Deep Learning and Neural Network

Marie Beurton-Aimar, LE Van Linh

December 15, 2020

Data Loading

Import

- Loading a matrix - by using pandas library :
  `dataTrain=pandas.read_excel(‘‘data-train.xlsx’’, header=0)`
- Normalizing the data - by using scikit-learn :
  `sts=StandardScaler`
  `DTrain=sts.fit_transform(dataTrain[dataTrain.columns[:-1]]`
- Convert and creating target variable (the last column is named 'classe' in the data file) - the variable is coded as int 0/1 :
  `yTrain = dataTrain.classe.astype('category').cat.codes.values`

Data Loading

Transform

- Tensor creating : special data type to manage data in pytorch.
- Convert numpy matrix of data in tensor :
  `tensor_DTrain = torch.FloatTensor(DTrain)`
- Create a tensor for target variable :
  `tensor_yTrain= torch.FloatTensor(yTrain)`

Data Loading

Images loading

- Several possibilities : give the path names or read a file containing image file names.
- One sample with path files :

```
data_dir ='Cat_Dog_data/train'
transform = transforms.Compose([transforms.Resize(255),
                transforms.CenterCrop(224),
transforms.ToTensor()])
dataset = datasets.ImageFolder(data_dir, transform=transform
dataloader = torch.utils.data.DataLoader(dataset, batch_size
                 shuffle=True)
images, labels = next(iter(dataloader))
helper.imshow(images[0], normalize=False)
```

Model in Pytorch

Designing Class

- By inheritance of `torch.nn.Module` class.
- Overload methods : `__init__` et `__forward__`
- `__init__` to describe layers and transfert functions
- __forward__ to describe the sequence of calculus from input to output.

Model in Pytorch

```
class MyPS(torch.nn.Module):
    # p explicative variable  = p neurons
    # only one layer - as perceptron
    def __init__(self,p):
        #call cons
        super(MyPS,self).__init__()

        #input layer (p variables) to output (1 neuron)
        self.layer1 = torch.nn.Linear(p,1)

        #sigmod function to output
        self.ft1 = torch.nn.Sigmoid()
```

Model in Pytorch

```
#output computing
#from  matrix of  x input-
def forward(self,x):
     #layer computing
     comb\_lin = self.layer1(x)
     #applying sigmoid function
     proba = self.ft1(comb_lin)
     return proba
```

Model in Pytorch

Evaluating

- Optimizing criteria : MSE for example

$$MSE(x, y) = (x - y)^2$$

```
criteria\_ps = torch.nn.MSELoss()
```

- Optimizing algorithm : ADAM - kind of stochastic gradient descent.

```
#Model instanciation
#.shape[1] is the number of descriptors
ps = MyPS(tensor_DTrain.shape[1])
#algorithm
#needs parameters
optimizer\_ps = torch.optim.Adam(ps.parameters())
```

Model in Pytorch

Loss

- Each layer has weights - parameters - initialized randomly at the beginning (if not pre-trained).
- Display initial values :

```
print(ps.layer1.weight)
```

- Possibility to display intercept :

```
print(ps.layer1.bias)
```

Model in Pytorch

Running

- Output computing :

```
ypred = ps.forward(tensor_DTrain)
# or ypred = ps(tensor_DTrain)
```

MSE computing

- First transform the previous result in vector and computing MSE

```
MSE1st = criteria_ps(yPred.squeeze(),tensor_yTrain)
```

Learning loop

```
def train_session(X,y,classifier,criterion,optimizer,n_epochs=10000):
     #to collect loss at each iteration
     losses = numpy.zeros(n_epochs)
     #learning loop
     for iter in range(n_epochs):
               #gradient init
               #Have to do if not PyTorch cumulate values
               optimizer.zero_grad()
               #compute output
               yPred = classifier(X)
               #loss computing
               loss = criterion(yPred.squeeze(),y)
               #collect loss
               losses[iter] = loss.item()
               #retropropagation
               perte.backward()
               #update weights
               optimizer.step()
     #end
     return losses
```

Using the Model

```
#Learning step
losses = train_session(tensor_DTrain,tensor_yTrain,ps,
                       criteria_ps,optimizer_ps)
#Display loss
import matplotlib.pyplot as plt
plt.plot(numpy.arange(0,pertes.shape[0]),pertes)
plt.title("Evolution fnct de perte")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
```

Applying for test

```
#sample of test - loading
pdTest = pandas.read_excel("myfile.xlsx",header=0)
print(pdTest.info())
#normalize (with train parameters)
#Reuse the sts object
DTest = sts.transform(pdTest[pdTest.columns[:-1]])
print(DTest)
# tensor creating for data
tensor_DTest = torch.FloatTensor(DTest)
# tensor creating for y
tensor_yTest = torch.FloatTensor(
          pdTest.classe.astype('category').cat.codes.values)
```

Test peformances

```
#evaluation library
from sklearn import metrics
#function for testing
def test_session(X,y,classifier):
      #apply test to sample data
      tensor_proba = classifier(X)
      #transform in numpy vector
      proba = tensor_proba.squeeze().detach().numpy()
      #recode probability in 0/1
      pred = numpy.where(proba > 0.5, 1, 0)
      #convert predicted y in numpy
      yobs = y.detach().numpy()
      #compute confusion matrix
      mc = metrics.confusion_matrix(yobs,pred)
      #succes rate
      acc = metrics.accuracy_score(yobs,pred)
      #return matrix and accuracy
      return mc,acc
```

Multi-Layers description

```
#In init replacing the layer description by
#input layer (p variables) to second layer (2 neurones)
self.layer1 = torch.nn.Linear(p,2)
#go to output layer  - first hidden
self.ft1 = torch.nn.ReLU()
#middle layer (2 neurones) to output (1 neurone)
self.layer2 = torch.nn.Linear(2,1)
#function to  sigmod
self.ft2 = torch.nn.Sigmoid()
```

Multi-Layers description

```
#In forward
#apply linear to first layer
comb_lin_1 = self.layer1(x)
#transform using transfert function
out_1 = self.ft1(comb_lin_1)
#applying second layer
comb_lin_2 = self.layer2(out_1)
#transformation to output
out_2 = self.ft2(comb_lin_2)

return out_2
```

Multi-Layers evaluation

```
#criteria
criteria_pmc = torch.nn.MSELoss()
#instanciation
pmc = MyPMC(tensor_DTrain.shape[1])
#optimiser
optimiser_pmc = torch.optim.Adam(pmc.parameters())
```

Multi-Layers evaluation

```
#run learning
losses = train_session(tensor_DTrain,tensor_yTrain,
        pmc,criteria_pmc,optimiser_pmc)
#volution
plt.plot(numpy.arange(0,losses.shape[0]),losses)
plt.title("Loss evolution")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
# run test
mc,acc = test_session(tensor_ZTest,tensor_yTest,pmc)
# confusion matrix
print(mc)
```