Lab Deep Learning / Multi-Layer Perceptron for binary-classification / in pytorch

Author: geoffroy.peeters@telecom-paris.fr (mailto:geoffroy.peeters@telecom-paris.fr)

For any remark or suggestion, please feel free to contact me.

Objective:

The objective of this lab is to develop a two hidden layers MLP to perform **binary classification**.

We will use a MLP with 2 hidden layer with $n_{h1} = 20$ and $n_{h2} = 10$ hidden units and `relu` activation functions. You will perform 1000 iterations (epochs) of SGD to find the parameters.

Note: for this lab, we do not separate the dataset into a train, validation and test part.

Data normalization

You should normalize the data to zero mean and unit standard deviation

Model

There are various ways to write NN model in pytorch.

In this lab, you will write three different implementations:

- **Model A**: manually defining the parameters (W1,b1,W2,b2,W3,b3), writing the forward equations, writting the loss equation, calling the .backward() and manually updating the weights using W1.grad. You will write the loop to perform 1000 epochs.
- **Model B**: using the Sequential class of pytorch
- **Model C**: a custom torch.nn.Module class for this.

For Model B and C, you will use the ready made loss and optimization from the nn and optim packages. You can use the same code to optimize the parameters of Model B and C.

Loss

Since we are dealing with a binary classification problem, we will use a Binary Cross Entropy loss (use `torch.nn.BCELoss` for Model B and C).

Parameters update/ Optimization

For updating the parameters, we will use as optimizer a simple SGD algorithm (use `torch.optim.SGD` for Model B and C) with a learning rate of 0.1.

Don't forget that an optimizer is applied to a set of parameters (`my_model.parameters()` gives the parameters of the network for Model B and C). Once the gradients have been computed (after the backpropagation has been performed), you can perform one step of optimization (using `optimizer.step()` for Model B and C).

Backward propagation

Backpropagation is automatically performed in pytorch using the `autograd` package. First, reset the gradients of all parameters (using `optimizer.zero_grad()` for Model B and C), then perform the backpropagation `loss.backward()` .

Your task:

You need to add the missing parts in the code (parts between `# --- START CODE HERE` and `# --- END CODE HERE`)

Documentation:

- NN: https://pytorch.org/docs/stable/nn.html (https://pytorch.org/docs/stable/nn.html)
- Autograd: https://pytorch.org/docs/stable/autograd.html (https://pytorch.org/docs/stable/autograd.html)
- Optim: https://pytorch.org/docs/stable/optim.html (https://pytorch.org/docs/stable/optim.html)

Load the python packages

Entrée [ ]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable

import numpy as np
import matplotlib.pyplot as plt

student = False
```

Entrée [ ]:

```
print(torch.__version__)
```

Dataset

We take the usual circle dataset from sklearn.

Entrée [ ]:

```
from sklearn import datasets
X_np, y_np = datasets.make_circles(n_samples=1000, noise=0.2, factor=0.5)
```

We convert the `numpy tensors to torch tensors` . The difference being that the latters allows to do automatic gradient differentiation (back-propagation).

Entrée [ ]:

```
X = torch.from_numpy(X_np).float()
y = torch.from_numpy(y_np).float()
y = y.view(len(y), 1)
```

Entrée [ ]:

```
print(X.size())
print(y.size())
print(X.mean(dim=0))
print(X.std(dim=0))
```

Normalization

Entrée [ ]:

```
X -= X.mean(dim=0)
X /= X.std(dim=0)
print(X.mean(dim=0))
print(X.std(dim=0))
```

Definition of the hyper-parameters

Entrée [ ]:

```
n_in = X.shape[1]
n_h1 = 20
n_h2 = 10
n_out = 1

nb_epoch = 10000
alpha = 0.1
```

Model 1 (writing the network equations)

Here, you will define the variables and write the equations of the network yourself (as you would do in numpy). However you will use `torch tensors` instead of `numpy array` .

Why ? because torch tensors will allows you to automatically get the gradient. You will use `loss.backward` to launch the backpropagation from `loss` . Then, for all tensors you created and for which you declared `requires_grad=True` , you will get the gradient of `loss` with respect to this variable in the field `.grad` .

Example `W1 = torch.tensors(..., requires_grad=True)` ... `loss.backward` will have the gradient $\frac{dLoss}{dW1}$ in `W1.grad` .

Don't forget that the weight $W_1, W_2, \cdots$ matrices should be initialized randomly with small values; while the bias vectors $b_1, b_2, \cdots$ can be initialized to zero.

Entrée [ ]:

```
# --- We first initialize the variables of the network (W1, b1, ...)
if student:
    # --- START CODE HERE (01)
    W1 = ...
    W1.requires_grad = True
    b1 = ...

    W2 = ...
    W2.requires_grad = True
    b2 = ...

    W3 = ...
    W3.requires_grad = True
    b3 = ...
    # --- END CODE HERE

# --- We then write a function to perform the forward pass (using pytorch opertaors
# --- taking X as input and returing hat_y as output

def model(X):
    if student:
        # --- START CODE HERE (02)
        A0 = ...
        Z1 = ...
        A1 = ...
        Z2 = ...
        A2 = ...
        Z3 = ...
        A3 = ...
        hat_y = ...
        # --- END CODE HERE

    return hat_y

# --- We then iterate over epochs (we do not perform split into mini-batch here)
# --- For each iteration, we
# ---   a) perform the forward pass,
# ---   b) compute the loss/cost,
# ---   c) compute the backward pass to get the gradients of the cost w.r.t. the pa
# ---   d) perform the update of the parameters W1, b1, ...
for num_epoch in range(0, nb_epoch):

    # --- a) Forward pass: X (n_in, N), hat_y (n_out, N)
    hat_y = model(X)

    # -- We clip hat_y in order to avoid log(0)
    eps = 1e-10
    hat_y = torch.clamp(hat_y, eps, 1-eps)

    # --- b) Computing the loss/cost
    if student:
        # --- START CODE HERE (03)
        loss = ...
        cost = ...
        # --- END CODE HERE

    if num_epoch % 500 == 0:
        print('epoch {}, loss {}'.format(num_epoch, cost))
```

```
    # --- c) Backward pass
    cost.backward()

    # --- "with torch.no_grad()" temporarily set all the requires_grad flag to fals
    with torch.no_grad():
        # --- d) perform the update of the parameters W1, b1, ...
        if student:
            # --- the gradients dLoss/dW1 is stored in W1.grad, dLoss/db1 is storec
            # --- START CODE HERE (04)
            W1 = ...
            b1 = ...
            W2 = ...
            b2 = ...
            W3 = ...
            b3 = ...
            # --- END CODE HERE

    # --- We need to set to zero all gradients (otherwise they are cumulated)
    W1.grad.zero_()
    b1.grad.zero_()
    W2.grad.zero_()
    b2.grad.zero_()
    W3.grad.zero_()
    b3.grad.zero_()
```

Model 2 (using nn.sequential)

Here, you will use the package `torch.nn` which comes with a predefined set of layers. The syntax is close to the one of `keras` (`Sequential`), but differs in the fact that layers are splitted into the matrix multiplication followed by a non-linear activations (`keras` merge both using the `Dense` layers).

The model created will have all its parameters accessible as a dictionary and can be accessed using `model.parameters()` . It is therefore a convenient way to write simple sequential networks.

Entrée [ ]:

```
if student:
    # --- START CODE HERE (05)
    my_model = ...
    # --- END CODE HERE
```

Model 3 (using a class definition)

Here, you will write the network using the recommended pytroch way; i.e. by defining a class. This class inherit from the main class `torch.nn.Module` . You only need to write the `__init__` method and the `forward` method.

In object programming, the `__init__` method defines the attributes of your class. Since the attributes of your network are the parameters to be trained (weights and biases), you should declare in the `__init` all the layers that involve parameters to be trained (mostly the `Linear` layers which perform the matrix multiplication).

The `forward` method contains the code of the forward pass itself. It can of course call attributes defined in the `__init___` method. It is the method used when calling `model(x)` .

As before, the model created will have all its parameters accessible as a dictionary and can be accessed using `model.parameters()` .

Classes are convenient way to write more complex network than what you can do with `nn.sequential` . Note that you can actually include a `nn.sequential` in your class.

Entrée [ ]:

```
class Net(torch.nn.Module):
    def __init__(self, n_in, n_h1, n_h2, n_out):
        super(Net, self).__init__()

        if student:
            # --- START CODE HERE (06)
            self.fc1 = ... # hidden layer 1
            self.fc2 = ... # hidden layer 2
            self.fc3 = ...  # output layer
            # --- END CODE HERE

    def forward(self, X):

        if student:
            # --- START CODE HERE (07)
            A0 = ...
            A1 = ...    # activation function for hidden layer 1
            A2 = ...    # activation function for hidden layer 2
            A3 = ...    # activation function for output layer
            # --- END CODE HERE

        return A3

# --- START CODE HERE
my_model = Net(n_in, n_h1, n_h2, n_out)
# --- END CODE HERE
```

Criterion and Optimization for model 2 and model 3

The code of Model 1 is self-contained, i.e. it already contains all necessary instruction to perform forawrd, loss, backward and parameter updates.

When using `nn.sequential` (model 2) or a class definition of the network (model 3), we still need to define

- what we will minimize (the loss to be minimized, i.e. Binary-Cross-Entropy). We can of course write the equation of it by hand but pytorch comes with a very large number of pre-build loss functions (within `torch.nn`)
- how we will minimize the loss, i.e. what parameter update algorithms we will use (SGD, momentum). We can of course write the equation of it by hand but pytorch comes with a very large number of pre-build loss functions (within `torch.nn`)

Entrée [ ]:

```
if student:
    # --- START CODE HERE (08)
    criterion = ...
    optimizer = ...
    # --- END CODE HERE
```

Training for model 2 and 3

Having defined the network, the citerion to be minimized and the optimizer, we then perform a loop over epochs (iterations); at each step we

- compute the forward pass by passing the data to the model: `haty = model(x)`
- compute the the loss (the criterion)
- putting at zero the gradients of all the parameters of the network (this is important since, by default, pytorch accumulate the gradients over time)
- computing the backpropagation (using as before `.backward()`)
- performing one step of optimization (using `.step()`)

Entrée [ ]:

```
loss_l = []
for num_epoch in range(nb_epoch):

    if student:
        # --- START CODE HERE (09)
        hat_y = ... # Forward pass: Compute predicted y by passing  x to the model
        loss = ... # Compute loss
        # Zero gradients, perform a backward pass, and update the weights.
        ... # re-init the gradients (otherwise they are cumulated)
        ... # perform back-propagation
        ... # update the weights
        # --- END CODE HERE

    loss_l.append(loss)

    if num_epoch % 500 == 0:
        print('epoch {}, loss {}'.format(num_epoch, loss.item()))

# ----------------
plt.plot(loss_l)
```

Entrée [ ]:

```

```