

Example of a TMultiLayerPerceptron code

We provide an example of a basic usage of the `TMultiLayerPerceptron` class from a programming package ROOT. A code documentation may be found at: <https://root.cern/doc/master/classTMultiLayerPerceptron.html>.

NETWORK TRAINING

We trained a neural network on the data consisting of two inputs – an effective moderation length λ and the logarithm of a true neutron energy E – and a single output: a logarithm of the resolution function $\mathcal{R}_\lambda$. We will assume that these values have already been prepared as the corresponding arrays. For example:

```
const int N=...;
double lambda[N],log_E[N],log_RF[N];

for (int i=0; i<N; i++) {
    lambda[i]=...;
    log_E[i]=...;
    log_RF[i]=...;
}
```

In order to initialize a `TMultiLayerPerceptron` object, a ROOT-native `TTree` dataset structure must first be declared and filled with the corresponding entries:

```
double x,y,z;

TTree tree("tree","tree");
tree.Branch("x",&x,"x/D");
tree.Branch("y",&y,"y/D");
tree.Branch("z",&z,"z/D");

for (int i=0; i<N; i++) {
    x=lambda[i];
    y=log_E[i];
    z=log_RF[i];
    tree.Fill();
}
```

The branches "x", "y", "z" now contain the inputs and outputs to be used by a `TMultiLayerPerceptron` object. Its declaration, optional setting of the user-specified parameters, starting the training and outputting the results is performed as:

```
TMultiLayerPerceptron mlp("x,y:15:15:15:z",&tree,"1","");
//mlp.SetLearningMethod(TMultiLayerPerceptron::kBFGS);
//mlp.SetDelta(0);
//mlp.SetEpsilon(0);
//mlp.SetTau(3);
//mlp.SetEta(0.1);
//mlp.SetReset(50);
mlp.Train(100,"graph,update=10");
mlp.DumpWeights("NN_weights.dat");
```

Among the most important parameters are the network structure ("x,y:15:15:15:z"), the number of training epoch (100) and the file for outputting the trained network weights ("NN_weights.dat"). The same labels as used in a `TTree` object ("x", "y", "z") must be used within a network structure string. It defines x,y as the input neurons, followed by the intermediate hidden layers structure (15:15:15), ending in the specification of the output neurons (here only z). For the rest of the parameters please refer to a [TMultiLayerPerceptron](#) page.

NETWORK LOADING

Thus trained perceptron is immediately ready to be evaluated as a function of the input parameters **x,y**, but we will demonstrate this on a network freshly initialized from a "NN_weights.dat" file (we assume a completely fresh code, without any of the previous declarations present). Even in this case a **TTree** object must be delivered – even if empty – though its branches do not have to be named as before ("x","y","z"). For generality we rename them here as "a","b","c":

```
double x,y,z;

TTree tree("tree","tree");
tree.Branch("a",&x,"a/D");
tree.Branch("b",&y,"b/D");
tree.Branch("c",&z,"c/D");
```

While declaring a network anew, the same hidden layers structure (15:15:15) must be given as during a training phase, while the inputs and outputs should have the same names as **TTree** branches¹:

```
TMultiLayerPerceptron mlp("a,b:15:15:15:c",&tree,"1","");
```

Loading a trained network is as simple as:

```
mlp.LoadWeights("NN_weights.dat");
```

During a network evaluation the input values are delivered via an array (here **input**). If we wanted to evaluate it over all earlier pairs of **lambda** and **log_E** points, we would use:

```
double input[2],output;

for (int i=0; i<N; i++) {
    input[0]=lambda[i];
    input[1]=log_En[i];

    output=mlp.Evaluate(0,input);
}
```

If the network had more than one output neuron (alongside **c**, or previously **z**), they would be evaluated by requesting a higher index (the first argument) in the **Evaluate** function. For example: **mlp.Evaluate(1,input)**.

¹ If a **TMultiLayerPerceptron** object is to be used only for evaluating a preloaded network – and not for further training – then delivering a “dummy” **TTree** object also seems to work:

```
TTree tree;
TMultiLayerPerceptron mlp("a,b:15:15:15:c",&tree,"1","");
```

The specific input and output labels ("a","b","c") are now completely arbitrary. However, specifying a **TTree** object with appropriate branches is still the safest option.