Apprentissage par réseaux de neurones profonds
Date de remise : 10 avril 2020 à 23h59
Version 1.0

Travail pratique TP2, en équipe de 1 à 2.

20 mars 2020

Pour tous les étudiants, vous devez fournir un rapport en un seul document (format pdf). **Attention ! N'oubliez pas d'attacher le code de toutes les questions dans la remise .zip du travail. Si le code est manquant, nous pourrons retirer jusqu'à 40% du total.** Aussi, assurez-vous que votre rapport ne soit pas trop volumineux en terme de données : utilisez une résolution raisonnable des images pour que la soumission soit en bas de 15 Mo, idéalement.

Instructions spéciales pour les étudiants en GLO-7030 : il y a une question de plus à répondre. Aussi, veuillez noter qu'une présentation déficiente dans le rapport (manque de clarté, orthographe et grammaire, police de caractère illisible sur figure, etc) pourra entraîner une pénalité allant jusqu'à 10 % de la note. Le rapport doit aussi obligatoirement être formaté avec LaTex. Plusieurs site web offrent des outils gratuits pour faire l'édition de manière collaborative, notamment `overleaf`. Assurez-vous que vos documents ne soient pas publics (pastebin, github, etc).

Vous pouvez utiliser la librairie `deeplib` ou `pytoune` (https://pytoune.org/) vu dans les laboratoires pour faciliter l'entraînement de vos réseaux.

1 Data augmentation (35 pts)

Pour cette question, vous utiliserez le dataset MNIST provenant de la librairie `torchvision`[1]. Dans la documentation, prenez note du paramètre **transform**. C'est ce paramètre qui servira à ajouter du *data augmentation*. Plusieurs transformations d'images sont déjà implémentées pour vous. Voir http://pytorch.org/docs/master/torchvision/transforms.html.

a) Dans le fichier `q1.py`, implémentez la fonction `mnist_dataset()` qui retourne trois objets `Dataset` de MNIST : un dataset d'entraînement, un dataset de validation et un dataset de test. Le dataset de validation doit être un sous-ensemble du dataset d'entraînement selon le pourcentage en argument. Les datasets de validation et de test n'ont pas de data augmentation. Le dataset d'entraînement a le pipeline d'augmentation suivant (dans l'ordre) :
   (a) Padding de l'image avec des 0
   (b) Rotation de l'image ou cisaillement (sheer)
   (c) Crop aléatoire de la taille originale d'une image de MNIST (28x28)

b) Définissez un réseau de neurones ayant huit couches convolutionnelles et de la Batch Norm, selon une approche ResNet. N'oubliez-pas que les blocs résiduels ont une structure assez stricte, avec la skip connection pour deux couches convolutionnelles (voir laboratoire 5). Ajouter du max pooling juste après les couches 4 et 6. Le nombre de filtres est à votre choix. Entraînez ce réseau avec les paramètres de data augmentation suivants :
   — pad=2, rotation=45
   — pad=14, rotation=5
   — pad=2, rotation=180
   — pad= ? ? ?, sheer=15 degrés (défini dans `RandomAffine`)

   Dans la liste précédente, par exemple, pad=2 indique qu'on ajoute 2 pixels dans chaque direction (haut, bas, gauche, droite) et rotation=45 indique qu'on fait une rotation aléatoire de plus ou moins 45 degrés, pigé dans une distribution uniforme. Pour le cas `RandomAffine`, nous vous demandons d'identifer une bonne valeur de padding. Indiquez-la dans votre rapport.

   Pour chaque cas, présentez dans votre rapport un tableau des taux d'erreurs en entraînement, validation et en test selon les hyperparamètres de data augmentation. De plus, ajoutez des images dans votre rapports produites selon ces hyperparamètres en incluant l'étiquette de l'image (8 images par ensemble d'hyperparamètre). Donnez un brève description de ce que vous observez dans

1. http://pytorch.org/docs/master/torchvision/datasets.html#torchvision.datasets.MNIST

ces images. Y a-t-il des problèmes qui peuvent nuire à l'entraînement ? Attention : Les images doivent être représentatives des problèmes identifiés dans votre rapport, s'il y a lieu.

2 Fine-tuning et Normalisation (45 pts GLO-4030/30 pts GLO-7030)

Dans cette question, vous devez faire de la classification fine sur des espèces d'oiseaux. Pour ce faire, téléchargez les images du jeu de données 'CUB-200' [2].

Pour chaque classe, triez les images en ordre croissant selon le nom du fichier et utilisez les 15 premières images comme jeu de test. Utilisez les autres pour l'entraînement. Pour ce faire, vous pouvez utiliser la fonction fourni dans `q2-cub200.py`.

Par la suite, faites les entraînements suivants en utilisant le `ResNet18` fourni dans torchvision [3] :

a) En utilisant l'initialisation aléatoire par défaut.

b) En utilisant le modèle pré-entraîné, mais en *freezant* tous les paramètres de convolution.

c) En utilisant le modèle pré-entraîné, mais en *freezant* uniquement les paramètres dans `layer1`.

d) En utilisant le modèle pré-entraîné, mais en laissant tous les paramètres (incluant les couches de convolution) se faire ajuster par backprop.

Faites chaque entraînement deux fois. La première fois, utilisez les valeurs du dataset d'entraînement pour normaliser les données. La deuxième fois, utilisez les valeurs suivantes qui ont été utilisées pour l'entraînement sur ImageNet :

TABLE 1 – Coefficents de normalisation pour l'entraînement sur ImageNet.

	R	G	B
Moyenne	0,485	0,456	0,406
Écart-type	0,229	0,224	0,225

2. http://www.vision.caltech.edu/visipedia/CUB-200.html
3. http://pytorch.org/docs/master/torchvision/models.html

Voici quelques conseils et consignes supplémentaires :

1. **Pour les questions b) et c), *freezez* les paramètres de batch norm correspondant ainsi que les paramètres de 'conv1' et de 'bn1'.**
2. Pour réussir à entraîner le réseau, vous aurez à adapter la couche pleinement connectée.
3. Pour charger les images, vous pouvez utiliser la classe ImageFolder [4]
4. Vous devrez aussi vous assurer que toutes les images ont la même taille initiale. Pour ce faire, redimensionnez les images en (224x224) [5].
5. Pour la normalisation, vous pouvez utiliser la transformation Normalize [6].

Remettez votre code. Remplissez un tableau permettant de comparer la précision sur le jeu d'entraînement et la précision sur le jeu de test de chaque entraînement.

Discuter des différences de résultats selon la normalisation utilisée en entrée (basée sur le tableau 1 ou sur les statistiques des données de CUB-200). Discutez aussi de l'effet du pré-entraînement et du *freeze* des couches de convolution.

3 RNN (35 pts) GLO-7030 seulement

Pour ce numéro, vous devez utiliser une architecture récurrente de type RNN pour effectuer de la classification textuelle. Le jeu de donnée est une liste de mots auxquels sont associés une langue :

```
chat -> fr
cat -> en
gato -> es
...
```

Pour vous aider, nous vous fournissons du code de base dans le fichier `q3-RNN.py`.

La gestion des séquences à longueur variable sera une composante importante de cette question. En effet, le batching sur carte graphique implique que le réseau s'attend à avoir une matrice en entrée pour être en mesure de faire la forward/backward pass. Cette matrice exige d'avoir des entrées de tailles égales. Le modèle de base inclut dans le code gère ceci de manière très naïve : on créer des séquences ayant toutes les mêmes longueurs, soit la longueur maximale dans notre jeu d'entraînement. En utilisant un symbole spécial (souvent appelé le "padding token"), on peut donc transformer toutes les séquences pour qu'elles aient la

4. http://pytorch.org/docs/master/torchvision/datasets.html#imagefolder
5. http://pytorch.org/docs/master/torchvision/transforms.html#torchvision.transforms.Resize
6. http://pytorch.org/docs/master/torchvision/transforms.html#torchvision.transforms.Normalize

même longueur. Par exemple, si la longueur maximale dans le jeu d'entraînement est 10 et le "padding token" est "`@`", les mots seront transformés de la sorte :

```
chat@@@@@@
cat@@@@@@@
gato@@@@@@
```

Pour les sous-questions de 3.1 à 3.3, seul le code suffit. Pour la sous-question 3.4, veuillez répondre dans le rapport.

3.1 Architecture initiale avec la classe `WordClassifier`

Créez une architecture pour classifier les différents mots du jeu de données en complétant la classe `WordClassifier`. Pour vous aider, nous vous suggérons fortement d'utiliser la couche `Embedding` initialisée dans votre architecture.

3.2 Ajout de la fonction `collate_examples`

La méthode naïve de padding est inefficace en mémoire. Bien souvent, on veut faire ce qu'on appel du "padding on batch". Complétez la fonction `collate_examples` pour arriver à faire du "padding on batch".

3.3 Ajout de la fonction `WordClassifierHandlingPadding`

Les architectures récurrentes ne prennent pas en compte automatiquement le padding ajouté à nos exemples. À l'aide de la fonction `pack_padded_sequence` de PyTorch, complétez l'architecture `WordClassifierHandlingPadding` pour que votre modèle gère convenablement le padding dans vos exemples.

3.4 Comparaison entre les trois approches

Pour chacune des trois approches précédentes, comparez les temps de convergence et de généralisation par rapport aux différentes architectures/méthodes de gestion des données. Avec une bonne gestion des données, vous devriez atteindre au minimum 50% d'exactitude sur l'ensemble de test.

À titre indicatif : vous pouvez vous inspirer de ce tutoriel pour comprendre la manière de gérer les séquences de longueur variable avec un RNN : https://github.com/ngarneau/understanding-pytorch-batching-lstm. Notez que ce code n'est pas à jour avec la dernière version de PyTorch.