hpelm Documentation

Release 1.0.3

Anton Akusok

March 23, 2016

Contents

HP-ELM is a high performance Neural Network toolbox for solving large problems, especially in Big Data. It supports datasets of any size and GPU acceleration, both with modest memory consumption and fast non-iterative optimization. Train a neural network with 32000 neurons on MNIST dataset in 1 minute on your desktop! *[0]

[0] Neural Network with 32000 sigmoid neurons trained on MNIST training set with 60000 samples, at Intel i7-4790K(4.6GHz), 16GB RAM, GTX Titan Black — 62.8 seconds.

CHAPTER 1

Extreme Learning Machines (ELM)

Extreme Learning Machine is a training algorithm for Single hidden Layer Feed-forward Neural networks (SLFN). It's distinctive feature is random selection of input weights, after which the output weights are computed in one step. The one-step solution provides a huge speedup (> x1000) compared to iterative training algorithms for SLFN like error back-propagation (BP), also known as Multilayer Perceptron (MLP). ELM accuracy with default settings is comparable to MLP, so it is a ready-available fast replacement for MLP for any practical purposes.

ELM follows an opposite "philosophy" to a popular Deep Learning methodology, that aims for the best accuracy in the world in complex tasks at a cost of a very long training time, easily x1,000,000 compared to ELM. Deep Learning also incurs great development costs because it requires highly skilled scientific personnel for model tuning and many man-years of work. While Deep Learning fits best for global challenges like machine translation and self-driving vehicles, ELM is the best model for anything else: prototyping, any low-cost or short-term projects, and for obtaining results on Big Data in reasonable time.

CHAPTER 2

Parallel HP-ELM on multiple machines

HP-ELM uses all cores of a single machine, and is easily run on multiple machines that can access shared storage. See the manual and working example here: *Running HP-ELM in parallel*.

References for ELM

- Main website
- Recent surway
- HPELM toolbox paper

2.1 Running HP-ELM in parallel

An ELM model is very easy to run in parallel. Its solution has two main steps: compute helper matrices HH and HT (99% runtime for large dataset and many hidden neurons) and solve output matrix B from HH and HT (1% runtime). Partial matrices HH^p and HT^p are computed from different parts of input data independently, and then simply summed together: $HH = HH^1 + HH^2 + ... + HH^n$, $HT = HT^1 + HT^2 + ... + HT^n$. The final solution of B cannot be easily split across multiple computers, but it is very fast anyway.

Note: On a single computer HP-ELM already uses all the cores. Parallel HP-ELM takes advantage of distributing work across multiple machines, for instance on a computer cluster.

An example of running HP-ELM in parallel is given below. Separate code blocks are in different files.

1. Put data on a disk in HDF5 format. For example:

```
X = np.random.rand(1000, 10)
T = np.random.rand(1000, 3)
hX = modules.make_hdf5(X, "dataX.hdf5")
hT = modules.make_hdf5(T, "dataT.hdf5")
```

2. Create an HP-ELM model with neurons, and save it to a file.

```
model0 = HPELM(10, 3)
model0.add_neurons(10, 'lin')
model0.add_neurons(5, 'tanh')
model0.add_neurons(15, 'sigm')
model0.save("fmodel.h5")
```

3. Compute partial matrices HH^p, HT^p on different machines in parallel by running different Python scripts. All scripts can read data from the same data files (then you need to set parameters *istart* and *icount* that specify where to start reading data and how many rows to read). Scripts can also read data from separate files which you have prepared and distributed, or even from the given Numpy matrices (not sure about that :).

   All scripts write their partial matrices HH^p, HT^p to the same files on disk, incrementing existing data in these files. Writes are multiprocess-safe using file locks (from *fasteners* library). HP-ELM will create starting files with zero matrices HH, HT for you if they don't exist yet.

   Note: The folder where HH, HT files are located must be writable by all parallel scripts, because they use auxiliary files as write locks.

```
model1 = HPELM(10, 3)
model1.load("model.pkl")
model1.add_data("dataX.hdf5", "dataT.hdf5", istart=0, icount=100, fHH="HH.hdf5", fHT="HT.hdf5")
```

```
model2 = HPELM(10, 3)
model2.load("model.pkl")
model2.add_data("dataX.hdf5", "dataT.hdf5", istart=100, icount=100, fHH="HH.hdf5", fHT="HT.hdf5"
```

```
model3 = HPELM(10, 3)
model3.load("model.pkl")
model3.add_data("dataX.hdf5", "dataT.hdf5", istart=200, icount=800, fHH="HH.hdf5", fHT="HT.hdf5"
```

4. Run final solution step on one machine, and save the trained model. You can then get your predictions.

```
model4 = HPELM(10, 3)
model4.load("model.pkl")
model4.solve_corr("HH.hdf5", "HT.hdf5")
model4.save("model.pkl")
```

```
model5 = HPELM(10, 3)
model5.load("model.pkl")
model5.predict("dataX.hdf5", "predictedY.hdf5")
err_train = model5.error("dataX.hdf5", "predictedY.hdf5")
print "Training error is", err_train
```

2.2 Library Reference

Description of classes and methods in HP-ELM toolbox.

2.2.1 ELM

Created on Mon Oct 27 17:48:33 2014

@author: akusok

class hpelm.elm.**ELM**(*inputs*, *outputs*, *classification=''*, *w=None*, *batch=1000*, *accelerator=None*, *precision='double'*, *norm=None*, *tprint=5*)

Bases: object

Interface for training Extreme Learning Machines (ELM).

Parameters

- **inputs** (*int*) – dimensionality of input data, or number of data features
- **outputs** (*int*) – dimensionality of output data, or number of classes
- **classification** (*'c'/'wc'/'ml', optional*) – train ELM for classfication ('c') / weighted classification ('wc') / multi-label classification ('ml'). For weighted classification you can provide weights in *w*. ELM will compute and use the corresponding classification error instead of Mean Squared Error.
- **w** (*vector, optional*) – weights vector for weighted classification, lenght (*outputs* * 1).
- **batch** (*int, optional*) – batch size for data processing in ELM, reduces memory requirements. Does not work for model structure selection (validation, cross-validation, Leave-One-Out). Can be changed later directly as a class attribute.
- **accelerator** (*"GPU"/"basic", optional*) – type of accelerated ELM to use: None, 'GPU', 'basic', ...
- **precision** (*optional*) – data precision to use, supports signle ('single', '32' or numpy.float32) or double ('double', '64' or numpy.float64). Single precision is faster but may cause numerical errors. Majority of GPUs work in single precision. Default: **double**.
- **norm** (*double, optinal*) – L2-normalization parameter, **None** gives the default value.
- **tprint** (*int, optional*) – ELM reports its progess every *tprint* seconds or after every batch, whatever takes longer.

Class attributes; attributes that simply store initialization or *train()* parameters are omitted.

nnet
object

Implementation of neural network with computational methods, but without complex logic. Different implementations are given by different classes: for Python, for GPU, etc. See `hpelm.nnets` folder for particular files. You can implement your own computational algorithm by inheriting from `hpelm.nnets.SLFN` and overwriting some methods.

flist
list of strings

Available types of neurons, use them when adding new neurons.

Note: Below the 'matrix' type means a 2-dimensional Numpy.ndarray.

---

add_data(*X*, *T*)
Feed new training data (X,T) to ELM model in batches; does not solve ELM itself.

Helper method that updates intermediate solution parameters HH and HT, which are used for solving ELM later. Updates accumulate, so this method can be called multiple times with different parts of training data. To reset accumulated training data, use *ELM.nnet.reset()*.

For training an ELM use *ELM.train()* instead.

Parameters

- **X** (*matrix*) – input training data
- **T** (*matrix*) – output training data

add_neurons(*number*, *func*, *W=None*, *B=None*)
Adds neurons to ELM model. ELM is created empty, and needs some neurons to work.

Add neurons to an empty ELM model, or add more neurons to a model that already has some.

Random weights *W* and biases *B* are generated automatically if not provided explicitly. Maximum number of neurons is limited by the available RAM and computational power, a sensible limit would be 1000 neurons for an average size dataset and 15000 for the largest datasets. ELM becomes slower after 3000 neurons because computational complexity is proportional to a qube of number of neurons.

This method checks and prepares neurons, they are actually stored in *solver* object.

Parameters

- **number** (*int*) – number of neurons to add
- **func** (*string*) – type of neurons: "lin" for linear, "sigm" or "tanh" for non-linear, "rbf_l1", "rbf_l2" or "rbf_linf" for radial basis function neurons.
- **W** (*matrix, optional*) – random projection matrix size (*inputs* * *number*). For '**rbf_**' neurons, W stores centroids of radial basis functions in transposed form.
- **B** (*vector, optional*) – bias vector of size (*number* * 1), a 1-dimensional Numpy.ndarray object. For '**rbf_**' neurons, B gives widths of radial basis functions.

confusion(*T*, *Y*)

Computes confusion matrix for classification.

Confusion matrix C such that element $C_{i,j}$ equals to the number of observations known to be class i but predicted to be class j.

Parameters

- **T** (*matrix*) – true outputs or classes, size (N * *outputs*)
- **Y** (*matrix*) – predicted outputs by ELM model, size (N * *outputs*)

Returns **conf** – confusion matrix, size (*outputs* * *outputs*)

Return type matrix

error(*T*, *Y*)

Calculate error of model predictions.

Computes Mean Squared Error (MSE) between model predictions Y and true outputs T. For classification, computes mis-classification error. For multi-label classification, correct classes are all with Y>0.5.

For weighted classification the error is an average weighted True Positive Rate, or percentage of correctly predicted samples for each class, multiplied by weight of that class and averaged. If you want something else, just write it yourself :) See https://en.wikipedia.org/wiki/Confusion_matrix for details.

Another option is to use scikit-learn's performance metrics. Transform *Y* and *T* into scikit's format by `y_true = T.argmax[1]`, `y_pred = Y.argmax[1]`. http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics

Parameters

- **T** (*matrix*) – true outputs.
- **Y** (*matrix*) – ELM model predictions, can be computed with *predict()* function.

Returns **e** – MSE for regression / classification error for classification.

Return type double

load(*fname*)

Load ELM model data from a file.

Load requires an `ELM` object, and it uses solver type, precision and batch size from that ELM object.

Parameters **fname** (*string*) – filename to load model from.

predict(*X*)

Predict outputs Y for the given input data X.

Parameters **X** (*matrix*) – input data of size (N * *inputs*)

Returns **Y** – output data or predicted classes, size (N * *outputs*).

Return type matrix

project(*X*)

Get ELM's hidden layer representation of input data.

Parameters **X** (*matrix*) – input data, size (N * *inputs*)

Returns **H** – hidden layer representation matrix, size (N * number_of_neurons)

Return type matrix

save(*fname*)

Save ELM model with current parameters.

Model does not save a particular solver, precision batch size. They are obtained from a new ELM when loading the model (so one can switch to another solver, for instance).

Also ranking and max number of neurons are not saved, because they are runtime training info irrelevant after the training completes.

Parameters **fname** (*string*) – filename to save model into.

train(*X*, *T*, **args*, ***kwargs*)

Universal training interface for ELM model with model structure selection.

Model structure selection takes more time and requires all data to fit into memory. Optimal pruning ('OP', effectively an L1-regularization) takes the most time but gives the smallest and best performing model. Choosing a classification forces ELM to use classification error in model structure selection, and in *error()* method output.

Parameters

- **X** (*matrix*) – input data matrix, size (N * *inputs*)
- **T** (*matrix*) – outputs data matrix, size (N * *outputs*)
- **'V'/'CV'/'LOO'** (*sting, choose one*) – model structure selection: select optimal number of neurons using a validation set ('V'), cross-validation ('CV') or Leave-One-Out ('LOO')
- **'OP'** (*string, use with 'V'/'CV'/'LOO'*) – choose best neurons instead of random ones, training takes longer; equivalent to L1-regularization
- **'c'/'wc'/'ml'/'r'** (*string, choose one*) – train ELM for classification ('c'), classification with weighted classes ('wc'), multi-label classification ('ml') with several correct classes per data sample, or regression ('r') without any classification. In classification, number of *outputs* is the number of classes; correct class(es) for each sample has value 1 and incorrect classes have 0. Overwrites parameters given an ELM initialization time.

Keyword Arguments

- **Xv** (*matrix, use with 'V'*) – validation set input data, size (Nv * *inputs*)
- **Tv** (*matrix, use with 'V'*) – validation set outputs data, size (Nv * *outputs*)
- **k** (*int, use with 'CV'*) – number of splits for cross-validation, k>=3

- **kmax** (*int, optional, use with 'OP'*) – maximum number of neurons to keep in ELM
- **batch** (*int, optional*) – batch size for ELM, overwrites batch size from the initialization

Returns

e – test error for cross-validation, computed from one separate test chunk in each

split of data during the cross-validation procedure

Return type double, for 'CV'

2.2.2 HPELM

Created on Mon Oct 27 17:48:33 2014

@author: akusok

class hpelm.hp_elm.**HPELM**(*inputs*, *outputs*, *classification=''*, *w=None*, *batch=1000*, *accelerator=None*, *precision='double'*, *norm=None*, *tprint=5*)

Bases: hpelm.elm.ELM

Interface for training High-Performance Extreme Learning Machines (HP-ELM).

Parameters

- **inputs** (*int*) – dimensionality of input data, or number of data features
- **outputs** (*int*) – dimensionality of output data, or number of classes
- **classification** (*'c'/'wc'/'ml', optional*) – train ELM for classfication ('c') / weighted classification ('wc') / multi-label classification ('ml'). For weighted classification you can provide weights in *w*. ELM will compute and use the corresponding classification error instead of Mean Squared Error.
- **w** (*vector, optional*) – weights vector for weighted classification, lenght (*outputs* * 1).
- **batch** (*int, optional*) – batch size for data processing in ELM, reduces memory requirements. Does not work for model structure selection (validation, cross-validation, Leave-One-Out). Can be changed later directly as a class attribute.
- **accelerator** (*string, optional*) – type of accelerated ELM to use: None, 'GPU', ...
- **precision** (*optional*) – data precision to use, supports signle ('single', '32' or numpy.float32) or double ('double', '64' or numpy.float64). Single precision is faster but may cause numerical errors. Majority of GPUs work in single precision. Default: **double**.
- **norm** (*double, optinal*) – L2-normalization parameter, **None** gives the default value.
- **tprint** (*int, optional*) – ELM reports its progess every *tprint* seconds or after every batch, whatever takes longer.

Class attributes; attributes that simply store initialization or *train()* parameters are omitted.

nnet

object

Implementation of neural network with computational methods, but without complex logic. Different implementations are given by different classes: for Python, for GPU, etc. See hpelm.nnets folder for particular files. You can implement your own computational algorithm by inheriting from hpelm.nnets.SLFN and overwriting some methods.

flist
list of strings

Available types of neurons, use them when adding new neurons.

Note: The 'hdf5' type denotes a name of HDF5 file type with a single 2-dimensional array inside. HPELM uses PyTables interface to HDF5: http://www.pytables.org/. For HDF5 array examples, see http://www.pytables.org/usersguide/libref/homogenous_storage.html. Array name is irrelevant, but there must be **only one array per HDF5 file**.

A 2-dimensional Numpy.ndarray can also be used.

---

add_data (*fX*, *fT*, *istart=0*, *icount=inf*, *fHH=None*, *fHT=None*)
Feed new training data (X,T) to HP-ELM model in batches: does not solve ELM itself.

This method prepares an intermediate solution data, that takes the most time. After that, obtaining the solution is fast.

The intermediate solution consists of two matrices: *HH* and *HT*. They can be in memory for a model computed at once, or stored on disk for a model computed in parts or in parallel.

For iterative solution, provide file names for on-disk matrices in the input parameters *fHH* and *fHT*. They will be created if they don't exist, or new results will be merged with the existing ones. This method is multiprocess-safe for parallel writing into files *fHH* and *fHT*, that allows you to easily compute ELM in parallel. The multiprocess-safeness uses Python module 'fasteners' and a lock file, which is named fHH+'.lock' and fHT+'.lock'.

Parameters

- **fX** (*hdf5*) – (part of) input training data size (N * *inputs*)
- **fT** (*hdf5*) –
- **istart** (*int, optional*) – index of first data sample to use from *fX*, *istart* < N. If not given, all data from *fX* is used. Sample with index *istart* is used for training, indexing is 0-based.
- **icount** (*int, optional*) – number of data samples to use from *fX*, starting from *istart*, automatically adjusted to *istart* + *icount* <= N. If not given, all data starting from *start* is used. The last sample used for training is *istart*'+'*icount*-1, so you can index data as: istart_1=0, icount_1=1000; istart_2=1000, icount_2=1000; istart_3=2000, icount_3=1000, ...
- **fHT** (*fHH,*) – file names for storing HH and HT matrices. Files are created if they don't exist, or new result is added to the existing files if they exist. Parallel writing to the same *fHH*, *fHT* files is multiprocess-safe, made specially for parallel training of HP-ELM. Another use is to split a very long training of huge ELM into smaller parts, so the training can be interrupted and resumed later.

add_data_async (*fX*, *fT*, *istart=0*, *icount=inf*, *fHH=None*, *fHT=None*)
Version of *add_data()* with asyncronous I/O. See *add_data()* for reference.

Spawns new processes using Python's *multiprocessing* module, and requires more memory than non-async version.

error (*fT*, *fY*, *istart=0*, *icount=inf*)
Calculate error of model predictions of HPELM.

Computes Mean Squared Error (MSE) between model predictions Y and true outputs T. For classification, computes mis-classification error. For multi-label classification, correct classes are all with Y>0.5.

For weighted classification the error is an average weighted True Positive Rate, or percentage of correctly predicted samples for each class, multiplied by weight of that class and averaged. If you want something else, just write it yourself :) See https://en.wikipedia.org/wiki/Confusion_matrix for details.

Parameters

- **fT** (*hdf5*) – hdf5 filename with true outputs
- **fY** (*hdf5*) – hdf5 filename with predicted outputs
- **istart** (*int, optional*) – index of first data sample to use from *fX*, *istart* < N. If not given, all data from *fX* is used. Sample with index *istart* is used for training, indexing is 0-based.
- **icount** (*int, optional*) – number of data samples to use from *fX*, starting from *istart*, automatically adjusted to *istart* + *icount* <= N. If not given, all data starting from *start* is used. The last sample used for training is *istart*'+ '*icount*-1, so you can index data as: istart_1=0, icount_1=1000; istart_2=1000, icount_2=1000; istart_3=2000, icount_3=1000, ...

Returns **e** – MSE for regression / classification error for classification.

Return type double

predict (*fX*, *fY=None*, *istart=0*, *icount=inf*)

Iterative predict outputs and save them to HDF5, can use custom range.

Parameters

- **fX** (*hdf5*) – hdf5 filename or Numpy matrix with input data from which outputs are predicted
- **fY** (*hdf5*) – hdf5 filename or Numpy matrix to store output data into, if 'None' then Numpy matrix is generated automatically.
- **istart** (*int, optional*) – index of first data sample to use from *fX*, *istart* < N. If not given, all data from *fX* is used. Sample with index *istart* is used for training, indexing is 0-based.
- **icount** (*int, optional*) – number of data samples to use from *fX*, starting from *istart*, automatically adjusted to *istart* + *icount* <= N. If not given, all data starting from *start* is used. The last sample used for training is *istart*'+ '*icount*-1, so you can index data as: istart_1=0, icount_1=1000; istart_2=1000, icount_2=1000; istart_3=2000, icount_3=1000, ...

predict_async (*fX*, *fY*, *istart=0*, *icount=inf*)

Version of *predict()* with asyncronous I/O. See *predict()* for reference.

Spawns new processes using Python's *multiprocessing* module, and requires more memory than non-async version.

project (*fX*, *fH=None*, *istart=0*, *icount=inf*)

Iteratively project input data from HDF5 into HPELM hidden layer, and save in another HDF5.

Parameters

- **fX** (*hdf5*) – hdf5 filename or Numpy matrix with input data to project
- **fH** (*hdf5*) – hdf5 filename or Numpy matrix to store projected inputs, if 'None' then Numpy matrix is generated automatically.
- **istart** (*int, optional*) – index of first data sample to use from *fX*, *istart* < N. If not given, all data from *fX* is used. Sample with index *istart* is used for training, indexing is 0-based.

- **icount** (*int, optional*) – number of data samples to use from *fX*, starting from *istart*, automatically adjusted to *istart* + *icount* <= N. If not given, all data starting from *start* is used. The last sample used for training is *istart*'+'*icount*-1, so you can index data as: istart_1=0, icount_1=1000; istart_2=1000, icount_2=1000; istart_3=2000, icount_3=1000, ...

solve_corr (*fHH*, *fHT*)
Solves an ELM model with the given (covariance) fHH and (correlation) fHT HDF5 files.

Parameters

- **fHH** (*hdf5*) – an hdf5 file with intermediate solution data
- **fHT** (*hdf5*) – an hdf5 file with intermediate solution data

train (*fX*, *fT*, **args*, ***kwargs*)
Universal training interface for HP-ELM model.

Always trains a basic ELM model without model structure selection. L2-regularization is available as *norm* parameter at HPELM initialization. Number of neurons selection with validation set for trained HPELM is available in *train_hpv()* method.

Parameters

- **fX** (*hdf5*) – input data on disk, size (N * *inputs*)
- **fT** (*hdf5*) – outputs data on disk, size (N * *outputs*)
- **'c'/'wc'/'ml'** (*string, choose one*) – train HPELM for classification ('c'), classification with weighted classes ('wc') or multi-label classification ('ml') with several correct classes per data sample. In classification, number of *outputs* is the number of classes; correct class(es) for each sample has value 1 and incorrect classes have 0.

Keyword Arguments

- **istart** (*int, optional*) – index of first data sample to use from *fX*, *istart* < N. If not given, all data from *fX* is used. Sample with index *istart* is used for training, indexing is 0-based.
- **icount** (*int, optional*) – number of data samples to use from *fX*, starting from *istart*, automatically adjusted to *istart* + *icount* <= N. If not given, all data starting from *start* is used. The last sample used for training is *istart*'+'*icount*-1, so you can index data as: istart_1=0, icount_1=1000; istart_2=1000, icount_2=1000; istart_3=2000, icount_3=1000, ...
- **batch** (*int, optional*) – batch size for ELM, overwrites batch size from the initialization

train_async (*fX*, *fT*, **args*, ***kwargs*)
Training HPELM with asyncronous I/O, good for network drives, etc. See *train()* for reference.

Spawns new processes using Python's *multiprocessing* module.

validation_corr (*fHH*, *fHT*, *fXv*, *fTv*, *steps=10*)
Quick batch error evaluation with different numbers of neurons on a validation set.

Only feasible implementation of model structure selection with HP-ELM. This method makes a single pass over the validation data, computing errors for all numbers of neurons at once. It requires HDF5 files with matrices HH and HT: *fHH* and *fHT*, obtained from *add_data(..., fHH, fHT)* method.

The method writes the best solution to the HPELM model.

Parameters

- **fHH** (*string*) – name of HDF5 file with HH matrix
- **fHT** (*string*) – name of HDF5 file with HT matrix

- **fXv** (*string*) – name of HDF5 file with validation dataset inputs
- **fTv** (*string*) – name of HDF5 file with validation dataset outputs
- **steps** (*int or vector*) – amount of different numbers of neurons to test, choosen uniformly on a logarithmic scale from 3 to number of neurons in HPELM. Can be given exactly as a vector.

Returns

Ls – numbers of neurons used by *validation_corr()* method errs (vector): corresponding errors for number of neurons in *Ls*, with classification error if model

is run for classification

confs (list of matrix): list of confusion matrices corresponding to elements in Ls (empty for regression)

Return type vector

2.2.3 SLFN Solvers

Background solvers for Single Layer Feed-forward Network (SLFN) that do all heavy-lifting computations, a separate solver accelerates computations for separate hardware (GPU, etc.). Interface is defined by `SLFN` class.

Use different solvers by passing optional parameter `accelerator` to `ELM` or `HPELM`. Basic SLFN solver that follows paper notations, defines interface for all solvers.

Created on Sun Sep 6 11:18:55 2015 @author: akusok

class `hpelm.nnets.slfn.`**SLFN**(*inputs*, *outputs*, *norm=None*, *precision=<type 'numpy.float64'>*)

Bases: `object`

Single Layer Feed-forward Network (SLFN), the neural network that ELM trains.

This implementation is not the fastest but very simple, and it defines interface. Gives correct output, other solvers should provide the same output as this guy.

Parameters

- **outputs** (*int*) – number of outputs, or classes for classification
- **norm** (*double*) – output weights normalization parameter (Tikhonov normalizaion, or ridge regression), large values provides smaller (= better) weights but worse model accuracy
- **precision** (*Numpy.float32/64*) – solver precision, float32 is faster but may be worse, most GPU work fast only in float32.

neurons
list

a list of different types of neurons, initially empty. One neuron type is a tuple ('number of neurons', 'function_type', W, Bias), *neurons* is a list of [neuron_type_1, neuron_type_2, ...].

func
dict

a dictionary of transformation function type, key is a neuron type (= function name) and value is the function itself. A single function takes input parameters X, W, B, and outputs corresponding H for its neuron type.

HH, HT
matrix

intermediate covariance matrices used in ELM solution. Can be computed and stored in GPU memory for accelerated SLFN. They are not needed once ELM is solved and they can take a lot of memory with large numbers of neurons, so one can delete them with *reset()* method. They are omitted when an ELM model is saved.

B

matrix

output solution matrix of SLFN. A trained ELM needs only *neurons* and *B* to predict outputs for new input data.

add_batch(*X*, *T*, *wc=None*)

Add a batch of training data to an iterative solution, weighted if neeed.

The batch is processed as a whole, the training data is splitted in *ELM.add_data()* method. With parameters HH_out, HT_out, the output will be put into these matrices instead of model.

Parameters

- **X** (*matrix*) – input data matrix size (N * *inputs*)
- **T** (*matrix*) – output data matrix size (N * *outputs*)
- **wc** (*vector*) – vector of weights for data samples, one weight per sample, size (N * 1)
- **HT_out** (*HH_out,*) – output matrices to add batch result into, always given together

add_neurons(*number*, *func*, *W*, *B*)

Add prepared neurons to the SLFN, merge with existing ones.

Adds a number of specific neurons to SLFN network. Weights and biases must be provided for that function.

If neurons of such type already exist, they are merged together.

Parameters

- **number** (*int*) – the number of new neurons to add
- **func** (*str*) – transformation function of hidden layer. Linear function creates a linear model.
- **W** (*matrix*) – a 2-D matrix of neuron weights, size (*inputs* * *number*)
- **B** (*vector*) – a 1-D vector of neuron biases, size (*number* * 1)

fix_affinity()

Numpy processor core affinity fix.

Fixes a problem if all Numpy processes are pushed to CPU core 0.

get_B()

Return B as a numpy array.

get_corr()

Return current correlation matrices.

get_neurons()

Return current neurons.

Returns **neurons** – current neurons in the model

Return type list of tuples (number/int, func/string, W/matrix, B/vector

reset()
Resets intermediate training results, releases memory that they use.

Keeps solution of ELM, so a trained ELM remains operational. Can be called to free memory after an ELM is trained.

set_B(*B*)
Set B as a numpy array.

Parameters **B** (*matrix*) – output layer weights matrix, size (*L* * *outputs*)

set_corr(*HH*, *HT*)
Set pre-computed correlation matrices.

Parameters

- **HH** (*matrix*) – covariance matrix of hidden layer represenation H, size (*L* * *L*)
- **HT** (*matrix*) – correlation matrix between H and outputs T, size (*L* * *outputs*)

solve()
Redirects to solve_corr, to avoid duplication of code.

solve_corr(*HH*, *HT*)
Compute output weights B for given HH and HT.

Simple but inefficient version, see a better one in solver_python.

Parameters

- **HH** (*matrix*) – covariance matrix of hidden layer represenation H, size (*L* * *L*)
- **HT** (*matrix*) – correlation matrix between H and outputs T, size (*L* * *outputs*)

This is a fast Python implementation of SLFN.

Created on Sun Sep 6 11:18:55 2015 @author: akusok

class hpelm.nnets.slfn_python.**SLFNPython**(*inputs*, *outputs*, *norm=None*, *precision=<type 'numpy.float64'>*)
Bases: hpelm.nnets.slfn.SLFN

Single Layer Feed-forward Network (SLFN), the neural network that ELM trains.

add_batch(*X*, *T*, *wc=None*)
Add a batch using Symmetric Rank-K matrix update for HH.

get_corr()
Return current correlation matrices.

solve_corr(*HH*, *HT*)
Compute output weights B for given HH and HT.

Simple but inefficient version, see a better one in solver_python.

Parameters

- **HH** (*matrix*) – covariance matrix of hidden layer represenation H, size (*L* * *L*)
- **HT** (*matrix*) – correlation matrix between H and outputs T, size (*L* * *outputs*)

Nvidia GPU-accelerated solver based on Scikit-CUDA, works without compiling anything.

GPU computations run in asyncronous mode: GPU is processing one batch of data while CPU prepares the next batch. Loads GPU for 100% without waiting times, very fast and efficient. The requied Scikit-CUDA is a single-line install in Linux: `pip install scikit-cuda`. Tested on CUDA 7.

Created on Sat Sep 12 13:10:23 2015 @author: akusok

class hpelm.nnets.slfn_skcuda.**SLFNSkCUDA**(*inputs*, *outputs*, *norm=None*, *precision=<type 'numpy.float64'>*)

Bases: hpelm.nnets.slfn.SLFN

Single Layer Feed-forward Network (SLFN) implementation on GPU with pyCUDA.

To choose a specific GPU, use environmental variable `CUDA_DEVICE`, for exampe `CUDA_DEVICE=0 python myscript1.py & CUDA_DEVICE=1 python myscript2.py`.

In single precision, only upper triangular part of HH matrix is computed to speedup the method.

add_batch(*X*, *T*, *wc=None*)

Add a batch of training data to an iterative solution, weighted if neeed.

The batch is processed as a whole, the training data is splitted in *ELM.add_data()* method. With parameters HH_out, HT_out, the output will be put into these matrices instead of model.

Parameters

- **X** (*matrix*) – input data matrix size (N * *inputs*)
- **T** (*matrix*) – output data matrix size (N * *outputs*)
- **wc** (*vector*) – vector of weights for data samples, one weight per sample, size (N * 1)
- **HT_out** (*HH_out,*) – output matrices to add batch result into, always given together

add_neurons(*number*, *func*, *W*, *B*)

Add prepared neurons to the SLFN, merge with existing ones.

Adds a number of specific neurons to SLFN network. Weights and biases must be provided for that function.

If neurons of such type already exist, they are merged together.

Parameters

- **number** (*int*) – the number of new neurons to add
- **func** (*str*) – transformation function of hidden layer. Linear function creates a linear model.
- **W** (*matrix*) – a 2-D matrix of neuron weights, size (*inputs* * *number*)
- **B** (*vector*) – a 1-D vector of neuron biases, size (*number* * 1)

get_B()

Return B as a numpy array.

get_corr()

Return current correlation matrices.

get_neurons()

Return current neurons.

Returns neurons – current neurons in the model

Return type list of tuples (number/int, func/string, W/matrix, B/vector

reset()

Resets intermediate training results, releases memory that they use.

Keeps solution of ELM, so a trained ELM remains operational. Can be called to free memory after an ELM is trained.

set_B(*B*)

Set B as a numpy array.

Parameters **B** (*matrix*) – output layer weights matrix, size (*L* * *outputs*)

set_corr(*HH*, *HT*)
Set pre-computed correlation matrices.

Parameters

- **HH** (*matrix*) – covariance matrix of hidden layer represenation H, size (*L* * *L*)
- **HT** (*matrix*) – correlation matrix between H and outputs T, size (*L* * *outputs*)

solve()
Compute output weights B, with fix for unstable solution.

solve_corr(*HH*, *HT*)
Compute output weights B for given HH and HT.

Simple but inefficient version, see a better one in solver_python.

Parameters

- **HH** (*matrix*) – covariance matrix of hidden layer represenation H, size (*L* * *L*)
- **HT** (*matrix*) – correlation matrix between H and outputs T, size (*L* * *outputs*)

2.2.4 Model Structure Selection

These modules detect an optimal number of neurons in ELM, and remove extra unnecessary neurons. They are used in `ELM.train()` if you give optional flag `V` (validation), `CV` (cross-validation) or `LOO` (Leave-One-Out validation).
Created on Mon Oct 27 17:48:33 2014

@author: akusok

hpelm.mss_v.**train_v**(*self*, *X*, *T*, *Xv*, *Tv*)
Model structure selection with a validation set.

Trains ELM, validates model and sets an optimal validated solution.

Parameters

- **self** (*ELM*) – ELM object that calls *train_v()*
- **X** (*matrix*) – training set inputs
- **T** (*matrix*) – training set outputs
- **Xv** (*matrix*) – validation set inputs
- **Tv** (*matrix*) – validation set outputs

Created on Mon Oct 27 17:48:33 2014

@author: akusok

hpelm.mss_cv.**train_cv**(*self*, *X*, *T*, *k*)
Model structure selection with cross-validation.

Trains ELM, cross-validates model and sets an optimal validated solution.

Parameters

- **self** (*ELM*) – ELM object that calls *train_v()*
- **X** (*matrix*) – training set inputs
- **T** (*matrix*) – training set outputs

- **k** (*int*) – number of parts to split the dataset into, k-2 parts are used for training and 2 parts are left out: 1 for validation and 1 for test; repeated k times until all parts have been left out for validation and test, and results averaged over these k repetitions.

Returns **err_t** – error for the optimal model, computed in the 'cross-testing' manner on data part which is not used for training or validation

Return type double

Created on Mon Oct 27 17:48:33 2014

@author: akusok

hpelm.mss_loo.**train_loo**(*self*, *X*, *T*)

Model structure selection with Leave-One-Out (LOO) validation.

Trains ELM, validates model with LOO and sets an optimal validated solution. Effect is similar to cross-validation with k==N, but ELM has explicit formula of solution for LOO without iterating k times.

Parameters

- **self** (*ELM*) – ELM object that calls *train_v()*
- **X** (*matrix*) – training set inputs
- **T** (*matrix*) – training set outputs

2.2.5 HDF5 Tools

Different tools to work with datasets in HDF5 file format.

Created on Thu Apr 2 21:12:46 2015

@author: akusok

hpelm.modules.hdf5_tools.**make_hdf5**(*data*, *h5file*, *dtype=<type 'numpy.float64'>*, *delimiter=' '*, *skiprows=0*, *comp_level=0*)

Makes an HDF5 file from whatever given data.

Parameters

- **data** –
  - input data in Numpy.ndarray or filename, or a shape tuple
- **h5file** –
  - name (and path) of the output HDF5 file
- **delimiter** –
  - data delimiter for text, csv files
- **comp_level** –
  - compression level of the HDF5 file

hpelm.modules.hdf5_tools.**normalize_hdf5**(*h5file*, *mean=None*, *std=None*, *batch=None*)

Calculates and applies normalization to data in HDF5 file.

Parameters

- **mean** –
  - known vector of mean values
- **std** –

- known vector of standard deviations

- **batch** –
  - number of rows to read at once, default is a native batch size

2.2.6 Multiresponce Sparse Regression

hpelm.modules.mrsr.**mrsr**(*X*, *T*, *kmax*)

Multiresponse Sparse Regression (MRSR) algorithm in Python, accelerated by Numpy.

Finds most relevant inputs for a regression problem with multiple outputs, returns these inputs one-by-one. Fast implementation, but has complexity O(2^m) for *m* features in output.

Parameters

- **T** (*matrix*) – an (n x p) matrix of targets. The columns of T should have zero mean and same scale (e.g. equal variance).
- **X** (*matrix*) – an (n x m) matrix of regressors. The columns of X should have zero mean and same scale (e.g. equal variance).
- **kmax** (*int*) – an integer fixing the number of steps to be run, which equals to the maximum number of regressors in the model.

Returns **i1** – a (1 x kmax) vector of indices revealing the order in which the regressors enter model.

Return type vector

Reference: Timo Simila, Jarkko Tikka. Multiresponse sparse regression with application to multidimensional scaling. International Conference on Artificial Neural Networks (ICANN). Warsaw, Poland. September 11-15, 2005. LNCS 3697, pp. 97-102.

2.2.7 Multiresponce Sparse Regression v2

hpelm.modules.mrsr2.**mrsr2**(*X*, *T*, *kmax*, *norm=2*)

Multi-Responce Sparse Regression implementation with linear scaling in number of outputs.

Basically an L1-regularized regression with multiple outputs, regularization considers all outputs together, method returns the best input features one by one and can be stopped early. Compared to an original MRSR this method is slower for small problems, but has a linear complexity in the number of outputs instead of exponential one, so it is suitable for auto-encoders and other tasks with large output dimensionality.

Parameters

- **T** (*matrix*) – an (n x p) matrix of targets. The columns of T should have zero mean and same scale (e.g. equal variance).
- **X** (*matrix*) – an (n x m) matrix of regressors. The columns of X should have zero mean and same scale (e.g. equal variance).
- **kmax** (*int*) – an integer fixing the number of steps to be run, which equals to the maximum number of regressors in the model.
- **norm** (*from Numpy.linalg.norm*) – norm to use in MRSR2, can be *1* for L1 or *2* for L2 norm, default *2*.

Returns **i1** – a (1 x kmax) vector of indices revealing the order in which the regressors enter model.

Return type vector

Reference: Better MRSR implementation according to: "Common subset selection of inputs in multiresponse regression" by Timo Similä and Jarkko Tikka, International Joint Conference on Neural Networks 2006

Created on Sun Jan 26 13:48:54 2014 @author: Anton Akusok

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

R

S

T

V