Un réseau de neurones pour OpenSolarMap (2/3)

 27/06/2016

Après avoir essayé un algorithme très simple, puis un ou plusieurs algorithmes classiques, il est parfois (mais pas toujours) nécessaire de mettre en place un algorithme spécialisé dans le problème à résoudre. Les réseaux de neurones sont une catégorie d'algorithmes qui ont fait leurs preuves de manière spectaculaire dans le domaine du traitement d'images.

Introduction

Au delà de l'effet de mode dont ils bénéficient, les réseaux de neurones constituent bel et bien une avancée majeure en traitement d'images et dans bien d'autres domaines. Ce champ de recherche représente une proportion importante des articles parues dans les revues de référence en machine learning : NIPS et ICML. Le domaine jouit également d'une pleine reconnaissance académique comme l'illustre la chaire annuelle de l'INRIA au Collège de France en « Informatique et sciences numériques » consacrée par Yann LeCun aux réseaux neuronaux. Cette chaire, cours et séminaires inclus, constitue d'ailleurs une excellente introduction aux techniques des réseaux neuronaux, parmi les multiples ressources disponibles librement sur Internet.

Les réseaux de neurones sont étudiés depuis les années 50 avec l'invention du perceptron. Mais ceux qui bouleversent la communauté du machine learning depuis 2011 se dénomment plus précisément « réseaux de neurones profonds à convolution » (« deep convolutional neural networks », abrégé parfois en CNN pour Convolutional Neural Networks ou encore ConvNets) :

- **neurone** : Même s'il y a une lointaine analogie entre les neurones biologiques et les neurones informatiques, ils constituent deux domaines d'étude à ne pas confondre. Un neurone informatique prend en entrée plusieurs valeurs numériques et applique une fonction à ces entrées. Le résultat numérique de cette fonction constitue l'unique sortie du neurone. Le neurone Rectified Linear Unit (ReLU) est majoritairement employé : chaque entrée est multipliée par un coefficient (ou poids) puis cette somme est renvoyée si elle est positive, zéro est renvoyé sinon.

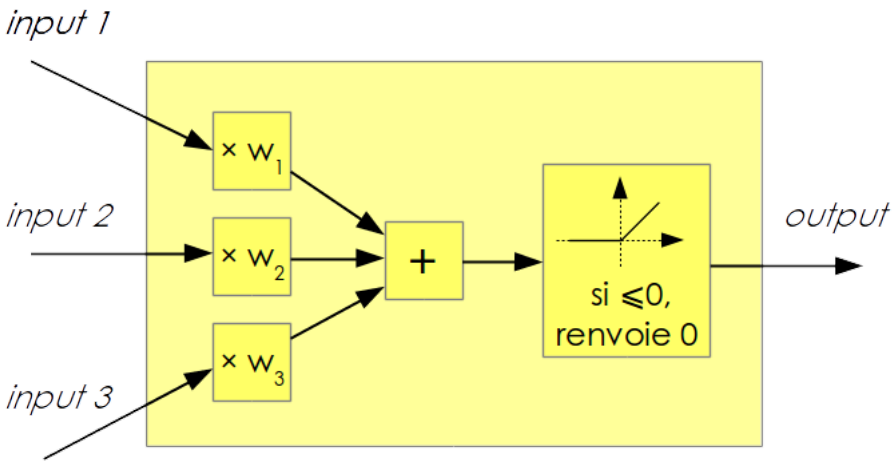


Figure 1 : neurone de type ReLU à 3 entrées.

- **réseau** : Les neurones sont disposés en un réseau qui prend la forme de plusieurs couches successives. La première couche prend en entrée les valeurs de l'image (ou d'un autre type d'entrée comme du texte ou du son). Les sorties de la première couche constituent les entrées de la deuxième couche, etc. Les sorties de la dernière couche sont les sorties du réseau, mais les valeurs numériques qui transitent entre les couches sont cachées à l'utilisateur. De là vient en partie leur réputation d'être des « boîtes noires ».

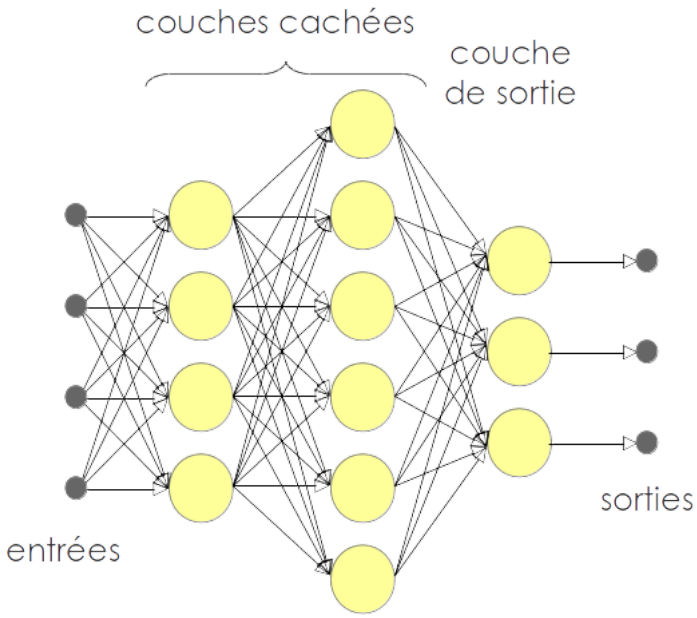
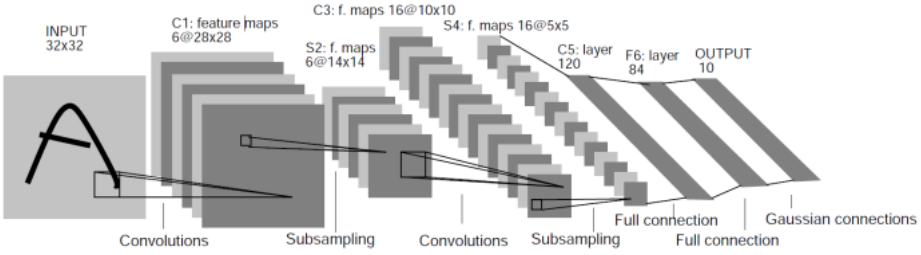


Figure 2 : réseau de neurones à 3 couches

- **convolution** *(paragraphe technique à lire en seconde lecture)* : Les réseaux à convolution opèrent généralement sur des images. La première couche de neurones est de la même forme que l'image en entrée. La sortie de cette première couche, comme toutes les sorties intermédiaires, forment des images. Au sein d'une couche de neurones, les paramètres de chaque neurones sont choisis de telle sorte que la couche applique un filtrage linéaire puis une rectification. Un filtrage linéaire est la convolution entre une image d'entrée et un filtre linéaire. Un filtre linéaire, dans le cas du traitement d'images, se représente come une petite image, typiquement de taille 3 par 3 pixels ou 5 par 5. Les réseaux à convolution ont l'avantage de tirer partie de la structure géométrique de l'image d'entrée. De plus, chaque couche de neurones est paramétrée par un filtre linéaire qui est beaucoup plus simple à apprendre que dans le cas général. Par exemple, pour une image d'entrée de 224 par 224 pixel en noir et blanc, une couche de neurones de la même taille est composé de 224 x 224 neurones et si chaque neurone est connecté à chaque pixel d'entrée, il y a 224 x 224 paramètres par neurones. Cela fait un total de 224 x 224 x 224 x 224 = 2.517.630.976 paramètres pour cette seule couche. Il faudrait donc des milliards d'images pour faire apprendre correctement un tel réseau. En comparaison, paramétrer la couche de neurones par un filtre de 3 x 3 pixels ne requiert d'apprendre que 9 valeurs numériques. Concrètement, cela revient à mettre la majorité des poids des neurones à zéro, et à partager tous les poids restants entre les neurones de la couche. Dans un réseau à convolution, des étapes de réduction de la taille de l'image s'intercalent entre les couches de neurones. Pour le réseau LeNet 5, deux étapes de réduction, appelées « subsampling » alternent avec les deux étapes de convolution. Les dernières couchent perdent la structure géométrique en dépliant l'image sur une dimension, mais le nombre de paramètres à apprendre est raisonnable du fait de la petite taille des images. Enfin, il faut préciser que, de la même manière qu'une image d'entrée peut contenir plusieurs canaux de couleur (rouge, vert et bleu par exemple), les images intermédiaires se composent de plusieurs canaux. Dans le cas de LeNet 5, les images intermédiaires se composent de 6 puis de 16 canaux. Au fil de l'apprentissage du réseau, chaque canal va se spécialiser dans la reconnaissance d'une forme géométrique particulière.
- **profond** : Les réseaux de neurones traditionnels, étudiés dans les années 70, utilisaient entre 1 et 3 couches de neurones. On parle de réseaux profonds pour parler des architectures avec un nombre élevés de couches qui peut dépasser la centaine ! Les couches proches de l'image d'entrée se spécialisent dans la détection de features géométriques très simples (des coins, des lignes…) alors les couches finales détectent des features abstraites qui dépendent de l'usage du réseau (des lettres pour un réseau de reconnaissance d'écriture, des objets, des espèces d'animaux…).



Fgure 3 : réseau de neurones LeNet 5

Comme tous les algorithmes dits supervisés, les réseaux de neurones sont « appris » sur un échantillon de données d'exemples labellisés. La méthode d'apprentissage utilisée, nommée backpropagation, va modifier petit à petit les paramètres de chaque couche pour augmenter la qualité des prédictions jusqu'à atteindre une situation (localement) optimale. Une fois la phase d'apprentissage terminée, le réseau est capable de faire des prédictions sur de nouvelles images.

Les réseaux de neurones présentent des performances spectaculaires et inattendues. Un des enjeux théorique actuel est de comprendre ces performances et de les confirmer par des garanties théoriques. C'est par exemple l'objet des recherches actuelle de Stéphane Mallat qui travaille sur la méthode de scattering, apparentée aux réseaux de neurones.

Librairie, réseau et code utilisés

Librairie de deep-learning

Les librairies de deep-learning ne manquent pas. La difficulté est de choisir l'outil qui répond le mieux aux besoins du projet. Pour répondre aux besoins d'OpenSolarMap, l'outil idéal devra :

- être utilisable facilement et rapidement, c'est-à-dire gérer lui-même la totalité des calculs
- laisser la possibilité de faire des modifications simples sur le réseau utilisé
- être en open source pour pouvoir être essayé dans l'heure (et passer à un autre outil s'il ne correspond pas parfaitement)
- proposer une API en python car c'est le langage que nous utilisons majoritairement à l'AGD
- rassembler une communauté active, pour disposer d'exemples d'utilisation, pour disposer de nombreuses questions/réponses sur stackoverflow.com, pour pouvoir compter sur l'aide de la communauté en dernier recours
- en bonus, pouvoir utiliser la puissance de calcul des GPU.

Parmi d'autres solutions qui auraient satisfait ces critères, j'ai choisi Keras sur les recommandations d'un confrère data-scientist. Je n'ai pas eu à regretter ce choix, mais si Keras n'avait pas convenu, j'aurais sans doute testé les outils Caffe, Lasagne ou le très populaire Torch même s'il est écrit en lua.

Réseau de neurones VGG16

Concevoir un réseau de neurones est une tâche compliquée qui nécessite une expérience approfondie. En revanche, utiliser une réseau de neurones déjà prêt est beaucoup plus simple et rapide à mettre en œuvre. Le « transfer learning » une technique standard expliquée dans ces notes du cours de vision CS231n de Standford.

Il y a déjà de nombreux réseaux pré-entraînés disponibles. Le projet Model-Zoo de l'université de Berkeley est une liste de ces réseaux qui est d'un grand usage pour faire ce choix. Les réseaux listés par Model-Zoo sont proposés en Caffe, mais la plupart des modèles sont directement utilisables avec d'autres librairies. J'ai choisi arbitrairement le célèbre réseau à 16 couches VGG16 du Visual Geometry Group de l'université d'Oxford utilisé lors de la competition ILSVRC de 2014 (ImageNet 2014). Il s'agit d'un réseau généraliste. D'autres réseaux auraient pu convenir aussi bien et peut-être même mieux. Ce réseau est directement utilisable avec Keras en utilisant le GitHub Gist suivant : https://gist.github.com/baraldilorenzo/07d7802847aaad0a35d3

```
model = Sequential()
model.add(ZeroPadding2D((1,1),input_shape=(3,224,224)))
model.add(Convolution2D(64, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(64, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(128, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(128, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1000, activation='softmax'))
```

La première ligne définit un modèle séquentiel. D'autre architectures plus compliquées existent, mais le réseau VGG16 est un empilement de couches dont chaque couche prend en entrée le résultat de la couche précédente. les lignes

```
model.add(Convolution2D(xxx, 3, 3, activation='relu'))
```

définissent des couches de neurones ReLU avec une taille de filtre de 3 par 3 pixels et un nombre de canaux de 34, 128, 256 ou 512. Les lignes

```
model.add(MaxPooling2D((2,2), strides=(2,2)))
```

définissent les étapes de réduction de la taille de l'image le long du réseau. La méthode Max Pooling est utilisée et chaque étape divise par 2 la taille. Les lignes

```
model.add(ZeroPadding2D((1,1)))
```

sont un détail d'implémentation. Pour compenser la taille du filtre de 3 pixels, une bordure de 1 pixel est ajouté à l'image d'entrée avant chaque convolution pour que la sortie de la convolution soit de même taille. Enfin, le dernier bloc définit 3 couches de

neurones fully-connected.

La taille d'entrée spécifiée est 224 par 224 pixels (par 3 canaux). Chaque couple Zero Padding / Convolution ne modifie pas la taille (éventuellement le nombre de canaux) et chaque Max Pooling divise la taille par 2 (sans modifier le nombre de canaux). On peut en déduire que l'opération Flatten prend en entrée une image de 7 par 7 pixels avec 512 canaux et renvoie un vecteur de taille 25.088. Les deux couches suivantes comptent 4096 neurones puis la dernière en compte 1000, qui correspondent aux 1000 classes à prédire pour la compétition ImageNet.

Modification du réseau

Le réseau a été entraîné pour classer une image parmi 1000 classes de la compétition ImageNet et non pas pour distinguer l'orientation des toitures. Mais une propriété des réseaux de neurones est qu'ils sont également très efficace pour s'adapter à des problèmes voisins. Seules les dernières couches sont spécialisées dans la tâche à accomplir tandis que les premières couches résolvent des problèmes de détection très généraux. Le transfert learning utilise cet avantage pour utiliser très rapidement un réseau pour une nouvelle tâche. La méthode la plus simple, sans fine-tuning, a été utilisée pour OpenSolarMap.

Changement de la taille d'entrée

Le réseau VGG16 prend en entrée des images de taille 224 par 224, or les images de toits ont une taille moyenne de 95 par 93 pixels (voir figure 4). On a le choix d'agrandir les images avant d'appliquer le réseau, ou de changer la taille d'entrée du réseau vers une valeur plus proche de la taille moyenne. La première approche fournirait des images majoritairement floues au réseau, c'est pourquoi j'ai choisi la seconde approche. La taille de 96 par 96 est idéale, car après les 5 étapes de réduction, la taille de l'image intermédiaire tombe « juste » sur 3 par 3 pixels.

La ligne 2 du gist est remplacée par :

```
1 model.add(ZeroPadding2D((1,1),input_shape=(3,96,96)))
```

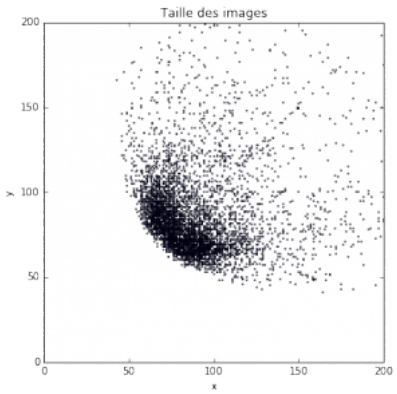


Figure 4 : répartition des images de toits par taille (x, y)

Transfer learning

Les dernières couches sont spécifiques au concours ImagetNet et ne sont pas d'utilité ici. Ils sont donc retirés et le dernier bloc est remplacé par

```
1 model.add(Flatten())
```

Le choix d'enlever 3 couches est arbitraire. La seule contrainte à respecter est de conserver un nombre de features raisonnables, inférieur à 10.000. Avec ce choix, on compte alors 3 x 3 x 512 = 4608 features en sortie du réseau. Ce n'est pas une prédiction de la classe, mais ces features peuvent être utilisées par un classifieur comme une régression linéaire.

Résultats

Les 4608 features calculées par le réseau alimentent un modèle de régression logistique, régularisé cette fois. Le paramètre de régularisation varie et le taux d'erreur (figure 5) affiche une classique courbe en U du dilemme biais-variance. C'est la valeur qui donne le meilleur résultat qui est retenue. La force de la régularisation est un hyperparamètre, donc il est indispensable de calculer la performance sur un échantillon de validation. Le taux d'erreur sur cet échantillon est de 7%.

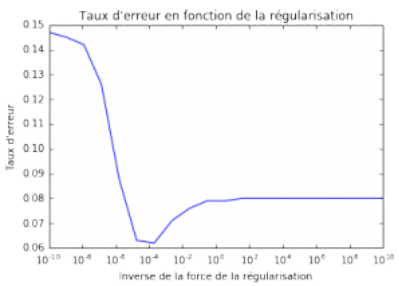


Figure 5 : taux d'erreur en fonction de la force de la régularisation

Ce taux d'erreur est bien moins bon que ce qu'il serait possible d'obtenir avec des méthodes « state-of-the-art ». Tout d'abord, il faudrait tester d'autres réseaux que le VGG16, essayer différents choix de couches à enlever voire faire du fine-tuning… Ici, il s'agissait surtout de démontrer qu'il est possible d'arriver très rapidement à des résultats satisfaisants, en utilisant un réseau, une librairie et du code déjà prêt à l'emploi.

À propos de l'auteur: Michel Blancard

Tags: Datasciences Energy Machine-learning OpenSolarMap