Projet Mind : le Deep Learning

DEROUET Lucien
ESTRADE Benoit
RT1

Sommaire

Partie I : Introduction au Deep Learning

Le machine Learning (ou littéralement apprentissage machine) est un champs d'étude qui qui se fonde sur des approches statistiques pour donner aux ordinateurs la capacité « d'apprendre » à partir de données, c'est-à-dire d'améliorer leurs performances à résoudre des tâches sans être explicitement programmés pour chacune. L'apprentissage automatique comporte généralement deux phases. La première consiste à estimer un modèle à partir de données lors de la phase de conception du système. La seconde phase correspond à la mise en production : le modèle étant déterminé, de nouvelles données peuvent alors être soumises afin d'obtenir le résultat correspondant à la tâche souhaitée. On différencie trois types d'apprentissage les plus couramment étudiés.

- L'apprentissage par **renforcement** : l'algorithme apprend un comportement étant donné une observation. L'action de l'algorithme sur l'environnement produit une valeur de retour qui guide l'algorithme d'apprentissage.
- L'apprentissage **supervisé** : il consiste à apprendre une fonction de prédiction à partir d'exemples annotés.
- L'apprentissage **non-supervisé** : il consiste à trouver des structures sous-jacentes à partir de données non étiquetées

L'apprentissage profond (Deep Learning, noté DL) que nous étudions ici est inclus dans le domaine de l'apprentissage supervisé. Bien que très répandu et populaire actuellement, l'apprentissage profond n'est pas un domaine qui a émergé récemment. En effet depuis 1950, il a connu sous différentes appellations plusieurs vagues d'intérêt.

En 1957, le perceptron (l'apprentissage profond dans sa forme la plus simple) fut théorisé par Franck Rosenblatt. Il sera étudié plus en détail par la suite. La première vague d'intérêt pour l'apprentissage profond est née.

Dans les années 1980, une nouvelle vague portée par un mouvement nommé le Connexionnisme apparaît. Elle est basée sur l'intuition que des comportements complexes, voir même intelligents, peuvent être issus d'éléments très simples mis en réseaux : les neurones artificiels. L'héritage principal de cette vague fut l'algorithme de la rétropropagation (backpropagation) permettant l'entraînement d'un réseaux de neurones. Cependant cette vague connu de nombreuses critiques de la part d'une partie de la communauté scientifique à cause de sa volonté de rapprocher le comportement et le fonctionnement du cerveau humain avec celui d'un réseau de neurones artificiels. De plus, l'entraînement des réseaux pâtissait des faibles puissances de calcul de l'époque, ce qui essoufla l'engouement pour cette technologie, au profit d'algorithmes de machine learning, comme les SVM en 1995.

Nous connaissons actuellement la troisième vague d'intérêt, sous le nom de Deep Learning, lancée en 2006 suite à de grandes avancées dans la recherche. De plus sa grande efficacité vient aussi de l'explosion de la puissance de calcul à disposition, mais aussi de la quantité et variété de données pour l'entraînement.

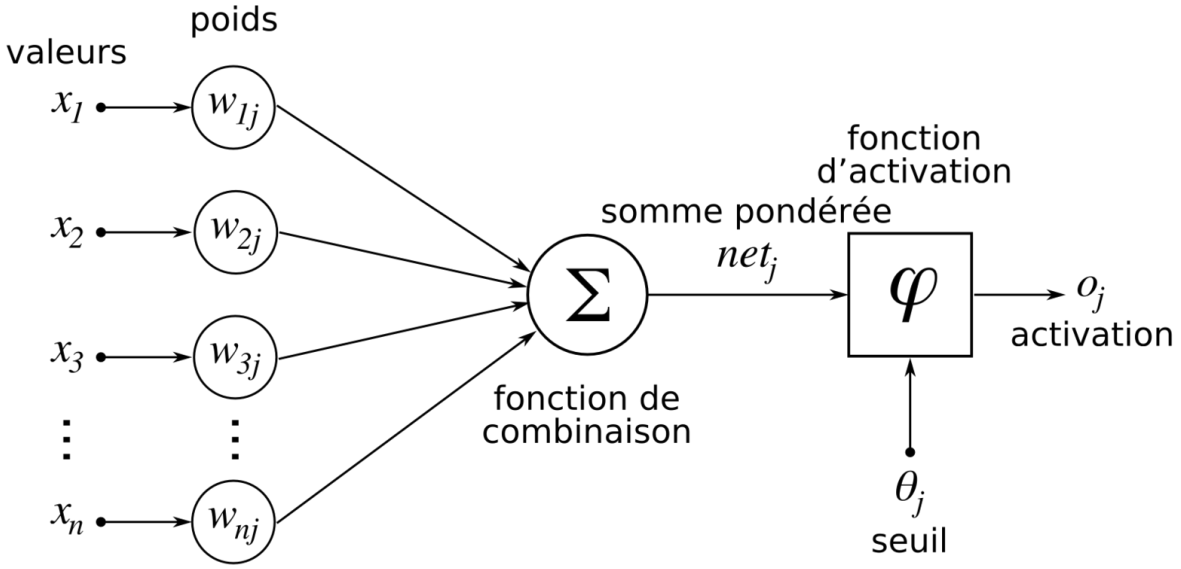
Dans ce projet, nous avons tenté d'appréhender les techniques du Deep Learning et de les implémenter par nos propres moyens.

I. Le neurone artificiel

A. Le neurone artificiel

Le neurone artificiel est le composant fondamental du deep learning. L'unique rapprochement que l'on peut faire avec le neurone biologique est la présence de valeurs d'entrée, reliées au sein du neurone, déterminant ou non une activation. La sortie d'un neurone sera donc prise en entrée d'un autre neurone, tout comme les liens synaptiques, avec les neurones biologiques.

Les réseaux de neurones consistent donc en une association d'un ensemble de neurones, suivant des architectures plus ou moins complexes.

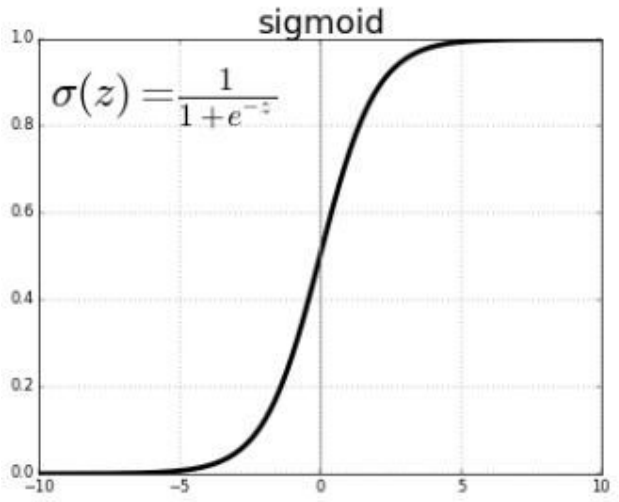


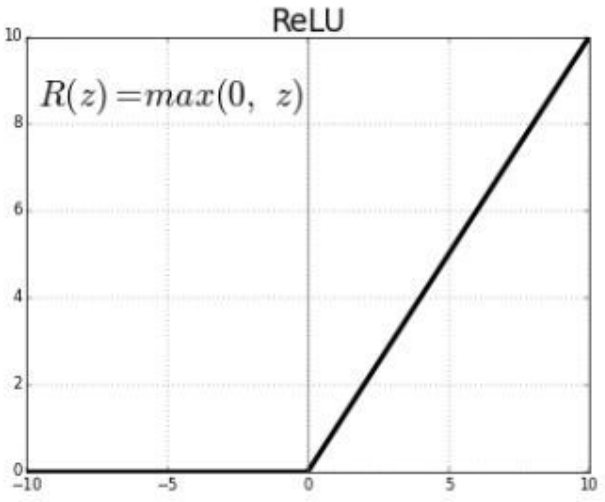
Observons de gauche à droite le schéma ci dessus. Le neurone reçoit en entrée un certain nombre de valeurs, qui sont pondérées par la valeur de leur poids (ou paramètre) correspondant. Le neurone choisit, suivant les valeurs de poids, d'accorder plus ou moins d'importance à certaines valeurs. La fonction de combinaison applique simplement le poids à la valeur correspondante, dans le cas d'un vecteur de valeurs et d'un vecteur de poids, c'est simplement un produit scalaire (ou somme pondérée).

Un seuil, plus souvent appelé biais, ajouté à la somme pondérée permet d'influencer la réponse du neurone quelle que soit son entrée. La fonction d'activation détermine la réponse du neurone, la valeur qu'il va renvoyer : cette valeur est appelée l'activation.

B. Les fonctions d'activation

Il existe de nombreuses fonctions d'activation, possédant des propriétés et usages différents. Actuellement les fonctions d'activation les plus largement utilisées sont sigmoid et ReLU.



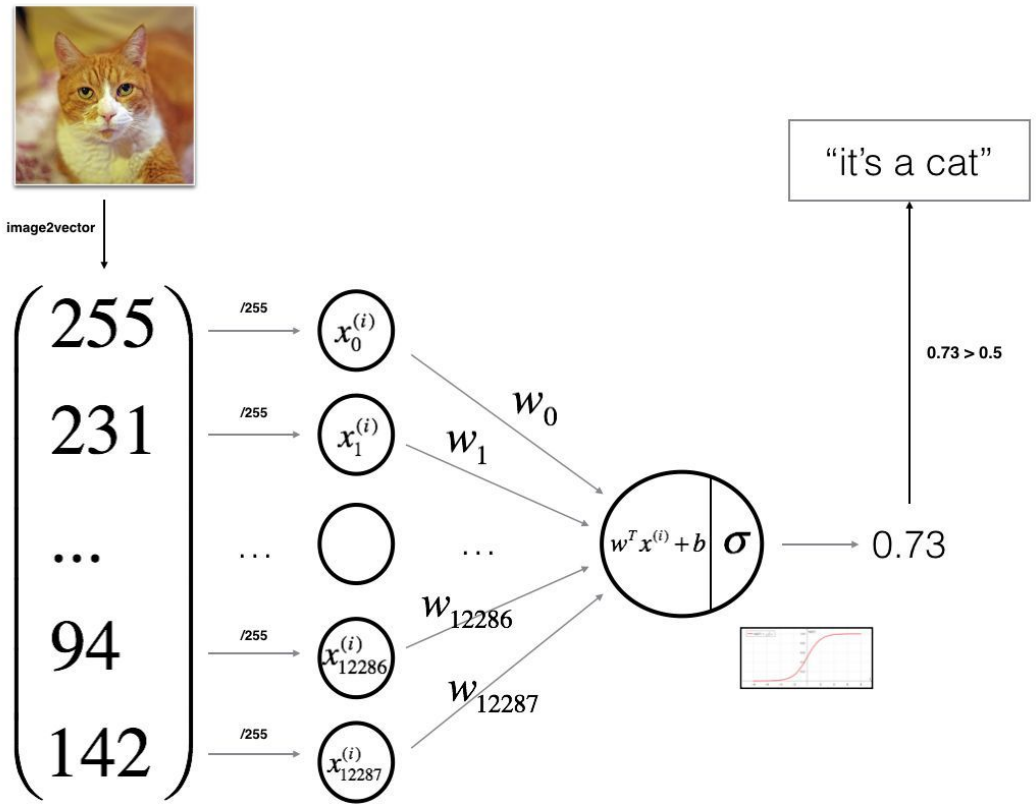


La fonction sigmoid permet d'obtenir en sortie du neurone une valeur entre 0 et 1, pouvant s'apparenter à une probabilité ou à une certitude. Cette propriété est utile pour les classifications, car elle indique si le modèle est certain de sa prédiction, à tort ou à raison, en fonction de la qualité de l'entraînement. Cette fonction est donc employée par les neurones en sortie du réseau.

La fonction ReLU (Rectified Linear Unit) est utilisée par les neurones des couches intermédiaires (dites couches cachées), elle permet aux neurones d'avoir un comportement non linéaire, facilitant l'apprentissage de fonctions très complexes, pour lesquelles la sortie n'a pas de relation linéaire avec l'entrée.

II. Le perceptron

Le perceptron est le modèle le plus simple, composé d'un unique neurone. Il peut être employé entre autres réaliser des classifications d'images



Pour faire passer une image en entrée, il suffit d'en "dérouler" les trois matrices RGB qui la représentent. La classification est binaire, le perceptron ne peut que différencier deux classes, sur le schéma, il s'agit de différencier un chat, d'une image qui ne représente pas un chat. On considère que le perceptron identifie un chat si sa réponse est supérieure à 0.5, sinon ce n'en est pas un.

Pour que le perceptron puisse réaliser cette tâche, il faut l'entraîner afin de définir une valeur pour chacun des poids. Il faut alors introduire une notion d'erreur, que l'on va quantifier, puis chercher à diminuer. Cette erreur est définie par une fonction d'erreur (Cost Function). Il en existe plusieurs. Le choix dépend de la tâche à réaliser. Celle que nous utiliserons est une des plus employées pour les classifications : la cross entropy cost function.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right]$$

La cross entropy cost function

- m : est le nombre de “training example”, le nombre d’images dont on dispose pour l’entraînement.
- y (i) : est le label de l’image d’entraînement n° i : un 1 si c’est un chat, 0 sinon.
- h_θ (x(i)) : est l’activation calculée pour la i-ème image.

Cette fonction combine deux éléments : un terme disparaît suivant la valeur de y. Si y vaut 1, le facteur (1 - y (i)) annule le terme de droite. Il ne reste que -log(h (x i)), qui aura une valeur nulle si l’activation est 1. En effet si l’activation prédit un chat, à 100% de confiance, il n’y a donc une erreur nulle. A l’inverse, l’erreur augmentera à mesure que l’activation se rapproche de 0.

La fonction est analogue pour y=0, l’erreur sera nulle pour une activation égale à 0 et infinie pour une activation à 1.

Maintenant que l’on sait quantifier l’erreur, il faut la diminuer, nous cherchons donc un minimum de la fonction coût. Le minimum global est souvent très difficile à atteindre, un minimum local peut toutefois apporter des performances très satisfaisantes. Nous avons donc un problème d’optimisation, résolu par la méthode de la descente de gradient.

En effet, en partant d’un point défini aléatoirement, le gradient nous donne la direction des valeurs croissantes de la fonction, pour une variation infinitésimale des paramètres. L’opposé du gradient donne donc toujours la direction d’un minimum.

La descente de gradient est un algorithme itératif, qui consiste à mettre à jour les paramètres avec la valeur de gradient calculée à chaque itération

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) \quad \theta_j := \theta_j - \alpha\frac{\partial}{\partial \theta_j}J(\theta).$$

Gradient de la cost function

Mise à jour des paramètres

La largeur de chaque pas est pondérée par un coefficient alpha, appelé la “learning rate”. C’est un hyperparamètre à définir précisément, car sa valeur joue sur la qualité de l’apprentissage. Une valeur trop élevée empêchera la convergence, car le pas trop important ‘sautera’ le minimum à chaque itération, dans certains cas il peut même y avoir une divergence.

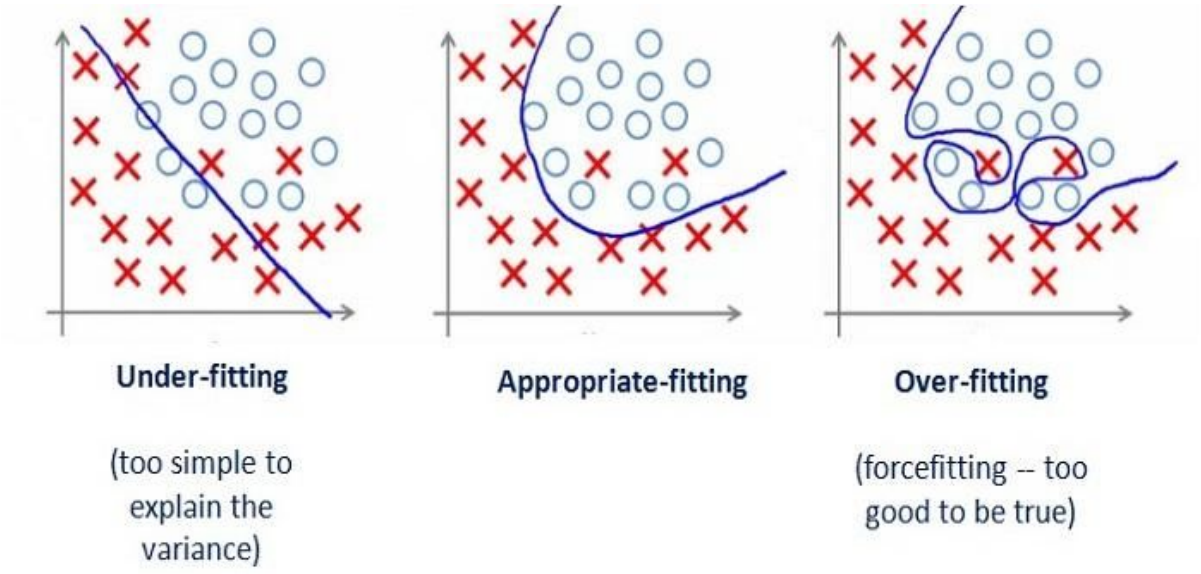
A l'opposé, une valeur trop faible provoquera une descente très lente. L'entraînement du modèle se fera donc sur un ensemble de données d'entraînement, en mettant à jour les poids suivant la descente de gradient, durant un nombre d'itération défini à l'avance.

La performance du système se détermine grâce à un autre ensemble de données non labellisées différentes des données d'entraînement : l'ensemble de test. Il s'agit de mesurer la performance du modèle sur des données qui lui sont inconnues. Il est intéressant de comparer la performance de classification sur les d'entraînement, et sur les données de test.

III. Améliorer l'apprentissage

De nombreux obstacles sont rencontrés lors de l'entraînement d'un modèle.

A. Les problèmes de sur-apprentissage et de sous-apprentissage



Le sous-apprentissage (underfitting) arrive lorsque l'erreur est très élevée sur les données d'entraînement comme sur les données de test. Cela signifie que le modèle n'ar rien appris, ou très peu.

Le sur-apprentissage à l'inverse se caractérise par une erreur très faible sur les données d'entraînement, mais une très élevée sur les données de test. Le modèle a sur-interprété les données d'entraînement. Il a appris, mais de façon tellement fine qu'il ne parvient pas à généraliser son apprentissage à d'autres données. Sur l'image ci-dessus, on voit que la frontière évite les deux 'X', alors qu'on peut considérer que c'est une erreur acceptable, compte tenu de l'ensemble des données.

B. Limiter l'over/under fitting

Pour palier à ces problèmes, diverses méthodes existent.

- La régularisation

La régularisation permet d'éviter l'overfitting. Une des méthodes les plus utilisée est la régularisation L2. Elle consiste à ajouter la somme des paramètres au carré (norme L2 au carré) pondérée par un coefficient lambda, à la fonction coût.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Cross entropy cost function régularisée

Cela a pour effet de répartir l'apprentissage sur tous les paramètres, et empêcher le modèle de donner trop d'importance (donc d'augmenter trop leur valeur) à certains. Des valeurs trop importantes sont donc pénalisantes, car elles augmentent l'erreur commise.

- Hyper parameter tunning

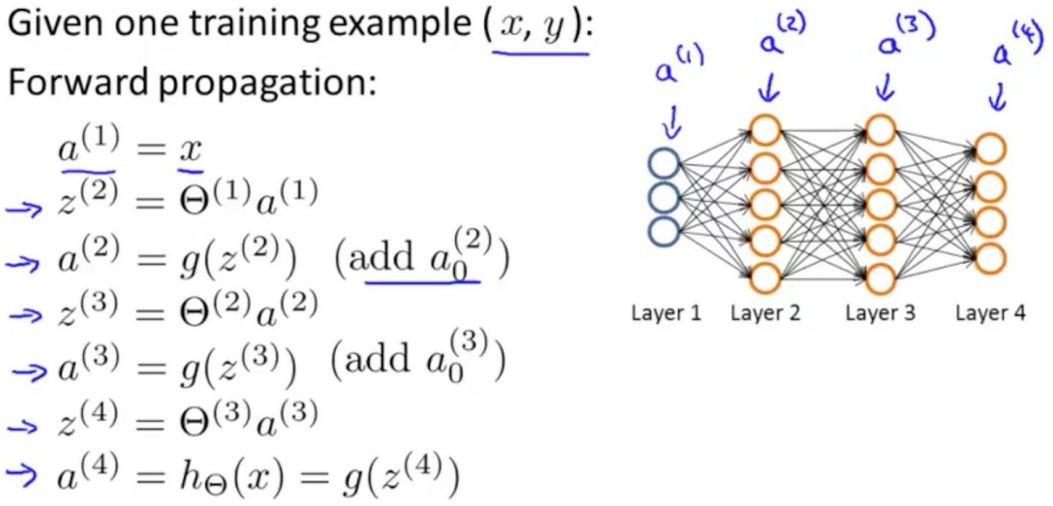
Ce procédé consiste à définir précisément les meilleurs valeurs possibles pour tous les hyper paramètres, comme la learning rate évoquée précédemment. De plus, la quantité de donnée d'entraînement est à définir de façon avisée, car si elle peut avoir un effet de régularisation dans le cas d'un overfitting, elle peut être néfaste dans le cas de l'overfitting.

L'étude de la courbe d'apprentissage, l'évolution de l'erreur en fonction du nombres d'itération effectuées est un bon support d'analyse pour contrer ces phénomènes.

IV. Les réseaux de neurones

A. Construction

Maintenant que le perceptron est clairement défini, il est facile de passer aux réseaux de neurones. Il s'agit simplement d'associer des neurones, suivant une architecture en couches. Nous ne traiterons que des réseaux "fully connected", où tous les neurones d'une couche sont connectés à tous ceux de la couche suivante.



Propagation avant dans un réseau de neurones

Les réseaux de neurones sont dits 'profonds' quand leur nombre de couches augmente. Cela permet d'apprendre des fonctions très complexes. Cette structure permet au modèle de définir lui même ses propres 'features', propriété des données auxquelles il va s'intéresser. En effet partir des données d'entrée, le modèle calcule l'activation de la première couche (qui sera un vecteur, contenant l'activation de tous les neurones) et celle-ci sera prise en entrée de la couche suivante, et ainsi de suite.

Il est aussi possible d'avoir plusieurs neurones sur la couche de sortie, ce qui permet d'avoir un vecteur en sortie du réseaux, et donc de pouvoir faire des classification sur plus de classes entre autres. Le label de chaque classe est donc un vecteur.

Le calcul de l'activation de toutes les couches successivement est la propagation avant (forward propagation).

A la différence du perceptron, il n'y a plus de vecteur de paramètres, mais une matrice par couche, indiquant les poids que chaque neurone associe à l'activation de la couche précédente.

B. L'apprentissage

La fonction de coût est exactement la même, à la différence que nous sommons l'erreur commise sur tous les neurones de la couche de sortie.

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left[-y_k^{(i)} \log((h_\theta(x^{(i)}))_k) - (1 - y_k^{(i)}) \log(1 - (h_\theta(x^{(i)}))_k) \right]$$

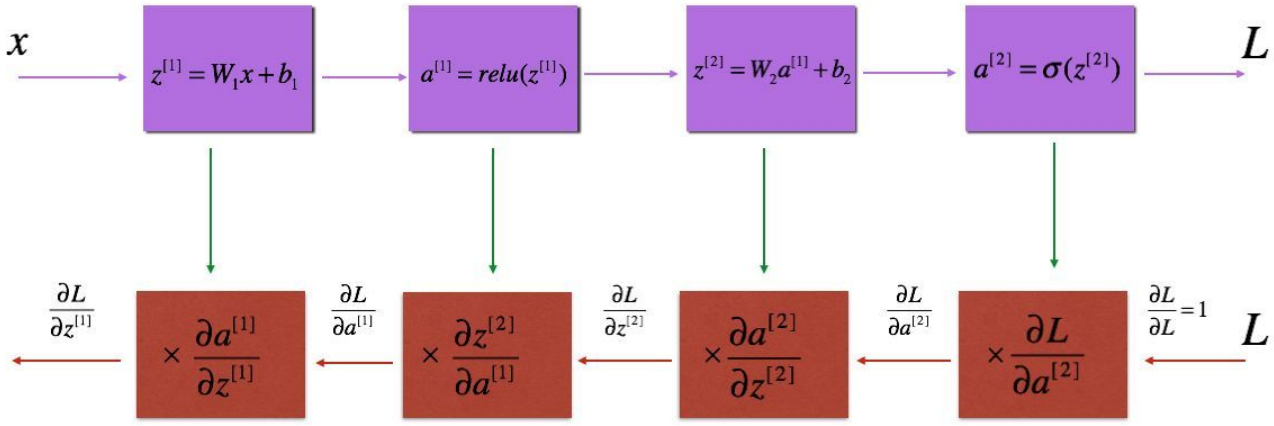
- K : le nombre d'unité en sortie
- m : nombre d'images d'entraînement

L'optimisation de la fonction coût se fait aussi en descente de gradient, à la différence que le calcul du gradient est rendu plus compliqué à cause de la structure en couche.

L'algorithme de la rétropropagation (back propagation) est utilisé pour calculer le gradient à chaque itération. Il basé sur la règle des dérivées chaînées :

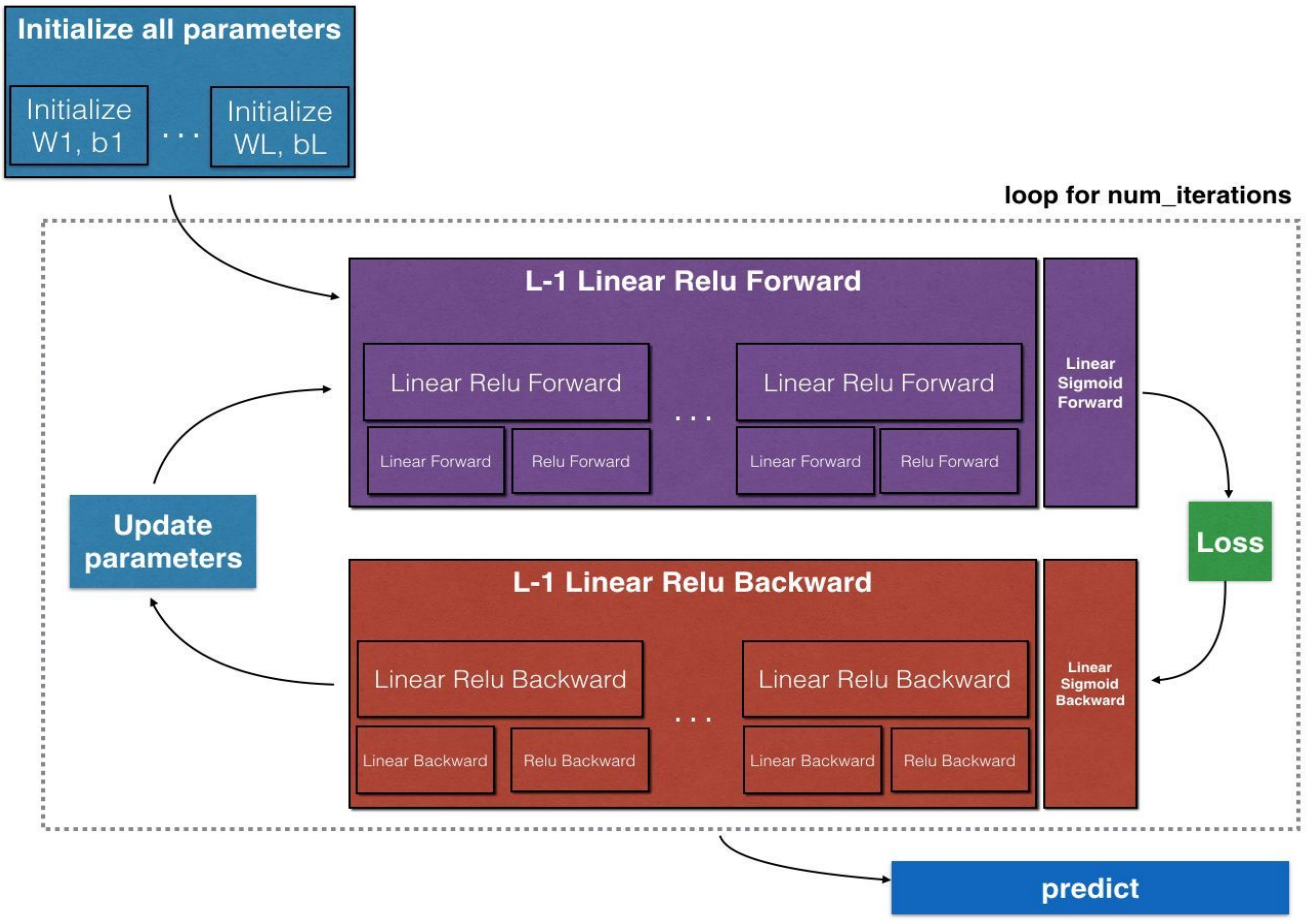
$$\frac{\partial h}{\partial x_i} = \sum_{k=1}^{n} \frac{\partial f}{\partial g_k} \frac{\partial g_k}{\partial x_i}$$

La rétropropagation consiste donc à calculer le gradient de la couche de sortie, puis chaîner les dérivées partielles jusqu'à parvenir au gradient suivant les paramètres de la première couche.



Shéma de la remontée du réseau, par les dérivées partielles

C. Synthèse sur l'entraînement d'un réseau de neurones



L'entraînement commence par l'initialisation aléatoire des poids, puis suivant un nombre fini d'itération, la propagation avant est réalisée pour parvenir à l'erreur commise, le gradient est calculé de la couche de sortie vers la couche d'entrée par rétropropagation, puis les paramètres sont mis à jour.

V. Amélioration des réseaux

Les réseaux de neurones pâtissent des même difficultés d'entraînement que le perceptron, l'entraînement étant même plus compliqué.

A. La régularisation

- L2

La régularisation L2 s'applique aussi aux réseaux de neurones, à la différence que l'on somme le carré de tous les poids de toutes les matrices.

- Dropout

Le dropOut est une autre méthode très utilisée, consistant à retirer de façon probabiliste certains neurones d'une couche durant la propagation avant. On 'retire' un neurone en mettant à zéro son activation.

Le but de cette méthode est encore de répartir l'apprentissage sur l'ensemble des poids du réseaux, et non de se focaliser une portion réduite d'unités. Si c'est le cas, il se retrouve pénalisé par une importante erreur si un des neurones auquel des poids importants sont associés est retiré.

Cette méthode apporte cependant du bruit dans l'apprentissage, à cause de son caractère aléatoire, mais est appréciée par sa performance, notamment en vision par ordinateur, domaine dans lequel les couches comportent beaucoup d'unités, à cause de la grande dimension de l'entrée.

B. Choix des hyper paramètres

Ce choix est encore plus compliqué avec les réseaux de neurones, car il y a beaucoup d'hyper paramètres.

Il faut définir l'architecture du réseau : nombre de couches cachées, leur nombre d'unités. Dans le cas d'un underfitting ou overfitting, il peut être intéressant de revoir l'architecture du modèle. Un modèle trop profond, avec beaucoup d'unités aura tendance à plus facilement sur-interpréter et est plus coûteux en calcul, alors qu'un réseau trop petit peut avoir des difficultés d'entraînement.

Les paramètres de régularisation, lambda pour la régularisation L2 et la probabilité de retirer un neurone dans le cas du dropout sont à définir précisément. Une régularisation trop forte peut empêcher mener à l'underfitting, alors qu'une régularisation trop faible n'aura pas d'effet.

C. Accélération de la convergence

L'apprentissage d'un réseaux de neurones est long car très coûteux en terme de calculs. Plusieurs méthodes permettent d'accélérer la convergence du gradient.

- Normalisation

La normalisation permet d'accélérer les calculs, car elle permet la manipulation de données plus simples : moyenne à zéro, et de taille réduite. Pour cela, il faut centrer-réduire les données. Il faut soustraire la moyenne des données, et diviser par l'écart type.

- Mini batch :

Jusqu'ici, nous n'avions uniquement considéré la propagation avant et la rétropropagation en batch, c'est à dire que toutes les données d'entraînement passent dans le réseaux à chaque itération. Cela permet une descente très directe vers le minimum, mais elle est très coûteuse en temps de calcul. De plus, sur des ensembles de données dépassant la quantité de RAM disponible, les calculs se trouvent très ralentis.

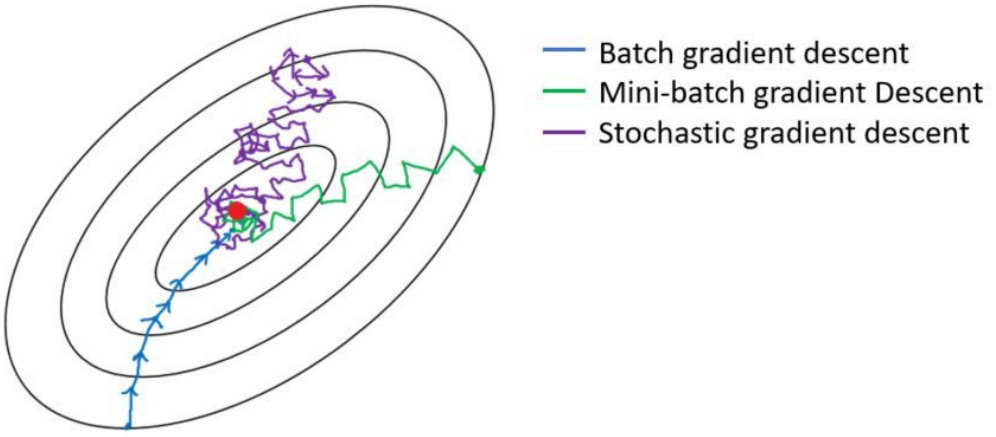
Une solution est d'employer la descente en "mini-batch". Un mini batch est un sous-ensemble de l'ensemble d'entraînement, de quelques dizaines d'éléments généralement. La descente consiste alors à faire un cycle propagation avant/arrière mini-batch après mininbacth. Une fois que l'ensemble de données a entièrement été parcouru, l'itération est terminée, on parle plutôt de 'epoch' dans ce cas. Le nombre 'd'epoch' est défini par l'utilisateur.

L'intérêt de cette descente est qu'elle est bien plus rapide que la descente en batch, elle permet de faire un pas après chaque mini-batch au lieu d'attendre de faire les calculs sur l'ensemble complet des données. Une itération permet donc de réaliser autant de pas que de mini-batch contenu par l'ensemble de données.

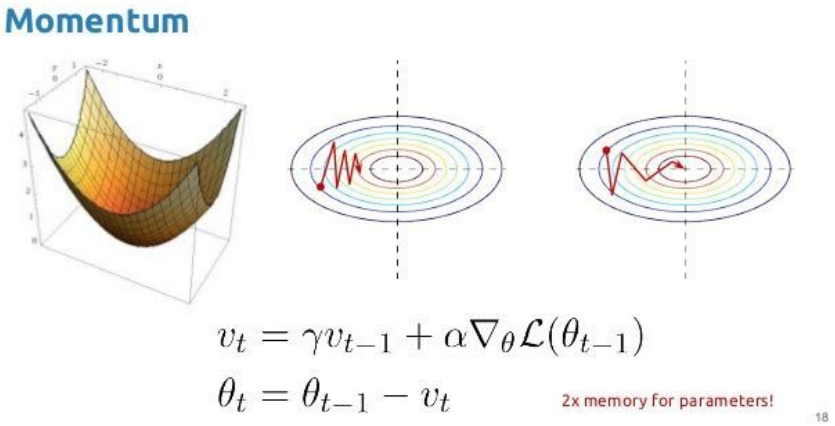
Le deuxième avantage de cette méthode est qu'elle apporte du bruit dans la descente. En effet, le gradient calculé à partir d'un unique mini-batch ne contient pas l'information de l'entièreté des données, rendant la descente moins directe. Ce bruit est très utile, car il permet de 'sauter' un minimum local et poursuivre la descente, alors qu'avec la descente en batch, l'algorithme se serait arrêté dans ce minimum local.

- La descente stochastique (SGD)

La descente stochastique est très semblable à la descente en mini-batch, à la différence que la taille d'un mini-batch est réduite à un. Cette méthode apporte encore plus de bruit, et accélère la descente. Son désavantage est qu'elle ne peut s'arrêter après convergence, l'algorithme va osciller autour du minimum.



- Momentum



Le momentum est une méthode permettant de rendre la descente en mini-batch ou descente stochastique plus rapide, en limitant les écarts de direction de descente et d'amplitude.

Cela consiste à faire la moyenne mobile exponentielle des valeurs de gradients avant de mettre à jour les paramètres. De cette façon, l'information de la direction et l'amplitude des pas précédents est conservée, et est complétée par la valeur du gradient actuel. Dans la formule, la paramètre gamma permet de doser l'importance donné aux valeurs moyennes précédentes, le nouveau pas est donc moins soumis aux fluctuations causées par le mini-batch courant.

VI. Implémentations réalisées

Nous avons réalisé plusieurs implémentations des algorithmes présentés précédemment.

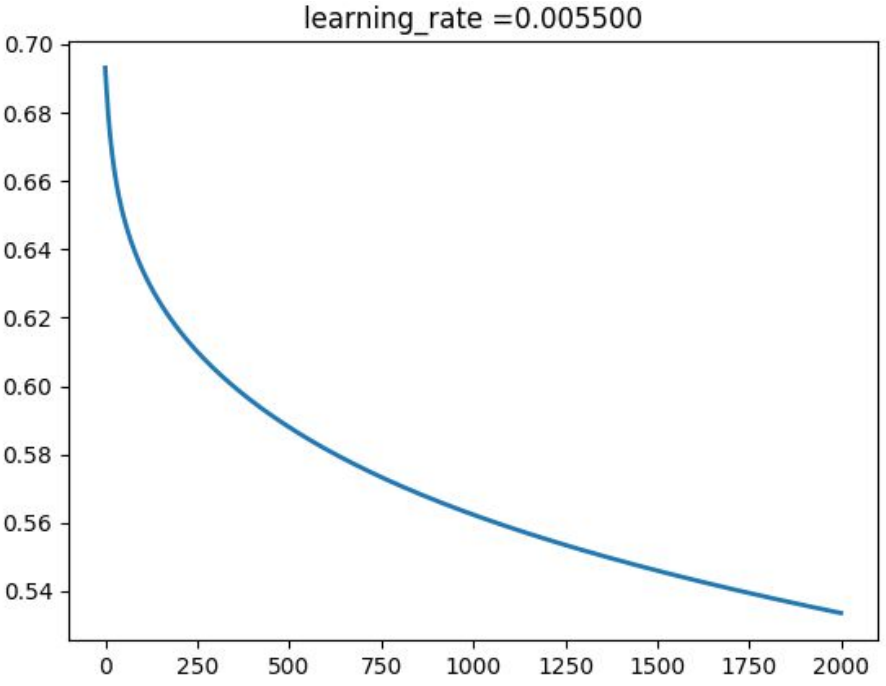
Le choix de ne pas utiliser de framework orienté deep learning, mais uniquement les librairies python numpy pour la manipulation des matrices, et matplotlib pour tracer les graphes a été fait pour avoir une meilleure compréhension des algorithmes. L'idée était de concevoir intégralement les modèles, d'implémenter sois-même les algorithmes et les formules, pour en avoir une compréhension plus profonde.

Les modèles sont entrainés à partir du dataset CIFAR-10, rassemblant 50 000 images RGB de 32x32, appartenant à 10 classes différentes.

Le choix de ne pas avoir utilisé de framework fut sans surprise en dépit des performances. Nous n'avons pas non plus à disposition des machines très performantes, donc l'entraînement des réseaux est loin d'être optimal. Les modèles ont étés entraînés sur quelques milliers d'images uniquement, pendant quelques minutes seulement.

Le code est joint au rapport, il peut normalement être lancé facilement si les librairies sont installées, ainsi que le dataset. Il suffira alors de changer la valeur de la variable "dataset_path" par le chemin du dataset.

- Le premier modèle implémenté est le perceptron, sans régularisation, avec uniquement deux classes.

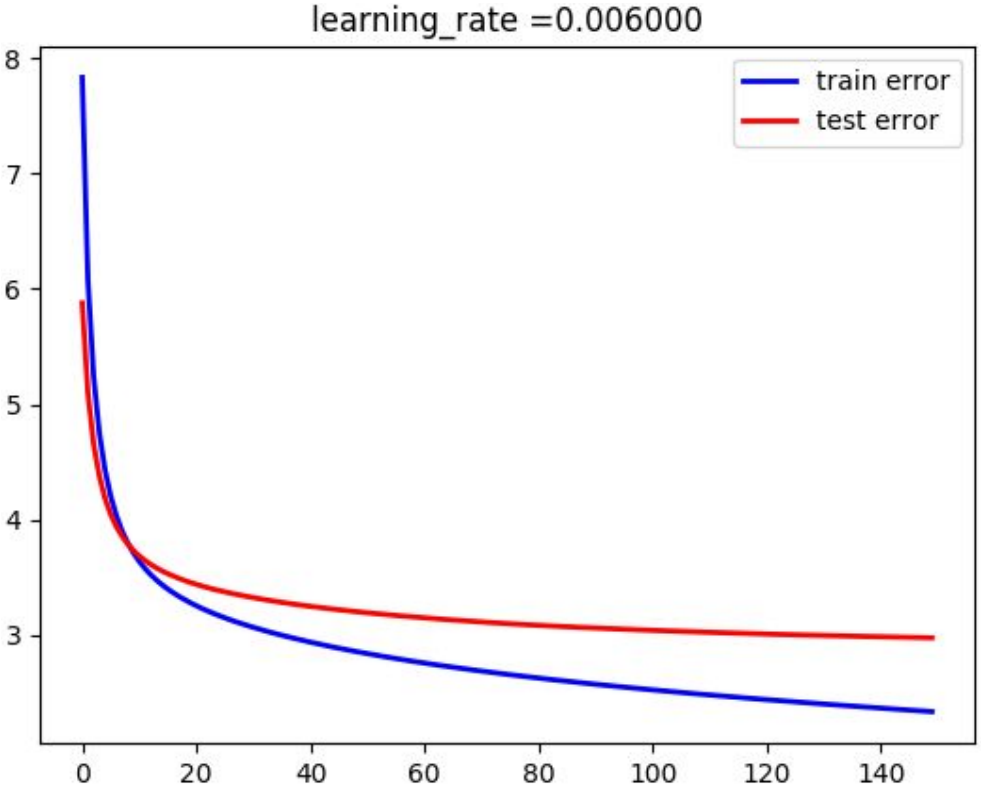


Variation de l'erreur commise sur le training set, en fonction du nombre d'itérations

```
iteration 1700: cost=0.540657
iteration 1800: cost=0.538183
iteration 1900: cost=0.535811
alpha=0.005500, trainSet:73.600000 testSet:67.800000
```

Après quelques minutes d'entraînement, le modèle réalise 73% de bonnes classifications sur le training set, et 67% sur le test set. Le modèle n'est pas très performant, sachant qu'une classification au hasard (guesstimate) aurait donné 50% de réussite.

- Le deuxième modèle est un réseau de neurones simple, avec une seule couche cachée, une descente de gradient en batch et régularisation L2. Nous pouvons voir sur le graphe de la cost function que l'écart entre l'erreur sur les données de test et d'entraînement grandit sur les dernières itérations, le modèle est donc en sur-apprentissage.

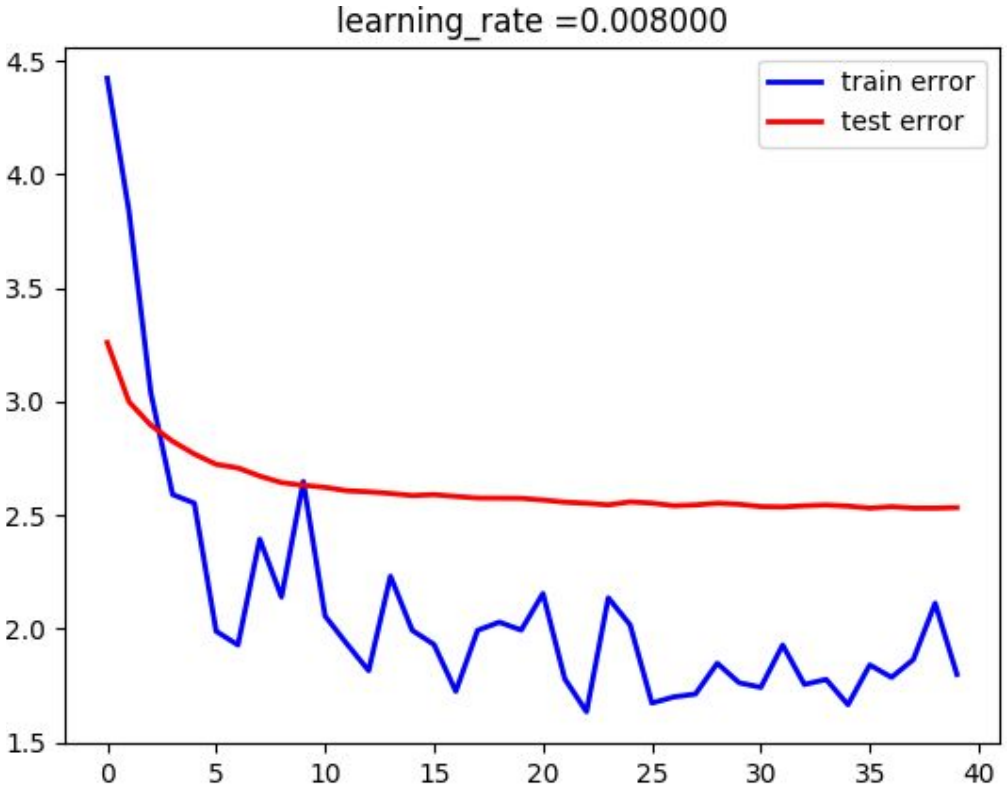



```
lucien@lucien-Latitude-E6420:~/Documents/RT-1/PM_DeepLearning/logisticReg
(ง'̀-'́) ./L_DeepNeuralNet_1.py
Deep neural network 1

Batch gradient descent, gradient checking, L2 Regularisation
Clipping input data to the valid range for imshow with RGB data ([0..1] for fl
or [0..255] for integers).
cost iter 2.348811: 100%|                    | 150/150 [02:29<00:00,  1.11
alpha=0.006000, trainSet:0.511500 testSet:0.296667
```

Nous pouvons constater l'écart entre le pourcentage de classification correcte sur les données d'entraînement de 51%, contre seulement 30% sur les données des test.

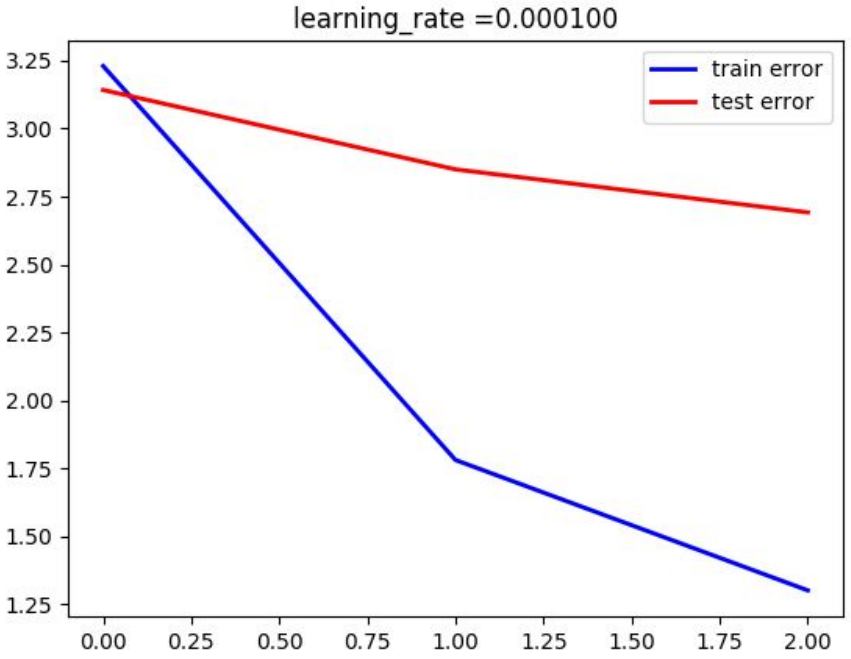
- Le troisième modèle est un réseaux de neurones reprenant l'architecture du précédent, avec une descente en mini-batch, momentum et régularisation par dropout.




```
lucien@lucien-Latitude-E6420:~/Documents/RT-1/PM_DeepLearning/logisticReg
(ಠ_•) ./L_DeepNeuralNet_2.py
Deep neural network
Mini-batch grad descent with momentum and drop out regularization
cost iter 1.798964: 100%|                                        | 40/40 [03:22<00:00,  5.20s/it]
(40, 1)
alpha=0.008000, trainSet:0.503200 testSet:0.418000
```

Nous pouvons constater que la courbe d'erreur sur l'entraînement est très bruitée, ce qui est dû aux mini-batchs et au dropout. Cependant, le modèle est plus performant que le précédent, avec 41% de bonne classification sur les données de test. L'écart de performance entre le trainSet et le testSet est d'ailleurs moins important.

- Ce dernier modèle reprend le précédent mais emploie le “adam optimizer”. Les performances sur le testSet sont égales au modèle précédent alors qu’on peut remarquer un sur-apprentissage. Il est donc possible d’améliorer grandement les performances. De plus, ce modèle n’a été entraîné qu’en 1 minute, il semble donc bien plus rapide que le précédent, qui a mis plus de 3 min, sachant qu’il comporte 1000 unités sur la couche cachée contre 300 dans le modèle précédent.



La faible qualité de la courbe vient du fait qu’un point est gardé en mémoire à chaque itération, et il n’y en eut que 3 pour l'entraînement du modèle.

```
lucien@lucien-Latitude-E6420:~/Documents/RT-1/PM_DeepLearning/logisticReg
(ಠ_•́) ./L_DeepNeuralNet_3.py
Deep neural network
Mini-batch grad descent with adam optimizer and drop out regularization
cost iter 1.301761: 100%|                                  | 3/3 [00:58<00:00, 19.49s/it]
(3, 1)
alpha=0.000100, trainSet:0.616875 testSet:0.416000
```

Performances : 61% de précision sur le training set contre 41% sur les données de test.

Partie II : Les Réseaux de Neurones Convolutifs

I. La convolution

A. Convolution discrète

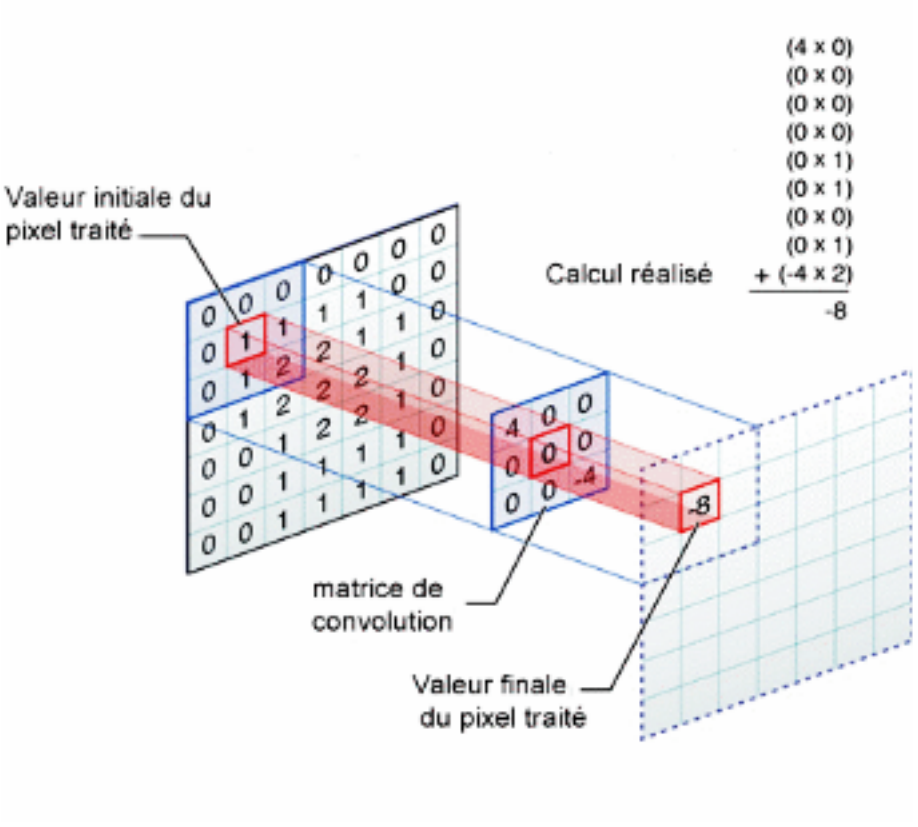
On retrouve ici la forme discrète du produit de convolution applicable aux images :

$$(f * g)(n) = \sum_{m=-\infty}^{\infty} f(n-m)g(m) = \sum_{m=-\infty}^{\infty} f(m)g(n-m)$$

La multiplication dans le domaine fréquentiel de l'image correspond à la convolution dans le domaine spatial. Un filtre (ou noyau de convolution) est une matrice de taille impaire, très souvent carrée. Ici, on utilise un filtre carré de taille d . Le produit de convolution d'une image monochromatique $f(i,j)$ avec un filtre $h(i,j)$ est donné par :

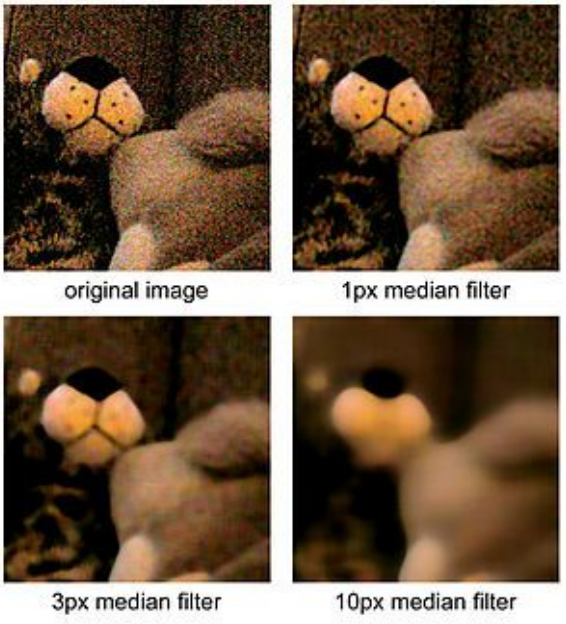
$$F(i,j) = (f * h)(i,j) = \sum_{n=-\frac{d-1}{2}}^{\frac{d-1}{2}} \sum_{m=-\frac{d-1}{2}}^{\frac{d-1}{2}} f(i,j)h(n-i\,,m-j)$$

Cette opération équivaut la somme pondérée entre les pixels de l'image et les coefficients du filtre. Ici une Illustration du parcours du filtre sur la matrice image :

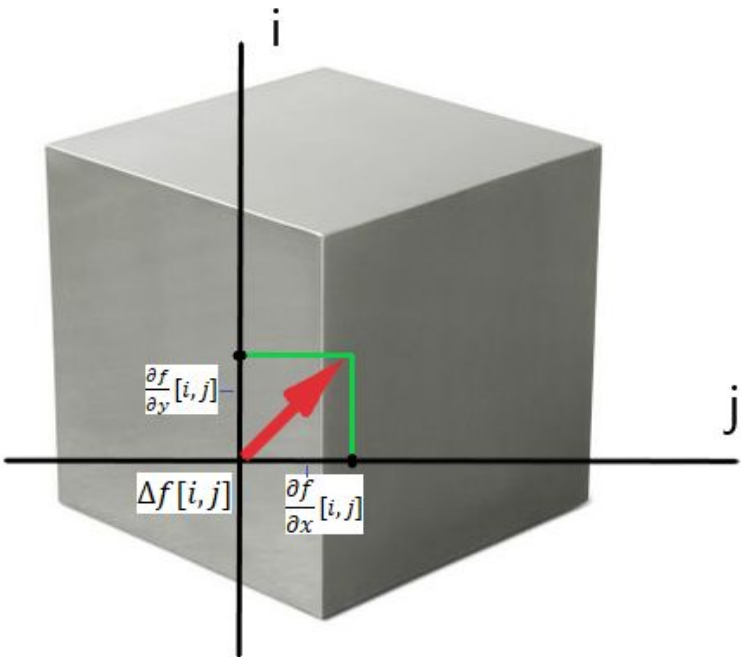


B. Les filtres

Il existe une variété de filtre avec une des usages différents. Un exemple d'utilisation courant est le **filtre médian** qui peut être utilisé pour la réduction du bruit. Le filtre récupère la valeur médiane des pixels initiaux et supprime localement les valeurs aberrantes.



Concentrons nous sur les problématiques de classification d'images et plus spécifiquement les filtres "de contours". Pour détecter un contour sur une image, on récupère les maxima locaux de la dérivée première. Les filtres différentiels permettent d'obtenir une approximation du changement de la variation locale d'intensité d'un groupe de pixel dans une direction. Ces **filtres nous indiquent les changements abrupts de luminosité** de l'image et donc exhibent les contours probables de celle-ci.



La formule du gradient continue est la suivante :

$$\frac{\partial f}{\partial x} = \lim_{h->0} \frac{f(x+h) - f(x)}{h}$$

Le passage en discret se fait de cette manière :

$$\frac{\partial f}{\partial x} = \frac{f(x_{i+1}) - f(x_i)}{1} = f(x_{i+1}) - f(x_i)$$

Plusieurs implémentations des filtres différentielles existent. Voici quelques approximations des dérivées directionnelles :

- Filtre de Prewitt :

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{et} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix} * \mathbf{A}$$

- Le filtre de Sobel :

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{et} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

Ici, on peut observer le résultat du filtrage d'une image par un filtre de Prewitt:

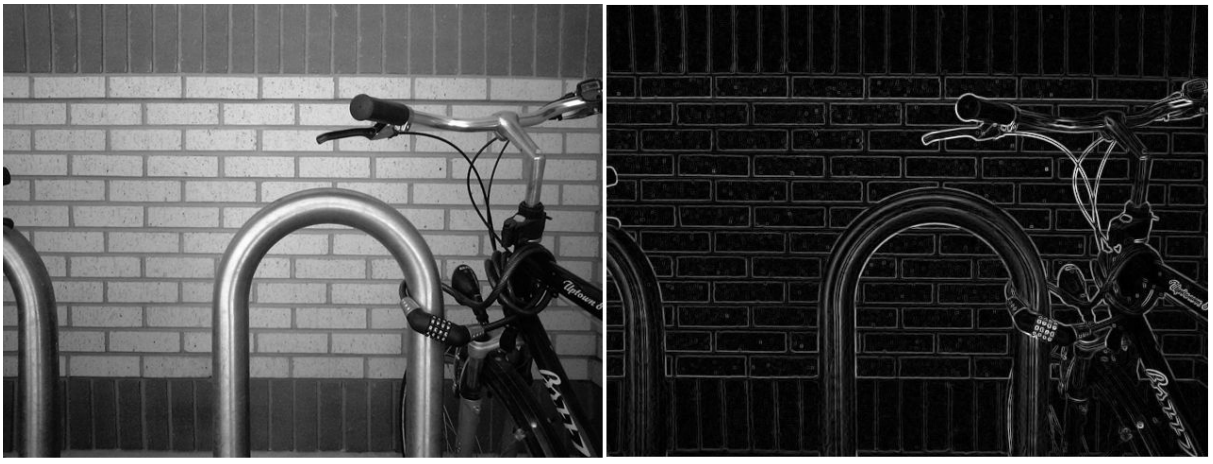


Image originale

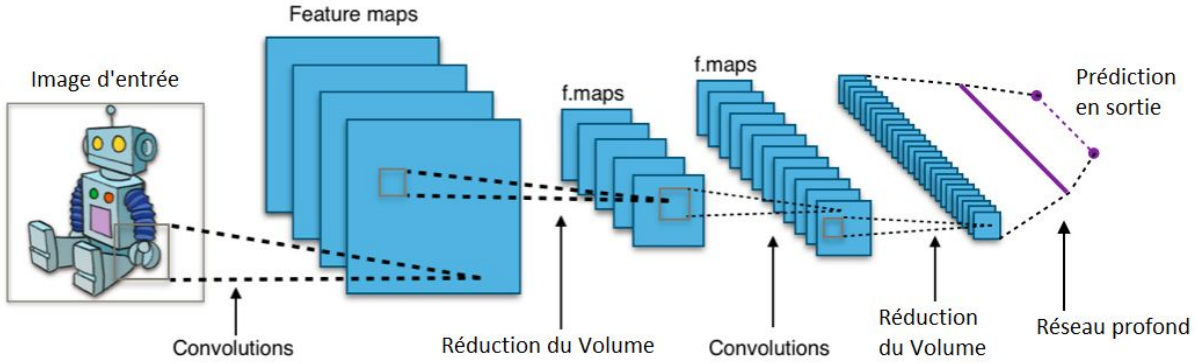
Image filtrée

II. Le model de Yann Lecun pour la reconnaissance d'image.

L'idée originale de Yann Lecun en 1989 a été d'utiliser la rétropropagation sur l'image originale pour **apprendre les coefficients des filtres automatiquement**. Les réseaux de neurones convolutifs qui en résultent sont une variation des perceptrons multi-couches. La différence est que le modèle des filtres permet une réutilisation des paramètres appris et donc une économie du nombre total de paramètres à apprendre. Nous reviendrons par la suite là-dessus. L'architecture classique du CNN (Convolutional neural Network) consiste en ces **trois couches** qui seront répétés et successivement empilés (voir schéma ci-dessous):

1. La **convolution** de l'image rentrante par les filtres.
2. L'application de la **fonction d'activation**.
3. La **réduction du volume** et des dimension (Pooling Layer)

Après plusieurs itérations de ce modèle, la matrice résultante est déroulée puis rentrée dans un réseau profond traditionnel (appelée "couche fully connected"). C'est cette dernière couche qui permet de déduire comment le réseau pénalise l'écart entre la prédiction et le label de l'image.



A) Rétropropagation

Attardons-nous sur la rétropropagation appliquée aux paramètres des filtres. Nous nous plaçons dans un cas où l'image est en noir et blanc pour simplifier la démonstration. Tout d'abord, précisons la notation :

- l est la couche numéro l qui va de 1 à la couche maximum notée L.
- E est l'erreur obtenue après la propagation avant.
- $w^l_{m',n'}$ est le pixel à l'adresse (m' , n') du filtre.
- b est le biais.
- $x^l_{i,j}$ est l'image rentrant convolué à la couche l soit:

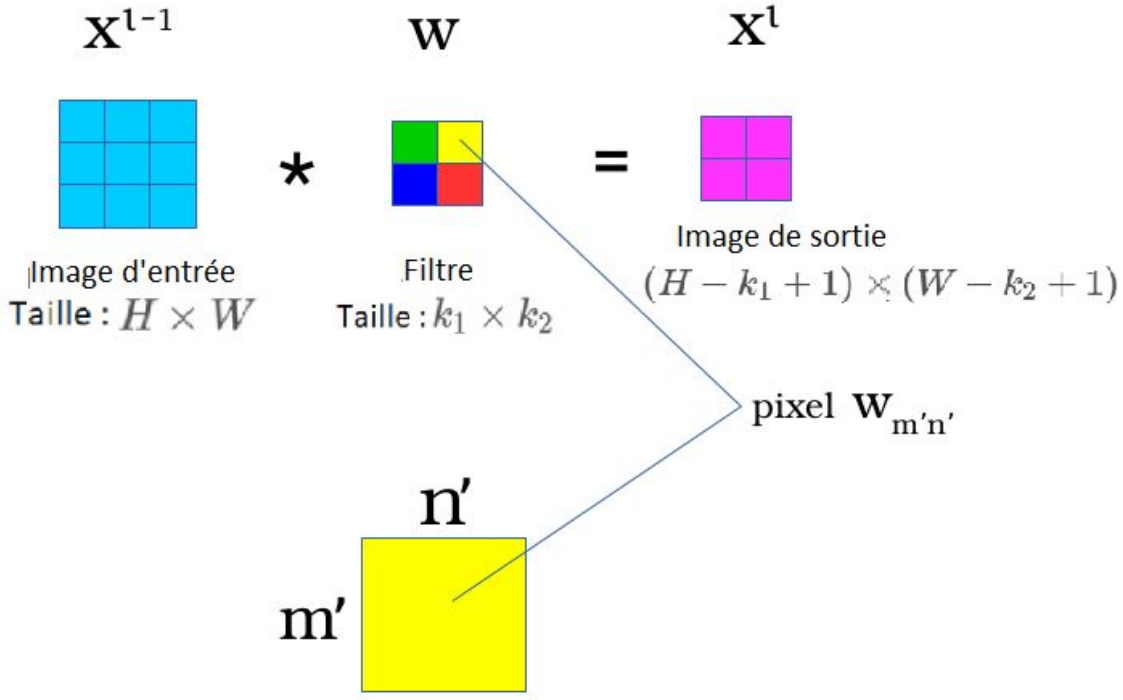
$$x^l_{i,j} = \sum_m \sum_n w^l_{m,n} o^{l-1}_{i+m,j+n} + b^l$$

- $o^l_{i,j}$ est le vecteur de sortie à la couche L soit :

$$o^l_{i,j} = f(x^l_{i,j})$$

- $f(\cdot)$ est la fonction d'activation.

Le reste de la notation est définie dans le diagramme ci-dessous.



1. Rétropropagation Etape I

Nous cherchons à calculer $\frac{\partial E}{\partial w^l_{m',n'}}$ qui peut être interprété comme la mesure du changement à opérer sur un pixel du filtre en fonction du coût. Le gradient pour chaque paramètre peut être obtenu en chainant les dérivées.
Prenons :

$$\delta^l_{i,j} = \frac{\partial E}{\partial x^l_{i,j}}$$

Alors on a :

$$\begin{aligned}\frac{\partial E}{\partial w^l_{m',n'}} &= \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2} \frac{\partial E}{\partial x^l_{i,j}} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}} \\ &= \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2} \delta^l_{i,j} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}}\end{aligned}$$

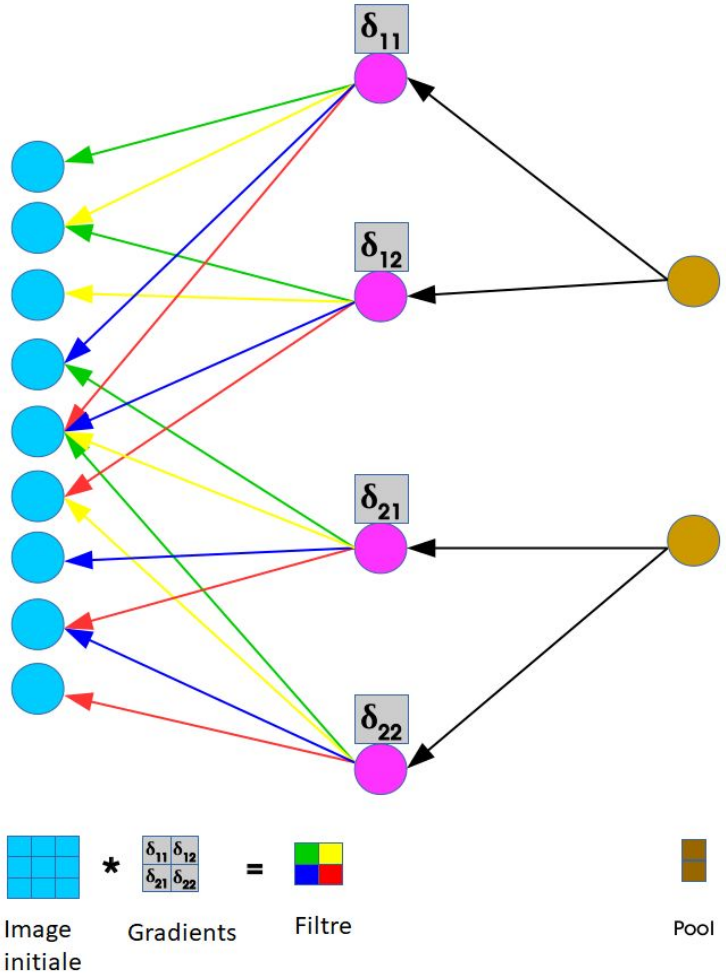
or on a :

$$\begin{aligned}\frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}} &= \frac{\partial}{\partial w^l_{m',n'}}\left(w^l_{0,0}o^{l-1}_{i+0,j+0} + \cdots + w^l_{m',n'}o^{l-1}_{i+m',j+n'} + \cdots + b^l\right) \\ &= \frac{\partial}{\partial w^l_{m',n'}}\left(w^l_{m',n'}o^{l-1}_{i+m',j+n'}\right) \\ &= o^{l-1}_{i+m',j+n'}\end{aligned}$$

En utilisant la relation entre corrélation croisée et produit de convolution, on a :

$$\begin{aligned}\frac{\partial E}{\partial w^l_{m',n'}} &= \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2} \delta^l_{i,j} o^{l-1}_{i+m',j+n'} \\ &= \mathrm{rot}_{180^\circ}\left\{\delta^l_{i,j}\right\} * o^{l-1}_{m',n'}\end{aligned}$$

L'opérateur de rotation équivaut à prendre la transposée de la matrice des gradients. Le diagramme ci-dessous montre la remontée du gradient une fois le réseau "déplié".



2. Rétropropagation Etape II

Nous cherchons à calculer les gradients :

$$\delta_{i,j}^{l} = \frac{\partial E}{\partial x_{i,j}^{l}}$$

Les coefficients deltas peuvent être interprétés comme la mesure de l'influence qu'a un pixel de l'image initiale sur la fonction coût E.

On a :

$$\frac{\partial E}{\partial x_{i',j'}^{l}} = \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \frac{\partial E}{\partial x_{i'-m,j'-n}^{l+1}} \frac{\partial x_{i'-m,j'-n}^{l+1}}{\partial x_{i',j'}^{l}}$$

$$= \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \delta_{i'-m,j'-n}^{l+1} \frac{\partial x_{i'-m,j'-n}^{l+1}}{\partial x_{i',j'}^{l}}$$

En simplifiant quelque étapes de calculs et en remplaçant $x^{l+1}_{i'-m,j'-n}$, on obtient :

$$\begin{aligned}\frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^{l}_{i',j'}} &= \frac{\partial}{\partial x^{l}_{i',j'}}\left(w^{l+1}_{m',n'} f\left(x^{l}_{0-m+m',0-n+n'}\right) + \cdots + w^{l+1}_{m,n} f\left(x^{l}_{i',j'}\right) + \cdots + b^{l+1}\right) \\ &= \frac{\partial}{\partial x^{l}_{i',j'}}\left(w^{l+1}_{m,n} f\left(x^{l}_{i',j'}\right)\right) \\ &= w^{l+1}_{m,n}\frac{\partial}{\partial x^{l}_{i',j'}}\left(f\left(x^{l}_{i',j'}\right)\right) \\ &= w^{l+1}_{m,n} f'\left(x^{l}_{i',j'}\right)\end{aligned}$$

Reprenons l'équation précédente, on trouve :

$$\frac{\partial E}{\partial x^{l}_{i',j'}} = \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1}\delta^{l+1}_{i'-m,j'-n} w^{l+1}_{m,n} f'\left(x^{l}_{i',j'}\right)$$

Qui devient en utilisant la relation entre corrélation croisée et produit de convolution :

$$\begin{aligned}\frac{\partial E}{\partial x^{l}_{i',j'}} &= \mathrm{rot}_{180^\circ}\left\{\sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1}\delta^{l+1}_{i'+m,j'+n} w^{l+1}_{m,n}\right\} f'\left(x^{l}_{i',j'}\right) \\ &= \delta^{l+1}_{i',j'} * \mathrm{rot}_{180^\circ}\left\{w^{l+1}_{m,n}\right\} f'\left(x^{l}_{i',j'}\right)\end{aligned}$$

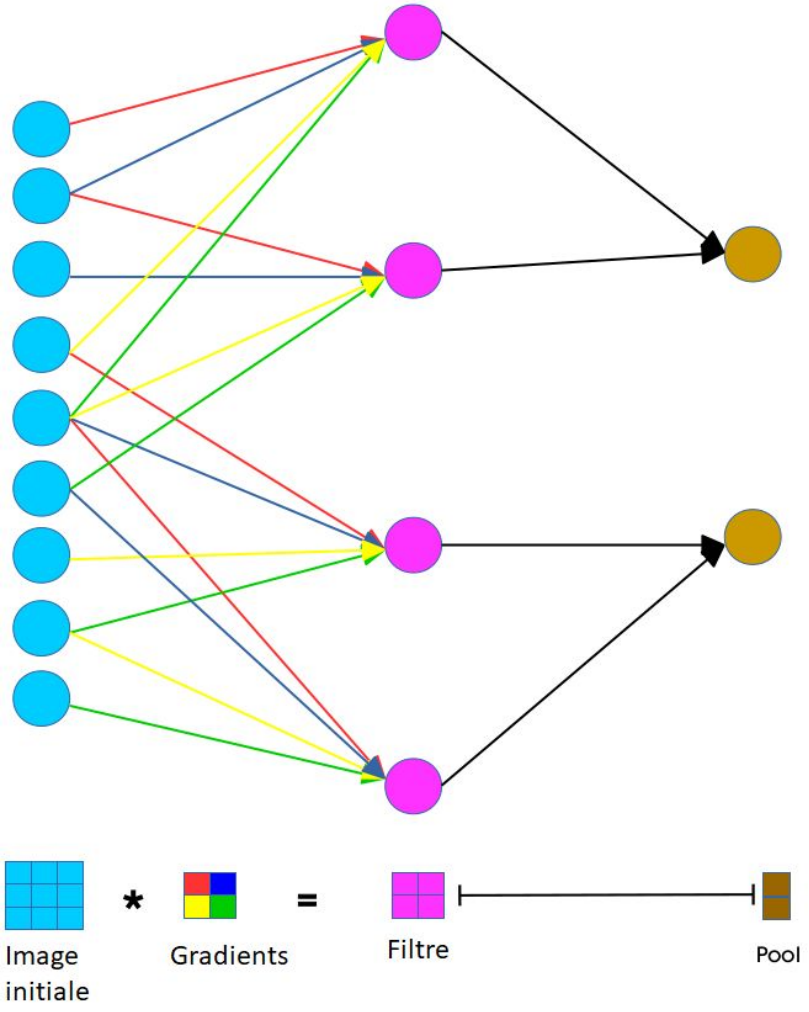
Ceci montre comment la rétropropagation peut s’appliquer sur des paramètres dans un réseau convolutionnel.

B. Ce que permettent les Réseaux convolutifs

Un intérêt notable de la convolution est son efficacité par rapport au nombre relativement faible de paramètres à apprendre. Il y a deux raisons principales pour cela.

1. La convolution par filtre n'utilise qu'une région restreinte de l'image rentrante pour trouver chaque valeur sortante. C'est ce qu'on appel la "**sparse connectivity**" (ou "connectivité clairsemée"). Une telle architecture garantit que les filtres appris produisent la réponse la plus forte à un modèle d'entrée local.

2. Un filtre est utilisé pour parcourir l'ensemble de l'image initiale contrairement aux réseaux profonds où on associe chaque paramètre à un pixel de l'image. Un même **filtre peut être utile à plusieurs endroits** d'une seule image. C'est la notion de partage de paramètre ("**parameter sharing**"). Remarquons comment un réseau convolutif, une fois "déplié", est finalement très similaire à un réseau neuronal traditionnel.

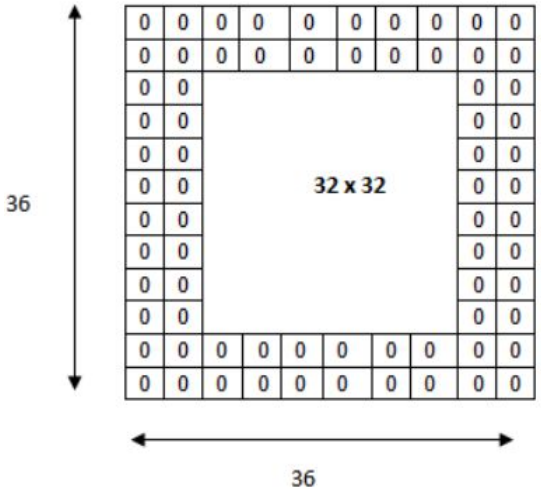


II. Le jeu des dimensions

Nous avons vu précédemment que la convolution par filtre entraînait une **baisse de la dimension**. Pour une matrice entrante carré de taille H et un filtre de taille k, la matrice qui en ressort sera de taille $((H - k + 1) * (H - k + 1))$. Prenons en exemple une situation où le filtre est de dimension (5 x 5) et l'image rentrante est de dimension de (32 x 32). Cela nous fait une image sortante de dimension (28 x 28).

A) Le padding

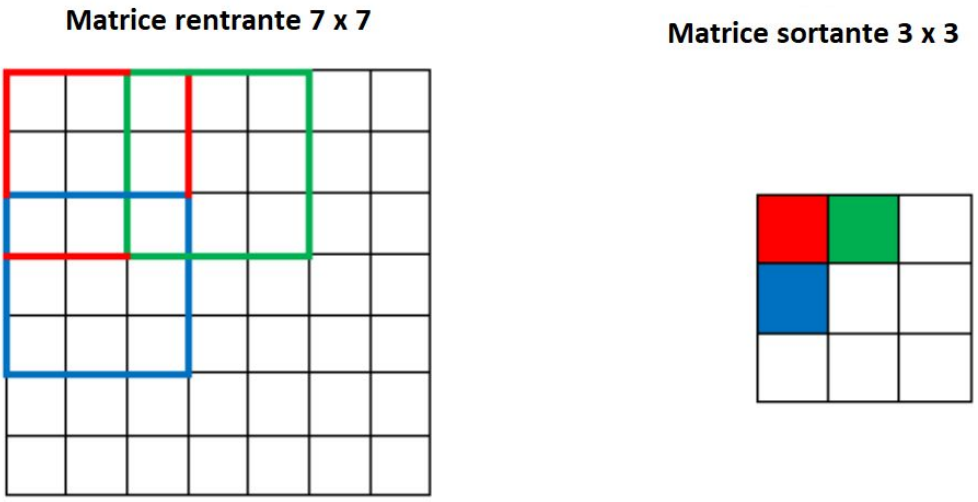
Pour pallier cet effet, la technique la plus largement utilisée est le "padding". L'idée est de contrôler la dimension des matrices sortantes en **gonflant artificiellement la dimension** de la matrice rentrante. On va donc encadrer la matrice entrante avec une ou plusieurs couches de zéros selon la taille du filtre. Ici on utilise un padding de $P = 2$.



La convolution d'une telle matrice avec un filtre de taille (5 x 5) nous donne donne bien une matrice sortante de taille (32 x 32). Pour déterminer P, on utilise la formule $P = \frac{K-1}{2}$ où K est la taille du filtre.

B. Stride

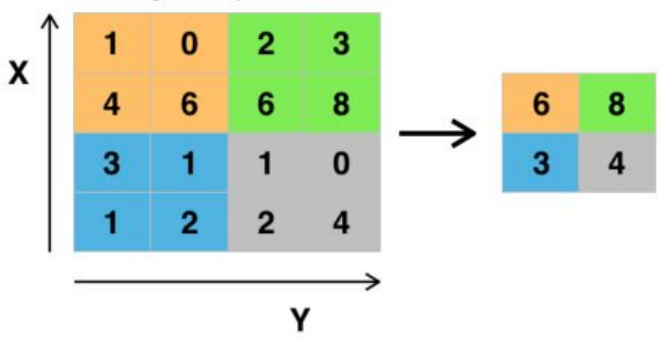
A l'inverse, il peut être intéressant de **réduire la taille de la matrice** de sortie pour diminuer la complexité des algorithmes (nombre de paramètres à apprendre et volume des matrices). Pour cela, on utilise notamment une technique appelée le "stride" (ou enjambement). Lors de la convolution, le filtre parcours la matrice entrante non-plus pixel par pixel, mais "enjambe" plusieurs pixels. Prenons une matrice rentrante de taille (7 x 7). Utiliser un stride de 2, nous donne une matrice sortante de taille (3 x 3) au lieu (5 x 5).



La formule pour calculer la matrice sortante est $MatriceSortante = \frac{(H - K + 2P)}{S} + 1$ où S est le stride. De manière générale, on n'utilise très peu de strides supérieur à deux.

C. Le Pooling

Le pooling est une technique qui **réduit** de manière très efficace le volume de la matrice sortante. L'idée est de conserver un maximum d'information utile et de réduire l' overfitting par la même occasion. Le pooling consiste à appliquer un filtre de taille k avec un stride $S = k$. Dans la très grande majorité, on utilise le filtre "max-pool" qui va renvoyer dans la matrice sortante la plus grande valeur que le filtre parcours. Dans cet exemple, un filtre de taille (2 x 2) parcourt la matrice toutes les deux cases (strides = 2). On passe d'une matrice de taille (4 x 4) à une nouvelle matrice de taille (2 x 2).



Le **Pooling par maximum** est de loin le plus utilisé. Il en existe d'autre comme le pooling par moyenne qui renvoie dans la matrice sortante la moyenne de la matrice que le filtre parcours.

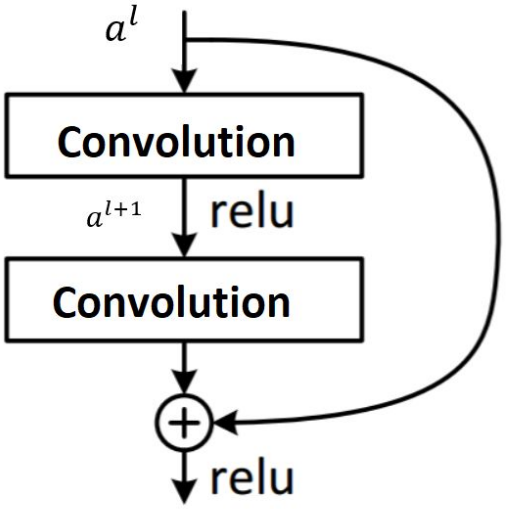
III) Quelques détails supplémentaires

A) Les Residual Neural Network

La méthode resnet (pour Residual Neural Network) consiste à utiliser les paramètres appris au début du réseau pour les utiliser deux couches plus loin.

$$a^{l+2} = f\left(z^{l+2} + a^{l}\right)$$

$$a^{l+2} = f\left(w^{l+2} * a^{l+1} + b^{l+2} + a^{l}\right)$$

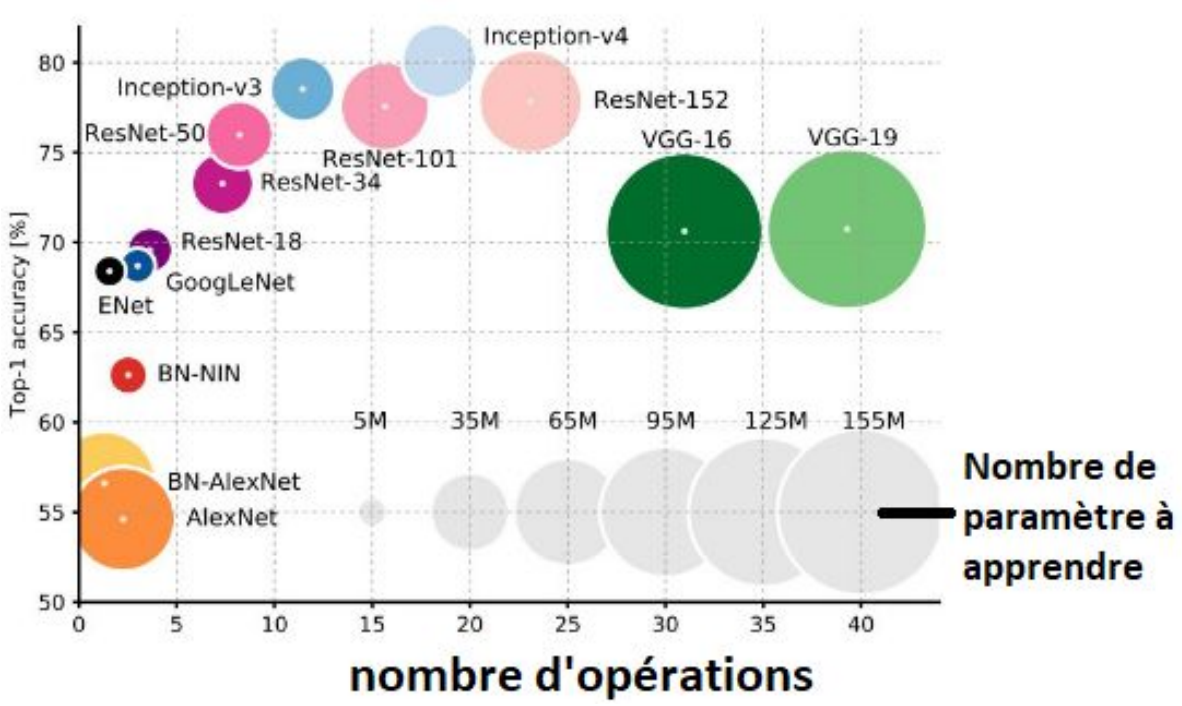


Dans le cas où on utilise la régularisation L2 ou le weight decay, alors w^{l+2} tend vers 0. Selon la régularisation utilisée, b^{l+2} va aussi tendre vers zéro. Alors a^{l+2} tend vers a^{l} ce qui revient à apprendre la fonction identité. Il permet donc de construire des réseaux beaucoup plus profonds **sans craindre que le système diverge**.

B. Choix des hyper paramètres

La configuration classique d'un CNN est celle qui a été montrée précédemment. Chaque couche de convolution a tendance à augmenter l'overfitting. Il convient alors de contrer ce phénomène par une couche de Pooling et de multiplier cette alternance jusqu'à ce que le réseau puisse extraire suffisamment d'informations du jeu de donnée. De manière relativement conventionnel, on applique la fonction d'activation après chaque convolution ce qui permet d'éviter que le calcule soit trop linéaire.

Pendant de nombreuse années, la méthodes des chercheurs consistaient à augmenter toujours le nombre de filtre et la taille du réseau (model qui atteignit son paroxysme avec les modèles VGGNet). Aujourd'hui ces méthodes laissent place à de nouvelles techniques plus efficaces, comme les modèles résiduels. Ce ne sont pas nécessairement les réseaux les plus profonds qui obtiennent le plus de réussite. Il arrive un certain palier où la profondeur du réseau ne permet pas de meilleurs résultats. Regardons pour nous en convaincre les résultats du concours ILSVRC2017 ci-dessous. Le test ILSVRC2017 est un jeu de donnée de 200 catégories de 1,2 millions de photos. Le concours test la reconnaissance et la localisation d'un objet sur une image (et non pas seulement la reconnaissance comme nous l'avons vu). La marge de progression entre le ResNet-50 et le ResNet-152 est faible.



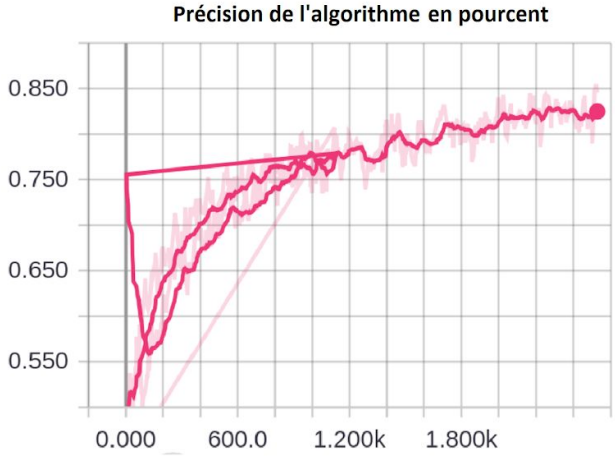
De la même manière, ce ne sont pas les algorithmes avec le plus de filtres qui l'emportent. Le succès des modèles Inception et ResNet en témoignent avec un nombre de paramètre à apprendre globalement inférieur à 60 million.

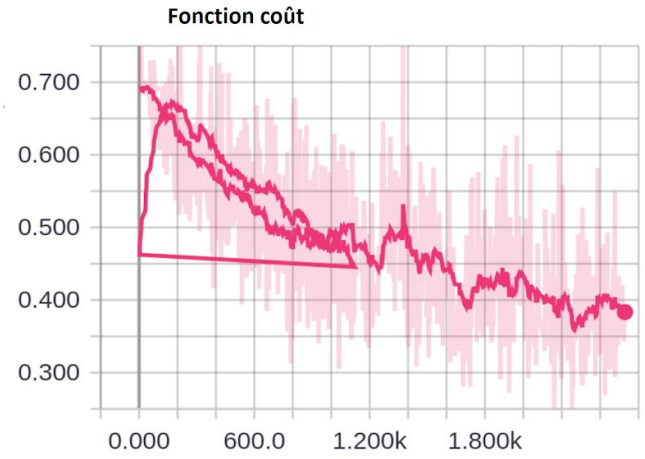
IV. Nos essais

L'application concrète des méthodes par convolution est très complexe à mettre en oeuvre. Aussi avons-nous choisi de faire confiance à des outils déjà existants. Pour choisir une bibliothèque, le plus important à nos yeux était qu'elle soit suffisamment populaire pour trouver de l'aide en ligne en cas de besoin. Nous avons choisi **TFLearn**, une librairie construite sur TensorFlow pour concevoir nos algorithmes. Ceci nous permet aussi d'utiliser l'outil Tensorboard pour tracer les courbes et avoir une vue globale du réseau.

Essais n°1

Notre premier essais consistait en un réseau pouvant classifier des photos de chats et chiens (deux classes) à partir d'un jeu de donnée de 25 000 photos. Le tableau ci-dessous en dévoile l'architecture. Notre premier réseau (très simple) n'était initialement que de deux convolution et nous a donné un taux de réussite de 52 %. Le simple fait d'ajouter 3 couches de convolution supplémentaires (indiquées en rouge) nous a permis de passer à un taux de **réussite de 84 %.** Toutes nos autres tentatives n'ont pas réussi à dépasser ce résultat.





(Activation : Relu)

Type	Nombre de filtre	Taille des filtres
Convolution 1	32	5
Pooling par maximum	strides = 1	5
Convolution 2	64	5
Pooling par maximum	strides = 1	5
Convolution 3	32	5
Pooling par maximum	strides = 1	5
Convolution 4	64	5

Pooling par maximum	strides = 1	5
Convolution 5	32	5
Pooling par maximum	strides = 1	5
Fully Connected - régression type : ADAM		

Essais n°2

Le deuxième essais était la classification sur le cifar-10. Ce test est basé sur une implémentation **déjà existante**. Nous avons uniquement changé l'architecture du réseau. Celle-ci est est donnée ci-dessous. On trouve une précision de près de 70 %. A noter que grâce aux bibliothèques et contrairement aux essais précédents avec le cifar-10, le **réseau à parcourus l'entièreté** des 60 000 images du jeu de donnée.

(Activation : Relu)

Type	Nombre de filtre	Taille des filtres
Convolution 1	32	3
Convolution 2	64	3
Pooling par maximum	strides = 2	2
DropOut	rate = 0.25	
Convolution 3	128	3
Pooling par maximum	strides = 2	2
Convolution 4	128	2
Pooling par maximum	strides = 2	2
DropOut	rate = 0.25	
Fully Connected		

Conclusion Générale

La réussite actuelle des modèles neuronaux est la concrétisation de dizaines d'années de recherche scientifique, l'augmentation exponentielle de la puissance de calcul à disposition, ainsi qu'une explosion de la quantité de données d'entraînement. Ces efforts internationaux ont menés à la création d'un paysage d'innovations mathématiques sophistiquées.

Nous avons pu étudier et comprendre les méthodes d'optimisations sous-jacentes au Deep Learning - de la conception théorique jusqu'à l'application pratique. En outre, il nous a été donné de voir comment le Deep Learning, comme n'importe quel autre champs de recherche, est demandeur en terme de ressource scientifique et informatique. Cette expérience nous a permis de poser un premier pas dans un monde à la fois complexe et passionnant.

Conclusions personnelles

Lucien :

Ce projet fut très enrichissant sur de nombreux aspects. Il a d'abord permis de découvrir de façon théorique comme expérimentale le deep learning. Domaine qui m'intéresse et m'intrigue encore plus après ce projet.

De plus, ce projet m'a permis de prendre confiance, et m'a montré qu'il est possible de s'attaquer en autonomie à un domaine dans lequel la théorie mathématique est aussi présente, et fondamentale pour bien appréhender la pratique. Cela m'a incité à faire des recherches complémentaires, et à ne pas hésiter à lire des articles issus de la recherche, ou du moins des articles rigoureux, se basant en partie sur des outils non étudiés.

Benoit :

La construction du projet a été progressive. Nous avons su développer et adapter nos ambitions au fur et à mesure que nous avancions dans le semestre. Nous avons pu enrichir nos connaissances sur les réseaux profonds et découvrir les réseaux convolutifs. Un travail de recherche assidu a été nécessaire en complément de la mise en pratique de nos savoirs. Ce projet nous a donné l'occasion d'implémenter les modèles de classification étudiés et de le faire en autonomie. Il constitue pour moi une expérience réjouissante qui préfigure ce que mon travail sera amené à devenir au fur et à mesure que je progresse vers le monde professionnel.

Bibliographie

Cours sur le deep learning

- https://www.coursera.org/specializations/deep-learning
- http://neuralnetworksanddeeplearning.com/index.html

En complément du Coursera :

- Deep Learning, d'Aaron Courville, Ian Goodfellow et Yoshua Bengio

La convolution :

- https://fr.wikipedia.org/wiki/Produit_de_convolution

Les filtres :

- Diaporama de cours SY16
- https://fr.wikipedia.org/wiki/D%C3%A9tection_de_contours

Les CNN:

- https://en.wikipedia.org/wiki/Convolutional_neural_network

La rétroPropagation:

- https://youtu.be/Ilg3gGewQ5U
- Backpropagation Applied to Handwritten Zip Code Recognition : http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf
- https://jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/?fbclid=IwAR3k1QrkwQucRpKn1DMHUNoNgLAkyy6bWaMBDq7S-2uNCDKb9FnxAY507JA

Résultat du Concours Image-NET 2017:

- https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5

Utilisation des bibliothèque TensorFlow :

- Documentation TFLearn : http://tflearn.org/doc_index/
- Documentation Tensorflow : https://www.tensorflow.org/api_docs/python/tf
- Les 50 premiers épisodes des tutoriels SentDex: https://www.youtube.com/playlist?list=PLQVvvaa0QuDfKTOs3Keq_kaG2P55YRn5v