Multi-Layer Perceptron Builder

File Name: MLP_builder_EXPLANATION.mlx

Creator: Marino Massimo Costantini

Last Update: November 29, 2023, 14:30 (GMT)

Table of Contents

Introduction

The MATLAB function explained, named `MLP_builder.m`, is designed to create and train a Multi-Layer Perceptron (MLP) neural network using a specified dataset provided in a .CSV file. The function provides customizable options for parameters such as the dataset file, shuffling preferences, layer sizes, hidden layer structure, activation functions, and some training options.

This function interfaces with a Python file 'MLP_builder.py' to construct and train the MLP according to the specified configuration. After training in Python is completed, the function extracts key features: weights and biases are saved into a Python dictionary named 'MLP_dict', scaling factors used for data normalization are stored in a Python array, and the losses are recorded in a Python array of tuples.

What you CANNOT do with this toolchain?

- Despite the flexibility to create and train an MLP of any size and any number of layers tailored to your dataset, there is a *limitation on the output layer, which cannot exceed 10 neurons*, limiting the maximum number of outputs to 10.

What could be improved in future versions?

- Code generation to implement the deployment code into an embedded system that use microcontrollers

Necessary Requirements:

- Ensure that your system has a version of Python compatible with MATLAB. Verify the compatibility of your Python version by visiting MATLAB Python Compatibility. If your current version is incompatible, you can install a suitable version of Python directly from Python Downloads. Please pay attention to the specified version on the former link.
- MATLAB Python Integration: Configure MATLAB to interface with a compatibile Python version. You can find more information on this link https://es.mathworks.com/help/matlab/matlab_external/create-object-from-python-class.html. You can confirm that the integration is functioning by executing the following

command in your command window: `>>pyenv`. You can also try to call basic python function as `>>py.numpy.array(magic(3))`. The expected output structures are outlined below:

```
Command Window
>> pyenv

ans =

  PythonEnvironment with properties:

         Version: "3.11"
      Executable: "C:\Users\marin\AppData\Local\Programs\Python\Python311\python.EXE"
         Library: "C:\Users\marin\AppData\Local\Programs\Python\Python311\python311.dll"
            Home: "C:\Users\marin\AppData\Local\Programs\Python\Python311"
          Status: Loaded
    ExecutionMode: InProcess
        ProcessID: "29620"
      ProcessName: "MATLAB"

>> py.numpy.array(magic(3))

ans =

  Python ndarray:

     8     1     6
     3     5     7
     4     9     2

    Use details function to view the properties of the Python object.

    Use double function to convert to a MATLAB array.
```

- Ensure that the Python file 'MLP_builder.py' and 'yourfilename.csv' are present in the current folder, together with the function itself 'MLP_builder.m'. Moreover if you want to perform the further deployment, ensure in your folder there are the corrispective functions and files: 'DeployMLP_Matlab.m', 'DeployMLP_Python.m' plus 'DeployMLP_system.slx' and the subsystem reference 'DeployMLP.slx'.
- Ensure that the structure of the '.csv' file aligns with your requirements. You can create an empty table using the Matlab command 'T=table()' and then write the table to a CSV file using 'writetable(T,'yourdataset.csv')', similar to the operations performed in the Matlab script 'dataset_builder.m'. *The first N columns should represent the inputs, while the remaining M columns should correspond to the outputs.* It is important to verify that N and M are correctly defined based on your data and share the same length. An example is provided below for reference.

	A	B	C	D	E
	yourdataset				
	In1	In2	In3	Out1	Out2
	Number	Number	Number	Number	Number
1	In1	In2	In3	Out1	Out2
2	1.94182648131446	0.324918657746345	0.215200296063467	2.91234602725121	1.04083583200079
3	0.795084055475428	-1.71758429099259	1.36958697914531	3.18626070191878	-1.95483905279027
4	-1.64027274920151	-0.503014754158564	-1.80432444884512	-7.55626084989542	0.0874654444573587
5	-0.117049894076415	-1.86564151072126	1.03521777671575	1.12296192534956	-1.98987020391568
6	0.54145155198261	-0.37542764132606	0.703999496462451	2.2780224000439	0.0064249055396734
7	0.101077420781588	0.63183244907776	1.49771637724114	5.22605900158277	0.996844517046934
8	-1.43464225253115	0.300962179299932	-0.676227057189636	-3.16236124480013	-0.920839895098947
9	-0.897938072721148	1.90382495245478	0.073998677840978	1.22788291325657	-0.308088507038257
10	1.94841093797156	-0.233442647324619	-0.184219177250513	1.1623107588954	1.0172100814111
11	1.17999323142957	1.1113478942361	-1.28198773129091	-1.55462206820708	-0.834737749891887
12	1.73333964126906	0.277833703436868	1.38959782892412	6.17996683147829	1.25274693173235

Useful sources

- Explore the fundamental concepts of MLPs by referring to the content available at this link: MLP Features.
- If you find yourself uncertain about choosing an activation function for your neural network and seek in-depth explanations, I recommend consulting the following article: Activation Functions in Neural Networks.
- For a detailed review of the fundation of the Adam optimizer used in this application, referring to Analytics Vidhya's comprehensive guide on deep learning optimizers can be helpful.

Initialization

Prior to utilizing this function, ensure the clearance of all variables, including global ones. This step is crucial, especially when global variables are utilized within the 'DeployMLP_Matlab.m' function, ensuring effective management of nested functions.

```
clc; clear; close all
My_MLP = MLP_builder('initialize')
```

```
My_MLP = struct with fields:
          File_csv_name: 'yourfile.csv'
            ShuffleSeed: 16
            ShuffleRate: 1
              InputSize: 2
             OutputSize: 2
    OutputActivationFun: ''
            HiddenSizes: []
    HiddenActivationFun: ''
           LearningRate: 0.0100
               Momentum: 0.9000
       SumSquaredWeight: 0.9000
              BatchSize: 32
                 Epochs: 10
       OnlineLossesINFO: 1
        FinalPlotLosses: 1
```

```
% Above you can see the default settings that are suppose to be changed
```

Now that the initiaized structure contains all the variables that our MLP need to be fed to set the train. Notice that when you create the MLP_builder, some field are already fullfilled as instances. Remember about the Outputsize must be at most 10 due to the variable-size signals limitations in simulink.

Model setup

File name

```
My_MLP.File_csv_name       =   'yourdataset.csv';
```

Data shuffling

```
My_MLP.ShuffleSeed                =   19;
My_MLP.ShuffleRate                =   1;
```

A priori sizes of input & output layers: ensure they are consistent with your .csv file

```
My_MLP.InputSize      =   uint8(3);
My_MLP.OutputSize     =   uint8(2);
```

Hidden layer(s) dimension: e.g. [4,4] means that there are 2 hidden layers with 4 neurons each.

```
My_MLP.HiddenSizes    =   [2,3];
```

This version allows customization by inserting a different activation function for each layer, including the output layer. The MLP is highly customizable.

Activation Functions list:

```
'sigmoid';
'tanh';
'relu';
'lin';
'softmax';
'step';
'sine';
```

These activation functions provide diverse behaviors suitable for various types of tasks. The choice of activation function depends on the nature of the problem and the architecture of the neural network being employed.

```
My_MLP.HiddenActivationFun    =  {'relu','sine'}
```

```
My_MLP = struct with fields:
          File_csv_name: 'yourdataset.csv'
            ShuffleSeed: 19
            ShuffleRate: 1
              InputSize: 3
             OutputSize: 2
```

```
    OutputActivationFun: ''
            HiddenSizes: [2 3]
    HiddenActivationFun: {'relu'  'sine'}
           LearningRate: 0.0100
               Momentum: 0.9000
      SumSquaredWeight: 0.9000
              BatchSize: 32
                 Epochs: 10
      OnlineLossesINFO: 1
       FinalPlotLosses: 1
```

```
My_MLP.OutputActivationFun   =  'sigmoid';
```

Training settings

```
My_MLP.LearningRate        =   0.007;
My_MLP.Momentum            =   0.9;
My_MLP.SumSquaredWeight    =   0.85;
My_MLP.BatchSize           =   32;
My_MLP.Epochs              =   100;
```

Final settings

```
My_MLP.FinalPlotLosses    = true;
```

Completed structure

```
My_MLP
```

```
My_MLP = struct with fields:
          File_csv_name: 'yourdataset.csv'
            ShuffleSeed: 19
            ShuffleRate: 1
              InputSize: 3
             OutputSize: 2
    OutputActivationFun: 'sigmoid'
            HiddenSizes: [2 3]
    HiddenActivationFun: {'relu'  'sine'}
           LearningRate: 0.0070
               Momentum: 0.9000
      SumSquaredWeight: 0.8500
              BatchSize: 32
                 Epochs: 100
      OnlineLossesINFO: 1
       FinalPlotLosses: 1
```

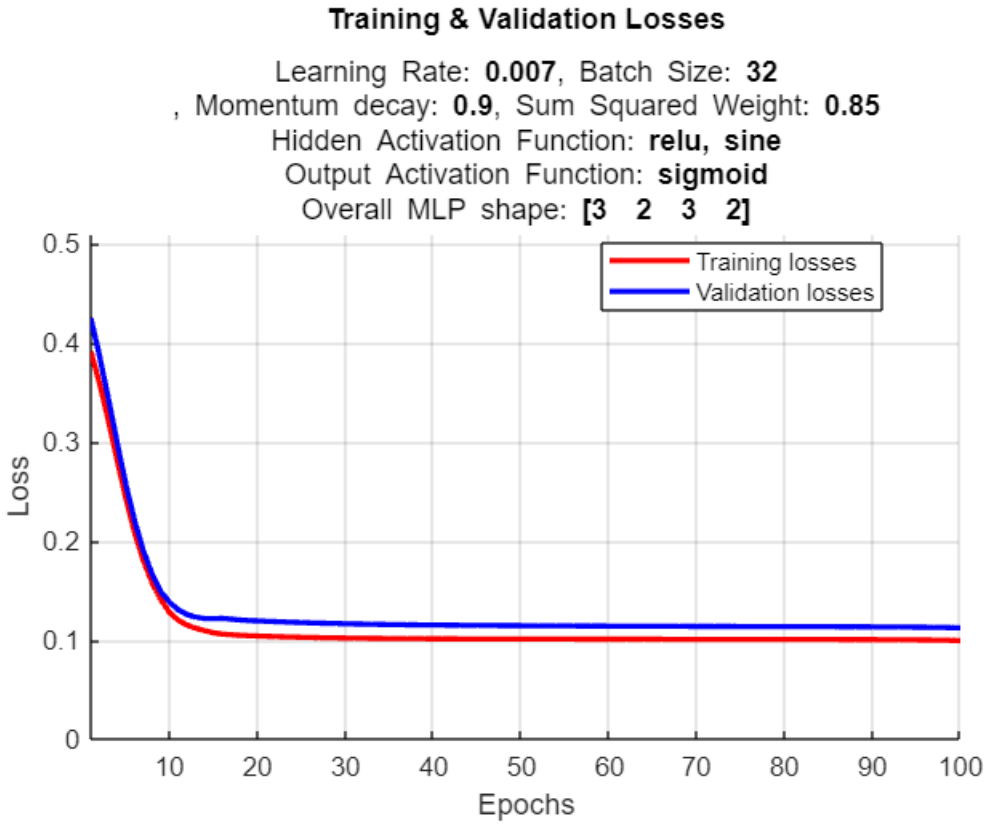
Training

```
MLP_builder('train',My_MLP)
```

```
-------- MatLab preliminary check --------
All parameters are consistent.
The file OnlineLosses.exeis not yet present in the current folder.
File added
-------- Python Processing START --------
```

```
Review of the given specification:
- Input  layer size: 3
- Output layer size: 2
- Number of columns dataset provided: 5
- Input and Output layer sizes are consistent.
- Hidden Layers Activation Function: 'relu', 'sine'
- Output Activation Function: 'sigmoid'
- MLP overall shape:  [3, 2, 3, 2]

---------- MLP: Training and Validation ----------
- Training and validation successfully terminated -
```



Deployment

There are three possibilities regarding the way of deploying data. It is possible to choose the best in according with the purpose of the MLP:

1) Simulink

The deployment in Simulink is useful when the MLP has to work as a controller, for example, or as any other online block. Keep in mind that the function written inside the Simulink Subsystem Reference Block is not the same as the one used in the Matlab function 'DeployMLP_Matlab'. This is because Simulink has a lot of limitations, such as not allowing variable-size signals or cells and structures due to code generation. You can check the following link for a better understanding of Simulink signals: Simulink Data Types.

If you are interested in trying to use cells as input signals in a Simulink function, you may refer to the following guide: Cell Array Restrictions for Code Generation.

```
load MLP_Matlab_data.mat
test_array_sim=0.5*ones(1,decoding_matrix(1,1));
```

```
out=sim("DeployMLP_Simulink.slx");
```

```
One-time calculations in progress...
One-time calculations ended...
Elapsed time is 0.010863 seconds
----------------------------------------
```

```
y_sim=out.Y(:,1,1)'
```

```
y_sim = 1×2
    2.4160    0.0884
```

2) Matlab

On the contrary, in Matlab, things are easier since there are no Simulink limitations. The main code remains the same, but we have fewer rules to follow. For example, we can allow the size of the input and output to vary. Additionally, here, we can process more input arrays through the same function call. In the following instance, a test with 5 inputs in one call is attempted. This can be useful to check the speed of the code compared with the Python one used to deploy data.

In the example below, a large value for N has been chosen arbitrarily to illustrate the functionality of the function. Typically, matrices passed to functions like 'DeployMLP_Matlab' *should not exceed the maximum array size preference, which is set by default to 15.8 GB*.

```
N=10000;
test_array_BIG=0.5*ones(N,decoding_matrix(1,1));
load MLP_Matlab_data.mat
y_mat_fun=DeployMLP_Matlab(test_array_BIG,W_conc,b_conc,scale,decoding_matrix);
```

```
MLP deployment (Matlab)...
One-time calculations in progress...
One-time calculations ended...
----------------------------------------
y_mat_fun = 10000×2
    2.4160    0.0884
    2.4160    0.0884
    2.4160    0.0884
    2.4160    0.0884
    2.4160    0.0884
    2.4160    0.0884
    2.4160    0.0884
    2.4160    0.0884
    2.4160    0.0884
    2.4160    0.0884
     ⋮
Elapsed time matlab function call call: 18.8828 msec
Number of Input processed: 10000
Average elapsed time for each Input: 0.001888 msec
------------------------------------------------
```

3) Python

It is possible to use the mother Python file that has been used to create and train the MLP. In fact, one of the .mat files that results from the MLP_builder() function call, named MLP_Python_data.mat, contains the variables needed to provide the DeployMLP_Python() function with. This allows for performing the deployment directly in Python, and then using Matlab solely to display the results and save the predictions, as if it were a regular Matlab function. This has been done to assess the speed of Python in comparison to Matlab.

```
load MLP_Python_data.mat
y_py=DeployMLP_Python(test_array_BIG,MLP_py_shape,MLP_struct,MLP_py_activation_funct
ions,scale);
```

```
MLP deployment (Python)...
y_py =
[[2.4160 0.0884]
 [2.4160 0.0884]
 [2.4160 0.0884]
 ...
 [2.4160 0.0884]
 [2.4160 0.0884]
 [2.4160 0.0884]]
Elapsed time python file call: 0.0000 msec
Number of Input processed: 10000
Average elapsed time for each Input: 0.0000000 msec
-------------------------------------------------
```

How to create the MLP Simulink block

To generate the MLP Simulink block, you can easily utilize the copy-paste operation within the .slx file. Alternatively, for a more precise approach, you may employ the following function. This function will establish a new Simulink model file containing the desired block. Simply supply the name for the Simulink file, and subsequently save it in the preferred folder. As in the previous cases, this operation must be performed in the folder containing the toolchain.

```
MLP_SimulinkBlock('NewFileWithMLPBlock')
```