure 3.20, nous avons ajouté des transformations (décoloration aléatoire, rotation..) afin de détériorer l'image et de rendre notre réseau plus robuste.



FIGURE 3.19 – Génération d’images de mots



FIGURE 3.20 – Génération d’image abîmée

3.3.3.2 Modèle CRNN

Nous avons appris sur le modèle CRNN, représenté figure 3.1, implémenté par Alexandre Fabre en tensorflow.

Type	Configuration
CTC	-
LSTM bidirectionnel	hidden units :256
LSTM bidirectionnel	hidden units :256
Séquences	-
Normalisation	-
Convolution	n :512, fenêtre : 2×2 , s :1, p :0
Max Pooling	fenêtre : 1×2 , s :2
Convolution	n :512, fenêtre : 3×3 , s :1, p :1
Normalisation	-
Convolution	n :512, fenêtre : 3×3 , s :1, p :1
Max Pooling	fenêtre : 1×2 , s :2
Convolution	n :512, fenêtre : 3×3 , s :1, p :1
Normalisation	-
Convolution	n :512, fenêtre : 3×3 , s :1, p :1
Max Pooling	fenêtre : 1×2 , s :2
Convolution	n :512, fenêtre : 3×3 , s :1, p :1
Max Pooling	fenêtre : 1×2 , s :2
Convolution	n :512, fenêtre : 3×3 , s :1, p :1
Entrée	32×100 niveau de gris

TABLE 3.1 – La configuration du CRNN

Ici n est le nombre de filtres appliqués, s est le pas utilisé lors de l'application d'un filtre et p indique la méthode à choisir pour l'application des filtres sur les bords de l'image. Nous pouvons oublier ou rajouter des valeurs autour de la matrice pour que le calcul soit possible.

Nous pouvons remarquer dans la figure 3.1 des LSTM bidirectionnels. La différence avec ceux présentés précédemment est qu'ils envoient de l'information à leurs deux voisins (à droite et à gauche).

Cela permet un contexte plus robuste, nous pouvons par exemple apprendre qu'un "x" ne peut pas suivre un "z", mais également qu'un "z" précède rarement un "r"

La dernière étape est l'utilisation d'une Classification Temporelle Connexioniste (CTC). Elle sert à réunir les sorties des LSTM, qui sont des chiffres (pixels en entrées), pour former un mot. Il va aider le LSTM à éviter les sorties ne contenant pas d'information importante (des pixels blancs par exemple) et trouver le chemin optimal pour rassembler toutes les sorties importantes afin de les rassembler et de créer un caractère. La différence entre le chemin réel

et le chemin optimal nous permet de calculer l'erreur faite par le CTC, pour ensuite essayer de la diminuer en modifiant différents poids.

3.3.3.3 Résultats

Les images de mot, trouvées dans la section détection de mots, sont envoyées dans notre réseau CRNN qui réalise une reconnaissance de caractères. Ainsi, pour chaque image nous récupérons le mot correspondant. La phase d'apprentissage utilisant les images générées a partir du dictionnaire a durée deux jours :

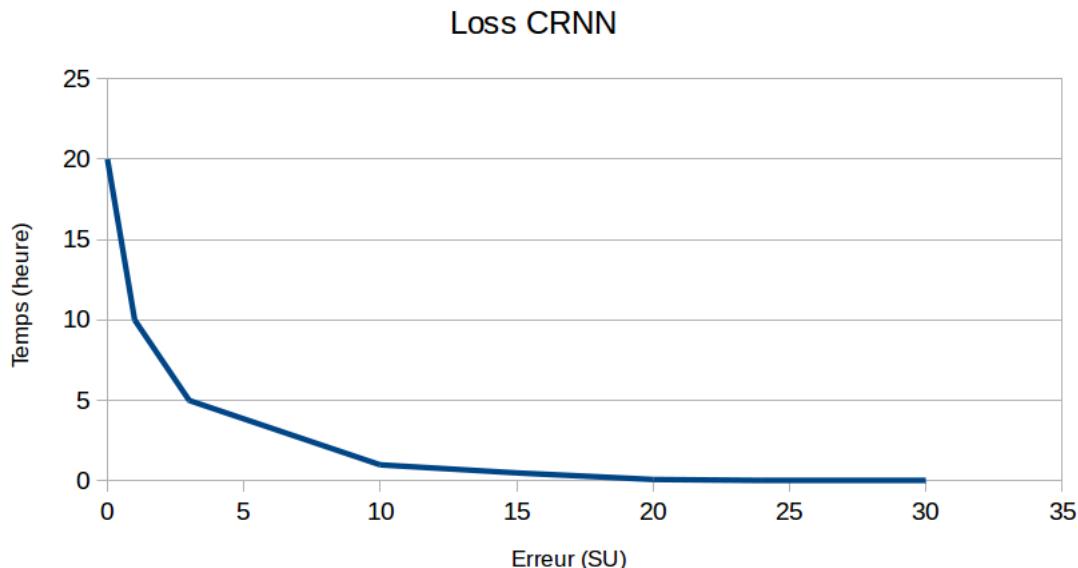


FIGURE 3.21 – L'erreur commise en fonction du temps

Si nous nous concentrons sur la fin de la figure 3.21, nous voyons que l'erreur converge vers 0.03, ce qui est encore trop grand. Les résultats ne sont pas convaincant, le réseau fait beaucoup de fautes, seulement un mot sur dix est totalement juste. Nous avons sans doute utilisé une police et une taille de police ne correspondant pas à tous les documents. Nous pensons qu'avec un apprentissage plus long sur des polices plus appropriées, les résultats seront meilleurs.

Nous avons dans le chapitre précédent transformé l'image du document en fichier texte. Pour annoter le document, nous allons maintenant résumer automatiquement le fichier texte et obtenir une phrase décrivant le document.

L'âge d'or d'internet apporte une quantité gigantesque d'informations. Nous nous trouvons aujourd'hui surchargé d'informations et c'est pour cette raison que résumer automatiquement des documents devient une tâche essentielle.

Notre modèle utilise des RNNs (décrit précédemment) ainsi qu'un modèle d'attention.

4.1 Modèle d'Attention

Un modèle d'attention dans un RNN permet au réseau de se concentrer petit à petit sur un sous ensemble important de l'entrée, le traiter puis passer à une autre partie de l'entrée. Notre modèle d'attention va aussi être entraîné avec une backpropagation.

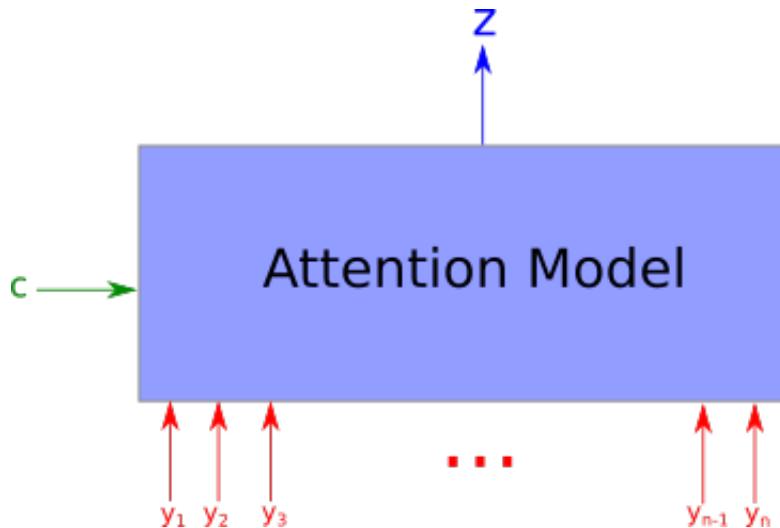


FIGURE 4.1 – Model d'attention
[9]

Le modèle d'attention représenté figure 4.1 est une méthode qui prend n arguments y_1, \dots, y_n , des vecteurs de chiffres représentant des mots par exemple, et un contexte que nous appelons c . Ce modèle ressort un vecteur z qui nous informe où se situe l'information importante de nos entrées (sur quels mots le modèle doit se concentrer) par rapport au contexte c . Le vecteur z est en fait un vecteur de poids de chacune des entrées. Les poids sont calculés par rapport au contexte c .

Dans notre exemple, le contexte peut être le début de la phrase trouvée.

Allons plus en détail sur cette boîte noire. Un modèle d'attention peut par exemple se représenter comme sur la figure 4.2 :

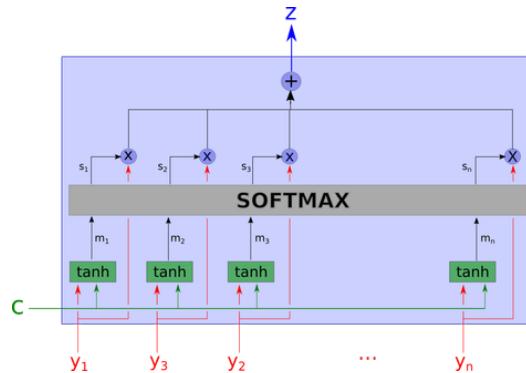


FIGURE 4.2 – A l'intérieur d'un model d'attention
[9]

Dans un premier temps, sur la figure 4.3, nous calculons les m_1, \dots, m_n avec une couche de tangente hyperbolique en prenant en compte le vecteur du contexte (C) et les mots d'entrée y_i . Nous remarquons que chaque m_i est calculé sans prendre en compte les autres vecteurs y_j pour $j \neq i$. Ils sont calculés indépendamment.

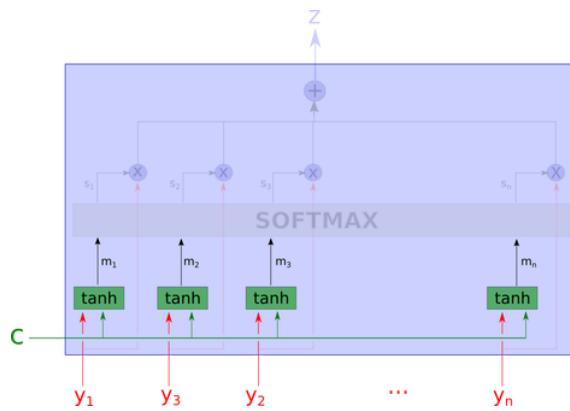


FIGURE 4.3 – Nous prenons en compte le vecteur du contexte
[9]

Puis nous calculons l'importance des mots m_i en calculant un poids pour chacun. Dans la figure 4.4, nous utilisons un softmax. Nous allons prendre chacun des m_i et nous allons les classer par ordre d'importance en leur attribuant un poids pour chacun.

Il est défini comme ceci $\text{softmax}(x_1, \dots, x_n) = \left(\frac{e^{x_i}}{\sum_j e^{x_j}} \right)$

Pour comprendre, nous pouvons utiliser la version une version simplifiée du softmax, l'argmax. Cette fonction renvoie le max des arguments, il va donc attribuer un poids de 1 à l'argument le plus important, et zéro aux autres. Si un argument est vraiment plus grand que tous les autres, alors les deux fonctions coïncident.

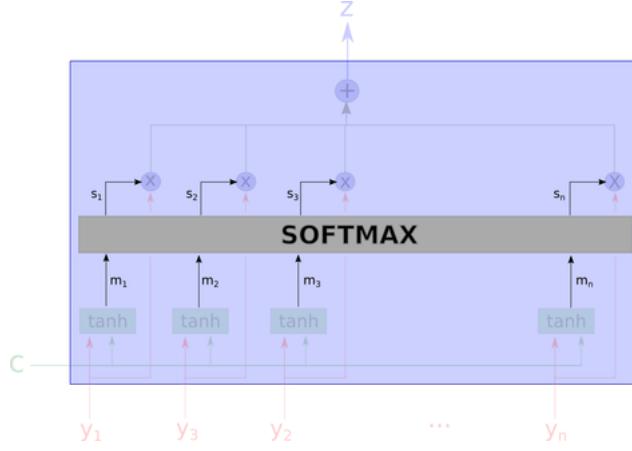


FIGURE 4.4 – Calcule des poids avec une couche SoftMax
[9]

Nous avons dans la figure 4.4, le softmax des m_i . On peut donc identifier ce softmax comme le maximum d'importance des variables en rapport à un contexte.

Il renvoi un vecteur de probabilité et $\sum_i s_i = 1$.

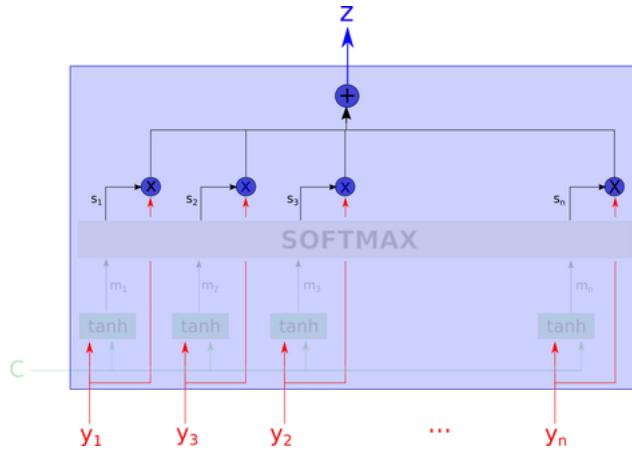


FIGURE 4.5 – Moyenne pondérée pour calculer la sortie
[9]

Nous comprenons sur la figure 4.5 que la sortie z est la moyenne pondérée des entrées et les poids représentent l'importance de chaque variable selon le contexte : $z = \sum_i s_i y_i$.

4.2 Résumé automatique d'un document

Il y a deux approches pour résumer un texte :

- L'approche d'extraction consiste à sélectionner des passages d'un texte et les organiser de sorte à faire un résumé compréhensible. Nous pouvons imaginer cette méthode comme un marqueur qui va surligner des passages d'un texte.
- L'approche abstraite consiste à utiliser des techniques de génération du langage pour écrire nous-même des nouvelles phrases. Nous pouvons imaginer cette méthode comme un stylo qui va écrire lui-même un nouveau texte.

La majorité des programmes résumant automatiquement sont basés sur une approche d'extraction des mots, car elle est plus facile à mettre en œuvre. En effet, si nous sélectionnons des parties du texte, il y a beaucoup de chance pour que la phrase soit grammaticalement correcte, lisible et en rapport avec le texte de départ. Ces programmes sont fructueux quand ils sont appliqués à des documents courts. Pour de longs documents, il y a des chances que les phrases choisies ne soient pas liées et que le résumé ne soit pas clair. De plus, cette méthode est trop restrictive pour produire des résumés comme le ferait un humain. En somme, l'approche abstraite est plus dure mais est essentielle et donne de meilleurs résultats.

4.3 Le modèle "Sequence to Sequence"

Notre réseau est basé sur le modèle sequence to sequence avec attention représenté sur la figure 4.6

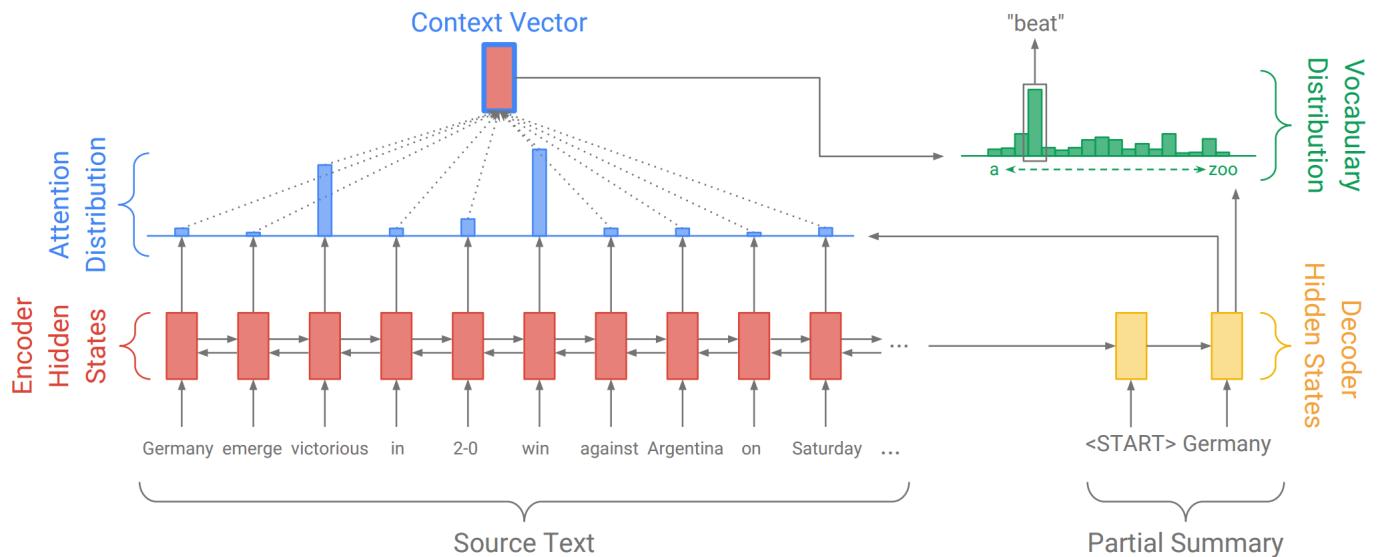


FIGURE 4.6 – Modèle "Seq to Seq avec attention
[14]

Tous les documents sont en anglais, donc nos exemples vont être en anglais. Cependant, les résultats sont transposables si nous mettons un vocabulaire français à la place d'un vocabulaire anglais.

Dans cet exemple, notre texte source est un article commençant par nous informer que samedi, l'Allemagne a gagné contre l'Argentine 2-0 ("Germany emerge victorious in 2-0 win against Argentina on Saturday").

Premièrement, l'encodeur (composé de RNN bidirectionnel) va lire la phrase mot pour mot et va produire une séquence de couches d'états cachés produits par les LSTMs bidirectionnels. Quand l'encodeur a tout lu, le décodeur (composé lui aussi de RNN bidirectionnels) va lire les entrées et sortir une séquence de mots formant un résumé. Pour chaque état, le décodeur reçoit comme entrée le mot trouvé d'avant (pour le premier état il va recevoir un <START> token qui lui indique que c'est le premier mot). Il l'utilise pour décoder l'état caché.

Le modèle d'attention est une distribution de probabilité de tous les mots du texte. Intuitivement il va indiquer au réseau où chercher pour l'aider à trouver le prochain mot de la phrase. Dans la phrase de la figure précédente, nous pouvons remarquer qu'après avoir trouvé Germany, il se concentre sur les mots "win" et "victorious" qui vont apparaître prochainement dans la phrase.

Ensuite, nous utilisons cette distribution pour produire une somme pondérée des couches cachées de l'encodeur, pour donner le vecteur du contexte.

Le contexte peut être vu comme ce que nous avons lu dans le texte jusqu'à ce point. Au final, ce contexte et le décodeur vont être utilisés pour créer une distribution de probabilité de tout les mots du vocabulaire. Le mot avec la plus large probabilité est choisi et le décodeur passe au prochain mot.

La possibilité que le décodeur a de générer des mots dans n'importe quel ordre (ainsi que des mots qui n'apparaissent pas dans le texte) fait que le modèle seq-to-seq est un modèle très puissant pour résumer des textes.

4.3.1 Problèmes

Cette approche présente toutefois deux inconvénients principaux

- Premier problème : elle peut parfois se tromper sur des détails factuels, par exemple comme 2-0 n'apparaît pas souvent dans le vocabulaire, il est tout à fait possible qu'il se trompe sur le score.
- Deuxième problème : il peut y avoir des répétitions dans le résumé ("Germany beat Germany beat Germany beat...")

En fait, ces problèmes sont très communs quand nous travaillons avec des RNNs. Et comme souvent avec le deep-learning, il est difficile d'expliquer pourquoi, mais nous avons quelques idées.

- Explication du premier problème : si un mot n'apparaît pas souvent dans le texte durant l'entraînement du réseau, il va être mal classé avec des mots qui n'ont rien à voir. Il va donc probablement ne jamais être utilisé, car comme il y a beaucoup de couche LSTMs, il est difficile de copier un mot du texte s'il n'apparaît pas souvent dans la phase d'entraînement. De plus, même s'il est bien classé, le réseau peut se tromper avec des mots proches (du même cluster). Par exemple pour les noms de villes ou les prénoms. En bref, la copie d'un mot important du texte est difficile.
- Explication du deuxième problème : cela peut être dû à une importance trop forte du mot précédent. Par exemple, si le mot précédent est faux, Argentina a été remplacé par Germany. Le cas est assez fréquent, car les pays sont classé assez proche et les deux mots apparaissent assez fréquemment dans le texte. Cela peut créer une répétition, car après Germany le modèle va vouloir remettre "beat". Nous pouvons donc arriver à un résultat "Germany beat Germany beat Germany beat...", ce qui est catastrophique.

4.3.2 Solution : Pointer-Generator Networks

Copie inexacte

Une solution pour le premier problème (copie inexacte) est d'utiliser le réseau "pointer-generator". C'est un réseau hybride, mélange des deux méthodes de résumé automatique, qui peut choisir de copier des mots du texte en les pointant. Mais il garde la possibilité de générer des mots nouveaux avec un vocabulaire fixe.

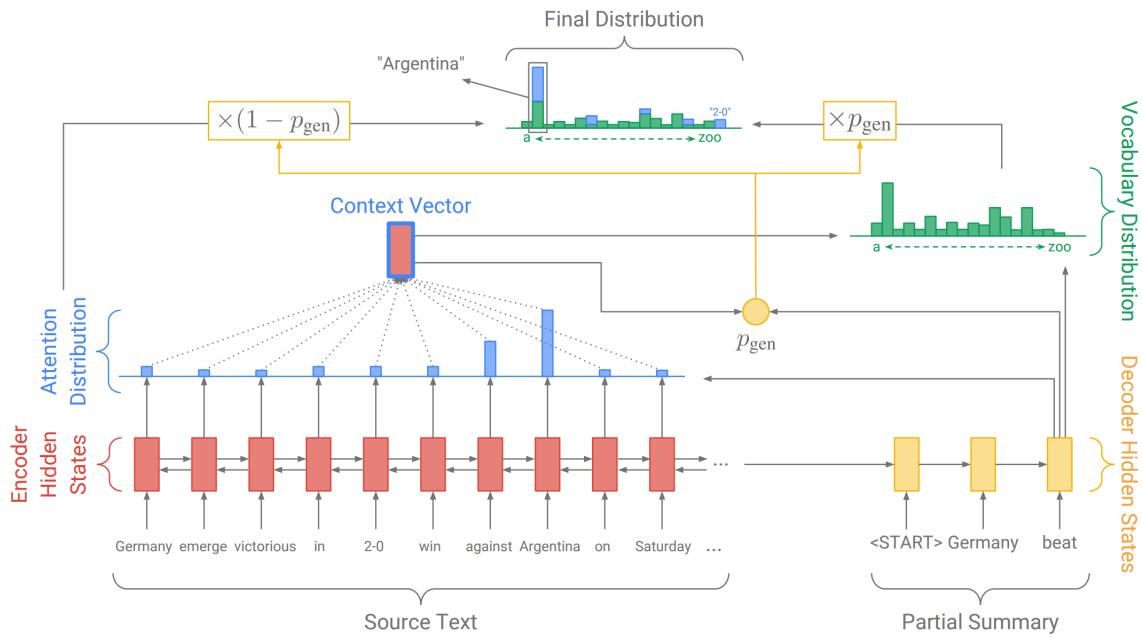


FIGURE 4.7 – Réseau pointeur-générateur
[14]

La figure 4.7 représente la troisième étape du décodeur, après avoir généré deux mots : "germany beat". Comme précédemment nous calculons les distributions de l'attention et du vocabulaire. Cependant, maintenant nous allons calculer une probabilité de génération du mot, on va l'appeler p_{gen} , c'est un scalaire prenant ses valeurs entre 0 et 1. Il représente la probabilité de générer le mot du vocabulaire plutôt que de copier un mot du texte original. Il est utilisé pour pondérer la distribution du vocabulaire P_{vocab} (utilisé pour la génération) et la distribution d'attention a que nous utilisons pour choisir les mots que l'on veut copier dans le texte. La probabilité finale du mot que nous voulons, devient

$$P_{final}(w) = p_{gen} * P_{vocab}(w) + (1 - p_{gen}) \sum_{i:wi=w} a_i$$

La formule indique juste que la probabilité de produire le mot w est égale à la probabilité de le générer du vocabulaire, plus la probabilité de trouver n'importe où dans le texte original. Cette méthode a plusieurs avantages

- La copie de mot du texte original devient beaucoup plus facile. Le réseau a simplement besoin de donner plus d'attention au mot trouvé.
- Nous pouvons copier des mots du texte qui ne sont pas dans le vocabulaire, afin de réduire celui ci, et de gagner de l'espace mémoire et du temps de calcul.
- Comme nous gagnons du temps de calcul, il est donc plus rapide à entraîner et va donner de meilleurs résultats que le modèle "sequence-to-sequence"

Le modèle "pointer-generator" correspond à la combinaison des deux méthodes de résumé automatique, extractive et d'abstraction. Ce qui permet d'obtenir de meilleurs résultats que les méthodes seules.

Répétition

Pour éviter les répétitions, nous allons utiliser la méthode de "coverage". L'idée est d'utiliser la distribution d'attention pour garder une trace des mots déjà rencontrés et de pénaliser le réseau pour parcourir des mots qu'il a déjà parcouru.

Le "vecteur coverage", le vecteur qui permet de savoir où nous nous trouvons est la somme de la distribution d'attention. Pour un mot donné, son "coverage" est la quantité d'attention qu'il a reçu jusque là :

$$c^t = \sum_{t'=0}^{t-1} a^{t'}$$

Puis nous allons introduire un terme de pénalité, une erreur que nous voulons minimiser au cours de l'apprentissage :

$$Loss_t = \sum_i \min(a_i^t, c_i^t)$$

Cela va permettre à la distribution d'attention de baisser les probabilités sur les mots sur les endroits qu'il a déjà rencontrés (le début de la phrase).

4.3.3 Datasets

newsir16

Pour entraîner notre réseau nous avons utilisé l'ensemble d'article newsir16.

C'est un grand dictionnaire contenant 265,512 articles venant de différents blogs et 734,488 articles de journaux. Ce dictionnaire est construit de cette sorte :

- identifiant unique d'un article
- Le titre
- Le contenu (la taille moyenne d'un article est ici de 405 mots)
- La source
- La date de publication
- le type (journal ou blog)

Pour notre réseau, nous avons seulement besoin du contenu des articles que l'on donne au réseau et du titre qui va nous servir de résumé (la sortie de référence).

CNN / Daily Mail dataset

En 2015, Hermann et al ont créé pour la recherche et l'analyse de document deux datasets de taille conséquente d'articles de CNN et du DailyMail. Ce dataset contient 90000 d'articles de CNN et 197000 articles du DailyMail accompagnés de leurs résumés.

4.4 Word Embedding

Le "word embedding" est le fait consiste à représenter le texte par des nombres. Les mots deviennent des vecteurs, ce qui facilite pour le calcul de leurs ressemblances.

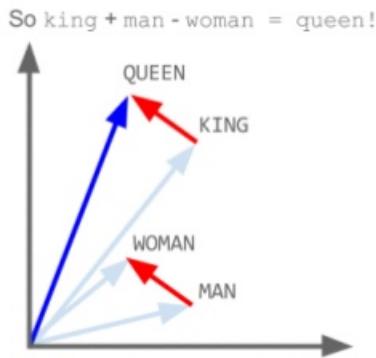


FIGURE 4.8 – Représentation des mots par des vecteurs
[2]

Nous pouvons remarquer sur la figure 4.8 que le mot King est représenté par un vecteur. Et pour représenter le mot Queen dans cet espace vectoriel créé par l'embedding, il suffit d'ajouter le vecteur (*Man – woman*), ce qui a aussi du sens sémantiquement.

Par exemple, la distance Euclidienne entre deux vecteurs représentant des mots, est une bonne méthode pour mesurer les ressemblances entre les mots, aussi bien linguistiquement (même racine par exemple) que sémantiquement (tous les deux sont des noms de villes par exemple).

Glove

"Global Vectors for Word Representation" (Glove) va tout d'abord compter les mots du texte et va créer une matrice d'occurrences de mots selon le contexte :

Pour les phrases

- I like deep learning.
- I like NLP.
- I enjoy flying.

la matrice d'occurrence va être le nombre de fois qu'un mot apparaît dans la même phrase qu'un autre mot.

Nous remarquons figure 4.9 qu'avec seulement 3 petites phrases, la matrice est déjà grande. Sur un article nous ne pourrons pas travailler avec cette matrice, il va falloir réduire ses dimensions en la vectorisant. On aura, à la fin de l'apprentissage, un vecteur par mot.

Counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

FIGURE 4.9 – Matrice d'occurrence

Nous allons comme montrer sur la figure 4.10 l'entraîner sur un réseau de neurone FC de deux couches, sur un grand corpus de texte.

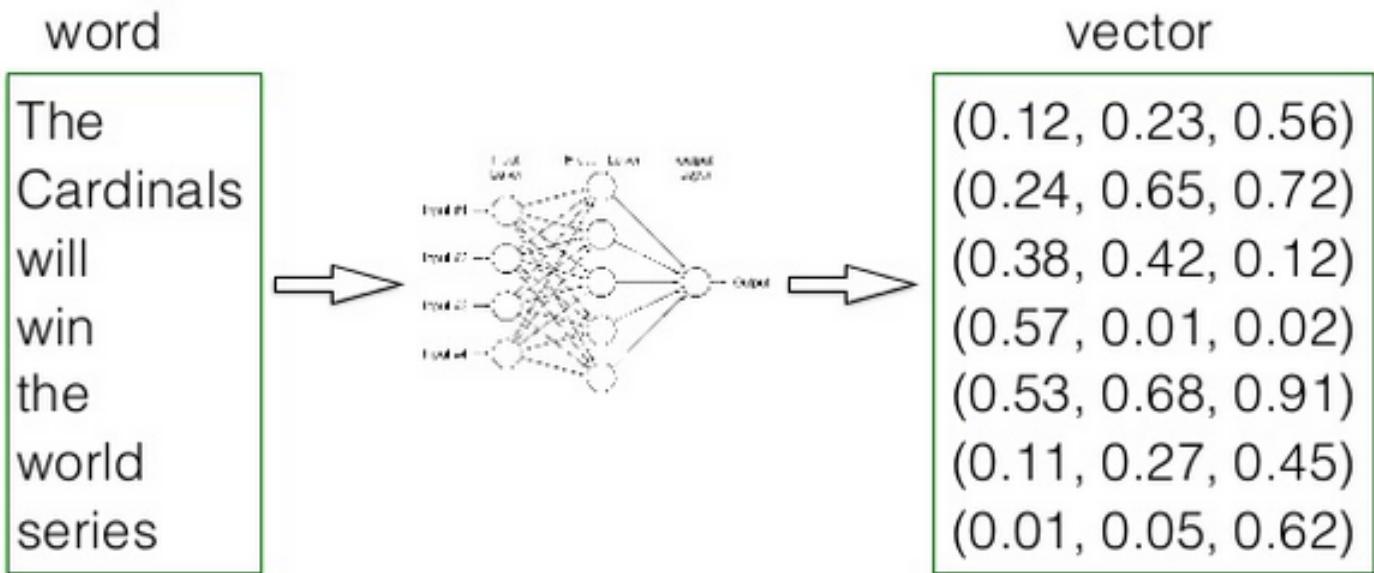


FIGURE 4.10 – Apprentissage de la matrice d'occurrence afin de représenter les mots [2]

Nous avons donc ici représenté tous les mots du texte en vecteur. Nous allons maintenant les passer dans notre réseau "Pointer-Generator".

4.5 Résultats

Pour l'apprentissage, la dimension des couches cachées des LSTMs bidirectionnels est de 256, pour l'encodeur et le décodeur.

Nous avons appris sur le modèle décrit précédemment, composé de LSTMs bidirectionnels pour encoder et décoder avec un modèle d'attention pour aider à décoder.

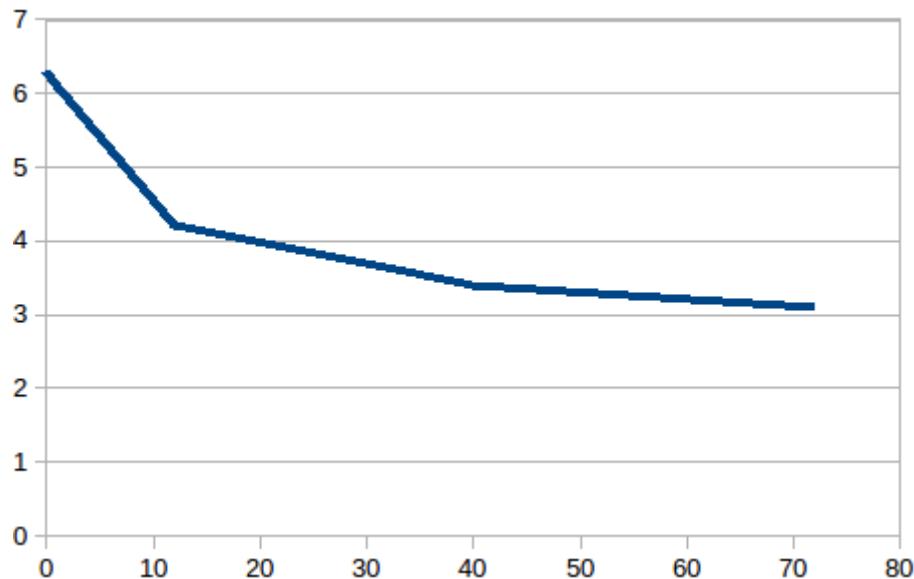


FIGURE 4.11 – L'erreur (Sans unité) en fonction du temps (en heure)

Ici l'erreur est calculée avec une exponentielle décroissante, pour avoir une courbe plus lisse. Nous remarquons que l'apprentissage se passe correctement, et que l'erreur va converger vers 3.

4.5.1 Exemple

Nous avons testé sur beaucoup d'articles, voici un exemple de résumé automatique d'un article trouvé sur internet n'ayant pas servi durant la phase d'apprentissage.

- Article : In nationwide referendum the Turkish population has voted for a change in the country's constitution. It will give the president more power and reduce the influence of parliament. 51.3% of the voters said "Yes" to a change , while the "No" side received 48.7%. For months, the population has been divided on the issue. The new constitution is the biggest change in the structure of Turkey since it was founded in the early 20th century. The referendum was a victory for Turkish President Recip Erdogan, who, together with his ruling AKP Party , called the country's people to expand presidential powers. Erdogan became Turkey's president in 2014 after being Prime Minister for over a decade. In the last few years he gained more and more power, especially after the attempted coup last summer. With the new constitution in place Erdogan could stay president until 2029. Recip Erdogan insists that the new constitution will make Turkey more modern and easier to govern. Opponents of Erdogan claim that the change will make the president too powerful and will turn the country into a dictatorship ruled by one person. They say that, in future, the president cannot be controlled or supervised by parliament or the courts. In Turkey's new constitution the president will have wide-ranging powers. He will not only be able to appoint his own ministers and choose the vice president, but also have

the power to dissolve parliament and declare a state of emergency. He will also be able to appoint judges to the highest court, similar to the American President. The European Union has been highly critical of the referendum and stated that a change towards more presidential power will not help Turkey become a member of the EU. It is afraid that, once Erdogan has more power, the country will disregard human rights and introduce the death penalty."

- **GENERATED SUMMARY** : In nationwide referendum the Turkish will give the president more power and reduce the influence of parliament. 51.3% of the voters said "Yes" to a change , while the "No" side received 48.7%. For months, together with his ruling AKP Party , called the country's people to expand presidential .

Nous pouvons voir que le résumé est juste et qu'il ne sort que des informations importantes.

4.5.2 Problèmes

- Copie :

Même si notre modèle produit des résumés de façon abstraite, nous remarquons que la formulation est souvent proche du texte. La copie de mot (l'approche d'extraction de mots) est très présente dans le modèle. Il y a donc beaucoup de paraphrases dans nos résumés, ce qui est juste, mais pas ce qu'un être humain ferait.

- Erreur sur le choix des informations importantes :

Parfois il choisit de l'information secondaire, ce qui fait que le résumé comporte des informations presque inutiles (moins importantes que d'autres informations en tout cas).

- Permutation de mots :

Parfois, le modèle permute deux mots qui se ressemblent (par exemple des noms de pays) dans la phrase. Ce qui peut rendre la phrase complètement fausse. Sur l'exemple précédent, la sortie pourrait être Argentina beat Germany 2-0, ce qui est factuellement faux.

- Utilisation d'un pronom sans le référencer avant :

Parfois il pense qu'un pronom a de l'importance et va l'utiliser pour le résumé. Cependant, quand le résumé fait plusieurs phrases, il va quelques fois oublier de le référencer avant. Nous aurons donc un "He" dans le résumé, sans savoir de qui il s'agit.

Conclusion et perspectives

Un stage de ce gabarit, dans un domaine comme l'apprentissage profond, nous a permis d'acquérir de nouveaux savoirs et de nouvelles méthodes. Ce stage, uniquement en Python, nous a permis de nous perfectionner dans ce langage de programmation orienté objet. De plus, la richesse de la librairie Tensorflow nous a fait découvrir un large éventail de fonctions d'apprentissage statistique ainsi que d'analyse et de traitement d'images.

Le stage consistait à l'implémentation d'un modèle de deep learning, permettant de classifier et d'annoter des images de documents (articles, lettres ...). Au début nous avons rencontré un certain nombre de problèmes pour l'installation de la carte graphique et de toutes les librairies utilisées pour le calcul GPU (Cuda, Cudnn ...). Mais, nous avons réussi à mettre en place différentes méthodes qui nous ont permis d'arriver à plus de 75% de bonne prédiction (pour la classification) sur plus de 16 types de documents différents. Puis pour l'annotation, nous nous sommes dirigé vers le résumé automatique de texte, mais il fallait avant cela transformer nos images en texte en utilisant la reconnaissance optique de caractère. Durant cette partie, il a été intéressant de travailler en binôme avec Alexandre Fabre pour ne pas coder des fonctions ou des modèles déjà mis en place. Cela m'a permis, étant donné qu'il travaillait sur le sujet depuis plusieurs mois déjà, grâce à ses explications, de gagner beaucoup de temps. Cependant même si cette partie fonctionne sur certains documents, la liste est loin d'être exhaustive, mais il fallait passer aux résumés automatiques. Cette partie m'était complètement inconnue, j'ai pu apprendre une nouvelle façade de l'apprentissage profond en découvrant les LSTMs, les modèles d'attentions ... Pour arriver à des résultats très satisfaisant malgré les problèmes demeurant.

A la fin de ce stage, nous voyons qu'il y a encore de gros progrès à faire dans les domaines d'OCR et de résumé automatique. Pour avoir de meilleurs résultats peut être fallait-il d'avantage pré-traiter nos données où utiliser d'autres modèles (de deep learning ou non) afin d'avoir un texte plus fiable. Il y a encore des pistes à essayer pour le futur.

Annexes

Annexe du chapitre II

1 Code source pour la création du réseau AlexNet

```
1 import tensorflow as tf
2 import numpy as np
3 def to_rgb1(im):
4     w, h = im.shape
5     ret = np.empty((w, h, 3), dtype=np.uint8)
6     ret[:, :, 0] = im
7     ret[:, :, 1] = im
8     ret[:, :, 2] = im
9     return ret
10
11 class AlexNet(object):
12
13     def __init__(self, x, keep_prob, num_classes, skip_layer,
14                  weights_path = 'DEFAULT'):
15         self.X = x
16         self.NUM_CLASSES = num_classes
17         self.KEEP_PROB = keep_prob
18         self.SKIP_LAYER = skip_layer
19
20         if weights_path == 'DEFAULT':
21             self.WEIGHTS_PATH = 'bvlc_alexnet.npy'
22         else:
23             self.WEIGHTS_PATH = weights_path
24
25         # creation of AlexNet model.
26         self.create()
27
28     def create(self):
29
30         # 1st Layer: Conv (w ReLu) -> Pool -> Lrn
31         conv1 = conv(self.X, 11, 11, 96, 4, 4, padding = 'VALID', name = 'conv1')
32         pool1 = max_pool(conv1, 3, 3, 2, 2, padding = 'VALID', name = 'pool1')
33         norm1 = lrn(pool1, 2, 2e-05, 0.75, name = 'norm1')
34
35         # 2nd Layer: Conv (w ReLu) -> Pool -> Lrn with 2 groups
36         conv2 = conv(norm1, 5, 5, 256, 1, 1, groups = 2, name = 'conv2')
37         pool2 = max_pool(conv2, 3, 3, 2, 2, padding = 'VALID', name = 'pool2')
38         norm2 = lrn(pool2, 2, 2e-05, 0.75, name = 'norm2')
39
40         # 3rd Layer: Conv (w ReLu)
```

```

41 conv3 = conv(norm2, 3, 3, 384, 1, 1, name = 'conv3')
42
43 # 4th Layer: Conv (w ReLu) splitted into two groups
44 conv4 = conv(conv3, 3, 3, 384, 1, 1, groups = 2, name = 'conv4')
45
46 # 5th Layer: Conv (w ReLu) -> Pool splitted into two groups
47 conv5 = conv(conv4, 3, 3, 256, 1, 1, groups = 2, name = 'conv5')
48 pool5 = max_pool(conv5, 3, 3, 2, 2, padding = 'VALID', name = 'pool5')
49
50 # 6th Layer: Flatten -> FC (w ReLu) -> Dropout
51 flattened = tf.reshape(pool5, [-1, 6*6*256])
52 fc6 = fc(flattened, 6*6*256, 4096, name='fc6')
53 dropout6 = dropout(fc6, self.KEEP_PROB)
54
55 # 7th Layer: FC (w ReLu) -> Dropout
56 fc7 = fc(dropout6, 4096, 4096, name = 'fc7')
57 dropout7 = dropout(fc7, self.KEEP_PROB)
58
59 # 8th Layer: FC and return unscaled activations (for tf.nn.
60 softmax_cross_entropy_with_logits)
61 self.fc8 = fc(dropout7, 4096, self.NUM_CLASSES, relu = False, name='fc8')
62
63
64 def load_initial_weights(self, session):
65
66
67 # Load weights in the memory
68 weights_dict = np.load(self.WEIGHTS_PATH, encoding = 'bytes').item()
69
70 # Loop over all the nodes
71 for op_name in weights_dict:
72
73     #Get the layer we must finetune
74     if op_name not in self.SKIP_LAYER:
75         with tf.variable_scope(op_name, reuse = True):
76
77             # Link weights to the proper variables
78             for data in weights_dict[op_name]:
79
80                 # bias
81                 if len(data.shape) == 1:
82
83                     var = tf.get_variable('biases', trainable = False)
84                     session.run(var.assign(data))
85
86                 # Weight
87                 else:
88
89                     var = tf.get_variable('weights', trainable = False)
90                     session.run(var.assign(data))
91
92

```

```

93     """
94     Layer definition
95     """
96     def conv(x, filter_height, filter_width, num_filters, stride_y, stride_x, name,
97             padding='SAME', groups=1):
98
99         # Get how many input there is
100        input_channels = int(x.get_shape()[-1])
101
102    # Convolution
103    convolve = lambda i, k: tf.nn.conv2d(i, k,
104                                         strides = [1, stride_y, stride_x, 1],
105                                         padding = padding)
106
107    with tf.variable_scope(name) as scope:
108        # Create weight and bias variables.
109        weights = tf.get_variable('weights', shape = [filter_height, filter_width,
110                                   input_channels/groups, num_filters])
111        biases = tf.get_variable('biases', shape = [num_filters])
112
113    # Convolution
114    conv = convolve(x, weights)
115
116    # Add the bias
117    bias = tf.reshape(tf.nn.bias_add(conv, biases), conv.get_shape().as_list())
118
119    # ReLU application
120    relu = tf.nn.relu(bias, name = scope.name)
121
122    return relu
123
124    def fc(x, num_in, num_out, name, relu = True):
125        with tf.variable_scope(name) as scope:
126
127            weights = tf.get_variable('weights', shape=[num_in, num_out], trainable=True)
128            biases = tf.get_variable('biases', [num_out], trainable=True)
129
130            # Multiply weight and add bias
131            act = tf.nn.xw_plus_b(x, weights, biases, name=scope.name)
132
133            if relu == True:
134                # Apply ReLU non linearity
135                relu = tf.nn.relu(act)
136                return relu
137            else:
138                return act
139
140    def max_pool(x, filter_height, filter_width, stride_y, stride_x, name, padding='SAME'):
141
142        return tf.nn.max_pool(x, ksize=[1, filter_height, filter_width, 1],
143                             strides = [1, stride_y, stride_x, 1],
144                             padding = padding, name = name)

```

```

144
145 def lrn(x, radius, alpha, beta, name, bias=1.0):
146     return tf.nn.local_response_normalization(x, depth_radius = radius, alpha = alpha,
147                                              beta = beta, bias = bias, name = name)
148
149 def dropout(x, keep_prob):
150     return tf.nn.dropout(x, keep_prob)

```

2 Data

```

1 #Dataset class creation (to process them and put them randomly in batch)
2 path="/home/kergann/STAGE/Donnee_Traitement/TobaccoTrue(36G)/rvl-cdip(2)/images/"
3 class ImageDataGenerator:
4     def __init__(self, class_list, horizontal_flip=False, shuffle=False,
5                  mean = np.array([104., 117., 124.]), scale_size=(227, 227),
6                  nb_classes = 16):
7         # Initialisation
8         self.horizontal_flip = horizontal_flip
9         self.n_classes = nb_classes
10        self.shuffle = shuffle
11        self.mean = mean
12        self.scale_size = scale_size
13        self.pointer = 0
14        self.read_class_list(class_list)
15        if self.shuffle:
16            self.shuffle_data()
17
18    def read_class_list(self,class_list):
19        """
20        Get images and labels.
21        """
22        with open(class_list) as f:
23            lines = f.readlines()
24            self.images = []
25            self.labels = []
26            n_files_train=100000
27            h=0
28            lines_rand=np.zeros(n_files_train).tolist()
29            while h<n_files_train:
30                lines_rand[h]=str(random.choice(lines))
31                h+=1
32                for l in lines_rand:
33                    items = l.split()
34                    self.images.append(items[0])
35                    self.labels.append(int(items[1]))
36
37        self.data_size = len(self.labels)
38
39    def shuffle_data(self):
40        """
41        Randomly mix the data
42        """
43        images = list(self.images)

```

```

44     labels = list(self.labels)
45     self.images = []
46     self.labels = []
47     # Creation of the new index list
48     idx = np.random.permutation(len(labels))
49     for i in idx:
50         self.images.append(images[i])
51         self.labels.append(labels[i])
52
53     def reset_pointer(self):
54         self.pointer = 0
55
56         if self.shuffle:
57             self.shuffle_data()
58
59     def next_batch(self, batch_size):
60         #Take next_batch and the corresponding labels
61         paths = self.images[self.pointer:self.pointer + batch_size]
62         labels = self.labels[self.pointer:self.pointer + batch_size]
63         #print(paths)
64         self.pointer += batch_size
65         images = np.ndarray([batch_size, self.scale_size[0], self.scale_size[1], 3])
66         for i in range(len(paths)):
67             img = mpimg.imread(path+paths[i])
68             img = to_rgb1(img)
69             #Application of an horizontal flip if we want
70             if self.horizontal_flip and np.random.random() < 0.5:
71                 img = cv2.flip(img, 1)
72             size_image = 227
73             # resize to get a standard size for all image
74             img = imresize(img, (size_image, size_image, 3))
75             img = img.astype(np.float32)
76             img -= self.mean
77             images[i] = img
78
79         # Creation of the New labels in a new matrix (16*batchsize). For one input (one line),
80         # put 1 on the right label placement and 0 otherwisel 'endroit correspondant au label
81         #
82         one_hot_labels = np.zeros((batch_size, self.n_classes))
83         for i in range(len(labels)):
84             one_hot_labels[i][labels[i]] = 1
85             #print("Y",i)
86             print("100 done")
87         return images, one_hot_labels

```

3 Transfer Learning sur AlexNet

```

1 import os
2 import numpy as np
3 import tensorflow as tf
4 from datetime import datetime
5 from alexnet import AlexNet
6 from datagenerator import ImageDataGenerator

```

```

7  from glob import glob
8  import io
9  import random
10 import matplotlib.image as mpimg
11 from scipy.misc import imresize
12
13 class_names=['letter', 'form', 'email', 'handwritten', 'advertisement', 'scientific
    report', 'scientific publication', 'specification',
    'file folder', 'new article', 'budget', 'invoice', 'presentation', '
    questionnaire', 'resume', 'memo']
14
15 # Paths
16 Locationtest="/home/kergann/STAGE/Donnee_Traitement/TobaccoTrue(36G)/rvl-cdip(2)/
    labels/test.txt"
17 Fichiertest=sorted(glob(Locationtest))
18 OpenTest=open(Locationtest)
19 lignesTest=OpenTest.readlines()
20 path="/home/kergann/STAGE/Donnee_Traitement/TobaccoTrue(36G)/rvl-cdip(2)/images/"
21 train_file = '/home/kergann/STAGE/Donnee_Traitement/TobaccoTrue(36G)/rvl-cdip(2)/
    labels/train.txt'
22 val_file = '/home/kergann/STAGE/Donnee_Traitement/TobaccoTrue(36G)/rvl-cdip(2)/labels
    /test.txt'
23
24 # parameters
25 learning_rate = 0.01
26 num_epochs = 250
27 batch_size = 100
28 dropout_rate = 0.5
29 num_classes = 16
30 train_layers = ['fc8', 'fc7'] # We can choose which layers we want to finetune
31
32 # Path on where to save
33 filewriter_path = "model_epoch/finetune_alexnet/DocumentClassif"
34 checkpoint_path = "model_epoch/finetune_alexnet/"
35
36 # Input and output tensor
37 x = tf.placeholder(tf.float32, [batch_size, 227, 227, 3])
38 y = tf.placeholder(tf.float32, [None, num_classes])
39 keep_prob = tf.placeholder(tf.float32)
40
41 # Initialisation
42 model = AlexNet(x, keep_prob, num_classes, train_layers)
43 score = model.fc8
44 # List of variables which will be trained
45 var_list = [v for v in tf.trainable_variables() if v.name.split('/')[0] in
    train_layers]
46
47 # Loss calculation
48 score = model.fc8
49 with tf.name_scope("cross_ent"):
50     loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = score,
        labels = y))
51
52

```

```

53 # Training
54 with tf.name_scope("train"):
55     # Get gradients of all trainable variables
56     gradients = tf.gradients(loss, var_list)
57     gradients = list(zip(gradients, var_list))
58
59     # Create optimizer and apply gradient descent to the trainable variables
60     optimizer = tf.train.GradientDescentOptimizer(learning_rate)
61     train_op = optimizer.apply_gradients(grads_and_vars=gradients)
62
63     # visualisation in Tensorboard
64     for gradient, var in gradients:
65         tf.summary.histogram(var.name + '/gradient', gradient)
66     for var in var_list:
67         tf.summary.histogram(var.name, var)
68     tf.summary.scalar('cross_entropy', loss)
69
70     # Prediction and accuracy calculation
71     with tf.name_scope("accuracy"):
72         label_prediction = tf.argmax(score, 1, name='predicted_label')
73         correct_pred = tf.equal(tf.argmax(score, 1), tf.argmax(y, 1))
74         accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
75
76     # Tensorboard visualisation
77     tf.summary.scalar('accuracy', accuracy)
78     merged_summary = tf.summary.merge_all()
79
80     # Initialisation to save
81     writer = tf.summary.FileWriter(filewriter_path, graph=tf.get_default_graph())
82     saver = tf.train.Saver()
83
84     # Initialize the data generator seperately for the training and validation set
85
86     train_generator = ImageDataGenerator(train_file, horizontal_flip = True, shuffle = True)
87     val_generator = ImageDataGenerator(val_file, shuffle = False)
88
89     # Batches
90     train_batches_per_epoch = np.floor(train_generator.data_size / batch_size).astype(np.int16)
91     val_batches_per_epoch = np.floor(val_generator.data_size / batch_size).astype(np.int16)
92
93     # Session start
94     with tf.Session() as sess:
95         # Initialisation
96         sess.run(tf.global_variables_initializer())
97
98         # Tensorboard visualisation of the graph
99         writer.add_graph(sess.graph)
100
101        # Load weights either initial or from a checkpoint
102        #model.load_initial_weights(sess)

```

```

103 saver.restore(sess, "model_epoch/finetune_alexnet/model_epoch13.ckpt")
104
105 print("{}".format(datetime.now()))
106 print("{} Open Tensorboard at --logdir {}".format(datetime.now(),
107                                                 filewriter_path))
108 #Loop
109 for epoch in range(num_epochs):
110
111     print("{}".format(datetime.now(), epoch+1))
112     step = 1
113     while step < train_batches_per_epoch:
114         batch_xs, batch_ys = train_generator.next_batch(batch_size)
115         # Training
116         sess.run(train_op, feed_dict={x: batch_xs,
117                                       y: batch_ys,
118                                       keep_prob: dropout_rate})
119         # Get the variables we wanted
120         if step%display_step == 0:
121             s = sess.run(merged_summary, feed_dict={x: batch_xs,
122                                                     y: batch_ys,
123                                                     keep_prob: 1.})
124             writer.add_summary(s, epoch*train_batches_per_epoch + step)
125             acctr = sess.run(accuracy, feed_dict={x: batch_xs,
126                                                   y: batch_ys,
127                                                   keep_prob: 1.})
128             correct= sess.run(correct_pred,feed_dict={x: batch_xs,
129                                                       y: batch_ys,
130                                                       keep_prob: 1.})
131             label_prediction=sess.run(label_prediction,feed_dict={x: batch_xs,
132                                                       y: batch_ys, keep_prob: 1.})
133             step += 1
134             print("acc de train" ,acctr)
135             print("nbr correct train", np.sum(correct))
136             print("label", label_prediction)
137
138     # Test
139     print("{}".format(datetime.now()))
140     test_acc = 0.
141     test_count = 0
142     for _ in range(val_batches_per_epoch):
143         batch_tx, batch_ty = val_generator.next_batch(batch_size)
144
145         acc = sess.run(accuracy, feed_dict={x: batch_tx,
146                                           y: batch_ty,
147                                           keep_prob: 1.})
148
149         correct_test= sess.run(correct_pred,feed_dict={x: batch_tx,
150                                                       y: batch_ty,
151                                                       keep_prob: 1.})
152         label_prediction=sess.run(label_prediction,feed_dict={x: batch_tx,
153                                                       y: batch_ty,
154                                                       keep_prob: 1.})
155         test_acc += acc

```

```

156     test_count += 1
157     print("nbr correct test", np.sum(correct_test))
158     print("acc de 100 test", acc)
159     print("le label", label_prediction)
160
161     test_acc /= test_count
162     print(test_acc)
163     val_generator.reset_pointer()
164     train_generator.reset_pointer()
165     print("{} Saving checkpoint of model...".format(datetime.now()))
166
167     # Save
168     checkpoint_name = os.path.join(checkpoint_path, 'model_epoch'+str(epoch+1)+'.'
169                                     'ckpt')
170     save_path = saver.save(sess, checkpoint_name)
171
172     print("{} Model checkpoint saved at {}".format(datetime.now(), checkpoint_name))

```

Annexe correspondant au chapitre III

1 Modèle séquence to séquence

```
1 class SummarizationModel(object):
2     "A class for the sequence-to-sequence model, with pointer-generator mode"
3
4     def __init__(self, hps, vocab):
5         self._hps = hps
6         self._vocab = vocab
7
8     def _add_placeholders(self):
9         hps = self._hps
10    # encoder part
11    self._enc_batch = tf.placeholder(tf.int32, [hps.batch_size, None], name='enc_batch')
12    self._enc_lens = tf.placeholder(tf.int32, [hps.batch_size], name='enc_lens')
13    if FLAGS.pointer_gen:
14        self._enc_batch_extend_vocab = tf.placeholder(tf.int32, [hps.batch_size, None], name='enc_batch_extend_vocab')
15        self._max_art_oovs = tf.placeholder(tf.int32, [], name='max_art_oovs')
16    # decoder part
17    self._dec_batch = tf.placeholder(tf.int32, [hps.batch_size, hps.max_dec_steps], name='dec_batch')
18    self._target_batch = tf.placeholder(tf.int32, [hps.batch_size, hps.max_dec_steps], name='target_batch')
19    self._padding_mask = tf.placeholder(tf.float32, [hps.batch_size, hps.max_dec_steps], name='padding_mask')
20
21    if hps.mode=="decode" and hps.coverage:
22        self.prev_coverage = tf.placeholder(tf.float32, [hps.batch_size, None], name='prev_coverage')
```

Fonction d'encodeur et de décodeur utilisé.

```
1 def _add_encoder(self, encoder_inputs, seq_len):
2     with tf.variable_scope('encoder'):
3         cell_fw = tf.contrib.rnn.LSTMCell(self._hps.hidden_dim, initializer=self.
4             rand_unif_init, state_is_tuple=True)
5         cell_bw = tf.contrib.rnn.LSTMCell(self._hps.hidden_dim, initializer=self.
6             rand_unif_init, state_is_tuple=True)
7         (encoder_outputs, (fw_st, bw_st)) = tf.nn.bidirectional_dynamic_rnn(cell_fw,
8             cell_bw, encoder_inputs, dtype=tf.float32, sequence_length=seq_len,
9             swap_memory=True)
10    encoder_outputs = tf.concat(axis=2, values=encoder_outputs) # concatenate the
```

```

    forwards and backwards states
7   return encoder_outputs, fw_st, bw_st

1 def _add_decoder(self, inputs):
2     hps = self._hps
3     cell = tf.contrib.rnn.LSTMCell(hps.hidden_dim, state_is_tuple=True, initializer=
4         self.rand_unif_init)

5     prev_coverage = self.prev_coverage if hps.mode=="decode" and hps.coverage else
6     None # In decode mode, we run attention_decoder one step at a time and so need to pass in
7     the previous step's coverage vector each time

8     outputs, out_state, attn_dists, p_gens, coverage = attention_decoder(inputs, self
9         ._dec_in_state, self._enc_states, cell, initial_state_attention=(hps.mode==""
10        "decode"), pointer_gen=hps.pointer_gen, use_coverage=hps.coverage,
11        prev_coverage=prev_coverage)

12    return outputs, out_state, attn_dists, p_gens, coverage

```

Puis nous l'ajoutons créons le graphe.

```

1 def _add_seq2seq(self):
2     hps = self._hps
3     vszie = self._vocab.size() # size of the vocabulary
4
5     with tf.variable_scope('seq2seq'):
6         # Some initializers
7         self.rand_unif_init = tf.random_uniform_initializer(-hps.rand_unif_init_mag, hps
8             .rand_unif_init_mag, seed=123)
9         self.trunc_norm_init = tf.truncated_normal_initializer(stddev=hps.
10             trunc_norm_init_std)
11         # Add embedding matrix (shared by the encoder and decoder inputs)
12         with tf.variable_scope('embedding'):
13             embedding = tf.get_variable('embedding', [vszie, hps.emb_dim], dtype=tf.
14                 float32, initializer=self.trunc_norm_init)
15             if hps.mode=="train": self._add_emb_vis(embedding) # add to tensorboard
16             emb_enc_inputs = tf.nn.embedding_lookup(embedding, self._enc_batch) # tensor
17                 with shape (batch_size, max_enc_steps, emb_size)
18             emb_dec_inputs = [tf.nn.embedding_lookup(embedding, x) for x in tf.unstack(
19                 self._dec_batch, axis=1)] # list length max_dec_steps containing shape (
20                 batch_size, emb_size)
21
22         # Add the encoder.
23         enc_outputs, fw_st, bw_st = self._add_encoder(emb_enc_inputs, self._enc_lens)
24         self._enc_states = enc_outputs
25
26         # Add the decoder.
27         with tf.variable_scope('decoder'):
28             decoder_outputs, self._dec_out_state, self.attn_dists, self.p_gens, self.
29                 coverage = self._add_decoder(emb_dec_inputs)
30
31         # Add the output projection to obtain the vocabulary distribution
32         with tf.variable_scope('output_projection'):
33             w = tf.get_variable('w', [hps.hidden_dim, vszie], dtype=tf.float32,
34                 initializer=self.trunc_norm_init)

```

```

27     w_t = tf.transpose(w)
28     v = tf.get_variable('v', [vsize], dtype=tf.float32, initializer=self.
29         trunc_norm_init)
30     vocab_scores = [] # vocab_scores is the vocabulary distribution before applying
31         softmax. Each entry on the list corresponds to one decoder step
32     for i, output in enumerate(decoder_outputs):
33         if i > 0:
34             tf.get_variable_scope().reuse_variables()
35         vocab_scores.append(tf.nn.xw_plus_b(output, w, v)) # apply the linear layer
36     vocab_dists = [tf.nn.softmax(s) for s in vocab_scores] # The vocabulary
37         distributions.
38
39     # For pointer-generator model, calc final distribution from copy distribution and
40         vocabulary distribution , then take log
41     if FLAGS.pointer_gen:
42         final_dists = self._calc_final_dist(vocab_dists, self.attn_dists)
43         # Take log of final distribution
44         log_dists = [tf.log(dist) for dist in final_dists]
45     else: # just take log of vocab_dists
46         log_dists = [tf.log(dist) for dist in vocab_dists]

```

Bibliographie

- [1] Tutorialsplay's administrator. Tutorialsplay. <https://www.toptal.com/machine-learning/machine-learning-theory-an-introductory-primer/>, 2014.
- [2] Allen. Word2vec – deep learning. <http://www.lifestyletrading101.com/word2vec-deep-learning/>, 2016.
- [3] Yassine Alouini. Deep learning with keras : convolutional neural networks demystified. <https://dsotb.quora.com/Deep-learning-with-Keras-convolutional-neural-networks-demystified>, 2016.
- [4] Jason Brownlee. What is deep learning? <<http://machinelearningmastery.com/what-is-deep-learning/>>, 2016.
- [5] colah. Understanding lstm networks. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>, 2015.
- [6] DanVanderkam. Extracting text from an image using ocropus. <http://www.danvk.org/2015/01/09/extracting-text-from-an-image-using-ocropus.html>, 2015.
- [7] Pierre-Alain Jachiet. Classification d'image : Les réseaux de neurones convolutifs en toute simplicité. <https://fr.linkedin.com/pulse/classification-dimage-les-r%C3%A9seaux-de-neurones-en-toute-jachiet>, 2016.
- [8] Jim Joong Kim. Tensorflow dev summit 2017. <https://www.slideshare.net/hackergolbin/tensorflow-dev-summit-2017>, 2017.
- [9] leonardblier. Attention mechanism. <https://blog.heuritech.com/2016/01/20/attention-mechanism/>, 2016.
- [10] None. <http://www.voidcn.com/article/p-xzaufdsw-dx.html>, 2016.
- [11] None. Rnn. <http://dlwiki.finfra.com/algorithm:rnn>, 2016.
- [12] None. Cnn. <http://www.voidcn.com/article/p-metaqybz-zq.html>, 2017.
- [13] Gabriel Peyré. Le traitement numérique des images. <http://images.math.cnrs.fr/Le-traitement-numerique-des-images.html>, 2011.
- [14] Abigail See. Get to the point : Summarization with pointer-generator networks. <https://arxiv.org/pdf/1704.04368.pdf>, 2017.
- [15] ujjwalkarn. An intuitive explanation of convolutional neural networks. <<https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets>>, 2016.

