Classification supervisée - TP5

Rémy Degenne

30 Mars 2022

Mise en place: dans le dossier du cours, créer un notebook (avec `jupyter notebook`) et lui donner un titre.

1 Documentation

Documentation de PyTorch: `https://pytorch.org/docs/stable/index.html`
Documentation de scikit-learn: `https://scikit-learn.org/0.24/index.html`
Documentation de matplotlib: `https://matplotlib.org/stable/tutorials/introductory/usage.html`
Documentation de numpy: `https://numpy.org/doc/1.18/`

2 Imports

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda
```

`PyTorch` est une librairie python dédiée au Machine Learning, en particulier avec des réseaux de neurones. Elle comprend beaucoup plus de fonctions liées aux réseaux que `scikit-learn`.

En particulier, `torch.nn` contient des implémentations de nombreux types de couches de réseaux.

- `nn.Conv2d` : couche convolutive
- `nn.Linear` : transformation linéaire
- `nn.MaxPool2d` : couche de pooling

Voir la liste des couches: `https://pytorch.org/docs/stable/nn.html` .

`torch.nn.functional` (que l'on a importé sous le nom `F`) contient des fonctions utiles comme des fonctions d'activation, par exemple `F.relu`.

3 Construire un Réseau de Neurones

On crée une classe `Net` qui représentera notre réseau.

```
class Net(nn.Module):
    def __init__(self):
      super(Net, self).__init__()
      # Ici on definit des couches

    def forward(self, x):
      #Ici on indique dans quel ordre on applique les couches aux donnees
```

On définit des couches dans la méthode `__init__` de notre classe. la méthode `forward` indique au réseau quelles opérations appliquer aux données.

```
class Net(nn.Module):
    def __init__(self):
      super(Net, self).__init__()
      # On definit des couches
      # couche convolutive avec 1 image en entree, 32 images en sortie, noyau de taille 3, stride 1
      self.conv1 = nn.Conv2d(1, 32, 3, 1)
      # couche lineaire
      self.fc1 = nn.Linear(5408, 100)
      self.fc2 = nn.Linear(100, 10)

    # Cette fonction indique comment une donnee x est transformee par le reseau
    def forward(self, x):
      # On utilise la couche conv1 definie plus haut
      x = self.conv1(x)
      # On applique ensuite une fonction d'activation relu
      x = F.relu(x)
      # pooling
      x = F.max_pool2d(x, 2)
      # on applatit l'image x pour avoir un vecteur
      x = torch.flatten(x, 1)
      # couche lineaire, puis activation
      x = self.fc1(x)
      x = F.relu(x)
      # derniere couche lineaire
      x = self.fc2(x)

      # sortie du reseau
      output = F.softmax(x, dim=1)
      return output

my_nn = Net()
print(my_nn)
```

- Décrire la structure du réseau de l'exemple (par exemple avec un schéma).

La définition de notre réseau peut ne pas être valide. Si les dimensions des couches ne sont pas spécifiées correctement, le code ci-dessus ne retournera pas d'erreur, mais le réseau ne sera pas utilisable. Pour le tester, on peut observer la sortie du réseau sur de fausses données:

```
# On cree une image aleatoire 28*28
random_data = torch.rand((1, 1, 28, 28))

my_nn = Net() # on cree un reseau de classe Net
result = my_nn(random_data) # on calcule la sortie du reseau sur la donnee aleatoire
print (result)
```

- Construire un perceptron multic-couche à deux couches, avec des fonctions d'activation ReLU.
- Construire un réseau avec deux couches convolutives, chacune suivie d'une fonction d'activation ReLU et d'une couche de pooling. Après ces couches, on utilisera au moins deux couches linéaires (couches de MLP).
- Ajouter une couche de dropout au réseau convolutif.

4 Construire un jeu de données

On va utiliser un jeu de donnees appelé FashionMNIST, qui contient des images 28×28 de vêtements, avec 10 classes (types de vêtements).

```
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

val_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

On peut visualiser les données avec le code suivant:

```
labels_map = {
    0: "T-Shirt",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
    8: "Bag",
    9: "Ankle Boot",
}
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data), size=(1,)).item()
    img, label = training_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(labels_map[label])
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

PyTorch utilise une classe `DataLoader` pour gérer les données.

```
train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
val_dataloader = DataLoader(val_data, batch_size=64, shuffle=True)
```

L'objet `train_dataloader`, quand on l'appelle avec `train_features, train_labels = next(iter(train_dataloader))`, renvoie un groupe de 64 images avec leurs étiquettes (le nombre 64 est spécifié dans `batch_size`). Quand on arrive au bout des exemples, l'argument `shuffle=True` lui indique de mélanger les données.

5 Entrainer un réseau

Avant de pouvoir entrainer notre classeur, on doit choisir certains hyper-paramètres:

- nombre d'époques
- taille d'un "batch": nombre d'exemples utilisés entre chaque mise à jour des poids
- taille d'un pas de gradient

```
epochs = 5
batch_size = 64
learning_rate = 1e-3
```

Entrainer un réseau dans PyTorch est plus compliqué que dans scikit_learn, mais on a un contrôle plus fin de ce que l'on fait. Il faut d'abord choisir la fonction de perte utilisée pendant l'apprentissage (on va choisir les poids pour essayer de minimiser cette fonction), par exemple

```
loss_fn = nn.CrossEntropyLoss()
```

Il faut aussi choisir l'algorithme d'optimisation utilisé, c'est à dire la manière de changer les poids au cours de l'apprentissage. On peut par exemple utiliser l'algorithme de gradient stochastique (SGD) comme dans le cours

```
optimizer = torch.optim.SGD(my_nn.parameters(), lr=learning_rate)
```

Les fonctions suivantes implémentent une époque d'apprentissage d'une part, et le calcul de l'erreur de validation d'autre part:

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # On calcule la prediction du modele et sa perte
        pred = model(X)
        loss = loss_fn(pred, y)

        # On calcule le gradient et on met a jour les poids
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f} [{current:>5d}/{size:>5d}]")


def val_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    val_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            val_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    val_loss /= num_batches
    correct /= size
    print(f"Validation Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {val_loss:>8f} \n")
```

- Expliquer ce que font ces deux fonctions

On peut maintenant entrainer le reseau:

```
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, my_nn, loss_fn, optimizer)
    val_loop(val_dataloader, my_nn, loss_fn)
print("Done!")
```

- Reproduire et entrainer le résear LeNet 5.
- Afficher des courbes de taux de succès d'entrainement et de validation d'un réseau de neurones convolutif.

- Observer la sortie de la première couche du réseau sur quelques exemples.
- Compétition: construire un réseau ayant la meilleure performance possible sur l'ensemble de validation (en ne l'entrainant que sur l'ensemble d'entrainement).