Reconnaissance de plaque d'immatriculation

TP4 : Réseau de neurones et apprentissage profond

L'objectif de ce TP est de concevoir et d'entraîner un réseau de neurones afin qu'il apprenne à reconnaître par classification les caractères d'une plaque d'immatriculation. Contrairement aux deux premiers TPs, les descripteurs ne seront pas "hand-crafted" mais automatiquement créés par le réseau via apprentissage profond.

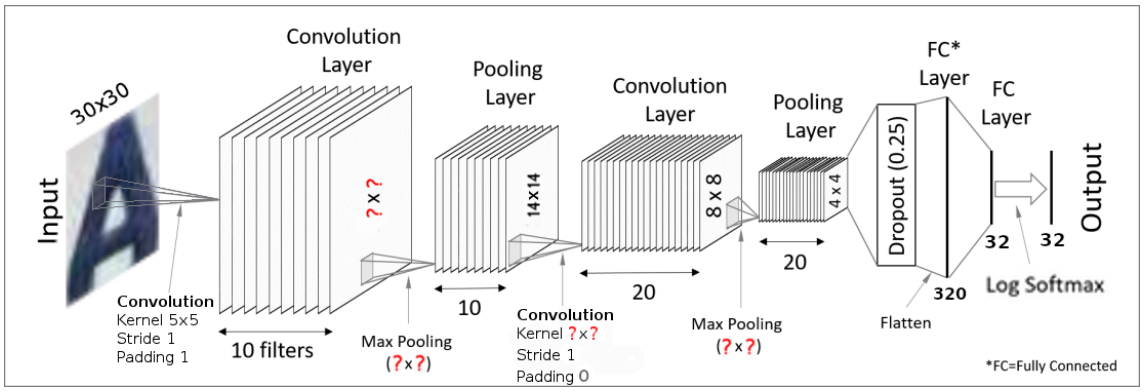

Figure 1: Exemple d'architecture d'un réseau de neurones convolutif simple

I. Jeux de données et entraînement

Dans le TP précédent, nous avons tout d'abord découvert l'élément de base de PyTorch. En effet, la représentation des données d'un réseau de neurones sous PyTorch se fait par l'intermédiaire d'un référentiel de données appelé tensor. Un tensor est un conteneur qui généralise les matrices à l'espace N-dimensionnel.
Dans un second temps, nous avons examiné le paquet autograd de PyTorch qui permet de calculer automatiquement les dérivés d'une suite d'opérations sur un tensor. L'un des principaux concepts des réseaux de neurones étant la rétropropagation (processus de mise à jour des poids d'un modèle), le paquet autograd est par conséquent central à tout réseau de neurones PyTorch.
Enfin, nous avons appris à définir un réseau de neurones convolutifs à l'aide du paquet nn. Ce dernier généralise les blocs de construction courants (couche de convolution, pooling, fonctions d'activation, etc.) sous forme de fonctions réutilisables. Un réseau est habituellement créée en étendant la classe nn.Module dans laquelle nous définissons les couches (fonction init) ainsi que la façon dont elles sont reliées (fonction forward).

Pour aller plus loin et être en mesure d'utiliser un réseau de neurones convolutifs, il nous reste deux points essentiels à traiter : les données d'entrées et l'exploitation de ces données pour entraîner notre réseau.

I.I Datasets

I.I.1 Chargement de notre propre jeu de données

Dans cette partie, nous allons exploiter le réseau de neurones créée précédemment (Figure 1) afin de classifier les caractères des plaques d'immatriculation.

Exercice : Compléter le code suivant dans un fichier Python *dataset_tools.py* afin de télécharger et charger les données d'entraînement et de test du jeu de données.

```
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

preprocess = transforms.Compose([
    transforms.Resize([30, 30]),
    transforms.ToTensor()])


    # Chargement des donnees
    data_dir = "[pathto]/dataset"
    image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
     data_transforms[x]) for x in ['train', 'test']}
    dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x],
    batch_size=1, shuffle=True, num_workers=4) for x in ['train', 'test']}
    dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'test']}
    class_names = image_datasets['train'].classes
```

Il est ensuite possible de parcourir les données chargées à l'aide d'une simple boucle `for` :

```
    for batch_idx, (data, target) in enumerate(train_loader):
        [...]
```

Exercice : Parcourir les données d'entraînement et expliquer ce que contiennent les variables `batch_idx`, `data` et `target`.

Torchvision propose également des fonctions pratiques pour visualiser les données. Par exemple, il est possible de créer une grille d'un batch complet d'images à l'aide de la fonction :
`torchvision.utils.make_grid(data)`.

Exercice : À l'aide de la fonction `imshow()` donnée ci-dessous et de la fonction `torchvision.utils.make_grid()`, visualiser les premiers batch de données de votre base d'entraînement.

```
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

I.II Entraînement du modèle

Voici les étapes d'une procédure classique d'apprentissage d'un réseau neuronal :

ÉTAPE 1 Définir un réseau neuronal qui possède des paramètres à apprendre (poids)

ÉTAPE 2 Itérer sur l'ensemble des données (ou batch) de la base d'apprentissage et de test

ÉTAPE 3 Alimenter le réseau et calculer la sortie

ÉTAPE 4 Calculez l'erreur (Loss), c'est-à-dire dans quelle mesure la sortie calculée diffère de la cible

ÉTAPE 5 Rétropropager les gradients à travers le réseau

ÉTAPE 6 Mettre à jour les poids du réseau

Nous avons vu comment définir (ÉTAPE 1) et alimenter (ÉTAPE 3) un réseau, nous savons comment télécharger, charger et parcourir les données d'un dataset (ÉTAPE 2) et nous avons étudié la rétropropagation (ÉTAPE 5). Il nous reste donc à appréhender le calcul de l'erreur (ÉTAPE 4) et la mise à jour des poids du réseau (ÉTAPE 6).

I.II.1 Calcul de l'erreur

Une fonction Loss calcule une valeur qui exprime la distance entre une paire d'entrées : sortie du réseau et cible. Lors de la conception d'un réseau neuronal, vous avez le choix entre plusieurs fonctions d'erreur, certaines sont plus adaptées à certaines tâches. Dans cet exemple, nous utiliserons la Negative Log Likelihood (NLL). Dans PyTorch, la fonction NLL est intégrée dans `torch.nn.functional` et est appelée `nll_loss`. Son utilisation est simple puisqu'il suffit d'ajouter une ligne à notre boucle d'apprentissage avec pour arguments la sortie du réseau, ainsi que la cible :

```
loss = F.nll_loss(output, target)
```

I.II.2 Mise à jour des poids du réseau

Comme pour le calcul de l'erreur, il existe plusieurs solutions pour mettre à jour jour les poids du réseau (voir cours). La règle d'optimisation la plus simple et généralement la plus efficace est celle de la descente du gradient stochastique (SGD) qui suit :

$$poids = poids - taux\ d'apprentissage \times gradient \tag{1}$$

SGD ainsi que d'autres algorithmes d'optimisation telles que Nesterov-SGD, Adam, RMSProp et d'autres sont intégrées dans un petit paquet appelé `torch.optim`. Pour initialiser un algorithme d'optimisation, il faut lui fournir les paramètres du réseau qu'il doit optimiser et éventuellement lui spécifier des options spécifiques telles que le taux d'apprentissage. Par exemple, pour instancier un `optimizer` de type SGD devant mettre à jour les poids d'un réseau appelé `net` avec un taux d'apprentissage égale à 0.01 :

```
import torch.optim as optim

optimizer = optim.SGD(net.parameters(), lr=0.01)
```

Dans la boucle d'apprentissage, la mise à jour des poids du réseau en suivant la règle d'optimisation choisie s'effectue à l'appel de la fonction `step` de l' `optimizer` :

```
optimizer.step()
```

Il est à noter qu'il ne faut pas oublier de mettre à zéro les gradients (`optimizer.zero_grad()`) avant d'appeler la passe de rétropropagation (ÉTAPE 5).

Exercice : Vous disposez désormais de tous les éléments pour entraîner votre réseau. Compléter le code `training.py` fourni en annexe. Les différentes étapes de la procédure

d'apprentissage mentionnées ci-dessus ainsi que d'autres informations sont indiquées dans les commentaires pour vous aider. Lorsque votre code est complété, exécuter l'apprentissage de votre réseau et commenter les informations retournées.

À la fin de l'apprentissage, les poids de votre réseau sont sauvegardés dans un fichier :

```
torch.save(net, 'W_CNN.pt')
```

Vous pouvez charger ensuite ces poids avec la fonction :

```
net = torch.load('W_CNN.pt')
```

Exercice : Dans un nouveau fichier Python *inference.py*, écrire un code permettant :

- d'initialiser votre réseau avec les poids entraînés au cours de l'exercice précédent
- de tester votre réseau sur vos propres images d'entrées.