The Neural Network Objects

Johannes Steffens[1], Marcel Kunze, Helmut Schmücker
Institut für Experimentalphysik 1, Ruhr-Universität Bochum

Neural Network Objects (NNO) is a C++ class library that implements the most popular conventional neural networks together with novel incremental models that have been invented at Bochum University. The package is publicly available and has proven versatile in a broad range of applications over the past years. In the context of the Pico Analysis Framework NNO has now been completely revised in order to take full advantage of the ROOT framework for data management and graphics.

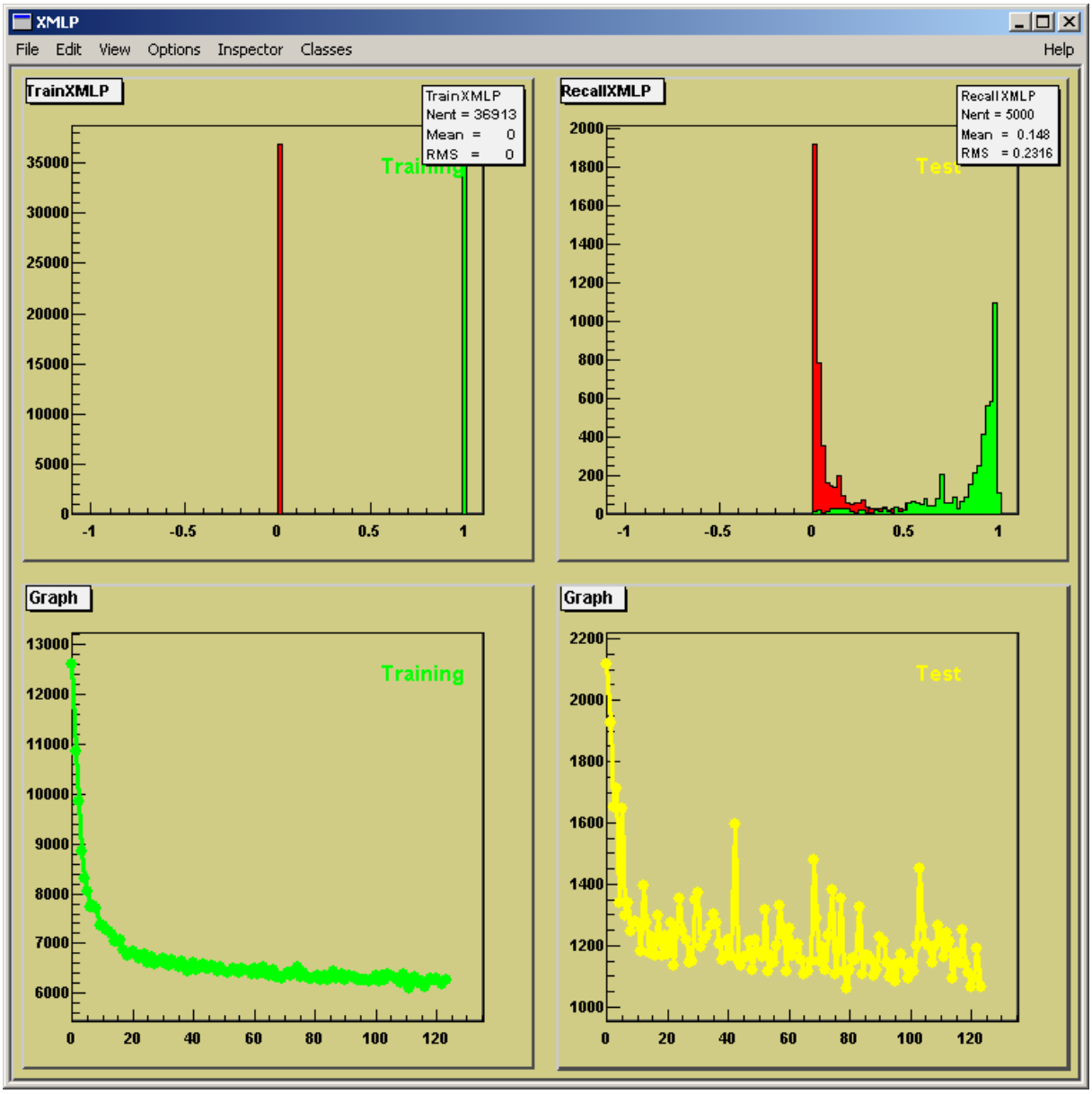

Fig.1: Live updating graphics to control the progress of network training
The example shows the network output for training and test sample (upper row) and the corresponding error function as the training cycles commence.

[1] Now director of software development at Eyematic Interfaces Inc., LA

Architecture

At the time being the package comprises

Supervised Training Models	Multi-Layer Perceptron (TMLP, TXMLP) Fisher Discriminant (TFD) Supervised Growing Cell Structure (TSGCS) Supervised Growing Neural Gas (TSGNG) Neural Network Kernel (TNNK)
Unsupervised Training Models	Learning Vector Quantisation (TLVQ) Growing Cell Structure (TGCS) Growing Neural Gas (TGNG)

The design foresees that all models are derived from the same abstract base class ***VNeuralNet***. The common base class enforces a unique interface to data management, training and recall cycles and graphics operations at one central place. ***VSupervisedNet*** and ***VUnsupervisedNet*** both inherit from ***VNeuralNet*** and take care of the different learning paradigms. In addition specific implementations of the networks can utilize a plotter to produce a live updating graphics window to control the training progress: The abstract ***VNeuralNetPlotter*** interface allows to plug in a graphics engine, like the default **TSimpleNeuralNetPlotter**.

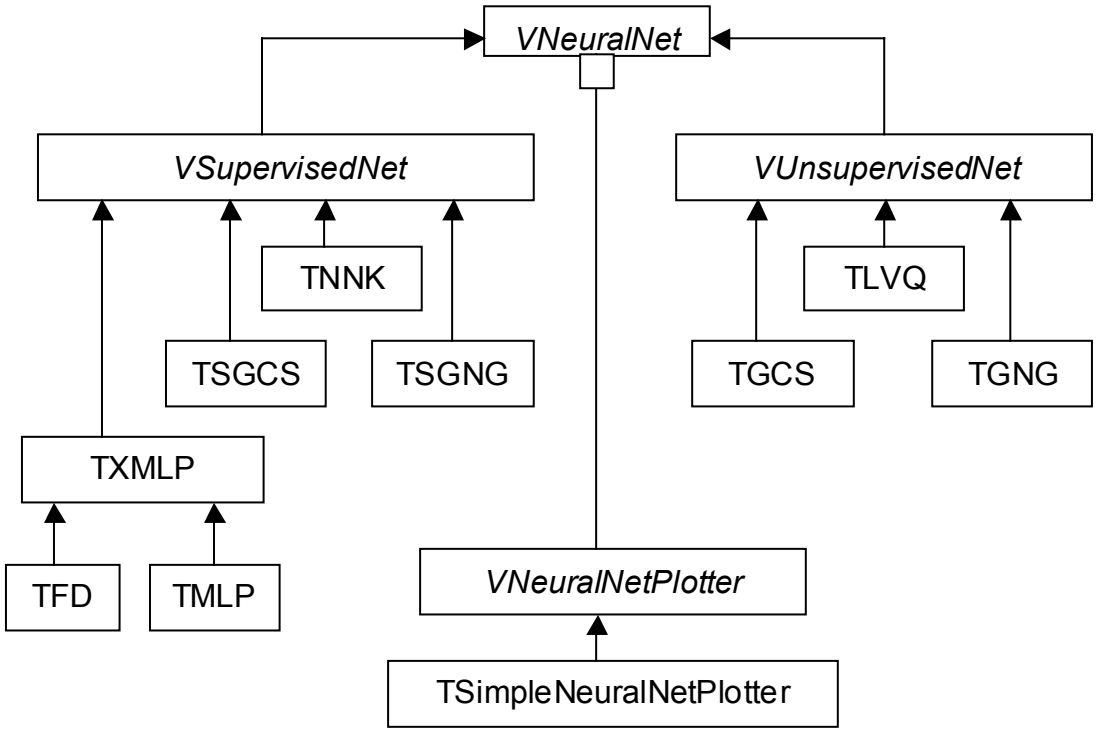

Fig.2: NNO class hierarchy
All networks implement the ***VNeuralNet*** interface and use a ***VNeuralNetPlotter*** to show their training progress.

Implementation

The ***VNeuralNet*** abstract interface defines the following contract for the implementation of specific neural network models:

```
// Abstract interface for all networks
virtual void AllocNet() = 0;
virtual void InitNet() = 0;
virtual void WriteText() = 0;
virtual void WriteBinary() = 0;
virtual void ReadText() = 0;
virtual void ReadBinary() = 0;
virtual Double_t* Recall(NNO_INTYPE* in,NNO_OUTTYPE* out=0) = 0;
virtual Double_t Train(NNO_INTYPE* in,NNO_OUTTYPE* out=0) = 0;
```

AllocNet acquires resources and is executed during network construction. *InitNet* sets up the network weight matrix. *WriteText* persists the network as an ASCII file, *WriteBinary* produces a binary file. The corresponding reading versions are able to regenerate a network in the state it was at the time when it was saved. The *Recall* method takes an input vector as parameter and returns the corresponding network output. The *Train* function takes a pair of input/output vectors, performs a *Recall*, modifies the weight matrix to better adapt the input probability density function and returns the squared error of the sample.

Besides the abstract interface, concrete methods have been implemented to support the execution of training cycles and to set training parameters:

```
// Training and testing
Double_t     TrainEpoch(TDataServe *server, Int_t nEpoch=1);
Double_t     TestEpoch(TDataServe *server);
void         BalanceSamples(Bool_t yesNo = kTRUE);
virtual void SetMomentumTerm(Double_t f);
virtual void SetFlatSpotElimination(Double_t f);
```

TDataServe is a mini database to support management of input/output vector relations. It allows to partition datasets into training and test samples, retrieve arbitrary samples and shuffle a data set prior to a new training cycle. *TrainEpoch* and *TestEpoch* are functions to train and test networks with a complete set of vectors out of a **TDataServe** object. The *BalanceSamples* option allows to have equal training statistics for good and bad samples, independent of the number of vectors per sample. The application of a momentum term might lead to faster convergence in some applications by noting the direction of gradient descent, the flat spot elimination might improve training progress in regions where the derivatives of the error matrix are near zero.

Each network implementation has to implement the abstract interface mentioned above. As an example for the integration of an independent neural network implementation into the context of NNO we have managed to support J.P. Ernenwein's Neural Network Kernel: The ***TNNK*** interface yields seamless access to the Neural Network Kernel in the scope of NNO.

NetworkTrainer

Network training requires to identify pairs of input vectors and output vectors out of a dataset of good and bad samples to describe the problem at hand. It usually takes a certain amount of time to select suiting quantities and assemble corresponding training and test files prior to network training and write a corresponding training program or macro. However, it turns out that ROOT is performing enough to allow for interactive training of large networks with large data samples out of arbitrary ROOT files in one go. In that spirit a *NetworkTrainer* program has been written on the basis of the NNO package. *NetworkTrainer* assists to

- Assemble training and testing data sets out of ROOT trees
- Define the network architecture
- Define a training schedule
- Persist networks
- Generate C++ code to perform network recall

At the time being *NetworkTrainer* reads an ASCII steering file when it launches (a GUI is in preparation). The steering file knows the following directives:

Parameter	***Type*** ***I = input*** ***O = output*** ***H = hidden*** ***C = cells***	***Description***
fisher	vector (I O)	Multi-layer perceptron (0 hidden layer)
mlp	vector (I H O)	Multi-layer perceptron (1 hidden layer)
xmlp	vector (I H H O)	Multi-layer perceptron (2 hidden layers)
tnnk	vector (I H H O)	Multi-layer perceptron (Neural Network Kernel)
sgng	vector (I C O)	Supervised growing neural gas
sgcs	vector (I C O)	Supervised growing cell structures
gng	vector (I C)	Growing neural gas
gcs	vector (I C)	Growing cell structures
lvq	vector (I C)	Learning vector quantization
start	int	First training epoch
stop	int	Last training epoch
epoch	int	Number of training samples per epoch
test	int	Number of test samples per epoch
networkpath	string	Directory to save the trained networks
datapath	string	Directory to look up data files
file	string	ROOT training file containing good and bad samples
pro	string	ROOT training file containing good samples (1D output only)

con	string	ROOT training file containing bad samples (1D output only)
tree	string	ROOT tree that acts as source to assemble the vectors
cut	string	ROOT TFormula for preselection of samples
input	string	Input vector, ROOT TFormulae (separated by colon)
output	string	Output vector, ROOT TFormulae (separated by colon)
transfer	string	Transfer function (TR_FERMI,TR_LINEAR,TR_LINEAR_BEND,TR_SIGMOID)
momentum	float	Momentum term
scale	float	Global scale factor to apply to input layer
inscale	vector	Scale factors to apply to input layer
outscale	vector	Scale factors to apply to output layer
autoscale	bool	Determine scale factors to apply to input layer
plot	bool	Produce graphics output (1D output only)
balance	bool	Enforce presentation of equal number of good and bad samples

A sample steering file for training of a selector to separate different charged particles in a typical HEP experiment could look like the following:

```
# Training of PIDSelectors with NNO

#define the network topology
xmlp 7 15 10 1
transfer TR_FERMI
momentum 0.2
balance true
plots true
test 10000
start 1
stop 200

#define the data source
datapath ../Data
networkpath ../Networks
file PidTuple1.root
file PidTuple2.root

#set up the input layer (use branch names)
tree PidTuple
cut mom>0.5&&dch>0&&dch<10000
input mom:acos(theta):svt:emc:drc:dch:ifr:ifrExp:ifrAdd
autoscale true

#set up the output layer (use branch names)
#Particles pid = {electron=1,muon,pion,kaon,proton}
output abs(pid)==3
```

The example above reads two input files, assembles a data server using all samples surviving the cut and runs for 200 training epochs with a 7-15-10-1 multi-layer perceptron using all available samples. In the course of the training after each epoch a persistent network file

NNOxxxx.TXMLP is saved into the Networks directory, where xxxx denotes the epoch number. At the end, NetworkTrainer produces a template recall function that can be plugged into another program that wants to make use of a network. For the above example the file RecallTXMLP.cpp looks like is shown below for illustration purposes:

```
// TXMLP network trained with NNO NetworkTrainer at Fri Apr 27
// Input parameters mom:acos(theta):svt:emc:drc:dch:ifr:ifrExp:ifrAdd
// Output parameters abs(pid)==3
// Training files:
//../Data/PidTuple1.root
//../Data/PidTuple2.root

#include "PAFNNO/TXMLP.hh"

Double_t* Recall(Double_t *invec)
{
     static TXMLP net("TXMLP.net");
     Float_t x[7];
     x[0]      = 0.76594 *    invec[0]; // mom
     x[1]      = 2.21056 *    invec[1]; // acos(theta)
     x[2]      = 0.20365 *    invec[2]; // svt
     x[3]      = 2.2859  *    invec[3]; // emc
     x[4]      = 1.75435 *    invec[4]; // drc
     x[5]      = 0.00165 *    invec[5]; // dch
     x[6]      = 0.85728 *    invec[6]; // ifr
     return net.Recall(x);
}
```

References

The Neural Network Objects (20 kB), J.Steffens, M.Kunze
A Comparison between the Performance of SGNG and MLP (240 kB), R.Berlich, M.Kunze
Neural Network Application, J.P.Ernenwein
The Pico Analysis Framework, S.Berger, M. Kunze, H.Schmücker