PBN Toolit: A Versatile Toolkit for Generative Networks

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

June 2, 2021

Abstract

This document describes the PBN Toolkit.

1 Installation and Prerequisites

1.1 Prerequisites and Installation

—— TBD ———-

1.2 GPU and Smegma Library

—— TBD ———-

1.3 Starting PBN-Tk

—— TBD ———-

2 Basic Operation, Controls and Features

2.1 GUI Layout

Figure 1 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

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".

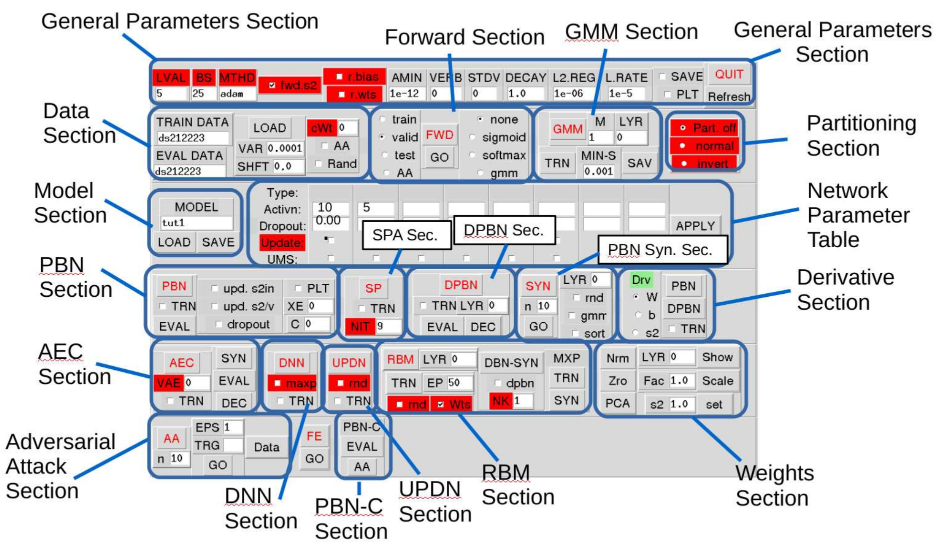

Figure 1: The PBN-Tk Graphical Interface.

GUI Section	Documentation Section	GUI Section	Documentation Section
General Parameters Section	4	Forward Section	5
GMM Section	12	Data Section	2.2
Partitioning Section	10.5	Model Section	3
Network Parameter Table	3.3	PBN Section	10
SPA Section	10.4	DPBN Section	11
PBN Synth Section	10.2	AEC Section	6
DNN Section	7	UPDN Section	9
RBM Section	8	Weights Section	13
Adversarial Attack (AA) Section	18	PBN-C Section	17

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

2.2 Data Input

2.2.1 Creating New Data Sets

To create a new data set, 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 3.1. 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

2.3 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, 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:

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

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

2.4 Additional Data controls

In the Data Section, there are some additional controls.

1. **VAR**: Over-rides the minvar parameter in model file (See Section 3)
2. **AA**: Adversarial Attack (See Section 18)
3. **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 17.
4. **Rand, SHFT**: These affect random shifts and dither 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. Unchecking Rand turns off random shift, checking turns it on. With 'Rand' checked, a Gaussian dither is also applied to the input data, but only for the Gaussian assumption (i.e. nonlin=0 , see Sections 2.2, 3.1). 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.

3 Defining a Network

3.1 The Model File

To define a model with name 'newmodel', you must create the network definition file 'newmodel.py'. 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' or 'fc' or 'dbn'
2. **nchan_out**: integer, number of neurons (columns in weight matrix for fc layers) : or number of kernels (for conv layers)
3. **nonlin**: Two-dimensional tuple of integers that specifies the input and output activations of the layer. Valid 'output' activation are:

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

Only activations 0, 5, and 10 are recommended. The input activation must be the same as the output activation of the previous layer. This will be tested when loading a model and an error will be printed if not right. 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 2.2.

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.
   - '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 14.
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 14.

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)
2. **partition_size** : Controls partitioning (see Section 10.5)
3. **minvar** : This parameter is used when adding random dither and as a variance floor when estimating variance. When not specified, value comes from global minvar parameter (see Section 2.4)

3.2 Model File Example

The following code segment is from "tut1.py" :

```
-----------------------
  nl_in = cfg.nonlin_in
  if nl_in==0:
    nl=10
  else:
    nl = nl_in
  nl_out = nl
```

```
   cfg.lyrs[0].type = 'fc'
   cfg.lyrs[0].minvar = 0.0
   cfg.lyrs[0].nchan_out = 32
   cfg.lyrs[0].nonlin = (nl_in,nl)

   cfg.lyrs[1].type = 'fc'
   cfg.lyrs[1].nchan_out = 16
   cfg.lyrs[1].nonlin = (nl,nl)
   cfg.lyrs[1].input_dropout_fac = .0

   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).

3.3 Loading, Saving, 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]_lyr1.mat,   [newmodel]_[data set]_lyr2.mat, .....
Example:
  tst2_ds81_lyr1.mat
```

Layer numbers begin with zero (0,1,2,...) except when determining filenames, then they start with 1. If you want PBN-Tk to initialize a layer, make sure to delete the file 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. The output activation function 'nonlin' and the 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 10.2) The "Dropout" field allows you to over-ride the input dropout parameter in the network definition script (Section 3.1). The 'Activn' parameter is the output activation of the layer, and over-rides the output nonlin parameter set in the model file (Section 3.1). The input activation of the next layer will be set to this by default. The network can be temporarily shortened by putting a space " " in the "Activn" field to disable a layer. Modifications will not take effect until you press "APPLY". These modifications cannot be saved. If you want to change them permanently, make the changes to the model function [model-name].py

The data weights (or kernels if CONV) can be seen by pressing "Show" in the Weights Section. The layer can be selected using 'LYR' entry box. Layers start at 0.

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 top of the PBN-Tk window is on.

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
```

4 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!
   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. **'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 15 for more information.
5. **'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 15 for more information.
6. **'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 15 for more information.
7. **AMIN**: Factor added to diagonal of the solution matrix before inversion, needed only for training PBN or DPBN, or synthesizing with these.
8. **VERB**: Verbosity. When greater than 0, prints more things out.
9. **STDV** : Standard deviation for denoising autoencoder (works only for input nonlin=0).
10. **DECAY**: Decay factor applied for each batch for layer weight and bias. Use 1.0, 0.9999, etc. Applied each batch.
11. **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.
12. **L.RATE**: Learning rate, suggest 1e-3 for DNN, 1e-4 to 1e-5 for PBN.
13. **SAVE**: Check this to save model parameters after each epoch.
14. **PLT** : (checkbox) Check this for plotting displays when running "FWD" , "SYN" and other methods.
15. **QUIT**: Exit PBN-Tk.
16. **Refresh**: Reload some python modules if they have been changed in the background , also clears plot history (if 'PLT' checked).

5 Forward Section

5.1 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 18). 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 12).

5.2 Classifier function

If the last layer is fully connected and it has an output dimension equal to the number of classes (nclass in Section 2.2), then the classification mode will be automatically enabled. It is necessary also that the 'tch' variable is included in the data (Section 2.2). 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 12 for more about using GMM. If the loaded network is a DBN, the DBN classifier results will be printed and/or plotted (See Section 8.4).

6 Auto-encoder (AEC)

6.1 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. 5). 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.

6.2 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.

6.3 Denoising Auto-encoder

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

6.4 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 15.

6.5 Classifying Autoencoder

For classifier networks (See section 5.2), 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.

7 Deep Neural Network (DNN) Classifier

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 2.2). 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 3.1) equal to the number of data classes (See “nclass” in Section 2.2). 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 3.1), 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 5 above).

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 3.1).

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 3.1.

8 Restricted Boltzmann machine (RBM)

8.1 Training an RBM

RBM controls are located in the RBM section. 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: The "NK" field controls the number of Gibbs iterations. It should be set to 1 except when training a DBN (See Section 8.4).

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

8.2 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.

8.3 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.

8.4 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 8.2). The last layer must be of type "dbn" (See section 3.1). 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 5.2).

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 reconstruct 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 9), 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 9).

9 Up-Down Algorithm (UPDN)

Several layers of stacked RBMs and DBNs can be fine-tuned with the Up-Down algorithm [1]. 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.

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 8.4.

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 5.2), 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.

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 15.

10 Projected belief network (PBN)

10.1 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 8.2) or UPDN (Section 9) .

The PBN is based on the SPA (Section 16). Therefore, it is necessary to insure that the SPA is correctly estimated. This is greatly helped by SPA prediction (Section 10.4). 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 (thise 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 5.2). 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 3.1), you can enable estimating and updating the input variance (otherwise s2 =1). See Section 15 for more information on the “s2” parameter.
- **”upd.s2/v”** : Same as above, for remaining layers.
- **”dropout”** : Enables ”partial dropout” for the layers with non-zero dropout parameter. This feature is obsolete and may not work.
- **”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 5.2.
- **”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 5.2.

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 16). The variance parameter (s2) of each layer is printed (See 15). Train until J stops increasing. Check the derivatives if there are problems (Section 10.3). 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 8), followed by UPDN (section 9). For better parameters, use some DECAY (example DECAY=0.9999) or L2.REG (example: L2.REG= 0.05).

10.2 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 12) 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.

10.3 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.

10.4 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'. The 'HIST' value determines how many SP iterations are analyzed for computing derivatives for PBN and DPBN. After changing either of these, you must re-compile.

To use SP training, click 'SP' to compile, then 'TRN' to start and stop training.

10.5 PBN Partitions

These controls are in the Partition Section. The PBN-TK has the ability to reserve partitions (of the neurons) in each layer for use in generative or discriminative function. The partition sizes can be set in the network definition file (see Section 2.0). This is an advanced topic. Normally, set Part. off.

10.6 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).

11 Deterministic projected belief network (DPBN)

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 16). Therefore, it is necessary to insure that the SPA is correctly estimated. This is greatly helped by SPA prediction (Section 10.4). 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 13.3), or (b) pre-train using stacked RBM (8), or (c) first train the network as a PBN (Section 10).

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. 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.

12 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). 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 2.2) 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 10.2) If LYR is set to (blank), the GMM will train and synthesize visible data directly.

13 Weights Section

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

13.1 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.

13.2 Show: viewing weights

Press “Show” to plot the weights.

13.3 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.

13.4 PCA: initializing weights

Press the “PCA” button to initialize the weights using principle component analysis (PCA).

13.5 Zro: zeroing weights

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

13.6 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 15.

14 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.

14.1 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
```

14.2 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.

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

Every activation function 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:

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

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 13). 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).

16 Saddle-Point Prediction and Estimation for PBN and DPBN

Saddle-Point (SP) estimation 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 SP 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 $\mathbf{x} = [x_1, x_2 \ldots x_N]$, 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, $\boldsymbol{\alpha} = [\alpha_1, \alpha_2 \ldots \alpha_N]$. 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 SP training (Section 10.4). 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 SP training (Section 10.4). 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).

17 Class-dependent PBN Classifier (PBN-C)

The PBN-Tk implements class-dependent PBN classifier. 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 create a classifier experiment, press "EVAL" in the PBN-C Section.

18 Adversarial Attack (AA)

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

19 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 [2] 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	ds81	ds91	ds71
"8"	28x28	ds82	ds92	ds72
"9"	28x28	ds83	ds93	ds73
"3,8,9"	28x28	ds818283	ds919293	ds717273
"3"	14x14	ds21	ds31	ds41
"8"	14x14	ds22	ds32	ds42
"9"	14x14	ds23	ds33	ds43
"3,8,9"	14x14	ds212223	ds313233	ds414243

References

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

[2] 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.