Gestion de données massives

Lecocq Thomas Pascal Sitbon

February 18, 2016

Abstract

Ce document sert d'accompagnement à notre projet de Gestion de données massives pour l'ENSAE. Nous avons décider de participer au challenge Kaggle *Data Science Bowl 2* - https://www.kaggle.com/c/second-annual-data-science-bowl. Nous proposons un algorithme de deep learning basé sur les *convolutional neural networks*, implémenté en python à l'aide des package Theano, Keras et au logiciel CUDA. Notre objectif est double, d'une part nous allons comparer le temps d'execution des calculs de notre modèle entre l'utilisation du GPU et du CPU. Nous allons également tenter d'identifier les causes qui font que l'utilisation du GPU conduisent à de meilleures performances.

Keywords: GPU, CUDA, Deep Learning, CNN, Theano, Keras.

1 Le Challgenge et ses données

Nous avons choisit de participer au challenge Kaggle Data Science Bowl 2 pour notre projet de gestion de données massives. L'objectif est de prédire le volume maximal et le volume minimal d'un coeur à partir de 30 images IRM de ce coeur en action. Nous disposons de plusieurs milliers (5331 patients différents) points de données. L'utilisation de techniques de gestions de données massives est alors indispensable, afin de réduire le temps d'execution de nos algorithmes. Notre algorithme de Deep Learning est basé sur différents packages python incluant Theano, Keras et sur le logiciel CUDA.

2 Installation de CUDA sur nos machines

Nous avons réussi à installer Cuda et Theano sur nos machines respectives. Nous pensions qu'il serait intéressant de mettre la démarche à disposition pour les élèves:
Windows 8.1 / VS2013(Community) / CUDA7.0

1. Install Visual Studio
2. Install Miniconda (includes Python)
3. conda install pip six nose numpy scipy
4. Install CUDA Toolkit 7.0
5. conda install mingw libpython

6. pip install theano
7. Add a "THEANO_FLAGS" environment variable with value "floatX=float32,device=gpu,nvcc.fastmath=True"
8. Add path to VS's C++ compiler (VC.exe) to your PATH environment variable e.g. C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin

Windows 10 / VS2013(Community) / CUDA 7.5

1. Install Anaconda
2. conda install pip six nose numpy scipy - already ships with Anaconda I think
3. conda install mingw libpython
4. pip install theano
5. Install Visual Studio 2013 Community Edition
6. Install CUDA Toolkit 7.5
7. Add a "THEANO_FLAGS" environment variable with value "floatX=float32,device=gpu,nvcc.fastmath=True"
8. Add path to VS's C++ compiler (cl.exe) to your PATH environment variable, e.g. C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin

3 Description de notre modèle

CNN avec 3x(2 layers convolutionnels puis un layer de max pooling avec dropout) puis un layer dense avec régularisation l2 et un dernier dropout. Les différents layers de convolution fonctionnent ici comme des tenseurs 4D puisque les données en input possèdent 3 dimensions: chaque input est composé de 30 images 2D (64 pixels x 64) se suivant (donc le temps est la troisième dimension). En particulier, le premier layer convolutionnel prend des données de dimension (30, 64, 64) en entrée, tandis que le dernier layers dense prend les valeurs des 1024 différents neurones du layer précedant en entrée pour déterminer le volume final. La fonction d'optimisation choisie est la fonction Adam avec un taux d'apprentissage de $10\hat{(}$-4), la RMSE nous sert de fonction de perte, et la fonction d'activation des différents layers est la même pour tous, ici on se sert de la fonction Rectified Linear Unit (ReLU). L'ensemble est résumé en Figure 1 dans le code à l'origine de notre modèle. La Figure 1 est sans doute plus compréhensible aussi, grâce à la simplicité d'écriture des modèles de Deep Learning du package Keras.

4 Comparaison des performances CPU/GPU

Un des avantages de Theano est qu'il peut tourner indifféremment sur CPU ou GPU et qu'il peut générer du code C et/ou Cuda lors de l'execution de certains script ce qui lui permet d'obtenir des performances étonnamment bonnes pour un package python. Il est généralement admis que (bien que cette valeur dépende

Figure 1: Le code qui crée notre modèle

```
model = Sequential()
model.add(Activation(activation=center_normalize, input_shape=(30, 64, 64)))

model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(Convolution2D(64, 3, 3, border_mode='valid'))
model.add(Activation('relu'))
model.add(ZeroPadding2D(padding=(1, 1)))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.25))

model.add(Convolution2D(96, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(Convolution2D(96, 3, 3, border_mode='valid'))
model.add(Activation('relu'))
model.add(ZeroPadding2D(padding=(1, 1)))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.25))

model.add(Convolution2D(128, 2, 2, border_mode='same'))
model.add(Activation('relu'))
model.add(Convolution2D(128, 2, 2, border_mode='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(1024, W_regularizer=l2(1e-3)))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))

adam = Adam(lr=0.0001)
model.compile(optimizer=adam, loss='rmse')
return model
```

fortement du problème étudié) le gain de performance lors du passage du CPU au GPU pour le traitement de calculs très distribués pse situe aux alentours de x40. Nous avons fait tourner notre modèle sur l'ensemble des 5331 groupes d'images successivement sur CPU puis sur GPU avec le matériel suivant:

1. CPU utilisé: Intel Core i7-4700HQ @ 2.4
2. GPU utilisé: GeForce GTX 870M

Pour forcer l'utilisation du CPU ou du GPU, il suffit de changer la variable d'environnement "THEANO_FLAGS" entre "device=gpu" et "device=cpu". Les résultats sont affichés dans le tableau plus bas. Comme attendu, le gain en temps d'execution est conséquent (quasiment x37) sans entrainer de changement au niveau de la performance de notre modèle.

1 epoch	CPU	GPU
temps	4155 s	123 s
RMSE	34.05	33.62
nombre de coeurs	4	1344
fréquence	2.4 GHz	0.9 GHz

On voit donc bien que la RMSE obtenue est sensiblement le même (la différence est due aux conditions initiales) pour CPU et GPU. Nous avons également calculé le CRPS (car c'est la métrique utilisée par les organisteurs de cette

compétition Kaggle) sur notre base d'entrainement (scindé en une base d'entrainement de 4265 éléments et 1066 données de validation), on obtient les résultats suivants:

	Train	Test
CRPS	0.071	0.074

Le CRPS (pour Continuous Ranked Probability Score) est calculé à partir de distributions cumulées de probabilité et nous a donc demandé d'abord de calculer ces distributions à partir des valeurs en sortie du neural network et d'un paramètre représentant l'incertitude du modèle (ici on utilise directement la RMSE).
Une fois les fonctions de répartitions déterminées, le CRPS se calcule selon la formule: $CRPS = \frac{||X_{train} - X_{predict}||_2^2}{N_{sample}}$
Sur nos simulations le CRPS diminue bien avec le nombre d'itérations de l'algorithme. Nous arrivons en 324 ème place/430 sur le classement du challenge Kaggle. Il existe de nombreuses pistes d'améliorations pour notre modèle (nombre de layers, type de layer...).

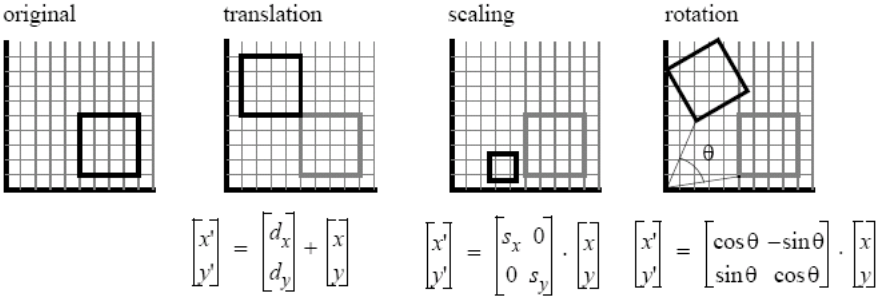
Figure 2: Classement Kaggle

320	↑56	Danny Malter	0.064896	2	Mon, 01 Feb 2016 05:06:14
321	↓29	AndrewPetrovics	0.071004	2	Wed, 23 Dec 2015 19:45:27
322	new	hassiktir	0.077681	2	Thu, 28 Jan 2016 23:14:10 (-2.1d)
323	↓30	⌐╦╦═(▪8ↄ-◦	0.079286	1	Tue, 05 Jan 2016 19:04:34
324	new	**ML Senpaiz**	**0.090272**	**2**	**Mon, 01 Feb 2016 16:48:35 (-2h)**
325	↓31	Bryan Johnson	0.090367	1	Fri, 08 Jan 2016 04:12:57
326	new	Darth Dave Diode	0.091475	1	Sat, 30 Jan 2016 03:46:39
327	new	Artem Yankov	0.092512	1	Wed, 27 Jan 2016 07:09:50
328	↓32	drumrick	0.093163	1	Mon, 14 Dec 2015 20:26:31
329	↓32	Jon Carlisle	0.094974	1	Thu, 21 Jan 2016 15:15:58
330	↑38	Erli Zhou	0.095207	3	Thu, 28 Jan 2016 20:07:48
331	↑41	Max	0.119942	3	Mon, 01 Feb 2016 15:04:12

5 Analyse de l'accelération produite par l'utilisation du GPU

Comme décrit plus haut, il existe une différence fondamentale entre la composition, en terme de hardware, des CPU et des GPU. Si ces seconds se comportent presque comme des CPU, ils se distinguent par leur très grand nombre de coeurs (qui sont généralement individuellement beaucoup moins puissants aussi). Dans notre cas, par exemple, le GPU utilisé possède 336 fois plus de coeurs que le CPU utilisé. Cette caractéristique permet aux GPU d'obtenir de bonnes performances lors du traitement de signaux (et en particulier d'images). En effet, les images,

Figure 3: Exemples de transformations



vidéos et par extension les signaux quelconques, sont généralement représentés en informatique par des matrices. Par conséquent, réaliser une opération de traitement de ces signaux revient le plus souvent à appliquer une transformation mathématique à ces matrices. Ces transformations matricielles requièrent d'effectuer de nombreux calculs "simples" qui sont le plus souvent indépendants les uns des autres et peuvent donc être distribués sur le grand nombre de coeurs du GPU.

De nombreuses opérations classiques lors de l'entrainement de modèles de Deep Learning ne sont en fait que des calculs matriciels et peuvent donc être distribuées efficacement sur les nombreux coeurs d'un GPU pour obtenir de meilleurs performances que lors de l'utilisation d'un CPU. C'est comme cela que Theano permet d'obtenir une accelération lorsqu'il a été configuré pour utiliser le GPU : le package envoie les opérations matricielles (et certaines autres opérations qui profitent aussi du grand nombre de coeurs) au GPU, et les autres opérations au CPU. En particulier, il est indiqué sur le site internet officiel de Theano que les multiplications de matrices, les convolutions ainsi que les grosses opérations élement-par-élement peuvent être accelérées entre x5 et x50 lorsque les arguments sont suffisamment gros pour tenir 30 processeurs occupés.
Cependant, certaines limites existent. On peut déjà noter que seuls les calculs avec des types de données *float32* ne peuvent être accelérés. Certaines opérations comme l'indexation ou le mélange de dimensions sont, elles, aussi rapides d'exécution sur CPU que sur GPU.

6 Conclusion

Ce projet nous a permis de nous familiariser avec l'utilisation du GPU pour la distribution de calculs. Dans notre cas (calculs sur réseaux de neurones) l'algorithme se terminait 37 fois plus vite sur GPU que sur CPU. Ce projet nous a également permit de nous familiariser avec la librairie de deep learning Keras.

7 Annexes

Figure 4: Exemple d'entrainement du modèle sur CPU

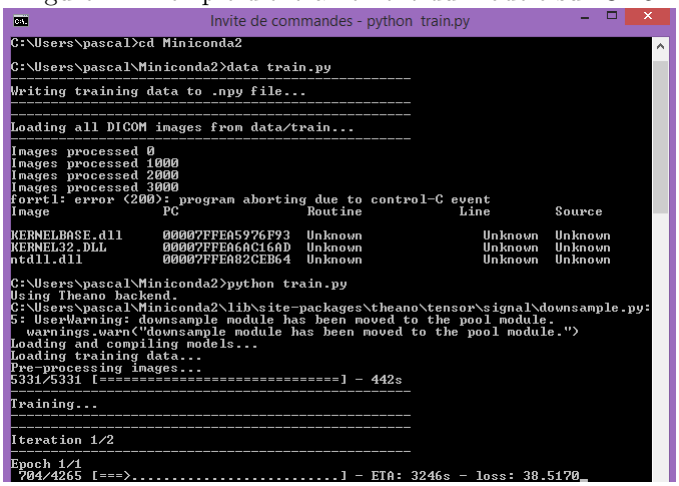


Figure 5: Exemple d'entrainement du modèle sur GPU