INESC Interim Report

Joana da Hora Martins

Vera Palma

Tutorial on LASCA Code

Work conducted under the scope of project LASCA

Porto, June 2013

Contents

1. Structure

The code includes 4 main folders. Next description presents the content of each folder:

- EPSO Files

  This folder includes files concerning the EPSO evolution in small space: **fitnessfunction.cpp**, **fitnessfunction.h**, **myfitnessHWS.h**, **particle.cpp**, **particle.h**, **swarm.cpp** and **swarm.h**. See a detailed description at epso.inescporto.pt.

- HW Files

  This folder is similar to the above one but for EPSO evolution in original space.

- ITL_networks

  This folder contains the files for training a network. The train can be done by ITL concepts [1] or by backpropagation. A detailed description of this part of code can be found at [1].

- Source Files

  This folder only includes one file, which is the main file of C++ project. All the instructions will be given here (**LASCA_main.cpp**).

- Header Files

  enumerations.h file includes all the enumerations needed over the code.

2. Tutorial on LASCA code

This section includes a description on how to use the LASCA code.

Once you download and install it, please certify that the libraries of armadillo and boost are installed.

The only code file you need to manipulate is the **LASCA_main.cpp**. We assume the reader is familiar with the LASCA theory, alongside with the ITL neural networks and EPSO algorithm, all fully detailed in other reports.

The first option concerns the variable “Train_Global”, where you can choose between the options “ITL” – meaning that the neural networks further applied will be set to an ITL criteria, “PROP” meaning that the NN will follow the classic backpropagation algorithm, and “PCA_train” meaning that no train will be executed, and that the neural networks will be set by PCA weights only:

```
enum GLOBAL_Train Train_Global = ITL; //can vary between ITL or PROP  or PCA_train
```

Next you need to define the initialization method for both the weights and bias from 1st and 2nd halves. For the first half the options available are PCA and random. For the weights of the 2nd half the options are random and transpose. For the bias of both layers, the options are random or zeros:

```
const enum InitializationW1Methods Init_W1_Method= PCA; //RANDOM_W1, PCA
const enum InitializationW2Methods Init_W2_Method = TRANSPOSE;//RANDOM_W2, TRANSPOSE
const enum InitializationB1Methods Init_B1_Method = ZEROS_B1; // RANDOM_B1 or ZEROS_B1
const enum InitializationB2Methods Init_B2_Method = ZEROS_B2; // RANDOM_B2 or ZEROS_B2
```

Next, the choice on the activation functions to use can be made with the variables "NeuronActFunct1" concerning all neurons of second layer, and "NeuronActFunct2" concerning all neurons of 3rd layer. Both have three options: hyperbolic tangent "NEURON_ACT_TANH", logistic "NEURON_ACT_LOGISTIC" and linear "NEURON_ACT_LINEAR":

```
//NEURON_ACT_LOGISTIC, NEURON_ACT_TANH, NEURON_ACT_LINEAR
const enum ActivationFunctions NeuronActFunct1 = NEURON_ACT_TANH;
const enum ActivationFunctions NeuronActFunct2 = NEURON_ACT_LINEAR;
```

Next, the normalization method to apply to data can be defined with the variable "NormMethod", which has the following options: normalization using the min max method for each neuron "ENTRANCE_MIN_MAX", the method using the min max method for all neurons "GLOBAL_MIN_MAX", the Z-score method for each neuron "ENTRANCE_ZSCORE", the Z-score method for all neurons "GLOBAL_ZSCORE" or none:

```
// ENTRANCE_MIN_MAX, GLOBAL_MIN_MAX, ENTRANCE_ZSCORE, GLOBAL_ZSCORE, NONE
const enum NormalizationMethod NormMethod = GLOBAL_MIN_MAX;
```

When using an ITL criterion to train the neural networks, it is necessary to perform an estimation on the bandwidth. Two methods are provided to perform this estimation: the silverman rule [2] "SILVERMAN_RULE" or the rule suggested in [3] "PRINCIPE_RULE":

```
const enum sigmaOption sigmaMethod = SILVERMAN_RULE; //SILVERMAN_RULE or PRINCIPE_RULE
```

The decision where to use or not the Oja's rule is defined through the variable "OjaRule":

```
const enum OjaRule OjaDecision = OjaNO; // OjaYES and OjaNO
```

The number of runs to executed is defined with the variable "NumberSeeds":

```
unsigned int NumberSeeds=10;
```

The standard label to name all output files is defined:

```
string Label_data= ("PROP_");
```

There are other variables which are normally kept unchanged, as they assume the values suggested in literature. These variables are not referred here, since they are already detailed in [1].

The switch on "Train_Global" variable includes the definition on the number of epochs to include and on the values to start the adaptive learning rate. These necessarily assume differentiation depending on the criterion chosen to train the neural network. Therefore, for the method "PROP" the definition includes the variables "numEpochs1" and "learnCoeff1", since in this case the neural network is trained as a whole:

```
switch (Train_Global)
	{
		case PROP:
		{
			numEpochs1 = 2000;
			learnCoeff1.push_back(0.5f);
```

For the case of a ITL criterion, the differentiation on the number of epochs and on the starting values for the adaptive learning coefficients is necessary, since in this case the neural network is trained into two separated parts. Moreover, different training methods are defined train first and second halves. The methods available for the first half, following an unsupervised approach, are maximization of entropy in the hidden layer "ENTROPY", maximization of mutual information between the first and second layers using the Cauchy Schwartz estimator "CAUCHY_SCHWARTZ", also maximizing mutual information but with the Euclidean distance estimator "EUCLIDEAN_DISTANCE", and the minimization of the mutual information between each pair of different neurons composing the hidden layer "UNI":

```
		case ITL:
		{
			numEpochs1 = 2000;
			numEpochs2 = 2000;

			//ENTROPY, CAUCHY_SCHWARTZ, EUCLIDEAN_DISTANCE, UNI
			W1_Method = CAUCHY_SCHWARTZ;
			// W2_PROP,   CIM,   Corr
			W2_Method = W2_PROP;

			learnCoeff1.push_back(0.5f);
			learnCoeff2.push_back(0.05f);
```

The transference of swarms from S to S', and from S' to S, can be performed following two options: the first considers that velocities are also transferred as $V_{ik}^{t} = X_{ik}^{t} - X_{ik}^{t-1}$ since the neural network is only suitable to reproduce the particles "TRANSFER_VEL". The second option considers a random initialization of velocities each time a space transition is made:

```
enum Initi_Vel	Vel_init = RANDOM_VEL;	//TRANSFER_VEL RANDOM_VEL
```

If you are using the code concerning the hydro-wind coordination problem, two options are available: the problem with 8 reservoirs "Reservoirs8", or the problem with 12 reservoir "Reservoirs12". If you are using the code concerning the optimization benchmark functions, a wider range of possibilities is available, and the option made is necessary to include to define not only the "Function_selected" but also "myFf" and "myFfs".

```
enum MATH_FUNCTIONS Function_selected  = F_ShifRastrigin; //F_Rosenbrock F_Sphere F_Alpine  F_Griewank  F_ShifSphere F_ShifRastrigin F_ShifSchwefel
	ShifRastrigin myFf;   //Rosenbrock Sphere  Alpine  Griewank  ShifSphere ShifRastrigin  ShifSchwefel
```

```
enum ProblemSelection ProblemSelect = Reservoirs8; //Reservoirs8, Reservoirs12
```

```
      ShifRastriginS myFfs; //RosenbrockS
SphereS AlpineS GriewankS ShifSphereS
ShifRastriginS ShifSchwefelS
```

Next the number of iterations to include in each part of LASCA algorithm: Part A “maxIterI”, Part B “maxIterS” and Part C “maxIterF”.

```
unsigned int maxIterI=100;
unsigned int maxIterS=20;
unsigned int maxIterF=20;
```

Next, the selection on the number of instances to be stored to train the neural networks. These include train, test and validation datasets:

```
unsigned int trainDim=1000;/*NOTA IMPORTANTE: a alteração do terinoDim PODE IMPLICAR uma
adaptação das var treinoBegin e treinoEnd*/
unsigned int testDim=500;
unsigned int validationDim=500;
```

Specific of optimization functions code

The selection of the dimension composing each layer (only available for the optimization benchmark functions) is made with the variables “DimIn” and “DimHidden”, respectively.

```
unsigned int DimIn = 120, DimHidden = 50;//dimensão da camada interior7
```

The switch over “Function_selected“ allows the definition of the remainder parameters for each function: the boundaries of the function, the values for the EPSO parameters tau and cp, for both original and reduced spaces, and the number of particles.

```
switch (Function_selected)
	{
		case F_Rosenbrock:
		{
			minPosMath = -30.0;
			maxPosMath = 30.0;

			vec_tauG.push_back(0.8);
			vec_cpG.push_back(0.9);

			#ifdef Hybrid_EPSO
				vec_tauP.push_back(0.9);
				vec_cpP.push_back(0.75);
			#endif
			numParticles = 400;
			minimize = true;
			break;
		}
```

Specific of hydro-wind functions code

The number of particles composing each swarm is defined as:

```
unsigned int numParticles=100;
```

Since the hydro-wind problem has a pre-specified dimension on the original space, in this case the only dimension that is changeable is the "DimHidden":

```
switch (ProblemSelect)
	{
		case Reservoirs8:
		{
			numR=8.0f, temp=12.0f;
			DimIn=temp*numR;
			DimHidden=50;//dimensão da camada interior
```

Finally, the definition of the EPSO parameters for both spaces is provided in:

```
//=====================================
//for space S
vector <double> vec_tauG; double TauG;
vector <double> vec_cpG; double CpG;
vec_tauG.push_back(0.9);
vec_cpG.push_back(0.7);
int DimcpG = vec_cpG.size();
int DimtG = vec_tauG.size();

//=====================================
//for hybrid approach
#ifdef Hybrid_EPSO
	vector <double> vec_tauP; double TauP;
	vector <double> vec_cpP; double CpP;
	vec_tauP.push_back(0.8);
	vec_cpP.push_back(1.0);
	int DimtP = vec_tauP.size();
	int DimcpP = vec_cpP.size();
#endif
```


Bibliography

[1] V. Palma e J. Hora, "Theoretical Concepts of ITL Neural Networks," INESC Interim Report, Porto, 2012.

[2] B. W. Silverman, Density Estimation for Statistics and Data Analysis, London: Chapman & Hall/CRC, 1986.

[3] J. C. Principe, Information Theoretic Learning - Renyi's Entropy and Kernel Perspectives, New York: Springer, 2010.