PBN Toolit: Reference Manual

Dr. Paul M. Baggenstoss
Fraunhofer FKIE, Fraunhoferstr 20
53343 Wachtberg, Germany
p.m.baggenstoss@ieee.org
http://class-specific.com/csf/index.html
http://class-specific.com/pbntk/index.html

August 21, 2024

Abstract

This document describes the PBN Toolkit.

1 Installation and Prerequisites

1.1 Prerequisites and Installation

The Toolkit uses the Theano framework, but a version is available that runs on Aesara. Aesara is a newer version of Theano, but GPU support is as of now non-existent or limited. If you are not using the GPU, it is recommended to use the Aesara version. Additional prerequisites are `numpy` and `scipy`. Other miscellaneous well-known packages are needed, including `tkinter`, `cPickle`, `matplotlib`, and `sklearn`. Up to date installation instructions for Lunix and Windows can be found in the `readme` file at:

```
http://class-specific.com/pbntk
```

1.2 C Compiler

Theano and Aesara have a C-compiler interface that allows inserting C-code into some functions. In PBN-Tk there are many examples of this for activation functions, random sampling, and other functions. If a Gnu C++ compiler is installed (Windows or Linux), a speed-up factor as high as 10 can be achieved. There are instructions in the readme file for setting up Gnu C compiler in Windows for Theano or Aesara.

1.3 GPU and Smegma Library

An additional factor of up to 10 performance increase can be had using a GPU. To use PBN Toolkit with the GPU, you will need to additionally install `CUDA` drivers and libraries and `cuDNN`, `pygpu`, `libgpuarray` and `scikit-cuda` (SKCUDA). Guides to install these are available online. PBN Toolkit also uses a special library `smegma` that is written as an add-on to `magma`, so MAGMA must be first installed. Contact the author for details.

1.4 Environment Variables

It is beyond the scope of this document to list the setup for all possible platforms. Here is an example setup for using the CPU:

```
export PBNTK_BACKEND=THEANO
export THEANO_FLAGS=mode=FAST_RUN,device=cpu,floatX=float64
```

Here is an example setup for using CUDA:

```
export PBNTK_BACKEND=THEANO
export THEANO_FLAGS=mode=FAST_RUN,device=cuda,floatX=float64,magma.library_path=
/home/paul.baggenstoss/software/magma-2.5.2/lib,magma.include_path=
/home/paul.baggenstoss/software/magma-2.5.2/include
```

Here is an example setup for using Aesara:

```
export PBNTK_BACKEND=AESARA
export AESARA_FLAGS=mode=FAST_RUN,device=cpu,floatX=float64
```

1.5 Windows Installation Guide

Linux is recommended, but the toolkit also works well using Windows. Note that the Windows version that I tried runs fast, but is very slow to compile functions. A Windows installation guide is available at

```
http://class-specific.com/pbntk/readme.txt
```

1.6 Starting the Toolkit

Start the Toolkit by running the main program `pbntk.py` or `cdbn_gui.py`, or `pbntk.pyc` depending on which version you have.

```
$ python pbntk.py
OR if you have source code:
$ python cdbn_gui.py
```

If you are running a script (not using GUI - see Section 17), use:

```
$ python pbntk.py script_file.txt
```

2 Block Diagram

A block dagram of the PBN Toolkit is shown in Figure 1. In simple terms, the toolkit is composed of a feed-forward network (top row, from left to right, marked "forward path" in the figure) and two reconstruction paths. The reconstruction paths show two different ways to reconstruct the input data from the network output. The input data is assumed to have been created by Gaussian-like data (theoretical Gaussian data source) passed through an activation function. This theoretical activation function, called *nonlin* in Section 3.2, plays a role in reconstruction and defines the PBN type for the first layer.

The forward path is a traditional forward (perceptron) network consisting of layers. Each layer consists of a linear transformations (either convolutional or dense), denoted by a weight matrix $\mathbf{W}$, and an added bias denoted by $\mathbf{b}$, and ending with an activation function, called "non-linearity". In the Toolit, layers are numbered staring with layer 0.

The network output is passed to either a classifier cost function (for DNN, see Section 9), or to an output probability distribution (for PBN, see Section 12).

The center reconstruction path, marked "reconstruction path" in the figure is a traditional perceptron network. This path is used by some algorithms in the Toolkit including the auto-encoder (AEC) (see Section 8), the variational auto-encoder (VAE) (see Section 8.2), and the up-down algorithm (UPDN) (see Section 11). It is also used on a layer-by-layer basis by the restricted Boltzmann machine (RBM) (see Section 10).

By default, the reconstruction path features tied weights, so that the weight matrices used for reconstruction are the same as the corresponding matrices in the forward path. This is a common practice in building auto-encoders [1]. Tied weights is shown by grey lines in the figure. Tied weights can be disabled by checking

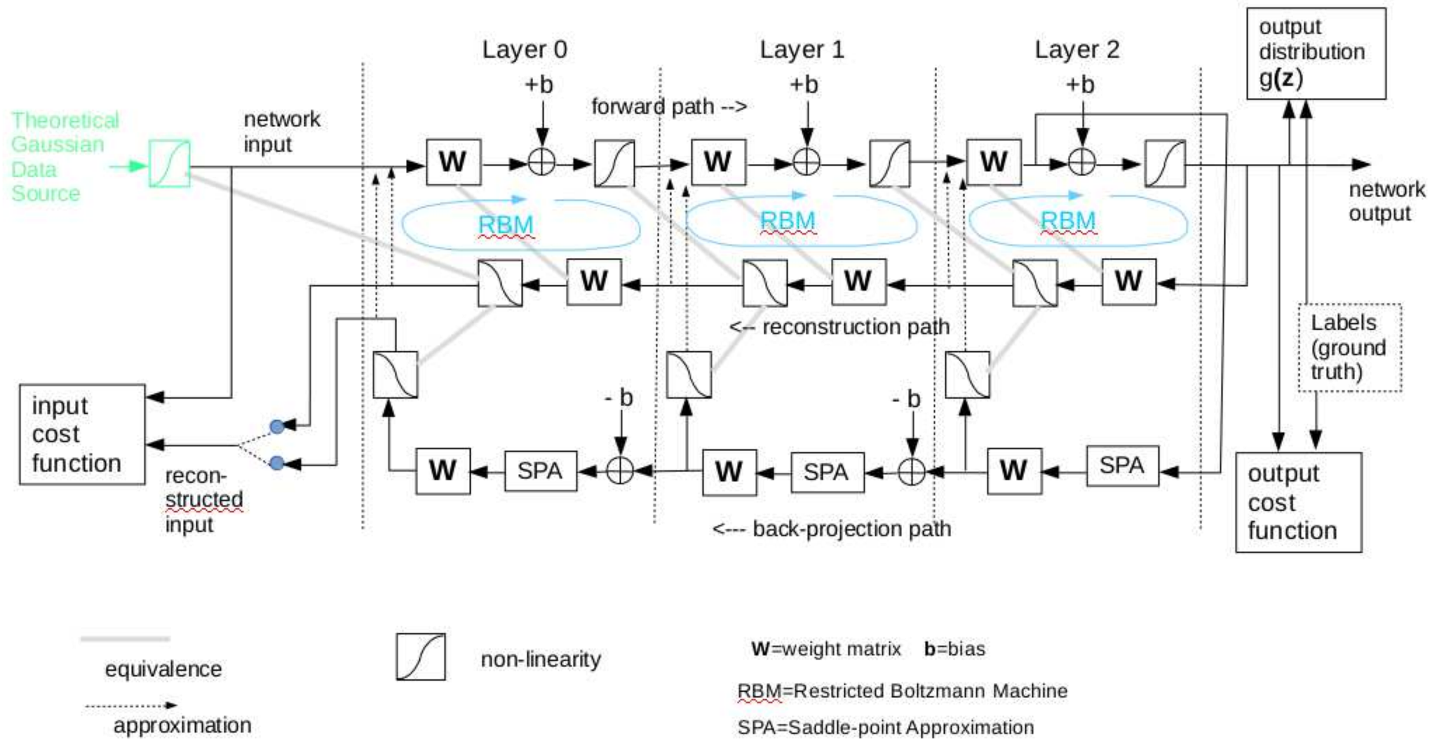

Figure 1: The PBN-Tk Block Diagram.

the "r.wts" checkbox in the toolkit, so that separate reconstruction weights are enabled. However, when tied weights are used, there will be a separate trainable scale factor used in the reconstruction path. To allow these scale factors to change, check "upd.s2" in the PBN Section.

In the reconstruction path, a separate reconstruction bias is not normally used. Reasons for this are explained in Section 19. However, by checking the "r.bias" checkbox, this policy is disabled and reconstruction bias is enabled in algorithms that use the traditional perceptron network reconstruction path. The non-linearity used in a layer of the reconstruction path is usually the same as the non-linearity at the output of the previous layer in the forward path. This equivalence is also shown by grey lines in the figure.

The reconstructed input data (at the reconstruction network output) is passed to a reconstruction cost function to determine the goodness of fit.

The lower reconstruction path, marked "back-projection path" is used by the PBN and DPBN algorithms to reconstruct data (see Sections 12 and 13). This is a way to reconstruct by backing-up through the forward network, and can be theoretically justified as an optimal way to reconstruct data [2] from a feed-forward network. The reconstruction weights are tied to the forward network weights by defintion. There are two approaches to back-projection, stochastic and deterministic, corresponding to the two different algorithms PBN and DPBN in the Toolkit, respectively.

Here is a short mathematical explanation of DPBN: Let $\mathbf{x}$ be the data vector at the input to a layer, and let $\mathbf{z} = \mathbf{W}'\mathbf{x}$ be the output of the linear transformation in the forward path, and suppose $\mathbf{z}$ has lower dimension than $\mathbf{x}$. The deterministic way to reconstruct $\mathbf{x}$ from $\mathbf{z}$ is to solve for the vector of coefficients $\mathbf{h}$ such that $\sigma(\mathbf{Wh} + \mathbf{a}_0) \simeq \mathbf{x}$. where $\sigma(\ )$ is the non-linearity at the output of the previous layer, and $\mathbf{a}_0$ is the fixed base parameter for the maximum entropy reference hypothesis, normally zero. The exact equation to be solved is:

$$\mathbf{W}'\sigma(\mathbf{Wh} + \mathbf{a}_0) = \mathbf{W}'\mathbf{x} = \mathbf{z}.$$

Vector $\mathbf{h}$ is the vector between "SPA" and "$\mathbf{W}$" in the figure. The dotted arrow in the figure means "approximation", which is a way to say $\sigma(\mathbf{Wh} + \mathbf{a}_0) \simeq \mathbf{x}$. Under certain conditions, this is called the saddle point approximation (SPA). The conditions are that $\sigma(\ )$ is the derivative of the cumulative generating function (CGF) of the *a priori* distribution of $\mathbf{x}$. This is all explained in [2]. Note than in the back-projection path, the bias that was added in the forward path needs to be subtracted.

3 Basic Operation, Controls and Features

3.1 GUI Layout

Figure 2 shows the PBN-Tk GUI. In the figure, each section of the GUI is labeled and will be refered to in the following text. Table 1

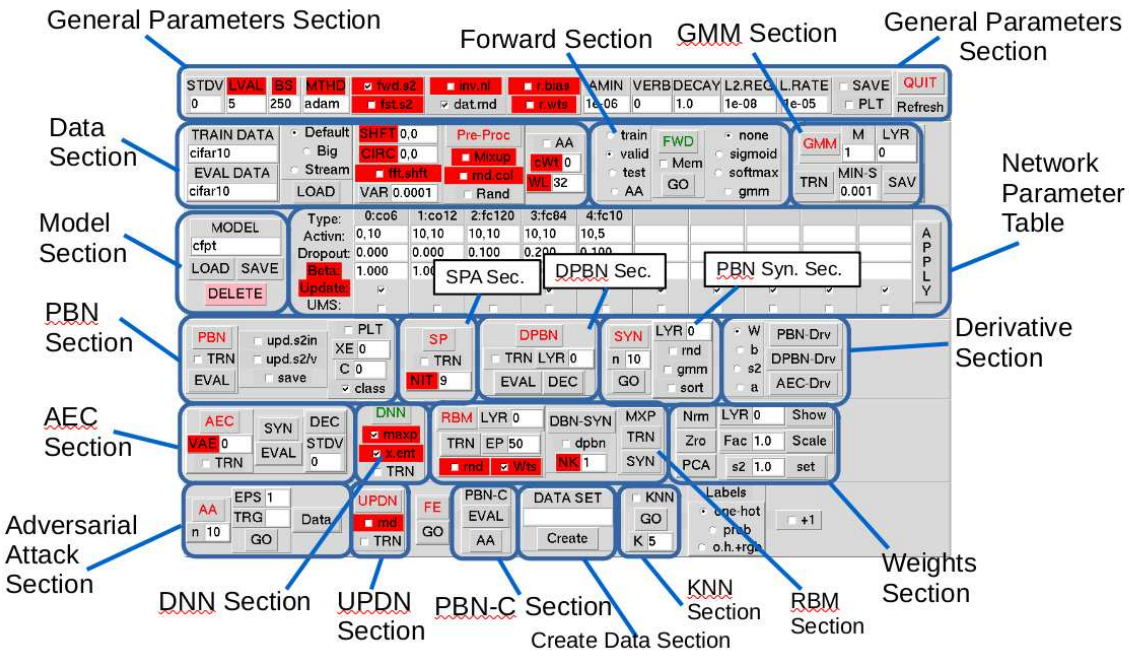

Figure 2: The PBN-Tk Graphical Interface.

Table 1 gives the correspondence between the document sections and the sections of the GUI interface. To avoid confusion, a document section is refered to as "Section XXX", and a GUI section is refered to as "XXX Section".

GUI Section	Documentation Section	GUI Section	Documentation Section
General Parameters Section	Section 5	Forward Section	Section 6
GMM Section	Section 15	Data Section	Section 3.2
Create Data Section	Section 12.7	Model Section	Section 4
Network Parameter Table	Section 4.3	PBN Section	Section 12
SPA Section	Section 12.6	DPBN Section	Section 13
PBN Synth Section	Section 12.4	AEC Section	Section 8
DNN Section	Section 9	UPDN Section	Section 11
RBM Section	Section 10	Weights Section	Section 16
Adversarial Attack (AA) Section	Section 25	PBN-C Section	Section 21
Data Creation Section	Section 23	KNN Section	Section 22

Table 1: Correspondence of GUI sections to sections in the Documentation

3.2 Data Input

To create a new data set, you will need to create data files in one of three formats: “Default”, “Big” , and “Stream”. These formats are selected by the radio buttons in the Data Section.

3.2.1 Default Data Format

The default data format is the fastest because it stores all the data in memory (in GPU memory if GPU is enabled), eliminating the need for data transfers during training. To create a new data set in the default format, you will need to create a MATLAB data file named `[data set].mat`, where `data set` is a chosen data set name. In Python, these files can be read and written using the `loadmat` and `savemat` functions in the `scipy.io` package. The file needs to contain the following variables:

Required variables:

- **X** : data matrix with shape N-by-nsamp, nsamp is the number of samples and N is the total dimension. Variable X must always be a 2-dimensional matrix. Data can be 3, 2, or 1 dimensional, but X must be always re-shaped to N-by-nsamp. The integer variables N1, N2, nchan define the actual dimension of the data. MATLAB commands that produce the right data shapes are illustrated below.

  1. For 1D (vector) data,
     ```
     X = data(1:N,1:nsamp);
     ```
  2. For 2D (image) data,
     ```
     X = reshape(  data(1:N2,1:N1,1:nsamp) , N1*N2,nsamp);
     ```
  3. For 3D (color image) data,
     ```
     X =reshape(  data(1:N2,1:N1,1:nchan,1:nsamp) , N1*N2*nchan,nsamp);
     ```

- **N2** : Integer. First MATLAB data dimension (third dimension in Python)
- **N1** : Integer. Second MATLAB data dimension (second dimension in Python)
- **nchan** : Integer. Third MATLAB data dimension (first dimension in Python)
- **nclass**: Integer. Number of data classes (for a classification experiment)
- **nonlin**: Integer. Nonlinearity (activation function). This is the type of activation function that “could have” produced the data. It is also the activation function used to reconstruct visible data. It is also important to know this because it defines the type of PBN for the first layer. More about activation functions is found in Section 4.1. There are three possibilities depending on the range of the data values:

nonlin	DATA VALUES	Assumption
0	$[-\infty, \infty]$	Gaussian
5	$[0, 1]$	Uniform
10	$[0, \infty]$	Truncated Gaussian

Optional variables:

1. **tch**: matrix of dimension (nclass-by-nsamp) with values of 0 or 1, one-hot encoding of the ground truth for training data
2. **Xv**: Validation data matrix (like X) of dimension (N-by-nv) where nv = number of validation samples

3. **tchv**: matrix of dimension (nclass-by-nv) with values of 0 or 1, one-hot encoding of the ground truth for validation data
4. **X2**: Test data matrix (like X) of dimension (N-by-nt) where nt = number of test samples
5. **tch2**: matrix of dimension (nclass-by-nt) with values of 0 or 1, one-hot encoding of the ground truth for test data
6. **lpx**, **lpxv**, **lpx2**: log-likelihood values for the train, validation, and test partitions for the ACID TEST (See Section 12.3).

3.2.2 Big Data Format

By checking the "Big" radio button in the Data Section, a special data file format is used, designed for large data sets. Data is not read in all at once, but rather is read one batch at a time by efficiently seeking through the file. The data must exist in binary files with filenames "[dataset]_[type].bin", where "dataset" is the data set name as entered in the TRAIN DATA field, and type is one of "train", "test", or "valid". The binary files have a 24-byte header consisting of six unsigned 32-bit integers (uint32) containing the variables *nonlin, nsamp, nclass, nchan, N1, N2*, which are defined above, except for *nsamp*, which is the number of samples in the file. All three files do not need to exist, but to train, the "[dataset]_train.bin" must exist. The rest of the file consists of the *nsamp*$\times$*DIM* floating point numbers of type float32 where *DIM* is the total dimension *nchan*$\times$*N1*$\times$*N2*. The following Python excerpt demonstrates how to read one batch of data from the binary file. From this example, it is also clear how to create the files.

```
# reads a batch of data (samples i1:i2)
def read_batch(filename,i1,i2)
    # read header
    fp = open(filename, 'rb')
    hdr = np.fromfile(fp, np.uint32,6)
    nonlin_in  = hdr[0] # nonlin
    nsamp      = hdr[1] # nsamp
    nc         = hdr[2] # nclass
    input_chan = hdr[3] # nchan
    nrow_in    = hdr[4] # N1
    ncol_in    = hdr[5] # N2

    # seek to the desired starting sample
    # second argument of seek:
    # 0 - offset is relative to start of file
    # 1 - offset is relative to current position
    # 2 - offset is relative to end of file
    n0=i1*input_chan*nrow_in*ncol_in*4 # float32 is 4 bytes
    bs = i2-i1
    n=bs*input_chan*nrow_in*ncol_in
    fp.seek(n0,1)
    x = np.fromfile(fp, np.float32,n)
    fp.close()
    x=x.reshape((bs,input_chan,nrow_in,ncol_in))
    return x
```

In Big data mode, the RBM algorithms will not work because they need to have all the training data in memory. The UPDN algorithm will work, and is a better alternative anyway.

3.2.3 Stream Data

"Stream" data mode is a third data mode that can be selected. In 'Default" and "Big" data mode, data is organized into events where each event is identified as a certain data class and, each event has the same time shape, in all dimensions. But, data is not always organized into separate equal-size events, where each event belongs to a data class. Sometimes the data is a continuous stream and the ground-truth (label data) applies only to short time-segments. Data classes may also overlap. By checking the "Stream" radio button in the Data Section, a special data file format is used, designed for large data sets where each data event can have different size in one dimension (normally time dimension), but must have the same size n the remaning dimensons (such as frequency and feature depth). In Stream Mode, the data is processed in sliding windows of fixed length, equal to the "WL" entry field in the Data Section. However this field is not meant to be settable, it is filled in when loading a model (See Section 3.4).

Sliding windows are grouped into batches for efficient processing. In "Default" and "Big" data mode, processing occurs in batches where each sample in a batch is a different event. In Stream mode, however, a batch holds a number of time-ordered sliding windows from a single recording (or more if one recording ends within a batch). The amount of time-shift applied to each successive window is automatically determined by the toolkit, based on the window length (parameter "WL") and the convolutional filter sizes applied to the time dimension in all the convolutional layers. The time shift is computed and displayed only when loading a model (see Section 4.3) (actually the number displayed is the overlap between windows) . It is determined so that the output of the network produces a continuous stream of results. In stream mode, the "valid" convolutional border (see Section 18) is required to assure seamless streaming across sliding windows. The stream may run over several larger recordings (events). Each event is contained in a different binary file. When one event is complete, the toolkit seamlessly switches to the next event so it is possible that one batch may contain windows from the end of one event and from the start of the next event. The ground-truth information is also a stream with dimension M, where M is the number of classes.

Ground truth data is normally probabilities (floating-point numbers between 0 and 1 indicating the probability a given class is present). Class probabilities of two or more classes may overlap. If all class probabilities are zero at a time step, it indicates that no information is available, and the time step is ignored in the calculation of the total classifier cost function.

Note that ground-truth data must also be entered with the same time resolution as the input data. Ground-truth data gets automatically pooled (down-sampled) in each layer, using the same filter sizes as for the data, so that it has the same time resolution as the network output and is synchronized with it.

Fully-connected layers can be used at the end of the network, but they will be seen as a convolutional layer where the filter length equals the input time width, producing just one time sample for the given input sliding window. Ground-truth data will be similarly summarized, producing just one ground-truth probability vector per window.

One must be careful when training with streaming data because due to the time-ordering, the Toolkit randomizes the order of the events - it is not possible to randomize the samples in a batch. Randomizing the samples in a batch is important for stochastic gradient descent training. This is especially true when input data is organized in order of data class. The danger is that the network will start training with just one class type. Randomizing data in a batch is also essential when using batch normalization because all data or most classes should be present in a given batch so that the batch statistics are representative of the data set as a whole. Batch normalization is not yet implemented in the Toolkit. When using streaming, the learning rate should be low enough so that the network parameters do not change too much over a short time and that all data classes, regardless where they occur in time, have equal influence.

Unlike Default and Big data modes which use a single file containing all events with a fixed size, Streaming mode works with separate files for each event. A text file, named "[dataclass].txt" contains the file names (full path or path relative to current directory) of each event file, one on each line. These event files are of a special binary type. The file format can be deduced from the functions `read_binary_stream_file` and `write_binary_stream_file` in module `data_sets.py`, with syntax:

```
X,tch,nonlin,names,xshape,yshape = read_binary_stream_file(fn,read_data)

write_binary_stream_file(fn,X,tch,nonlin,names)
```

The variables "X" and "tch" are the data and ground truth variables with shapes:

```
X.shape = (nseg,dim,N2)
tch.shape = (nseg,nc,[N2,1])
```

Variable "nonlin" is explained in Section 3.2.1. Variable "names" is a list of strings (class names) for the ground-truth data classes that are contained in the file. By including this in the file, it is possible to join files that have a different set of classes into a single data set. The Toolkit will use the union of all the data class names that it finds in the data set. Variable "names" must be of length "nc". Variable "xshape" and "yshape" are the shapes of variables "X" and "tch", respectively. These are needed in case "read_data" is set to False, in which case "X" and "tch" are returned as null (empty) variables.

The ground truth variable `tch` has shape (nseg,nc,[N2,1]). The first two dimensions are the time and feature dimensions, respectively. Because ground truth is synchronized with the data, "nseg" must be the same for the input data. The ground truth feature dimension "nc" must match the output feature dimension of the network, which is normally equal to the number of data classes, but can also be arbitrary. Ground truth can also have a third dimension. An example is ground-truth data for spectrograms, where the class identity can vary as a function of frequency. The last dimension of the ground truth array `tch` is called "N2". There are two possibilities (a) N2=1, and (b) "N2" matches the last dimension "N2" of the input data, which in the case of spectrogram data, is the number of frequency bins. In the first case, ground truth varies only as a function of time. In the second case, it varies as a function time and frequency.

When using Stream data mode, you may need to load data and model twice. This is because the window length (WL) is not known until the model is loaded, and this is needed when data is loaded (chicken and egg problem). For safety, always do : load data, load model, load data, load model.

If you load a data set different from the one that was loaded before loading a model, you can press "Refresh", which takes the place of re-loading a model again. This is necessary if you are evaluating data that was not used for training!!

3.3 Loading Data

Once you have created a new data set, enter the data set name in "TRAIN DATA" entry box in the Data Section, then press "LOAD". For example, in the Default data mode, to load 'ds81.mat', enter 'ds81' The first time, it will take longer. Once data is loaded, the "EVAL DATA" field will be updated to match the "TRAIN DATA" field. The "EVAL DATA" can be changed to reflect the desired data classes for evaluation (when one of the "EVAL" buttons is pressed). To avoid having to input data set every time you start PBN-Tk, you can set the default in defaults.py:

```
cfg.train_class = 'ds45'
```

WARNING : entering a data class name in "TRAIN DATA" field, but not pressing "LOAD" can result in the wrong data class being assumed.

3.4 Additional Data controls

In the Data Section, there are some additional controls.

1. **VAR**: Variance of Gaussian dither applied to the input data only for the Gaussian assumption (i.e. nonlin=0 , see Sections 3.2, 4.1).

2. **SHFT**: This affects random shifts applied to data. For spectrogram data, SHFT="7,1" applies random time shift in the range [-7, 7], and random frequency shift in the range [-1,1]. The shifts are non- integer (vernier) shifts using frequency-domain approach.

3. **CIRC**: This affects random shifts applied to data. If "CIRC" is 1, then the shifting is circular (no zero padding). CIRC="1,1" applies circular shifting to both dimensions. When CIRC=2 (in either dimension) uses non-circular shifts, but causes edge pixels to be replicated when shifting (instead of shifting in zeros). Note that CIRC=2 only has an effect when `fft.shft` is checked.

4. **fft.shft**: When selected, implements advanced shifting using FFT for vernier shifts (not integer), for CIRC=1 or CIRC=2 (see above). When unchecked, CIRC=2 is not implemented.

5. **PP**: Pressing this re-compiles the pre-processing function. Press this if you have changed SHFT or CIRC.

6. **Rand**: Checking Rand turns on random shift and Gaussian dither. Parameter "Rand" also has an effect when displaying re-synthesized data: when "Rand" is selected, a randomly-selected data batch is used instead of just the first batch.

7. **Mixup**: Checking this turns on Mixup data augmentation during training. When mixup is selected, a batch of data is randomly mixed with the corresponding sample in a order-reversed batch. Random mixing is done by choosing a random linear mixing ratio between the two events. The labels are also mixed at the same ratio.

8. **AA**: Adversarial Attack (See Section 25)

9. **cWt**: Class weighting (integer). Set to the index of the desired class (1 to nclass). If set to zero, it will weight all data equally. This is useful for training on data from just one class. See Section 21.

10. **WL**: Sets the window length for streaming data mode (See Section 3.2.3)

4 Defining a Network

4.1 The Model File

To define a model with name 'newmodel', you must create the network definition file 'newmodel.py' and place it in a sub-folder 'models/'. It is important to keep the model definitions in a separate folder so they do not get confused with actual Python code. As examples, you can use 'tut1.py' as an example of a fully-connected network, and "tut2.py" which is a convolutional example. The file is a python script that fills in the layer structure with the following fields:

REQUIRED FIELDS:

1. **type**: string, 'conv', 'fc', 'dbn', or 'nl', to specify CONVOLUTIONAL, FULLY CONNECTED (dense), DEEP BELIEF NETWORK, or NON-LINEARITY layer types respectively. DBN layer types are described in Section 10.5. NL layers are described in Section 7. An NL layer is a compound (stacked) non-linearity and acts as the output non-linearity for the previous layer. Adding an NL layer does not change the number of actual network layers.

2. **nchan_out**: integer, number of neurons (columns in weight matrix for fc layers) : or number of kernels (for conv layers). Required for all types except 'nl'.

3. **nonlin**: Two-dimensional tuple of integers that specifies the input and output activations of the layer. For "nl" layer type, the base non-linearity is specified by the second entry. When a "nl" layer type is specified, the output non-linearity of the layer that precedes it must be 0 (linear), so the input non-linearity of a "nl" layer is always 0. Valid 'output' activation are:

   The allowed activation functions are shown below. All activation functions can be used as OUTPUT activations. But, only some can be used as INPUT activations, and fewer can be used with the PBN as INPUT activations.

	OUTPUT	INPUT	INPUT (PBN)
0 = none (linear)	X	X	X
1 = Sigmoid	X	X	(use 5)
5 = TED (similar to sigmoid)	X	X	X
6 = softmax	X		
8 = softplus	X	X	(use 10)
9 = Relu	X	X	
10 = T-Gauss (similar to softplus),	X	X	X

   For PBN, activations 0, 5, and 10 are recommended. The input activation of a layer should match the output activation of the previous layer, although some substitutions can be made, shown in Table as (use XX). For example, if a layer has Sigmoid output activation (nonlin=1), then you should use TED (nonlin=5) as the input activation in the next layer. If a layer has Softplus output activation (nonlin=8), then you should use TG (nonlin=10) as the input activation of the next layer. When loading a model and an warning will be printed if they do not match. For the first layer, the input activation will be automatically set to the data activation (nonlin) that is loaded from the data file as explained in Section 3.2. The layer preceding an NL layer (see Section 7) must have a linear (nonlin=0) output activation because the NL layer acts as its output activation.

REQUIRED FIELDS FOR CONV LAYERS:

1. **filt_row, filt_col** : Integers. The filter kernel size , if spectrogram: (time, freq)

2. **border_mode** : String.

   - 'valid' : no border assumption, kernel stays within data rectangle, produces output maps smaller than input maps. Valid mode is mandatory for streaming mode (see Section 3.2.3).
   - 'half' : border half the size of kernel, output map same size as input data (before downsampling). Filter kernel sizes must be odd numbers!

   More on setting up a convolution layer is given in Section 18.

3. **pooling** : Tuple of integers of length two. Downsampling factors. If spectrogram: (time, freq). For 'valid' border mode, data size (rows/cols) minus the filter size (rows/cols) must be divisible by the downsampling (rows/cols). For example, for MNIST data (28,28), with (16,16) kernels, the difference is (12,12). Therefore pooling of (3,3),(2,2), (4,4),(6,6), (6,2), etc... are allowed. For 'half' border mode, any pooling can be used, but it is recommended that the input data size be divisible by pooling.

   It is necessary for PBN or DPBN training that each layer has a significant dimension reduction with respect to the previous layer. Suggested is a factor of 1.5 reduction at least in each layer. Layers where output dimension and input dimension are the same (1:1 layers) are also allowed. Total output dimension gets printed when the model is loaded. More on setting up a convolution layer is given in Section 18.

REQUIRED FIELDS FOR NL LAYERS:

1. **nnl** : Number of non-linearities. See Section 7.

FIELDS FOR Streaming Mode:

1. **WL** : Window length (optional). See Section 3.2.3. When present, over-rides the WL field in the graphical interface. Note that whenever WL changes, you may need to reload data and model a second time.

OPTIONAL FIELDS:

1. **input_dropout_fac** : Dropout factor applied to input of layer (if this is first layer, lyrs[0], then this will be applied to visible data). This has effect for DNN and for PBN (when 'use dropout' is checked)

4.2 Model File Example

To create your own network, you will need to create a model file, which os a Python function. The following code segment is from "tut1.py" :

```
 -----------------------
def make_network(cfg):
   cfg.nlayer = 3
   nl_in = cfg.nonlin_in
   if nl_in==0:
     nl=10
   else:
     nl = nl_in
   nl_out = 5

   cfg.lyrs[0].type = 'fc'
   cfg.lyrs[0].nchan_out = 32
   cfg.lyrs[0].nonlin = (nl_in,nl)

   cfg.lyrs[1].type = 'fc'
   cfg.lyrs[1].nchan_out = 8
   cfg.lyrs[1].nonlin = (nl,nl)

   cfg.lyrs[2].type = 'fc'
   cfg.lyrs[2].nchan_out = 3
   cfg.lyrs[2].nonlin = (nl,nl_out)

----------------------
```

Here, we see that the input activation for the first layer is set to the global variable "cfg.nonlin_in", which is filled in when data is loaded. The output activations are all set to the same value as the input activation, except if the input activation is 0 (linear), then they are set to 10 (TG). The output activation is set to 5 (TED).

4.3 Loading, Saving, Deleting, and Modifying a Network Model

To load a model, enter the model name, then press "LOAD" in the Model Section. Model parameters (bias and weights) are loaded if the files exist. Each layer is stored in a different file with names

```
  [newmodel]_[data set]_lyr0.mat,   [newmodel]_[data set]_lyr1.mat, .....
Example:
  tst2_ds81_lyr0.mat
```

Layer numbers begin with zero (0,1,2,...), also when determining filenames. Previous versions of the toolkit added 1 to each filename, so they began with "lyr1", "lyr2", ... But this is obsolete. To allow accessing the old file naming convention, the button "+1" is provided on the bottom right of the GUI. You can convert from old to new convention by checking "+1" before loading, then unchecking before saving.

If you want PBN-Tk to initialize a layer, make sure to delete the files before pressing "LOAD". If files exist that used a different network configuration, they will cause errors. Delete the files.

The network layer sizes, activation functions, and dropout factors will be displayed in the Network Parameter Table. Using the "Activn" field, the input and output activation functions 'nonlin' (as a comma-separated pair) can be modified. Using "Dropout", the input dropout factor of each layer can be modified. If the checkbox 'Update' is un-checked, then the parameters of that layer will remain unchanged while training. The checkbox UMS affects synthesis (Section 12.4) The "Dropout" field allows you to over-ride the input dropout parameter in the network definition script (Section 4.1). The 'Activn' parameter specifies the input and output activation of the layer. The 'Beta' parameter specified beta for the given layer (See Section 12.2).

The network can be temporarily shortened by clearing or putting a space " " in the "Activn" field to disable a layer. Modifications will not take effect until you press "APPLY". Modifications to the "Activn" field will be saved to the layer parameter files when the model is saved, and retrieved when loading, but will not be saved to the model definition file. If you want to change them permanently, make the changes to the model definition file [model-name].py

The data weights (or kernels if CONV) can be seen by pressing "Show" in the Weights Section (See Section 16). The layer can be selected using 'LYR' entry box. Layers start at 0, and layer -1 refers to the input data. When viewing NL layers, both weights and biases will be displayed.

Model parameters can be saved by pressing "SAVE" in the Model Section, or when training saved after each epoch when the "save" checkbox on the PBN Section is on. The parameters are saved in a MATLAB-compatible file (also readable in Python with the scipy.io package). Layer weights and bias values are stored in variables "W" and "b", respectively.

Model parameters can be deleted by pressing "DELETE" in the Model Section. This removes the model files so that the next time you press "LOAD", new parameters will be initialized.

To avoid having to input model name, etc, every time you start PBN-Tk, you can set the default in defaults.py:

```
cfg.prefix = "cdbn45"
```

If you want to start with fresh (random weights), just delete all the model parameter files. For example,

```
$ rm tut1_ds414243_lyr*.mat
```

which will be automatically done by pressing "DELETE" in the Model Section.

5 General Parameters

These controls are located in the General Parameters Section. Note: any parameter shown in red background color will require re-compilation to take effect.

General Parameters:

1. **LVAL**: Label signal value, only used in classifier mode.
2. **BS**: batchsize. This and other parameters shown in red will require re- compiling when changed. When changing BS, you need to re-load the data, the model, and re-compile. Note that any changes to the model parameters need to be saved before re-loading! If you set the batch size higher than the number of events in a parttition, you will get an error when trying to use that partition.
   WARNING : When training, the training data size (number of training samples) should be divisible by the batch size. If this is not possible, use a batch size slightly greater than a fraction of the data size. For example, if there are 1038 training samples, and you want a batch size near 100, use BS=104. Do not use a batch size greater than any of the data partitions (i.e. training, validation, test), or an error may occur.
3. **MTHD** : Optimization method: 'adam', 'nest' (Nesterov), 'mom' (Momentum), or 'sgd' (stochastic gradient descent without smoothing). Shown in red because when changing it, you will need to re-compile.
4. **'fwd.s2'**: Forward 's2'. parameter 's2' is the scale or variance parameter that affects the input to a layer. Checking this box (default) allows using 's2' scaling in the forward direction. See Section 19 for more information.
5. **'fst.s2'**: Fast 's2'. enables fast estimation if parameter 's2' in input layer for Gaussian input nonlin. See Section 19 for more information.
6. **'inv.nl'**: Invert NL. You must check this check-box to use NL layer inversion for the first layer in UPDN and PBN algorithms. This only affects the reconstruction cost that is printed out. See Section 7 for more information.
7. **'dat.rnd'**: Random Data Order. When checked, randomizes input data order. This is best for training. Un-ckeck this box for evaluating data that is in a special order, so that it gets saved properly.
8. **'r.bias'**: Reconstruction bias. Reconstruction bias is normally not used except in RBMs. Checking this box allows estimating a reconstruction bias for other models (AEC). See Section 19 for more information.
9. **'r.wts'**: Reconstruction weights. Reconstruction weights are normally shared with analysis weights. Checking this box allows estimating a separate set of reconstruction weights for AEC. See Section 19 for more information.
10. **AMIN**: Factor added to diagonal of the solution matrix before inversion, needed only for training PBN or DPBN, or synthesizing with these.
11. **VERB**: Verbosity. When greater than 0, prints more things out.
12. **DECAY**: Decay factor applied for each batch for layer weight and bias. Use 1.0, 0.9999, etc. Applied each batch.
13. **L2.REG**: L2 regularization factor. Has similar effect like DECAY. Use about 1e-5 for training DNN, use as much as 0.02 for training PBN.
14. **L.RATE**: Learning rate, suggest 1e-3 for DNN, 1e-4 to 1e-5 for PBN.
15. **SAVE**: Check this to save model parameters after each epoch.
16. **PLT** : (checkbox) Check this for plotting displays when running "FWD" , "SYN" and other methods.
17. **QUIT**: Exit PBN-Tk.

18. **Refresh**: This button refreshes some variables including streaming parameters. In has additional initialization functions that depends on wether the “PLT” checkbox, that appears directly next to it, is checked or not. If “PLT” is checked, plot history is cleared (see Section 14), and clears velocity variables in the optimization algorithms (adam/nest/mom) (See Section 5). This can help if there were any numerical overflows that have corrupted the training algorithm. If “PLT” is not checked, it reloads python modules if they have been changed in the background , useful for code development.

6 Forward Section

The forward network is the top row in Figure 1. In certain conditions, the network is also a classifier and the classification results are calculated and displayed.

6.1 Using the forward network

The forward function implements the feed-forward network. It does not train the network, but just calculates the output of all the hidden variables given the data input. It can also plot hidden variables and network output and compute classifcation results and save data to files. Controls are located in the Forward Section. To compile the network forward algorithm, press the ”FWD” button. To evaluate data using the feed-forward network, press the ”GO” button. The radio buttons on the left select which data is used: ”train”, ”valid”, ”test” , or ”AA” data (see Section 25). If the ”PLT” checkbox is checked (top right of window), then an intensity image of hidden variables of each layer will be plotted. For the last layer, the sigmoid or softmax can be applied before rendering by selecting the radio buttons. Or, the GMM classifier is plotted if “gmm” is selected (Section 15). Note that to use the GMM, the output of the Forward algorithm must be saved in memory by selecting the ”Mem” checkbox.

6.2 Classifier function

If the last layer is fully connected (FC) and it has an output dimension equal to the number of classes (nclass in Section 3.2), then the classification mode will be enabled (when ’class’ checkbox is checked in the PBN Section). For classifier function, it is necessary also that the ’tch’ variable, which holds the ground-truth classfication labels, is included in the data (Section 3.2). In classifier mode, running the FWD function will print out the classification results on the selected data partition. A classification result is also possible using a GMM classifier applied to output layer. The result of the ’gmm’ classifier will be displayed and/or printed of radio button ’gmm’ is selected. See Section 15 for more about using GMM. If the loaded network is a DBN, the DBN classifier results will be printed and/or plotted (See Section 10.5). Classifier mode can be disabled for some algorithms (PBN,AEC,DPBN,UPDN) by un-checking the “class” button in the PBN Section.

7 Non-Linearity (NL) Layers

The Toolkit provides three main non-linearities: linear (i.e. no non-linearity) with `nonlin=0`, truncated exponential (TED) a sigmoid-like non-linearity with `nonlin=5`, and a truncated Gaussian (TG) a softplus-like non-linearity with `nonlin=10`. Sigmoid `nonlin=1` is an alternative output non-linearity to TED and Softplus `nonlin=8` is an alternative output non-linearity to TG. However, all these non-linearities describe a simple functional shape. If a more complex non-linearity is desired, that can be trained, you can add an NL layer as an network input layer, or after any layer. When adding after a normal layer, the normal layer must have output `nonlin=0`. An NL layer implements the function

$$f(x_i) = \frac{\sum_{k=1}^{K} e^{w_k} \sigma_k(x_i e^{a_{i,k}} + b_{i,k})}{\sum_{k=1}^{K} e^{w_k}},$$

where K is the number of non-linearities ('nnl' parameter specified in the model definition), and $\sigma_k(\ )$ are the activation functions. Using the exponential function for the weights and scale factors is just a way to insure that they are positive.

The types of the individual functions $\sigma_k(\ )$ depend on two things: the number of non-lnearities (K or "nnl") and the specified non-linearity. When a non-linearity type of "0" or "10" is specified (Gaussian or Truncated Gaussian), then exactly one of the K function types (base non-linearity) will be what is specified, and all the remaining ones will be TED (type 5). However, when the non-linearity type "5" is specified then all K function types will be TED. The base non-linearity is always the last one of the group. This is done because the overall function shape will take after the base non-linearity, with the multiple TED non-linearities adding "bumps" or "wiggles" in various places. Because the derivative of the TED non-linearity looks like a Gaussian "bump", it makes an ideal base component in a multi-component non-linear function. More information is found in [3].

An NL layer can add significant function-approximation capability with few parameters. A normal FC layer requires $N \times M$ parameters, but an NL layer has only $3 \times N \times K$ parameters, where K is small (2 or 3). An NL layer, even `nonlin=0`, is also very useful as an input layer (layer 0) because it allows separate scaling of each input dimension. In a PBN, an input NL layer effectively adds a simple PDF estimator to each input pixel and significantly improves likelihood values. The back-projected PDF of an NL layer with `nonlin=5` or `nonlin=1` is effectively a Gaussian mixture! Figure 3 illustrates this property for the data values of a single pixel.

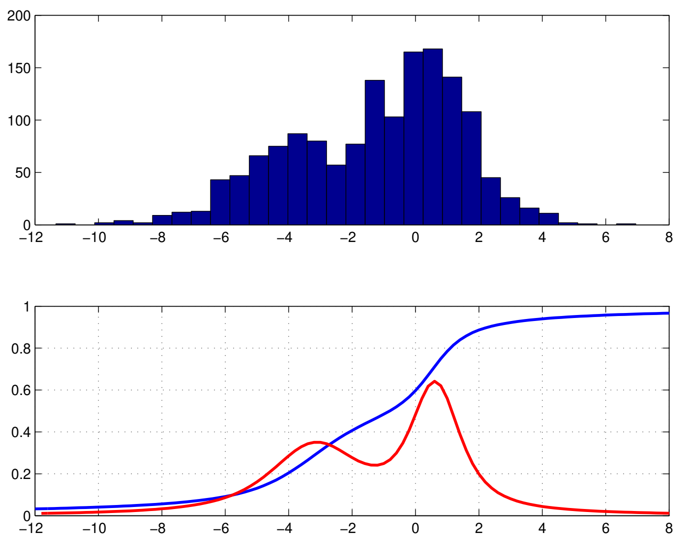

Figure 3: Illustration of effective PDF estimation afforded by NL Layer. Top: histogram of pixel values. Bottom: NL transfer function and its derivative after training.

NL layers do add significant computational load, which is parallelized if GPU is enabled. It is best to train the network first with NL layers held unchanged by unchecking "Update" in the network parameters Section for the NL layers.

A block diagram of a 2-layer PBN-Toolkit network with NL layers is illustrated in Figure 4. Note that the NL layers act as activation functions for the layer that preceded it. Depending on the type of reconstruction is selected, the NL layer may need to be inverted. All compound activation functions are invertible because they are always-increasing functions. When comparing with Figure 1, note that each NL layer takes the place of the non-linearity at the output of the previous layer. Also, if an NL layer is used at the network input, it takes the place of the "theoretical" input non-linearity. In the traditional reconstruction path (center row), a standard non-linearity is used, and this must correspond to the base-non-linearity of the NL layer that comes prior to it. This equivalence is shown by grey lines in the figure. A more complicated reconstruction

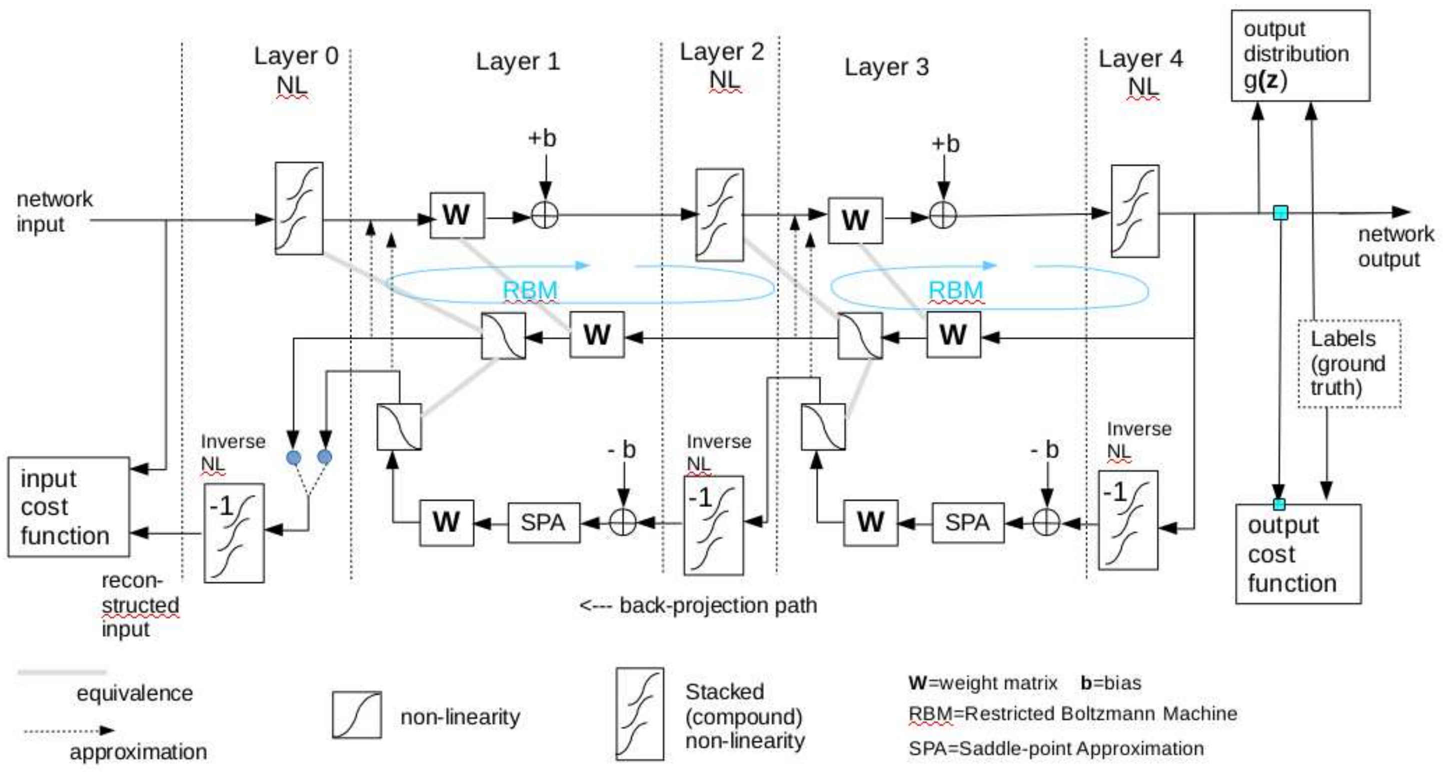

Figure 4: The PBN-Tk Block Diagram with NL Layers.

is required for the back-projection reconstruction path because the compound NL layer must be inverted. Inversion is done by finding the input value that caused the given output value using a combination of bisection and Newton-Raphson.

7.1 Reconstruction with NL Layers

The Toolkit provides several ways to reconstruct input data from the network output (see Figure 4). For NL layers, the way data is reconstructed depends on the general reconstruction algorithm. For DPBN (Section 13), the NL layer is inverted by brute force (the input sample is solved for that produces the given NL layer output). Inversion is done by bisection until the error is small, then a few iterations of Newton-Raphson are appled to get the error to near machine precision. Efficient C code and GPU code is used. The inversion of NL layers is illustrated in Figure 4 with a “-1” in the NL layer box. Inversion is prevented when the “inv.nl” checkbox in the general parameters Section is unchecked.

NL inversion is not used for AEC (Section 8), stacked RBM (Section 10.2), and UPDN (Section 11), where reconstruction is done by multiplication with the network weight matrix followed application of the simple base non-linearity (see Figure 4). This rule applies to all NL layers except the network input layer. An NL network input layer is always inverted by brute force.

8 Auto-encoder (AEC)

In Figure 1, an auto-encoder consists of the forward path (top row), the reconstruction path (center row), and the reconstruction (input) cost function.

8.1 General

Controls for AEC are located in the AEC Section. To train an auto-encoder, press the "AEC" button once to compile. Then, enable training by checking the "TRN" checkbox. Stop training by un-checking the "TRN" checkbox. Hidden variables can be seen using the "FWD" button with PLT checked (see Sec. 6). To view re-synthesized visible data, check the "PLT" check box in the General Parameters Section, then press "SYN" button in the "AEC" section. This displays the first 10 samples of a batch of data. The number of samples to display can be changes using the "n" field in the PBN Synth Section.

The "EVAL" button will evaluate the AEC using the data partition selected in the Forward Section (train, valid, test, AA), on the data sets specified in the "EVAL DATA" field in the Data Section (comma separated list). Mean square reconstruction error will be printed. By checking the "SAVE" checkbox in the General Parameters Section, evaluation results will be saved to files.

The "DEC" button runs just the decoder network on hidden variables read from an external file "zin.mat", which must be of shape batchsize-by-dim, where dim is the network output dimension.

The "SYN" button synthesizes data from the auto-encoder. Synthesis occurs from hidden variables out of the forward network using the data partition selected in the Forward Section.

The "DEC" button runs just the decoder network on hidden variables read from an external file "zin.mat", which must be of shape batchsize-by-dim, where dim is the network output dimension. But, if the "rnd" checkbox is checked in the PBN SYN Section, then "zin.mat" is ignored and random synthesis is used. The "gmm" checkbox also works as in Section 12.4, but the GMM needs to be set up and traned as in Section 15.

8.2 Variational Auto-encoder (VAE)

To compile a beta variational auto-encoder (beta-VAE), enter a beta value greater than zero in the "VAE" field, then re-compile by pressing "AEC". If zero, a standard AEC will result. For $\beta = 1$, a standard VAE will be compiled, and for $\beta > 1$, it will be a beta-VAE. Note that for a VAE, the chosen output non-linearity for the last layer is ignored - because a VAE does not use a non-linearity in the output layer, which is a special type of layer that defines a probability density of the output variables.

8.3 Denoising Auto-encoder

To create a denoising autoencoder, enter noise standard deviation into the "STDV" field in the AEC Section. This feature works only if the input nonlin = 0 (Gaussian).

8.4 Reconstruction Bias and Weights

The controls "r.bias" and "r.wts" in the General Parameters Section allows separate reconstruction bias and weights for the AEC and some other models. The PBN-Tk does not normally use a separate reconstruction bias parameter, nor does it use a separate reconstruction weight matrix. For reconstruction, it just uses the transpose of the analysis weights. This is called "tied" weights. When using tied weights ("r.wts" not checked), it is a good idea to enable training of the "s2" parameters. To do this, check "upd.s2" and "upd.s2in" in the PBN section, then compile AEC. Then, when training AEC, the "s2" parameters will change. For a beter explanation, see Section 19.

8.5 Classifying Autoencoder

For classifier networks (See section 6.2), the last layer is **not** used for autoencoding. In this case, you can enter a constant in the "XE" field in the PBN Section. This will add cross-entropy classifier cost function to the training, and create an autoencoder/classifier. Classifier mode can be disabled for some algorithms (PBN,AEC,DPBN,UPDN) by un-checking the "class" button in the PBN Section.

8.6 AEC Derivatives

The derivatives computed by THEANO for the AEC cost function can be checked numerically using the Derivative Section, as described for PBN and DPBN in Section 12.5.

9 Deep Neural Network (DNN) Classifier

9.1 DNN Basics

In Figure 1, an DNN consists of the forward path (top row), and the output classifier cost function.

To train a DNN, you first need data and a network that is compatible with DNN. Data requirements: For a classification experiment, the data file must have nclass > 1 and include the 'tch' variable (see Section 3.2). To see validation or testing results, the variables 'Xv','tchv' , 'X2','tch2' must be included, respectively. The last layer must be fully connected with an output dimension (See "nchan_out" in Section 4.1) equal to the number of data classes (See "nclass" in Section 3.2). The output nonlinearity of the last layer does not matter, it will be set to Softmax.

To train the network as a classifier using cross-entropy cost function, press the "DNN" button once to compile the function. Then, enable training by checking the "TRN" checkbox. Stop training by un-checking the "TRN" checkbox again. Data dropout regularization is specified for each layer in the model file (see Section 4.1), and can be temporarily over-ridden in the Network Parameter Table. You must press "APPLY" for the changes to take effect, but you do not need to re-compile. Classifier results and hidden variables can be seen using the "FWD" button (see Sec 6 above).

If you want to use mean-square error cost function (i.e. not cross-entropy), then un-check the "x.ent" checkbox prior to compiling.

To use 'max pooling' instead of straight down-sampling in convolutional layers, check the 'maxp' checkbox in the DNN section. The DNN and FWD functions must be re-compiled after changing this. It is sometimes useful to train some of the layers and leave other layers unchanged. This is done by unchecking "Update" for some layers. After changing this click APPLY, then re-compile. You will get better DNN performance with MAX-Pooling. Max pooling is not compatible with any other models so uncheck "maxp" before using PBN, AEC, or any other algorithms.

Different types of regularization are available to create better models. For weight decay, set "DECAY" to a value like 0.999 while training. When using lots of decay, you will need a large learning rate, such as 1e-3. Instead of decay, you can use L2 regularization by setting "L2.REG" to a value like 1e-5. If you want to use dropout regularzation, specify the input dropout probability in network parameter table, then press "APPLY". You can also set them permanently in the model defintion file (See "input_dropout_fac" in Section 4.1).

For better DNN initialization, it is recommended to start with a stacked RBM, and/or UPDN, although it is also fine to start with random weights. To get random weights, just delete the model files as explained in Section 4.1.

9.2 DNN Cost Function

The DNN normally uses a softmax output non-linearity (over-riding whatever non-linearity was chosen) followed by cross-entropy cost. However, in Streaming data mode (Section 3.2.3), he default is mean square error cost. However, if "class" checkbox in the PBN Section is checked, then the ground truth data will be normalized so it sums to 1, then used as a probability in conjunction with the cross-entropy cost.

10 Restricted Boltzmann machine (RBM)

In Figure 1, an RBM consists of a 1-layer segment of the forward path (top row), and the corresponding 1-layer segment of the reconstruction path (center row). This is shown as a circular path in the figure.

Note that the RBM algorithm only works with Default data type. If you are using Big or Streaming data, you can train stacked RBMs with the UPDN algorithm (Section 11).

10.1 Training an RBM

RBM controls are located in the RBM section. Training an RBM is fundamentally different than training any other algorithm in the Toolkit because the "TRN" button functions differently. It is not possible to start and stop training at will. Pressing "TRN" button trains for a selected number of epochs. To train an RBM, select the layer number (starting at 0) in the "LYR" field. To train, press "TRN". The algorithm will stop after "EP" epochs. To clear the compiled functions, press the "RBM" button, then the RBMs will be re-compiled the next time "TRN" is pressed. Normally, deterministic sampling is used by the RBM, replacing stochastic sampling by the distribution mean, which corresponds to the activation function chosen. Stochastic sampling will be used if the 'rnd' checkbox is checked. If the "Wts" checkbox is unchecked, the weights will not be updated, only the bias will be changed. This is useful during initialization. For stochastic sampling, check the "rnd" checkbox. Activation functions and sampling distributions are:

nonlin#	Act. Fn.	Sampling Distribution
0	linear	Gaussian
1	sigmoid	Binary (Bernoulli)
5	TED	Truncated Exponential
10	TG	Truncated Gaussian

and can be separately chosen for input and output distribution of the layer.

The "NK" field controls the number of Gibbs iterations. It should be set to 1 except when training a DBN (See Section 10.5).

10.2 Stacked RBM

To train a stacked RBM, keep increasing "LYR" and re-training . A network trained as a stacked RBM can reconstruct visible data similar to an AEC. To re-synthesize visible data from a stacked RBM, first compile the autoencoder (AEC) with only the layers you want to use. You can disable layers by entering a space in the Activation field in the Network parameter Table, then pressing APPLY. After compiling the AEC, use the "SYN" button in the AEC section to re-synthesite data. Note that reconstruction bias is not generally used in PBN-Tk. A bias variable is only defined for the forward path. Using a reconstruction bias is unnecessary because the effect can be achieved using an extra column in the weight matrix. A reconstruction bias is only used for the deep belief network (DBN) top layer. However, using a reconstruction bias can be forced by checking the "r.bias" checkbox in the General Parameters Section.

10.3 RBM with MAX-Pooling (MXP).

To create an RBM using max-pooling (with pooling positional information used in the forward path is stored for data reconstruction), use the TRN button in the MXP part of the RBM Section, which operates like the normal "TRN" button. To re-synthesize visible data from a stacked RBM with max-pooling, use the "SYN" button in the MXP part of the RBM Section.

10.4 RBM with NL layers.

When a layer is followed by a "nl" layer type, the RBM training algorithm will use the NL in the Gibbs sampling of the forward path. If stochastic sampling is enabled, the "nl" layer type defines a mixture distribution (See [3]).

10.5 Deep Belief Network (DBN)

A DBN is a stacked RBM where the data labels are injected into the data of the last (top) layer. Training a DBN estimates a joint probability distribution (Gibbs distribution) between data and labels, and can be used to classify data [Hinton 2006]. To train a DBN, first train all the layers not including the top (last) layer as a stacked RBM (Section 10.2). The last layer must be of type "dbn" (See section 4.1). The DBN layer is trained exactly like an RBM layer, except there is the additional control "NK" which sets the number of Gibbs iterations in the top layer. When the last layer is a DBN, then the DBN classifier will be evaluated when the forward function is run (See Section 6.2).

When a non-zero value is entered in the "XE" field in the PBN Section, a cross-entropy classifier cost based on the free-energy DBN classifier is added to the cost function for training (multiplied by this constant). To use this feature, put a non-zero value in the "XE" field and re-compile the RBM (by pressing "RBM" once, then training). Once compiled with a non-zero value in the "XE" field, you can change the XE value without re-compiling. This feature can produce much better classifer performance.

To synthesize random data from a DBN, use the "DBN-SYN" button in the RBM Section. This will initialize the top layer with random numbers, then apply NK Gibbs iterations to the top layer, then re-construct visible data. The procedure to train a DBN is: train the stacked RBM layer by layer using RBM, then train the entire network using UPDN (Section 11), then finally train the top layer DBN using NK=4. When the "dpbn" checkbox is checked, data will be synthesized using the DPBN.

Training a DBN top-layer using the RBM Section trains just the top layer. You can also train the entire deep belief network using UPDN (Section 11).

11 Up-Down Algorithm (UPDN)

Several layers of stacked RBMs and DBNs can be fine-tuned with the Up-Down algorithm [4]. In Figure 1, the UPDN algorithm trains the forward path (top row), and the reconstruction path (center row). The UPDN algorithm makes an excellent initial set of parameters for PBN and DPBN. Controls are in the UPDN Section. To train using up-down, press "UPDN" once to compile the function. Then, enable training by checking the "TRN" checkbox. Stop training by un-checking the "TRN" checkbox. The visible data reconstruction error is printed each iteration. Use "SYN" in the AEC Section to test reconstruction. Normally, deterministic sampling is used unless the "rnd" checkbox is set.

You can also train a stacked RBM using the UPDN algorithm if you enable only updates to the last layer, which is controlled by the Update checkboxes in the Network Parameter Table.

When the top layer is a DBN, some of the controls in the RBM Section affect the top layer. These include the number of Gibbs iteration set by the "NK" field, and the "rnd" checkbox. For example, if you want deterministic iterations in all layers except the top layer, and you want stochastic iterations in the top layer, then uncheck the "rnd" checkbox in the UPDN Section, and check the "rnd" checkbox in the RBM section. For more information about DBNs, see Section 10.5.

When a non-zero value is entered in the "XE" field in the PBN Section, a cross-entropy classifier cost based on the free-energy DBN classifier of the top layer is added to the cost function for training (multiplied by this constant). This trains the entire network in order to improve the free-energy classifier result of the top layer! To use this feature, put a non-zero value in the "XE" field and re-compile UPDN. Once compiled with a non-zero value in the "XE" field, you can change the XE value without re-compiling. This feature can produce much better classifer performance.

If the last layer is a classification layer (See Section 6.2), then this layer will not be trained as part of the UPDN algorithm, but instead as a classifier. The "XE" value in the PBN Section controls the amount of cross entropy added to the cost function. Cross-entropy cost is computed from the free-energy classiciation of the ast layer (top-level RBM). This mode can be disabled by un-checking "class" in the PBN Section. Note that you need to re-compile UPDN if "XE" is made non-zero, but "XE" can change in value without re-compiling.

By checking the “upd.s2” and “upd.s2in” check boxes in the PBN Section, the scale factor “s2” can be trained. However, when “upd.s2” is trained with the UPDN algorithm, you must disable “fwd.s2”. Networks trained with “fwd.s2” in one state are not compatible with networks trained in the other state. See Section 19.

12 Projected belief network (PBN)

In Figure 1, the PBN algorithm trains just the forward path (top row), including the output distribution $g(\mathbf{z})$.

12.1 Compiling and Training a PBN

PBN controls are located in the PBN Section. Before training, you need to compile. Press ”PBN” once to compile (takes a long time). Although the PBN will work on randomly-initialized weights, it saves time to start with good initial parameters obtained using stacked RBM (Section 10.2) or UPDN (Section 11) .

The PBN is based on the SPA (Section 20). Therefore, it is necessary to insure that the SPA is correctly estimated. This is greatly helped by SPA prediction (Section 12.6). Run the SPA prediction until the initial SPA error is small (about 1e-3 if possible). Then, it is ready for PBN training. Enable PBN training by checking the ”TRN” checkbox. Stop training by un-checking the ”TRN” checkbox. As the PBN trains, make sure the “e2z2” is small (near machine precision). You can also see the “e2z” value, which is the initial SPA error provided by SPA prediction. If the “e2z” value is large, repeat the SPA prediction training. Once SPA prediction is small, it should stay small since SPA prediction is updated as the PBN trains. Gaussian layers (with input nonlin=0) do not use SPA, so the error is fixed and cannot be improved.

If there still are problems with SPA error, try increasing “NIT” in the SPA Section, then re-compile.

A PBN can be a classifier at the same time (Section 6.2). Classifier results and hidden variables can be seen using the ”FWD” button (see above).

It is sometimes useful to train some of the layers and leave other layers unchanged. This especially true near the end of convergence. The likelihood function of a PBN is dominated by the first layers. To concentrate on the end layers, disable updating the first few layers. This is done by unchecking ”Update” for some layers. After changing this click APPLY, then re-compile.

Additional PBN controls:

- **”upd.s2in”** : For data where the input activation (of layer 0) equals 0 or 10 , i.e. Gaussian or truncated Gaussian, which is specified in model definition script – see Section 4.1), you can enable estimating and updating the input variance (otherwise s2 =1). See Section 19 for more information on the “s2” parameter.
- **”upd.s2/v”** : Same as above, for remaining layers.
- **”save”** : Saves network parameters eact 10 epochs. Also affects other algorithms other than PBN, such as DPBN.
- **”plt”** : Enabling this checkbox will cause the Toolkit to plot cost as a function of epochs. Also affects DPBN and AEC algorithms.
- **”XE”** : For classification experiments, adds categorical cross-entropy (times the factor ”XE”) to the PBN cost function. Only has an effect for classification experiments. This can be used together with or as alternative to using ”C”. For more information, see Section 6.2.
- **”C”** : This factor controls a class-depenent prior output density and specifies how much discriminative influence is used in PBN training. This is an alternative to using ”XE”. Only has an effect for classification experiments. For more information, see Section 6.2.

- **"class"** : When un-checked , disables classifier mode in PBN, AEC, UPDN, DPBN.

As the PBN trains, various quantities are printed out. The total log-likelihood (J) is the main quantity, usually a large negative number, than must increase (become less negative or more positive). The change in each epoch (del) is also printed. The Saddle-point errors (e2z, e2z2) are also printed. Make sure these are OK (See Section 20). The variance parameter (s2) of each layer is printed (See 19). Train until J stops increasing. Check the derivatives if there are problems (Section 12.5). Non-increasing J can be caused by too-high learning rate, too high L2 regularization, too much DECAY, too high SPA error, or bad initial weights. The best way to initialize a PBN is with stacked RBM (section 10), followed by UPDN (section 11). For better parameters, use some DECAY (example DECAY=0.9999) or L2.REG (example: L2.REG= 0.05).

12.2 Using the Beta parameter

You may set the beta parameter in each layer in the Network Parameter Table. This factor, normally one, weights the output distribution of the layer. Each layer applies the PDF projection theorem, in which

$$p(\mathbf{x}) = \frac{p_0(\mathbf{x})}{p_0(\mathbf{z})} \cdot \beta \cdot g(\mathbf{z}), \tag{1}$$

where $g(\mathbf{z})$ is the "given" distribution of the layer output variable, which in a multi-layer network is, itself, written like equation (1) recursively. For this reason, the beta parameters of the layers multiply together, so if three layers in a row have $\beta = .1$, then the weighting appllied to the $g(\mathbf{z})$ of the last layer is 0.001. Therefore, with more than one layer with small beta, training the PBN will all but ignore the last layers. The parameter β is normally 1. When training a layer using PDF projection (i.e. a PBN), it will strike a compromise between (a) extracting information, and (b) achieving the desired outiut distribution $g(\mathbf{z})$. But, when beta is set to a small number, such as $\beta = 0.01$, then training the layer will extract as much information as possible from the input data $\mathbf{x}$, but will not put effort into achieve the desired output distribution. When using beta less than 1, it is recommended to train only that layer by unchecking 'Update' in the remaining layers. After a layer has been trained with low beta, set it to 1 again and disable further training for that layer.

Note that changing beta or 'Update' in any layer requires re-compiling PBN.

12.3 Evaluating a PBN and Acid Test

To evaluate the log-likelihood function of a PBN, press the "EVAL" button in the PBN Section. This evaluates the data partition selected in the Forward Section, and will save results if the "SAVE" button in the General Parameters Section is checked.

If the "PLT" button is checked in the General Parameters Section while EVAL is pressed, the Toolkit will run and display the results of an ACID TEST. An Acid Test is a complete end-to-end test of the PBN as a statistical model. First, the MATLAB utility `make_acid_dataset.m` is used to generate some data under a precicely-known data distribution. You can specify input nonlinearities 0, 5, 10, and 4 (Gaussian, TED, truncated Gaussian, and Exponential, respectively). The MATLAB utility saves the data under some filename such as "acid0.mat", "acid4.mat", "acid5.mat", or "acid10.mat", then is loaded into the toolkit by entering "acid0", "acid4", "acid5", or "acid10" into the TRAIN DATA field and pressing LOAD. Then, a simple model is imported. For this purpose, enter "acid" into the MODEL field and press LOAD. This will load the model file `models/acid.py`. The idea is to train the PBN, then evaluate by pressing EVAL with PLT checked in the General Parameters Section. The Toolkit will display on a graph, the theoretical likelihood values plotted against the estimated values from the PBN. The theoretical values are imported from the data det file as variable `lpx`, `lpxv`,`lpx2`, depending on the selected data partition. The values should, more or less, appear on a straight X=Y line on the graph. This has been tested for all four input nonlinearities, which validates the calculation of the log-likelihood values.

12.4 PBN Synthesis

Controls are located in the PBN Synthesis Section. To compile the PBN synthesis functions, press ”SYN”. To re-synthesize visible data after it has passed through the network, press ”GO”. The field ”LYR” determines the last layer to be used. The output of this layer will be used to start reconstruction (layers start at 0). For example, with LYR=0, the visible data will be re-constructed after passing only through first layer. In the trivial case where LYR= -1, the reconstructed visible data will equal the visible data itself. The parameter “n” determines the number of samples to reconstruct for display (when PLT checkbox is set on the top right of PBN-Tk window).

Random synthesis. If the ’rnd’ checkbox is selected, synthesis will start with randomly-generated hidden variables. The type of random distribution will be selected to match the output activation function. Gaussian for nonlin = 0 (Gaussian), uniform for nonlin=5 (TED) or 1 (Sigmoid), and truncated Gaussian for nonlin=10. If the ’gmm’ checkbox is selected as well, the GMM (Section 15) will be used to synthesize data. Make sure a GMM has been created and trained for the selected layer before doing this. Note that the GMM will train on and synthesize data before any activation function. To synthesize visible data directly from GMM, use LYR=-1.

If the ’sort’ checkbox is selected, syntheic data will be sorted in order of increasing log-likelihood value. To use this, you need to compile PBN first.

Data is synthesized working backward from the output of layer ‘LYR”. In each layer, you may choose to use uniform manifold sampling (UMS) by checking the “UMS” ckeckbox for that layer in the Network Parameter table.

12.5 PBN and DPBN Derivatives

This is controlled by the Derivative Section. To test the calculation of derivatives, select the layer you want to test (use ”LYR” in PBN Synthesis Section), and press ”PBN” or ”DPBN”. The number of tests is specified by the epoch count ’EP’ in the RBM section. At first, it is recommended to use just 10 samples. The type of derivative is selected by the radio buttons ”W” is for weight matrix, “b” is for bias, ”s2” for variance. Use ”L.RATE” (General Parameters Section) to adjust the ”delta” of the numerical derivative. Check ’PLT’ checkbox if you want to display the results. The ’TRN’ function is a simplified PBN OR DPBN training (uses just one batch). If BS is set to the entire training data size, it will create a simplified gradient training algorithm. The number of iteration (epochs) is set by ’EP’ in the RBM Section.

12.6 Saddle-Point (SP) prediction training

These controls are in the SPA Section. In the background, training the PBN or DPBN requires computing the saddle point, which is an iterative algorithm, This algorithm needs an initial SP estimate (normally just 0). It is more efficient to obtain an initial estimate of the saddle point using a simple neural network. These initial SP neural networks are trained automatically when either PBN of DPBN are trained, but can be separately trained using this function. The number of iterations in the SP estimation (not the initial SP neural networks) is controlled by ’NIT’. After changing NIT of these, you must re-compile (SP, PBN, DPBN).

To use SP training, click ’SP’ to compile, then ’TRN’ to start and stop training. It is recommended to increase the L.RATE when training SP.

12.7 Creating Data Sets from Features

The Toolkit provides the capability to create new data sets from the output of any layer of an existing network. Controls are in the Create Data Section. To do this, select the desired layer output in the LYR field in the GMM Section (starting at 0), type in the data set name in the DATA SET field, and press CREATE. Output for all data partitions (train, test, validate) will be saved to [data set].mat. For file format, see Section 3.2.

12.8 PBN Classifier

The PBN-Tk supports training a network both as a PBN and as a classifier at the same time. If the network output dimension equals the number of classes 'nclass', classifier mode is automatically enabled. To add discriminative influence to the training, use either cross-entropy ('XE') (suggest value of 1000) or 'C' (suggest value 2). This mode is disabled by un-checking "class".

13 Deterministic projected belief network (DPBN)

In Figure 1, the DPBN algorithm trains the forward path (top row), and the back-projection path (bottom row) using the input cost function.

These controls are in the DPBN Section. The deterministic projected belief network (DPBN) is a set of stacked deterministic PBN layers. It can be used to reconstruct visible data from hidden variables deep in the network and trained as a type of auto-encoder. You can control the depth of the DPBN by selecting the layer number "LYR" (use 0 for a 1-layer network, 1 for a 2-layer network, etc..)

To train a DPBN, press "DPBN" to compile, the check "TRAIN" to start training, or to stop training.

The DPBN, like the PBN, is based on the SPA (Section 20). Therefore, it is necessary to insure that the SPA is correctly estimated. This is greatly helped by SPA prediction (Section 12.6). Run the SPA prediction until the initial SPA error is small (about 1e-3 if possible).

The DPBN is far more sensitive to initialization than the PBN. Any network, even a network with random weights can be trained as a PBN. But a DPBN is different. It suffers from failed samples. A failed sample occurs when the saddle point cannot be found. This can only occur in the DPBN or in PBN Synthesis, if the feature value presented to the SPA algorithm is not a derived from a sample at the input of the layer. In DPBN and in PBN Synthesis, the data propagates from the end of the network, back to the visible data. Therefore, at each layer there is the possibility of SPA failure. Luckily, this can almost always be brought to zero (success probability to 1.0) through training.

As the DPBN trains, the toolkit prints out "frac", the fraction of successful sampling. This should be 1.000. The "trick" of training a DPBN is to get the "frac" value up to 1.000. Then, training will be easy. Tricks to get "frac" value higher include (a) start with random weights, but with very small values (by scaling weights in each layer by 0.1 - see Section 16.3), or (b) pre-train using stacked RBM (10), or (c) first train the network as a PBN (Section 12). Another good approach is to get a shortened network to work first, then pre-train the added layer as an RBM before adding the new layer to the DPBN.

As the DPBN trains, it prints out the SPA error "e2z2", which should be small (near machine precision). If there are problems with SPA error, try SPA prediction training, or try increasing "NIT" in the SPA Section, then recompile. Once SPA prediction error is small, it should stay small since SPA prediction is updated as the DPBN trains.

The PBN-Tk supports training a network both as a DPBN and as a classifier at the same time. First, the network output dimension must equal the number of classes 'nclass'. Then, classifier mode is automatically enabled by PBN-Tk, but can be disabled by un-checking "class" in the PBN Section. To add discriminative influence to the DPBN training, use 'XE' in the PBN Section (suggest value of 100). The EVAL button evaluates the data partition selected in the Forward Section, and may display or save results according to the "PLT" and "SAVE" buttons in the General Parameters Section. The DEC button will use the DPBN to decode (reconstruct) hidden variable data stored in the file "zin.mat", which must be of shape batchsize-by-dim, where dim is the network output dimension.

14 Plotting during training

You can monitor the progress of PBN, DPBN, AEC, and UPDN training using the "PLT" checkbox in the PBN Section. When training these algorithms, a graph of history of (a) the cost function, (b) classification errors (if available), and (c) the reconstruction is displayed. For the PBN and DPBN reconstruction error, you need to specify the network depth for reconstruction in the PBN Syn Section. To clear plotting history, press "Refresh" (near top right of PBN-Tk window) while the "PLT" checkbox (just to the left of Refresh) is checked.

15 Gaussian Mixture Model (GMM)

These controls are in the GMM Section. PBN-Tk has the capability to create a GMM to estimate the distribution of the network hidden variables. To use the GMM, first set the layer in the LYR field (layers start at 0) and set the "Mem" checkbox in the Forward Section. This will store the correct layer output in the GMM memory. The GMM will train on the output of this layer (actually, the GMM trains on the output of the linear transformation before bias and activation function). Use LYR=-1 to train the GMM on visible data. Next, select training data checkbox in the Forward section, and then 'GO'. This will create and save the training data for the GMM. Next, set the number of GMM components 'M' and press 'GMM' to create the GMM parameters, then 'TRN' to train the GMM. Keep pressing 'TRN' until the log-likelihood stops increasing.

The field 'MIN-S' refers to the minimum standard deviation. The square of this value is added to the diagonal of the covariance matrixces to prevent ill-conditioning.

GMM classier. If the number of classes ('nclass' in Section 3.2) is greater than 1, then the GMM can work as a classifier. A separate GMM will be trainined on each class data. To use the GMM in a classifier, select 'gmm' checkbox in the 'FWD' output activation section section. When 'GO' in the Forward Section is pressed, the GMM likelihood classifier output will be shown and classification error printed.

The GMM can also be used to synthesize hidden variables, for PBN data synthesis (Section 12.4) If LYR is set to (blank), the GMM will train and synthesize visible data directly.

16 Weights Section

This section allows simple operations on the weight and bias parameters of a layer. First select the layer number in the "LYR" field (layers start at 0).

16.1 Nrm: normalizing a layer

Press "Nrm" to normalize the columns of the weight matrix and adjust bias to produce zero-mean, constant variance hidden variables at the input to the activation function. May only work for dense (FC) layers. Note that you can always normalize that last layer of a DPBN using Nrm since it has no effect on re-synthesis. This also works to initialize a NL layer (Section 7).

16.2 Show: viewing weights

Press "Show" to plot the weights.

16.3 Scale: scaling weights

Enter a scale factor in the "Fac" field, then press "Scale". This scales the network layer weights and bias vectors by this factor.

16.4 PCA: initializing weights

Press the "PCA" button to initialize the weights using principle component analysis (PCA). Only works for FC layers.

16.5 Zro: zeroing weights

Press the “Zro” button to set the bias of the layer so as to produce a zero-mean output.

16.6 setting s2: setting variance parameter

This manually sets the “s2” parameter to the desired value. Enter the value, then press “Set”. The “s2” parameter is explained in Section 19.

17 Scripting

The Toolkit permits a primitive scripting language with simple looping to execute a sequence of commands without bringing up the graphical interface. This is designed to make it easier to do repeated tasks (training many models) in batches. To use scripting, add the filename of the script to the python command:

```
python pbntk.py script.txt
```

When a script file is given, the graphical user interface is suppressed. An example of a script follows:

```
SCRIPT PARAM cWt 0
SCRIPT PARAM L.RATE 1e-4
SCRIPT PARAM XE 1000
SCRIPT PARAM upd.s2in 1
SCRIPT PARAM BS 125
#
SCRIPT LOADDATA esc50_mel4a
SCRIPT LOADMODEL esc3
SCRIPT PBNINIT
#
SCRIPT STARTLOOP 50
SCRIPT    PARAM cWt II+3
SCRIPT    LOADMODEL esc3
SCRIPT    PARAM L.RATE 1e-6
SCRIPT    PBNTRAIN 2
SCRIPT    PARAM L.RATE 1e-5
SCRIPT    PBNTRAIN 20
SCRIPT    SAVEMODEL esc3
SCRIPT ENDLOOP
SCRIPT QUIT
```

Any line beginning with “SCRIPT” will be executed. The “PARAM” command allows you to set any parameter that can be set with the GUI. The looping function begins with STARTLOOP and ends with ENDLOOP. The STARTLOOP command specifies the number of times to loop. Whenever the character pair “II” (loop counter) is seen, this is replaced by the loop counter value (0, 1, 2, ...), however when the “II+k” is encountered, where ”k” is an integer, it is replaced by the loop counter value plus k (k, k+1, k+2,...) The following example loads 100 different data sets, with names “dataset1”, “dataset2”, “dataset3”, etc, trains using PBN for 100 epochs then saves the model.

```
SCRIPT STARTLOOP 100
SCRIPT    LOADDATA datasetII+1
SCRIPT    LOADMODEL esc3
SCRIPT    PARAM L.RATE 1e-5
SCRIPT    PBNTRAIN 100
SCRIPT    SAVEMODEL esc3
SCRIPT ENDLOOP
```

A partial list of commands is:

```
SCRIPT QUIT
SCRIPT LOADDATA [data class]
SCRIPT LOADMODEL [model class]
SCRIPT Refresh
SCRIPT PBNINIT
SCRIPT PPINIT
SCRIPT DNNINIT
SCRIPT DPBNINIT [last layer]
SCRIPT DPBNTRAIN [epochs]
SCRIPT PBNTRAIN [epochs]
SCRIPT PBNEVAL [data class]
SCRIPT DNNTRAIN [epochs]
SCRIPT FWD [data class to evaluate (optional)]
SCRIPT FWDINIT
SCRIPT SAVEMODEL [model class (optional)]
SCRIPT DELETE [model class]
SCRIPT RBMINIT
SCRIPT UPDNINIT
SCRIPT UPDNTRAIN [epochs]
SCRIPT RBMTRAIN [epochs]
SCRIPT PARAM [param type] [value 1] [value 2]
   param types:
       Dropout [lyr] [0/1]
       maxp [0/1]
       Seed [seed]
       do_class [0/1]  # controls "class" checkbox in PBN Section
       CIRC [1st dim] [2nd dim]
       SHFT [1st dim] [2nd dim]
       Rand [0/1]
       SP_NIT  [nit]
       nlayer [n layers]
       upd.s2 [0/1]
       upd.s2in [0/1]
       EPOCHS [n epochs]
       cWt [class nr]
       dataset_type [Big,Stream,Default]
       partition [train/valid/test]
       L.RATE [lrn rate]
       L2.REG [l2 reg]
       BS [batch size]
       RBMLYR [which RBM layer]
       XE [cross-entropy factor]
       ADD1 [0/1]
       C [PBN C value]
       DECAY [decay value]
       SAVE [0/1] : sets save for output variables
```

```
save : controls saving during training
fst.s2 [0/1]
upd.s2in [0/1]
Update [layer] [0/1]  : allows/prevents parameters from layer to update
```

18 Setting up Convolution

A convolutional layer is specified in the model file using the following required variables:

```
cfg.lyrs[0].type = 'conv'
cfg.lyrs[0].filt_row = 8 # in time
cfg.lyrs[0].filt_col = 8 # in freq
cfg.lyrs[0].pooling = (3,3) # row, col
cfg.lyrs[0].nchan_out = 9
cfg.lyrs[0].border_mode='valid' # 'valid' or 'half'
cfg.lyrs[0].nonlin = (nl_in,nl)
```

The kernel size is specified by "filt_row" and "filt_col". These correspond to the time and frequency dimensions for spectrograms, assuming the data (in MATLAB) is reshaped from a 3-dimensional data matrix of size X(1:nfreq,1:ntime,1:nsamp), which in the Python world is of dimension X[0:nsamp][0:ntime][0:nfreq]. The pooling variable specifies the down-sampling rates (it is not MAX-pooling unless the "maxp" checkbox is checked for DNNs) for the time and frequency dimensions. Variable "nchan_out" specifies the number of kernels (and corresponding number of output feature maps). The "border_mode" variable can be either "valid" or "half" and specify how the edges of the input maps are treated, and how large the output maps will be.

18.1 Border mode "half"

The parameter "border_mode" defines the zero-padding used in convolution. For "half" border mode, the data is surrounded with a border of zeros before convolving. The width of the border is (K-1)/2, where K is the kernel width. This applies separately to each of the two dimensions. By doing this, the convolution output has the same size as the input. This will be illustrated below for a 1-dimensional convolution with a kernel of length K=5 convolving with data of length N=6. For border_mode='half', K must be odd. Let "D" represent a data value, and 'k' a kernel value, and "." be zeros. There is a border of (K-1)/2 = 2 zeros.

```
inputs:       1 2 3 4 5 6
------------------------------
          . . D D D D D D . .
output 1: k k k k k
output 2:   k k k k k
output 3:     k k k k k
output 4:       k k k k k
output 5:         k k k k k
output 6:           k k k k k
```

In the example above, each output is a scalar computed by correlation (multiplying together and summing) the samples "D" and filter values "k". You see there are exactly as many outputs (6) as inputs. That is the idea behind 'half' border mode – preserving the feature map size.

When pooling is considered, it reduces the number of outputs. When pooling (down-sampling) 'p' is considered, we keep every p-th output. In this case, For example, pooling (down-sampling) factor of 2 gives just 3 outputs:

```
inputs:       1 2 3 4 5 6
-------------------------------
          . . D D D D D D . .
output 1: k k k k k
output 2:     k k k k k
output 3:         k k k k k
```

18.2 Border mode "valid"

With 'valid' border mode, there is no zeros-padding, resulting in a smaller convolution output. Consider the case with N=6 and K=4 :

```
inputs:       1 2 3 4 5 6
-------------------------------
              D D D D D D
output 1:     k k k k
output 2:       k k k k
output 3:         k k k k
```

There are just 3 outputs. In general, n_out = N-K+1. When pooling "p=2" is considered:

```
inputs:       1 2 3 4 5 6
-------------------------------
              D D D D D D
output 1:     k k k k
output 2:         k k k k
```

As you see above, the last output reaches exactly to the end of the data. This is necessary for good reconstruction and only happens if (N-K) is divisible by p. In this case, N-K=4, which is divisible by p=2. The PBN-Tk will print an error of this is not right.

19 The "s2" Parameter and Tied vs. Untied Weights

Every activation function (except for NL layers) is associated with a scale factor, called "s2", and written mathematically as σ^2. Suppose $y = f(x)$ is an activation function (such as Sigmoid). In the PBN-Tk, this is always implemented as follows:

$$y = f(x; \sigma^2) = \sigma f(\sigma x).$$

The factor σ, which is the square root of σ^2 is applied twice: once at the input of the activation function, multiplying x, and once at the output of the activation function, multiplying y. For the linear activation (nonlin=0), this simplifies to

$$y = f(x; \sigma^2) = \sigma^2 x.$$

Activation functions are applied in the forward direction (analysis) and in the backward direction (reconstruction) whenever activation functions are applied. For the PBN, σ^2 has a statistical interpretation, but in simpler models such as autoencoder (AEC) or deep neural networks (DNN) or forward network (FWD), it just acts as a scale factor.

Why is a scale factor needed? Suppose the input to the layer is dimension N and the output is dimension M. The M output hidden variables are calculated using

$$h_j = f\left(\sum_{i=1} W_{i,j} x_i + b_j\right),$$

which is written in vector notation as

$$\mathbf{h} = f\left(\mathbf{W}'\mathbf{x} + \mathbf{b}\right).$$

Matrix $\mathbf{W}$ is size $N \times M$ and bias vector $\mathbf{b}$ is dimension $M \times 1$. To reconstruct $\mathbf{x}$, the reconstruction network would use

$$\mathbf{x} = g\left(\mathbf{A}\mathbf{h} + \mathbf{a}\right),$$

where $g(\ )$ is the reconstruction activation function (maybe different from $f(\ )$), and $\mathbf{A}$ is size $N \times M$ reconstruction weights and reconstruction bias vector $\mathbf{a}$ is dimension $N \times 1$. The forward activation of one layer is always the same as the reconstruction activation of the next (down-stream) layer. This is how analysis and reconstruction works without a scale factor "s2".

In the PBN-Tk, we advocate using the same weight matrix in analysis and reconstruction, and in addition, do not advocate using a reconstruction bias. Therefore, we reconstruct using just

$$\mathbf{x} = g\left(\mathbf{W}\mathbf{h}\right).$$

A reconstruction bias vector is not really needed. It carries information about how to reconstruct the data, but this information is not present in the hidden variables. We could just as well add vector $\mathbf{a}$ as a new column to matrix $\mathbf{W}$. Then, it would be used for reconstruction, but also for analysis. A similar argument can be made for using the same weight matrix for analysis and reconstruction.

The only problem that can occur when using the same weight matrix for analysis and reconstruction happens when the input data is very large or very small in value. To handle this, it is necessary to use a scale factor in the reconstruction path. In the PBN-Tk, the analysis and reconstruction steps including scale factor are:

$$\mathbf{h} = \sigma_2 f\left(\mathbf{W}'(\sigma_2\mathbf{x}) + \mathbf{b}\right), \quad \mathbf{x} = \sigma_1 g\left(\mathbf{W}(\sigma_1\mathbf{h})\right),$$

where σ_2 is square root of the "s2" parameters of **next** layer, and where σ_1 is square root of the "s2" parameters of **current** layer. Note that the square root of the "s2" parameter is applied twice: before the activation function, and after. For the linear activation, $f(x;\sigma^2) = \sigma^2 x$, the scalng is by σ^2 Notice also that σ_1 is only used for reconstruction, and σ^2 is only used at the output of the layer. Therefore, if $\mathbf{x}$ has very high amplitudes in comparison with $\mathbf{h}$, this is accounted for by a large σ_1. In the PBN, σ^2 has an interpretation as the layer input variance. For the last layer, there is no "next layer", and σ_2 is assumed to be 1.

One must be careful when allowing σ^2 of an intermediate layer to change during training. This is because of σ^2 changes, then the next time the forward algorithm is run, a different scale factor will be used, changing the values of the hidden variables at the input to the layer. It is therefore recommended **not** to check "upd.s2" while training an RBM of an intermediate layer. The input layer does not have this problem, so "upd.s2in" can always be checked.

If it is absolutely necessary to update σ^2 of an intermediate layer, then uncheck the "fwd.s2" button and recompile. Then, σ^2 will always be assumed to equal 1 in the forward direction, and will only affect the reconstruction calculation. This is useful, for example for training the top-layer RBM in a DBN. But, be sure that $\sigma^2 = 1$ for all the other intermediate layers. The "s2" parameter can be manually reset in the Weights Section (Section 16). The input layer (layer 0) can always have $\sigma^2 \neq 1$.

Also, if it is desired to use a separate reconstruction weights, check the "r.wts" checkbox (and recompile). Also, if it is desired to use a reconstruction bias, check the "r.bias" checkbox (and recompile).

20 Saddle-Point Prediction and Estimation for PBN and DPBN

Saddle-Point approximation (SPA) is central to PBN and DPBN algorithms. Let $\mathbf{x}$ be an $N \times 1$ layer input vector. When the $N \times M$ weight matrix $\mathbf{W}$ is multiplied by $\mathbf{x}$, an $M \times 1$ feature $\mathbf{z}$ results:

$$\mathbf{z} = \mathbf{W}'\mathbf{x}.$$

It is assumed that $M < N$, so that $\mathbf{z}$ has smaller dimension than $\mathbf{x}$ (the layer reduces the dimension). Vector $\mathbf{z}$ is the output of the linear transformation of one network layer, before we have applied a bias and activation function. Now suppose that $\mathbf{x}$ is not known but we know $\mathbf{z}$, and we'd like to reconstruct $\mathbf{x}$. The SPA provides an "optimal" way to do this assuming that $\mathbf{x}$ is a random vector from some distribution.

We assume that $\mathbf{x}$ is a vector of length N, written $\mathbf{x} = [x_1, x_2 \dots x_N]$, where x_i are drawn independently from a distribution $x_i \sim p(x; \alpha_i)$ based on parameters α_i. This is the assumed sampling distribution for the layer input. In PBN-Tk, we assume either Gaussian, truncated Gaussian (TG), or truncated exponential (TED). The sampling parameter $\boldsymbol{\alpha}$ is also a vector of length N, $\boldsymbol{\alpha} = [\alpha_1, \alpha_2 \dots \alpha_N]$. The sampling distribution has a mean that depends on the parmeter given by

$$\lambda(\alpha) = \mathbb{E}\left\{p(x; \alpha)\right\}.$$

This is also known as the activation function corresponding to the given sampling distribution. Assuming that $\mathbf{x}$ was generated this way, the optimal (expected mean) estimate of $\mathbf{x}$ is given by

$$\hat{\mathbf{x}} = \lambda(\mathbf{W}\mathbf{h}),$$

where $\mathbf{h}$ is the saddle point. The saddle point must solve the following formula:

$$\mathbf{W}'\lambda(\mathbf{W}\mathbf{h}) = \mathbf{z}.$$

Since $\mathbf{W}'\hat{\mathbf{x}} = \mathbf{z}$, $\hat{\mathbf{x}}$ must be a good estimate of $\mathbf{x}$ because $\mathbf{z} = \mathbf{W}'\mathbf{x}$. Finding the saddle point is done by a Newton-Raphson iteration to find $\mathbf{h}$ by minimizing the error

$$\|\mathbf{z} - \mathbf{W}'\lambda(\mathbf{W}\mathbf{h})\|^2.$$

When training the PBN or DPBN, the saddle point error, denoted by `e2z2` is printed. It should be very small, ideally near machine precision, $1e-30$ for double precision and $1e-15$ for single precision (the error is relative to $\|\mathbf{z}\|^2$). If it is too large, you can increase the number of iterations ("NIT" in the SPA Section).

To speed up the Newton-Raphson iteration, an initial estimate of $\mathbf{h}$ is made using an *auxiliary* matrix $\mathbf{A}$ that is the same shape as $\mathbf{W}$:

$$\hat{\mathbf{h}} = \mathbf{A}'\mathbf{A}\mathbf{z}.$$

This auxiliary matrix is trained using the SPA training (Section 12.6). This matrix is stored as parameter `Aa` in the parameter files. This matrix is also fine-tuned automatically as the PBN/DPBN trains. When training the PBN, the initial saddle point error (based on initial estimate of $\mathbf{h}$), denoted by `e2z` is printed. This should be around $1e-2$ or better. If not, try to run the SPA training (Section 12.6). However, as long as `e2z2` is small enough, it is not critical. To speed up the algorithm, you can decrease the number of iterations ("NIT" in the SPA Section).

21 Class-dependent PBN Classifier (PBN-C)

The PBN-Tk implements class-dependent PBN classifier (discriminatively aligned PBN). To use this mode, first train a PBN separately on each data class by setting the data class using the class weighting field "cWt" in the Data Section. This can be changed without re-compiling PBN. This is set to the class number (1 through nclass) that you want to train. If set to zero (default), the PBN trains on all classes. Whenever "cWt">0, the names of the parameter files that store the network parameters are inserted with "_cX_", where "X" is the class number. Finally, to run a classifier experiment, press "EVAL" in the PBN-C Section. A class-dependent (discriminatively aligned) D-PBN will be tested if the "dpbn" checkbox is checked in the RBM Section.

22 KNN Classifier

The Toolkit can classify data using a KNN classifier. Set up the layer output to be saved, as in the GMM (Section 15). Then select "K" for the KNN, select data partition, and press GO to see the results.

23 Data Creation

The Toolkit can create data sets from the output of selected network layers. This is more efficient than needing to calculate the network output each time. To use this feature, first check the "Mem" checkbox in the Forward Section so that data is saved in memory. Next, select the desired output layer (starting at 0 to save the output of the first layer) in the LYR field in the GMM Section. Finally, enter the desired data set name and press "Create" in the Data Creation Section. The input non-linearity of the created data set wll correspond ot the output non-linearity of the chosen network layer. A MATLAB (OCTAVE) readable file will result.

24 Advanced PBN Applications

24.1 Expand-Contract (EC) Layer Groups

```
--------- TBD -----------
```

24.2 Gaussian Layer Groups (GG)

```
--------- TBD -----------
```

25 Adversarial Attack (AA)

```
--------- TBD -----------
```

26 Data sets

A number of data sets are provided for use with PBN-Tk to supply some simple problems for learning purposes. These are based on the MNIST data set [5] which is composed of 60000 training samples of the handwritten characters "0" through "9". We have selected just three characters, "3", "8", and "9". In some cases, we have down-sampled the data 2:1 to 14x14 images, and have used only 500 samples of each character. The full-size (28x28) data is also available. To demonstrate the use of different data value ranges, which greatly influences the PBN, we have transformed the data, which normally has pixel values that reside in the range [0,1] to "gaussian" range $[-\infty, \infty]$, the "truncated gaussian" (TG) range, $[0, \infty]$, and the "truncated exponential" (TED) range, which is in [0,1] line the original data, but has some dither. Some data sets have all three characters, and some have just one. All have 500 training samples per character. The various data sets are listed below:

Characters	Data Size	Gaussian Range	TG Range	TED Range
"3"	28x28	mn3_0(ds81)	ds91	ds71
"8"	28x28	mn8_0(ds82)	ds92	ds72
"9"	28x28	mn9_0(ds83)	ds93	ds73
"3,8,9"	28x28	mn389_0(ds818283)	mn389_10(ds919293)	mn389_5(ds717273)
"3"	14x14	ds21	ds31	ds41
"8"	14x14	ds22	ds32	ds42
"9"	14x14	ds23	ds33	ds43
"3,8,9"	14x14	mn389r_0(ds212223)	mn389r_10(ds313233)	mn389r_5(ds414243)

27 Miscelaneous Topics

27.1 Determining Optimal Batch Size

Before loading data, set batch size (BS). By default it is 25 samples, which is OK for CPU and for this small test problem, but you can set it to 125, 250, or even 500 to make best use of the GPU. It is more efficient if the number of testing samples (in this case 1500) is divisible by BS. If this is not possible, make BS slightly larger than a fraction of the testing samples. For example, if there are 410 training samples, use BS=103, but **not** BS=100 because with BS=100, the Toolkit will need 5 batches, but just 4 batches with BS=103. Note also that the bach size cannot be larger than the number of events (samples) in a given data partition. If you set the batch size higher than the number of events in a parttition, you will get an error when trying to use that partition.

27.2 Initializing a DPBN

Using a stacked RBM and/or UPDN algorithm can get the sampling efficiency quite high, even to 1. If this fails, the network can be trained as a PBN, and this will almost always result in an efficiency near or equal to 1.

To test the sampling efficiency, you can reconstruct some data. Compile the SPA synthesis layers by pressing ”SYN” in the PBN-Syn Section. Once compilation is done, reconstruct a few samples by selecting the network depth by entering the layer number in “LYR” field. For example, LYR=0 will reconstruct from layer 0 output, that is after just one layer. Next, press ”SYN”. If the “PLT” box is checked, the reconstructed samples will be displayed. The Toolkit will report the number of failed samples in the batch, so if the sampling efficiency is near one, this should be 0.0 You should succeed in re-constructing samples from the full network depth that you want for the DPBN.

Once you get 0 failed samples on a test batch, it means the sampling efficiency is near 1 and you are ready to train the DPBN. Select the layer depth in the LYR field in the DPBN Section, like you did for synthesis in the PBN-Syn Section. So, set LYR to the last layer, (layer 1 for “tut1” and layer 2 for “tut2”). Then, compile the DPBN algorithm by pressing DPBN. To start and stop training, click “TRN”. Use L.RATE = 1e-4 at first. As the DPBN trains, it reports

```
C(  5)=  0.01385 lp=  -56.56 del= 0.0277224 t=0.54 frac=0.9993 e2z2=[ 7.1e-06, 1.3e-21,] s2=[ 1.00
```

where C is the mean square reconstructed pixel error, lp is the log-likelihood (a probabilistic reconstruction measure) and del is the change in lp, which should be positive. It also reports “frac”, the actual sampling efficiency. In our test, we got “tut1” to have a sampling efficiency of 0.9993 just using stacked RBMs. Sometimes, training the DPBN with frac less that 1 will eventually get to 1.0. In our experiment, frac stayed at 0.9993 (that is by the way just 1 failed sample out of 1500). Sometimes, it is necessary to re-compile the DPBN algorithm if it has trained with frac¡1 because the state variables can become corrupted with large values.

If DPBN training does not result in frac=1, there are some options. If the efficiency is greater than 0.99, you can increase the SPA iteration count by for example using “NIT=12” or higher in the SPA Section, then re-compile both DPBN (and PBN if needed). If this does not work, you will need to train the network as a PBN first.

To train as a PBN, compile the PBN algorithm (press PBN in the PBN Section), then start training by pressing “TRN”. As the DPBN trains, it reports:

```
J( 21)=    198.322 del=  0.045057 t=  0.7 e2z=[ 0.0e+00, 0.0e+00,]
                                          e2z2=[ 2.9e-27, 1.5e-27,]
                                           e2x=[ 1.9e-03, 4.2e-02,]
                                            s2=[   1.000,   1.000,]
```

References

[1] P. Li and P. Nguyen, "On random deep weight-tied autoencoders: Exact asymptotic analysis, phase transitions, and implications to training," *ICLR*, 2019.

[2] P. M. Baggenstoss, "A neural network based on first principles," in *ICASSP 2020, Barcelona (virtual)*, (Barcelona, Spain), Sep 2020.

[3] P. M. Baggenstoss, "Trainable compound activation functions for machine learning," *Accepted at EUSIPCO 2022*, 2022.

[4] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," in *Neural Computation 2006*, 2006.

[5] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov 1998.