INSA - ModIA, 4^e année, Pierre Weiss.

Segmentation d'images par réseaux de neurones

1 Préliminaires

Objectifs pédagogiques Ce TP a plusieurs ambitions :
- Introduire le langage Pytorch.
- Définir un réseau de neurones simple.
- Définir une procédure d'entraînement.
- Comparer différents algorithmes d'optimisation pour une tâche complexe.
- Renforcer vos capacités de programmation.

Outils d'édition Je n'ai pas édité de Jupyter Notebook pour ce TP. Bien que je les trouve très pratiques pour la réalisation de TP et de compte-rendus, je préfère nettement travailler sous des environnements de développement de type Spyder, Visual studio code ou PyCharm. Ceux-ci deviennent nécessaires pour du développement "professionnel" en entreprise. N'hésitez donc pas à changer vos habitudes et à les essayer pour ce TP et surtout votre vie en entreprise.

Objectif pratique On définit une image en niveaux de gris u comme une fonction de $\Omega = \{1, \dots, n\}$ dans $\mathbb{R}$. L'espace Ω peut être interprété comme une discrétisation de l'espace $[0,1]^2$. Dans ce TP, on suppose qu'on peut le partitionner en deux domaines réguliers Ω_1 et $\Omega_2 = \Omega \backslash \Omega_1$. Sur chacun des domaines, on suppose que l'image est constituée d'une texture différente. Dans les codes, on va générer les textures comme des processus gaussiens stationnaires. On peut les obtenir en convolant un bruit blanc gaussien, avec un filtre qui décrit la matrice de covariance du processus. La figure 1 montre un exemple d'image générée au hasard suivant ce modèle.

L'objectif est le suivant :

Etant donnée une image u (à gauche), construire un réseau capable d'identifier l'ensemble ω_1 (en jaune, à droite).

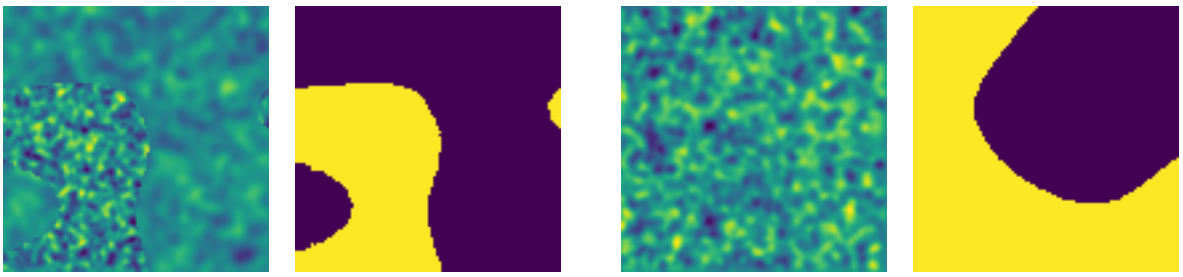

FIGURE 1 – Exemples d'images générées aléatoirement (gauche) et de la partition associée (droite) pour deux statistiques de processus différentes.

2 Quelques notions de Pytorch

Pourquoi Pytorch ? Actuellement, deux langages dominent le paysage de l'intelligence artificielle par réseaux de neurones : TensorFlow (Google) et Pytorch (Facebook). Pytorch semble actuellement préféré au niveau académique, car – au départ – il était plus flexible, en permettant de mieux générer ses propres modèles. Aujourd'hui, la flexibilité ne semble plus forcément meilleure, mais pourquoi changer ses habitudes lorsque Pytorch offre l'essentiel des fonctionnalités nécessaires ? Il est tout à fait possible

que l'un des deux langages finisse par dominer l'autre, mais pour le moment, ils co-existent tous les deux. Restez à l'affût des nouveautés de l'un et de l'autre...

Les tenseurs En Pytorch, le type de base essentiel est le *Tenseur*. C'est un tableau multidimensionnel dont la taille est toujours :

$$\text{nbatch} \times \text{nchannels} \times n_1 \times n_2 \times \ldots \times n_d. \tag{1}$$

Ce type est réellement pensé pour l'optimisation du calcul sur architecture parallèle et les descentes de gradient stochastiques :

- nbatch : nombre d'éléments dans le batch. Ceci permet d'appliquer simultanément un prédicteur sur nbatch éléments et de paralléliser le calcul. Note : il faut notamment adapter la taille du batch à votre capacité mémoire.
- nchannels : nombre de canaux. Historiquement, les modèles ont été utilisés pour le traitement des images. Le nombre de canaux représente le nombre de couleurs, e.g. 1 pour une image en niveaux de gris, 3 pour une image en couleurs.
- $n_1 \times n_2 \times \ldots \times n_d$: nombre d'élements dans chacune des d dimensions du tenseurs. Pour un signal 1D (audio), on a $d = 1$, pour une image 2D, on a $d = 2$, une image 3D ou une séquence vidéo $2D + t$, $d = 3$, une séquence d'images 3D, $d = 4$ et ainsi de suite...

Calcul CPU/GPU Une grande force de Pytorch est sa capacité à exploiter différentes architectures matérielles CPU/GPU sans difficulté importante. Il y a encore quelques années, il était très difficile de coder efficacement sur carte GPU dans le langage CUDA développé par NVidia. Aujourd'hui, il semble nettement préférable d'acheter une carte NVidia pour effectuer du calcul parallèle (recommandation pour votre futur ordinateur). Le développement de langages plus ouverts tels que OpenCL (permettant d'utiliser les cartes graphiques de constructeurs différents) semble en effet s'arrêter.

À retenir

La différence entre calcul CPU et GPU est abyssale : *un processeur CPU (e.g. Intel i7) typique a une puissance de 5GFlops, une carte graphique GPU grand public permet de réaliser 8TFlops.* Il y a donc un *facteur* > 1000 d'efficacité entre les deux, avec un prix équivalent !

Dans le début du code, vous trouverez les lignes suivantes. Elles permettent de détecter la présence ou non d'une architecture matérielle compatible avec le calcul CUDA et d'adapter le type des variables.

```
use_cuda=torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
if use_cuda :
    dtype = torch.cuda.FloatTensor
else:
    dtype = torch.FloatTensor
print("GPU: ", use_cuda)
```

Lors de la génération de tenseurs, on prendra garde à bien spécifier leur type avec des lignes du type

```
xs = torch.linspace(-size_image/2, size_image/2, size_image).type(dtype)
```

Notez ici, que la syntaxe Pytorch est très similaire à celle utilisée par Numpy. De fait, il existe des correspondances quasi directes entre la majorité des fonctions des deux langages (e.g. np.exp VS torch.exp, np.linspace VS torch.linspace, ...)

Installation Pour ce TP, vous pourrez travailler sur une machine distante. Par exemple, vous pouvez essayer :

- Google Colab
- Deep Note
- Plusieurs autres plateformes
- Si vous y avez accès, le JupyterHub du CNRS

Si vous avez une carte graphique Nvidia, si souhaitez travailler localement (ce qui a priori est préférable) et que vous n'avez encore jamais installé Pytorch, je vous recommande les étapes suivantes :

— Si vous avez une carte Nvidia, installez les bons drivers (a priori c'est déjà le cas).

— Installer Anaconda.

— Générer un environnement pour le TP et l'activer. Sous Linux, il suffit d'utiliser un terminal. Sous Windows, il faudra utiliser Anaconda prompt (je ne connais pas). Sous Mac, je ne sais même pas si tout ceci est possible car Mac a décidé de créer ses propres cartes graphiques.

```
conda create --name TP2
conda activate TP2
```

— Installer les packages nécessaires :

```
conda install numpy
conda install -c conda-forge matplotlib
```

— Installer Pytorch. Configurez bien l'installeur en fonction de votre installation. Par exemple, pour ma machine sous Linux, j'obtiens la ligne de commande du type :

```
conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch
```

3 Génération de la base de données d'images

1. Observez et essayez de comprendre la fonction de génération d'images.

```
def generate_data(batch_size, size_image, sigma1, sigma2, sigma)
```

2. Changez les paramètres sigma1, sigma2, sigma pour comprendre leur rôle respectif et observez la diversité des types d'images qu'on peut générer.
3. Pour quelles valeurs de sigma1 et sigma2 est-ce que le problème d'identification semble difficile ?

4 Construction d'un réseau de neurones

1. Observez et essayez de comprendre la fonction définissant le réseau de neurones.

```
class my_first_CNN(nn.Module)
```

2. Vous pouvez notamment exécuter les lignes suivantes pour mieux comprendre l'architecture resultante.

```
predictor = my_first_CNN(num_channels=16,bias=True).type(dtype)
predictor.to(device)
print(predictor)
```

3. Combien de couches possède le réseau ? Quelles sont les fonctions d'activation ?
4. Quel peut être l'intérêt d'ajouter *torch.sigmoid* à la fin ?

5 Définition de l'algorithme de minimisation

Le choix d'un algorithme d'optimisation (standard) en Pytorch est très simple. En effet, le module *torch.optim* possède déjà de nombreux solveurs (SGD, Adam, RMSProp,...). Pour utiliser l'une ou l'autre méthode, il suffit d'écrire des lignes du genre :

```
optimizer = optim.Adam(predictor.parameters(), lr = learning_rate, betas=(0.9,0.999),
    eps=1e-8)
optimizer = optim.SGD(predictor.parameters(), lr = learning_rate, momentum = 0.9)
```

1. Allez voir la documentation pour comprendre comment définir un algorithme d'optimisation.
2. Analysez le code qui suit :

```
while i<niter_train:
    i+=1
    x, omega1 = generate_data(batch_size, size_image, sigma1, sigma2, sigma)
    optimizer.zero_grad()
    prediction = predictor(x)
    loss = torch.sum((prediction - omega1.type(dtype))**2)/batch_size
    loss.backward()
    optimizer.step()

```

3. Expliquez ce que fait chaque ligne.
4. Quelle est la fonction perte utilisée ici ? Si vous avez des idées, n'hésitez pas à la modifier.

6 Entraînement et choix d'algorithme

Il est temps d'essayer d'entraîner notre réseau. Pour ce faire, il faut choisir les différents paramètres. Les plus importants sont :

— sigma1, sigma2, sigma : définissent la base de données et encodent la difficulté de la tâche de segmentation.
— optimizer : le choix d'un algorithme d'optimisation et de ses paramètres (momentum, beta, ...)
— batch_size et learning_rate : caractérisent l'algorithme d'optimisation.
— size_image : suivant votre architecture matérielle, vous pourrez considérer des images plus ou moins grandes. Pour ce TP, on peut se contenter de petits domaines.

1. Essayez plusieurs algorithmes. Typiquement SGD avec et sans inertie, RMSProp, Adam. Comparez leur performance lorsque le learning rate est bien choisi.
2. Que se passe-t'il si le learning rate est trop grand pour ces algorithmes ?
3. Pour un temps de calcul donné (disons 2 minutes), essayez d'obtenir le meilleur prédicteur possible. Quel choix vous semble le meilleur ?