Python softwares for machine and deep learning

Bartłomiej Błaszczyszyn ([1]), Mohamed Kadhem Karray ([2])

(1) INRIA ; (2) Orange Labs OLN/RNM/SRA
Project : Cross-Layer and QoE

28 novembre 2022

Acknowledgement : We thank our colleagues R. Fara, A. Jouidi, and, M. Mhedhbi for their help during our first steps in this subject.

Outline

Introduction

Python softwares

- Basic softwares
- Hints to start
- Example : Linear fitting with sklearn

PyTorch package

- Tensors
- Devices
- Image datasets
- Differentiation

PyTorch for neural networks

- Example : Linear fitting with PyTorch
- Logistic regression with PyTorch
- Multiclass logistic regression with PyTorch
- Optimization in PyTorch
- Multilayer neural network

Bibliography

Introduction

- Data-science (machine and deep learning) is becoming a widespread tool to solve problems in many domains including telecommunications
- Data-science has solid mathematical foundations inherited from statistics :
  - not in the scope of the present presentation
  - The reader is refered to mathematical books to understand the essential concepts and results of data-science ; e.g. [1]
- We shall focus here on useful softwares to write codes for data-science ; specifically
  - Python language and associated softwares
- Mathematics and softwares are oftenly imbricated in data-science writings
  - We believe that it is much more clear to present mathematics and softwares separately

Python softwares ([1])

- *Python* : Programming language : https ://www.python.org
- Two alternatives to use Python for machine and deep learning :
  - *GoogleColab* : Editor to write and excute Python code in a navigator, without installing any software locally : https ://colab.research.google.com (requires a Google account)
  - *Anaconda* : Development environment (and libraries manager) for Python : `https://www.anaconda.com`. It includes :
    - *Spyder* : Editor, debugger (Alternative : PyCharm)
    - *Jupyter* : Web-based application for creating notebooks combining code, text, and visualizations
- Courses on Python [2], [6], and its use for machine and deep learning ; e.g. [3], . . .
- Datasets for machine learning
  - *Kaggle* : https ://www.kaggle.com (requires a Kaggle account)
  - *CIFAR* : https ://www.cs.toronto.edu/˜kriz/cifar.html

1. Please refer to the Web sites for the licence terms.

Python packages (libraries)

- Python has *packages* which are software libraries with specific functionalies ; e.g.
  - *math* : mathematical functions
  - *NumPy* : for creating and manipulating vectors and matrices
  - *pandas* : to represent datasets in memory (with *DataFrames* : matrices with named columns and numbered rows)
  - *matplotlib* : make plots
  - *scikit-learn* (*sklearn*) : machine learning package
  - *PyTorch* : Neural network package (in particular for deep learning)
    - Documentation `https://pytorch.org/docs/stable`
    - Source for PyTorch examples below : [3]
    - Alternatives : *TensorFlow, kera* ; Usage examples on GoogleColab

Hints to install packages

- Some packages are installed by default ; e.g. math, panda, numpy, . . .
- To install packages on Google Colab
  - Use 'pip' command ; e.g. !pip install PackageName
- To install PyTorch with Anaconda :
  - Connect as administrator
  - Run : **Anaconda Powershell** as administrator
  - Execute : **conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch**
- With Anaconda, some packages are on specific channels, for example to install *shap* package (game theoretic approach to explain the output of machine learning model), execute :
  - **conda config –add channels conda-forge**
  - **conda config –set channel_priority false**
  - **conda install shap**

Hints for Jupyter

- Jupyter : Web-based application for creating notebooks (file extension '**.ipynb**')
- To change the directory for jupyter 'notebooks' :
  - Run **Anaconda Powershell**
  - Execute **jupyter notebook --notebook-dir=YourDirectory**
- In a notebook,
  - to run a Unix command, begin it with % ; e.g. %ls
  - to run a DOS command, begin it with ! ; e.g. !dir
- *JupyterLab* : alternative to Anaconda
- See notebook **00PythonTutorial.ipynb** for a rapid tutorial for Python [2]

Using packages

- Example (Notebook **01UsingPackages.ipynb**) :

```
import math as m
m.sqrt(4)
import numpy as np
D=np.array([[1,4],[0,2],[7,5]]) # create 3x2 matrix
print(D) # display D
import pandas as pd
F=pd.DataFrame(data=D,columns=['X','Y'])
print(F) # display the DataFrame
```

Hints for Google Colab

- Google Colab may be seen as a Jupyter notebook stored in Google drive
- Colab notebooks run the code on Google's cloud servers
- Data can be imported into Colab notebooks from
  - local directory
    - Hit the file icon in the left of the notebook
    - then select import and navigate to the appropriate file
  - Google Drive account, GitHub, ...
    - Some sample codes are given by hitting <> in the left of the notebook
  - *Wget* is a free software package for retrieving files from Internet ; e.g.
    - !wget http ://files.fast.ai/data/examples/dogscats.tgz

Hints for Kaggle

- Kaggle : Datasets for machine learning
- Usage :
  - Register on https ://www.kaggle.com
  - Search for desired database ; e.g. ATP matches (for tennis) : file **ATP.csv**
  - To load data (in format of Pandas DataFrame) ;
    - in Kaggle notebook : df=pd.read_csv("/kaggle/input/atpdata/ATP.csv")
    - in Jupyter, download the data file **ATP.csv** from kaggle on a **LocalDirectory** then : df=pd.read_csv("LocalDirectory/ATP.csv")

Complementary tools

- *GitHub* (https ://github.com) : website and cloud service that helps developers store and manage their code, as well as track and control changes to it
- *GitLab* (https ://about.gitlab.com/) : almost similar to GitHub
  - GitLab is open-source, wheras GitHub is not open source.

Linear fitting with sklearn (cf. 02LinearFitting_sklearn.ipynb)

- Example :[4] : $y = wx + b$

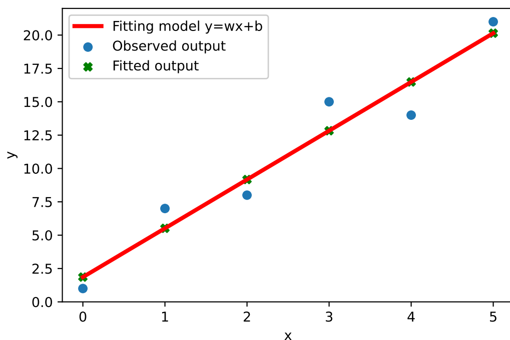
```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score,mean_squared_error
x=np.array([[0],[1],[2],[3],[4],[5]])
y=np.array([1,7,8,15,14,21])
model=LinearRegression()
model.fit(x,y)
```

Linear fitting with sklearn ; cont'd

```
plt.scatter(x,y,label='Observed output')
plt.xlabel("x")
plt.ylabel("y")
print('w=',model.coef_)
print('b=',model.intercept_)
plt.plot(x,model.predict(x),c='r',lw=3,label='Fitting model y=wx+b')
plt.scatter(x,model.predict(x),c='g',marker='X',label='Fitted output')
plt.legend()
plt.savefig('02LinearFitting_sklearn.eps')
plt.show()
coeff_determination=model.score(x,y)
print('Determination coefficient=',coeff_determination)
MSE=mean_squared_error(y,model.predict(x))
print("Mean squared error=",MSE)
```

Linear fitting with sklean : Result

```
w=[3.65714286]
b=1.8571428571428559
```




```
Determination coefficient=0.9362285714285714
Mean squared error=2.6571428571428584
```

PyTorch tensors [3] (cf. 03Tensors_PyTorch.ipynb)

- A tensor is a multidimensional matrix

```
import torch
import numpy as np
x=torch.ones(2,4,3) # matrix of dimension 2 × 4 × 3
y=x.numpy() # convert from torch to numpy
z=torch.from_numpy(y) # convert from numpy to torch
print(y)
print(z)
```

- Be careful with types and precision

```
x=torch.randn(4)
a=torch.from_numpy(np.ones(4))
print(a) # tensor([1., 1., 1., 1.], dtype=torch.float64)
b=a.float()
print(b) # tensor([1., 1., 1., 1.])
(x+a)==(x+b) # Result : tensor([False, False, False, True])
```

PyTorch tensors

- Broadcasting : When adding two tensors, their sizes are expanded to be of equal sizes (by replicating coefficients)

```
x=torch.tensor([[1.], [10.]]) # column vector 2×1
print(x)
y=torch.tensor([[0, 1]]) # row vector 1×2
print(y)
print((x+y)) # matrix 2×2
```

$$\begin{pmatrix} 1 \\ 10 \end{pmatrix} + \begin{pmatrix} 0 & 1 \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} 1 & 1 \\ 10 & 10 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 10 & 11 \end{pmatrix}$$

- More details on https ://pytorch.org/docs/stable/notes/broadcasting.html

PyTorch tensors

- In-place modification

```
x=torch.tensor([1])
y=x.add(-1)
print(x) # Result : tensor([1])
y=x.add_(-1) # The sign '_' modifies x in-place
print(x) # Result : tensor([0])
```

- Shared memory : The torch tensor and the associated numpy array share the same memory

```
a=np.ones(2)
b=torch.from_numpy(a)
a[0]=0
print(b) # Result : tensor([0., 1.]
b[1]=2
print(a) # Result : [0. 2.]
```

PyTorch tensors

- torch.unsqueeze(input, position) : generates a new tensor as output by adding a new dimension of size one at the desired position

```
x=torch.tensor([50, 25, 75, 100, 150])
print("Shape of x :", x.shape) # Result : torch.Size([5])
x0=torch.unsqueeze(x,0)
print("x0=torch.unsqueeze(x,0)=",x0)
print("Shape of x0 :", x0.shape) # Result : torch.Size([1, 5])
x1=torch.unsqueeze(x,1)
print("x1=torch.unsqueeze(x,1)=",x1)
print("Shape of x1 :",x1.shape) # Result : torch.Size([5, 1])
```

PyTorch : Devices [3] (cf. 04Devices_PyTorch.ipynb)

- Devices (Processors) :
  - CPU (central processing unit) : typical PC's processor
  - GPU (graphics processing unit) : typically on the cloud ; e.g. processor of GoogleColab
  - GPU is faster than CPU
  - TPU (tensor processing unit) : typically on the cloud ; developed specifically for tensors
- Check GPU : Is cuda available ? On GoogleColab :
  - Navigate to Edit/Notebook Settings
  - Select GPU from the Hardware Accelerator drop-down

```
import torch
import numpy as np
torch.cuda.is_available() # Result : True
```

PyTorch : Devices

- Put tensor on GPU :

```
x=torch.ones(1)
print(x.device) # Result : cpu
if torch.cuda.is_available() :
    y=x.to("cuda")
    print(y.device) # Result : cuda :0
    y=y.to("cpu")
    print(y.device) # Result : cpu
    z=torch.ones_like(x, device="cuda") # create tensor on GPU
    print(z.device) # Result : cuda :0
u=z.cpu().numpy() # go to cpu to get numpy array
```

PyTorch : Image datasets [3] (cf. 05ImageDatasets_PyTorch.ipynb)

- Example : CIFAR10 image dataset
  - The following code should be run on GoogleColab

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import torchvision
data_dir='content/data'
cifar=torchvision.datasets.CIFAR10(data_dir, train = True, download = True)
cifar.data.shape # Result : (50000, 32, 32, 3) ; 50000 images
```

PyTorch : Image datasets ; cont'd

```
x=torch.from_numpy(cifar.data).permute(0,3,1,2).float()
x=x/255
x=torch.narrow(x, 0, 0, 48) # select 48 first images
def show(img) : # Show images
    npimg=img.numpy()
    plt.figure(figsize=(20,10))
    plt.imshow(np.transpose(npimg, (1,2,0)), interpolation='nearest')
show(torchvision.utils.make_grid(x, nrow=12))
x.narrow(1,1,2).fill_(0) # Kills the green and blue channels
show(torchvision.utils.make_grid(x, nrow = 12))
```

Differentiation with PyTorch [3] (cf. 06Differentiation_PyTorch.ipynb)

- Let $f = \exp(wx)$, we aim to compute $\frac{\partial f}{\partial w}$

```
import torch
import numpy as np
w=torch.from_numpy(np.array([0.5])) # w=0.5 as a tensor
w.requires_grad_(True) # make it possible to differentiate with respect to
w
def fun(x) :
    return torch.torch.exp(w*x)
x1=torch.from_numpy(np.array([0.1]))
f=fun(x1)
f.backward() # Make differentiation
print(w.grad) # Result : tensor([0.1051],...
print(x1*torch.exp(w*x1)) # Check the result
```

Differentiation with PyTorch ; cont'd

- Accumulation of gradients for tensor.backward()

```
f=fun(x1)
f.backward() # Make differentiation a second time
print(w.grad) # Result : tensor([0.2103],...
# f.backward() # Gives an error because backward deletes the
computational graph
```

- Clear the gradients

```
w.grad.data.zero_() # Result : tensor([0.],...
```

- Retain computational graph

```
f=fun(x1)
f.backward(retain_graph=True)
f.backward() # retain_graph permits to differentiate a second time
print(w.grad) # Result : tensor([0.2103],...
```

Linear fitting with PyTorch [3] (cf. 07LinearFitting_PyTorch.ipynb) :

- Example : $y = wx + b$

```
import numpy as np
import torch
x=np.array([[0],[1],[2],[3],[4],[5]])
y=np.array([[1],[7],[8],[15],[14],[21]])
dtype=torch.FloatTensor
x=torch.from_numpy(x).type(dtype)
y=torch.from_numpy(y).type(dtype)
# Initial values of parameters
w=torch.from_numpy(np.array([[1.0]])).type(dtype)
b=torch.from_numpy(np.array([1.0])).type(dtype)
learning_rate = 1e-2
# Neural network with unidimensional input and output
net=torch.nn.Sequential(torch.nn.Linear(1,1))
```

Linear fitting with PyTorch ; cont'd

```
for m in net.children() :
    m.weight.data=w.clone()
    m.bias.data=b.clone()
loss_fn=torch.nn.MSELoss(reduction='sum')
optimizer=torch.optim.SGD(net.parameters(), lr=learning_rate)
for epoch in range(100) :
    yt=net(x)
    loss=loss_fn(yt,y)
    print("Iteration", epoch, ", loss=",loss.item())
    optimizer.zero_grad() # clear gradients for next train
    loss.backward() # compute gradients by back propagation
    optimizer.step() # update the parameters by applying gradients
print("Parameters after training :")
for param in net.parameters() :
    print(param) # Result : w=3.6579, b=1.8544
```

Linear fitting with PyTorch

- The function *torch.nn.MSELoss(reduction='sum')* is

$$l(\tilde{y}, y) = \sum_{j=1}^{n} (\tilde{y}_j - y_j)^2, \quad \text{where } \tilde{y}_j = wx_j + b$$

  - Minimizing this function with respect to the parameters $w, b \in \mathbb{R}$ corresponds to *least-squares fitting*; cf. [1, Definition 2.1.3 p.30]
- The function *torch.optim.SGD()* corresponds to *stochastic gradient descent* algorithm [1, § 10.2 p.164]

Logistic regression with PyTorch

- In logistic regression, we have to maximize the *log-likelihood function* [1, Equation (2.4.5) p.49] ; i.e. to minimize

$$l(p, y) = -\sum_{j=1}^{n} \left[y_j \log(p_j) + (1 - y_j) \log(1 - p_j) \right], \; p_j = \frac{1}{1 + e^{-\beta x_j}}$$

  with respect to the parameter β
  - In PyTorch, the function *torch.nn.BCELoss(reduction='sum')* implements the above function (BCE for Binary Cross Entropy)
- The function *torch.nn.BCEWithLogitsLoss(reduction='sum')* implements

$$\tilde{l}(z, y) = -\sum_{j=1}^{n} \left[y_j \log(\phi(z_j)) + (1 - y_j) \log(1 - \phi(z_j)) \right]$$

  where $\phi(x) = \frac{1}{1+\exp(-x)}$ is the *sigmoid* function.
  - We have to minimize

$$l(\phi(\beta x), y) = \tilde{l}(\beta x, y)$$

  - Using $\tilde{l}$ is more numerically stable than using l

Multiclass logistic regression with PyTorch

- For logistic regression with K classes, the negative log-likelihood function equals [1, Equation (2.5.11) p.51]

$$l(p,y) = -\sum_{j=1}^{n}\sum_{k=1}^{K} \mathbf{1}\left\{y_j = k\right\} \log\left(p_j\left(k\right)\right), \quad p_j\left(k\right) = \frac{e^{\beta_k . x_j}}{\sum_{l=1}^{K} e^{\beta_l . x_j}}$$

  - In PyTorch, the function *torch.nn.NLLLoss(reduction='sum')* implements the above function
  - Note that

$$l(p,y) = -\sum_{j=1}^{n} \log \mathrm{sofmax}_{y_j}\left(\beta_1 . x_j, \ldots, \beta_K . x_j\right),$$

  where *softmax* is the function
  $(z_1, \ldots, z_K) \mapsto \left(\frac{e^{z_1}}{\sum_{l=1}^{K} e^{z_l}}, \ldots, \frac{e^{z_K}}{\sum_{l=1}^{K} e^{z_l}}\right)$

Optimization in PyTorch

- Loss functions are in *torch.nn* and optimizers in PyTorch are in *torch.optim*
- There are variants of the stochastic gradient optimization algorithms commonly used by deep learning practitioners :
  - *Momentum*, *Nesterov accelerated gradient*, *Adagrad*, *Momentum*, *RMSprop*, *Adam*, ...
  - For a survey of these algoritms see [7]

Multilayer neural network

- We already built a simple neural network in example "linear fitting with PyTorch" comprising a single node :

```
net=torch.nn.Sequential(torch.nn.Linear(1,1))
```

- More general neural networks can be built by stacking layers from the input towards the output respectively. Here is an example of a neural network with one hidden layer comprizing 10 nodes :

```
import torch
import torch.nn as nn
net=nn.Sequential(nn.Linear(1,10),nn.ReLU(),nn.Linear(10,1),nn.Sigmoid())
```

- where the activation functions are
  - nn.ReLU$(x) = \max(x, 0)$
  - nn.Sigmoid$(x) = \frac{1}{1+\exp(-x)}$
- An alternative way to build a neural network is to write a sub-class of the class *torch.nn.Module*

Multilayer neural network

- Create a sub-class, say 'Net', of the class *torch.nn.Module* : one has to implement the constructor __init__(self, ...) and the neural network function forward(self, x) :

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class Net(nn.Module) :
    def __init__(self) :
        super(Net,self).__init__()
        self.f1=nn.Linear(in_features=1,out_features=10) # hidden
        self.f2=nn.Linear(in_features=10,out_features=1) # output
    def forward(self,x) :
        x=self.f1(x)
        x=F.relu(x) # activation function for hidden layer
        x=self.f2(x)
        return F.sigmoid(x) # activation function for output layer
```

Multilayer neural network

- The neural network function should be called as follow :

net=Net() # instance of the subclass Net

x=torch.randn(6,1) # input

y=net(x) # calculate the output ; avoid calling net.forward() explicitly

- Gradients calculated automatically (with back propagation) if you use functions compatible with *Autograd* (automatic backward differentiation)
  - this allows update of the parameters by the optimizer
  - Parameters added as class attributes are seen by Net.parameters()
- Cf. **08Fitting_PyTorch.ipynb** [5] for examples using multilayer neural networks to fit data generated as follows :
  - $y = x^2 + Z$; or $y = \sin(x) + Z$
  - where Z is a Gaussian random variable

Bibliography

[1] B. Błaszczyszyn and M. K. Karray.
Data science : From multivariate statistics to machine and deep learning.
Book in preparation, 2022.

[2] S. Korokithakis.
Tutorial - Learn Python in 10 minutes - Web site, 2021.
https://www.stavros.io/tutorials/python.

[3] M. Lelarge, J-J. Vie, A. Bursuc, and K. Scaman.
Deep Learning Do It Yourself ! - Web site, 2020.
https://dataflowr.github.io/website.

[4] L. Ben Othman.
Introduction to machine learning - Lecture notes, 2021.
Faculté des Science de Tunis.

[5] B. Phillips.
Regression with neural networks in PyTorch - Web site, 2021.
https://medium.com/@benjamin.phillips22/simple-regression-with-neural-networks-in-pytorch-313f06910379.

[6] M. Pilgrim.
Dive into Python 3.
Apress, 2009.

[7] S. Ruder.
An overview of gradient descent optimization algorithms.
Computing Research Repository in arXiv, abs/1609.04747, 2016.