BORDEAUX INP Enseirb-Matmeca

Électronique 3ème année

Spécialité traitement du signal et de l'image

Documentation PyTorch

Auteur :
M. Romain LOSILLA

Encadrant :
M.Frédéric CHATRIE

Écrit le 09/01/2025

1 Introduction

Cette documentation contient la description des différentes fonctions de la bibliothèque PyTorch qui nous permettre de développer notre modèle d'IA dans le projet. On trouvera donc dans cette documentation l'exploration des commandes de base de cette bibliothèque et la description de la création d'un MLP et d'un CNN.

2 Commandes

Nous allons commencer par la description des différentes commandes utiles dans la création d'une IA. Mais avant de commencer, introduisons la notion de Tensor qui est fondamentale dans cet environnement. Un Tensor est un type d'objet qui possède le même comportement qu'un NumPy array à la différence qu'il peut être exécuté sur GPU et CPU. Ainsi, la bibliothèque torch comporte de nombreuses fonctions présentes dans NumPy. La fonction permettant d'utiliser un GPU (NVIDIA) est la suivante : torch.device("cuda")

2.1 Commandes liées à la manipulation de la base de données

Une fois que nous possédons une base de données, nous devons la manipuler pour se conformer aux attentes de notre réseau ou simplement pour la charger lors de la phase d'apprentissage. La première tâche que nous voulons exécuter est la création de la classe Dataset qui nous permettra d'effectuer trois tâches : l'initialisation de la classe, connaître le nombre de données de la classe et obtenir une donnée de la base de données à l'indice donné en argument. Ainsi, on doit créer trois méthodes :

- **__init__** : Cette méthode permet d'effectuer l'initialisation de la classe. On peut y effectuer de nombreuses manipulations, mais l'idée est de charger les chemins d'accès à chaque donnée et de lui associer son étiquette.
- **__len__** : Cette méthode permet de retourner le nombre de données présentes dans la base de données.
- **__getitem__** : Cette méthode prend en argument un indice et renvoie la donnée associée à l'indice dans la base de données ainsi que l'étiquette de la donnée.

Maintenant que nous pouvons créer la classe Dataset, nous pouvons créer la classe qui contiendra les données d'apprentissage de notre réseau. L'étape suivante est de créer un DataLoader, nous permettant de charger un nombre de données précisé par la variable mini-batch. Cette fonction est utile car elle permet de prendre en charge le chargement de données d'une façon abstraite et aléatoire afin de maximiser l'apprentissage de notre modèle. On l'utilise avec la commande :

$$torch.utils.data.DataLoader(nom\_classe\_Dataset, batch\_size, shuffle = True)$$

Elle prend en paramètre la classe créée précédemment, le nombre de données que l'on veut charger en même temps et si on veut que ces données soient prises aléatoirement. Il peut arriver que l'on utilise notre réseau avec des nouvelles données qui n'ont pas les mêmes propriétés que les précédentes. Ainsi, on peut appliquer des transformations aux nouvelles données pour qu'elles soient conformes à celles de l'apprentissage. On peut donc utiliser la classe transforms de torchvision afin d'effectuer nos différentes manipulations sur les données. Un exemple d'utilisation de transformation est pour effectuer un fine-tuning. On peut donc utiliser les fonctions suivantes :

- $torchvision.transforms.Resize(new_dim)$: Permet de redimensionner les données d'origine.
- $torchvision.transforms.ToTensor()$: Permet de transformer la donnée en Tensor.
- $torchvision.transforms.Normalize(mean, std)$: Permet de normaliser les données d'origine à partir de la moyenne et de l'écart-type donnés en entrée.

2.2 Commandes liées à la description de l'architecture

Maintenant que nous pouvons gérer notre base de données parfaitement, il nous faut construire le réseau qui va apprendre sur celle-ci. Pour ce faire, nous devons créer une classe qui correspondra à notre réseau de neurones. Cette classe se construit comme un héritage du module nn de torch qui correspond au regroupement des fonctions permettant de créer un réseau de neurones. Notre classe comporte deux méthodes :

- **__init__** : Cette méthode permet d'effectuer l'initialisation de la classe. C'est dans cette méthode que nous devons définir notre réseau de neurones en créant les différentes couches. Dans la bibliothèque torch.nn, on retrouve toutes les couches classiques telles que : nn.Linear(in_features, out_features), nn.MaxPool2d(kernel_size), nn.ZeroPad2d(padding), nn.BatchNorm2d(num_features), nn.Conv2d(in_channels, out_channels, kernel_size), nn.ReLU().

— **forward** : Dans cette fonction, nous devons organiser notre réseau en créant l'ordre dans lequel les données vont traverser le réseau.

2.3 Commandes liées à l'apprentissage du modèle

L'étape d'apprentissage est l'étape cruciale d'un réseau de neurones. Elle va déterminer si le réseau sera utile. Pour ce faire, nous devons apprendre sur la base de données que nous aurons au préalable chargée dans un DataLoader. Avant d'effectuer cela, nous devons choisir la fonction d'optimisation et de perte que nous allons utiliser au cours de l'apprentissage. Ainsi, on peut extraire ces deux fonctions grâce aux commandes suivantes :

$$torch.optim.Adam(parameters, learning\_rate)$$
$$torch.nn.CrossEntropyLoss()$$

Une fois ces fonctions extraites, nous pouvons appliquer la boucle d'apprentissage. Une fois que notre apprentissage est fini, nous voulons sauvegarder notre modèle pour ne pas le réapprendre par la suite. Pour ce faire, on utilise la fonction suivante, qui va nous permettre d'enregistrer un dictionnaire contenant les différents poids associés à notre modèle :

$$torch.save(model.state\_dict(), file_name)$$

Pour charger un modèle, on doit d'abord définir le modèle cible puis charger le dictionnaire et charger les poids correspondants à chaque couche :

$$loaded\_model = model()$$
$$loaded\_model.load\_state\_dict(torch.load("model.pth"))$$

Finalement, on est maintenant capable de créer une IA et de l'enregistrer.

3 Exemple de code

Maintenant que nous connaissons les fonctions classique de Pytorch, on peut les appliquer sur la création d'un CNN et son apprentissage. Voici l'architecture du CNN implémenter :

```
class CNN(f.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.layer1 = f.Sequential(
            f.Conv2d(1, 16, kernel_size=5, stride=2, padding=2),
            f.ReLU())
        self.layer2 = f.Sequential(
            f.Conv2d(16, 32, kernel_size=5, stride=2, padding=2),
            f.ReLU())
        self.layer3 = f.Sequential(
            f.Conv2d(32, 32, kernel_size=5, stride=2, padding=2),
            f.ReLU())
        self.layer4 = f.Sequential(
            f.Conv2d(32, 10, kernel_size=7, stride=1, padding=0))
        #self.fc = f.Linear(7*7*32, num_classes)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out= self.layer3(out)
        out= self.layer4(out)
        out = out.reshape(out.size(0), -1)#flatten
        #out = self.fc(out)
        return out
```

Ensuite, on va utiliser les GPU si celui-ci est disponible. On utilise doc la variable suivante :

```
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

Ensuite, on définit les Dataloader que nous allons utiliser :

```
    train_loader = torch.utils.data.DataLoader(dataset = training_set,
                                       batch_size=batch_size,
                                       shuffle=True,
                                       num_workers=2)
    valid_loader = torch.utils.data.DataLoader(dataset = valid_set,
                                       batch_size=batch_size,
                                       shuffle=False,
                                       num_workers=2)
```

On passe ensuite à l'étape d'apprentissage, en fixant les hyperparamètres et en faisant le choix de nos fonctions de perte et d'optimisation :

```
H = 30
lr = 1e-2 #learning rate
beta = 0.9 #momentum parameter
n_epoch = 10#number of iterations
input_size = 784

model = CNN(10)
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=beta)
criterion = nn.CrossEntropyLoss()
```

Finalement, on applique la boucle d'apprentissage :

```
    for epoch in range(n_epoch):

    loss_tot = 0

    for i, (images, labels,_) in enumerate(train_loader):

        # Reshape images to (batch_size, input_size), actual shape is (batch_size, 1,
            28, 28)
        #images_vec = images.view(-1, input_size)

        #Forward Pass
        O = model.forward(images)

        #Compute Loss
        l = criterion(O, labels)

        #Print Loss
        loss_tot += l.item()
        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Batch Loss: {:.4f}'
                   .format(epoch+1, n_epoch, i+1, num_batch, l.item()/len(labels)))

        #Backward Pass (Compute Gradient)
        optimizer.zero_grad()
        l.backward()

        #Update Parameters
        optimizer.step()
```

4 Conclusion

Cette documentation regroupe les différentes fonctions utiles dans la création d'un réseau de neurones. Elle expose la nature d'un Tensor, les fonctions liées à la base de données, à l'architecture du réseau et à l'apprentissage de celui-ci. Finalement, nous avons présenté un exemple d'utilisation de ces fonctions dans le cadre d'une application concrète de classification d'images.