TP Deep Learning

Julien Mille
INSA Centre Val de Loire - Département GSI, 5A, option ACAD
Université de Tours - Master BDMA
Laboratoire d'Informatique Fondamentale et Appliquée de Tours (LIFAT)

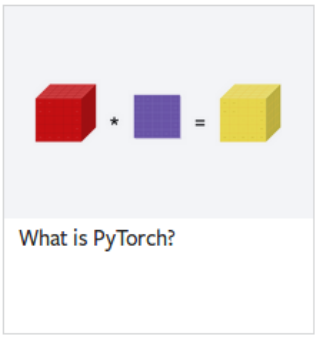
1 Installation

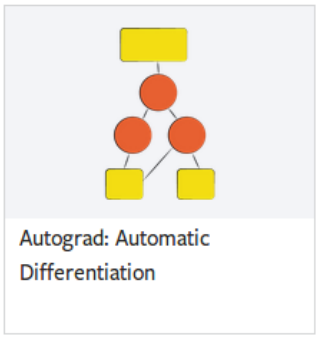
Les packages `opencv`, `pytorch` et `torchvision` doivent être installés dans votre environnement de développement Python. Si par exemple vous utilisez Anaconda, entrez les commandes suivantes :

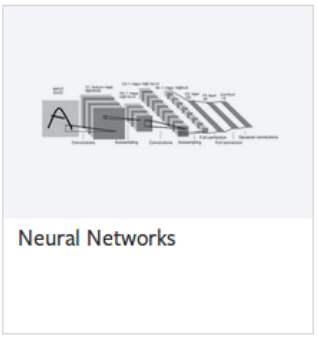
```
conda install opencv -c anaconda
conda install pytorch-cpu -c pytorch
pip install torchvision
```

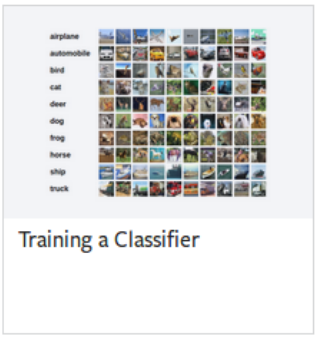
2 Tutoriaux

Allez sur `https://pytorch.org/tutorials/`. Allez dans *Getting Started* > *Deep Learning with PyTorch : A 60 Minute Blitz*. Vous allez suivre les 4 premiers tutoriels :









Ils vont vous permettre de vous familiariser avec les éléments de base de PyTorch (comme les `Tensor`s, les `Module`s, la différentiation automatique, les fonctions de *loss* et les algorithmes d'optimisation).

Attention, la version Windows de PyTorch ne supporte pas encore le *multithread*. Dans le code du tutoriel n°4, aux lignes 73 et 78, remplacez `num_workers=2` par `num_workers=0`. A la première exécution, le tutoriel 4 télécharge la base CIFAR10 (`https://www.cs.toronto.edu/ kriz/cifar.html`), ce qui nécessite un certain temps.

Si votre machine dispose d'un GPU compatible CUDA et que vous souhaitez en tirer parti, vous devez indiquer à PyTorch d'utiliser CUDA : les tenseurs et les couches du réseau seront stockés dans la mémoire GPU. Les calculs seront parallélisés au maximum. Pour ce faire, utilisez la fonction `cuda()` lors de l'initialisation du réseau,

```
net = Net().cuda()
```

et lorsque les images/labels sont chargés,

```
inputs, labels = data[0].cuda(), data[1].cuda()
```

Si le GPU est effectivement utilisé, vous devez observer un gain de vitesse assez net par rapport à la version CPU.

Pendant ces tutoriels, les documentations des bibliothèques sont à consulter sans modération :
- `https://pytorch.org/docs`
- `https://docs.scipy.org/doc`
- `https://matplotlib.org/contents.html`

En particulier, vous devez apprendre à utiliser les classes représentant les briques de base pour construire un réseau, dérivées de `torch.nn.Module`,
- `torch.nn.Linear` : couche dense (ou *fully-connected*),
- `torch.nn.Conv2d` : couche convolutive,
- `torch.nn.MaxPool2d` : couche de *max-pooling*,
- `torch.nn.ReLu`, `torch.nn.Tanh`, `torch.nn.Sigmoid` : couches d'activations non-linéaires,

les classes représentant les fonctions de *loss*, dérivées de `torch.nn._Loss`,
- `torch.nn.L1Loss`,
- `torch.nn.CrossEntropyLoss`,
- `torch.nn.NLLLoss`,

et les classes représentant les algorithmes d'optimisation, dérivées de `torch.optim.Optimizer`, notamment `torch.optim.SGD`.

3 Classification sur ImageNet

Le but des exercices est de réaliser une classification d'images provenant de la base ImageNET (`http://www.image-net.org/`), en vous inspirant de l'architecture du tutoriel n°4. Vous allez travailler avec un sous-ensemble de la base tiny-ImageNET [1], qui est elle-même une petite portion d'ImageNET.
La base comporte 20 classes. Chaque classe contient 500 images RGB 64×64. Les fichiers source fournis contiennent 2 classes dérivées de `torch.utils.data.Dataset`, qui permettent de parcourir les ensembles d'apprentissage et de test. Pour chaque classe, les 450 premiers exemples sont utilisés pour l'apprentissage, les 50 restants pour les tests.

Comme dans le tutoriel n°4, vous allez construire un réseau composé de plusieurs "séquences convolutives" suivies de plusieurs couches denses. On rappelle qu'une séquence convolutive est composée d'une couche de convolution, d'une fonction d'activation non-linéaire et d'un *max-pooling*.

3.1 Exercice 1

Tout d'abord, essayez avec 2 séquences convolutives (avec des activations ReLU) et de 2 couches denses. Testez avec des tailles de filtres (paramètre `kernel_size`) relativement petites, dans les couches de convolution (3×3, 5×5, 7×7). Choisissez le nombre de *feature maps* par couche (paramètres `in_channels` et `out_channels`), et le nombre de neurones dans la 1ère couche dense.

Utilisez la *cross entropy* comme *loss*, et la descente de gradient stochastique (SGD) pour l'optimisation. Affichez la fonction de coût au fur et à mesure des *epochs*. Sur l'ensemble de test, affichez la valeur de la fonction de coût et générez la matrice de confusion.

3.2 Exercice 2

Relancez l'apprentissage et le test, après avoir intercalé 1 ou 2 séquence(s) convolutive(s) supplémentaire(s), le but étant d'améliorer les résultats obtenus précédemment. Répétez cette opération tant que l'expérience est concluante !

1. La base tiny-ImageNET est disponible ici : `https://tiny-imagenet.herokuapp.com`

3.3 Exercice 3

Revenez à une architecture à 2 séquences convolutives et 2 couches denses. Vous allez automatiser la recherche des meilleurs hyperparamètres, en utilisant un ensemble de validation.

Modifiez la gestion du dataset de manière à avoir 3 ensembles : dans chaque classe, les 400 premiers exemples seront utilisés pour l'apprentissage, les 50 suivants pour la **validation** et les 50 derniers pour le test. Pour cela, vous devez comprendre le fonctionnement de `__getitem__` dans les classes `TinyImageNetDatasetTrain` et `TinyImageNetDatasetTest`. Modifiez ces classes et ajoutez une classe `TinyImageNetDatasetValidation`.

Choisissez l'ensemble des hyperparamètres (taille des filtres et nombre de *feature maps* des couches convolutives, nombre de neurones dans la 1ère couche dense) et les intervalles dans lesquels vous allez les faire varier. Pour chaque configuration, entrainez le réseau sur l'ensemble d'apprentissage, testez-le sur l'ensemble de validation, et retenez la configuration donnant le meilleur résultat. La configuration donnant le meilleur taux de reconnaissance sur l'ensemble de validation est celle qui sera conservée en phase de test.