Rapport MOOC Deep Learning

Thomas Kowalski

Avril 2019

Table des matières

1 Motivations **2**

2 Présentation des cours **2**

3 Avancement dans les cours **3**

4 Contenu des cours **3**
4.1 Introduction . . . 3
4.2 Vectorisation . . . 4
4.3 Hyperparamètres . . . 4
4.4 Séparation entraînement / tests . . . 5
4.5 Régularisation (L_1 ou L_2) . . . 5
4.6 *Dropout* . . . 5
4.7 Normalisation des entrées . . . 6
4.8 Descente du gradient par mini-batch . . . 6
4.9 Descente de gradient avec "inertie" . . . 6
4.10 *Root Mean Square Prop* . . . 6
4.11 Adam . . . 6
4.12 Diminution de α . . . 7
4.13 Amélioration des hyper-paramètres . . . 7
4.14 *Batch-normalization* . . . 7
4.15 Classification multi-classes et activation *softmax* . . . 7
4.16 Introduction à TensorFlow . . . 8
4.17 Réseaux de neurones convolutifs (CNN) . . . 9
4.18 Algorithmes de détection et localisation . . . 9

5 Travaux Pratiques **11**
5.1 Régression linéaire . . . 11
5.2 Classification planaire à une couche cachée . . . 11
5.3 Implémentation d'un réseau complet . . . 12
5.4 Utilisation du réseau complet implémenté juste avant . . . 12
5.5 Régularisation . . . 13
5.6 Modèle convolutif pas-à-pas . . . 14
5.7 Application du modèle convolutif . . . 14
5.8 Tutoriel Keras "The Happy House" . . . 15
5.9 Car detection with YOLO . . . 15

6 Projet : Création d'un réseau de reconnaissance de chiffres **17**
6.1 Introduction . . . 17
6.2 Choix des données . . . 17
6.3 Préparation des données . . . 17
6.4 Application de traitements aux données . . . 17
6.5 Présentation du modèle . . . 18

7 Conclusion **20**

1 Motivations

Le *deep learning* est une discipline qui a attiré mon attention dès sa démocratisation. Pendant les dernières années, et particulièrement pendant mon parcours en classes préparatoires, son explosion a permis à des entreprises de proposer des solutions et produits innovants, en utilisant les capacités calculatoires jusque-là jamais atteintes des ordinateurs modernes. De la classification de caractères à l'identification et au positionnement d'objets complexes sur des images de grande résolution, en passant par la génération d'images ou la recommandation personnalisée de nouveaux produits à l'utilisateur, le *deep learning* (au travers de "sous-disciplines" telles que le *computer vision*) semble transcender non pas seulement l'industrie informatique, mais l'ensemble de la société et des moyens de consommation contemporains.

Malheureusement, ayant choisi le parcours *Génie Logiciel* à l'ENSIIE, aucun cours relatif au *deep learning* ne m'était accessible. De plus, nombre des cours de statistiques et apprentissage automatique enseignés à l'ENSIIE me semblent aborder beaucoup de points théoriques, ce qui me semblait difficile à concilier avec en parallèle, le suivi d'un parcours d'informatique théorique en GL.

C'est pourquoi j'ai vu cette opportunité de MOOC au S4 (remplaçant l'UE SE2, qui était redondante avec ARMA) comme une chance unique de m'intéresser de plus près, et de m'initier au *deep learning*, afin d'en connaître au moins quelques éléments essentiels à une compréhension plus poussée du sujet à terme.

2 Présentation des cours

Après avoir cherché et comparé les MOOC sur plusieurs sites dédiés, j'ai trouvé la *Deep Learning Specialization* sur Coursera, et elle m'a semblé être la solution optimale.

coursera.org/specializations/deep-learning

En effet, elle aborde beaucoup de points différents, en partant des bases. Elle se divise en cinq parties :

- *Neural Networks and Deep Learning*, qui pose les bases des réseaux de neurones ainsi que les définitions mathématiques associées, et les "réflexes à prendre" ;
- *Improving Deep Neural Networks : Hyperparameter tuning, Regularization and Optimization*, qui se concentre sur des aspects à la fois pratiques et théoriques. En se basant sur les acquis du premier cours, elle expose différents moyens d'améliorer le fonctionnement des algorithmes et modèles utilisés (convergence plus rapide, apprentissage plus rapide, convergence améliorée dans les cas où le gradient est faible...) ;
- *Structuring Machine Learning Projects*, qui se concentre sur les bonnes pratiques à suivre dans le cadre des grands projets de *machine learning* ;
- *Convolutional Neural Networks* qui traite des réseaux de neurones convolutionnels (CNN) ;
- *Sequence Models* qui se spécialise sur les *sequence models*.

Ne disposant que d'un demi semestre, il me paraissait évident que mon apprentissage et mon expérience seraient limités par le temps. Vu la charge de travail annoncée par Coursera et malgré le suivi du cours également sur mon temps libre, il me paraissait impossible de suivre les cinq cours. C'est pourquoi j'ai décidé de prioriser ceux-ci :

- *Neural Networks and Deep Learning* (18 heures) car je n'avais encore aucune base en deep learning ;
- *Improving Deep Neural Networks : Hyperparameter tuning, Regularization and Optimization* (15 heures) car il était relativement court et me semblait crucial pour des implémentations efficaces et une compréhension complète des cours suivants ;
- *Convolutional Neural Networks* (21 heures) car le *computer vision* est une application classique, si ce n'est majoritaire, de l'apprentissage profond aujourd'hui.

Soit un total de 54 heures.

En pratique, j'ai pu suivre ces trois cours, mais n'ai pas eu le temps d'aller jusqu'à la fin du troisième.

3 Avancement dans les cours

N'ayant pas souhaité payer les 200 euros nécessaires pour l'obtention d'une certification pour le MOOC, je n'ai pas pu participer aux rendus de TP, ni aux quizz de connaissances se trouvant directement après la fin de cours. J'ai cependant pu accéder à tous les TP sauf deux (sous forme de *notebooks* Jupyter) et ai pu vérifier mes résultats (les résultats attendus étant fournis à chaque fois).

4 Contenu des cours

Le cours étant assez dense, j'ai décidé de ne faire qu'un résumé et des explications sommaires des concepts et techniques qui m'avaient été présentés.

4.1 Introduction

Le premier cours avait pour principal objectif de présenter les bases du deep learning.

L'objectif principal du deep learning est de prédire (problèmes de classification ou de régression) des valeurs en fonction d'entrées. Pour cela, on utilise des réseaux de calculs (appelés "réseaux de neurones") prenant en entrée les informations du problèmes et fournissant en sortie un réel, un entier, plusieurs réels... Le tout en ayant effectué entre les deux beaucoup de calculs avec des coefficients connus à l'avance.

Pour connaître ces coefficients, on doit d'abord passer par une phase d'entraînement du réseau, dans laquelle on utilise des entrées dont la sortie est connue pour "apprendre" la fonction que l'on souhaite prédire.

L'apprentissage s'effectue en deux étapes. Pour chaque exemple donné, on demande une prédiction au réseau. Pour cela, si x est le vecteur d'entrée, il calcule itérativement $z_1 = w_1 x$ puis $a_1 = f_1(z_1)$, $z_2 = w_2 z_1$ puis $a_2 = f_2(a_1)$, etc. jusqu'à la dernière couche du réseau, qui peut avoir plus ou moins de coordonnées. Cette étape s'appelle la *forward propagation* et les w_1 sont des matrices que l'étape d'apprentissage cherche à déterminer.

Pour pouvoir apprendre, un réseau a besoin de pouvoir définir ce qui est un résultat satisfaisant et ce qui ne l'est pas. Dans le cas le plus simple (classification binaire), on juge un résultat satisfaisant si le réseau répond "1" alors que la réponse était "1", "0" alors que la réponse était "0" et non satisfaisant sinon.
On peut dès lors poser une fonction de coût qui représente "combien le réseau s'est trompé". En effet, bien que les étiquettes soient 0 et 1 dans notre cas, le calcul étant fait avec des vecteurs et matrices réels, $\hat{y} \in [0; 1]$ n'est pas nécessairement entier. Par exemple :

$$\frac{1}{2}(y - \hat{y})^2$$

Dans la pratique, on utilise également souvent le *cross-entropy loss* :

$$-\left(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right)$$

Cette fonction est celle que l'on souhaite minimiser lors de l'apprentissage. En première approximation, on se dit que plus le coût est faible, moins on fait d'erreurs de prédiction. Reste à savoir comment optimiser cette fonction.

De plus, on peut poser le coût "total" (sur tous les exemples) qui est le suivant :

$$\frac{1}{n} \sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

Remarque En anglais, plusieurs terminologies existent : le *loss* qui représente l'erreur sur un exemple et le *cost* qui représente l'erreur sur plusieurs exemples (généralement la moyenne des *loss* avec éventuellement un terme correctif). En français, j'utiliserai toujours le *coût* pour les deux concepts.

L'idée de la *backpropagation* de mettre à jour les poids du réseau de neurones conformément à leur incidence sur la prédiction (et donc sur la fonction de coût).

Pour optimiser le coût, on utilise un algorithme de descente du gradient. Pour cela, on calcule la dérivée de la fonction de coût (analytiquement, que l'on insère dans notre programme). Grâce à la règle des dérivées en chaîne, on peut obtenir l'incidence de chaque terme sur la fonction de coût.

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \frac{\partial y}{\partial x}$$

Pour "descendre" le gradient, il suffit donc de calculer le gradient au point actuel (évaluation de la dérivée), par rapport à chaque paramètre (un paramètre = une valeur que l'on cherche à apprendre), et à appliquer l'itération classique :

$$x_{i+1} = x_i - \alpha \times \frac{\partial \mathcal{J}}{\partial x_i}$$

Avec $\mathcal{J}$ la fonction de coût. Le $-$ vient du fait que si on va dans le sens du gradient positif, on maximise la fonction.

Remarque Le calcul de $\frac{\partial \mathcal{J}}{\partial x_i}$ se fait analytiquement à l'avance avec les règles de dérivation classiques (dérivée d'une somme, d'un produit, d'une composée...).

4.2 Vectorisation

Jusque là, tous les exemples ont été faits sur des versions itératives des algorithmes. Pour chaque entrée, on calcule itérativement les valeurs de chaque neurone de chaque couche. Dans la pratique, les performances de cette méthode sont très décevantes, pour plusieurs raisons. La première raison, qui semble évidente, est que nous utilisons Python (implémentation CPython ou Jupyter), qui n'est pas particulièrement rapide sans les extensions C. Itérer sur les collections est donc lent et l'apprentissage prend beaucoup de temps.

Pour résoudre ce problème, on souhaite vectoriser le modèle. Le principe de la vectorisation est le suivant. Au lieu d'avoir un vecteur par neurone de chaque couche, on a une matrice par couche. Dès lors, on peut évaluer tous les résultats d'une même couche de matrice W pour un exemple X en calculant WX.

L'énorme avantage de cette technique est que l'on peut alors utiliser des modules annexes (comme `numpy`) qui sont beaucoup plus rapides qu'une implémentation à la main en Python (par exemple `numpy` contient beaucoup de fonctionnalités écrites en C et simplement appelées par Python, ce qui améliore grandement le temps de calcul, notamment pour le calcul matriciel).

Dans un second temps, on peut vectoriser encore plus. Au lieu de simplement calculer les résultats pour chaque couche pour un exemple à la fois, on peut transformer notre vecteur entrée x en une matrice $X = (x_i)$ où chaque colonne est un exemple. On obtient ainsi immédiatement avec le produit $W_0 \times X$ les résultats de la première couche pour chaque exemple de la matrice, que l'on peut passer dans la fonction d'activation (simple en Python grâce au *broadcasting*).
En bref, le résultat pour plusieurs exemples peut s'obtenir de la façon suivante. Prenons X la matrice d'entrée, W_i les poids de la couche i, et f_i la fonction d'activation de la couche i.

$$\begin{aligned}
&Z_1 = W_1 X \ \textit{(}Z_1 \textit{ est une matrice)}\\
&A_1 = f_1(Z_1)\\
&Z_2 = W_2 A_1\\
&A_2 = f_2(Z_2)\\
&...\\
&Z_n = W_n A_{n-1}\\
&\textit{(Sortie) } A_n = f_n(Z_n)
\end{aligned}$$

Remarque Chaque fonction d'activation f_i doit être non-linéaire. En effet, si les fonctions d'activation de deux couches successives sont linéaires, alors l'action de ces deux couches sur leur entrée est elle-même une application linéaire $(W_{i+1} \circ f_i \circ W_i)$; ces deux couches sont donc équivalentes à une couche (différente).

4.3 Hyperparamètres

Avant d'aborder le chapitre suivant, il semble important d'aborder la question des *hyperparamètres*. En deep learning, on appelle hyperparamètres tous les paramètres qui sont définis (et restent les mêmes) avant le début de l'apprentissage. Par exemple, on peut citer :

— Le *learning rate* α ;
— Le nombre de couches ;
— Les formes de ces couches (nombre de neurones) ;
— Les fonctions d'activation.

4.4 Séparation entraînement / tests

Lors de l'élaboration d'un réseau de neurones, il est important de vérifier le fonctionnement de celui-ci en pratique. En effet, lors de l'entraînement, des effets indésirables (tels que l'*overfitting*) peuvent apparaître, et ceux-ci ne peuvent pas être détectés directement.

L'*overfitting* apparaît lorsque le réseau cherche trop à correspondre aux données d'apprentissage. Les prédictions sur le *set* d'entraînement sont très bonnes, mais dès lors qu'une nouvelle entrée apparaît, les prédictions sont aléatoires, comme l'illustre ce dessin :

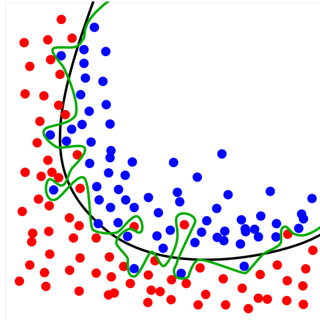


FIGURE 1 – *Overfitting* dans le plan. Alors que la limite de décision noire serait la plus adaptée au problème, le modèle s'est spécialisé au point d'avoir une précision de 100% sur les données d'entraînement ; le résultat sera en revanche bien moins intéressant avec la bordure verte qu'avec la bordure noire sur des données réelles.

Ce problème ne peut être détecté qu'en testant le réseau avec des données inconnues. C'est une des raisons qui poussent à diviser le *dataset* en deux parties : une partie *entraînement* et une partie *test*. Une fois le réseau entraîné, on utilise les données du *dataset* de test pour vérifier que les résultats sur des nouvelles données sont bons. Il arrive souvent qu'ils soient moins bons que sur les données d'entraînement, l'important est qu'ils restent satisfaisants. Dans le cas où ils ne le sont pas, on peut appliquer diverses techniques pour atténuer l'effet (que l'on verra plus tard).

4.5 Régularisation (L_1 ou L_2)

Le principe de la *régularisation* est de changer l'expression du coût sur n exemples :

$$\mathcal{J}(w,b) = \frac{1}{n} \sum_{exemples} \mathcal{L}(\hat{y}, y) + \frac{\lambda}{2l} \sum_{couches} \|w\|_2^2$$

Ce qui a pour effet de limiter la norme des matrices des couches cachées (en d'autres termes, cela pousse les coefficients des matrices à être petits).

La régularisation est particulièrement utile pour limiter l'*overfitting*.

4.6 *Dropout*

Le *dropout* est une autre méthode de régularisation, qui permet de limiter l'*overfitting*. Le principe est d'"éteindre" certains neurones de certaines couches à chaque itération de l'apprentissage, en les prenant au hasard et avec une certaine possibilité.

En d'autres termes, on peut poser le vecteur d (vecteur de *dropout*) défini tel que d est de même dimension que la "sortie" de la couche (appelons-la a) :

$$d^{(i)} = \begin{cases} 0 \text{ si } \texttt{rand()} < P \\ 1 \text{ sinon} \end{cases} \tag{1}$$

Puis, on obtient la version *dropout* de la sortie en faisant $a' = \frac{1}{P} a \cdot d$. La multiplication par $\frac{1}{P}$ se fait pour des raisons de normalisation.

Informellement, l'idée de la régularisation par *dropout* est de forcer le réseau à répartir le poids dans les matrices W_i en ne lui permettant pas de "compter" sur un neurone en particulier (puisque chaque neurone peut à tout moment être désactivé pendant l'apprentissage). Le réseau est alors obligé de répartir le poids de la matrice plus uniformément.

4.7 Normalisation des entrées

On souhaite que les entrées soient aussi uniformes que possible. En effet, dans le cas où une coordonnée du vecteur d'entrée serait sur une échelle beaucoup plus grande que les autres. Alors, on serait obligé d'utiliser un α très petit pour ne pas "louper" le minimum sur les coordonnées de petite échelle.

$$\mu = \frac{1}{n} \sum_{exemples} x^{(i)}$$

$$\sigma^2 = \frac{1}{n} \sum_{exemples} \left(\mu - x^{(i)}\right)^2 \text{ pt-à-pt}$$

Pour normaliser, on applique à chaque exemple x l'opération

$$x = \frac{1}{\|\sigma^2\|}(x - \mu)$$

4.8 Descente du gradient par mini-batch

La version actuelle de la descente du gradient du cours est la suivante pour N exemples :

- Répéter ... fois
  - Calculer le résultat pour les N exemples par le réseau ;
  - Déterminer le coût sur les résultats des exemples ;
  - Faire la *backpropagation* en en tenant compte.

Le problème apparaît lorsque le nombre d'exemples utilisé devient grand. En effet, on peut se retrouver à calculer le résultat pour 5000000 d'exemples à travers un réseau très mauvais avant d'effectuer un premier pas. La résolution de ce problème peut se faire par l'utilisation de *mini-batches*.

L'idée des *mini-batches* est de découper l'ensemble d'entraînement en $\frac{N}{n}$ *batches*, et d'appliquer l'algorithme de descente du gradient sur chacun des ces *batches* (plutôt que sur les N exemples), toujours en itérant plusieurs fois à travers le *dataset*.

4.9 Descente de gradient avec "inertie"

La descente du gradient est un algorithme efficace pour minimiser le coût. Cependant, il a un défaut qui se fait rapidement resentir : il n'est pas "capable" de prendre en compte ce qui a déjà été fait (par exemple si la direction qu'il a prise semble être la bonne).

Pour pallier ce manque, on peut utiliser l'algorithme suivant :

- Répéter ... fois
  - Calculer le résultat pour les $\frac{N}{n}$ exemples du *mini batch* par le réseau ;
  - Déterminer le coût sur les résultats des exemples ;
  - $v_{dW} = \beta v_{dW} + (1 - \beta)dW$;
  - $v_{db} = \beta v_{db} + (1 - \beta)db$;
  - Faire la *backpropagation* : $W = W - \alpha v_{dW}$, $b = b - \alpha v_{db}$

β est un nouvel hyper-paramètre.

4.10 *Root Mean Square Prop*

Il s'agit d'un autre algorithme permettant de limiter les oscillations sur les coordonnées qui ne mènent pas vers un minimum.

- Répéter ... fois
  - Calculer le résultat pour les $\frac{N}{n}$ exemples du *mini batch* par le réseau ;
  - Déterminer le coût sur les résultats des exemples ;
  - $S_{dW} = \beta S_{dW} + (1 - \beta)dW^2$ (pt-à-pt) ;
  - $S_{db} = \beta S_{db} + (1 - \beta)db^2$;
  - Faire la *backpropagation* : $W = W - \alpha v_{dW}$, $b = b - \alpha v_{db}$

4.11 Adam

L'algorithme *Adam* est un algorithme de descente de gradient qui combine les deux techniques sus-citées. Il s'est montré efficace dans énormément de problèmes d'apprentissage et c'est pourquoi il est présent dans tous les *frameworks* de deep learning et très souvent utilisé par défaut.

4.12 Diminution de α

Il est très difficile de trouver un *learning rate* α parfait. En effet, un α trop petit mène à un apprentissage très (trop) long, et un α trop grand à un manque de précision lorsque l'on arrive proche du minimum de la fonction de coût.

Pour allier le meilleur des deux mondes, on peut le faire diminuer au cours du temps (au cours des *epochs*), pour avoir un α grand au début pour se déplacement rapidement lors des premières itérations, et plus petit à la fin, pour se déplacer précisément autour du minimum global.

Pour cela, différentes approches sont possibles (et le choix se fait empiriquement selon le problème) :

$$\alpha_n = \frac{1}{1+\gamma \times n}\alpha_0 \text{ avec } \gamma \text{ un réel}$$

$$\alpha_n = \gamma^n \times \alpha_0 \text{ avec } \gamma < 1$$

$$\alpha_n = \frac{\gamma}{\sqrt{n}} \times \alpha_0$$

...

4.13 Amélioration des hyper-paramètres

Le cours évoque l'optimisation des hyper-paramètres dans deux vidéos. En bref, il conseille d'utiliser la méthode suivante. On note h le vecteur des hyperparamètres (par exemple, $h = (\alpha, \beta_1, \beta_2)$ si on souhaite utiliser *Adam*.

Au lieu d'utiliser une grille régulière pour le choix de nos paramètres, on échantillonne au hasard dans un espace bien choisi (on sait qu'on ne veut pas des valeurs trop grandes de α, que les β_i doivent être dans $[0;1]$).

L'avantage d'utiliser un ensemble aléatoire de configurations est qu'on pourra, en entraînant n fois un modèle, tester environ n valeurs de chaque paramètre, alors qu'on aurait $^3\sqrt{n}$ valeurs de chaque si on avait utilisé une grille régulière.

Une fois qu'on a localisé un sous-espace plus intéressant pour les hyperparamètres, on peut recommencer avec la même méthode dans cet espace restreint, pour obtenir des valeurs encore meilleures des paramètres.

4.14 *Batch-normalization*

L'idée de la *batch-normalization* est de normaliser non pas seulement les entrées, mais tous les neurones (juste après le calcul de Z et avant le calcul de $A(Z)$. Pour cela, on applique l'algorithme suivant :

- $\mu = \frac{1}{n}\sum_{mini-batch} Z^{(i)}$
- $\sigma^2 = \frac{1}{n}\sum_{mini-batch}\left(Z^{(i)-\mu}\right)^2$
- $Z^{(i)}_{norm} = \frac{Z^{(i)}-\mu}{\sqrt{\sigma^2+\epsilon}}$
- $\tilde{Z}^{(i)} = j \times \sqrt{Z^{(i)}_{norm} + \beta}$ où j et β sont apprenables

Les détails de l'implémentation sont cependant laissés aux *frameworks* dans le cours.

4.15 Classification multi-classes et activation *softmax*

Jusque-là, tous les réseaux que nous avons vu n'étaient capables que de donner un résultat binaire : 1 ou 0, et donc de répondre à une question du type "est-ce un chat ?", "le patient est-il malade ?"... Il existe cependant de nombreux cas où l'on souhaite pouvoir classifier parmi plusieurs classes, plutôt que parmi deux (reconnaissance de caractères, reconnaissance de formes...). Dans ce cas, on utilise couramment une activation *softmax*.

Si l'on a p classes, alors on définit la dernière couche du réseau comme étant une couche à p neurones (chaque neurone représentera la probabilité que l'entrée appartienne à chaque classe).

Soit $Z = W^L a^{L-1} + b^L$.

On pose $t = \exp(Z)$ *de même dimension que* Z

Pour chaque classe i on a la proba. $P_i = \frac{t_i}{\sum_j t_j}$

Mais comment représenter les classes ? En effet, on ne peut pas simplement, pour chaque exemple $x^{(i)}$ donner un $y^{(i)}$ entier entre 1 et 4, par exemple. En effet, cela ne convient pas pour l'apprentissage pour des raisons évidentes. On applique donc le traitement suivant.

Si les classes vont de 1 à n, la sortie est de dimension $(1, n)$. Pour chaque exemple, on a $y^{(i)}$ de dimension $(1, n)$ et

$$y_j^{(i)} = 1 \text{ si la classe de} x^{(i)} \text{ est j}$$
$$0 \text{ sinon}$$

On peut alors facilement entraîner le réseau de neurones.

4.16 Introduction à TensorFlow

C'est après avoir tout implémenté et testé "à la main" dans Python que le cours a introduit les API de deep learning. L'utilisation d'un module comme `TensorFlow` permet une abstraction et une généralisation plus simples des réseaux de neurones, mais pas seulement. Comme ces modules sont faits précisément pour l'apprentissage automatique, leurs performances pour l'entraînement sont également meilleures, et représentent un gain non négligeable devant une implémentation naïve avec `numpy`, même vectorisée.

Un exemple simple d'utilisation de `TensorFlow` :

```
import tensorflow as tf
x_train, y_train, x_test, y_test = get_data() # Une fonction perso

X = tf.placeholder(dtype = tf.float32, shape = (n_x, None), name = "X")
Y = tf.placeholder(dtype = tf.float32, shape= (n_y, None), name = "Y")

W1 = tf.get_variable("W1", [25,12288], initializer = \
        tf.contrib.layers.xavier_initializer(seed = 1))
b1 = tf.get_variable("b1", [25,1], initializer = \
    tf.zeros_initializer())
W2 = tf.get_variable("W2", [12,25], initializer = \
    tf.contrib.layers.xavier_initializer(seed = 1))
b2 = tf.get_variable("b2", [12,1], initializer = tf.zeros_initializer())
W3 = tf.get_variable("W3", [6,12], initializer = \
    tf.contrib.layers.xavier_initializer(seed = 1))
b3 = tf.get_variable("b3", [6,1], initializer = tf.zeros_initializer())

Z1 = tf.add(tf.matmul(W1, X), b1)
A1 = tf.nn.relu(Z1)
Z2 = tf.add(tf.matmul(W2, A1), b2)
A2 = tf.nn.relu(Z2)
Z3 = tf.add(tf.matmul(W3, A2), b3)

logits = tf.transpose(Z3)
labels = tf.transpose(Y)

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits = logits,
<     labels = labels
))

optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate).minimize(cost)

init = tf.global_variables_initializer()

tf.reset_default_graph()

with tf.Session() as sess:
    sess.run(init)
    sess.run([optimizer, cost], feed_dict={X: X_train, Y: Y_train})
```

4.17 Réseaux de neurones convolutifs (CNN)

Dans les problèmes de reconnaissance d'images (*Computer Vision*), notamment, un type de réseaux de neurones profond est très souvent utilisé : le réseau convolutif. Son fonctionnement est un peu différent de celui que l'on a vu jusqu'ici.

Je n'aborderai ici que les réseaux convolutifs 2D.

Alors que jusqu'ici, tous les réseaux que l'on avait utilisaient des vecteurs 1D en entrée et en sortie, on peut utiliser des matrices pour d'autres utilisations, les images en sont une évidente. De plus, dans les réseaux convolutifs, au lieu de calculer le résultat d'un neurone par $Z = WX$, on effectue une convolution entre X et un filtre (de taille plus petite que X). Chaque couche dispose de plusieurs filtres, et on a donc, pour un X 2D en entrée, un volume en sortie de chaque couche (à savoir un résultat de convolution "$=$" une image $\times$ n filtres de convolution).

Un réseau convolutif a pour principe de détecter des *features* sur l'image d'entrée, grâce aux filtres. L'énorme avantage qu'ils apportent par rapport à un réseau de neurones "dense" (beaucoup de neurones sur chaque couches reliés à beaucoup de neurones sur la couche d'après) est leur capacité à "compresser" les données (le résultat de la convolution pour plusieurs pixels et un filtre est un pixel, et non pas n^2 pixels) et donc à réduire grandement les besoins en neurones, et donc les temps de calcul.

Quelques soucis arrivent rapidement lorsque l'on utilise un réseau convolutif. Le premier est l'application du filtre au bord de l'image. En effet, impossible d'appliquer un filtre 3x3 centré sur le pixel (0, 0) d'une image (car il n'y a pas de pixel (-1, 0), entres autres). Pour résoudre ces effets, on utilise du *padding*, c'est à dire que sur toutes les zones où le problème apparaît, on rajoute des 0.
Par exemple, sur une image, pour appliquer un filtre 3×3, on devra rajouter un (convolution partielle) ou deux (convolution totale) zéros à gauche, à droite, en haut et en bas (pour qu'on puisse commencer la convolution quand le coin en bas à droite du filtre est sur le coin en haut à gauche de l'image).

Un autre point à aborder à propos des réseaux convolutifs est le *stride* utilisé par les filtres. Lorsque l'on calcule les convolutions, on peut faire avancer le filtre d'un pixel par un pixel, on utilise alors un *stride* de 1. Cependant, on peut également essayer de "marcher" de deux pixels en deux pixels, on a alors un *stride* de 2.

La dimension du volume de sortie de chaque couche est déterminée par la dimension du volume d'entrée, le nombre de filtres, le *stride* et le *padding* utilisés.

Au-delà de ces briques de base, les CNN utilisent également souvent des *pooling layers.* Elles ont pour objectif de réduire la taille de l'entrée afin d'accélérer les calculs. Un exemple simple de *pooling* est le *max pooling.* Si on souhaite réduire une image 4×4 vers une image 2×2, on divise le carré en quatre sous-carrés de taille 2×2 et on prend le maximum des quatre valeurs de chaque carré.

Dans le cas où l'on souhaite réduire une image vers une dimension "moins adaptée" (par exemple 5×5 vers 3×3), on fera une "fenêtre de *max pooling*" qui se déplacera comme un filtre et prendra à chaque fois le maximum sur le sous-carré en cours.

Une autre technique de *pooling* est le *average pooling* dont le principe est de prendre la moyenne du sous-carré.

4.18 Algorithmes de détection et localisation

Dans la pratique, il est souvent utile de pouvoir détecter et localiser précisément des formes sur une image. Plusieurs algorithmes existent dans cette optique.

Une première étape pour cela est de passer d'un algorithme de classification ("qu'est-ce ?") à un algorithme de classification et localisation ("qu'est-ce ET où est-ce ?"). D'une sortie simplement composée d'une étiquette (éventuellement encodée en *one-hot*), on passe à une sortie plus complexe, avec en plus une sortie booléenne ("y a t-il quelque chose ?"), une coordonnée x, une coordonnée y (du centre), et les dimensions de la *bounding box.*

Pour pouvoir localiser un (ou éventuellement plusieurs) objet sur une image, on peut utiliser une "fenêtre glissante". L'idée est de faire passer une "fenêtre de détection" sur chaque zone possible de l'image et de voir si on y reconnaît un objet que l'on aurait appris précédemment (voiture, piéton, *etc.*). On fait la même chose pour plusieurs tailles de fenêtre, afin de pouvoir détecter des objets de toutes tailles (et donc des objets plus ou moins loin).

Cette technique est relativement simple, que ce soit dans la compréhension ou l'implémentation. Son défaut est le coût calculatoire associé. En effet, faire une recherche pour plusieurs tailles de fenêtre est long. Une solution à ce problème est l'algorithme YOLO (*You Only Look Once.*)

L'algorithme YOLO divise d'abord l'image en carrés. Puis, il regarde l'ensemble de l'image en une fois, et au lieu d'appliquer la détection dans chaque carré, il essaie de trouver des caractéristiques de *bounding box*

(relatives, entre 0 et 1 : côté de la *bounding box* / côté de l'image, centre de la *bounding box* par rapport au centre du carré. Tout cela se fait très rapidement, puisque le réseau est quasiment totalement convolutif mais surtout qu'il n'applique la "détection" qu'une seule fois.

C'est pourquoi l'algorithme YOLO est très utilisé. Il permet de détecter beaucoup d'objets et ce avec une complexité relativement faible, ce qui le rend utilisable même dans des applications en temps réel (comme les voitures autonomes).

5 Travaux Pratiques

5.1 Régression linéaire

Le premier exercice est un exercice où on voit la régression linéaire comme un réseau de neurones à un neurone :

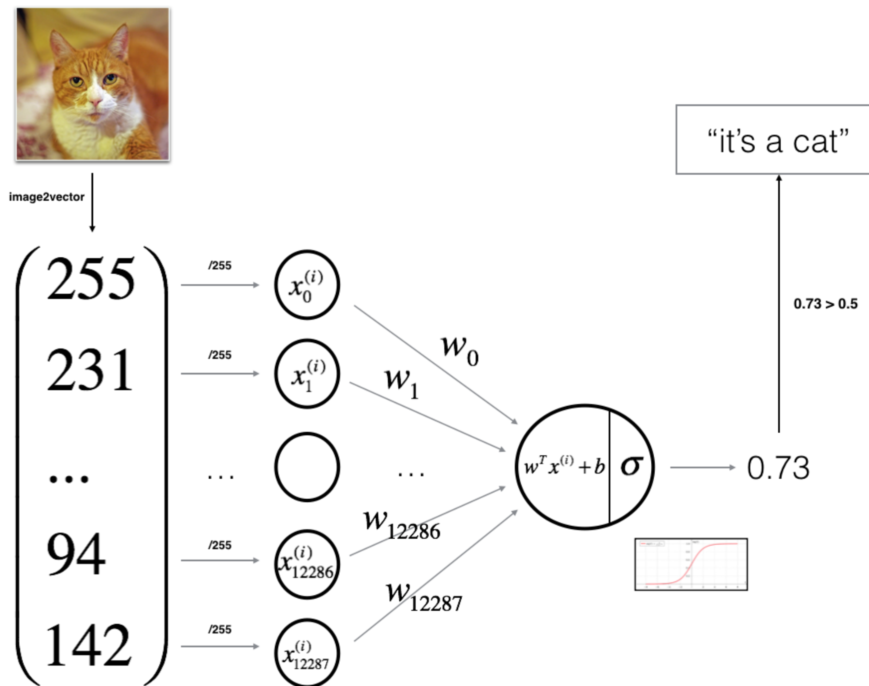


FIGURE 2 – Régression linéaire sur un chat

On voit donc l'image d'entrée (de résolution $W \times H$) comme un vecteur de taille $W \times H \times 3$ et on l'applique à un seul neurone (un vecteur w de même dimension que celui de l'entrée + un biais b, on obtient le résultat $z = w \cdot x + b$ que l'on passe par la fonction d'activation (dans notre cas la fonction sigmoïde), pour obtenir notre "confiance" en sortie.

Bien qu'il soit représenté sous forme de réseau de calcul, ce réseau de neurones est en fait une régression linéaire.

Après entraînement sur 209 images, on arrive à une justesse sur le dataset d'entraînement de 99% et sur le dataset de test de 70%, ce qui est satisfaisant étant donné la simplicité du modèle.

5.2 Classification planaire à une couche cachée

Dans le second exercice, il était demandé d'appliquer la même idée à une classification planaire, en utilisant cette fois une couche cachée à quatre neurones.

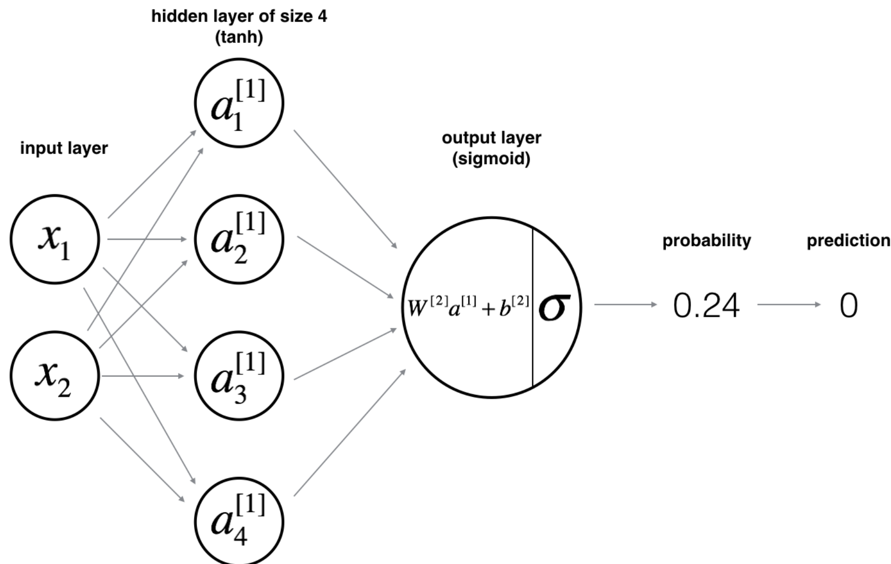


FIGURE 3 – Classification dans le plan

Dans le but de classifier des points :

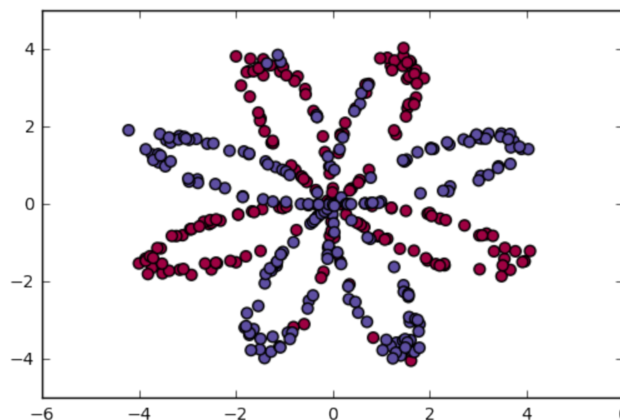


FIGURE 4 – Classification fleur

En effet, une classification par régression linéaire ne donne qu'une justesse de 47%, ce qui n'est pas du tout nécessaire. (C'est en fait logique : une régression linéaire sépare le plan en deux parties, on "voit" sur l'image que ce n'est ici pas suffisant).

Après un apprentissage de 9000 itérations, on arrive à une justesse de 90%, ce qui est bien plus acceptable que la justesse atteinte avec une régression linéaire.

5.3 Implémentation d'un réseau complet

Le dernier exercice de ce premier cours consistait à implémenter en entier un réseau de neurones profond à L couches ($L-1$ ReLU et 1 Sigmoid), qui pourrait apprendre et prédire (propagation et rétropropagation).

Dans un but d'introduction et de version "simple", il fallait d'abord implémenter un réseau à deux couches (donc une cachée, une couche de sortie) puis généraliser à L couches.

Bien que le TP soit très long, il n'avait aucune application directe.

5.4 Utilisation du réseau complet implémenté juste avant

Dans ce TP, on nous donnait un dataset d'images. L'objectif était de classifier les images (images de chat ou non-images de chat), ceci en utilisant les deux réseaux de neurones précédemment implémentés. Le but était de vérifier le fonctionnement des réseaux, mais aussi de comparer leurs performances.

Après avoir entraîné le premier réseau (2 couches), le coût converge vers 5%.

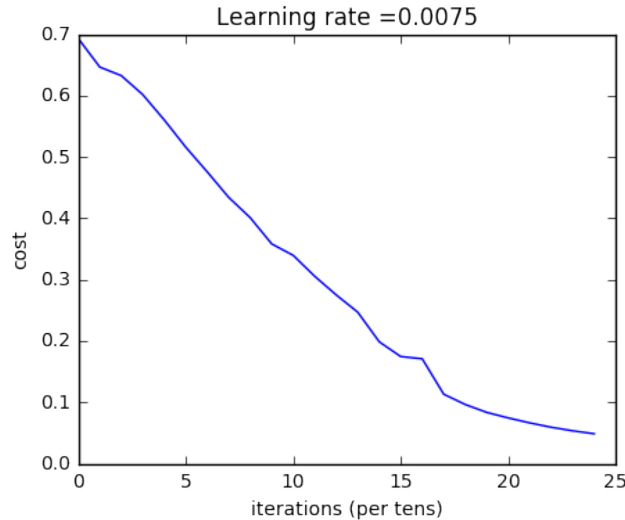


FIGURE 5 – Coût / Epochs

De plus, les performances sont les suivantes : 100% sur le dataset d'entraînement, mais seulement 70% sur le dataset de test. Le réseau à deux couches montre rapidement ses limites.

On utilise alors le réseau à L couches (en pratique 4) : $[12288, 20, 7, 5, 1]$, à savoir 20 neurones, 7 neurones et 5 neurones pour les couches cachées.

Après entraînement, ce nouveau modèle trouve un coût de 1% et les performances en termes de prédiction se révèlent bien meilleures : toujours 100% sur le set d'entraînement, mais cette fois 72% sur le set de test.

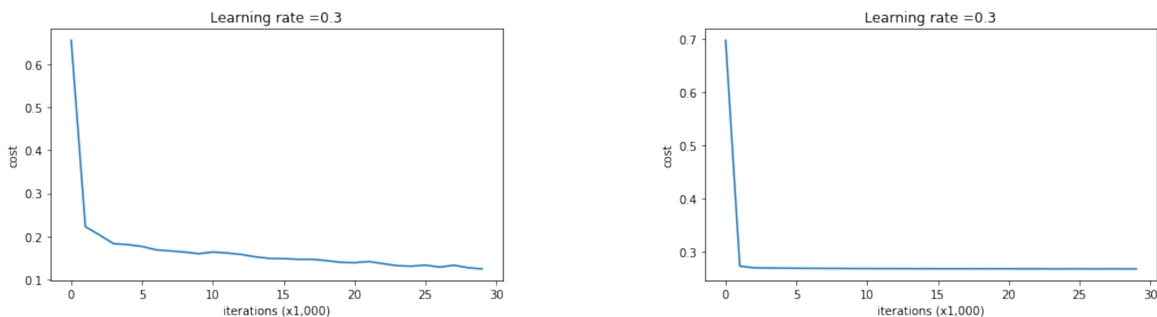
Bien ces résultats puissent sembler décevants (on reste en-dessous de 80-90% que l'on souhaite atteindre dans l'idéal), ils restent une amélioration non négligeable par rapport au modèle précédent, et ce toujours à un coût relativement faible en terme d'apprentissage).

5.5 Régularisation

*Le deuxième cours avait deux autres TP (*Initialization *et* Gradient Checking*), mais ceux-ci étaient inaccessibles aux utilisateurs gratuits.*

Le TP sur la régularisation consistait à utiliser le deep learning pour améliorer les performances d'une équipe de foot.

Le principal intérêt du TP était de montrer comment la régularisation (régularisation L2 dans notre cas) permettait d'améliorer les performances du modèle (ce qui se voyait dès l'étape d'apprentissage).

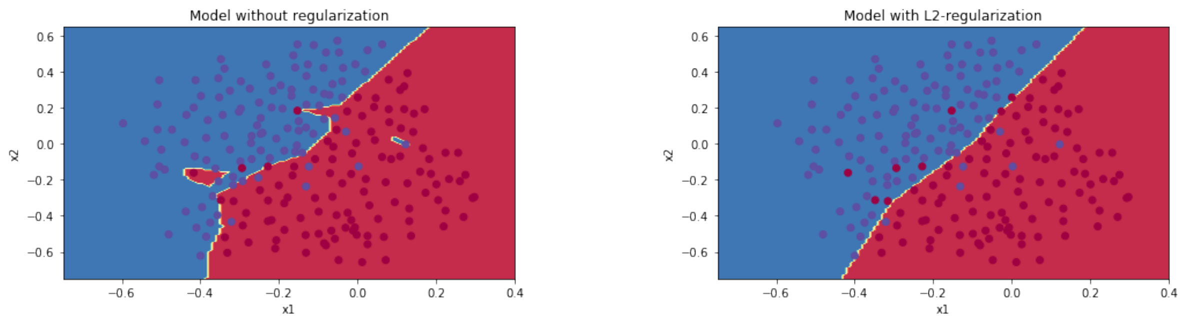


(a) Sans régularisation, l'apprentissage est plus poussif

(b) Avec régularisation, on va jusqu'à environ 0 très vite

FIGURE 6 – Apprentissage sans et avec régularisation L2

Cependant, il ne faut pas oublier que le principal intérêt de la régularisation est de réduire l'*overfitting*, ce qu'elle fait à merveille ici :



(a) Sans régularisation, le modèle overfit

(b) Avec régularisation, les performances sont bien meilleures

FIGURE 7 – Différence dans l'overfitting

5.6 Modèle convolutif pas-à-pas

Le premier TP de ce chapitre consistait à implémenter un modèle convolutif à l'aide de `numpy` uniquement (pas de *framework* de deep learning).

L'objectif était simplement de mettre en application les concepts du cours, en fait, on n'a même pas utilisé le modèle dans un réseau.

5.7 Application du modèle convolutif

Cette fois, on souhaitait utiliser TensorFlow pour classifier des nombres faits avec les doigts de 0 à 5.

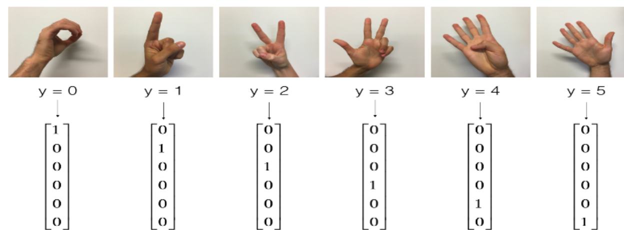


FIGURE 8 – Exemple du dataset SIGNS

Pour cela, on a mis en place un réseau convolutif grâce au *framework* `TensorFlow`. Le modèle est le suivant : CONV2D → RELU → MAXPOOL → CONV2D → RELU → MAXPOOL → FLATTEN → FULLYCONNECTED.

En utilisant les fonctions de `TensorFlow`, on peut automatiquement entraîner le réseau.

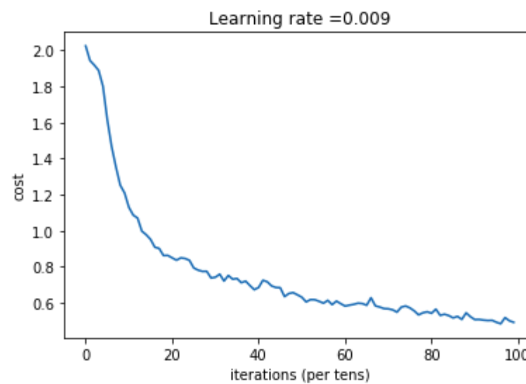


FIGURE 9 – Coût / Epochs

Et on obtient un très bon résultat.

5.8 Tutoriel Keras "The Happy House"

`TensorFlow` est un *framework* très puissant, il manque cependant parfois d'expressivité. Pour pallier cela, il existe plusieurs *frameworks* de plus haut niveau, `Keras` en est un.

L'idée de Keras est de rendre l'implémentation de réseau neuronaux encore plus simple, avec des abstractions encore plus poussées. Par exemple :

```
X_input = Input(input_shape)

# Zero-Padding: pads the border of X_input with zeroes
X = ZeroPadding2D((3, 3))(X_input)

# CONV -> BN -> RELU Block applied to X
#           Nombre de neurones
#           |     Taille des filtres
#           |     |        Strides .......
X = Conv2D(32, (7, 7), strides = (1, 1), name = 'conv0')(X)
X = BatchNormalization(axis = 3, name = 'bn0')(X)
X = Activation('relu')(X)

# MAXPOOL
X = MaxPooling2D((2, 2), name='max_pool')(X)

# FLATTEN X (means convert it to a vector) + FULLYCONNECTED
X = Flatten()(X)
X = Dense(1, activation='sigmoid', name='fc')(X)

model = Model(inputs = X_input, outputs = X, name='HappyModel')
```

Par la suite, il suffit d'entraîner le modèle :

```
model.compile(optimizer = "adam", loss = "binary_crossentropy", metrics = ["accuracy"])
model.fit(x = X_train, y = Y_train, epochs = 30, batch_size = 50)
```

5.9 Car detection with YOLO

Dans ce TP, on se proposait d'implémenter l'algorithme YOLO avec Keras sur un *dataset* destiné aux voitures autonomes.

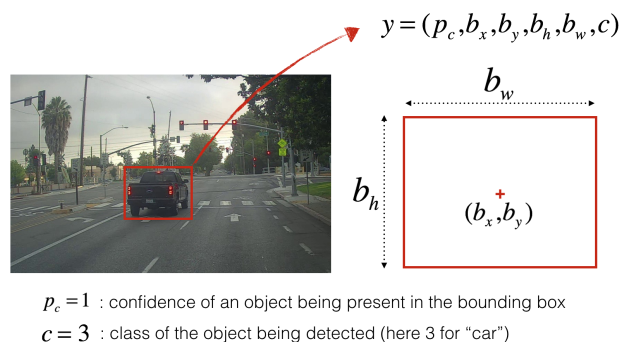


FIGURE 10 – Notre objectif

Le TP est très guidé, et réexplique l'ensemble des étapes pour implémenter YOLO. De plus, tout le travail de préparation des données est déjà fait.

Les étapes :
- Créer une fonction pour éliminer toutes les boîtes de probabilité faible ;
- Implémenter le calcul d'IoU entre deux boîtes
- Implémenter le *non-max suppression* (se fait très rapidement grâce à `Keras` ;
- Créer une fonction de conversion entre "sortie YOLO" et "sortie compréhensible" ;
- Charger les poids d'un modèle préentraîné et le tester.

```
out_scores, out_boxes, out_classes = predict(sess, "test.jpg")
```

```
Found 7 boxes for test.jpg
car 0.60 (925, 285) (1045, 374)
car 0.66 (706, 279) (786, 350)
bus 0.67 (5, 266) (220, 407)
car 0.70 (947, 324) (1280, 705)
car 0.74 (159, 303) (346, 440)
car 0.80 (761, 282) (942, 412)
car 0.89 (367, 300) (745, 648)
```

Figure 11 – Le résultat

6 Projet : Création d'un réseau de reconnaissance de chiffres

6.1 Introduction

Le code source de mon projet (et des informations sur comment l'utiliser) peuvent être trouvés ici :

github.com/KowalskiThomas/MOOCDL

Les fichiers étant très volumineux, je n'ai pas mis sur le Git le dataset préparé ainsi que les paramètres du réseau. Je peux vous les fournir sur clé USB si vous le souhaitez.

Comme indiqué dans les sections précédentes, le MOOC comprenait de nombreux "TPs" qui permettaient de familiariser avec les concepts mathématiques associés aux DNN mais aussi aux outils utilisés en pratique pour les implémenter et les utiliser dans des cas réels.
Malheureusement, ces travaux étaient très souvent assez complexes tout en étant très guidés, ce qui me donnait l'impression de maîtriser le sujet, alors que j'aurais été incapable de faire la même chose sans l'aide apportée par les nombreux commentaires dans les *notebooks*.

Dans le but de consolider mes acquis, et ayant conscience du manque d'expérience que j'avais après le MOOC, j'ai décidé de réaliser comme projet un classifieur de caractères. Mon premier objectif était de classifier des chiffres, et j'espérais pouvoir, à terme, classifier plus de caractères (lettres également, mais aussi certains caractères Unicode).

6.2 Choix des données

Un exercice classique en tant que débutant en deep learning consiste à mettre en place un classifieur efficace pour le dataset MNIST (base de données de caractères écrits à la main). Il est disponible sous format `h5` en téléchargement (et contient donc d'un côté les données prêtes à être utilisées et les labels associés).

Je me suis dit qu'il était dommage d'utiliser une telle base de données. En effet, le MOOC aborde le fait que dans la pratique, une grande partie du temps pour un *data scientist* est consacrée au tri, au nettoyage et à la préparation des données. Utiliser directement cette base aurait été sauter toute cette partie pourtant cruciale dans les cas réels. C'est pourquoi j'ai décidé de faire autrement.

En cherchant sur Google, j'ai trouvé un dépôt sur GitHub qui contient beaucoup de caractères écrits à la main.

github.com/kensanata/numbers

Celui-ci contient des nombres, écrits par différentes personnes, sous formes d'images (PNG) et classés dans des dossiers sous le format suivant : `<Identifiant_Personne>/<Chiffre>/<nom_fichier>.png`. J'ai décidé d'utiliser ces données car il me faudrait avant toute chose trier ces données et les rendre utilisable par mon programme.

6.3 Préparation des données

Dans le but de transformer ces fichiers `PNG` en données utilisables par mon DNN, et avant toute conception de celui-ci, j'ai utilisé la stratégie suviante :

- Obtenir la liste de tous les fichiers du dépôt ;
- Pour chaque fichier :
  - Déterminer son label à partir de son chemin ;
  - Charger ses données brutes grâce à `scipy.ndimage.imread` ;
  - Appliquer des traitements à la matrice (voir plus loin) ;
  - Stocker les données dans une liste Python temporaire.
- Exporter les données brutes (matrices et labels) dans un fichier binaire grâce au module `Pickle`.

Remarque Je n'ai volontairement pas utilisé le module `h5py` car après avoir comparé ses performances et celles de `Pickle`, le second s'est montré beaucoup plus efficace lors de l'écriture de données simples, bien que plus gourmand en espace disque.

6.4 Application de traitements aux données

Les données n'étant pas uniformes (des personnes différentes ont fourni des caractères au dépôt), il me fallait d'abord les filtrer et les normaliser autant que possible. Pour cela, j'ai appliqué différents filtres qui me semblaient nécessaires (ils sont appliqués juste après le chargement des données brutes pour éviter de devoir refaire des calculs à chaque entraînement du DNN).

Pour commencer, toutes les images ne sont pas de mêmes dimensions. Cela se révèle être un problème considérable dès lors que l'on souhaite entraîner le DNN ou évaluer l'image d'une image par le DNN. Pour palier cela, dès le chargement de l'image, je modifie légèrement les données en agrandissant l'image (en pratique, les côtés des images sont toujours entre 95 et 100 pixels, je choisis d'agrandir plutôt que de rétrécir l'image afin de ne pas couper des chiffres qui pourraient être écrits en bord d'image).

De plus, puisque la couleur de l'écriture ne m'importe pas, j'utilise `flatten=True` lors du chargement pour passer l'image en nuances de gris.

```
def load_data(file_name: str):
    im = scipy.ndimage.imread(file_name, flatten=True)
    im = scipy.misc.imresize(im, (96, 96))
    return im
```

Pour continuer, comme la plupart des chiffres ont été écrits au stylo-bille, une information de plus apparaît : la pression exercée sur le stylo lors de l'écriture. Pour supprimer cette information, je sature l'image :

```
def apply_max(arr):
    out = []
    for line in arr:
        line = list(map(lambda x: 0 if x < 230 else 255, line))
        out.append(line)

    out = numpy.array(out)
    out.reshape(arr.shape)
    return out
```

Je considère qu'un pixel est "écrit" s'il est à moins de 230 / 255, ce qui me donne les résultats suivants après application :

(a) Version bruitée

(b) Version saturée

Figure 12 – Avant / Après Saturation

6.5 Présentation du modèle

Après avoir lu différents articles sur Internet, j'ai remarqué qu'il était possible d'obtenir de bonnes précisions pour la classification de nombres manuscrits grâce à un DNN simple (pas un CNN).

Cependant, le MOOC indiquait que l'idéal pour la reconnaissance d'images était le CNN (notamment en matière de performances) et je souhaitais expérimenter plus profondément ce domaine. C'est pourquoi j'ai choisi de réaliser un CNN pour mon projet.

Le problème le plus important auquel j'aie été confronté est ma puissance de calcul limitée. En effet, malgré des tentatives répétées d'obtenir un espace en ligne (sur Google Cloud ou Amazon Web Services) pour entraîner mes modèles, je n'ai jamais réussi à en avoir un. De plus, bien que mon ordinateur dispose d'une carte graphique correcte, je n'ai jamais réussi à faire fonctionner le pilote correctement ce qui m'empêchait d'accéder au potentiel de CUDA. Enfin, le processus de compilation de TensorFlow manquant généralement de clarté sur le site officiel, et n'ayant pas le temps en ce moment de me consacrer à des tests nombreux et longs, je n'ai même pas pu compiler moi-même TensorFlow pour obtenir le gain de performance d'environ 45% apporté par des binaires compilés avec le support pour les instructions AVX. C'est pourquoi je devais au maximum garder des modèles simples, au nombre de couches, de filtres et de noeuds réduits, pour que ceux-ci restent exploitables.

Après plusieurs tests de modèles pris relativement au hasard (du plus simple : Conv2D 32 → Out (trop simple) au plus complexe : Conv2D 128 → Conv2D 64 → Conv2D 16 → Conv2D 8 → Out (dix heures d'entraînement estimées), j'ai finalement trouvé un modèle au potentiel décent et au temps d'entraînement raisonnable :

Conv2D 32 → Conv2D 16 → Conv2D 8 → Conv2D 4 → Out

Après un entraînement (d'environ une heure tout de même), j'ai pu constater des performances très correctes sur les données de test.

J'ai immédiatement souhaité "tester moi-même" mon premier réseau de neurones, et ai donc dessiné quelques chiffres comme je le pouvais avec ma souris :

J'ai comme résultat (pour le 2 par exemple) : `[[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]]` ce qui correspond bien au 2.

De manière générale, j'obtiens des résultats très satisfaisants (le bon chiffre est prédit à chaque fois), même si l'écriture des nombres, le format d'entrée (purement numérique au lieu d'être un scan) et la graphie (évidemment) sont très différents du *dataset* utilisé pour l'entraînement et le test.

Les résultats sont cependant beaucoup moins bon si les caractéristiques du trait sont très différentes du *dataset* d'entraînement. Par exemple, en dessinant avec une pointe très fine, le réseau est simplement perdu :

```
[[0.10184795 0.10486854 0.10112651 0.10192861
0.09898826 0.09993724 0.09978816 0.09855683 0.09461374 0.09834423]]
```

Le réseau est cependant capable de reconnaître le bon chiffre si le trait est un peu plus large.

(a) Difficile à catégoriser

(b) Catégorisé sans problème

7 Conclusion

Je ne regrette pas du tout d'avoir suivi ce MOOC. En tant qu'élève de la filière Génie Logiciel, je trouve dommage qu'aucun cours ne nous présente et ne nous forme au deep learning, dans la mesure où il est de plus en plus présent dans la société et dans le domaine de l'informatique, et ce dans toutes les industries. Cette opportunité de suivre un MOOC m'a permis de découvrir pour la première fois une discipline qui est l'objet de beaucoup de fantasmes dans notre société mais que je ne connaissais tout de même pas du tout.

Je suis sincèrement satisfait du niveau général et du contenu du cours proposé par Coursera. Le niveau est sans doute un peu plus bas que ce que j'aurais pu suivre (beaucoup de chapitres sautent les aspects mathématiques en insistant sur le fait qu'ils ne sont pas primordiaux, car le cours se veut universel) mais cela a un côté rassurant. De plus, les sujets abordés sont variés. Les chapitres partent réellement de la base mathématique et de modèles implémentés à la main en Python et restent dans ce mode de fonctionnement longtemps avant la présentation du premier *framework* étudié (seulement à la fin du second cours). Les éléments d'apprentissage sont bien répartis (les bases, puis les techniques utilisées en pratique, puis des cas pratiques d'utilisation du deep learning) et les travaux pratiques sont de grande qualité (bien que des fois un peu "copier-coller").

Finalement, le seul reproche que j'aurais à faire à ce MOOC est la quantité d'information et la longueur. Je savais en commençant que je n'aurais raisonnablement pas le temps de suivre tous les cours (il y en avait cinq, il y en a maintenant un sixième sur les GAN) mais j'avais en tête d'aller aussi loin que possible. Finalement, je n'ai même pas pu finir le cours sur les CNN du fait du manque de temps.

En conclusion, en tant que total novice de la discipline mais curieux de voir ce qu'était réellement le deep learning, quelles en étaient les fondements et comment il était appliqué dans des vrais cas d'utilisation, je suis totalement satisfait de ce que j'ai pu apprendre ce semestre.