ENS Lyon
2023/2024

Titouan Vayer

TD n° X : Graph neural network

- Exercise 1: Hands-on with Pytorch and automatic differentiation -

The aim of this exercise is to familiarize you with `PyTorch` (which will be at the core of python libraries for GNN). We will need to import the following libraries.

```
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

The first hands-on is just tu understand the concept of automatic differentiation (explanations on the board). Consider a function $f(x) = \frac{1}{2}x^3 - 3x^2 + 1$. To compute the gradient of f you can run the following code in `PyTorch`.

```
x = torch.Tensor([2]) # at which point x you need the gradient
x.requires_grad = True # tell PyTorch it needs to compute the gradients wrt x
loss = f(x) # computing the function value = forward pass
loss.backward() # backward pass, to prepare the computation of the gradient
```

(i) Run the code at check that it is correct.

We will then train very simple models with `PyTorch` to get familiar with it.

(ii) Create samples $\forall i \in [\![n]\!], y_i = f(x_i) + \varepsilon_i$ where $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$ and $x_1, \cdots, x_n$ are equally spaced between $-10, 10$. Plot the functions f along with the sample points.

(iii) We will train a neural network to estimation the true function f from the noisy samples. A feed-forward neural-network can be constructed this way

```
class NeuralNetworkRegressor(nn.Module):
    def __init__(self, d1 = 10):
        super(NeuralNetworkRegressor, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(1, d1),
            nn.ReLU(),
            nn.Linear(d1, d1),
            nn.ReLU(),
            nn.Linear(d1, d1),
            nn.ReLU(),
            nn.Linear(d1, 1),
        )

    def forward(self, x): # implements a forward pass
        x = self.flatten(x)
        pred = self.linear_relu_stack(x)
        return pred
```

What is the number of parameters of this neural-network ?

(iv) To train the model we will need a loss and a optimization method. We will use the `PyTorch` optimizer with predefined learning rate with `optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)` and the MSE loss with `loss_fn = nn.MSELoss()`. Instantiate a model and complete the following code to train the neural network. Plot the corresponding loss and the prediction.

```
losses = [] #store the losses at each epoch
for i in range(epochs):
    # Compute prediction and loss
    pred = model()
    loss = loss_fn(, )

    # Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    losses.append(loss.item())
```

(v) What happens when the number of parameters explodes ? (take for example $d_1 = 200$).

(vi) We will now experience classification on the two-moons dataset. You can import it following the code

```
from sklearn.model_selection import train_test_split
from sklearn import datasets, metrics
X, y = datasets.make_moons(n_samples= 1000, random_state = 42, noise = 0.1)

# Split train/test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, shuffle=False
)


#PyTorch requires torch.Tensor as input not numpy array
Xt = torch.from_numpy(X_train).type(torch.FloatTensor)
yt = torch.from_numpy(y_train).type(torch.FloatTensor)
```

Plot the dataset and along with the classes of each point.

(vii) We will now consider a different neural-network for classification. Complete the `predict` function of the following class.

```
class NeuralNetworkClassifier(nn.Module):
    def __init__(self, d1 = 10):
        super(NeuralNetworkClassifier,self).__init__()
        self.fc1 = nn.Linear(2,d1)
        self.fc12 = nn.Linear(d1, d1)
        self.fc2 = nn.Linear(d1,1)
        self.output_fun = nn.Sigmoid()

    def forward(self,x):
        x = self.fc1(x)
        x = nn.functional.relu(x)
        x = self.fc12(x)
        x = nn.functional.relu(x)
        x = self.fc2(x)
        return self.output_fun(x).ravel()
```

```
    def predict(self,x):
        # predict the class with the highest score
```

To train we will use the binary cross entropy (explanations on the board)

```
model = NeuralNetworkClassifier(#define it)
learning_rate = #define it
epochs = #define it

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
loss_fn = nn.BCELoss()
```

Complete the following code

```
losses = []
test_acc = []
for i in range(epochs):
    # Compute prediction and loss
    pred = model()
    loss = loss_fn(,)

    # Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    losses.append(loss.item())

    # Calculate the prediction on the test
    test_pred = model.predict(torch.from_numpy(X_test).type(torch.FloatTensor)).numpy(
    # Look at the accuracy on the test
    test_acc.append(metrics.accuracy_score(y_test,test_pred))
```

and plot the loss and the test accuracy.

(viii) Visualize the decision boundary with the help of the following code

```
from matplotlib.colors import ListedColormap

def pred_func():
    return #to complete

colors = ['red', 'blue']

def plot_decision_boundary(pred_func, X,y):
    cm_bright = ListedColormap(["#539CFF","#FFD053"])
    # Set min and max values and give it some padding
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    h = 0.01
    # Generate a grid of points with distance h between them
    xx,yy=np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Predict the function value for the whole grid
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=cm_bright)
    plt.scatter(X[:, 0], X[:, 1], c= [colors[y[i]] for i in range(y.shape[0])],
                cmap=cm_bright)
```