Lab Deep Learning/ Multi-Layer Perceptron for classification/ in python

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

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

Objective:

We want to implement a two layers Multi-Layer Perceptron (MLP) with 1 hidden layer in Python, for a classification problem.

The output of the network is simply the output of several cascaded functions :

- Linear transformations. We note the weights of a linear transformation with W
- Additive biases. We note the parameters of additive biases with b
- Non-linearities.

For this, we will implement:

- the forward propagation
- the computation of the cost/loss
- the backward propagation (to obtain the gradients)
- the update of the parameters

Furthermore, we define the following sizes :

- $n^{[0]}$: number of input neurons
- $n^{[1]}$: number of neurons in hidden layer
- $n^{[2]}$: number of neurons in output layer
- m : number of training datapoints

Cost function

The **cost** is the average of the the **loss** over the training data. Since we are dealing with a binary classification problem, we will use the binary cross-entropy.

$$\mathcal{L} = -\left(y \log(\hat{y}) + (1-y) \log(1-\hat{y}) \right),$$

where

- the y are the ground-truth labels of the data and
- the $\hat{y}$ the estimated labels (outputs of the network).

Forward propagation

- $\underbrace{Z^{[1]}}_{(m,n^{[1]})} = \underbrace{X}_{(m,n^{[0]})} \underbrace{W^{[1]}}_{(n^{[0]},n^{[1]})} + \underbrace{b^{[1]}}_{n^{(1)}}$

- $\underbrace{A^{[1]}}_{(m,n^{[1]})} = f(Z^{[1]})$
- $\underbrace{Z^{[2]}}_{(m,n^{[2]})} = \underbrace{A^{[1]}}_{(m,n^{[1]})}\underbrace{W^{[2]}}_{(n^{[1]},n^{[2]})} + \underbrace{b^{[2]}}_{n^{(2)}}$
- $\underbrace{A^{[2]}}_{(m,n^{[2]})} = \sigma(Z^{[2]})$

where

- f is a `Relu` function (the code is provided)
- σ is a sigmoid function (the code is provided)

Backward propagation

The backward propagation can be calculated as

- $\underbrace{dZ^{[2]}}_{(m,n^{[2]})} = \underbrace{A^{[2]}}_{(m,n^{[2]})} - \underbrace{Y}_{(m,n^{[2]})}$
- $\underbrace{dW^{[2]}}_{(n^{[1]},n^{[2]})} = \frac{1}{m}\underbrace{A^{[1]}}_{(m,n^{[1]})}{}^{T}\underbrace{dZ^{[2]}}_{(m,n^{[2]})}$
- $\underbrace{db^{[2]}}_{(n^{[2]})} = \frac{1}{m}\sum_{i=1}^{m}\underbrace{dZ^{[2]}}_{(m,n^{[2]})}$
- $\underbrace{dA^{[1]}}_{(m,n^{[1]})} = \underbrace{dZ^{[2]}}_{(m,n^{[2]})}\underbrace{W^{[2]}}_{(n^{[1]},n^{[2]})}{}^{T}$
- $\underbrace{dZ^{[1]}}_{(m,n^{[1]})} = \underbrace{dA^{[1]}}_{(m,n^{[1]})} \odot f'(\underbrace{Z^{[1]}}_{(m,n^{[1]})})$
- $\underbrace{dW^{[1]}}_{(n^{[0]},n^{[1]})} = \frac{1}{m}\underbrace{X}_{(m,n^{[0]})}{}^{T}\underbrace{dZ^{[1]}}_{(m,n^{[1]})}$
- $\underbrace{db^{[1]}}_{(n^{[1]})} = \frac{1}{m}\sum_{i=1}^{m}\underbrace{dZ^{[1]}}_{(m,n^{[1]})}$

Backward propagation

Based on the previous formulae, write the corresponding backpropagation algorithm.

Parameters update

- Implement a **first version** in which the parameters are updated using a **simple gradient descent**:
  - $W = W - \alpha dW$
- Implement a **second version** in which the parameters are updated using the **momentum method**:
  - $V_{dW}(t) = \beta V_{dW}(t-1) + (1-\beta)dW$
  - $W(t) = W(t-1) - \alpha V_{dW}(t)$

IMPORTANT IMPLEMENTATION INFORMATION !

The $\odot$ operator refers to the point-wise multiplication operation. The matrix multiplication operation can be carried out in Python using `np.dot(.,.)` function.

Your task:

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

Note

The code is written as a python class (in order to be able to pass all the variables easely from one function to the other).

To use a given variable, you need to use `self.$VARIABLE_NAME` , such as `self.W1` , `self.b1` , ... (see the code already written).

Testing

For testing your code, you can use the code provided in the last cells (loop over epochs and display of the loss decrease). You should a cost which decreases (largely) over epochs.

Load packages

Entrée [8]:

```
%matplotlib inline
import numpy as np
from sklearn import datasets
from sklearn import model_selection
import matplotlib.pyplot as plt

student = True
```

Define a set of functions

Entrée [9]:

```
def F_standardize(X):
    """
    standardize X, i.e. subtract mean (over data) and divide by standard-deviation

    Parameters
    ----------
    X: np.array of size (m, n_0)
        matrix containing the observation data

    Returns
    -------
    X: np.array of size (m, n_0)
        standardize version of X
    """

    X -= np.mean(X, axis=0, keepdims=True)
    X /= (np.std(X, axis=0, keepdims=True) + 1e-16)
    return X
```

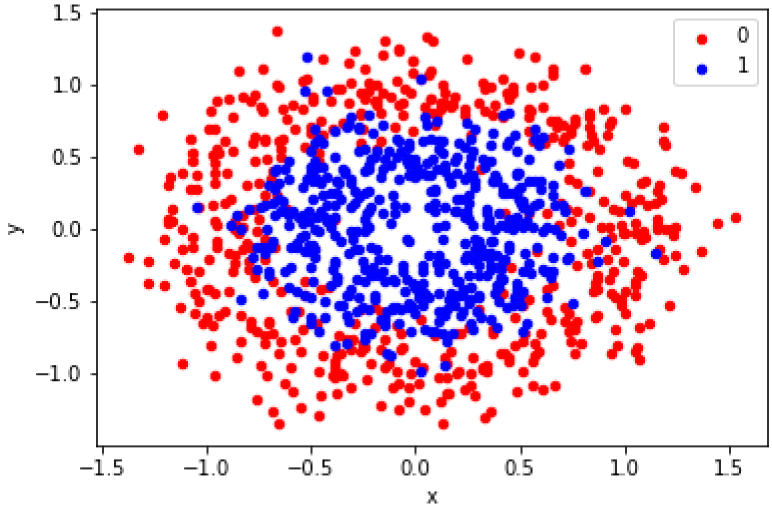
Entrée [10]:

```
def F_sigmoid(x):
    """Compute the value of the sigmoid activation function"""
    return 1 / (1 + np.exp(-x))

def F_relu(x):
    """Compute the value of the Rectified Linear Unit activation function"""
    return x * (x > 0)

def F_dRelu(x):
    """Compute the derivative of the Rectified Linear Unit activation function"""
    y = x
    y[x<=0] = 0
    y[x>0] = 1
    return y

def F_computeCost(hat_y, y):
    """Compute the cost (sum of the losses)

    Parameters
    ----------
    hat_y: (m, 1)
        predicted value by the MLP
    y: (m, 1)
        ground-truth class to predict
    """
    m = y.shape[0]

    if student:
        # --- START CODE HERE (01)
        loss =  ...
        # --- END CODE HERE

    cost = np.sum(loss) / m
    return cost

def F_computeAccuracy(hat_y, y):
    """Compute the accuracy

    Parameters
    ----------
    hat_y: (m, 1)
        predicted value by the MLP
    y: (m, 1)
        ground-truth class to predict
    """

    m = y.shape[0]
    class_y = np.copy(hat_y)
    class_y[class_y>=0.5]=1
    class_y[class_y<0.5]=0
    return np.sum(class_y==y) / m
```

Load dataset and pre-process it

Entrée [11]:

```
X, y = datasets.make_circles(n_samples=1000, noise=0.2, factor=0.5)

from pandas import DataFrame
# scatter plot, dots colored by class value
df = DataFrame(dict(x=X[:,0], y=X[:,1], label=y))
colors = {0:'red', 1:'blue'}
fig, ax = plt.subplots()
grouped = df.groupby('label')
for key, group in grouped:
    group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])
plt.show()
```



Entrée [12]:

```
print("X.shape: {}".format(X.shape))
print("y.shape: {}".format(y.shape))
print(set(y))

# X is (m, n_0)
# y is (m,)

# --- Standardize data
X = F_standardize(X)

# --- Split between training set and test set
# --- (m, n_0)
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size

# --- Convert to proper shape: (m,) -> (m, 1)
y_train = y_train.reshape(len(y_train), 1)
y_test = y_test.reshape(len(y_test), 1)

# --- Convert to oneHotEncoding: (nbExamples, 1) -> (nbExamples, nbClass)
n_0 = X_train.shape[1]
n_2 = 1

print("X_train.shape: {}".format(X_train.shape))
print("X_test.shape: {}".format(X_test.shape))
print("y_train.shape: {}".format(y_train.shape))
print("y_test.shape: {}".format(y_test.shape))
print("y_train.shape: {}".format(y_train.shape))
print("y_test.shape: {}".format(y_test.shape))
print("n_0=n_in: {} n_2=n_out: {}".format(n_0, n_2))
```

```
X.shape: (1000, 2)
y.shape: (1000,)
{0, 1}
X_train.shape: (800, 2)
X_test.shape: (200, 2)
y_train.shape: (800, 1)
y_test.shape: (200, 1)
y_train.shape: (800, 1)
y_test.shape: (200, 1)
n_0=n_in: 2 n_2=n_out: 1
```

Define the MLP class with forward, backward and update methods

Entrée [13]:

```
class C_MultiLayerPerceptron:
    """
    A class used to represent a Multi-Layer Perceptron with 1 hidden layers

    ...

    Attributes
    ----------
    W1, b1, W2, b2:
        weights and biases to be learnt
    Z1, A1, Z2, A2:
        values of the internal neurons to be used for backpropagation
    dW1, db1, dW2, db2, dZ1, dZ2:
        partial derivatives of the loss w.r.t. parameters
    VdW1, Vdb1, VdW2, Vdb2:
        momentum terms
    do_bin0_multi1:
        set wether we solve a binary or a multi-class classification problem

    Methods
    -------
    forward_propagation

    backward_propagation

    update_parameters

    """

    W1, b1, W2, b2 = [], [], [], []
    A0, Z1, A1, Z2, A2 = [], [], [], [], []
    dW1, db1, dW2, db2 = [], [], [], []
    dZ1, dA1, dZ2 = [], [], []
    # --- for momentum
    VdW1, Vdb1, VdW2, Vdb2 = [], [], [], []

    def __init__(self, n_0, n_1, n_2):
        self.W1 = np.random.randn(n_0, n_1) * 0.01
        self.b1 = np.zeros(shape=(1, n_1))
        self.W2 = np.random.randn(n_1, n_2) * 0.01
        self.b2 = np.zeros(shape=(1, n_2))
        # --- for momentum
        self.VdW1 = np.zeros(shape=(n_0, n_1))
        self.Vdb1 = np.zeros(shape=(1, n_1))
        self.VdW2 = np.zeros(shape=(n_1, n_2))
        self.Vdb2 = np.zeros(shape=(1, n_2))
        return


    def __setattr__(self, attrName, val):
        if hasattr(self, attrName):
            self.__dict__[attrName] = val
        else:
            raise Exception("self.%s note part of the fields" % attrName)



    def M_forwardPropagation(self, X):
        """Forward propagation in the MLP
```

```
        Parameters
        ----------
        X: numpy array (nbData, nbDim)
            observation data

        Return
        ------
        hat_y: numpy array (nbData, 1)
            predicted value by the MLP
        """

        if student:
            # --- START CODE HERE (02)
            self.A0 = ...

            self.Z1 = ...
            self.A1 = ...

            self.Z2 = ...
            self.A2 = ...

            hat_y = ...
            # --- END CODE HERE

        return hat_y


    def M_backwardPropagation(self, X, y):
        """Backward propagation in the MLP

        Parameters
        ----------
        X: numpy array (nbData, nbDim)
            observation data
        y: numpy array (nbData, 1)
            ground-truth class to predict

        """

        m = y.shape[0]

        if student:
            # --- START CODE HERE (03)

            self.dZ2 = ...
            self.dW2 = ...
            self.db2 = ...
            self.dA1 = ...

            self.dZ1 = ...
            self.dW1 = ...
            self.db1 = ...
            # --- END CODE HERE

        return


    def M_gradientDescent(self, alpha):
        """Update the parameters of the network using gradient descent
```

```
        Parameters
        ----------
        alpha: float scalar
            amount of update at each step of the gradient descent

        """
        if student:
            # --- START CODE HERE (04)
            self.W1 = ...
            self.b1 = ...
            self.W2 = ...
            self.b2 = ...
            # --- END CODE HERE

        return


    def M_momentum(self, alpha, beta):
        """Update the parameters of the network using momentum method

        Parameters
        ----------
        alpha: float scalar
            amount of update at each step of the gradient descent
        beta: float scalar
            momentum term
        """

        if student:
            # --- START CODE HERE (05)
            self.VdW1 = ...
            self.W1 = ...

            self.Vdb1 = ...
            self.b1 = ...

            self.VdW2 = ...
            self.W2 = ...

            self.Vdb2 = ...
            self.b2 = ...
            # --- END CODE HERE

        return
```

Perform training using batch-gradiant and epochs

Entrée [15]:

```
# hyper-parameters
n_1 = 10 # number of hidden neurons
nb_epoch = 5000 # number of epochs (number of iterations over full training set)
alpha=0.1 # learning rate
beta=0.9 # beat parameters for momentum


# Instantiate the class MLP with providing
# the size of the various layers (n_0=n_input, n_1=n_hidden, n_2=n_output)
myMLP = C_MultiLayerPerceptron(n_0, n_1, n_2)

train_cost, train_accuracy, test_cost, test_accuracy = [], [], [], []

# Run over epochs
for num_epoch in range(0, nb_epoch):

    # --- Forward
    hat_y_train = myMLP.M_forwardPropagation(X_train)

    # --- Store results on train
    train_cost.append( F_computeCost(hat_y_train, y_train) )
    train_accuracy.append( F_computeAccuracy(hat_y_train, y_train) )

    # --- Backward
    myMLP.M_backwardPropagation(X_train, y_train)

    # --- Update
    myMLP.M_gradientDescent(alpha)
    #myMLP.M_momentum(alpha, beta)

    # --- Store results on test
    hat_y_test = myMLP.M_forwardPropagation(X_test)
    test_cost.append( F_computeCost(hat_y_test, y_test) )
    test_accuracy.append( F_computeAccuracy(hat_y_test, y_test) )

    if (num_epoch % 500)==0:
        print("epoch: {0:d} (cost: train {1:.2f} test {2:.2f}) (accuracy: train {3:
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-15-65685b45c9c7> in <module>
     19
     20     # --- Store results on train
---> 21     train_cost.append( F_computeCost(hat_y_train, y_train) )
     22     train_accuracy.append( F_computeAccuracy(hat_y_train, y_
train) )
     23

<ipython-input-10-bd20d0845c5a> in F_computeCost(hat_y, y)
     31         # --- END CODE HERE
     32
---> 33     cost = np.sum(loss) / m
     34     return cost
     35
```

```
TypeError: unsupported operand type(s) for /: 'ellipsis' and 'int'
```

Display train/test loss and accuracy

Entrée [ ]:

```
plt.subplot(1,2,1)
plt.plot(train_cost, 'r')
plt.plot(test_cost, 'g--')
plt.xlabel('# epoch')
plt.ylabel('loss')
plt.grid(True)

plt.subplot(1,2,2)
plt.plot(train_accuracy, 'r')
plt.plot(test_accuracy, 'g--')
plt.xlabel('# epoch')
plt.ylabel('accuracy')
plt.grid(True)
```

Entrée [ ]:

```

```