Reconnaissance de plaque d'immatriculation

TP 3 : Réseau de neurones et apprentissage profond

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

I. PyTorch

Le but de cette première partie est de se familiariser et de maîtriser les bases de la bibliothèque PyTorch.

PyTorch est une bibliothèque Python open source d'apprentissage profond (Deep Learning) principalement développée par le groupe de recherche en intelligence artificielle de Facebook.

PyTorch permet notamment :

- d'effectuer des calculs "matriciels" en exploitant pleinement la puissance des GPUs ce qui accélère considérablement le processus de formation d'un réseau de neurones,
- d'effectuer automatiquement la rétropropagation (backpropagation) à travers un réseau de neurones,
- de générer un réseau dynamique permettant au réseau d'évoluer pendant sa formation et de faciliter son débogage,
- de s'appuyer sur une communauté importante et active (ex. torchvision).

I..1 Tensors

Les Tensors de PyTorch sont similaires aux Arrays de NumPy. Cependant les Tensors peuvent aussi bien être utilisés sur un CPU que sur le GPU pour accélérer les calculs.

Exercice : Ajouter le fichier *tensors_intro.py* à votre projet. Ce fichier contient les indications à suivre pour prendre en main les Tensors de pyTorch (déclarations, opérations, indexations, mutations, etc.).

I..2 Autograd

L'un des principaux concepts du deep learning est la rétropropagation du gradient de l'erreur. Cette étape est primordiale au processus de mise à jour des poids dans le réseau de neurones. L'optimisation des poids du réseau est généralement réalisée en suivant un algorithme de descente du gradient de la fonction de coût que l'on cherche à minimiser. Par conséquent, pour "descendre" le long du gradient, nous avons besoin de déterminer ce gradient qui est le fruit d'une suite de dérivés.

PyTorch possède le module autograd qui permet de calculer automatiquement les dérivées de toutes les opérations effectuées sur des Tensors. Toutes les opérations effectuées sur un Tensor sont enregistrées, puis autograd est capable de les rejouer à l'envers pour calculer les gradients. Exemple :

```
x = torch.ones((2,2), requires_grad=True)
```

Notez le second argument qui indique que nous souhaitons enregistrer toutes les opérations réalisées sur cette variable dès sa création. Après avoir effectué des opérations sur le Tensor, le calcul de la rétropropagation est appelé avec :

```
o.backward()
```

Suite à la rétropropagation, les gradients sont accessibles avec :

```
x.grad
```

Exercice : Créer et ajouter un nouveau fichier Python *autograd_intro.py* à votre projet, puis:

- Déclarer et initialiser un Tensor $\mathbf{x}$ de taille 2×2
- Déclarer et initialiser un Tensor tel que $\mathbf{y} = \mathbf{x} + 5$
- Déclarer et initialiser un Tensor tel que $\mathbf{z} = (3 \times \mathbf{y}^2)/\mathbf{x}$
- Déclarer et initialiser un Tensor tel que $\mathbf{o} = mean(\mathbf{z})$
- Calculer la dérivée $\frac{\partial \mathbf{o}}{\partial \mathbf{x}}$ à la main
- Vérifier votre calcul de $\frac{\partial \mathbf{o}}{\partial \mathbf{x}}$ à l'aide d'autograd

I..3 Neural Network

Les couches : Même s'il est possible de construire un réseau de neurones entier en utilisant uniquement la classe Tensor de PyTorch, ce serait très fastidieux. De plus, la plupart des réseaux de neurones sont composés de blocs de construction (de couches) similaires. C'est pourquoi PyTorch propose le paquet nn (neural network) qui généralise ces couches sous forme de fonctions réutilisables. Voici une liste de quelques exemples non-exhaustive[1] :

- Convolution
  - Convolution 1D :
    ```
    torch.nn.Conv1d(in_channels, out_channels, kernel_size,
      stride=1, padding=0, dilation=1, groups=1, bias=True)
    ```
  - Convolution 2D :
    ```
    torch.nn.Conv2d(in_channels, out_channels, kernel_size,
      stride=1, padding=0, dilation=1, groups=1, bias=True)
    ```
- Pooling

[1] Une liste exhaustive est disponible ici : https://pytorch.org/docs/stable/nn.html

  - Max Pooling 2D :
    ```
    torch.nn.MaxPool2d(kernel_size, stride=None, padding=0,
        dilation=1, return_indices=False, ceil_mode=False)
    ```
  - Average Pooling 2D :
    ```
    torch.nn.AvgPool2d(kernel_size, stride=None, padding=0,
        ceil_mode=False, count_include_pad=True)
    ```

- Non-linear activation
  - Sigmoid
    ```
    torch.nn.Sigmoid
    ```
  - ReLU (Rectified Linear Init) :
    ```
    torch.nn.ReLU(inplace=False)
    ```

- Fully Connected
  - Transformation linéaire :
    ```
    torch.nn.Linear(in_features, out_features, bias=True)
    ```

- Loss Function
  - Erreur quadratique moyenne :
    ```
    torch.nn.MSELoss(size_average=None, reduce=None, )
      reduction='elementwise_mean')
    ```
  - Entropie croisée :
    ```
    torch.nn.CrossEntropyLoss(weight=None, size_average=None,
      ignore_index=-100, reduce=None, reduction='elementwise_mean')
    ```

D'une manière générale, les paramètres (poids et biais) d'une couche sont accessibles à partir des attributs `weight` et `bias` du block créée.

Exercice : Créer et ajouter un nouveau fichier Python *nn_intro1.py* à votre projet, puis :

- Créer un block de convolution 1D ayant en entrée 1 canal et 2 en sortie (afficher ses paramètres),
- Déclarer $\mathbf{x}$, un Tensor 1D d'un seul élément aléatoirement initialisé,
- Alimenter votre couche de convolution avec votre Tensor $\mathbf{x}$ et conserver la sortie obtenue dans un Tensor $\mathbf{y}$
- Vérifier manuellement le contenu de $\mathbf{y}$
- Déclarer $\mathbf{o}$, un Tensor 1D initialisé comme étant la somme des éléments de $\mathbf{y}$
- Calculer manuellement $\frac{\partial \mathbf{o}}{\partial \mathbf{x}}$
- Vérifier votre calcul de $\frac{\partial \mathbf{o}}{\partial \mathbf{x}}$ à l'aide d'autograd

Déclaration d'un réseau de neurones : La façon la plus pratique de définir un réseau avec PyTorch est de créer une nouvelle classe qui étend la classe de base torch.nn.Module. La classe Module fournit un moyen simple d'encapsuler les paramètres d'un réseau, et inclut des fonctions pratiques tel que le transfert des paramètres entre le CPU et le GPU.

Un réseau est généralement défini en deux parties. Tout d'abord, les blocs de construction qui composeront le réseau sont déclarés dans la fonction `__init__`. Un même bloc peut être réutilisé plusieurs fois dans la construction du réseau. Les tailles d'entrée et de sortie de chaque bloc doivent être correctement spécifiées. Dans un second temps, les liaisons entre les différents blocs du réseau sont implémentées dans la fonction `forward()`. Les connexions entre les blocs du réseau proviennent des éléments déclarer dans la fonction `__init__`, d'opérations de la classe Tensors ou encore de fonctions plus spécifiques provenant de la classe torch.nn.Functional.

```
import torch.nn as nn
import torch.nn.functional as F

class myNet(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # Declarer les block qui composeront votre reseau
        self.block1 = ...
        self.block2 = ...
        ...

    def forward(self, x):
        # Relier les blocks ensemble pour
        # former les couches de votre reseau
        x = self.block1(x)
        x = self.block2(x)
                ...
        return x
```

Exercice :

- Identifier les dimensions manquantes dans l'architecture du réseau de neurones illustrée par la Figure 1.
- Dans un nouveau fichier Python *nn_intro2.py*, implémenter ce réseau de neurones convolutif avec PyTorch.
- Instancier un nouveau réseau à partir de votre implémentation et afficher son architecture.

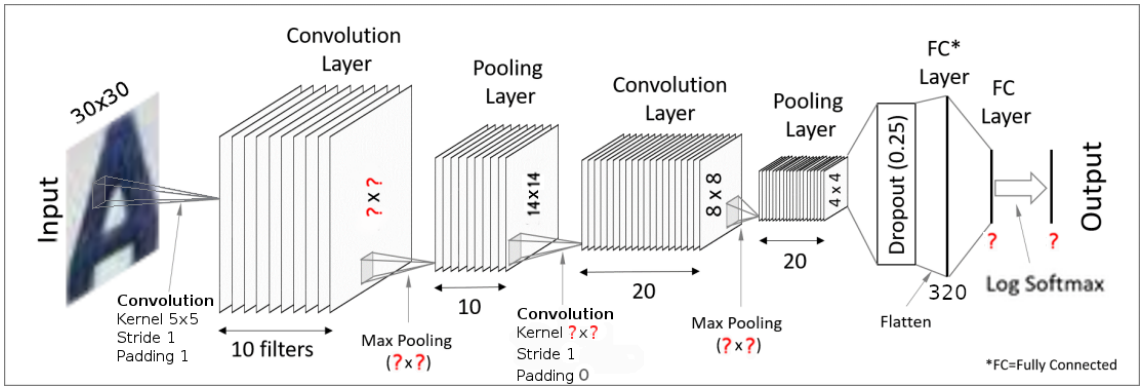

Figure 1: Exemple d'architecture d'un réseau de neurones convolutif simple