SQLI

Le Deep Learning pas à pas

Manuel Alves et Pirmin Lemberger

PARTIE II – Implémentation

L'engouement actuel pour le Deep Learning ne repose pas sur les seules avancées conceptuelles de Hinton et al. mais aussi sur des avancées technologiques. Après l'introduction aux concepts présentés dans la partie I de cet article, nous abordons ici les questions liées à l'implémentation de ces réseaux.

L'enjeu majeur de la performance

Pour un informaticien, l'implémentation d'un DBN repose principalement sur le calcul de la formule (7) (Lire Le Deep Learning pas à pas : les concepts)

$$-\frac{\partial}{\partial\theta}\log p_\theta(\mathbf{v}) = \frac{\partial F(\mathbf{v})}{\partial\theta} - \sum_{\mathbf{u}\in\mathcal{E}} p_\theta(\mathbf{u})\frac{\partial F(\mathbf{u})}{\partial\theta} \qquad (7)$$

Malgré les simplifications astucieuses obtenues en utilisant l'algorithme de Contrastive Divergence et en choisissant les RBM comme briques élémentaires, les expressions mathématiques à évaluer restent très coûteuses en temps de calcul. La question de la performance des algorithmes est donc vitale pour tout outil ou langage destiné à ce champ d'application. A ce titre, toutes les solutions sérieuses disponibles aujourd'hui sont capables d'exploiter l'immense réservoir de puissance de calcul que constituent les ordinateurs modernes, aussi bien en sollicitant le processeur principal (CPU) que les processeurs graphiques dédiés (GPU). Ces derniers sont fabriqués par des fondeurs spécialisés de cartes graphiques comme NVidia ou AMD. Ils sont à l'origine prévus pour le calcul 3D dans le monde du jeu vidéo ou de l'imagerie de synthèse, ou plus récemment sur le traitement vidéo. Quel lien alors avec le sujet du Deep Learning ?

Les GPU ont des fréquences d'horloge bien moindres que celles des CPU (20x) mais ils possèdent de nombreux cœurs (unités de calcul). A titre d'exemple, sur une carte graphique de dernière génération de marque NVidia le GPU intègre 5760 cœurs pour 12Go de mémoire. Ces composants sont parfaitement adaptés aux traitements parallélisables de données de grande dimension (multiplication de matrices, convolution etc.). Par ailleurs, les fabricants se sont attelés à repousser encore plus loin les possibilités offertes par ce type d'architecture. Les technologies *SLI* (NVidia) ou *Crossfire* (AMD) permettent utilisation jusqu'à 4 GPU dans la même machine, démultipliant quasi proportionnellement la puissance de calcul. Les librairies comme *CUDA* (chez NVidia), avec ses extensions *C ,C++, openAAC*..., permettent aux développeurs de coder des applications accélérées matériellement par les GPU compatibles, tirant parti du parallélisme offert par la plate-forme. Lorsqu'il s'agit d'entraîner un réseau de neurones comportant plusieurs milliards de connexions ou d'essayer simultanément plusieurs modèles sur un jeu de données, cette aptitude à la parallélisation constitue un atout majeur.

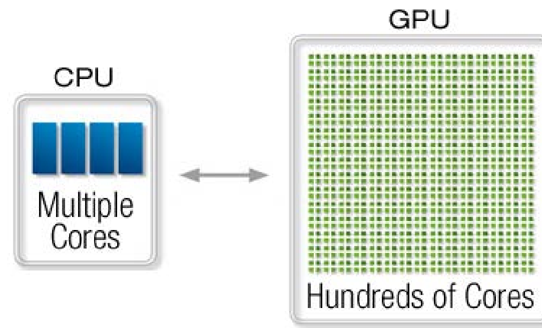

Figure 1: Architecture GPU vs CPU

Des travaux récents produits 2014, ont conduit à la construction d'un cluster de 64 GPU conçu spécifiquement pour l'entraînement d'un Convolutionnal Neuronal Network (CNN) à 18 couches de neurones pleinement interconnectées, soit 212.7M de poids synaptiques (w). Appliqué au cas d'usage académique *Image Net Large-Scale Visual Recognition Challenge* (ILSVRC) cette nouvelle architecture a décroché le nouveau record de taux de reconnaissance sur les 1,2 millions d'images classées dans 1000 catégories. Un résultat désormais supérieur à celui de l'œil humain !

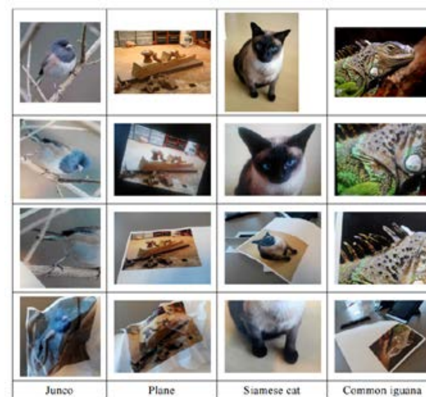

Figure 2 : Photos extraites du catalogue ILSRVC. Expérimentations de légendages automatiques.

Une offre pléthorique d'outils et langages

Pour un *data scientist* désireux de s'aventurer dans le codage d'un réseau de neurones ou d'un DBN, les frameworks et langages à disposition sont nombreux. Les principaux ténors du sujet sont : *Theano*, un projet de l'université de Montréal, *Torch*, une solution maintenue par des experts œuvrant chez de grands acteurs du web (Google, Facebook ou Twitter) et *Caffe*, une initiative de l'université de Berkeley. De nombreuses autres alternatives sérieuses fonctionnent en sur-couche. Citons *Pylearn2*, une librairie Python de machine learning reposant sur des fondations *Theano* et une alternative, en mode boîte blanche, à l'incontournable *sickit-learn*. Ajoutons la librairie Python Keras, sur base de *Theano* également, mais réellement spécialisé dans le Deep Learning. On trouve également des projets exploitant *Cuda-Convnet*, une implémentation performante *C++/Cuda* des réseaux de neurones classiques. Comme dans tant d'autres domaines, force est de constater qu'il n'y a pas de solution ultime qui surclasserait toutes les autres. Chaque solution aborde le sujet avec une approche spécifique qui conviendra à telle ou telle communauté sans pour autant créer l'unanimité.

Theano, outil par excellence pour coder un DBN ?

Theano est un compilateur, transformant du code Python en instructions *CUDA* et permettant de faire de la programmation symbolique. Il n'est donc pas exclusivement réservé aux DBN. La communauté est importante et active, et la documentation est des plus complètes pour un produit open-source. Le

principe est d'écrire le code, non pas comme un informaticien, mais plutôt comme un mathématicien et de déléguer à *Theano* le travail difficile et fastidieux de passer d'une formule mathématique à du code informatique efficace et performant.

Une somme ne s'écrit donc pas comme une boucle *for* mais comme une instruction Python **Theano.SUM**(). Le développeur ne calcule pas un dérivée numériquement $[f(x+h)-f(x)]/h$ mais l'exprime naturellement avec une instruction déclarative du type **Theano.grad()**, en passant en paramètre un objet **Theano.function**(). C'est *Theano* qui dans la phase de compilation va opérer la transformation de ces déclarations *Python* en opérations informatiques optimisées. En outre *Theano* s'appuie entièrement sur le package scientifique standard *NumPy*, aucune révolution du côté de la syntaxe et des manipulations de base. Enfin le code compilé, sur les parties pertinentes de calcul, est directement exprimé en instructions *CUDA*.

L'intérêt majeur d'intervenir à ce bas niveau d'abstraction est la possibilité de spécifier intégralement, en boîte blanche, une architecture de réseaux de neurones. Les fonctions de coûts les plus exotiques ont toute leur place, et le calcul de leurs gradients/dérivés ne sera plus un casse-tête d'informaticien grâce à *Theano*. La liberté est donc maximale et particulièrement utile en cas d'expérimentation. Cependant cette liberté a un coût. Le travail de programmation symbolique est loin de faire l'unanimité et son accès n'est pas des plus aisés, *a minima* il nécessite une nouvelle façon de penser. Autre point non négligeable : la latence induite par la compilation de son travail, et celle-ci peut prendre plusieurs minutes suivant les cas. L'expérimentation étant partie intégrante de la datascience cela peut s'avérer frustrant. Ceci conduit certains développeurs de réseaux à envisager d'autres solutions spécialisées comme *Torch* ou *Keras*.

Pour illustrer le propos, ci-dessous des extraits de code *Theano* pour instancier une classe RBM (*source : http://deeplearning.net/tutorial/code/rbm.py*). Les variables symboliques de Theano sont nommées ici vbias, hbias en place et lieu des paramètres a, b et w de la partie I de cet article.

```
# Paramètres a,b,w de la formule 5 préalablement initialisés dans le code
self.params = [self.W, self.hbias, self.vbias]
[...]
# Energie effective (formule 8) définit par :
    def free_energy(self, v_sample):
        vbias_term = T.dot(v_sample, self.vbias)
        wx_b = T.dot(v_sample, self.W) + self.hbias
        hidden_term = T.sum(T.log(1 + T.exp(wx_b)), axis=1)
        return -hidden_term - vbias_term
[...]
# Implémentation de contrastive divergence pour obtenir un échantillon nv_samples
# distribués selon la probabilité souhaitée
 [...]  nv_samples [...]  = theano.scan(
      # fonction implémentant 1 step de deep sampling
       self.gibbs_hvh,
      # où chain start, une initialisation astucieuse par tirage d'une des
      # configurations au hasard
       outputs_info=[None, None, None, None, None, chain_start],
      # Nombre d'itérations limitant le temps d'attente de la convergence
       n_steps=k
)
[...]
# Déclaration du gradient de coût, où chain_end est le dernier élément de la chaîne
# nv_samples obtenu par l'échantillonnage de Gibbs précédent
cc = T.mean(self.free_energy(self.input)) - T.mean(self.free_energy(chain_end))
gparams = T.grad(cc, self.params, consider_constant=[chain_end])
cost = get_pseudo_likelihood_cost(gparams)
[...]
# Entraînement RBM , et calcul de la fonction de coût
train_rbm = theano.function(
       [index],
```

SQLI

```
    # fonction de coût que Theano doit calculer
    cost,
   # nouvelles valeurs de toutes les variables partagées, ici self.params
   updates=updates,
   ,[...])
```

Caffe : une approche « packagée »

A un niveau d'abstraction totalement différent, nous trouvons *Caffe* et son approche très opérationnelle. Il s'agit d'un exécutable dédié aux réseaux de neurones (non profonds) et développé en *C++/Cuda*. L'approche est davantage celle d'un framework, c'est çà dire un canevas général offrant une grande souplesse par paramétrage. *Caffe* est nativement utilisable en ligne de commande mais offre également des interfaces (wrappers) en *Python* ou même *Matlab*. Le cas échéant pour une personnalisation plus avancée, la librairie originale en *C++* est accessible et pourra être re-compilée. Ce "moteur" construit ainsi des réseaux de neurones suivant les spécifications définies dans des fichiers plats à l'extension douteuse .prototxt. La syntaxe adoptée, *Protobuffer* (alternative à *JSON*) s'attache à décrire efficacement des structures d'objets, en texte plein, et donc aisément lisibles par un humain. On y trouve 3 types de déclarations d'objets à choisir parmi un catalogue bien fourni :

- **`Blob`** : définition des données à manipuler dans le réseau. Nous y trouverons principalement le set de données d'entraînement, la liste de labels/catégories, les données en sorties du réseau de neurones.
- **`Layer`** : composant fondamental de *Caffe*. Définition du type d'opération à réaliser dans une couche donnée comme les convolutions, les filtres, l'application d'une fonction sigmoïde, le calcul de fonctions de coût usuelles ...
- **`Solver`** : objet exclusivement dédié au paramétrage de l'entraînement du modèle constitué de l'ensemble des objets "**`Layers`**". On y trouve le choix de l'utilisation d'un SGD, le nombre maximal d'itérations autorisées, l'activation du calcul via GPU ...

Tout le travail consiste donc avec cette syntaxe "light" à empiler sur un jeu de données (**`Blob`**) des opérations d'activations (eq. **`Layers`**) dont la dernière est *a fortiori* une fonction de coût, à savoir un layer de type particulier **`EUCLIDEAN_LOSS`**, **`SOFTMAX_LOSS`**, **`HINGE_LOSS`**... Le nombre de neurones par couche est ajusté à l'aide du paramètre **`num_ouptut`**. En lançant l'apprentissage par invocation du **`Solver`**, *Caffe* va alors automatiquement itérer des allers-retours sur cette pile de **`Layers`**, le retour / back-propagation, se faisant lors du calcul de la fonction de coût. Par la suite, le calcul de gradient assuré de façon transparente par *Caffe* lui permet de réajuster les poids synaptiques.

Exemple de déclaration d'une couche d'un réseau de neurones avec *Caffe*

```
layers {
  name: "ip1"
  type: INNER_PRODUCT   # déclaration d'une couche/layer de neurones totalement
connectés
  blobs_lr: 1.  # coefficient de learning rate pour les paramètres « filters »
  blobs_lr: 2. # coefficient de learning rate pour les paramètres « biases »
# paramètres spécifiques au type de layer
  inner_product_param {
    num_output: 1000  # nombre de neurones
    weight_filler {
    type: "gaussian"
    }
}
```

A noter enfin que *Caffe* ne possède pas (encore ?) de layer RBM ou Autoencoder spécifiques aux réseaux de neurones profonds. Nul doute alors qu'un Caffe pour DBN, ou équivalent, sur un mode très packagé, devrait voir le jour tant les espoirs sont grands concernant cette nouvelle classe d'algorithmes. Rendre leur implémentation plus accessible est une voie inéluctable.

Conclusion

Après des décennies de stagnation et d'échecs, les architectures profondes constituent un véritable « revival » des réseaux de neurones. Dans des disciplines comme l'IA, la reconnaissance visuelle où des progrès de quelques pourcent sont considérés comme de véritables percées, les architectures profondes occupent aujourd'hui une place de premier plan. Peut-on considérer pour autant que la pierre philosophale de l'IA a été découverte ? Assurément non.

Les espoirs déçus des années 50 nous ont trop enseigné la prudence. Des débats passionnants existent aujourd'hui dans la communauté de l'IA et du machine learning au sujet de la pertinence des architectures profondes dans l'objectif de réaliser une authentique IA. L'avenir est-il à l'imitation de la nature comme semble le penser les partisans des RN ? Les grandes inventions du passé, comme l'électricité ou le moteur à explosion, suggèrent plutôt que les grandes innovations ont tendance à s'en démarquer. Les architectures profondes semblent adaptées à la reconnaissance de formes mais elles ne semblent pas être capables de raisonnement logique. Même s'il est peu vraisemblable qu'une seule classe d'idées comme celles des Hinton et al. puisse venir à bout d'un problème aussi colossal que la conception d'une IA, les réseaux de neurones profonds feront assurément partie des recherches dans cette direction ces prochaines années.